LiFS: An Attribute-Rich File System for Storage Class Memories

Sasha Ames Nikhil Bobb Kevin M. Greenan Owen S. Hofmann
Mark W. Storer Carlos Maltzahn Ethan L. Miller Scott A. Brandt

Storage Systems Research Center
University of California, Santa Cruz

Abstract

As the number and variety of files stored and accessed by a typical user has dramatically increased, existing file system structures have begun to fail as a mechanism for managing all of the information contained in those files. Many applications—email clients, multimedia management applications, and desktop search engines are examples—have been forced to develop their own richer metadata infrastructures. While effective, these solutions are generally non-standard, non-portable, non-sharable across applications, users or platforms, proprietary, and potentially inefficient. In the interest of providing a rich, efficient, shared file system metadata infrastructure, we have developed the Linking File System (LiFS). Taking advantage of non-volatile storage class memories, LiFS supports a wide variety of user and application metadata needs while efficiently supporting traditional file system operations.

1. Introduction

File system interfaces have changed relatively little in the three decades since the UNIX file system was first introduced. Metadata in standard file systems includes directory hierarchies and some fixed per-file attributes including file name, permissions, size, and access/modification times. While primitive, these interfaces have served well.

In the same time frame, demands on the storage subsystem have increased both quantitatively and qualitatively. Storage systems have grown, the amount of storage accessed by individual users has increased, and the variety of data stored has grown dramatically. General-purpose file systems are now used to store tremendous volumes of text documents, web pages, application programs, email files, calendars, contacts, music files, movies, and many other types of data. Although current file systems are relatively effective at reliably storing the data, the increasing size and complexity of the information stored has made management and retrieval of the information problematic. Simply stated, with so much information, it is difficult to find what one really wants. This problem has been addressed on the web with the development of search engines, and it is now often harder to find information on one's own hard drive than on the web.

To address this shortcoming, application developers have been forced to develop their own metadata infrastructures. Email applications, digital photo albums, digital music applications, desktop search applications, and many others have their own file system metadata to enable the organizing, searching, browsing, viewing/playing, annotating, and generally working with specific types of files. Many of these applications have special-purpose code for dealing with the specific properties of the type of data they manage, but they also include code that is not data-specific for organizing, annotating, browsing, etc. Because this code was developed as part of an application, it rarely adheres to any standard, is often not portable, it is difficult to share between applications, users, or platforms, it is typically owned by the company that developed it, and it is potentially inefficient.

Key functions of the operating system are to efficiently provide services that are used by a variety of applications, to abstract away low-level details by providing a useful high-level API, and to facilitate sharing of resources used by multiple users and applications. The wide variety of applications developing their own file system metadata infrastructure shows the need for this infrastructure to be provided by the file system. Recently, researchers (ourselves included) have taken steps in that direction by including additional per-file metadata in the form of $\langle key, value \rangle$ pairs. While useful, this is inadequate to support the needs of the wide variety of applications described above.

We present the Linking File System (LiFS). In addition to the standard file system operations, LiFS provides searchable application-defined file attributes and attributed links between files. File attributes are in the

This paper was published in the *23rd IEEE / 14th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST 2006)*, College Park, MD, May 2006.

form of application-defined $\langle key, value \rangle$ pairs. Links are explicit relationships between files that can themselves have attributes expressed as $\langle key, value \rangle$ pairs to express the nature of the relationship created by the link. These simple additions dramatically change the nature of file systems, enabling a wide variety of operations, providing a rich, shared metadata infrastructure, and allowing applications to focus on managing application-specific data instead of managing things that are best managed by the file system.

In LiFS, all files may contain data and links to other files. Thus, a traditional data file is one with contents and no links, and a traditional directory is one with no contents and directory containment links to other files. Many more interesting examples are possible. For example, a `.c` file can contain links to the `.h` files it includes. An executable can contain links to its source `.c` files, the compiler used to generate it, and the library files on which it depends. A document can contain a link to the application used to edit and view it. With respect to the application-specific infrastructures mentioned above, an email client, a digital photo album, and a music player now all need only provide a GUI for managing their specific type of data and manipulating the attributes of and relationships among the files they each manage, and the file system can take care of storing the attributes and relationships and efficiently supporting their manipulation, search, and other actions.

LiFS is enabled by storage class memories—nonvolatile, byte-addressable RAM—by making the reading, writing, indexing, and searching of such rich metadata fast and efficient. Our prototype is designed with MRAM in mind, but is implemented in Linux using standard DRAM. Our results demonstrate that the performance of LiFS is comparable to, if not better than, that of other Linux file systems, while providing far richer metadata semantics. As a proof-of-concept we have implemented a simple browser that functions alternatively as a file system browser, an email browser, a digital photo browser, or a music browser, depending upon the files and links contained in the file system hierarchy it is exploring. The following sections examine some motivating examples in more detail, discuss the LiFS design, present the details of our LiFS implementation, and show the performance of LiFS in various scenarios.

2. Motivating Examples

Relational links and attributes are surprisingly useful. In this section we will illustrate their utility in finding and organizing information and in coping with change in computing infrastructure. These areas are of particular interest in the context of rapid growth of personal and enterprise-level data and the emergence of utility computing requiring scalable IT management technologies.

2.1. Information Management

2.1.1. Searching Finding data on the vast Worldwide Web is often easier than finding the same content on a local hard drive. The same search engine technology that does well on the Internet typically performs poorly when applied to enterprise-level file systems. To our knowledge both of these statements are only based on anecdotal albeit common evidence but can be plausibly explained: web links convey relevance and semantic information that turns out to be very useful for searching and presenting search results [36]. However, traditional file systems only convey relationships among files through the hierarchical directory system.

Hierarchical directories are actually a compromise between the need of users to organize their data on one hand and file system designers who aim to reduce the cost of maintaining metadata on the other. This dearth of relationships between files is the primary reason that finding data in enterprise-wide file systems or even in local file systems appears to be harder than on the Internet.

There is in fact a wealth of explicit and implicit relationships among files. Because hierarchical directories make it difficult to express explicit relationships, this information often ends up being stored in application-specific files and obscured by proprietary file formats. There are also implicit relationships such as provenance, application and data dependencies, as well as contexts that may span applications; this information is typically not recorded at all. However, maintaining provenance and context relationships enables powerful search capabilities. For example, two downloaded files that are in some way related on the web, *e. g.*, originating from the same web site, normally are stripped from their context when stored on a local file system. Provenance preserves their relationships and provides important clues to context-sensitive searches.

The history of Internet search engines shows a progression of increasingly sophisticated ways of mining relationships, extending successful searching even to documents with content obscured by proprietary formats. The introduction of rich relationships for files will allow the successful use of advanced search technologies within personal or enterprise-wide file systems.

2.1.2. Repository Sharing Because applications are often the sole maintainer of meaningful relationships among files, repositories are often partitioned. As a result, each repository can only be accessed by

one application and relationships that span repositories are hard to represent. A good example is the common fact that notes, email, calendar entries, and instant message conversations each have their own application-specific repositories with no mechanism to represent relationships between, for example, an email and a calendar entry. Some commercial personal information systems such as Microsoft Outlook [31], and other systems currently under development such as Chandler [34] and Haystack [38] try to alleviate this problem by combining traditionally separate repositories into one application-specific repository. However, this approach only alleviates repository partitioning rather than solving the problem.

Another disadvantage of maintaining relationships on the application level is that it makes integration among applications unnecessarily hard: To maintain data relationships across applications, each application has to know how to communicate with another application's API to access and manipulate data in the other application's repository. Some alleviation of this $n : m$ scaling problem is offered by "glue" languages such as Apple's AppleScript [4] and more recent Automator [5], which are designed to enable end users to automate common tasks spanning multiple applications. However, the resulting scripts still interact with data repositories via the API of applications and quickly become obsolete due to API changes from new versions or substitutions of applications.

LiFS, on the other hand, provides a file-centric (as opposed to application-centric) infrastructure that allows applications to not only store files but to insert relationship information directly into the file system's metadata. All applications that take advantage of LiFS relational links and attributes automatically share one repository and integration among applications only needs to interact with the file system's API.

2.1.3. Navigation Rich relationships between files require more sophisticated navigation tools than a traditional file system browser. File system browsers are specialized for traversing hierarchies but are ill-suited for navigating non-hierarchical graph structures with different kinds of links. Web browsers, on the other hand, are designed to navigate hypertext graphs and have developed mechanisms to handle a variety of link types. For example, a web page can contain image source links that are immediately resolved to inline images while anchors are not resolved but displayed as clickable text. Other examples of link types are references to frames, style sheets, or more complicated Javascript constructs that allow asynchronous requests for updates without reloading the web page, such as those used by Google's Gmail, Maps, and Suggest.

Recent web browser designs offer a high degree of extensibility and the ability to render sophisticated user interfaces. We believe that this evolution of web browsers is not a coincidence but was both possible and necessary because of the Web's complex relational linking structure. There is no one good way to display complex structures. Instead, more or less specialized interfaces for interacting with these structures are adopted as these complex structures become commonplace. We are faced with a similar situation when designing file system browsers once we introduce complex linking structures as in LiFS.

The need to render a wide variety of structures has led to powerful architectures, culminating in Mozilla [45], that offer a very flexible and fast layout engine (Gecko [44]) and extensible component architecture that loads the entire specification of a user interface from files. This user interface specification framework is referred to as XML User interface Language (XUL [47]). It is powerful enough to fully specify the user interface of Firefox and Thunderbird, two instances of Mozilla, one for browsing the web and one for managing email. Furthermore, XUL's sister eXtensible Bindings Language (XBL [46]), allows dynamic modification of parts of the user interface.

Significant in the context of LiFS is the fact that XUL presents a fully implemented framework that allows the linking of file structures to specify how to display these structures. It is now conceivable to integrate formerly segregated navigation and management activities regarding notes, email, instant messages, calendar entries, and software development projects into one "file system browser".

Interestingly, the relationship between XUL components (UI content, skin, and scripts) is specified in RDF files [48]. These RDF files can be directly translated into LiFS linking structures. Furthermore, XUL populates the user interface by specifying queries to one or more RDF data sources. The Mozilla component architecture allows the creation of new RDF data sources. We are in the process of implementing a component that provides LiFS linking structures and file attributes as RDF data source.

In summary, by providing relational links we can leverage file system navigation in LiFS with Web browser technologies and use these technologies to combine file system content in ways that were not possible before because of repository partitioning and applications that are hard to integrate.

2.2. Infrastructure Change Management

Infrastructure change can, and often does, destroy the usefulness of data. However, infrastructures change

all the time: software and hardware get upgraded or replaced and a change as small as a single upgrade of an application can render data useless. This process of continual change is the primary reason for data obsolescence and the number one threat to digital preservation [11]. A special case of infrastructure change occurs when data is migrated. In the following section we will illustrate how relational links can make infrastructure change more manageable.

2.2.1. Obsolescence Users often discover data obsolescence when it is too late. A typical example is someone who has to amend the tax return for the year 2001 while filing for 2003 and has changed computing platforms sometime in 2002. While changing platforms, the user copies the file for the 2001 tax return to the new platform not realizing that each tax year and each computing platform requires a separate application. Two years later the user is unable to read the 2001 file because the retroactively purchased application for 2001 cannot access 2001 files generated on the old platform. The user is left with trying to reconstruct the former infrastructure which might be more difficult than recreating the 2001 file. Thus, the 2001 data was essentially lost.

The example illustrates that even well within typical record life cycles it is difficult to anticipate the consequences of infrastructure change *in time* unless the dependencies of data are made explicit. One way to address the rapid rate of change is to introduce another application for managing the context of data by keeping track of which application version created what data. However, such an application would be as exposed to obsolescence as any other application. File systems, on the other hand, have historically enjoyed low obsolescence and are therefore better suited to manage dependencies over longer time periods.

In LiFS we can represent the dependencies in this example with relational links. This allows the generation of a detailed list of consequences *before* deleting an application or changing the infrastructure in other ways that might cause damage.

2.2.2. Migration To continue with the tax return example, recall that the key failure that led to obsolete data was migrating tax data from one platform to another without realizing (for two years) that not all dependencies on the target platform were satisfied. To solve this problem, the source platform has to communicate data dependencies to the target platform and the target platform needs to figure out how to satisfy these dependencies. This functionality is similar to popular open source package managers which package recognize what other packages need to be downloaded for a given package in order for the software to function.

Import and export of LiFS metadata for communication of dependencies between file systems is still ongoing research. We anticipate that metadata will be exported as RDF, and that the import of RDF into metadata will involve a resolution process that either generates requests for missing files or generates alerts for non-satisfiable dependencies.

3. File System Design

3.1. Basic Goals

We have designed LiFS with the goal of providing attributed relationships between files and enhanced metadata with no perceptible performance overhead compared to traditional file systems. We assume that the storage system LiFS runs on will include a high-bandwidth, higher latency component with large amounts of storage (such as a hard disk), and a lower bandwidth, low-latency component with a small amount of storage proportional to the size of the larger storage. This lower bandwidth, low-latency component could be any type of byte addressable, non-volatile memory such as magnetic RAM [12] or any other storage class memory technology. The larger capacity, higher latency storage is used to hold the user data stored in the file system, while traditional and enhanced metadata resides on the smaller, faster storage class memory.

Three specific features of the file system that allow us to reach our design goals are named links between files, attributes on files, and attributes on links. Attributes are composed of $\langle key, value \rangle$ pairs such as $\langle author, john \rangle$. Files are named according to the name of the link by which they are accessed. For instance if there is a link named bar from source file foo, the target file is accessed as bar. Multiple links between two files may exist as long as they are disambiguated by a unique set of attributes. Multiple links are useful when there are multiple users of on a system, and more than one user or application wishes to have a link between two files with their own set of attributes.

In LiFS, the concept of a set of links from a source file replaces that of a directory. For example, if a file work/document.txt links to a file picture.jpeg, a user can change directories to work/document.txt and find picture.jpeg in the listing. Traditional directories are emulated via zero byte files with links to all member files. For backwards compatibility, a legacy application may set a special STAT attribute on a file in order to access that file as a directory, or vice versa.

System call	Function
rellink	Create a new relational link between files
rmlink	Remove a relational link between files
setlinkattr	Set attributes on an existing link between files
openlinkset	Returns an identifier for a set of links from a source file
readlinkset	Fills in standard directory entry structure with link name and attributes for the next link in a set

Table 1: New file system calls

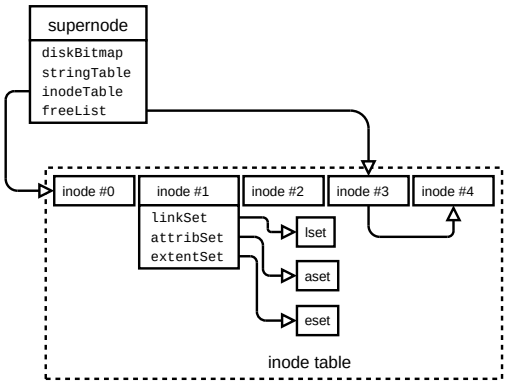


Figure 2: Structure of the LiFS supernode

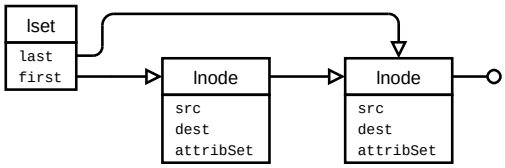


Figure 3: A set of links from a file in LiFS

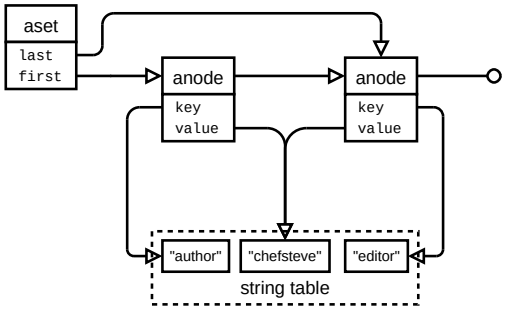


Figure 4: A set of attributes in LiFS

To offer this new functionality, we propose several new system calls, shown in Table 1, to manipulate links and attributes on links. The new system calls have a syntax very similar to that of current calls to manipulate and get information about directories, links, and extended attributes. For file attributes, we implement the standard getxattr and setxattr extended attribute calls.

3.2. In-Memory Data Structures

Traditional file system data structures are optimized for accessing both data and metadata on a high-latency disk. In contrast, we designed the file system data structures in LiFS to take advantage of low latency storage class memories. Traditional inodes have been augmented by link nodes (*lnodes*), attribute nodes (*anodes*) and extent nodes (*enodes*). Because memory access is very cheap, lnodes, anodes and enodes are arranged in linked lists, allowing for trivial insertion and deletion.

Our file system uses a hash table which eliminates duplicate storage of strings. The first time a string is used in LiFS, it is added to the table. When an identical string is later used, a lookup returns a pointer to the string in the string table. A reference count is kept for each string so that unused strings do not remain in memory. The string table is used to optimize string comparisons and searches in the file system. Strings for which there are entries in the string table can be tested for equality with a pointer comparison rather than comparing string data. Additionally, if a string is not in the string table we know that any search on that string in the metadata will not be satisfied.

All data structures necessary for file system usage are referenced from the supernode, shown in Figure 2. The supernode is stored at the beginning of non-volatile memory. The supernode references the string table, a bitmap of allocated blocks, the inode table, and the first free inode. The list of free inodes is embedded within the inode table itself, with each free inode pointing to the next. Each entry in the inode table stores traditional metadata about a file such as mode and size as well as a pointer to three linked lists: an lset, aset, and an eset.

Links in LiFS are maintained in an lnode structure. As shown in Figure 3, an *lset* contains a linked list of lnodes, each of which contains a source inode, destination inode and a set of attributes. The set of attributes is a pointer to an attribute set, or *aset*, shown in Figure 4. An aset contains a linked list of anodes, each of which has a pointer to the string table entry for the key and value of that attribute.

LiFS attempts to allocate disk space in extents—sequential series of blocks. An extent set, or *eset*, shown in Figure 5, contains a linked list of enodes, each of which specifies the limits of a single extent. When LiFS grows a file, it attempts to grow the last extent if possible. If this is not possible, LiFS allocates a new extent and corresponding enode. Free space is found by scan-

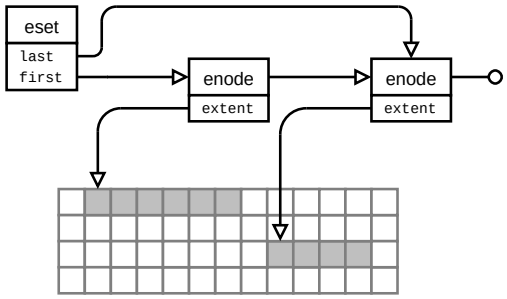


Figure 5: LiFS extent system for disk allocation

ning the bitmap of allocated blocks contained in the supernode using first fit.

4. Implementation

The novel features of LiFS such as relational links require modifications to the Linux Kernel. In Linux, the VFS (Virtual File System) layer provides access to file systems for user mode programs. We added both kernel system calls and VFS functions as required for our new functionality. All new system calls were based on the syntax of similar existing operations.

Our development process was made easier by use of FUSE (File system in User Space). FUSE is a userspace library and Linux kernel module that directs VFS calls to a userspace daemon. Implementing LiFS in userspace through FUSE freed us from the complexities of kernel development, albeit with considerable overhead. We modified FUSE to match our changes to the Linux VFS in order to support the new functionality in LiFS. Once again, the similarity in syntax and function to existing interfaces aided in development. As Figure 6 shows, LiFS resides in a userspace daemon that communicates with the FUSE kernel module through a file in the Linux proc file system. Both our new calls and standard file system calls are passed to the kernel via this channel.

For this paper we were not able to procure sufficient quantities of a non-volatile, byte-writable storage class memory; instead we used system DRAM. The major consequence of this approach is that many operations will run faster than they would in storage class memories. In the future, we will compensate for this speed up with artificial delays. The ability to model slower storage class memories using DRAM will allow us to simulate the performance of LiFS across a wide variety of non-volatile storage.

We assume that a storage class memory would be mapped into an arbitrary segment of the system address space. To imitate this mapping, we allocated several hundred megabytes of system memory exclusively for LiFS data structures. This memory is locked to prevent

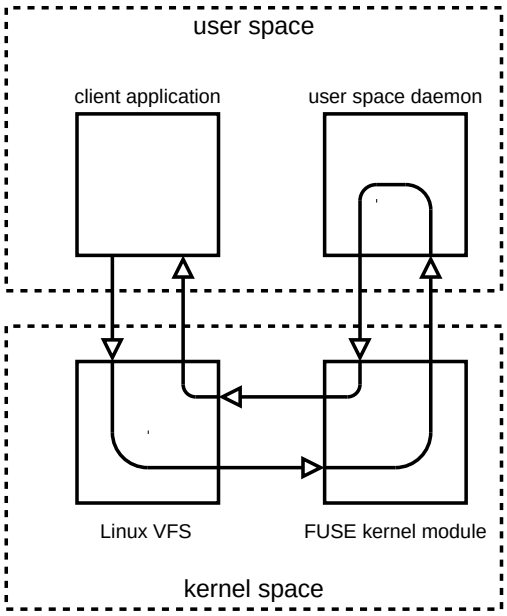


Figure 6: The relationship between FUSE and the FUSE/LiFS userspace daemon

it from being swapped out, based on the assumption that LiFS will have enough storage class memory to hold all the file system's metadata. All memory within this space is allocated using a custom allocator. The allocator allows relocation to arbitrary address spaces, anticipating the possibility that a storage class memory could be mapped into a different part of the system memory address space, perhaps after the system was rebooted or the memory was moved to a new machine. Since our metadata data structures are each a fixed size, we exploit this characteristic by optimizing our custom allocator to preallocate several pools of different constant size chunks. On memory allocation, a chunk of memory is retrieved from a pool of the appropriate size. On deallocation, the chunk is returned to its pool. This allows for very quick turnaround times.

Lookup operations in LiFS require resolution over all links as opposed to solely being resolved over directory structures. Consequently, lookups involve traversing a series of inodes and lnodes beginning at the root inode. Lookup is an iterative process which scans all links originating at its current node. If a link matches the corresponding part of the input pathname, then the target of that link becomes the current node. If no matching link is found, an error is returned.

We have implemented two optimizations to speed up lookups. The first optimization checks that the current path component is in the LiFS string table; if it is not there then it can be safely concluded that the lookup will not be successful. The second optimization is a lookup cache which stores full pathname to inode number map-

File System	Create Files	Read Files
LiFS with FUSE	1.043	0.720
ext2 with FUSE	1.445	1.012
XFS with FUSE	3.998	1.050

Table 7: Create and read time, in seconds, for 15,620 zero-byte files ($k = 5$, $d = 5$, $n = 4$). Times shown are the arithmetic mean of five test runs.

File System	Create Files	Read Files
LiFS with FUSE	12.195	2.027
ext2 with FUSE	2.613	1.679
XFS with FUSE	4.871	1.836

Table 8: Create and read time, in seconds, for 15,620 files ($k = 5$, $d = 5$, $n = 4$) of size 384 bytes. Times shown are the arithmetic mean of five test runs.

pings. Because there is no need for persistence in this cache, it is stored in DRAM as opposed to a storage class memory which stores all other LiFS data structures. The lookup cache is filled on successful lookups. Conversely, before the lookup operation begins, a request is made to the cache for the appropiate pathname to inode mapping.

The core LiFS code has 3,400 lines of C code. Modifications to the FUSE kernel module and userspace libraries required additional code, but these changes were necessitated by FUSE and would not be part of an in-kernel implementation. The small size of our implementation may be attributed to the simplicity of storing all file metadata as in-memory data structures. That the speed of such a simple implementation can compete with more optimized file systems such as ext2 (5,500 lines of code) demonstrates the power of storing metadata in a fast byte writable memory.

The simplicity of our implementation and its speed suggest the potential for significant performance improvements with additional optimization. We already optimize performance and storage requirements using the string table and other data structures which are not feasible in traditional disk-based file systems. We also plan to improve scalability of LiFS by replacing the linked lists prevalent in our implementation with balanced trees.

5. Results and Performance

LiFS was implemented and evaluated on a pair of Sun workstations running the Linux kernel 2.6.9-ac11. Each system was configured with an AMD Opteron 150 processor running at 2400 MHz and one gigabyte of RAM. For testing, LiFS was compared with the XFS and ext2 file system versions included with the Linux kernel. Benchmark testing was automated using Python version 2.3.4, gawk 3.1.3 and GCC 3.4.2. All of the following tests are run on freshly created file systems since the focus of this paper is on the performance of in-memory data structures and not on disk sub-system performance.

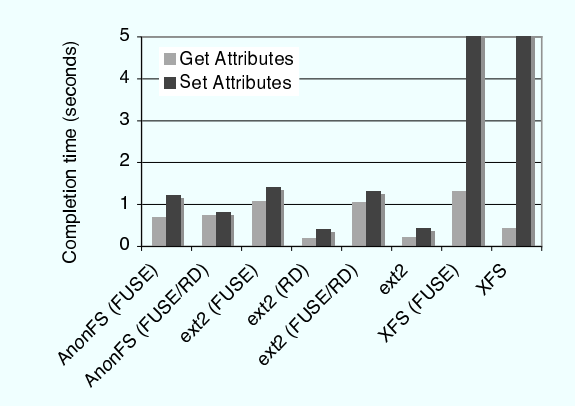
5.1. Standard File System Operations

In order to compare the baseline performance of LiFS, ext2 and XFS, we used a set of six standard file system operations: creating directories, creating files, reading files, setting extended attributes on files, retrieving extended attributes from files and removing directories. Systematic tests of these operations on each file system enabled a fair comparison between LiFS, ext2 and XFS. Each of our tests was run in kernel, through FUSE, on RAM-disk and through FUSE on RAM-disk. LiFS is currently not running in the kernel and XFS cannot be created on a RAM-disk, thus these three scenarios (LiFS in kernel, XFS on a RAM-disk, and XFS via FUSE on a RAM-disk) were omitted. To compare these file systems, we created complete k-ary trees of depth d, with n files per directory each containing a extended attributes.

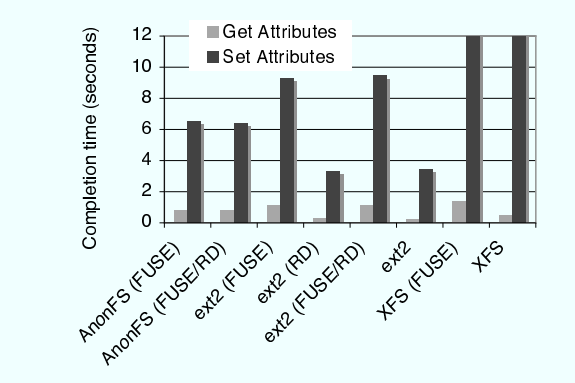
Table 7 shows LiFS performance for creating and reading zero byte files. It is clear that LiFS is competitive with ext2 through FUSE and XFS through FUSE in both tests. Table 8 shows the same tests for files 384 bytes in size. These results show that LiFS is still competitive with ext2 and XFS in the area of file reads but falls behind in file creation. This is due to the extent allocation scheme used in the current implementation of LiFS. For each extent allocation, the current implementation searches the list of free blocks sequentially, starting from the beginning. The results from these two tests indicate that a significant performance gain can be expected from an optimized extent allocation algorithm.

Figures 9(a) and 9(b) show the performance of LiFS, ext2, and XFS creating and retrieving extended attributes on 15,620 files ($k = 5$, $d = 5$, $n = 4$). Figure 9(a) shows the running time for two attributes set on each file, while figure 9(b) shows the running times for twenty attributes per file. In both cases, LiFS performs better than ext2 and XFS through FUSE. When setting twenty attributes per file, LiFS takes roughly 70% of the time required by ext2 under FUSE, while requiring about 73% of the time needed by ext2 to retrieve attributes. These figures also show that the running times for setting and getting file attributes scale well in LiFS as compared to ext2 and XFS.

Directory tree creation performance was tested by



(a) Setting and getting file attributes on 15,620 files ($k = 5$, $d = 5$, $n = 4$, $a = 2$). The times are in seconds and averaged over 5 runs. RD indicates file system on RAM-disk. Note that the XFS runs to set attributes both took over 21 seconds; the graph is cut off at 5 seconds to show details for the other runs.



(b) Setting and getting file attributes on 15,620 files ($k = 5$, $d = 5$, $n = 4$, $a = 20$). The times are in seconds and averaged over 5 runs. RD indicates file system on RAM-disk. Again, XFS, both with and without FUSE, was much slower, requiring over 109 seconds to complete the set attributes benchmark. The graph was cut off to show detail for the other file systems.

Figure 9: Time required to set and retrieve attributes in different file systems.

	Number of Directories		
File System	3,905	19,607	111,110
LiFS with FUSE	0.140	0.7882	4.292
ext2 with FUSE	0.226	1.2315	8.550
XFS with FUSE	0.710	3.3193	25.285

Table 10: Create time, in seconds, for directory trees of various sizes. Times shown are the arithmetic mean of five test runs.

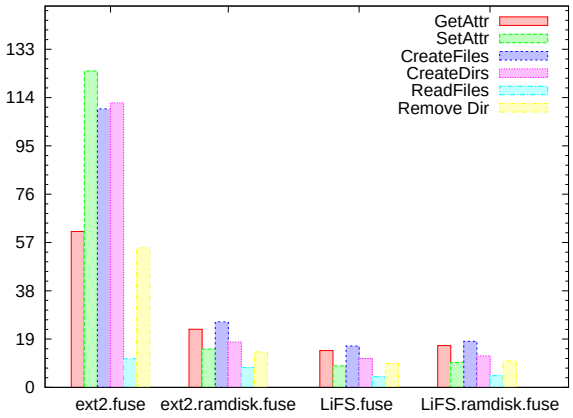


Figure 11: Running times of the 6 file system operations for ext2 and LiFS through FUSE. Results are averaged over 4 runs on a 111,110 directory tree with 1 zero-byte file per directory ($k = 10$, $d = 6$, $n = 1$).

generating directory trees for various values of k and d. Table 10 shows the results of the test, demonstrating that LiFS is comparable to ext2 and XFS. The running time of directory creation scales linearly with the number of directories. This result is comparable to ext2 and XFS. In addition, LiFS is faster than both ext2 through FUSE and XFS through FUSE.

Figure 11 shows the performance of ext2 and LiFS under FUSE and FUSE+RAM-disk. We were unable to create and mount an XFS file system on a RAM-disk, thus XFS is omitted. Since the file system is recreated for each test, the first iteration was working with a cold cache. To avoid extreme shifts in our averages, we discarded the first iteration and averaged over the remaining four. The results displayed in Figure 11 were taken from 4 runs of the 6 file system operations over 111,110 directories with 1 zero-byte file per directory ($k = 10$, $d = 6$, $n = 1$). It is clear from the tests of ext2 that a great deal of overhead is incurred when all of the operations are performed on disk. Because the operations performed in figure 11 use metadata exclusively, LiFS performs better than ext2 on RAM-disk in all cases.

5.2. LiFS Specific Operations

Performance testing of the LiFS specific operations focused on three operations: creating a link between two files, creating attributes for links and removing links. To test the performance of these operations, we created a directory tree and constructed a random graph, where the vertices are files and the edges are attributed links. A directory tree is built in the same manner as in our other tests. The links are then created using a ran-

Operation	Attributes per Link 2	30
Create Links	1.073	1.054
Create Attributes	1.148	2.751
Remove Links	1.086	1.288

Table 12: Time in seconds to create 15,620 random links over a directory tree ($k = 5$, $d = 5$, $n = 4$) with 2 and 30 attributes on each link.

dom number generator and a list of all of the files in the directory tree.

We tested the performance of link and link attribute creation on a directory tree having 15,620 files ($k = 5$, $d = 5$, $n = 4$). After creating the tree, 15,620 links are created between randomly selected files. Table 12 shows the time required to perform a variety of link operations for different numbers of link attributes. The running times shown in Table 12 are averaged over five runs. Link creation is a separate operation from the attribute creation and thus the average running time is not affected by the number of attributes.

The time required to remove links, shown in Table 12, demonstrates the results of a data-structure design choice in the version of LiFS used for testing. When removing a link between two files, a $\langle key, value \rangle$ pair is specified to unambiguously identify a link. This is due to the fact that LiFS allows for multiple links to be made between the same two files. The $\langle key, value \rangle$ pairs are currently stored in a linked list data structure and thus attribute retrieval must, on average, search through half of the link's $\langle key, value \rangle$ pairs. This results in the increased time required to delete a link with 30 attributes versus a link with only 2 attributes.

5.3. SSH Compile Performance

To demonstrate the performance of LiFS in a practical scenario, we compiled OpenSSH version 4.1 using GCC 3.4.2. As Table 13 demonstrates, LiFS running through FUSE is competitive with other file systems in tests both with and without a RAM-disk. Compiling OpenSSH on a LiFS file system running though FUSE took 24.28 seconds. Ext2 through FUSE took 23.40 seconds and XFS through FUSE required 23.49. In a complex series of operations, such as compiling a non-trivial source tree, completion time is not strictly bounded by the file system. Our results show that in such a scenario the extended capabilities of LiFS do not incur an overhead that introduces a new bottleneck to the system.

The OpenSSH compile times show a relatively minor performance hit when running through FUSE. This is to be expected, as compiling a source tree is not bound by

File System	Avg(sec)
ext2	22.52
ext2 through FUSE	23.40
ext2 through FUSE on RAM-disk	23.95
LiFS through FUSE	24.28
LiFS through FUSE on RAM-disk	23.75
XFS	23.20
XFS through FUSE	23.49

Table 13: Compile times for OpenSSH 4.1 (in seconds) on a variety of file systems. Times reflect the arithmetic mean of five trial runs.

Operation	ext2-FUSE	ext2	Speedup
Create Tree	0.2262	0.080	2.828
Create Files	2.613	1.146	2.280
Read Files	1.679	0.474	3.542
Operation	XFS-FUSE	XFS	Speedup
Create Tree	0.709	0.650	1.091
Create Files	4.872	4.282	1.138
Read Files	1.837	0.541	3.396

Table 14: Speedup ratios for selected operations comparing file systems through FUSE versus the same file system without a FUSE layer. Times shown are are the arithmetic mean of five test runs.

disk performance. A more dramatic example of performance gained by eliminating the FUSE layer is shown in Table 14. Reading files without the FUSE layer was over 3.3 times faster for both XFS and ext2. For file and directory tree creation the ext2 file system showed a much more dramatic performance gain compared to XFS. It is therefore difficult to speculate on the exact performance gain that can be expected by eliminating the FUSE layer in LiFS. However, we can assume based on our test results that the gain would be nontrivial.

Another source of potential performance increases for LiFS involves the data structures utilized in the current implementation. The version of LiFS utilized in the testing presented here makes extensive use of linked lists. This includes frequently accessed information such as attribute lists. This will obviously incur a performance penalty when accessing an attribute as, on average, half of the list must be traversed. This behavior is demonstrated in the amount of time required to remove a link as shown in Table 12. There is no design restriction which prevents the use of balanced trees in place of linked lists. Thus a performance increase would be expected by transitioning from linked list data structures to balanced tree data structures.

5.4. Discussion

We have shown that the running times for metadata-based operations in LiFS are faster than ext2 and XFS, while those requiring disk allocation are slower. These results are to be expected because our metadata structures reside in memory and the disk-based data structures are not yet optimized. The LiFS-specific operations regarding link and link-attribute creation have also shown to scale well and run in a reasonable amount of time despite our unoptimized implementation. We have shown that LiFS performs well in a real-world situation, by comparing the running times of a OpenSSH build on LiFS, ext2 and XFS. These results indicate that LiFS can be used in place of most existing file systems without incurring additional overhead. In this section we have also discussed the overhead associated with FUSE. We expect that an implementation of LiFS running in-kernel with optimized disk allocation algorithms be significantly faster than traditional file systems. In addition, the new operations introduced by LiFS should be similar in speed to existing operations.

6. Related Work

The concepts we use in the current and previous designs for LiFS borrow from various research areas ranging from semantic file systems to databases and the Web [2]. We first look at file systems with queryable metadata, such as semantic file systems, and file systems designed to run in nonvolatile memory. We then touch upon upcoming advanced commercial systems. Finally we look at the Semantic Web and archiving, and how they try to convey knowledge more accurately and for the long term.

6.1. Semantic and Other Queryable File Systems

The Semantic File System [21] was originally designed to provide flexible associative access to files. File attributes, expressed as $\langle key, value \rangle$ pairs, are extracted automatically with file type-specific transducers. A major feature of this work is the concept of virtual directories, in which a user makes an attribute-based query and the system creates a set of symbolic links to the files in the result set, providing access that crosses the directory hierarchy. A similar file system, Sedar [29], is a peer-to-peer archival file system with semantic retrieval. Sedar introduces the idea of semantic hashing to facilitate semantic searching and reduce storage and performance costs.

The Inversion File System [33] uses a database to store both file data and metadata. The database also provides transaction protection, fine-grained time travel, and instantaneous crash recovery. Each file is identified by a unique ID, but also has a name and directory associated with it. Moreover, Gupta, *et al.* [22] cite the difficulty of managing different but related sets of files as motivation for their fan-out unification file system, in which fan-out unification refers to merging two directories, and implicitly, entries in a directory are treated as members of a set.

The Logic File System (LISFS) uses a database to support queries for sets of files in the system [35]. Database tables are composed of mappings from keywords to objects. The contents of a directory is the set of objects that meets the criteria of the relation in a query.

Like the above mentioned file systems, the use of attributes in LiFS allows a user to perform expressive queries to locate files. However, these file systems all use secondary storage for metadata, either with or without a database, and must operate under such performance constraints. Additionally, none contain a linking mechanism that supports attributes, thus allowing inter-file relationships that can express the structure of the Web or establish data provenance and history.

6.2. In-Memory File Systems

Douglis, *et al.* compared various storage alternatives for mobile computers and found that flash memory exhibited low power consumption while still providing good read and acceptable write performance [16]. Their focus however was on power consumption and assumed traditional file system functionality.

Conquest [50] utilizes persistent RAM for storage of metadata and small files. Unlike HeRMES and LiFS, which plan to utilize MRAM, Conquest has explored the use of battery-backed DRAM as its form of persistent RAM. The eNVy system and the work of Kawaguchi *et al.* explored the use of flash memory as a primary non-volatile memory storage system [52, 24]. All of these these systems, however, present a traditional file system that lacks the advanced file system features facilitated by utilizing persistent RAM. The HeRMES system [32] proposed that magnetic RAM (MRAM) be used to store file system metadata, and proposed different metadata structures to take advantage of MRAM; however, HeRMES was not implemented, so many of the ideas remained untested.

File system on persistent memory presents a new set of challenges compared to magnetic disk storage. To this end much research has been performed to overcome these challenges. David Lowell's Rio Vista project provides atomicity for memory access operations in the form of transactions [27] while the Journaling Flash File System (JFFS) takes a slightly different approach with a

log-structured file system for flash memory [51]. Edel, *et al.* [17] noted that metadata overhead for a file system is 1–2 percent and suggested the space requirements could be reduced using various compression schemes and algorithms, demonstrating that inode storage space could be reduced by an order of magnitude. They also found that there was no significant difference in performance compared to non-compressed metadata.

Database performance has also been an area that researchers have hoped to improve through persistent memory. Molina and Salem have explored various ways that memory residence optimization could improve data access performance [20].

6.3. Advanced Commercial File Systems

WinFS is Microsoft's in-development file system, for which they have published a preliminary description of planned features[1]. WinFS appears to be a marriage of a database for metadata and NTFS for file stream performance. WinFS treats the contents of the file system as *Items* that cover a full range of granularity from simple descriptions to collections such as folders. The database backing allows SQL-type queries, XPATH searches [14], and the use of Microsoft's OPATH, a query language designed for a directed acyclic graphs [39].

Apple Computer's Spotlight is a metadata and content indexing system integrated into the HFS+ file system [6]. As with WinFS, metadata is stored in a database; Spotlight indexes file content and includes the results in the database as well. Apple's approach could benefit from a LiFS-like linking mechanism with metadata allowing relationships between content to be expressed; Spotlight currently only allows indexing on files, not the links between them. The Linux counterpart of Spotlight is Beagle [1] which provides an API and a plugin infrastructure for new file types and takes advantage of Inotify [18], a file system event service recently merged into the Linux kernel.

Sun Microsystem's ZFS allows administrators to configure individual file systems for users or application, all allocated from a single pool of storage [3]. Like ZFS, LiFS allows for the definition of unique and dynamic file systems per user or per application by virtue of links with attributes. Thus, it is possible to create file systems on demand utilizing either system. However, ZFS does not contain the rich metadata constructs present in LiFS.

[1]This comes with the caveat that all is subject to change. Further information is available at http://longhorn.msdn.microsoft.com/portal_nav.htm

6.4. The Semantic Web

The original World Wide Web has expanded upon the ability of traditional documents to convey knowledge by adding links. Within the World Wide Web and hypertext documents in general, links allow readers to automatically traverse from one document to another when the document refers to the other. The Semantic Web expands upon this by allowing the links themselves to contain information about that particular relationship from one document to another. On top of that basic framework, one may devise ontologies to further convey knowledge in ways not possible through previous means. [10].

To make the Semantic Web possible, authors at the W3C have been developing various standards for its implementations in a way analogous to the standardized HTML and HTTP for the World Wide Web. The group of Semantic Web standards fall into layers, with URI and Unicode on the bottom, XML, name spaces, and schemas making up the self-descriptive document layer in the middle, and the RDF layer on top, which provides a common framework for metadata across applications. Atop the three bottom layers are additional layers for ontology vocabularies, logic, proof, and trust [9, 25]. The ontology layer has room for different attempts to devise languages in which to describe ontologies, such as OIL [19].

We envision three potential expectations for the Semantic Web. For humans, it may be a readily accessible universal library. Moreover, and in line with the ideas of its inventors, the Semantic Web has increased potential for machine processing of its contents, and this introduces the other two perspectives: the knowledge navigator and the federated knowledge or database [30].

Whereas the Semantic Web allows for the addition of richer metadata for the World Wide Web, it does so on a global scale. LiFS allows for the same depth of knowledge representation through its links and attributes. It accomplishes this within the scale of the local file system, which to many users, contains the data on which the semantics of relationships matter most. Additionally, the Semantic Web RDF format can be basically broken into tuples of a subject, property, and object. Links within LiFS likewise contain a source, attributes and a target. Thus, we can express the same relationships locally that are possible given the richness of the Semantic Web. Based on their similarities, LiFS could make an excellent file system or storage layer for Semantic Web data.

6.5. Digital Preservation

Digital objects do not survive unless one makes a conscious effort to preserve them. This is in contrast to artifacts such as books, papyrus, and cuneiform tablets which exist until someone or something actively destroys them. One reason for the ephemeral quality of digital media is unreliable physical media: the shelf life of magnetic tape, hard drives, and CD-Rs can be less than a decade [13, 37, 7]. However, the loss of data due to unreliable media pales in comparison to the loss of data due to the rapid obsolescence caused by technological change. Moreover, digital objects are often related to other digital objects that might change names or disappear entirely [11]. There are now large national and international efforts to address these issues; these efforts aim to provide standards for an exhaustive list of aspects for digital preservation in museums and libraries [49, 26, 43].

Trusted digital repositories [40] adhering to the now dominant international standard of the OAIS Reference Model [15] are an important component of digital preservation. Many systems have been explored which utilize this model [42, 23, 41]. Common challenges of these digital repositories are scalability, interoperability with other repositories, and efficient workflow support for the entry of large numbers of digital objects.

All these digital repositories are designed for use on an institutional level. However, the combination of unreliable storage and obsolescence unintentionally destroys much of digital media long before it can be considered for digital libraries. Personal correspondence and images that survived from earlier times form a significant part of our cultural heritage [28]. Today, personal correspondence in the form of email and chats as well as personal photos and movies are largely kept on home computers that neither meet standards nor follow practices of national digital libraries and are therefore unlikely to survive.

The design of LiFS provides the infrastructure to make digital preservation an integral part of file systems. Links and attributes can be used to explicitly represent the dependencies of digital objects on the software infrastructure thereby preventing accidental obsolescence or at least alert users to obsolescence events introduced by a particular change in the software infrastructure. Our hope is that this will make it easier for users to maintain good digital preservation practices.

7. Future Work

Moving forward, we hope to improve LiFS' performance, storage overhead and availability. We plan to replace many of the linked lists in our data structures with faster balanced trees, as well as implement LiFS in the Linux kernel. It has been shown that inode storage space could be reduced by an order of magnitude without sacrificing performance by compressing inodes [17]. We are planning to add such capabilities to LiFS, and also to investigate how file compression can be achieved efficiently. We are also looking at how we can use fault tolerant data structures[8] to increase the reliability and availability of LiFS. Implementing an online file system consistency checker is another area we are investigating in which availability can be improved. Finally, we are investigating ways of providing LiFS rich metadata structure and performanc without the use of storage-class memories.

8. Conclusions

LiFS makes two important contributions. First, it supports far richer file system metadata via file attributes and attributed links between files. Second, it provides a common, high-performance metadata store for applications, further facilitating interactions between disparate applications through the file system.

LiFS' three major advantages over other file systems are its performance, simplicity, and expressiveness. LiFS clearly demonstrates the performance advantages of storing metadata in a storage class memory. Our tests showed that LiFS is faster than ext2 and XFS in it's metadata performance, even when they are running on a RAM-disk. However, our on-disk data storage is currently not comparable to these file system. We are planning to implement a more advanced extent based approach, such as that used by XFS, which we believe will produce significant performance improvements. As previously mentioned, implementing LiFS in the kernel without FUSE should also provide a further performance boost. LiFS also offers a simplicity not approachable by other file systems via our use of simple data structures. This simplicity is evident in the relatively small code base of LiFS compared to other file systems. We believe that this results in fewer bugs as a function of LiFS having fewer lines of code.

The most important reason for using LiFS, however, is its ability to provide a rich environment for expressing inter-file relationships. Just as flat files and UNIX pipes enabled the creation of small, targeted applications such as awk, eqn, and pic that could work together to perform complex tasks, we believe that the addition of highly flexible attributed links to the file system will enable many new domains for applications to cooperate. Individual application developers can take advantage of a common substrate that supports extensive use of linking and attributes to all applications,

allowing them to develop and enhance applications to provide and manage an increasingly interlinked single repository of data.

Acknowledgments

We would like to thank the faculty and students in the Storage Systems Research Center for their help and comments. This research was funded in part by National Science Foundation grant 0306650. Additional support for the Storage Systems Research Center was provided by Hewlett Packard Laboratories, Hitachi Global Storage Technologies, IBM Research, Intel, Microsoft Research, Network Appliance, Rocksoft, Symantec, and Yahoo.

References

[1] Main page - beagle. http://beaglewiki.org/Main_Page.

[2] A. Ames, N. Bobb, S. A. Brandt, A. Hiatt, C. Maltzahn, E. L. Miller, A. Neeman, and D. Tuteja. Richer file system metadata using links and attributes. In *Proceedings of the 22nd IEEE / 13th NASA Goddard Conference on Mass Storage Systems and Technologies*, Monterey, CA, Apr. 2005.

[3] Anonymous. In a class by itself - the Solaris 10 operating system. Technical report, Sun Microsystems, Nov. 2004.

[4] Apple Computer, Inc. AppleScript scripting language for Mac OS X. http://www.apple.com/macosx/features/applescript/, 2005.

[5] Apple Computer, Inc. Automator automation tool for Mac OS X. http://www.apple.com/macosx/features/automator/, 2005.

[6] Apple Developer Connection. Working with Spotlight. http://developer.apple.com/macosx/tiger/spotlight.html, 2004.

[7] Associated Press. CDs, DVDs not so immortal. In *CNN.com*, May 6 2004. last viewed on Jan 9, 2005 at http://www.longnow.org/10klibrary/darkarticles/ArtCDROT.htm.

[8] Y. Aumann and M. A. Bender. Fault tolerant data structures. In *FOCS '96: Proceedings of the 37th Annual Symposium on Foundations of Computer Science*, page 580, Washington, DC, USA, 1996. IEEE Computer Society.

[9] T. Berners-Lee. Semantic web roadmap, Sept. 1998. Available at http://www.w3.org/DesignIssues/Semantic.html.

[10] T. Berners-Lee and E. Miller. The semantic web lifts off. *ERCIM News*, 51, Oct. 2002.

[11] H. Besser. Digital longevity. In M. Sitts, editor, *Handbook for Digital Projects: A Management Tool for Preservation and Access*, chapter 9, pages 164–176. Andover: Northeast Document Conservation Center, 2000.

[12] H. Boeve, C. Bruynseraede, J. Das, K. Dessein, G. Borghs, J. De Boeck, R. C. Sousa, L. V. Melo, and P. P. Freitas. Technology assessment for the implementation of magnetoresistive elements with semiconductor components in magnetic random access memory (MRAM) architectures. *IEEE Transactions on Magnetics*, 35(5):2820–2825, Sept. 1999.

[13] F. R. Byers. Care and handling of CDs and DVDs: A guide for librarians and archivists. Report 121, Council on Library and Information Resources and National Institute of Standards and Technology, October 2003. Last viewed on Jan 5, 2005 at http://www.clir.org/pubs/reports/pub121/contents.html.

[14] J. Clark and S. DeRose. XML path language (xpath), 1999.

[15] Consultative Committee for Space Data Systems. Reference model for an open archival information system (oais). Standards Recommendation 650.0-B-1 (Bluebook, Issue 1), CCSDS, January 2002. This Recommendation has been adopted as ISO 14721:2003.

[16] F. Douglis, R. Cáceres, F. Kaashoek, K. Li, B. Marsh, and J. A. Tauber. Storage alternatives for mobile computers. In *Proceedings of the 1st Symposium on Operating Systems Design and Implementation (OSDI)*, pages 25–37, Monterey, CA, Nov. 1994.

[17] N. K. Edel, D. Tuteja, E. L. Mlller, and S. A. Brandt. MRAMFS: a compressing file system for non-volatile RAM. In *Proceedings of the 12th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS '04)*, pages 596–603, Oct. 2004.

[18] I. Edoceo. inotify for linux. http://www.edoceo.com/creo/inotify/.

[19] D. Fensel, F. van Harmelen, I. Horrocks, D. McGuinness, and P. Patel-Schneider. Oil: An ontology infrastructure for the semantic web. *IEEE Intelligent Systems*, 16(2):38–44, 2001.

[20] H. Garcia-Molina and K. Salem. Main memory database systems: An overview. *IEEE Transactions on Knowledge and Data Engineering*, 4(6):509–516, Dec. 1992.

[21] D. K. Gifford, P. Jouvelot, M. A. Sheldon, and J. W. O'Toole, Jr. Semantic file systems. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP '91)*, pages 16–25. ACM, Oct. 1991.

[22] P. Gupta, H. Krishnan, C. P. Wright, M. Zubair, J. Dave, and E. Zadok. Versatility and Unix Semantics in a Fan-Out Unification File System. Technical Report FSL-04-01, Computer Science Department, Stony Brook University, January 2004.

[23] K. J. Jon, D. Bainbridge, and I. H. Witten. The design of greenstone 3: An agent based dynamic digital library. Technical report, Department of Computer Science, University of Waikato, Hamilton New Zealand, December 2002. last viewed on Jan 9, 2005 at http://www.sadl.uleth.ca/greenstone3/gs3design.pdf.

[24] A. Kawaguchi, S. Nishioka, and H. Motoda. A flash-memory based file system. In *Proceedings of the Winter 1995 USENIX Technical Conference*, pages 155–164, New Orleans, LA, Jan. 1995. USENIX.

[25] M.-R. Koivunen and E. Miller. W3C semantic web activity, Nov. 2001.

[26] Library of Congress. Library of Congress announces awards of $15 million to begin building a network of partners for digital preservation. http://www.digitalpreservation.gov/about/pr_093004.html, September, 30 2004.

[27] D. E. Lowell and P. M. Chen. Free transactions with Rio Vista. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, pages 92–101, Dec. 1997.

[28] C. Lynch. The battle to define the future of the book in the digital world. *First Monday*, 6(6), June 2001. available at http://firstmonday.org/issues/issue6_6/lynch/index.html.

[29] M. Mahalingam, C. Tang, and Z. Xu. Towards a semantic, deep archival file system. Technical Report HPL-2002-199, HP Laboratories, Palo Alto, July 2002.

[30] C. C. Marshall and F. M. Shipman. Which semantic web? In *HYPERTEXT '03: Proceedings of the fourteenth ACM conference on Hypertext and hypermedia*, pages 57–66. ACM Press, 2003.

[31] Microsoft Corporation. Outlook software for Microsoft Windows. http://www.microsoft.com/office/outlook/prodinfo/default.mspx, 2003.

[32] E. L. Miller, S. A. Brandt, and D. D. E. Long. HeRMES: High-performance reliable MRAM-enabled storage. In *Proceedings of the 8th IEEE Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, pages 83–87, Schloss Elmau, Germany, May 2001.

[33] M. A. Olson. The design and implementation of the Inversion file system. In *Proceedings of the Winter 1993 USENIX Technical Conference*, pages 205–217, San Diego, California, USA, Jan. 1993.

[34] Open Source Application Foundation. What's compelling about Chandler: A current perspective. http://www.osafoundation.org/Chandler_Compelling_Vision.htm.

[35] Y. Padioleau and O. Ridoux. A logic file system. In *Proceedings of the 2003 USENIX Annual Technical Conference*, pages 99–112, San Antonio, TX, June 2003.

[36] L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford, Nov. 1998.

[37] M. Pollitt. Ever decreasing circles. In *The Independent*, 21 April 2004. Last viewed on Jan 5, 2005 at http://www.fbia.org/print.asp?ID=27963.

[38] D. Quan, D. Huynh, and D. R. Karger. Haystack: A platform for authoring end user semantic web applications. In *2nd International Semantic Web Conference (ISWC2003)*, Sanibel Island, FL, USA, Oct 20-23 2003. Last viewed on Jan 10, 2005 at http://haystack.lcs.mit.edu/papers/iswc2003-haystack.pdf.

[39] T. Rizzo and S. Grimaldi. Data access and storage developer center: An introduction to "WinFS" OPath, 2004. http://msdn.microsoft.com/data/default.aspx?pull=/library/en-us/dnwinfs/html/winfs10182004.asp.

[40] RLG. Trusted digital repositories: Attributes and responsibilities. Report, RLG-OCLC, Mountain View, CA, May 2002. Last viewed on Jan 9, 2005 at http://www.rlg.org/en/page.php?Page_ID=583.

[41] T. Staples, R. Wayland, and S. Payette. The Fedora Project: An open-source digital object repository management system. *D-Lib Magazine*, 9(4), April 2003. Last viewed on Jan 9, 2005 at http://www.dlib.org/dlib/april03/staples/04staples.html.

[42] R. Tansley, M. Bass, M. Branschofsky, G. Carpenter, G. McClellan, and D. Stuve. DSpace system documentation. Documentation, MIT Libraries, May 04 2005.

[43] The Joint Information Systems Committee. Supporting digital preservation and asset management in institutions. http://www.jisc.ac.uk/index.cfm?name=programme_404, October 2004.

[44] The Mozilla Organization. Gecko layout engine. http://www.mozilla.org/newlayout/, 2005.

[45] The Mozilla Organization. Mozilla internet application suite. http://www.mozilla.org/products/mozilla1.x/, 2005.

[46] The Mozilla Organization. XBL - extensible binding language 1.0. http://www.mozilla.org/projects/xbl/xbl.html, 2005.

[47] The Mozilla Organization. XML user interface language (XUL). http://www.mozilla.org/projects/xul/, 2005.

[48] The World Wide Web Consortium. Resource description framework RDF. http://www.w3.org/RDF/, 2005.

[49] U.S. National Archives & Records Administration. National Archives names two companies to design an electronic archives. http://www.archives.gov/media_desk/press_releases/nr04-74.html, August 3 2004.

[50] A.-I. A. Wang, G. H. Kuenning, P. Reiher, and G. J. Popek. Conquest: Better performance through a disk/persistent-RAM hybrid file system. In *Proceedings of the 2002 USENIX Annual Technical Conference*, Monterey, CA, June 2002.

[51] D. Woodhouse. The journalling flash file system. In *Ottawa Linux Symposium*, Ottawa, ON, Canada, July 2001.

[52] 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)*, pages 86–97. ACM, Oct. 1994.

Energy-Reliability Tradeoffs in Sensor Network Storage

Neerja Bhatnagar Kevin M. Greenan Rosie Wacha Ethan L. Miller Darrell D. E. Long
Storage Systems Research Center, University of California, Santa Cruz, CA 95064, USA

Abstract

Sensor nodes that store their data locally are increasingly being deployed in hostile and remote environments such as active volcanoes and battlefields. Observations gathered in these environments are often irreplaceable, and must be protected from loss due to node failures. Nodes may fail individually due to power depletion or hardware/software problems, or they may suffer correlated failures from localized destructive events such as fire or rockfall. While many file systems can guard against these events, they do not consider energy usage in their approach to redundancy. We examine tradeoffs between energy and reliability in three contexts: choice of redundancy technique, choice of redundancy nodes, and frequency of verifying correctness of remotely-stored data. By matching the choice of reliability techniques to the failure characteristics of sensor networks in hostile and inaccessible environments, we can build systems that use less energy while providing higher system reliability.

Categories and Subject Descriptors

D.4.5 [**Reliability**]: Backup Procedures, Fault-tolerance—*Distributed File Systems*; C.4 [**Performance of Systems**]: Fault tolerance

General Terms

energy-reliability tradeoffs

Keywords

energy, reliability, sensor network storage

1 Introduction

The availability of inexpensive gigabyte-scale local storage on sensor nodes [13] and the high cost of radio operations relative to storage operations are enabling sensor nodes that store data locally in between data collection events [12]. Storage-based sensor networks are used to monitor volcanoes, battlefields, habitats, seismic events, traffic, and the stability and integrity of engineered structures such as buildings and bridges [2, 20]. However, the difficulty of gathering data from sensor nodes in hostile and inaccessible environments has also made it harder to deploy base stations that accumulate nodes' data. Base stations installed with sensor networks are easily detected in contested land areas such as borders, and are an obvious target for network disruption. Base stations in inaccessible and natural environments are single points of failure because they may suffer from power outages or malfunction, causing data loss; in a volcano-based sensor network, "[f]ailures of the base station infrastructure were a significant source of network downtime" [20]. Some networks try to avoid this problem by deploying multiple base stations or specialized storage nodes [19], increasing the both the likelihood of the detection of the network, and the system cost. Data loss in centrally-controlled sensor networks is likely to be more severe because nodes do not retain the observations they have already uploaded to the base station. Moreover, a base station cannot easily transmit data to a receiver when none is nearby, as is often the case in remote environments. Such environments are better suited to occasional data collection, requiring nodes to reliably maintain their data over long periods of time.

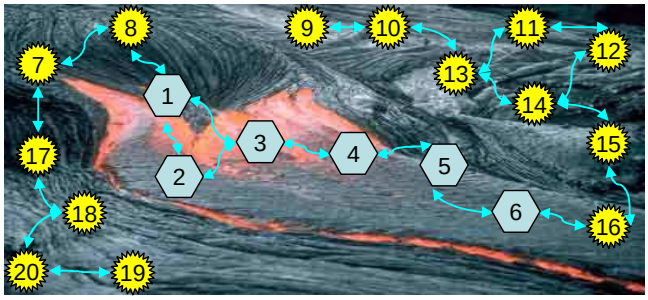


Figure 1. Sensor network on a volcanic lava flow. Nodes 1–6 have been destroyed by the flow.

Individual sensor nodes typically suffer from relatively high failure rates, as compared to traditional storage devices. Moreover, sensor nodes are more likely to suffer *correlated* failures due to environmental dangers. Individual failures may be caused by battery depletion, hardware or software errors, or physical damage. In contrast, correlated node failures may be caused by larger-scale physical damage caused by a destructive event such as flood, rockfall, or fire; for example, the lava flow in Figure 1 has obliterated nodes 1–6. Unfortunately, the latest data from destroyed nodes is the most valuable because it may record details of the event, making it even more important for the observations gathered by the nodes to survive their destruction. However, it is also imperative that sensor nodes create and maintain back-up copies of their data without overwhelming their energy budgets.

We discuss the tradeoffs between energy and reliability in sensor networks that store data for long periods of time: weeks to years. These tradeoffs can be made in three separate areas: redundancy techniques, choice of nodes which store the redundant data, and fre-

Hot EmNets'08 June 2–3, 2008, Charlotesville, Virginia, USA.

quency of integrity checks on the remotely stored redundant data. We do not expect the energy expenditure of reliable storage in sensor networks to be less than the energy expended by nodes to upload their data to a base station; rather, our goal is to make sensor network storage much more reliable by increasing the likelihood that sensor data survive despite individual and correlated node failures. By providing energy-efficient storage operations, sensor networks can more easily provide raw data, instead of aggregated and representative values, to their intended audience, potentially facilitating more robust forecast and analysis models.

We assume that the network is comprised of sensor nodes severely constrained in power, storage, and processing. We also assume that nodes have limited radio range, so communication with distant nodes requires multi-hop routing. Since our research is primarily concerned with energy-reliability tradeoffs, we fold the costs for interference and retransmission into the cost for transmitting data between nodes. We assume that each node has a battery-backed RAM for buffering data and NAND flash memory for persistent storage [12], though new non-volatile memory technologies such as phase change memories may further simplify the architecture [9].

2 Issues in Reliability

Analyzing tradeoffs between energy usage and file system reliability depends on making good choices for redundancy techniques, nodes for remote storage, and frequency of checking integrity of redundant data, while considering the high failure rate of sensor nodes and the likelihood of occurrence of correlated failures [14].

2.1 Redundancy Techniques

As with traditional file systems, sensor nodes may use either mirroring or erasure coding to store data reliably. Transmission costs dominate energy usage when mirroring is used because transmitting data costs two hundred times more energy [12] than storing the same amount of data locally. As a result, due to the relative position of nodes and the base station, the transmission cost of mirroring data to another node may be lower than that of uploading data to a base station. This is specifically the case when the transmitting node is in the center of the network and the base station is installed at the edge of the network, or vice versa. For example, in Figure 1, node 6 will have to transmit its data over five hops if the base station was installed near node 11. The storage overhead of mirroring is also very high: tolerating n failures requires the system to store $n+1$ copies of the data. In contrast, processing costs dominate energy usage for erasure codes.

We compared the performance (energy consumption expressed in mJ and throughput expressed in MB/s) of encoding using Reed-Solomon (RS) codes [15] based on GF(2^8) [5] to XOR-based codes [6, 21] on an ARM9E 400 MHz processor that consumes 94 mJ/s [1]. The first column in Table 1 represents the RS code implementation for parameters (n,m), where n is the number of data nodes, and m is the number of parity nodes. RS codes were implemented as table lookups, where each multiplication requires two lookups. Each lookup table is 256 bytes in size, consuming 512 bytes of memory. The second column in Table 1 represents the most fault-tolerant XOR-based codes for the same parameters. These codes have the storage efficiency of $n/(n+m)$. The last two rows present the performance of highly fault-tolerate XOR-based codes that we developed. The XOR_1 code we designed is an instance of a WEAVER code [6] that tolerates two-node failures.

Reed-Solomon codes consume 3–10 times more energy than XOR-based codes due to more complex finite field calculations [6], but provide higher reliability (*e.g.*, a $(5,3)$ RS code can tolerate *all* three-node failures but an XOR-based $(5,3)$ code may only be able to tolerate at *most* three-node failures). However, it may be possible to tolerate some node failures without losing data because very

Table 1. Energy Expenditure of Erasure Codes in mJ/s and Throughput in MB/s.

Code Size	Energy Expenditure (mJ)		Throughput (MB/s)	
	RS	**XOR**	**RS**	**XOR**
(5, 3)	3.515	1.205	2.674	7.798
(6, 2)	3.133	0.6	3	15.654
(9, 3)	4.82	0.524	1.95	17.953
(10, 2)	3.92	0.653	2.4	14.4
(17, 3)	5.193	0.588	1.81	15.99
(18, 2)	4.36	0.589	2.156	15.972
XOR_1	—	0.74	—	12.76
XOR_1	—	0.75	—	12.72

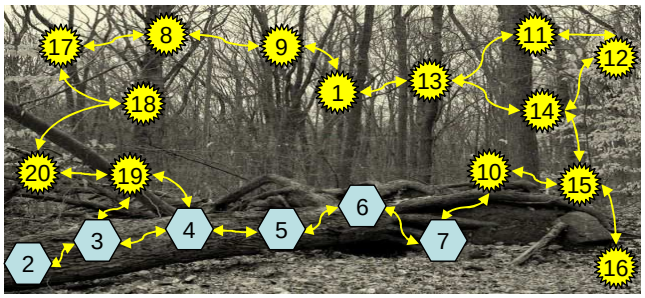


Figure 2. Node 2 replicates its data on nodes 3–7 but is more likely to suffer data loss even from a small destructive event.

closely-located sensor nodes may be observing similar phenomena. In order to tolerate correlated failures, closely-located sensor nodes must spread their information over a large physical area. The energy expenditure of XOR_1 and XOR_2 schemes is comparable to most XOR-based codes but better than that for RS codes. We are currently exploring the suitability of several less processor intensive XOR-based codes, based on the research done by Wylie and Swaminathan [21], to sensor networks.

2.2 Node Choice

The impact of correlated failures caused by localized damage can be mitigated by spreading redundant data over a large physical area. There is a cost in energy to send the data further away. For example, observations from node 2 in Figure 2 will be lost, despite replicating them on nodes 3–7, if a tree rooted near node 2 falls. Data from node 2 is more likely to survive if node 2 sends its data to nodes $\langle 3,8,12,16,20\rangle$ for redundant storage, as shown in Figure 3. Nodes $\langle 8,12,16,20\rangle$ are well-spread out; and so are less likely to fail simultaneously. Even though the number of nodes in node 2's redundancy group is the same in both examples, the latter scheme is more reliable but also more expensive both for node 2 and its neighboring nodes because multi-hop transmissions consume more energy.

Mirroring alone is energy-consuming for making sensor network storage reliable. In order to reduce energy expenditure, it may be better to mirror data only to nearby nodes and to use erasure codes for nodes that are further away. This approach can quickly replicate data nearby, guarding against individual node failure, and can use widespread replication to protect against correlated node failures. For example, in Figure 3, node 2 can mirror data to node 3 to safeguard against its own failure, but use erasure codes with nodes $\langle 8,12,16,20\rangle$ to safeguard against correlated failures. Systems such as OceanStore [16] use erasure codes to tolerate relatively large numbers of failed nodes; we plan to do the same for making sensor network storage reliable. Our file system has the

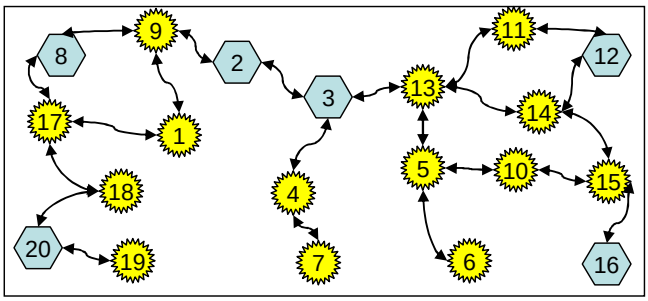


Figure 3. Node 2 replicates its data on nodes $\langle 3,8,12,16,20\rangle$ to increase its likelihood of surviving large destructive events.

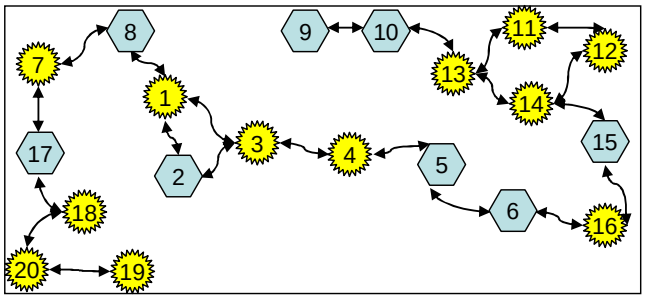


Figure 4. Nodes $\langle 2,5,6,8,9,10,15,17\rangle$ form an 8-node redundancy group such that the network can tolerate the failure of 3-node combinations such as $\langle 5,10,17\rangle$ and $\langle 2,6,10\rangle$.

advantage of using less-expensive XOR-based codes in place of RS codes by carefully placing redundant data on particular nodes. When using a $(5,3)$ XOR-based code, by arranging data so that the "fatal" three-node sets cover a large physical area, the sensor network can gain nearly all benefits of RS codes with the computational cost of XOR-based codes. For example, node 2 in Figure 4 might choose nodes $\langle 2,5,6,8,9,10,15,17\rangle$ in its eight-node redundancy group. If only three-node combinations, such as $\langle 5,10,17\rangle$ and $\langle 2,6,9\rangle$, caused data loss, then the system would be relatively safe since these node-sets cover a widespread area, and therefore, are less likely to suffer correlated failures simultaneously. The system could provide additional reliability by choosing some very distant nodes as part of its redundancy group, perhaps replacing node 5 with node 19 and node 6 with node 12.

We use a simple Markov model to analyze the availability of the Mirror$_4$, XOR$_1$, and XOR$_2$ schemes. Figure 5, depicts 4-way mirroring, but can easily be generalized to an n-node redundancy group. The transitions are exponentially distributed with mean failure rate λ, and mean repair rate μ. For simplicity, we let $\rho = \lambda/\mu$. State $(0,0)$ represents the failed state.

In the XOR$_1$ scheme, each node stores its own data and the XOR of data from two other nodes. For example, node A stores its own data and $B \oplus C$; node B stores its own data and $C \oplus D$; node C stores

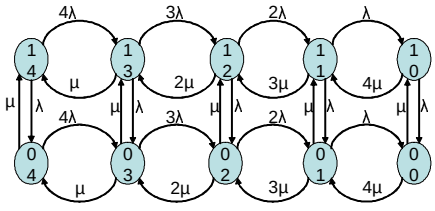


Figure 5. Markov model of a 5-node redundancy group depicting Mirror$_4$.

Table 2. MTTDL, in hours, for Mirror$_4$, XOR$_1$ and XOR$_2$ schemes with and without repair.

	Mirror$_4$	XOR$_1$	XOR$_2$
MTTDL with repair	4.87×10^{11}	2.42×10^{6}	6.50×10^{8}
MTTDL w/o repair	4932	1692	2772

its own data and $D \oplus E$; node D stores its own data and $A \oplus E$; and node E stores its own data and $A \oplus B$. In the XOR$_2$ scheme, each node stores its own data and data from four other nodes as two-node XORs. For example, node A stores its own data and $B \oplus C$ and $D \oplus E$; node B stores its own data and $D \oplus E$ and $A \oplus C$; node C stores its own data and $A \oplus E$ and $B \oplus D$; node D stores its own data and $A \oplus C$ and $B \oplus E$; and node E stores its own data and $A \oplus B$ and $C \oplus D$. The storage overhead of Mirror$_4$ is four times that of the original data set. The storage overhead of XOR$_1$ and XOR$_2$ schemes is, respectively, two and three times the original data set. Figure 6 shows that XOR$_2$ delivers availability similar to Mirror$_4$, but at a lower overhead. Mirror$_4$ can tolerate at most four node failures, while XOR$_1$ and XOR$_2$ schemes can, respectively, tolerate at most two- and three-node failures. Markov models provide good approximate analysis, but do not work well for "irregular" XOR codes or for systems that experience correlated failures; these are better suited to simulation.

The availability of a node's data when Mirror$_4$, XOR$_1$, and XOR$_2$ schemes are used to create redundancy in the sensor network are given by:

$$\text{Mirror}_4 = 1 - \frac{\rho^5}{(1+\rho)^5},$$

$$\text{XOR}_1 = \frac{10\rho^2+5\rho+1}{(\rho+1)^5}, \text{ and}$$

$$\text{XOR}_2 = 1 - \frac{5\rho+1}{(\rho+1)^5}.$$

These availability models are simple and assume that the nodes may be repaired. In the case of no repair, steady-state does not exist and so the system must be modeled using differential equations. These equations quickly become unmanagable, and so a better solution is to use simulation, which has the additional advantage of being able to model correlated failures.

Modeling mean-time-to-data-loss (MTTDL) is easier, and uses the same transition matrix that would be used for modeling with differential equations. We assume that both failures and repairs are exponentially distributed. We solve all these models by building a transition matrix M, as discussed by Schwarz [17], and computing

$$MTTDL = -[1,1,1,\ldots,1]\cdot M^{-1}\cdot[1,0,0,\ldots,0].$$

Table 2 presents the MTTDL for Mirror$_4$, XOR$_1$, and XOR$_2$ schemes, with and without repairs. For this example we assume that nodes are organized into five-node redundancy groups and choose $\rho = 5.56\times10^{-3}$, which assumes that failures occur on average every 3 months and nodes are repaired, on average, in 12 hours.

2.3 Frequency of Integrity Checks

Regardless of the technique used to generate redundancy, each sensor node must periodically check to ensure that its back-up data is still being stored correctly. If a node replicates its data to distant nodes, then its integrity checks and their responses must also travel further, thereby expending more energy. Moreover, the more frequently a node checks the correctness of its back-ups, the more energy it expends. Furthermore, additional energy is expended at the responding node which must generate a signature and transmit

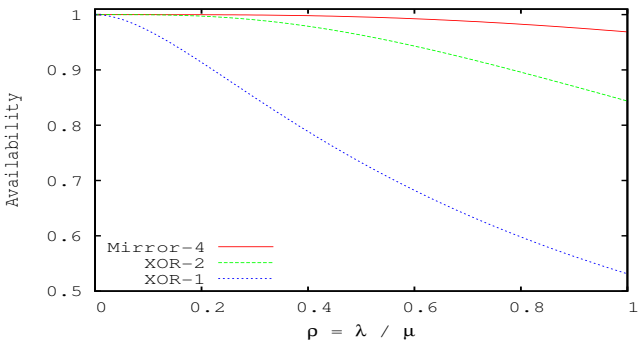


Figure 6. Data availability.

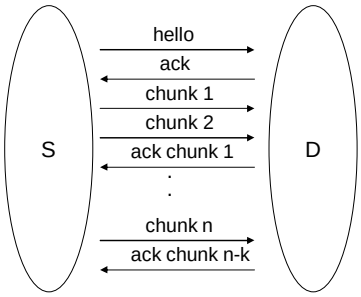


Figure 7. An originating node, *S*, uses this protocol to replicate its data to another node, *D*.

it back over multiple hops. However, in a system where node failure is frequent, it is necessary to detect small problems before they grow bigger and cause data loss. It may be energy-wise to allow small problems to become a little bigger, but not fatal, because the energy cost to restore redundancy is sub-linear. We are currently exploring the energy tradeoffs between more frequent integrity checks with that of the overall reliability of the system.

We plan to use algebraic signatures [18] to verify the correctness of remotely-stored redundant data. Although algebraic signatures are not cryptographically secure, they change in response to small changes in the data from which they are generated. Moreover, they can be used in conjunction with XOR or RS codes to ensure that a set of returned signatures is consistent. An algebraic signature operation requires a node to calculate a function on its own piece of stored redundant data, thereby, generating a small (4–8 byte) signature. When combined, these signatures obey the same relationship as the data from which they were generated; if the signatures form a valid code word in the XOR or RS scheme, the underlying data is highly likely to be consistent as well—the chance of agreement with an underlying error is approximately 2^{-b} for a b-bit signature.

3 Experimental Approach

We have developed a cost model to compute the total energy expenditure of making sensor network storage reliable. This total energy expenditure is comprised of I/O, processing, and radio costs, and includes the energy expended at the originating node as well as at each node that stores redundancy data. We also evaluate the storage overhead of mirroring, erasure coding, and of corresponding metadata. Our evaluation assumes that the energy expended per-byte to read/write data from SRAM and flash is the same. The cost of radio transmission is calculated by multiplying the number of bytes transmitted with the per-byte per-hop energy expenditure. Radio reception cost is calculated by multiplying the number of bytes received by the per-byte energy expenditure for reception. Most sensor nodes follow a "write-once, read-never, modify-never" access policy, therefore, nodes do not need to perform incremental back-ups: a file once written will not be modified during the nodes' deployment. Each node maintains metadata such as the originating node's ID, the chunk ID, and the receiving node's ID. Although it may be sufficient to store this metadata on either the originating node or on the receiving node, storing it on the originating node as well as each corresponding back-up node will help back-up the metadata, and prevent it from being a single point of failure.

An originating node, ready to back-up its data, sends a "hello" message, as shown in Figure 7, to n nodes to check if they have space to store its data. Back-up nodes that have sufficient storage respond with an "ack" message. The originating node spends energy in sending n "hello" messages and receiving m "ack" messages, where $m \leq n$. Each back-up node spends energy receiving a "hello" message and sending an "ack" message. Each originating node uses time stamps and message sequence numbers to keep track of what data has already been received and backed-up correctly by the back-up nodes. By doing this, if the connection between two communicating nodes is lost, the transmitting node can avoid unnecessary retransmissions. An originating node spends energy in transmitting data to each back-up node, while each back-up node spends energy in receiving and storing back-up data in its flash memory.

4 Optimizations

We are currently researching several optimizations that can help reduce energy requirements for making sensor network storage reliable. For example, it may be possible to piggy-back integrity check messages and responses on other network traffic such as "hello" or "ack" messages or on other traffic related to updating routing and neighborhood tables. Such piggy-backing has the potential of reducing transmission cost because integrity check messages are relatively small and the marginal cost of including additional information in another message is minimal. In order to reduce energy expenditure of reliability, some redundancy can be generated at remote nodes to reduce the total volume of data that must be transmitted over large distances. Sending all data to a remote node and letting it distribute it to its nearby neighbors may also be more energy efficient than the originating node distributing its data to all nodes. For example, in Figure 3, node 2 can transmit its data to node 10, which can distribute the data to nodes 5, 6, and 15. Energy expended in transmission can be further reduced by using some of the "routing" nodes or the intermediate nodes in the path between a source node and its destination back-up node.

5 Related Work

Koushanfar, *et al.* [8] identify computing, storage, communication, sensing, and actuating as resources and propose backing-up a resource running low with one that is abundantly available. However, the application software that computes resource availability may itself consume lots of energy. The solutions presented by Kamra, *et al.* [7] and Lin, *et al.* [10] are designed for sink-based network architectures. Although our solution is applicable to both distributed and centrally-controlled networks, we assume a distributed network architecture without a sink. Lin, *et al.* [11] use decentralized fountain codes to introduce redundancy into the network. Ghose, *et al.* [4] present a Resilient Data Centric Storage (R-DCS) scheme to reduce energy consumption while increasing resilience to node failures. Schemes presented by authors [4, 11] require a complete picture of the network. This may not always be possible with *ad hoc* networks [10]. In contrast, we assume nearly homogeneous nodes with no single point of failure. This assumption may not hold well in *ad hoc* networks deployed by dropping nodes from an

airplane or artillery shell. Dimakis, *et al.* [3] use decentralized erasure codes to reduce latency and unreliability between query times and the time at which data reaches the data collector. The authors assume a fixed ratio between the number of storage nodes and the number of nodes that contain original data.

6 Conclusion

"Sense and store" sensor networks are gaining popularity due to the recent availability of gigabyte-scale local storage on sensor nodes, and because storage operations are more energy efficient than radio operations. It is important to make the data stored locally on sensor nodes reliable because sensor nodes suffer from unusually high failure rates (both individual and correlated). We discussed three factors that influence energy-reliability tradeoffs—redundancy techniques, node choice, and frequency of integrity checks. We presented a simple analytical model for modeling the availability of a node's data, and are currently exploring these issues in more detail using simulation-based models. Our research on energy-reliability tradeoffs will enable long-term reliable storage in sensor nodes and enable their deployment in environments where frequent data collection is infeasible.

7 Acknowledgments

This work was supported in part by the Department of Education GANN program, the Department of Energy under award DE-FC02-06ER25768, the Institute for Scalable Scientific Data Management in cooperation with Los Alamos National Laboratory and industrial sponsors of the Storage Systems Research Center including Agámi Systems, Data Domain, Hewlett Packard, LSI Logic, NetApp, Seagate, and Symantec.

8 References

[1] www.arm.com/products/CPUs/ARM926EJ-S.html.

[2] CHONG, C.-Y., AND KUMAR, S. P. Sensor Networks: Evolution, Opportunities, and Challenges. In *Proc. of the IEEE* (2003), vol. 91, pp. 1247– 1256.

[3] DIMAKIS, A. G., PRABHAKARAN, V., AND RAMCHANDRAN, K. Ubiquitous access to distributed data in large-scale sensor networks through decentralized erasure codes. In *IPSN '05: Proceedings of the 4th international symposium on Information processing in sensor networks* (2005), IEEE Press, p. 15.

[4] GHOSE, A., GROSSKLAGS, J., AND CHUANG, J. Resilient data-centric storage in wireless ad-hoc sensor networks. In *Proceedings of the 4th International Conference on Mobile Data Management* (2003), Springer-Verlag, pp. 45–62.

[5] GREENAN, K., MILLER, E. L., AND SCHWARZ, T. Analysis and construction of Galois fields for efficient storage reliability. Tech. Rep. Technical Report UCSC-SSRC-07-09, University of California, Santa Cruz, 2007.

[6] HAFNER, J. L. WEAVER codes: Highly fault tolerant erasure codes for storage systems. In *Proceedings of the Second USENIX Conf. on File and Storage Technologies (FAST)* (2005).

[7] KAMRA, A., FELDMAN, J., MISRA, V., AND RUBENSTEIN, D. Data persistence for zero-configuration sensor networks. In *ACM Special Interest Group on Data Communications (SIGCOMM)* (2006).

[8] KOUSHANFAR, F., POTKONJAK, M., AND SANGIOVANNI-VINCENTELLI, A. Fault tolerance techniques in wireless ad-hoc sensor networks. In *Proc. of IEEE Sensors* (2002), vol. 2, pp. 1491– 1496.

[9] LAI, S. Current status of the phase change memory and its future. *IEDM Technical Digest* (2003), 10.1.1– 10.1.4.

[10] LIN, S., ARAI, B., AND GUNOPULOS, D. Reliable hierarchical data storage in sensor networks. In *19th Int'l Conf. on Scientific and Statistical Database Mgmt, SSDBM* (2007), pp. 26–35.

[11] LIN, Y., LIANG, B., AND LI, B. Data persistence in large-scale sensor networks with decentralized fountain codes. In *INFOCOM 2007. 26th IEEE Int'l Conf. on Computer Communications* (2007), pp. 1658–1666.

[12] MATHUR, G., DESNOYERS, P., GANESAN, D., AND SHENOY, P. Ultra-low power data storage for sensor networks. In *IPSN '06: Proceedings of the fifth international conference on Information processing in sensor networks* (2006), ACM, pp. 374–381.

[13] MITRA, A., BANERJEE, A., NAJJAR, W., ZEINALIPOUR-YAZTI, D., KALOGERAKI, V., AND GUNOPULOS, D. High-Performance Low Power Sensor Platforms Featuring Gigabyte Scale Storage. In *IEEE/ACM 3rd Int'l Workshop on Measurement, Modelling, and Perf. Anal. of WSNs* (2005).

[14] NATH, S., YU, H., GIBBONS, P. B., AND SESHAN, S. Subtleties in tolerating correlated failures in wide-area storage systems. In *Networked Systems Design and Implementation (NSDI)* (2006).

[15] PLANK, J. S. A tutorial on Reed-Solomon coding for fault-tolerance in RAID-like systems. *Software, Practice and Experience 27*, 9 (1997), 995–1012.

[16] RHEA, S., EATON, P., GEELS, D., WEATHERSPOON, H., ZHAO, B., AND KUBIATOWICZ, J. Pond: the OceanStore prototype. In *Proceedings of the Second USENIX Conf. on File and Storage Technologies (FAST)* (2003), pp. 1–14.

[17] SCHWARZ, T. *Reliability and Performance of RAID Systems*. PhD thesis, Univ. of California at San Diego, 1994.

[18] SCHWARZ, T. S. J., AND MILLER, E. L. Store, forget, and check: Using algebraic signatures to check remotely administered storage. In *ICDCS '06: Proceedings of the 26th IEEE International Conference on Distributed Computing Systems* (2006), IEEE Computer Society, p. 12.

[19] SHENG, B., LI, Q., AND MAO, W. Data storage placement in sensor networks. In *ACM International Symposium on Mobile Ad Hoc Networking and Computing* (2006), pp. 344–355.

[20] WERNER-ALLEN, G., LORINCZ, K., JOHNSON, J., LEES, J., AND WELSH, M. Fidelity and yield in a volcano monitoring sensor network. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation* (2006), USENIX Association, pp. 381–396.

[21] WYLIE, J. J., AND SWAMINATHAN, R. Determining fault tolerance of XOR-based erasure codes efficiently. In *Dependable Systems and Networks (DSN)* (2007), pp. 206–215.

Reliability of flat XOR-based erasure codes on heterogeneous devices

Kevin M. Greenan*†, Ethan L. Miller†, and Jay J. Wylie*
kmgreen@cse.ucsc.edu, elm@cs.ucsc.edu, jay.wylie@hp.com

Abstract

XOR-based erasure codes are a computationally-efficient means of generating redundancy in storage systems. Some such erasure codes provide irregular fault tolerance: some subsets of failed storage devices of a given size lead to data loss, whereas other subsets of failed storage devices of the same size are tolerated. Many storage systems are composed of heterogeneous devices that exhibit different failure and recovery rates, in which different placements—mappings of erasure-coded symbols to storage devices—of a flat XOR*-based erasure code lead to different reliabilities. We have developed redundancy placement algorithms that utilize the structure of flat* XOR*-based erasure codes and a simple analytic model to determine placements that maximize reliability. Simulation studies validate the utility of the simple analytic reliability model and the efficacy of the redundancy placement algorithms.*

1. Introduction

Erasure codes such as replication, RAID 5, and Reed-Solomon codes are the means by which storage systems are typically made reliable. Reed-Solomon codes and other maximum distance separable (MDS) codesk, provide the best tradeoff between fault tolerance and space-efficiency, but are computationally the most demanding type of erasure code. In addition to these traditional erasure codes, there are a number of proposals for novel, non-MDS erasure codes that exclusively use XOR operations to generate redundancy (e.g., [9, 10, 23]). Such XOR-based codes can be computationally more efficient than MDS codes, but offer an irregular tradeoff between performance, space-efficiency, and fault tolerance.

Methods to evaluate the space-efficiency and performance tradeoff for XOR-based codes are well understood [19, 11, 18]. However, some XOR-based erasure codes exhibit irregular fault tolerance: some subsets of failed storage devices of a given size lead to data loss, whereas other subsets of failed storage devices of the same size are tolerated. There have been many recent advances in understanding the fault tolerance [12, 23] and concomitant reliability [20, 13, 8] of such codes. However, all of these advances assume a *homogeneous* set of storage devices that all fail and recover at similar rates.

The contributions of this work are fourfold. First, we define a novel reliability problem in storage systems, the *redundancy placement problem*. Given a storage system comprised of *heterogeneous* storage devices with known failure and recovery rates, how should erasure-coded symbols be mapped to devices to maximize reliability? The redundancy placement problem is trivial for MDS codes because they exhibit regular fault tolerance—an m-tolerant MDS code will *never* lose data if m devices fail and *always* lose data if $m + 1$ devices fail—so all placements have the same reliability. For non-MDS codes though, the redundancy placement problem is non-trivial to solve. Second, we propose a simple analytic model related to mean time to data loss (MTTDL). The model is called the Relative MTTDL Estimate (RME), and it allows the *relative* reliability of different placements to be compared in a computationally efficient manner. Third, we propose two redundancy placement algorithms that use the structure of the XOR-based erasure code and the RME to determine a placement that maximizes (estimated) reliability. Fourth, we empirically demonstrate, via simulation, that the RME correctly orders different placements with regard to their reliability, and that the redundancy placement algorithms identify placements that maximize reliability.

The outline of the paper is as follows. In §2 and §3 we provide background on erasure codes, replica placement algorithms, and our prior work. We describe the RME and our redundancy placement algorithms in §4; we evaluate them in §5. We discuss the limitations of our work in §6 and then conclude in §7.

2. Background

An erasure code consists of n symbols, k of which are *data symbols*, and m of which are *parity symbols* (re-

*Hewlett-Packard Labs
†UCSC. Supported by the Petascale Data Storage Institute under Department of Energy award DE-FC02-06ER25768.

dundant symbols). We only consider *systematic* erasure codes—codes that keep the original data symbols and solely add parity symbols—because their use is generally considered a necessity to ensure good common case performance.

A maximum distance separable (MDS) code uses m redundant symbols to tolerate all erasures of size m or less [17]. Many MDS codes generate redundancy using k Galois field multiplies and $k-1$ XORs per parity symbol (e.g., Vandermonde Reed-Solomon codes). A Galois field multiplication operation can be transformed into multiple XOR operations (e.g., Cauchy Reed-Solomon codes). Parity-check array codes are another class of MDS codes that only use XOR operations to generate redundancy (e.g., RAID 4, EVENODD [3], and Row-Diagonal Parity [4]).

Hafner has categorized the construction of array codes as HoVer constructions: codes with parity symbols in both/either *Ho*rizontal and/or *Ver*tical dimensions of the array [10]. The class of codes we study are *horizontal* codes. We go beyond this, and refer to the codes we study as *flat* codes: horizontal XOR-based codes comprised of exactly one row (i.e., exactly one symbol, data or parity, per disk). Flat codes are distinct from most parity-check array codes which require multiple rows of symbols; RAID 4 is the exception because it is both a parity-check array code and a flat code.

The impact of erasure code choice on performance is a well-studied area [11]. In the grid storage community, the *read overhead* of certain classes of XOR-based erasure code is of interest. Plank et al. analyzed the *read overhead* of moderate-sized XOR-based codes using Monte Carlo methods [19] and of small-sized codes using deterministic methods [18].

A *replica placement* algorithm maps replicas to devices. Traditionally, this is done to improve performance: to reduce response time of accesses, to balance load, and for distributed caching. We exclusively consider the replica placement problem as it pertains to reliability. We use the term *redundancy placement* rather than *replica placement* because our emphasis is on the placement of erasure-coded data and parity symbols.

Our work on redundancy placement differs substantially from prior work on replica placement. In traditional RAID (erasure-coded) storage systems, many stripes of size n are placed on $N > n$ devices for performance reasons (e.g., parity declustering to reduce recovery time [1]). Thomasian and Blaum evaluate the reliability impact of various policies for mirrored disks [22], and Lian et al. [15] evaluate the difference in reliability between random and sequential placement policies for erasure-coded data. Both studies only consider homogeneous devices. In contrast, the competitive hill climbing replica placement algorithm places many distinct files, each replicated n times, on N heterogeneous servers in a manner that maximizes the availability of the least available file [5]. The Multi-Object Assignment Toolkit (MOAT) places many distinct objects, each replicated n times, on N heterogeneous devices to maximize the availability of multi-object operations in the face of correlated failures [24].

Replica placement algorithms place n replicas on $N > n$ devices. Our redundancy placement algorithms place n erasure-coded symbols on n heterogeneous devices. The non-MDS nature of flat XOR-based codes makes the redundancy placement problem both novel and non-trivial. There are other non-MDS XOR-based codes, e.g., Weaver codes [9].

There are many techniques beyond simple redundancy to improve storage system reliability, such as checksums, snapshots, scrubbing, auditing, and backup to tape. However, questions such as where to place backup copies or checksums to maximize reliability are outside of the scope of this work. Such questions require different models to answer that necessarily include metrics other than reliability, such as cost and performance. The work of Gaonkar et al. [7] automates the design of storage solutions that meet cost, performance, and reliability requirements. Their approach essentially solves a replica placement problem across heterogeneous tiers of storage that employ distinct reliability mechanisms.

3. Reliability of flat XOR-based codes

In this section, we describe our recent work on evaluating the reliability of flat XOR-based codes. First, we review the Minimal Erasures List (MEL), a fault tolerance metric for flat codes [23]. Second, we discuss the Fault Tolerance Vector (FTV), another fault tolerance metric for flat codes. Finally, we review the High-Fidelity Reliability (HFR) Simulator, a Monte Carlo reliability simulator, especially designed to simulate the reliability of (flat) XOR-based codes [8]. The redundancy placement algorithms we have developed use the structure of the MEL to make placement decisions. The FTV is used for comparison purposes in the evaluation section. The HFR Simulator is used to validate the efficacy of the redundancy placement algorithms.

Traditionally, the *Hamming distance* is used to describe the fault tolerance of an erasure code: a code tolerates all sets of erasures smaller than the Hamming distance. The Hamming distance completely describes the fault tolerance of MDS codes, since all erasures at or beyond the Hamming distance result in data loss. Flat erasure codes can be non-MDS: they tolerate some sets of erasures at and beyond the Hamming distance. We previously developed the Minimal Erasures (ME) Algorithm to efficiently analyze a flat erasure code and characterize its fault tolerance [23].

Consider the following definitions: A *set of erasures* is a set of erased symbols; an *erasure pattern* is a set of erasures

that results in at least one data symbol being irrecoverable; and, a *minimal erasure* is an erasure pattern in which every erasure is necessary and sufficient for it to be an erasure pattern. The ME Algorithm determines the *minimal erasures list* (MEL) of an erasure code: the list of the code's minimal erasures, which completely describes the fault tolerance of an erasure code. The MEL can be transformed into a *minimal erasures vector* MEV in which the ith element is the total number of minimal erasures of size i in the MEL. The length of the shortest minimal erasure in the MEL is the Hamming distance of the code, and so the first non-zero entry in the MEV corresponds to the Hamming distance.

We now present two flat codes in detail to provide examples of minimal erasures, the MEL, and the MEV. We denote each code by (k,m)-NAME where k is the number of data symbols, m is the number of parity symbols, and NAME is the class of the code. Every code is described by a listing of m bitmaps, one for each parity symbol, displayed as an integer. Since we only consider systematic codes, the parity symbols are $s_k, \ldots, s_{n-1}$. A bitmap describes the data symbols that participate in a parity equation and is an integer in the range $[1, \ldots, 2^k - 1]$. For instance, consider (4,4)-RAID 10 specified by the parity bitmaps $\langle 1, 2, 4, 8 \rangle$. The first parity symbol for this code, s_4, is simply a replica of s_0 and so the bitmap is 1 (i.e., $s_4 = s_0$ because $1 = 2^0$). A more complex example is (5,3)-FLAT with parity bitmaps $\langle 7, 11, 29 \rangle$. The first parity symbol for this code, s_5, has bitmap 7 because it is computed as the XOR of data symbols s_0, s_1, and s_2 (i.e., $s_5 = s_0 \oplus s_1 \oplus s_2$ because $7 = 2^0 + 2^1 + 2^2$).

The (4,4)-RAID 10 code is an example of a common RAID technique that simply replicates each data symbol. RAID 10 is a flat erasure code that tolerates any single disk failure. The MEL for the code is $\{(s_0, s_4), (s_1, s_5), (s_2, s_6), (s_3, s_7)\}$ and the MEV is $(0, 4, 0, 0)$. The MEL for (4,4)-RAID 10 is intuitive: whenever any pair of devices that store the same replicated symbol fails, data is lost. The MEV simply summarizes the count of minimal erasures of each size up to m. The MEL for (5,3)-FLAT is $\{(s_4, s_7), (s_0, s_1, s_4), (s_0, s_1, s_7), (s_0, s_2, s_6), (s_0, s_3, s_5), (s_1, s_2, s_3), (s_1, s_5, s_6), (s_2, s_4, s_5), (s_2, s_5, s_7), (s_3, s_4, s_6), (s_3, s_6, s_7)\}$, and, the MEV is $(0, 1, 10)$. This code better illustrates the irregularity that non-MDS flat codes can exhibit. There is no obvious structure or symmetry to the sets of device failures which lead to data loss.

The Fault Tolerance Vector (FTV) indicates the probability that data is lost given some number of failures. To construct the FTV, the MEL is transformed into the *erasures list* (EL). The erasures list consists of every erasure pattern for a code. The EL is a super set of the MEL, and every element in it is either a minimal erasure or a super set of at least one minimal erasure. The *erasures vector* (EV) is to the EL what the MEV is to the MEL, and is easily determined given the EL. Finally, the EV is transformed into the FTV. Let the ith entry of the EV be e_i. For a code with n symbols, the ith entry of the FTV is $e_i / \binom{n}{i}$. The FTV is the complement of the *conditional probabilities* vector described by Hafner and Rao [13].

The High-Fidelity Reliability (HFR) Simulator [8] is similar to the simulator developed by Elerath and Pecht [6]. Both are Monte Carlo reliability simulators that simulate disk failure, disk recovery, sector failure, and sector scrubbing, and both can use Weibull distributions for such failure and recovery rates. However, the HFR Simulator is *high-fidelity* in the sense that it simulates the reliability of non-MDS erasure codes that can tolerate two or more disk failures, with regard to both disk and sector failures. It is non-trivial to extend the methods of Elerath and Pecht in this manner.

The difficulty in simulating non-MDS codes is in efficiently determining if a specific set of failures leads to data loss. The HFR Simulator has two modes of bookkeeping that allow it to efficiently determine if a set of device failures leads to data loss: via the MEL, and via the FTV. Using the MEL permits the HFR Simulator to accurately determine if a specific set of failures leads to data loss. Therefore, it is the method we must use to simulate the reliability of a redundancy placement of a flat code on heterogeneous devices. The FTV is a coarse-grained metric that does not capture the details of a specific placement of symbols on heterogeneous devices; however, it describes the fault tolerance of the median placement and so is used for comparative purposes in §5.

4. Redundancy placement algorithms

We have developed two redundancy placement algorithms that identify placements of erasure-coded symbols on heterogeneous storage devices with known failure and repair rates which maximize reliability. One redundancy placement algorithm is based on brute force computation and the other is based on simulated annealing.

More formally, let S be the set of symbols in the erasure code and D be the *configuration* (set of heterogeneous devices). For a code with n symbols, $S = \{s_0, \ldots, s_{n-1}\}$ and $D = \{d_0, \ldots, d_{n-1}\}$. A placement, ρ, is a bijective function that uniquely maps each symbol in the erasure code to a single device, i.e., $\rho : S \leftrightarrow D$. The goal of the redundancy placement algorithms is to find a placement, ρ, that maximizes reliability.

4.1. Relative MTTDL estimate (RME)

We now introduce the simple analytic model that underlies both redundancy placement algorithms: the Relative MTTDL Estimate (RME). The RME can be used to compare

the reliability of different placements. It is constructed to correlate with the expected mean time to data loss (MTTDL), but it does not accurately estimate the MTTDL. The RME can be used to compare the relative merit of different placements, but not to determine if some placement meets a specific reliability requirement. The efficacy of the RME to correctly order placements by MTTDL is demonstrated in §5.

At a high level, the RME is the inverse of an estimate of the expected unavailability of a given placement. It is based on the MEL and a simple analytic device model. The MEL is a concise, exact description of a code's irregular fault tolerance. The MEL contains each minimal set of data and parity symbol failures that lead to data loss. Let $\mathrm{u}(d)$ be the expected unavailability of device d. To calculate $\mathrm{u}(d)$, the mean time to repair (MTTR) of d is simply divided by its mean time to failure (MTTF). This analytic model ignores sector failures and scrubbing, as well as the exact distribution of the device failures and repairs. Moreover, it is premised on failures being independent. The RME is calculated via the following function of the redundancy placement ρ, device unavailability u, and MEL:

$$\text{RME} = \left(\sum_{f \in \text{MEL}} \prod_{s \in f} \mathrm{u}(\rho(s)) \right)^{-1}.$$

The sum of products is inverted because RME values are values that should be maximized to improve reliability, just like MTTDL values.

The RME for the (4,4)-RAID 10 code described in §3 is as follows:

$$\begin{aligned} \text{RME} \quad = \quad & (\mathrm{u}(\rho(s_0))\mathrm{u}(\rho(s_4)) + \mathrm{u}(\rho(s_1))\mathrm{u}(\rho(s_5)) + \\ & \mathrm{u}(\rho(s_2))\mathrm{u}(\rho(s_6)) + \mathrm{u}(\rho(s_3))\mathrm{u}(\rho(s_7)))^{-1}. \end{aligned}$$

Consider a configuration in which the first 4 devices have expected device unavailability of 1.2×10^{-4} and the second 4 devices have expected device unavailability of 2.4×10^{-5}. Note that the more reliable a device is, the lower its device unavailability number, so the first 4 devices are less reliable than the second 4 devices in this configuration. Now consider two distinct placements. In the first placement, the first 4 symbols are placed on the first 4 devices, and the second 4 symbols are placed on the second 4 devices, and so the RME $= 86.8 \times 10^6$. In the second placement, the "odd symbols" (i.e., s_1,s_3,s_5, and s_7) are placed on the first 4 devices, and the "even symbols" on the second 4 devices, and so the RME $= 33.4 \times 10^6$. The first placement splits the pair of replicated symbols that occur in each minimal erasure, mapping one to a less reliable device and the other to a more reliable device. In contrast, the second placement places all of the symbols from two minimal erasures (the "odd symbols") on the less reliable devices, which, intuitively, is a less reliable placement. These RME values follow our intuition about the relative reliability of placements; this intuition is confirmed via simulation in §5.1.

There are several reasons for using the simple analytic model. First, the simplicity of the analytic device model permits efficient evaluation of the RME and so permits orders of magnitude more distinct placements to be evaluated than simulation methods in the same period of time. Second, the model only has to produce an RME that accurately orders different sets of device failures according to the likelihood that they contribute to data loss. The product of expected device unavailability accomplishes this task. Third, in a system with any redundancy, sector failures alone do not cause data loss; only multiple disk failures, or disk failures in conjunction with sector failures lead to data loss. Thus, the simple analytic model only needs to capture the reliability effects of disk failures. Fourth, we considered extending the approach of Hafner and Rao, who recently proposed a Markov model construction for XOR-based erasure codes based on homogeneous devices [13]. Extending their model to heterogeneous devices is not feasible because each device requires a distinct Markov model state per erasure pattern.

4.2. Brute force algorithm

The brute force redundancy placement (BF-RP) algorithm evaluates the RME for all possible placement and identifies the placement with the largest RME as the best placement. The RME is a simple equation that can be evaluated efficiently. Calculating an RME value requires $|\text{MEL}|$ additions and less than $m \times |\text{MEL}|$ multiplications. Consider the calculation of the RME for (4,4)-RAID 10 given above. It required four additions because $|\text{MEL}| = 4$, and four multiplications because each of the four minimal erasures consists of exactly two symbols. Since all minimal erasures consist of m or fewer symbols, each product requires $m - 1$ or fewer multiplications.

For a code with n symbols and n distinct devices, there are $n!$ possible placements to evaluate. Given the efficiency of the RME calculation, it is feasible to evaluate the RME for every possible placement for small codes. For example, in §5 the BF-RP algorithm is used to find the best placement for some codes with $n = 12$. Each such execution of the BF-RP performs $12! = 479001600$ RME calculations to determine the best placement.

4.3. Simulated annealing algorithm

For large codes, the factorial number of distinct placements make it is infeasible to apply the BF-RP algorithm. The best placement for a code maximizes the RME value. Therefore, the problem of finding the best placement can be understood as an optimization problem. Unfortunately, the nonlinear structure of the RME equation—all of the terms

in the summation are products of variables to be assigned via the optimization—precludes linear optimization techniques.

Fortunately, there are many non-linear optimization techniques. An approach that requires little work, in terms of formulating constraint equations, is simulated annealing [14]. This made simulated annealing, a stochastic optimization technique, appealing as the first optimization approach for us to evaluate. Simulated annealing uses randomization to find a solution; however, there is a chance that the solution found is not globally optimal.

The simulated annealing redundancy placement (SA-RP) algorithm takes an MEL and a configuration of devices as input. The SA-RP algorithm is initialized with a randomly selected placement. Each *step* in SA-RP is based on a random number of random swaps of mappings in the current placement. As the algorithm proceeds, the number of random swaps performed at each step decreases. This is the manner in which we capture the "cooling" aspect of simulated annealing, in which randomness is reduced over time so that some locally optimal placement is settled upon. In SA-RP, we include parameters to backtrack if a step that decreased the RME does not lead to a larger RME after some number of additional steps. The SA-RP algorithm is invoked multiple times, while keeping track of the best RME value found over different invocations. Because each invocation is initialized with a different random placement, repeated invocations find distinct locally maximal placements (RME values).

Unfortunately, simulated annealing does not lend itself to many practical rigorous statements about the quality of solution found. However, our empirical results indicate that the SA-RP algorithm quickly produces good solutions.

5. Evaluation

To evaluate the BF-RP and SA-RP algorithms, we consider configurations that have devices with failure models between two bounds. The first device failure model is based on that used by Elerath and Pecht (cf. Table 2 in [6]). Disk failures are distributed according to a Weibull distribution with parameters $\gamma = 0$, $\eta = 500000$, and $\beta = 1.12$. (Note that we "rounded up" the η parameter of 461386 hours used by Elerath and Pecht.) Disk recoveries are distributed according to a Weibull distribution with $\gamma = 6$, $\eta = 12$, and $\beta = 2$. We refer to the first device as the 500k device because its expected MTTF is 500 thousand hours. The 500k device is the most reliable device we consider in the evaluation. We refer to the least reliable device as the 100k device. The 100k device differs from the 500k device only in its MTTF: $\eta = 100000$ instead of $\eta = 500000$. To calculate the RME, only the MTTF for disk failure and the MTTR for a recovery is used. The HFR Simulator uses the specified Weibull distributions to simulate the MTTDL. All simulations are based on devices that exhibit only disk failures and recoveries; sector failures are not included in this evaluation.

There are two types of heterogeneous configurations we evaluate. *Bimodal configurations* consist of only two types of devices: 100k devices and 500k devices. For example, an 8-disk 3-bimodal configuration consists of 3 100k devices and 5 500k devices. *Uniform configurations* consist of one 100k device and one 500k device; the remaining devices have MTTF values uniformly distributed between $\eta = 100000$ and $\eta = 500000$. For example, an 8-disk uniform configuration consists of one device with each of the following η values: 100000, 157000, 214000, 271000, 328000, 385000, 442000, 500000. We evaluate 8-, 12-, and 20-disk configurations.

Table 1 lists the flat codes analyzed by the redundancy placement algorithms. The MEV and FTV for each code is listed. All of the codes have a Hamming distance of 2. The MEL is used to calculate the RME and so is more useful than the MEV for understanding the results in this section. The MEL for the (4,4)-RAID 10 and (5,3)-FLAT are given in §3. The MEL of the (6,2)-FLAT is $\{(s_0, s_1), (s_2, s_3), (s_2, s_6), (s_3, s_6), (s_4, s_5), (s_4, s_7), (s_5, s_7)\}$. The MEL for the larger codes is too verbose to list. The FTV is used for comparison purposes because the reliability simulated based on the FTV approximates the median reliability over all possible placements.

The specific flat codes listed in Table 1 were selected because, for the given values of k and m, they are the most fault tolerant flat codes [23]. One is the (4,4)-RAID 10 which was selected because it has a familiar structure. The specific values of k and m were selected so that all of the codes have a Hamming distance of 2. It takes many CPU days for the HFR Simulator to simulate a single data loss event for more fault-tolerant codes so we restricted the Hamming distance to ensure that the results of the redundancy placement algorithms could be validated via simulation.

Beyond the flat codes, we also included MDS codes with Hamming distance two to provide context. The placement of such codes does not affect their reliability because all sets of device failures of Hamming distance size or greater lead to data loss.

All MTTDL values in this section are measured in hours. The HFR Simulator is used to produce all of the MTTDL values [8]. Except where noted, MTTDL values in tables and annotated on histograms are based on simulations of 1000 data loss events. Each histogram consists of fifty bins equally sized from the shortest MTTDL to the longest MTTDL. The MTTDL values for data points in histograms are based on 100 data loss events and so exhibit greater variance.

code	Minimal Erasures Vector (MEV)	Fault Tolerance Vector (FTV)	parity bitmaps
(6,2)-FLAT	$(0, 7)$	$(0, 0.25,\ \ 1)$	$\langle 15, 51\rangle$
(5,3)-FLAT	$(0, 1, 10)$	$(0, 0.036, 0.29, 1)$	$\langle 7, 11, 29\rangle$
(4,4)-RAID 10	$(0, 4, 0, 0)$	$(0, 0.14,\ \ 0.43, 0.77, 1)$	$\langle 1, 2, 4, 8\rangle$
(10,2)-FLAT	$(0, 18)$	$(0, 0.27,\ \ 1)$	$\langle 127, 911\rangle$
(9,3)-FLAT	$(0, 5, 34)$	$(0, 0.076, 0.38, 1)$	$\langle 31, 227, 365\rangle$
(17,3)-FLAT	$(0, 19, 162)$	$(0, 0.10,\ \ 0.43, 1)$	$\langle 1023, 31775, 105699\rangle$
(16,4)-FLAT	$(0, 5, 80, 315)$	$(0, 0.026, 0.15, 0.48, 1)$	$\langle 511, 7711, 26215, 43691\rangle$

Table 1. Flat erasure codes.

code	100k	500k
(6,2)-FLAT	3.99×10^7	9.66×10^8
(5,3)-FLAT	2.88×10^8	6.89×10^9
(4,4)-RAID 10	6.59×10^7	1.83×10^9
(7,1)-MDS	1.01×10^7	2.55×10^8
(10,2)-FLAT	1.54×10^7	3.89×10^8
(9,3)-FLAT	5.28×10^7	1.40×10^9
(11,1)-MDS	4.06×10^6	1.03×10^8
(17,3)-FLAT	1.44×10^7	3.55×10^8
(16,4)-FLAT	5.42×10^7	1.32×10^9
(19,1)-MDS	1.55×10^6	3.60×10^7

Table 2. MTTDL of homog. config. in hours.

Table 2 lists MTTDL values of all the codes evaluated in this section in homogeneous configurations. Two configurations are listed: one based on 100k devices and the other based on 500k devices. Obviously, 500k homogeneous configurations are more reliable than 100k homogeneous configurations. The MTTDL of all the heterogeneous configurations fall between the MTTDL of these two homogeneous configurations.

5.1. Eight-disk configurations

In this section we exhaustively evaluate the three flat codes of size 8 on various configurations. First, consider the 4-bimodal distribution. Figures 1, 2, and 3 respectively show MTTDL histograms for (4,4)-RAID 10, (5,3)-FLAT, and (6,2)-FLAT. These histograms are constructed by simulating the MTTDL of the $8! = 40320$ possible placements.

Each histogram is annotated with a vertical line. The vertical line corresponds to the MTTDL for the FTV. The FTV is described in §3, it estimates the MTTDL of the median placement. In these figures, the FTV MTTDL is indeed near the median MTTDL over all possible placements.

Each histogram is also annotated with a series of lines labeled with integers. These lines are related to RME cal-

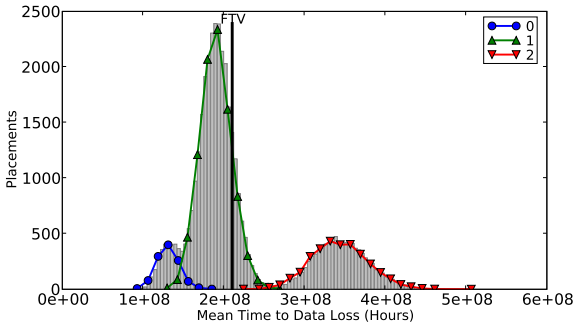


Figure 1. (4,4)-RAID10, 4-bimodal.

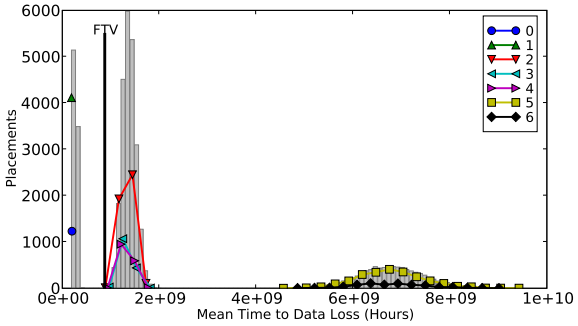


Figure 2. (5,3)-FLAT, 4-bimodal.

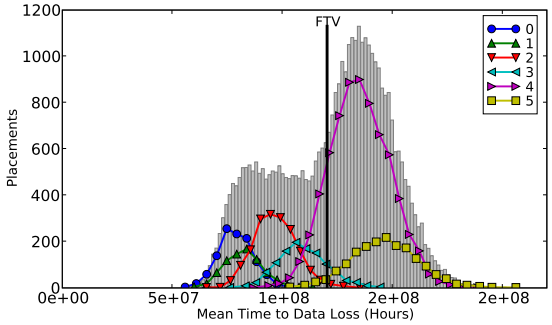


Figure 3. (6,2)-FLAT, 4-bimodal.

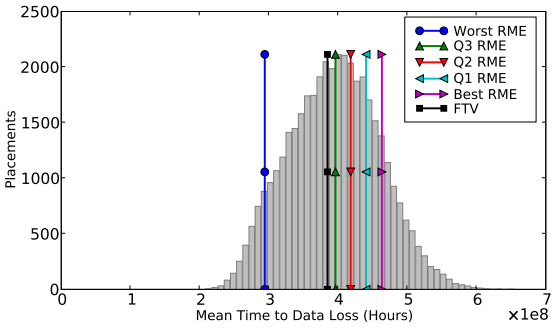


Figure 4. (4,4)-RAID10, uniform.

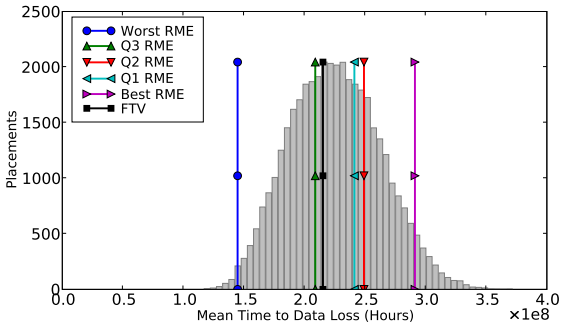


Figure 6. (6,2)-FLAT, uniform.

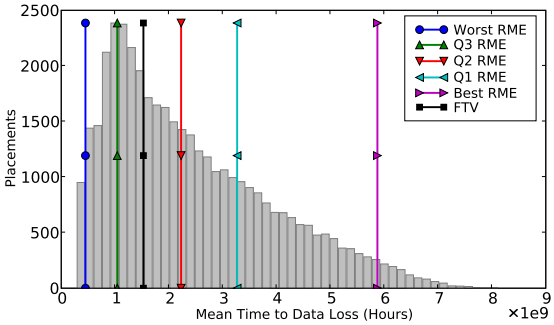


Figure 5. (5,3)-FLAT, uniform.

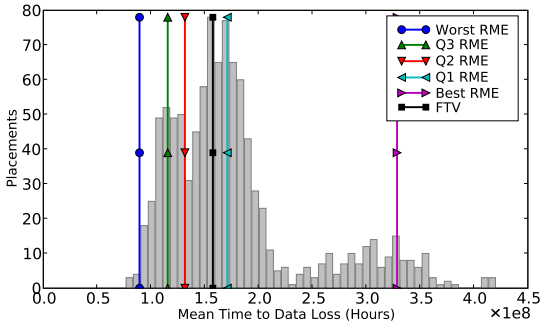


Figure 7. (9,3)-FLAT, 6-bimodal.

culations. For each of these codes, the BF-RP algorithm is used to determine the RME of each distinct placement. We were surprised to discover that for each of these codes, only a small number of distinct RME values were produced. From this, we hypothesized that there are *isomorphic placements*, i.e., different placements that have the same RME and MTTDL.

Each line on each histogram is effectively a sub-histogram for an isomorphic class of placements. The integer labels on the classes are in order of RME value, so line 0 has a lower RME than line 1. Note that we expected there to be no more than $\binom{8}{4} = 70$ distinct RME values for the 4-bimodal configuration because the first four and last four devices are identical. Figures 1, 2, and 3 show that there are respectively 3, 7, and 6 isomorphic classes.

To better understand isomorphic placements, consider (4,4)-RAID 10. The following are example placements for each isomorphic placement class: $0 : (s_1, s_3, s_5, s_7, s_0, s_2, s_4, s_6)$, $1 : (s_0, s_1, s_2, s_4, s_3, s_5, s_6, s_7)$, and $2 : (s_0, s_1, s_2, s_3, s_4, s_5, s_6, s_7)$. The first four symbols in each placement is on a 100k device, and the second four symbols are on a 500k device. We already discussed the placements for classes 0 and 2 in §4.1. The placement for class 1 is consistent with the prior discussion: one pair of replicated symbols is on the 100k devices and so we expect the MTTDL to fall between class 0 (two pairs of replicated symbols on 100k devices) and class 2 (no pairs of replicated symbols on 100k devices).

The ordering of sub-histograms in each experiment strongly support our hypothesis that the RME correctly orders different placements with regard to reliability. The spread within each sub-histogram is due to statistical variance; remember that each simulation to produce a histogram data point was run for only 100 iterations. Because the MEL for (5,3)-FLAT and (6,2)-FLAT contain more entries than that of (4,4)-RAID 10, they each have more isomorphic placement classes. The distribution of isomorphic placement classes is interesting: the density of placements in the median classes appears to be greater than in the "best" class. This suggests that good placements are less common.

Now consider the uniform configuration instead of the 4-bimodal configuration. Figures 4, 5, and 6 respectively show MTTDL histograms for (4,4)-RAID 10, (5,3)-FLAT, (6,2)-FLAT. The FTV MTTDL is annotated on these histograms. Sub-histograms for isomorphic placement classes are not presented. The uniform configuration leads to too many such classes to illustrate. To be more precise, there are respectively 105, 840, and 280 distinct classes. These values are all much lower than the $8! = 40320$ potential distinct RME values, and so these results also support

the idea of isomorphic placement classes. Instead of sub-histograms, a vertical line is shown for a placement from each of the following isomorphic placement classes: Worst RME, Q3 (third quartile) RME, Q2 (second quartile) RME, Q1 (first quartile) RME, and Best RME.

Our hypothesis is that the MTTDL of the placement from the Worst RME class would be less than that of the placement from the Q3 RME class, and so on. The results support this hypothesis. One exception is the results for Q1 RME and Q2 RME for (6,2)-FLAT which are out of order. The difference between the MTTDL results for Q1 RME and Q2 RME is small though; we conclude that the Q1 RME and Q2 RME placements simply have quite similar reliabilities.

We expect the FTV MTTDL to provide an estimate of the median placement, because it is computed using the FTV, a probability vector derived from the MEV. As a consequence the FTV MTTDL should align with the Q2 RME MTTDL. Our results support this hypothesis, allowing us to use the FTV MTTDL as a reference when comparing the reliability of placements.

Table 3 summarizes results for all of the bimodal configurations and the uniform configuration. For each code, the FTV MTTDL and the Best RME MTTDL are listed. In all cases, the Best RME MTTDL is better than that of the FTV MTTDL.

All of these experiments support our hypothesis that the RME can be used to identify placements of erasure-coded symbols that maximizes reliability. The reliability difference between the worst placements and best placements range from from around a factor of two for (4,4)-RAID 10 to an order of magnitude for (5,3)-FLAT. Table 3 shows that across all configurations, differences between the FTV MTTDL and the Best RME MTTDL range from no difference to a factor of six.

5.2. Twelve-disk configurations

For 12-disk configurations, it is not feasible to evaluate every possible placement via simulation, but it is feasible to do so via the RME metric. We ran the BF-RP algorithm for the (9,3)-FLAT, and (10,2)-FLAT codes for all possible bimodal configurations and the uniform configuration. We also ran the SA-RP algorithm on these configurations. We run the SA-RP algorithm for a total of 1000000 steps; if the RME does not improve in 25 steps, the placement reverts to the last best placement for this execution; if the best RME placement does not improve in 1000 steps, a new execution is initialized with a random placement. In all cases, the SA-RP algorithm identified a placement from the same isomorphic placement class as the BF-RP algorithm (i.e., its RME is the same as the Best RME).

To determine the quality of the placements selected by the BF-RP and SA-RP algorithms, we simulated the Best RME MTTDL and the FTV MTTDL for a subset of configurations. The results are listed in Table 4. In most cases, the MTTDL of the placement with the Best RME is significantly better than that of the FTV. For the 9-bimodal configuration, the MTTDL values for (10,2)-FLAT are effectively the same.

From the BF-RP results, we also can identify the Worst RME, Q3 RME, Q2 RME, and Q3 RME placements. We simulated the MTTDL of these placements as well as 1000 random placements to generate low fidelity histograms. An example of such a histogram for (9,3)-FLAT in the 6-bimodal configuration is given in Figure 7. The histograms further support the hypothesis that the RME metric correctly orders placements by reliability.

5.3. Twenty-disk configurations

For 20-disk configurations, it is infeasible to evaluate every possible placement via simulation or the RME metric. Instead, we use the SA-RP algorithm to identify an Approximate Best RME placement for these configurations. We ran the SA-RP algorithm for the (17,3)-FLAT, and (16,4)-FLAT codes for all of the bimodal configurations and the uniform configuration. The SA-RP algorithm is run in the same manner as for the 12-disk configurations.

To determine the quality of the placements selected by the SA-RP algorithm, we simulated the MTTDL of the Approximate Best RME placement found by SA-RP and compare it with the FTV MTTDL for a subset of configurations. The results are listed in Table 5. In all cases, the Approximate Best RME MTTDL is significantly better than the FTV MTTDL.

6. Discussion

The redundancy placement algorithms based on the RME effectively find reliable placements. However, we have not characterized how sensitive the redundancy placement algorithms are to different failure models. Specifically, we do not have a good characterization of the conditions necessary for the RME to correctly order placements by their reliability.

We believe that extensive simulation will permit us to do such characterization. Unfortunately, the HFR Simulator is currently too slow to run the potentially millions of analyses necessary to do such characterization. The existence of isomorphic placement classes suggests an avenue for speeding up the redundancy placement algorithms. If we identify the sets of symbols that are *equivalent*, i.e., that if swapped yield a new placement in the same isomorphic class, then both the BF-RP and SA-RP algorithms can operate over a much smaller state space. Such a speedup may facilitate better RME characterization.

The specific device models used in the evaluation are based on the distributions that Elerath and Pecht used [6]. We believe that the models of Elerath and Pecht are as good

	(4,4)-RAID 10		(5,3)-FLAT		(6,2)-FLAT		
configuration	FTV	Best RME	FTV	Best RME	FTV	Best RME	(7,1)-MDS
1-bimodal	8.60×10^8	8.39×10^8	3.30×10^9	6.90×10^9	4.98×10^8	6.20×10^8	1.19×10^8
2-bimodal	4.74×10^8	5.97×10^8	1.94×10^9	6.53×10^9	2.94×10^8	3.74×10^8	6.71×10^7
3-bimodal	3.01×10^8	4.35×10^8	1.23×10^9	6.40×10^9	1.72×10^8	2.54×10^8	4.41×10^7
4-bimodal	1.69×10^8	3.49×10^8	9.24×10^8	6.37×10^9	1.33×10^8	1.47×10^8	2.96×10^7
5-bimodal	1.51×10^8	1.81×10^8	6.07×10^8	6.62×10^9	8.76×10^7	9.67×10^7	2.05×10^7
6-bimodal	1.09×10^8	1.19×10^8	4.53×10^8	6.89×10^9	6.50×10^7	7.42×10^7	1.61×10^7
7-bimodal	8.29×10^7	8.42×10^7	3.40×10^8	1.35×10^9	4.99×10^7	5.18×10^7	1.26×10^7
uniform	4.34×10^8	4.88×10^8	1.56×10^9	6.11×10^9	2.34×10^8	2.79×10^8	5.60×10^7

Table 3. MTTDL of 8-disk configurations in hours.

	(9,3)-FLAT		(10,2)-FLAT		
configuration	FTV	Best RME	FTV	Best RME	(11,1)-MDS
3-bimodal	3.45×10^8	7.88×10^8	9.83×10^7	1.31×10^8	3.13×10^7
6-bimodal	1.57×10^8	3.30×10^8	4.25×10^7	5.14×10^7	1.27×10^7
9-bimodal	8.81×10^7	1.06×10^8	2.57×10^7	2.54×10^7	5.75×10^6
uniform	2.70×10^8	5.63×10^8	8.01×10^7	9.59×10^7	2.77×10^7

Table 4. MTTDL of 12-disk configurations in hours.

as any currently available. Recently published analyses of failure data [21, 16, 2] will hopefully result in better failure models. We expect that such models will change the MTTDL values, but not the placement that is most reliable. The RME is based on the assumption that failures are independent. The RME equation may have to change if significant correlation is found in failure models.

When we developed the RME metric, we assumed that sector failures would have a secondary effect on placement decisions and so could be excluded from the RME metric. We have some initial results for the RME metric in systems with sector failures. For codes with a Hamming distance greater than 2, the RME still correctly order placements by reliability. For codes with a Hamming distance of 2, data loss events are dominated by single-disk, single-sector failures. For such codes, if every symbol occurs in at least one minimal erasure of size two, e.g., like (6,2)-FLAT, then placement had little affect on overall reliability. Whereas, for (5,3)-FLAT, only the symbols s_4 and s_7 occur in a minimal erasure of size 2, and so placements based on the RME maximize reliability.

7. Conclusions

We introduced the novel *redundancy placement problem* in which a mapping, called a *placement*, of the symbols in a flat XOR-based code onto a set of heterogeneous storage devices with known failure and recovery rates must be found to maximize reliability. To solve this problem, we developed the Reliability MTTDL Estimate (RME), a simple model based on estimated device unavailability and the Minimal Erasures List (MEL), a concise characterization of the fault tolerance of an XOR-based code. We developed two redundancy placement algorithms, the BF-RP algorithm based on brute force computation, only suitable for small redundancy placement problems, and the SA-RP based on simulated annealing, and suitable for larger problems. Simulation results based on the High-Fidelity Reliability (HFR) Simulator provide extensive empirical evidence that the RME correctly orders different placements for a given code by MTTDL. Additional simulation results suggest that the placements found by the SA-RP algorithm are significantly more reliable than the median placement. The results of BF-RP algorithm lead us to realize the existence of *isomorphic placements*, sets of placements which have the same MTTDL.

References

[1] G. A. Alvarez, W. A. Burkhard, and F. Cristian. Tolerating multiple failures in RAID architectures with optimal storage and uniform declustering. In *ISCA-1997: 24th Annual International Symposium on Computer Architecture*, pages 62–72, Denver, CO, June 1997. ACM.

	(17,3)-FLAT		(16,4)-FLAT			
configuration	FTV	Approx. Best RME	FTV	Approx. Best RME	(19,1)-MDS	(18,2)-MDS
5-bimodal	8.94×10^7	1.29×10^8	3.59×10^8	1.39×10^9	9.63×10^6	1.50×10^{10}
10-bimodal	4.36×10^7	5.02×10^7	1.89×10^8	1.33×10^9	4.48×10^6	5.71×10^9
15-bimodal	2.33×10^7	2.61×10^7	8.72×10^7	2.85×10^8	2.53×10^6	1.94×10^9
uniform	8.49×10^7	9.62×10^7	3.38×10^8	8.71×10^8	8.49×10^6	1.27×10^{10}

Table 5. MTTDL of 20-disk configurations in hours.

[2] L. N. Bairavasundaram, G. R. Goodson, S. Pasupathy, and J. Schindler. An analysis of latent sector errors in disk drives. *SIGMETRICS Perform. Eval. Rev.*, 35(1):289–300, 2007.

[3] M. Blaum, J. Brady, J. Bruck, and J. Menon. EVENODD: An efficient scheme for tolerating double disk failures in RAID architectures. *IEEE Trans. Comput.*, 44(2):192–202, 1995.

[4] P. Corbett, B. English, A. Goel, T. Grcanac, S. Kleiman, J. Leong, and S. Sankar. Row-diagonal parity for double disk failure correction. In *FAST-2004: 3rd USENIX Conference on File and Storage Technologies*, pages 1–14. USENIX Association, March 2004.

[5] J. R. Douceur and R. P. Wattenhofer. Optimizing file availability in a secure serverless distributed file system. In *Symposium on Reliable Distributed Systems*. IEEE, 2001.

[6] J. F. Elerath and M. Pecht. Enhanced reliability modeling of raid storage systems. In *DSN-2007*, pages 175–184. IEEE, June 2007.

[7] S. Gaonkar, K. Keeton, A. Merchant, and W. H. Sanders. Designing dependable storage solutions for shared application environments. In *DSN-2006: The International Conference on Dependable Systems and Networks*, pages 371–382. IEEE, June 2006.

[8] K. M. Greenan and J. J. Wylie. High-fidelity reliability simulation of erasure-coded storage. Technical Report (to appear), Hewlett-Packard Labs.

[9] J. L. Hafner. WEAVER Codes: Highly fault tolerant erasure codes for storage systems. In *FAST-2005: 4th USENIX Conference on File and Storage Technologies*, pages 212–224. USENIX Association, December 2005.

[10] J. L. Hafner. HoVer erasure codes for disk arrays. In *DSN-2006: The International Conference on Dependable Systems and Networks*, pages 217–226. IEEE, June 2006.

[11] J. L. Hafner, V. Deenadhayalan, T. Kanungo, and K. Rao. Performance metrics for erasure codes in storage systems. Technical Report RJ–10321, IBM, August 2004.

[12] J. L. Hafner, V. Deenadhayalan, K. Rao, and J. A. Tomlin. Matrix methods for lost data reconstruction in erasure codes. In *FAST-2005: 4th USENIX Conference on File and Storage Technologies*, pages 183–196. USENIX Association, December 2005.

[13] J. L. Hafner and K. Rao. Notes on reliability models for non-MDS erasure codes. Technical Report RJ–10391, IBM, October 2006.

[14] S. Kirkpatrick, C. Gelatt Jr.., and M. Vecchi. Optimization by simulated annealing. *Science*, 220, 4598:671–680, May 1983.

[15] Q. Lian, W. Chen, and Z. Zhang. On the impact of replica placement to the reliability of distributed brick storage systems. In *ICDCS 2005: Proceedings of the 25th International Conference on Distributed Computing Systems*, pages 187–196. IEEE, 2005.

[16] E. Pinheiro, W.-D. Weber, and L. A. Barroso. Failure trends in a large disk drive population. In *FAST-2007: 5th USENIX Conference on File and Storage Technologies*. USENIX Association, 2007.

[17] J. S. Plank. Erasure codes for storage applications. Tutorial slides, presented at *FAST-2005: 4th Usenix Conference on File and Storage Technologies*, December 2005.

[18] J. S. Plank, A. L. Buchsbaum, R. L. Collins, and M. G. Thomason. Small parity-check erasure codes - exploration and observations. In *DSN-2005: The International Conference on Dependable Systems and Networks*, pages 326–335. IEEE, July 2005.

[19] J. S. Plank and M. G. Thomason. A practical analysis of low-density parity-check erasure codes for wide-area storage applications. In *DSN-2004: The International Conference on Dependable Systems and Networks*, pages 115–124. IEEE, June 2004.

[20] K. Rao, J. L. Hafner, and R. A. Golding. Reliability for networked storage nodes. In *DSN-2006: The International Conference on Dependable Systems and Networks*, pages 237–248. IEEE, June 2006.

[21] B. Schroeder and G. A. Gibson. Disk failures in the real world: What does an MTTF of 1,000,000 hours mean to you? In *FAST-2007: 5th USENIX Conference on File and Storage Technologies*, pages 1–16. USENIX Association, 2007.

[22] A. Thomasian and M. Blaum. Mirrored disk organization reliability analysis. *IEEE Trans. Comput.*, 55(12):1640–1644, 2006.

[23] J. J. Wylie and R. Swaminathan. Determining fault tolerance of XOR-based erasure codes efficiently. In *DSN-2007*, pages 206–215. IEEE, June 2007.

[24] H. Yu, P. B. Gibbons, and S. Nath. Availability of multi-object operations. In *NSDI-2006: Proceedings of the Symposium on Networked Systems Design and Implementation*, May 2006.

Reliability Mechanisms for File Systems Using Non-Volatile Memory as a Metadata Store

Kevin M. Greenan and Ethan L. Miller
Dept. of Computer Science, University of California, Santa Cruz
Santa Cruz, CA, USA
kmgreen@cs.ucsc.edu, elm@cs.ucsc.edu

ABSTRACT

Portable systems such as cell phones and portable media players commonly use non-volatile RAM (NVRAM) to hold all of their data and metadata, and larger systems can store metadata in NVRAM to increase file system performance by reducing synchronization and transfer overhead between disk and memory data structures. Unfortunately, wayward writes from buggy software and random bit flips may result in an unreliable persistent store. We introduce two orthogonal and complementary approaches to reliably storing file system structures in NVRAM. First, we reinforce hardware and operating system memory consistency by employing page-level write protection and error correcting codes. Second, we perform on-line consistency checking of the filesystem structures by replaying logged file system transactions on copied data structures; a structure is consistent if the replayed copy matches its live counterpart. Our experiments show that the protection mechanisms can increase fault tolerance by six orders of magnitude while incurring an acceptable amount of overhead on writes to NVRAM. Since NVRAM is much faster and consumes far less power than disk-based storage, the added overhead of error checking leaves an NVRAM-based system both faster and more reliable than a disk-based system. Additionally, our techniques can be implemented on systems lacking hardware support for memory management, allowing them to be used on low-end and embedded systems without an MMU.

Categories and Subject Descriptors

D.4.3 [**File Systems Management**]: Directory structures; D.4.5 [**Reliability**]: Fault-tolerance

General Terms

Performance, design, reliability

Keywords

non-volatile memory, file system reliability, metadata, online consistency checking, error correcting codes

EMSOFT'06, October 22–25, 2006, Seoul, Korea.

1. INTRODUCTION

Non-volatile byte-accessible RAM is finally becoming a reality, as magnetic RAM (MRAM) is available in quantity and other technologies are maturing. Such technologies provide great opportunities for portable devices to store large quantities of information in a small, power-conserving package. Moreover, storing file system metadata in a non-volatile RAM can significantly reduce the latency of file system metadata operations even in systems that use disks to store their data. Current systems simply use RAM as a cache for block devices such as disks and flash memory; however, caching in this manner has two problems. First, cache contents with a relatively long life must occasionally read from and written back to slower memory, incurring additional latency. Second, all memory resident metadata must be synchronized with the more permanent disk-resident metadata structures, incurring code complexity and performance overheads—metadata designed to be stored on disk is optimized for large block sizes rather than the more natural smaller sizes made possible by in-memory metadata.

Portable devices such as cell phones will likely use new non-volatile memory technologies as the primary store, making higher reliability a critical issue. Even systems that use other storage technologies such as disk and flash memory can store file system metadata in byte-addressable non-volatile memory, providing the performance of in-memory storage with the reliability and data longevity of a disk store. Although there are many non-volatile memory technologies, including flash RAM, magnetic RAM (MRAM) [13], and battery-backed dynamic RAM, our work focuses on MRAM and other similar byte-addressable non-volatile memory technologies; flash memory must be written a block at a time, making it less suitable for the small writes that metadata updates require. We assume that MRAM will be accessed through an interface similar to DRAM and thus, unlike disk and most flash memory, will be directly addressable. Though this interface simplifies data access, it also brings with it the possibility of wild writes and data corruption. Since MRAM contents are never synchronized to disk, the system must ensure that data in MRAM is *always* completely consistent. To achieve this consistency, we propose methods which not only add additional memory-level protection and guarantee consistency, but also provide failure notification.

Our approach consists of two levels: guarantees at the metadata store level, and consistency at the file system level. Metadata store level guarantees are enforced using protection mechanisms such as page protection and check and recovery mechanisms through error correcting codes that protect "blocks" of memory. In addition to page protection [7], we use in-memory error correcting codes (ECC) that guar-

antee correctness up to a prescribed threshold. Beyond that, there is a high probability that an unrecoverable error notification will be sent to the system. Unlike page protection, however, memory-based ECC does *not* require hardware support for memory management, and is thus suitable for portable or low-power embedded devices that lack an MMU.

The file system consistency level ensures that the filesystem metadata is in a consistent state by enforcing atomic writes, logging all operations and periodically checking the filesystem by replaying the log. The consistency checker then compares the structures that result from replay with those that were actually generated by the file system. This is particularly useful when errors are too large for correction via memory-level checks or when the file system itself contains a bug that writes an incorrect value to the metadata and then computes the (correct) ECC for that incorrect value. In most cases, the online consistency checker can localize the error to the individual data structure in which it occurred. Essentially, the memory-level protection mechanisms protect the file system from other processes, while the file system level mechanisms protect the file system from itself by ensuring that the file system metadata is in a consistent state.

We show that the combination of these approaches can dramatically reduce the occurrence of errors in a memory-based file system. The use of orthogonal techniques catches different types of errors, increasing file system reliability. The overhead required for this improved reliability seems high at about 1.5×; however, this is relative to a memory-based file system that is much faster than existing disk-based file systems. The result of implementing our techniques in a file system that permanently stores its metadata is a file system that is both faster and more reliable than disk-based file systems.

2. RELATED WORK

There has been a great deal of previous work on using non-volatile RAM in file systems, and in providing reliable memory-based storage. This section summarizes that work, showing how our research builds on previous work in this area.

2.1 NVRAM in File Systems

Baker, *et al.* [4] observed that the use of NVRAM in a distributed file system can improve write performance and file system reliability. They use NVRAM in conjunction with volatile RAM to form a consistent cache, thus improving reliability and performance in a distributed file system. The goal of the eNVy storage system [21] was to improve the performance and utilization of a flash-based storage system. Instead of using disks for high capacity storage, eNVy used flash memory for persistent storage and SRAM as a non-volatile write buffer.

More recently, HeRMES [12] posited that file system performance would improve dramatically if metadata were stored in MRAM. The HeRMES work also claimed that on-line consistency checking of metadata is a requirement for the metadata store, and that including it may be possible without degrading performance. Conquest [20] also used persistent RAM to store small files, metadata, executables and shared libraries. The distinction between memory regions is similar to that between the protected and non-protected regions used in the NVRAM store of our work. LiFS [1, 2] is a relational link-based file system that stores all of its metadata structures in MRAM. While all of these systems promise higher performance, none of these systems include the combination of page protection via memory protection and online consistency checking that we describe. Thus, they are subject to corruption that cannot be fixed by rebooting because the in-memory metadata is the *only* copy.

2.2 Safe Persistent Memory

The Rio file cache [7, 9] effectively makes a region of memory safe across crashes by turning off write permission bits in the page table, protecting memory from software errors and wild writes during a crash. One aspect of our memory protection scheme is very similar to that of Rio. Unlike Rio, where protected regions enable a safe write *cache* in RAM, we support long-term data storage in MRAM, requiring larger memory space to be consistent over significantly longer durations. A larger distinction is that we augment the write locking mechanism with error correcting codes and data structure integrity checking to guard against software errors in the file system itself.

Write-protected data structures are used in the context of database management systems to limit software error propagation [17]. The authors propose three different update models for write-protected data. This work is similar to ours in that regions of data are protected at the page level. Furthermore, the expose page update model closely resembles the update scheme used for the protected regions in our model.

2.3 File System Consistency

The popular `fsck` program [11] attempts to restore file system consistency by scanning all of the file system metadata. Since the elapsed time of `fsck` is a function of the file system size, this operation often takes a great deal of time to complete and does not scale to very large file systems. McKusick also discusses the use of a background version of `fsck` [10], which is essentially `fsck` running on a snapshot of a file system. Even though background `fsck` can run while changes are made to the file system, it requires a long latency that is not reasonable for our purposes.

File systems such as XFS [18] and LFS [16] use log-based recovery to restore file system consistency. Using such recovery mechanisms lowers the time necessary to perform file system recovery. The on-line consistency checker presented in this paper uses an approach similar to these log-based recovery mechanisms. However, existing log-based recovery is typically run only after a system crash; our system performs recovery continuously, avoiding crashes and data corruption.

2.4 Fault Tolerance

Aumann and Bender [3] propose the addition of redundant links to standard data structures such as linked lists and trees. By adding the redundant links in a butterfly structure, they place bounds on the number of faults the data structure can tolerate. Although such structures would be very helpful for tolerating failures in the underlying metadata structures, we chose to not include fault-tolerant data structures in this work.

The remote file service (RFS) [22] is a proposed framework that can be used by network file systems to speed up repair upon corruption due to security breach or human error. RFS relies on an external resource to determine which processes cause data corruption. Once the external resource flags a process or set of processes as contaminated, RFS uses information in its log to perform backward recovery. RFS recovers the file system with respect to a set of contaminated processes, while our scheme recovers with respect to inconsistent metadata structures. In addition, our scheme does not rely on an external resource for detection of cor-

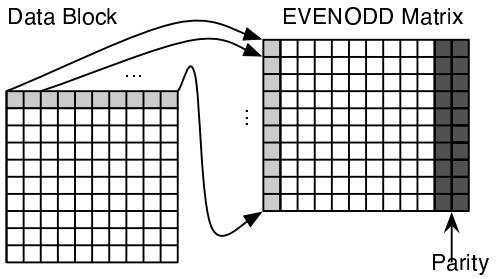


Figure 1: Data block mapping to an EVENODD matrix. Regions of each data block are sequentially copied to each column of the EVENODD matrix. Here, a region of ten symbols in the data block (shaded) is mapped to the EVENODD matrix.

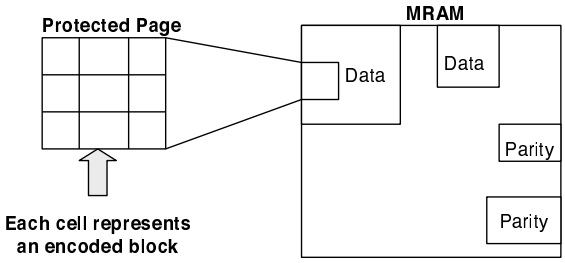


Figure 2: Protected regions in MRAM. Two protected regions—data and parity—are shown here. Processes using the protected regions can only access the data portion, which is organized into pages of encoded blocks. The regions are protected through page-write locks and error correcting codes.

rupt data. Although there is some similarity between RFS and our recovery scheme, the goals of each are somewhat orthogonal.

3. DESIGN

Our approach to protection and consistency consists of two layers, one at the metadata store level and another at the filesystem level. By using two orthogonal techniques to protect the integrity of in-memory data structures, our system ensures that data stored in memory traditionally considered "less-safe" is actually *more* safe than data stored on disk.

3.1 Metadata Store Consistency

As previously discussed, error correcting codes are used as a form of fault tolerance in the metadata store. In our scheme, any encoding algorithm that separates the data and parity symbols may be used for error correction. The mechanisms covered in this paper rely on EVENODD [5, 6] codes for error correction. EVENODD codes are XOR-based, and are thus computationally cheaper than other codes such as Reed-Solomon. Unfortunately, EVENODD codes incur more storage overhead than Reed-Solomon codes.

3.1.1 EVENODD as an ECC

EVENODD codes are typically used for tolerating failures in RAID architectures. Traditionally, the EVENODD scheme requires m data disks and two parity disks. The disks are organized into columns of an $(m-1) \times (m+2)$ matrix, where the first m columns represent data disks and the last two columns represent parity disks; m should be a prime number. Each element in a particular column is a symbol from the corresponding disk. The first parity column holds the horizontal parity of the data columns such that each element, i, of the first parity column is computed by XORing the ith elements from the data columns. The second parity column holds the diagonal parity of the data column. The diagonal parity is computed in a way that ensures any errors in a single column can be detected. Once the disk in error is detected, the horizontal parity can be used to reconstruct the appropriate column of the matrix. The mapping from a block of memory to an EVENODD matrix is given in Figure 1.

Instead of encoding data from disks, we encode blocks of memory into an EVENODD matrix. If we choose $m = 257$, we can encode 256-byte chunks of memory into each of the 257 columns of the matrix, which allows for a burst error of at most 256 bytes. Each EVENODD matrix is constructed by serializing a region of memory and mapping a $m-1$-byte moving window to a column. This moving window is illustrated by rows of the data block in Figure 1. A 2-dimensional parity scheme is essentially equivalent to this EVENODD scheme, since cross-column bursts—errors that span more than one column—cannot be detected. For example, suppose bytes 0 through 255 are mapped to column 1 and bytes 256 through 511 are mapped to column 2 of some matrix. A burst error over bytes 253 through 260 would be a cross-column error burst, and might not be caught.

Even though we did not implement functionality to detect cross-column error bursts, existing schemes can tolerate some cross-column burst errors covering two columns using EVENODD [14]. In the remainder of this section, $E(n, n-m)$ will be used to denote an encoding in which n is the total number of data symbols and m is the number of appended parity symbols.

3.1.2 MRAM Layout

MRAM is a byte-addressable storage technology, which, like many other random access storage media, can be organized into blocks or pages. In this context, pages of blocks will be used as the unit of storage. Figure 2 illustrates the page layout of MRAM in this scheme. Memory is organized into protected and unprotected regions. In order for the protection mechanisms to operate on a region of MRAM, the region must be declared as protected. Figure 2 illustrates the layout of MRAM with protected regions. Each protection region is organized into two sections: data and parity. By making a distinction between the two regions, data can be read at no additional cost. The data section of a protected region is organized into pages of blocks of size $n-m$, where each block is encoded using a $E(n, n-m)$ encoding. The parity section is organized in a similar manner with a block size of m. Placement of the data and parity sections are not necessarily known before creation, although it is important to ensure that these sections are not physically adjacent. Figure 2 shows an example with two protected regions in MRAM. As shown, each region has one data section and a corresponding parity section. Note that the data section of each protected region should be the only portion of the region visible to a process.

3.1.3 Write Protection

An error correcting code is augmented with write protection to protect against corruption in MRAM. We use locking techniques similar to the Rio file cache [7] to protect pages from wild writes. Every page in a protected region is write

```
1: Input : (data, addr, size)
2: (dataSet, paritySet) ← getBlocks(addr, size)
3: for (data_blk, parity_blk) ∈ (data_set, parity_set) do
4:     scratch ← blk_data
5:     write appropriate portion of data to scratch
6:     new_parity ← encode(scratch)
7:     unprotect(data_blk, parity_blk)
8:     data_blk ← scratch
9:     parity_blk ← new_parity
10:    protect(data_blk, parity_blk)
11:    if check(data_blk, parity_blk) ≠ OK then
12:        throw exception
13:    end if
14: end for
```

Figure 3: Write algorithm for protected regions. Line 2 detects all blocks affected by the write. Lines 4–6 copy each block to a scratch region, update the block and re-encode the block, returning the new parity. Each affected block is unprotected, updated and re-protected in lines 7–10. Finally, each block is checked in lines 11–13.

locked until a process is performing a write on a particular page. It is assumed that kernel-level tasks will obey page-level write locks, which guarantees that while a page is locked it will not be subject to wild writes. Protection is strictly enforced using the simple algorithm for writing data to a protected region.

As shown in Figure 3, multiple steps are required to write data out to a protected region of MRAM. MRAM byte address and data size are used to determine the blocks that will be affected by the write. Each affected block is updated to reflect all of the appropriate changes. First, the block is copied to a scratch location outside of its protected region. Next, the appropriate data is written to the copy of the block, which is immediately re-encoded. In order to overwrite the old block encoding with the new encoding, the pages of both the data and parity blocks must be unlocked for writing. The pages are unlocked and the new data and parity are written on top of the old values. The pages are relocked and decoded to ensure no corruption occurred while the pages were unlocked.

There are a few issues with the algorithm as described above. Currently, it is unclear where a block should be copied when performing writes in a protected region. By copying a block to another protected region we may encounter an infinite protection chain, since a copied block would then be subject to the original write policy. It is possible to use a cheaper, less fault-tolerant region for copied blocks. For instance, scratch regions using checksums could be used for this purpose. Currently, no restrictions are placed on the location of a block copy, as long as the scratch regions are placed outside of the respective protected region.

In addition to checking the integrity of blocks during the write algorithm, periodic integrity checks can be performed on groups of blocks. The checks can simply check the integrity of a random group of blocks or contain more complicated functionality such as considering a group of blocks that have passed a threshold (*i. e.*, not checked in a long time). Currently, we simply employ a "sweeping check" that performs an integrity check on all of the blocks within a particular address interval.

3.2 Filesystem-level Consistency

So far, we have described protection on the metadata-store level using error correcting codes and page-level write protection. However, these techniques cannot identify errors

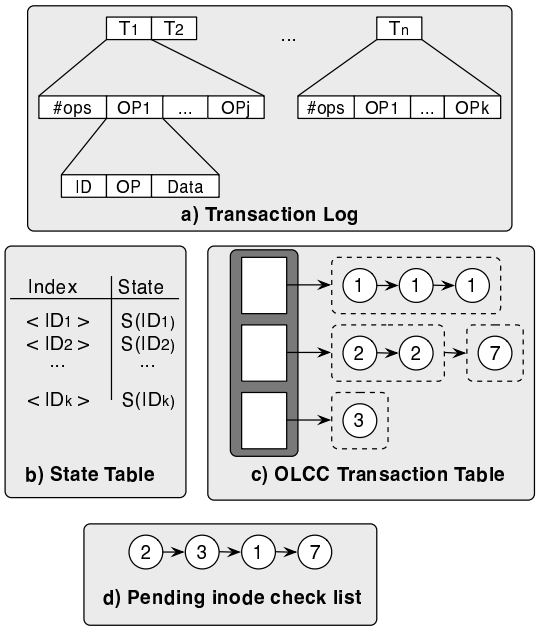


Figure 4: Data structures for logging and file system consistency checking. a) The log is a FIFO of transactions, which contain a series of metadata operations. b) The state table holds the initial state of logged inodes. c) The OLCC transaction table clusters inode changes. In this figure, the changes for inodes 1, 2, 3 and 7 are clustered. d) The pending inode list is the set of updated inodes that require a replay and check.

caused by file system bugs that correctly update both the data and parity in the metadata store but result in incorrect file system data structures. For example, establishing a new hard link to an inode might fail to increment the reference count; this error would not be flagged by page-level protection. The filesystem mechanisms take care of rolling back if MRAM writes fail and periodically checking the log against the actual metadata to ensure the integrity of the file system data structures. The consistency checker could be written by a separate design team using the file system specification; this approach would increase the likelihood that bugs in either the file system or the checker would be caught. By including these online consistency checks, recovery time is reduced and faults in the metadata can be caught before they are propagated.

3.2.1 *Additional Structures*

In order to maintain integrity on the filesystem level, a few structures must be created. First, all of the file system metadata transactions are logged. This log may be placed in a protected region of memory to ensure that it is not modified accidentally. Depending on the overhead incurred by the encoding, however, this protection may simply be in the form of write protection. Figure 4 contains the structures required to maintain filesystem metadata integrity.

All of the metadata operations for each file system call are batched and placed into a transaction. Each transaction contains a series of operations, and each operation consists of an ID, an operation identifier and a data field. An ID field is used to uniquely identify the structure that the corresponding operation is changing (*i. e.*, inode number). The

last two fields contain the operation identifier and the data associated with the operation (*i. e.*, link with $\langle src, dest \rangle$). Each transaction contains the number of operations necessary for the transaction and may contain one to many of these operation structures. The transaction log is a FIFO log of all of these transactions. The structure of the transaction log is illustrated in Figure 4a.

Three more structures are required for online consistency checking. The first structure, shown in Figure 4b, is a state table that holds the state of all structures that have changed since the last consistency check. The state of each structure is indexed by inode number. Each time a transaction requires a change to an inode, the table is checked. If an entry exists in the state table for that particular inode, we do nothing and continue writing to the log. If the inode is not in the state table, then an entry containing the live-state of the inode (*i. e.*, size, permissions, etc.) is created. The state table has two purposes: it holds initial state for roll-forward within the consistency checker and can be used by the consistency checker to determine if a particular inode has been updated since the checker was invoked.

In addition to the state table, the consistency checker maintains a persistent hash table of operations, the online consistency checker (OLCC) transaction table, as shown in Figure 4c. The table contains transaction operations indexed by inode number, with each bucket in the table consisting of a cluster of operation entries for an inode. These clusters are temporally ordered because inserted entries are traversed and removed in FIFO order from the log. The consistency checker uses the operation entries in a cluster to replay the updates of a single inode. A pending inode list is used to determine which inodes require a consistency check and is illustrated in Figure 4d. The list simply contains an inode number per entry, and is temporally ordered.

3.2.2 Online Consistency Checker

All of the structures described above are used by the online consistency checker, as this section describes. Immediately before the consistency checker is started, a new location is allocated for the log and state table, so future transactions will be written to newly initialized structures. The old log and both old and new state table are used by the consistency checker.

The consistency checker first traverses the log in FIFO order and inserts each operation associated with every transaction into the transaction table. After updating the transaction table, the consistency checker traverses the pending inode list and individually processes each inode cluster in the transaction table. Before processing a cluster of operations on an inode, the consistency checker must check the current state table to see if any updates have been made to the corresponding inode. If an entry is in the live state table, the check is deferred until the next consistency check and the consistency checker moves on to the next cluster. If no entry exists in the live state table for the corresponding inode, then a lock is placed on the structure and all of the operations within the cluster are replayed on the state taken from the old state table entry for the inode. Once all of the operations within a cluster are replayed, they are removed from the transaction table and the replayed inode is compared to the live inode. If the two states differ, then a consistency problem exists and a notification is generated. Any clusters remaining in the list after the checker completes will be processed on the next iteration of the consistency checker. Both the old state table and old log are freed from memory once the consistency checker completes. The basic algorithm for the consistency checker is shown in Figure 5. We assume the recovery of a corrupted inode will simply involve overwriting the live inode with the replayed inode.

```
1:  old_log ← live_log
2:  live_log ← init_new_log()
3:  old_st_tbl ← live_st_tbl
4:  live_st_tbl ← init_new_st_tbl()
5:  ⟨olcc_tbl, pend_list⟩ ← insert_into_olcc_tbl(old_log)
6:  for inode_num ∈ pend_list do
7:      inode ← st_tbl_lookup(old_st_tbl, inode_num)
8:      ⟨op[], data[]⟩ ← get_ops(olcc_tbl, inode_num)
9:      inode ← replay(inode, op[], data[])
10:     live_inode ← get_live_inode(inode_num)
11:     if compare(inode, live_inode) ≠ OK then
12:         throw exception
13:     end if
14: end for
```

Figure 5: Basic algorithm for the online consistency checker. A new log and state table are created in lines 1–4. The contents of the old log are added to the OLCC transaction table in line 5. Lines 6–14 describe the operations performed for each inode in the pending inode list. Note that line 8 fetches all of the changes for a particular inode. For brevity, locking and state table checks are left out of the pseudocode.

4. PROTOTYPE IMPLEMENTATION

The MRAM protection and file system consistency mechanisms were independently written and tested. Due to the fact that large quantities of MRAM are currently unavailable (current chips hold only 4 Mbits), the MRAM-level mechanisms were incorporated into an MRAM simulator, which was implemented as a user space MRAM allocator.

The MRAM allocator uses the `malloc` call to allocate a large region of DRAM. Once the MRAM allocator obtains a region of DRAM and initializes its internal structures, the offset and size of the simulated MRAM are passed to the protection module, which initializes the data and parity sections of MRAM. Page-level protection comes in the form of `mprotect` system calls. All encoding and decoding is done using a simple EVENODD library created for the experiments in the next section. The library does not include write optimizations [5] and cross-column error tolerance [14]. Since cross-column error tolerance is not included in the EVENODD library, 1-byte errors were used for fault injection.

Data access within the MRAM simulator requires two calls: `toMRAMPointer` and `toNormalPointer`. MRAM pointers represent addresses relative to their position within the simulated MRAM, which range from 0 to the size of the region. Normal pointers simply represent the actual address of the heap allocated memory, which the allocator obtains using `malloc`. The `toMRAMPointer` and `toNormalPointer` calls enable programs using the MRAM allocator to map MRAM relative addresses to their virtual addresses for manipulation and vice versa.

Mechanisms for protected regions and logging were incorporated into an experimental file system called LiFS [1]. The consistency checking code currently resides outside of LiFS, but runs against a log generated by LiFS. LiFS is implemented in user space using the FUSE kernel module and supporting libraries [8], extending file system metadata to handle relational links between objects in a file system. All of the metadata in LiFS is stored in a persistent MRAM store.

In the current implementation, the log and its supporting structures are stored in the protected region of MRAM along with the file system metadata. Thus, all metadata writes, log appends and state table insertions involve the write algorithm described in Section 3. This not only increases the complexity of the code, but it also results in slightly more expensive file system operations. The latency of write operations could be decreased by subjecting log appends to write protection without encoding. This choice may be acceptable due to the relatively short life of the log compared to the file system and the effectiveness of write protection.

Logging was added by creating transactions in calls that result in structural change to the metadata; this approach is similar to that used by other logging file systems such as `ext3` [19]. A transaction is created at the beginning of the system call, all metadata update operations are added to the transaction in the body of the call and the transaction is finally added to the log before the call returns. All metadata operations with the exception of extended attributes and extents are being captured by the logging code.

The consistency checker currently resides outside of the file system, but can still be used to verify the logged metadata transactions against live metadata by calling it with the location of the log. In order to decrease the latency of the consistency checker, it runs in an unprotected region.

In the future, the persistent information held by the consistency checker would reside in a write protected area. This information will be copied out to a scratch region every time the checker runs and will be written back out to the protected area once the checker completes. This is done since most of the consistency checker's structures do not have a long life; thus there is no need to incur protection overhead. As stated, protection will be used on the unprocessed structures.

5. PROTOTYPE PERFORMANCE

Four metrics are necessary in order to effectively analyze the performance of the reliability mechanisms presented in the previous sections: fault tolerance, raw MRAM write performance, file system performance on a workload focused on metadata operations, and consistency checker performance. We first measured fault tolerance by injecting faults while running a workload against the file system. Next, we ran a metadata-centric workload against various configurations of LiFS, which is used to determine the overhead introduced by logging, write protection and encoding. A breakdown of the protected region write overhead is presented to give the reader an idea of how these mechanisms would perform in the kernel. Finally, we analyzed the properties of the file system consistency checker.

It is important to note that LiFS is currently running in user space; thus, the evaluation in terms of running time or throughput should not be compared to any kernel-based file system. In order to get a clean comparison, LiFS is essentially compared to itself throughout.

5.1 Experimental Setup

The prototype was implemented and evaluated on a Sun workstation running the Linux kernel version 2.6.9-ac11. The workstation was configured with an AMD Opteron150 processor running at 2400 MHz with 1 GB of RAM. The protected regions were protected with EVENODD codes with either 16 or 8 columns with a size of 8 or 16 byte-length symbols, respectively. Thus, each 96-byte encoded block has a 64-byte data section (EVENODD(96,64)) and each 288-byte encoded block has a 256-byte data section (EVENODD(288,256)). A 200 MB protected region was created for each experiment.

The current EVENODD library is based on [5, 6], which can tolerate up to $m-1$-byte error bursts that do not occur in more than one EVENODD column. Cross-column optimization [14] only requires additional decoding complexity and should not affect running time or throughput in the general case. In order to avoid the problems associated with cross-column error bursts, we used 1-byte error injections to test the efficacy of the MRAM-level mechanisms. A modified decoder could tolerate much larger bursts.

5.2 Error Injection

Software faults can be simulated by spawning threads that continuously attempt invalid writes to the protected region. If one of the threads attempts a write to a protected page, a segmentation fault is raised. A signal handler is in place to catch the fault and increment a counter. This aggressive injection estimates the benefit of the protection mechanisms when subjected to a large number of "wild writes"—writes to incorrect locations.

In our first scenario faults were randomly injected into the entire 200 MB protected MRAM region, while a process simultaneously performed a workload of 250,000 valid 16-byte writes. After 10 runs with this scenario, only 4 of the targeted injections did not result in a segmentation fault (*i. e.*, was not caught by page protection). This result confirms the validity of the Rio file cache [7], while also raising a red flag: these 4 writes could corrupt an entire file system. The purpose of the ECC is to prevent the wild writes that make it through the Rio protection from corrupting data stored in a protected region. Furthermore, it should be noted that our analysis produced accelerated faults in the protected region. This approach was necessary to show that although page protection alone results in a great deal of fault tolerance, a few mechanical errors are likely to slip through over time.

In order to analyze the more vulnerable points in the protected write algorithm, the test above was repeated on a much smaller, targeted region. The aforementioned injection scenario was rerun on a 160,000 byte region, with the active writes directed to this region. Figure 6 shows the results of this targeted attack. Roughly 10,000 injected writes were attempted in all three trials shown in Figure 6. Each run is divided into three categories: errors not detected, errors caught by page protection and errors caught by error correcting codes.

As shown in the graph, 1.5–2.2% of the erroneous writes bypassed page protection. Of the writes bypassing page protection, over 90% were caught by EVENODD encoding. The errors were either caught during the write algorithm or by the "sweeping check" described in Section 3 that was run after the workload completed.

There are two explanations for the cases in which errors were undetected. First, we found that the `mprotect` system call incurred a great deal of overhead in terms of latency. Some of these uncaught injections may have occurred between the call to `unprotect` and the data copy from the scratch region to the home location in protected memory. Thus, the error was injected, but was quickly overwritten by the correct data from the scratch region. We also found that due to the granularity of the page protection mechanisms, injection may occur in regions that have not been subject to writes. In this case the error would not be checked until a process performs a valid protected write or until the background checker is run.

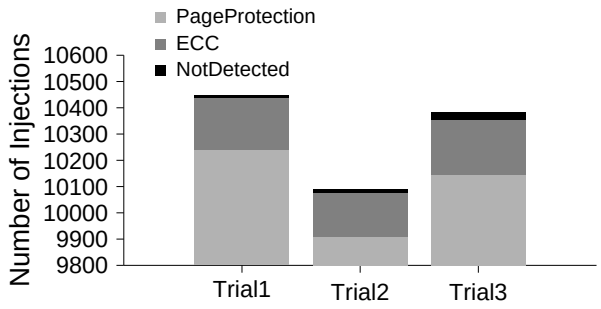


Figure 6: Effectiveness of MRAM-level protection with raw writes. Most of the errors are caught by page protection, while a majority of those that escape page protection are caught by the error correcting codes. Note that the y axis starts at 9000 injections.

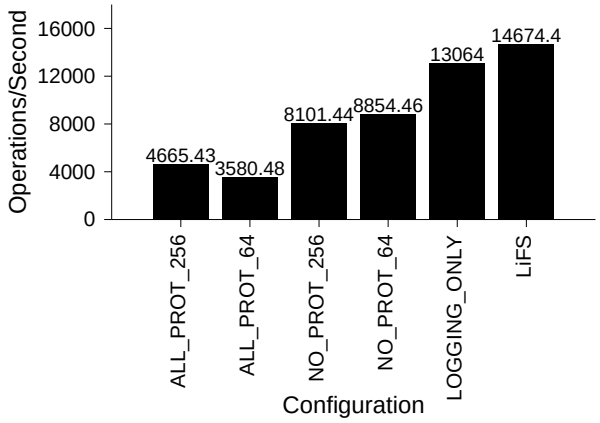


Figure 7: Throughput of various LiFS configurations with a metadata-centric workload, with values shown above each bar.

We have shown that page protection used in conjunction with error correcting codes can improve fault tolerance by roughly six orders of magnitude over a scheme using no protection mechanisms. Page-level write protection accounts for a majority of the fault tolerance, with the error correcting codes protecting persistent data in MRAM when errors get through page-protection.

5.3 Metadata-centric File System Workload Performance

Since we are mainly concerned with the effect of subjecting LiFS to a large amount of metadata change, we created a simple workload with a great deal of metadata writes. This workload first creates 100 directories, writes 100–500 zero byte files to each directory, creates one link per file, changes the permissions of the directories and finally touches all of the files. The goal was to generate a large number of metadata changes and measure the latency with various configurations of protection within LiFS, without relying on any of the bottlenecks that currently exist in LiFS, such as extent allocation.

Figure 7 shows the average throughput in operations per second of six different variations of the workload on five configurations of LiFS. The ALL_PROT configurations, represent LiFS configured with page protection using EVENODD encoding with 64 or 256 bytes of data per code and logging. The NO_MPROT configurations are similar to the ALL_PROT configurations, but lack page protection with the `mprotect` system call. The last two experiments were run with logging only and without any of the protection mechanisms.

Turning on all protections generally resulted in a 3–4× latency overhead relative to stand-alone LiFS, as Figure 7 shows. At first this result seems high, especially since EVENODD encoding requires a constant number of XOR operations per `encode`/`decode` call. However, the second set of experiments run without hardware page protection show the root cause of the slowdown: overhead associated with the mprotect system call. We believe most of the `mprotect` overhead is due to frequent context switches. By combining the data given in Figure 7 with performance estimates for page protection reported from Rio [7], we expect that all of the protection mechanisms will incur less than a 2× overhead when incorporated into a kernel-based file system.

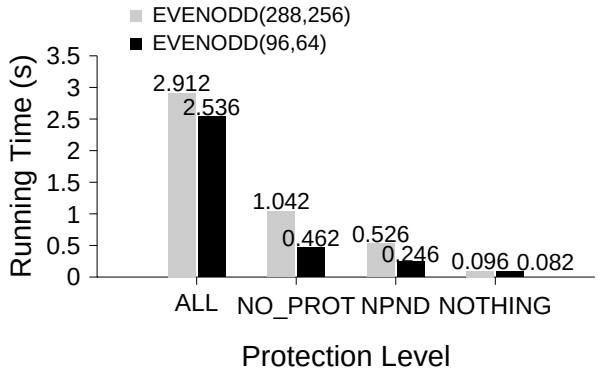


Figure 8: This figure shows the time required to do raw writes to protected regions of MRAM using both 256 byte and 64 byte data regions. The ALL scheme is a fully protected region, while NO_PROT is a region without page protection and NPND is a region with nothing but data encodes. NOTHING is a region with no page protection or data encoding.

5.4 Breakdown of Write Overhead

In order to effectively analyze the overhead associated with our protection mechanisms, we performed 250,000 16–byte writes on four MRAM configurations: one with all protection mechanisms, one that uses encoding and decoding but not `mprotect`, one with no protection and no decoding (but with encoding), and one with no protection mechanisms. We omitted logging from these experiments because, as shown in Figure 7, logging does not incur a significant amount of overhead compared to page protection and error correcting codes.

The average running time over 10 runs of the four scenarios is given in Figure 8. Each scenario was run with a 256-byte and a 64-byte block configuration. This figure shows that the `mprotect` system call accounts for most of the write overhead. Encoding and decoding incurs a 5–10× overhead over writes without any protection mechanisms. Encoding alone causes a factor of three overhead over no encoding using 256 byte blocks and a factor of five for 64 byte blocks.

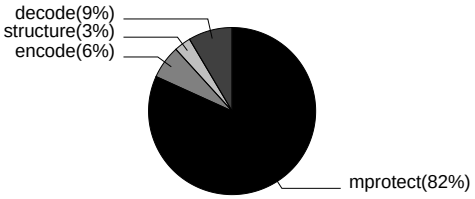


Figure 9: Breakdown of raw write overhead for EVENODD on 64 byte blocks.

Finally, we show the breakdown of writes, which is based on the data given in Figure 8. Figure 9 illustrates the breakdown of write overhead with respect to protection, encoding/decoding and organizational structures. Again, this figure shows that most of the write overhead is due to the `mprotect` calls. Given a kernel implementation, the protection overhead is expected to decrease such that encoding/decoding accounts for most of the overhead. As stated earlier, such an implementation should result in 2–3× overhead, instead of 3–4× overhead on metadata-centric workloads.

5.5 FS Integrity Check Performance

5.5.1 *Verifying FS Integrity*

In order to analyze the validity of the file system consistency checker errors were injected into MRAM while running a file system workload. Errors were injected by choosing a random inode, obtaining the address of the inode in MRAM and writing 1–8 bytes to a random address within the inode. The injections were performed by a separate thread of execution and written to a log for later comparison; there were 100 injections done for each experiment. The on-line consistency checker was run after each file system workload terminated, writing any detected inconsistencies to a log. The contents of the OLCC log and the injection log were compared to determine whether the OLCC caught all of the errors.

A total of five workloads were used in the validity test; the first workload created 10,000 files and every subsequent workload created 10,000 more files. We ran each workload three times with the injector turned on. All 1500 injected errors were correctly detected by the consistency checker, with no false positives.

5.5.2 *Running Time*

We constructed a log with the file system workload from Section 5.3 to test the running time of the consistency checker. As stated earlier, the consistency checker does not currently reside in LiFS, but can access the log. The consistency checker builds its structures outside of the protected region and pulls the old log in from the protected region and compares the state of every live inode to the replayed inode from the state table. This operation is expected to be relatively fast, since all of the inode changes are clustered in a hash table.

The on-line consistency checker was run against a series of logs constructed from workloads similar to the metadata-centric workloads. Figure 10 shows the average elapsed time required to run the consistency checker against all of the operations in the log. The number of operations loaded into the transaction table varies from about 200,000 to 2 million. As shown in the figure, the consistency checker takes a relatively small amount of time to run. Thus, a time-dependent

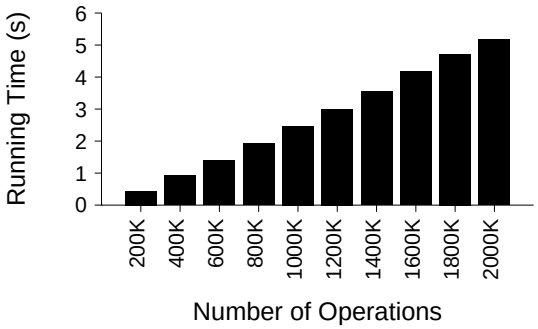


Figure 10: Consistency checker performance with various log sizes. The logs were generated using the metadata-centric file system workload. The number of directories is held constant at 100 and the number of files per directory starts at 100 and increases by 100 for each bar.

policy for consistency checking would probably suffice, since roughly 2 million metadata operations can be replayed and checked in about 5 seconds. We expect that the consistency checker's structures will be stored in protected MRAM and copied out when needed, which should not add much to the elapsed time of the consistency checker or logging routines.

6. FUTURE WORK

Our first goal is to fully incorporate the consistency checker into LiFS, increasing the cohesion between the MRAM-level protection mechanisms and the consistency checker, and logging extents and extended attributes. By allowing the consistency checker and lower-level mechanisms to communicate, errors beyond the threshold limit of the encoding may be fixed. Extents were not immediately incorporated due to a new allocator, which was created in parallel with this work.

As mentioned in the performance section, we need to confirm our intuition with respect to the few errors not caught by `mprotect` getting through our mechanisms. We would also like to experiment with other encoding schemes. EVENODD is faster, but there are other encoding schemes that have better storage efficiency.

Finally, we wish to incorporate all of these mechanisms and LiFS itself into the Linux kernel. Doing so would make a highly reliable, very fast in-memory file system available for Linux. Unlike existing in-memory file systems, LiFS is optimized for memory-style access and is thus simpler and significantly more space-efficient than the traditional approach of using disk-based file systems on a RAM disk. Once the system is available for Linux, we hope to make it available for embedded devices as well.

7. CONCLUSIONS

In this paper, we have presented three orthogonal mechanisms to ensure reliability in file systems that use non-volatile byte-addressable RAM for long-term data storage. We showed that page protection blocks almost all of the invalid, targeted writes from processes outside the file system; however, it cannot handle bugs in the file system itself that involve well-intentioned, but incorrect, writes to critical data structures. Moreover, it does not catch *all* mistargeted writes. We showed that the use of error correcting codes on a block basis can detect and correct errors that page

protection fails to block, further increasing reliability. We also showed that the use of a replay log processed by an independently-written online consistency checker can detect errors introduced into the file system, further improving reliability.

We have also shown that these protection mechanisms do not excessively degrade performance. On our workloads, which are nearly exclusively writes, performance is reduced by at most a factor of 3–4, much of which is caused by the low performance of the kernel-based page protection proposed in earlier work. By moving the entire file system into the kernel, we estimate the performance of our write-intensive workload to be about half the speed of an unprotected workload. Since reads incur no additional overhead and many metadata operations are reads [15], the actual effect on a real workload would be a 1.5× slowdown over an unprotected memory-based file system.

Finally, we have shown that we can periodically check file system metadata integrity at a low cost. The file system level checks are in place to maintain the consistency of the file system structures. Due to the low cost of running the consistency checker, we can ensure the file system is in a consistent state at all times. This check is very fast, and might be able to replace page protection if hardware page protection is difficult or slow, perhaps due to implementation issues in the TLB. Additionally, neither the online consistency checker nor the ECC techniques make use of an MMU, making them potentially useful in portable or low-power devices with less complex CPUs that lack MMU hardware.

By combining the existing technique of page protection with page-based error-correcting codes and log replaying, our techniques for ensuring file system integrity in non-volatile memory reduce the number of errors by more than six orders of magnitude over an unprotected file system in memory. The use of log replaying allows our design to constantly check the file system to ensure its integrity and guard against software errors in the file system, while the combination of page protection and error correction ensure that errors elsewhere in the operating system do not corrupt memory-based file system information. Using these techniques, designers can build in-memory structures that need never be flushed to disk without worrying that they will be corrupted over time. By doing this, designers can keep metadata structures in non-volatile RAM without fear of corruption. These techniques also facilitate the design of file systems for low-power portable devices that lack disks by protecting in-memory data structures without the need for hardware page protection. Using these techniques, memory-based file systems can be as reliable as, if not more reliable than, traditional disk-based file systems.

8. ACKNOWLEDGMENTS

We would like to thank the faculty and students in the Storage Systems Research Center, particularly Nikhil Bobb and Mark Storer, for their help and comments. This research was funded in part by National Science Foundation grant 0306650. Additional funding for the Storage Systems Research Center was provided by support from Hewlett Packard Laboratories, Hitachi Global Storage Technologies, IBM Research, Intel, Microsoft Research, Network Appliance, Rocksoft, Veritas, and Yahoo.

9. REFERENCES

[1] Ames, A., Bobb, N., Brandt, S. A., Hiatt, A., Maltzahn, C., Miller, E. L., Neeman, A., and Tuteja, D. Richer file system metadata using links and attributes. In *Proceedings of the 22nd IEEE / 13th NASA Goddard Conference on Mass Storage Systems and Technologies* (Monterey, CA, Apr. 2005).

[2] Ames, S., Bobb, N., Greenan, K. M., Hofmann, O. S., Storer, M. W., Maltzahn, C., Miller, E. L., and Brandt, S. A. LiFS: An attribute-rich file system for storage class memories. IEEE.

[3] Aumann, Y., and Bender, M. A. Fault tolerant data structures. In *Proceedings of the 37th Annual Symposium on Foundations of Computer Science* (Oct. 1996), IEEE, pp. 580–591.

[4] Baker, M., Asami, S., Deprit, E., Ousterhout, J., and Seltzer, M. Non-volatile memory for fast, reliable file systems. In *ASPLOS '92* (Oct. 1992), ACM, pp. 10–22.

[5] Blaum, M., Brady, J., Bruck, J., and Menon, J. EVENODD: An efficient scheme for tolerating double disk failures in RAID architectures. *IEEE Transactions on Computers 44*, 2 (1995), 192–202.

[6] Buyya, R., and Cortes, T., Eds. *High Performance Mass Storage and Parallel I/O: Technologies and Applications*. John Wiley & Sons, Inc., New York, NY, USA, 2001.

[7] Chen, P. M., Ng, W. T., Chandra, S., Aycock, C., Rajamani, G., and Lowell, D. The Rio file cache: Surviving operating system crashes. In *ASPLOS '96* (Oct. 1996), pp. 74–83.

[8] FUSE. http://fuse.sourceforge.net/.

[9] Lowell, D. E., and Chen, P. M. Free transactions with Rio Vista. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)* (Dec. 1997), pp. 92–101.

[10] McKusick, M. K. Running fsck in the Background. In *Proceedings of the BSDCon* (2002), pp. 55–64.

[11] McKusick, M. K., and Kowalski, T. *4.4 BSD System Manager's Manual*. O'Reilley and Associates, Inc., Sebastopol, CA, 1994, ch. 3, pp. 3:1–3:21.

[12] Miller, E. L., Brandt, S. A., and Long, D. D. E. HeRMES: High-performance reliable MRAM-enabled storage. In *Proceedings of the 8th IEEE Workshop on Hot Topics in Operating Systems (HotOS-VIII)* (Schloss Elmau, Germany, May 2001), pp. 83–87.

[13] Nahas, J., Andre, T., Subramanian, C., Garni, B., Lin, H., Omair, A., and Martino, W. A 4Mb 0.18um 1T1MTJ 'toggle' MRAM memory. In *IEEE International Solid-State Circuits Conference* (Feb. 2004).

[14] Raphaeli, D. The burst error correcting capabilities of a simple array code. *ACM Transactions on Internet Technology 51*, 2 (2005), 722–728.

[15] Roselli, D., Lorch, J., and Anderson, T. A comparison of file system workloads. In *Proceedings of the 2000 USENIX Annual Technical Conference* (June 2000), pp. 41–54.

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

[17] Sullivan, M., and Stonebraker, M. Using write protected data structures to improve software fault tolerance in highly available database management systems. In *Proceedings of the 17th Conference on Very Large Databases (VLDB)* (Barcelona, Spain, 1991).

[18] Sweeney, A., Doucette, D., Hu, W., Anderson, C., Nishimoto, M., and Peck, G. Scalability in the XFS file system. In *Proceedings of the 1996 USENIX Annual Technical Conference* (Jan. 1996), pp. 1–14.

[19] Tweedie, S. EXT3, journaling file system, July 2000.

[20] Wang, A.-I. A., Kuenning, G. H., Reiher, P., and Popek, G. J. Conquest: Better performance through a disk/persistent-RAM hybrid file system. In *Proceedings of the 2002 USENIX Annual Technical Conference* (Monterey, CA, June 2002).

[21] Wu, M., and Zwaenepoel, W. eNVy: a non-volatile, main memory storage system. In *ASPLOS '94* (Oct. 1994), ACM, pp. 86–97.

[22] Zhu, N., and Chiueh, T.-C. Design, implementation, and evaluation of repairable file service. In *Proceedings of the 2003 International Conference on Dependable Systems and Networking (DSN 2003)* (2003), pp. 217–226.

PRIMS: Making NVRAM Suitable for Extremely Reliable Storage †

Kevin M. Greenan
kmgreen@cs.ucsc.edu

Ethan L. Miller
elm@cs.ucsc.edu

Storage Systems Research Center
University of California, Santa Cruz

Abstract

Non-volatile byte addressable memories are becoming more common, and are increasingly used for critical data that must not be lost. However, existing NVRAM-based file systems do not include features that guard against file system corruption or NVRAM corruption. Furthermore, most file systems check consistency only after the system has already crashed. We are designing PRIMS to address these problems by providing file storage that can survive multiple errors in NVRAM, whether caused by errant operating system writes or by media corruption. PRIMS uses an erasure-encoded log structure to store persistent metadata, making it possible to periodically verify the correctness of file system operations while achieving throughput rates of an order of magnitude higher than page-protection during small writes. It also checks integrity on every operation and performs on-line scans of the entire NVRAM to ensure that the file system is consistent. If errors are found, PRIMS can correct them using file system logs and extensive error correction information. While PRIMS is designed for reliability, we expect it to have excellent performance, thanks to the ability to do word-aligned reads and writes in NVRAM.

1 Introduction

Byte-addressable, non-volatile memory (NVRAM) technologies such as magnetoresistive random access memory (MRAM) and phase-change memory (PRAM) have recently emerged as viable competitors to Flash RAM [1, 2]. These relatively low capacity technologies are perfect for permanent metadata storage, and can greatly improve file system performance, reliability and power consumption. Unfortunately, due to the increased chance of data corruption, storing permanent structures in NVRAM is generally regarded as unsafe, particularly when compared to disk. The simplicity of most memory access interfaces makes erroneous writes more likely, resulting in data corruption—it is far easier to manipulate structures in memory than on disk. Such behavior is common in OS kernels, in which buggy code can issue erroneous *wild writes* that accidentally overwrite memory used by another module or application.

The goal of PRIMS (Persistent, Reliable In-Memory Storage) is to provide reliable storage in NVRAM without hindering the access speed of byte-addressable memory. Given the limitations of current in-memory reliability mechanisms, we believe that a log-based scheme using software erasure codes is the most effective way to ensure the consistency of persistent, memory-resident data. We present a log-based approach that has the ability to detect and correct errors at multiple byte-granularity without using page-based access control or specialized hardware support. PRIMS consists of a single, erasure-encoded log structure that is used to detect and correct hardware errors, software errors and file system inconsistencies.

2 Motivation

Modern operating systems protect critical regions of memory using access control bits in the paging structures. While page-level access control is an effective tool for preventing wild writes in write caches, it is not the best solution for protecting small, persistent structures in byte-addressable memory because every protected write requires a TLB flush and two structure modifications to mark a page as read-write and read-only. During periods of frequent small writes, these permission changes have a dramatic effect on performance.

Disk interfaces also decrease the likelihood of wild writes by requiring access through device drivers containing complex I/O routines. The probability of rogue code accidentally corrupting disk blocks while evading the controlled device driver interface is extremely low. This strict I/O interface greatly improves data reliability with respect to software errors, but hinders performance on low-latency media, such as NVRAM.

In addition to software errors, hardware errors such as random bit flips and cell wear may occur on the media, leading to data corruption. Hardware-based error correction schemes require a specialized controller and resolve errors beyond the correction capability by rebooting the system. Obviously, rebooting is not an option when protecting persistent data in memory; thus, a more robust scheme is necessary. Hardware-based error correction is also computed independent of any software implementation; as a result, wild

†This research was funded in part by NSF-0306650, the Dept. of Energy-funded Petascale Data Storage Institute, and by SSRC industrial partners.

writes will most usually result in values that have consistent hardware-based ECC values. In order to provide a high level of flexibility and resiliency, error correction should be moved into a software module that provides tunable redundancy with respect to the target environment.

As the size of storage systems increases, the probability of corruption and time required to make repairs increase correspondingly. In an effort to prevent permanent damage as a result of corruption and minimize system downtime, PRIMS performs on-line checks that verify the consistency of file system structures and all NVRAM-resident data. We believe such integrity checking should be a requirement for modern file systems.

3 Related Work

A great deal of work has gone into many of the performance implications associated with the use of NVRAM in file systems. In addition, methods to achieve reliability in file systems and data caches have been used for a great deal of time. As we will see, no single method has the ability to effectively handle both software and hardware errors in NVRAM.

Baker, *et al*. [3] observed that the use of NVRAM in a distributed file system can improve write performance and file system reliability. More recently, HeRMES [10] posited that file system performance would improve dramatically if metadata were stored in MRAM. Conquest [16] also used persistent RAM to store small files, metadata, executables and shared libraries. While these systems promise higher performance, none of the systems provided improved reliability. As a result, they are potentially unsafe to use for long-term metadata storage because they are subject to corruption that cannot be fixed by rebooting—the in-memory metadata is the *only* copy.

The Rio file cache [4, 7] utilizes page permission bits to set the access rights for areas of memory, providing a safe non-volatile write cache. In general, page protection does not have a profound effect on write throughput; page protection does heavily degrade write performance in workloads consisting of small I/Os. Since most metadata updates are relatively small, we believe that page protection should not be used to protect persistent metadata.

Earlier work on metadata reliability in NVRAM used a combination of error correcting codes and page-protection via the `mprotect` system call to ensure the integrity of persistent, memory-resident metadata [5]. The cost of page protection significantly hurt performance, while on-line consistency checks of memory-resident metadata proved to be inexpensive.

Mondriaan memory protection [17] provides fine-grained isolation and protection between multiple domains within a single address space. SafeMem [12] deterministically scrambles data in ECC memory to provide protected regions of memory, providing `mprotect`-like protection. After scrambling the data, ECC is re-enabled and the ECC controller faults when the region is accessed. While these approaches provide an interesting alternative to `mprotect`, both require specialized hardware support and prevent data corruption without correcting the corruption when it occurs. Our approach assumes that error *recovery* is just as important is error *detection* in ensuring the reliability of persistent data in NVRAM.

The popular `fsck` program [9] and its background variant [8] attempt to restore file system consistency by scanning all of the file system metadata. Since the running time of `fsck` is a function of the file system size, this operation often takes a great deal of time to complete and does not scale to very large file systems. Chunkfs [6] divides on-disk data into individually repairable chunks promising faster `fsck` and partial on-line consistency checks. These techniques are more similar to ours, and Henson, *et al*. explore tradeoffs between performance and reliability in an effort to speed up the file system repair process.

DualFS [11] physically separates data and metadata onto different devices or different partitions on the same device. The metadata device is treated as a log-based file system and the data device resembles an FFS file system. The separation of metadata and data results in fast crash recovery, simplified log cleaning and improved file system performance. File systems such as XFS [15] and LFS [13] also use log-based recovery to restore file system consistency. Unfortunately, these systems only apply recovery mechanisms after a crash, which may lead to system downtime or data loss. In addition, they do not proactively check file system integrity while the file system is running, potentially leading to latent errors that may go undetected for days or weeks.

4 Design

The performance implications and reliability constraints associated with current memory protection schemes force us to take a new approach when providing reliability for persistent data in memory. The overall structure of PRIMS is based on five key design requirements:

Place metadata in NVRAM and data on disk Moving metadata from disk to NVRAM allows PRIMS to exploit parallelism between the independent devices, enabling better performance, simple structures and the ability to do fast, on-line metadata checks.

Periodically scan file system structures in NVRAM On-line consistency checks are expected to decrease or prevent down-time in the face of media or software failures.

Avoid the need for hardware support An MMU is not necessary and ECC schemes and reliability levels can be changed.

Allow recovery from software and hardware faults Since we plan to store persistent data on easily accessible, low-reliability media, consistency must be maintained by protecting against both software and hardware faults.

Provide reliability with minimal overhead Persistent data is stored in NVRAM in an effort to improve performance, thus any reliability mechanisms should have a small impact on performance.

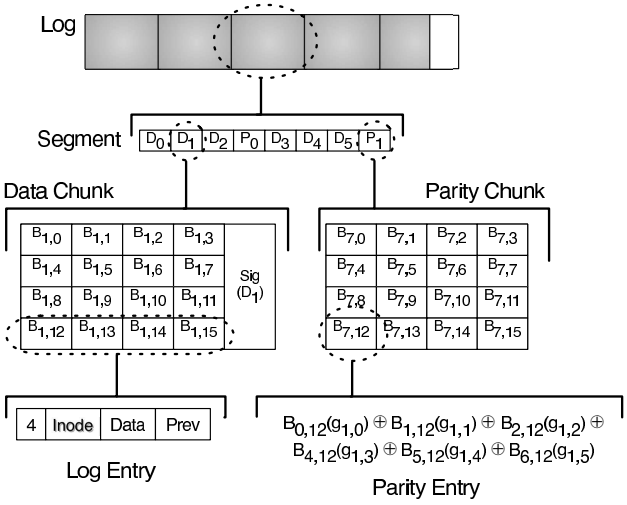


Figure 1: An example of the erasure-encoded log. Each parity block is computed from corresponding data blocks in a segment. Signatures are used in each data chunk to detect integrity violations. Corruption is detected at the chunk level and corrected by decoding the corrupted chunk's segment.

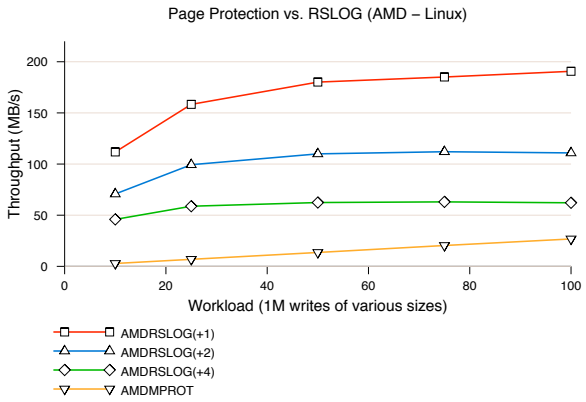


Figure 2: Throughput when writing data using our log-based encoding vs. writes surrounded by mprotect calls. These results were generated from the average throughput over 20 trials of each workload on an 2.4 GHz AMD Opteron machine running Linux. As shown in the graphs, depending on the choice of encoding, small write throughput in the encoded log is much greater than page protection for the same workload.

The most widely-used memory protection technique—page table-based protection—provides isolation and protection through paging and access control using the page protection bits in each page table entry (PTE). Because page-level access control requires modifying PTEs and flushing the TLB each time page permissions are changed, a small but constant penalty is incurred for each write. This penalty is relatively small for large writes, but greatly degrades performance for workloads consisting of small writes.

Access control provides very little in terms of hardware and software error recovery. These mechanisms are designed to prevent different parts of the operating system from performing unauthorized data access. Thus, if a hardware error occurs or a piece of software evades the access control mechanisms—performing an unauthorized write—the original data cannot be recovered. When dealing with persistent data, even a single instance of corruption could lead to loss of file system data.

To address the issues listed above we developed PRIMS. In PRIMS, all file system metadata is written to a log. Maintaining log consistency is extremely crucial because the log holds the *only* copy of the file system metadata—metadata is never written to disk. Thus, NVRAM-resident metadata may be compromised due to wild writes, random bit flips, media wear or, in the case of multiple NVRAM banks, media failure. PRIMS uses the log structure to verify the consistency of file system metadata by replaying log transactions against the live state of the system, and by checking that written data matches the signatures stored for the data. As an additional measure, module identification information may be added to each write, giving PRIMS the ability to determine which part of the OS issued the write.

PRIMS contains a set of logically contiguous, fixed-sized segments, as shown in Figure 1, with each segment containing the same number of fixed-sized chunks. Data is appended to the log by writing to the current data chunk and writing parity updates to the appropriate parity chunks in the current segment. Each chunk is composed of indivisible, fixed-sized blocks; these blocks may be any size, and may be significantly smaller than standard disk blocks. The log-structure gives PRIMS a way to distinguish between *live* and *dead* regions, where a live region contains writable segments and a dead region contains filled segments. If a dead region is changed, PRIMS will detect an unauthorized change and reverse it using the erasure encoding.

PRIMS uses linear erasure codes, such as Reed-Solomon codes, to allow flexible numbers of data chunks and associated redundancy chunks. Linear erasure codes have a reputation for being slow because of the expensive Galois field multiples required for encoding; however, we obtain reasonable encoding speeds by using multiplication tables and incremental parity updates. PRIMS uses algebraic signatures rather than cryptographically secure hash functions such as SHA-1 to perform periodic integrity checks [14]. To check the integrity of a segment, PRIMS must do two things: verify that each chunk is consistent with its algebraic signature, and verify that the algebraic signatures for a segment are consistent with one another. The use of algebraic signatures and linear erasure codes enables this approach, since the computation of algebraic signatures and the use of erasure codes based on the same Galois field commute. These operations can be done together, or they can be decoupled, allowing sets of signatures over a segment to be verified in the background and the correspondence of data and a signature to be verified when the data is used. If an error is found during verification PRIMS treats the chunk as an erasure and recovers the correct data.

Figure 2 shows the performance of our log structure compared to an approach that uses page protection. These

preliminary results are taken from a simple erasure encoded log implementation on two architectures (Intel and AMD) and two operating systems (Linux 2.6.17 and Mac OS 10.4.8) using a Reed-Solomon and algebraic signature library we are developing. Because Intel performance was similar to that on AMD, we only present the AMD results in Figure 2. The experiment performs one million writes, each between 10 and 100 bytes, to the log and to a protected region. Writes to the protected region require two calls to `mprotect` to unprotect and protect a page. Log writes require a single data write and one or more parity updates. In our test, we chose to use 1, 2 and 4 parity chunks per segment. As Figure 2 shows, the use of `mprotect` is not appropriate for small-write workloads that require frequent permission changes. For writes of size 10–50 bytes, our log-based approach outperforms page protection by at least a factor of six, and in most cases more than an order of magnitude. Though not shown in Figure 2, page protection outperforms the encoded-log at write sizes of at least 1 KB, confirming the utility of page protection in write caches for page-based data. We believe that most of the `mprotect` overhead was due to modifying the page tables and flushing regions of the TLB during each protection call.

5 PRIMS Advantages

The goal of PRIMS is to provide a reliable system that permanently places small pieces of data, particularly metadata, in NVRAM. However, our initial design provides a number of additional benefits beyond higher performance.

Storage system capacity is increasing at an exponential rate. As capacity grows, so does the probability of error and time required for recovery. Instead of focusing solely on performance, we are exploring the tradeoffs between performance and reliability. In doing so we are creating a system that will exhibit high performance and maintain firm consistency guarantees. In addition, distributed file systems will benefit from the potential power savings and performance boost associated with storing metadata and indices exclusively in NVRAM. One clear advantage is the ability to search the file system even when a great deal of the disks are spun-down.

PRIMS is being designed to run on any platform or hardware architecture. All error correction is computed in software, so there is no need for hardware-based ECC. Moreover, PRIMS does not even require an MMU because it does not rely upon page tables for protection, reducing hardware costs and resulting in tunable fault tolerance and encoding across independent banks of NVRAM. The software-based erasure-encoded log is expected to tolerate wild writes, file system bugs, media errors and media wear regardless of the underlying platform, making it well-suited for portable devices and single-host file systems as well as distributed systems.

6 Status

We are currently implementing PRIMS using off-the-shelf DRAM as a stand-in for either phase-change RAM or MRAM. We are designing metadata structures that take advantage of the high performance and reliability of PRIMS to provide a significantly faster, more reliable file system than is possible using disk-based metadata. While our preliminary tests do not give any indication of how effective an erasure-encoded log will be at protecting metadata, we expect that our mechanisms provide better protection than page-based access control by catching a wider range of hardware and software errors. We plan to have an in-kernel prototype of PRIMS available in the coming months, and will use this prototype to demonstrate both the improved performance and improved reliability that storing file system metadata in reliable NVRAM can provide.

References

[1] Samsung Introduces the Next Generation of Nonvolatile Memory-PRAM, Sep 2006.

[2] Toshiba and NEC Develop World's Fastest, Highest Density MRAM, Feb 2006.

[3] M. Baker, S. Asami, E. Deprit, J. Ousterhout, and M. Seltzer. Non-volatile memory for fast, reliable file systems. In *Proceedings of ASPLOS-V*, pages 10–22, 1992.

[4] P. M. Chen, W. T. Ng, S. Chandra, C. Aycock, G. Rajamani, and D. Lowell. The Rio file cache: Surviving operating system crashes. In *Proceedings of ASPLOS-VII*, Oct. 1996.

[5] K. M. Greenan and E. L. Miller. Reliability mechanisms for file systems using non-volatile memory as a metadata store. In *Proceedings of ACM/IEEE EMSOFT '06*, Oct. 2006.

[6] V. Henson, A. van de Ven, A. Gud, and Z. Brown. Chunkfs: Using divide-and-conquer to improve file system reliability and repair. In *Proceedings of HotDep '06*, 2006.

[7] D. E. Lowell and P. M. Chen. Free transactions with Rio Vista. In *Proceedings of SOSP '97*, pages 92–101, 1997.

[8] M. K. McKusick. Running fsck in the Background. In *Proceedings of the BSDCon*, pages 55–64, 2002.

[9] M. K. McKusick and T. Kowalski. *4.4 BSD System Manager's Manual*, chapter 3, pages 3:1–3:21. O'Reilley and Associates, Inc., Sebastopol, CA, 1994.

[10] E. L. Miller, S. A. Brandt, and D. D. E. Long. HeRMES: High-performance reliable MRAM-enabled storage. In *Proceedings of HotOS-VIII*, pages 83–87, May 2001.

[11] J. Piernas, T. Cortes, and J. M. Garcìa. DualFS: a new journaling file system without meta-data duplication. In *Proceedings of ICS '02*, pages 137–146, 2002.

[12] F. Qin, S. Lu, and Y. Zhou. SafeMem: Exploiting ECC-memory for detecting memory leaks and memory corruption during production runs. In *Proceedings of HPCA-XI*, 2005.

[13] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1):26–52, Feb. 1992.

[14] T. Schwarz, S. J. and E. L. Miller. Store, forget, and check: Using algebraic signatures to check remotely administered storage. In *Proceedings of ICDCS '06*, July 2006.

[15] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck. Scalability in the XFS file system. In *Proceedings of USENIX Annual Tech. '96*, Jan. 1996.

[16] A.-I. A. Wang, G. H. Kuenning, P. Reiher, and G. J. Popek. Conquest: Better performance through a disk/persistent-RAM hybrid file system. In *Proceedings of USENIX Annual Tech. '02*, Monterey, CA, June 2002.

[17] E. Witchel, J. Cates, and K. Asanović. Mondrian memory protection. In *Proceedings of ASPLOS-X*, Oct 2002.

A Spin-Up Saved is Energy Earned: Achieving Power-Efficient, Erasure-Coded Storage

Kevin M. Greenan[†]
Univ. of California, Santa Cruz
kmgreen@cs.ucsc.edu

Darrell D.E. Long[†]
Univ. of California, Santa Cruz
darrell@cs.ucsc.edu

Ethan L. Miller[†]
Univ. of California, Santa Cruz
elm@cs.ucsc.edu

Thomas J. E. Schwarz, S.J.
Santa Clara University
tjschwarz@scu.edu

Jay J. Wylie
HP Labs
jay.wylie@hp.com

Abstract

Storage accounts for a significant amount of a data center's ever increasing power budget. As a consequence, energy consumption has joined performance and reliability as a dominant metric in storage system design. In this paper, we show that the structure of an erasure code—which is generally used to provide data reliability—can be exploited to save power in a storage system. We define a novel technique in power-aware systems called *power-aware coding* and present generic techniques for reading, writing and activating devices in a power-aware, erasure-coded storage system. While our techniques have an effect on energy consumption, fault tolerance and performance, we focus on a few examples that illustrate the tradeoff between power efficiency and fault tolerance. Finally, we discuss open problems in the space of power-aware coding.

1 Introduction

Traditionally, storage systems are measured in terms of performance and reliability. Due to the increasing amount of data stored in recent years and the significant amount of power required to store such data, a great deal of work has gone into measuring and minimizing the power consumption of storage systems [2, 12, 14, 6, 15, 11]. Storage accounts for roughly 27% of a data center's power budget [1]; thus, proactively activating and deactivating disks can effectively lower the energy footprint of a data center.

Almost every storage system generates redundancy to provide data reliability in the face of failures. In most cases, an erasure code is defined across a group of disks to provide reliability. When a disk fails within a group, other disks in the group are used to recompute the contents of the failed disk.

[†]Supported in part by the Petascale Data Storage Institute under Dept. of Energy award FC02-06ER25768 and by the industrial sponsors of the Storage Systems Research Center.

In addition to providing fault tolerance, erasure codes may also be used to prevent disk activation. We call this technique *power-aware coding*. Consider the case where all disks are working correctly (no failures). If the system can tolerate the failure of any single disk in the group, then one disk can remain inactive for an extended period of time and need not be activated if a read request is directed to the inactive disk. Instead of activating the disk, its contents can be reconstructed from the active disks.

The number of active disks required to reconstruct the contents of one or more inactive disks is determined by the erasure code. Given k data disks, the class of codes, called maximum distance separable (abbreviated MDS and explained in Section 2), require at least k active devices to reconstruct the contents of an inactive device. While any erasure code may be used to save power, we believe that another class of codes, called non-MDS, are best suited for power-aware coding because less than k active devices are generally required to rebuild the contents of an inactive device.

To date, a handful of power-aware redundancy techniques have been proposed. Pinheiro *et al.* place data and parity on different disks; deactivating parity disks during light loads and staging parity updates in non-volatile RAM [6]. PARAID is a power-aware disk array architecture that trades logical capacity for power efficiency by replicating blocks from inactive disks onto spare regions of active disks [12]. e-RAID introduced the idea of *transformed reads* for RAID-1 and RAID-5, which allows the system to rebuild the contents of an inactive disk from cache, other active disks, or both [11].

Our work is similar to e-RAID, which prevents activation under RAID-1 and RAID-5. Both RAID-1 and RAID-5 have trivial solutions with respect to servicing read requests from inactive devices: redirect the request to an active mirror or recompute from the active devices, respectively. The RAID-1 technique trades a great deal of space for power efficiency; the RAID-5 technique is

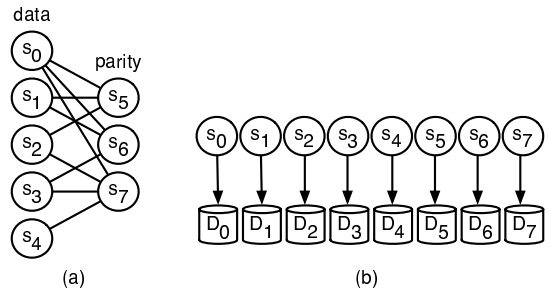


Figure 1: A flat XOR-based code with 5 data symbols and 3 parity symbols. Each symbol is mapped to a unique disk.

limited in utility, since all but one disk must be active to service any read. Recent analysis suggests that storage systems should have the ability to tolerate more than one failure [3], indicating the need for codes that provide more fault tolerance than RAID-5. The power-aware coding techniques we propose and the associated problems we identify apply to any erasure code.

The contributions of this work are fourfold. First, we present power-aware coding, which provides a way to evaluate the tradeoff between fault tolerance and power efficiency in storage systems. Second, we discuss challenges and initial work within the scope of power-aware coding including writing, reading and activating disks in an erasure-coded, power-aware storage system. Third, we present an example to illustrate tradeoffs in power-aware, erasure-coded storage systems. Finally, we discuss open problems in the space of power-aware coding.

2 Background: Erasure Codes

An erasure code is made up of codewords that have n symbols. Storage systems typically use systematic erasure codes, where each codeword contains k data symbols and $m = n - k$ parity (redundant) symbols. A code symbol generally refers to a sector or set of sectors on a single storage device. The *Hamming distance* of a code provides a compact representation of fault tolerance. Any set of failures strictly less than the Hamming distance may be tolerated. Traditionally, storage systems use maximum distance separable (MDS) codes, which provide optimal fault tolerance by having a Hamming distance of $m+1$ (they can tolerate up to m failures out of n symbols). In this work we focus on so-called *flat-codes*: codes that map exactly one symbol per codeword to a device.

Any code that is not MDS is called *non-MDS*. While MDS codes can tolerate up to any m failures, the Hamming distance of a non-MDS code is strictly less than m, but may tolerate some failures at or beyond the Hamming distance. In this paper, we focus on a class of non-MDS codes called *flat XOR-based codes* [13]. An example of a flat XOR-based code is shown in Figure 1-a. This code is described by a bipartite graph, called a Tanner graph [10]. The data symbols form the left nodes in the graph, while the parity symbols are the right nodes. A data symbol contributes to a parity symbol if an edge connects the corresponding nodes in the graph. The parity equations are derived by following the edges connected to each parity node in the graph. For instance, the parity node for symbol s_5 is adjacent to s_0, s_1 and s_2, therefore, we compute s_5 as $s_0 \oplus s_1 \oplus s_2$. From the graph, we see that s_6 is computed as $s_0 \oplus s_1 \oplus s_3$ and s_7 is computed as $s_0 \oplus s_2 \oplus s_3 \oplus s_4$.

Since we consider only systematic codes, a $k \times n$ generator matrix, G, is used to compute the $m = n - k$ parity symbols from k data symbols. The first k columns of the matrix form a $k \times k$ identity matrix. The last m columns are used to compute each of the m parity symbols from the data symbols. Thus, every systematic generator matrix will have the form $G = (I_k|P)$, where the elements of the matrix are taken from a finite field. A codeword is computed as $d \cdot G = c$, where d is a $1 \times k$ vector containing the k data symbols.

The Tanner graph of the flat XOR-based code shown in Figure 1-a is transformed into a generator matrix by creating a column for each node in the graph . The nodes s_0, s_1, s_2, s_3 and s_4 form the first five columns, where the column for symbol s_i is the unit vector with a 1 in the i-th element. A parity column (s_5, s_6 and s_7) is a linear combination of the columns that correspond to the data symbols involved in the parity equation. For example, the column corresponding to symbol s_5 is the sum of the 0th, 1st and 2nd columns of the matrix.

The generator matrix for an MDS code is typically created by starting with a specific type of matrix (i.e. Vandermonde) and performing elementary matrix operations to transform the matrix into one of the form $(I_k|P)$, where the entries of the matrix are in the finite field of 2^8, 2^{16} or 2^{32} elements.

While non-MDS codes are not as fault-tolerant as MDS codes with the same number of data and parity symbols, rebuild of a single symbol typically involves less than k symbols. This is not the case for MDS codes, where k available symbols are required to reconstruct the contents of 1 to m symbols. The suitability of non-MDS codes to power-aware storage follows from this observation—data symbols on inactive devices may be reconstructed from fewer active devices compared to MDS codes.

3 Power Aware Coding

We define *power-aware coding* in terms of a set of disks and an *erasure code instance* across the disks. An erasure code instance is a mapping of erasure-coded symbols—data or parity—to disks. An example code instance is shown in Figure 1-b. As shown, we assume that the number of symbols in a codeword is equal to the number of disks and each symbol is mapped to a unique disk. In this case, each code symbol, s_i, is mapped to disk D_i. For brevity, we assume that all disks have the same capacity and characteristics; however, our techniques have the ability to support disks with different characteristics.

The crux of power-aware coding is to prevent spinning up inactive disks when servicing read requests by treating each inactive disk as an erasure. As an example, consider the setup shown in Figure 1. Suppose disks D_0, D_5, D_6 and D_7 are currently active; all others are inactive. If the system receives a read request for disk D_4 we can service the request as $D_0 \oplus D_5 \oplus D_6 \oplus D_7$ instead of activating disk D_4, since $s_4 = s_0 \oplus s_5 \oplus s_6 \oplus s_7$. The comparable MDS code with 5 data disks and 3 parity disks would require a disk activation, since 4 active devices are not sufficient to recover the contents of an inactive device.

Three conditions are necessary for an erasure-coded system to be power-aware. First, the system must have policies that service writes in a way that minimizes operational power consumption, while maintaining a sufficient level of reliability. Second, a read policy will dictate if data is accessed directly off a disk or reconstructed using redundant information. Finally, when disk activation is necessary to service a read request, a policy is needed to determine how to efficiently schedule disk activations.

3.1 Servicing Writes

We assume the system has a total of N disks; thus, there are a total of $\frac{N}{n}$ code instances. Writes into the system are serviced by deterministic disk activations. A *write group* is a list of disks that will be active within a single code instance at the same time to perform this function. Every disk in the system must be a member of at least one write group and will most likely belong to several write groups. When a write group is active, its disks are also active. Exactly *one* write group per code instance will be active at a time. Each write group is identified by a tuple containing a unique id, a *begin time* and an *end time*. A *power schedule* is a list of these tuples; thus, it temporally specifies how disks are deterministically activated and deactivated within a code instance.

Servicing writes via write groups assumes that the storage system is write-anywhere or defers writes, since writes are only handled by disks in the current write group. Approaches such as Pergamum [9] and write offloading [5] also use these techniques to save power.

The total power consumed by the storage system is heavily dependent on the power schedule. A write group will likely be active for a number of hours and keep the number of active disks to a minimum. In addition, the number of active disks determine which data can be reconstructed; thus, a proper balance is required to service both writes and reads into the system.

3.2 Servicing Reads

Read requests are satisfied by either accessing an active disk or reconstructing the appropriate content using the active disks (via the erasure code). In some cases, the information provided on the active disks will be insufficient for serving the request. A *transient disk activation* occurs when a disk must be activated to service a read request. A transient activation may be used to directly service the request or as part of data reconstruction if the request involves multiple disks. Since a transient activation involves a disk that is not a member of the current write group, it will be deactivated after some fixed period and will not service writes. In addition to reads, transient activations may also be used to perform background operations such as disk scrubbing [8].

Choosing to perform reconstruction, transient activation or a combination of the two depends on the environment and workload. There may be cases where a transient activation may be more power efficient than reconstructing the data from active disks. The system should optimize for each read request based on the state of the system and number of disks involved.

3.3 Scheduling Transient Activations

Most systems handle read requests to inactive disks using the *naive strategy*, which simply activates the disks involved in the request. A great deal of transient activations can have a dramatic effect on system power consumption and reliability. In addition, recent analysis shows that the system reliability will decrease if disks are power cycled too often [8]. In order to minimize power consumption and maintain a reasonable level of reliability, the system should minimize the number of transient disk activations.

4 Power-Aware Techniques

In this section we cover initial policies for servicing writes, servicing reads and handling transient disk activations. These policies serve as a starting point for constructing erasure-coded, power-aware systems.

4.1 Power Schedule

Each of the $\frac{N}{n}$ code instances in the system must have a policy for generating its own write groups. Consider a simple policy, called *single-data connected-parity*, for constructing write groups. Under this policy, write groups are generated based on the parity equations for the code instance. A write group is generated for each data disk. We assign each data disk, D_i, to a unique write group, W_i. A parity disk D_j is added to write group W_i if D_i contributes to the parity equation for D_j. This policy is biased towards writes because it allows all parity updates to immediately complete, since a data disk and its associated parity disks will be active at the same time. Under this policy, the write groups for the code instance in Figure 1 would be $\{D_0, D_5, D_6, D_7\}$, $\{D_1, D_5, D_6\}$, $\{D_2, D_5, D_7\}$, $\{D_3, D_6, D_7\}$ and $\{D_4, D_7\}$. If every code instance implements this policy, then data may be written in parallel across $\frac{N}{n}$ disks. Write groups may be defined by more than one data disk in environments that must sustain heavy write workloads.

The duration of a write group—calculated by subtracting the *end time* entry from the *begin time* entry in the write group's corresponding tuple—is a tunable parameter based on utilization, workload and frequency of in-

tegrity checks. For this reason, analyzing the appropriate duration of a write group is left to future work.

4.2 Power-Aware Read Algorithm

At a high level, the power-aware read algorithm treats inactive devices as erased and relies on matrix methods to determine if partial or whole-stripe reconstruction is possible using disks that are already active [4]. If reconstruction of any erased data is possible, the matrix transformations result in appropriate recovery equations. Instead of marking a disk as failed (or erased), we mark all inactive devices *tentatively lost*. A tentatively lost device is made available through activation. When a read request involves data that is tentatively lost, we try to reconstruct the elements in a way that minimizes the number of disk activations.

Our read algorithm relies on a function that determines if lost data is recoverable, and if so, the equations needed to reconstruct. The recovery equations for tentatively lost data are computed using the underlying generator matrix, G. A matrix, G', is constructed by zeroing out the columns in G that correspond to the tentatively lost elements. In order to determine the recovery equations we must find a pseudo-inverse, R (as defined by Hafner *et al.* [4]), of G'. Suppose the vector c' is the vector $c = d \cdot G$ with zeroes in the positions corresponding to tentatively lost elements. Then $c' \cdot R = d'$, where the non-zero elements of d' are the corresponding recoverable elements of d and the zero elements are unrecoverable. In this case, R contains the recovery equations and c' contains the available data and parity symbols.

The power-aware read algorithm is shown in Algorithm 1. The algorithm takes the inactive disks involved in the read request (I), the generator matrix for the underlying code (G) and the set of currently inactive devices (L) as input. The function `recoverable` uses the aforementioned matrix methods to determine the disks that are recoverable based on the underlying code and the set of currently available (or recoverable) disks. The `recoverable` function returns a list of recoverable disks, L', and the corresponding recovery equations. The `activate_disk` function, which is explained in Section 4.3, determines the disk (or disks) to activate given the read request and state of the system.

As an example, suppose $I = \{D_2, D_4\}$ and $L = \{D_1, D_2, D_3, D_4\}$ in the code instance shown in Figure 1. In the first iteration, `recoverable` returns $(\{D_4 = D_0 \oplus D_5 \oplus D_6 \oplus D_7\}, \{D_4\})$. The second iteration begins with $I = \{D_2\}$ and $L = \{D_1, D_2, D_3\}$ and `recoverable` returns $(\emptyset, \emptyset)$, therefore, a disk must be activated. Since D_2 is the only disk left in I, `activate_disk` returns $(\{D_2 = D_2\}, \{D_2\})$. The loop invariant evaluates to `false` at the beginning of the third iteration and the algorithm returns the corresponding recovery equations.

Algorithm 1 Recover I using G and L

```
1:  while I ≠ ∅ do
2:      (eqns, L') ← recoverable(G, L)
3:      if L' = ∅ then
4:          (eqns, L') ← activate_disk(L, G, I)
5:      else
6:          I ← I − L'
7:          L ← L − L'
8:      end if
9:      all_eqns.append(eqns)
10:     return all_eqns
11: end while
```

4.3 Disk Activation Algorithm

Since our approach takes advantage of the underlying erasure code, there exist many cases where the naive activations can be avoided. If the read request contains a single inactive disk that cannot be reconstructed, then we simply activate that disk. If more than one inactive disk is in a read request, we must determine the minimum number of activations required to service the request.

Algorithm 2 performs a brute force search of potential disks to activate by generating all possible combinations of disk activations (i.e. powerset of inactive disks). The powerset function, $\mathcal{P}$, orders the combinations in ascending order by size. For each combination, s, the algorithm determines if the request can be satisfied when the disks listed in s are activated (via `is_fully_recoverable`). Once a satisfactory combination is chosen, the algorithm returns the disks to activate and an updated list of inactive devices. Since the combinations are ordered, this algorithm will return the minimum number of activations needed to service the request.

Algorithm 2 Determine the minimum number of disk activations required to service request I when disks in L are inactive.

```
1: for s ∈ P(L) − ∅ do
2:     try ← L − s
3:     if is_fully_recoverable(try, I, G) then
4:         L ← L − s
5:         return (s, L)
6:     end if
7: end for
```

5 Example Usage

In this section, we clarify a few of the tradeoffs and illustrate the utility of non-MDS codes for power-efficiency in a small system containing 8 disks. We have chosen three erasure codes to illustrate the various tradeoffs in terms of power consumption and fault tolerance: (5,3)-FLAT, (4,4,2)-FLAT and (6,2)-MDS. The (5,3)-FLAT code is the code shown in figure 1. The (6,2)-MDS is an MDS code with 6 data symbols and 2 parity symbols. The (4,4,2)-

FLAT is a flat XOR-based code with data symbols s_0, s_1, s_2 and s_3, where each data symbol is connected to exactly 2 parity symbols. The parity equations of (4,4,2)-FLAT are: $s_4 = s_2 \oplus s_3$, $s_5 = s_0 \oplus s_3$, $s_6 = s_0 \oplus s_1$ and $s_7 = s_1 \oplus s_2$. As we will see later in this section, these codes were chosen due to similar, but not identical, fault tolerance properties.

We assume all disks are identical and P_r, P_a and P_i is the power (in watts) consumed by each disk when reading, active but not reading and inactive, respectively. Furthermore N_a and N_i represent the number of active and inactive disks in the system. A read request of size R (MB) takes a disk with transfer rate D_R (MB/s) and average rotational latency D_L (s) approximately $TTS = D_L + \frac{R}{D_R}$ seconds to service. Finally, each disk consumes P_{sp} watts during a transition from inactive mode to active mode, which takes T_{sp} seconds. We leave the energy calculation for CPU, network, and so on to future work.

Our analysis utilizes the energy consumption values of an IBM Ultrastar 36Z15. The values are: $P_r = 13.5$ W, $P_a = 10.2$ W, $P_i = 2.5$ W, $D_R = 55$ MB/s, $P_{sp} = 13.5$ W, $T_{sp} = 10.9$ s and $D_L = 2$ ms.

We consider four possible system configurations. System A uses the (5,3)-FLAT code and the single-data connected-parity write group policy. System B uses the (6,2)-MDS code and the single-data connected-parity write group policy. System C uses the (6,2)-MDS code and a write group policy similar to e-RAID, where all but two disks are active. Finally, system D uses the (4,4,2)-FLAT and the single-data connected-parity write group policy.

5.1 Power Consumption due to Write Groups

We estimate the power consumption of each system configuration due to a given write group policy and erasure code. To simplify the analysis, we approximate the energy consumption without considering the workload. We believe that our calculation, while inaccurate in an absolute sense, is sufficient for comparison. We understand that detailed simulation is required to obtain accurate energy consumption numbers.

Each system will have N_a active devices and N_i inactive devices. On average, each system will consume approximately $(N_a \cdot P_a) + (N_i \cdot P_i)$ watts. System A and D have, on average, 3 devices active and each consume approximately 43.1 W. Systems B and C will have 3 and 6 devices active at all times and consume 43.1 W and 66.2 W, respectively. The logical (usable) capacity of each system can be used to normalize to watts-per-data-disk. The normalized energy consumption of each system is: system A is 8.62 W, system B is 7.18 W, system C is 11.03 W and system D is 10.78 W.

5.2 Reconstruction Ability of Each System

Each of the three systems have different capabilities in terms of reconstructing data present on inactive devices. System C has the ability to service any read request without activating any disks, since 6 active disks is sufficient to reconstruct the contents of any two inactive disks. Similarly, system B needs at least 6 active devices to reconstruct the contents of an inactive disk. Every write group in system B contains 3 disks; thus, it is unlikely system B will have the ability to service reads via reconstruction.

The reconstruction capability of system A sits somewhere between that of system B and C. For instance, while the write group containing D_0, D_5, D_6 and D_7 is active, any request to D_4 can be serviced without activating any disks, because $D_4 = D_0 \oplus D_5 \oplus D_6 \oplus D_7$. As another example, suppose a read request is directed at D_2 and D_3 under the same write group. While a similar request in system B would require activating both disks, only a single disk activation (D_2) is required to service the request in system A, since $D_3 = D_2 \oplus D_5 \oplus D_6$.

The reconstruction capability of system D highlights the tradeoff between storage efficiency and prospective power savings. For example, if the write group containing D_0, D_5 and D_6 is active, all but one data symbol can be reconstructed: $D_1 = D_6 \oplus D_0$ and $D_3 = D_0 \oplus D_5$. The symmetric structure of this code makes this true for all write groups. One could argue that the same reconstruction ability is possible by simply activating three data disks, but there would be no parity disks active to service writes into the system. This code illustrates a nice balance when optimizing power efficiency for both reads and writes.

5.3 Disk Activation vs. Data Reconstruction

In order to evaluate the efficacy of our read policies, we calculate the approximate power consumption when a 50 MB read request is targeted at a single inactive disk. We save more complicated cases, such as combining activation and reconstruction, for future work. System A, C and D will service the read request by either activating the disk or performing partial reconstruction. System B must activate the disk to service the request. The approximate power (in Joules) required when activating a disk to service the request is

$$(P_{sp} \cdot T_{sp}) + (TTS \cdot (N_a \cdot P_a + N_i \cdot P_i + P_r)),$$

where the first term in the summation represents the power consumed to activate the disk and the second term is the power consumed when servicing the request. The disk activation power consumption of systems A, B and D is 198.71 J, since all systems have, on average, 3 disks activated at any given time. The power consumption of system C is approximately 219.76 J. Note that these power consumption numbers are highly dominated by the power required to spin-up an inactive disk, thus avoiding such activations is crucial.

If the contents of the inactive disk can be reconstructed from the active disks, we can approximate the power consumed as $(TTS \cdot (N_a \cdot P_r + N_i \cdot P_i))$. In this case, we assume that all active disks participate in the rebuild operation. To be fair, we assume $N_a = 4$ for system A, since 4 disks must be active to perform reconstruction under the single-

data connected-parity policy. The reconstruction power consumption of systems A, C and D are 58.30 J, 78.35 J and 48.28 J, respectively. Since system B cannot reconstruct the contents of any inactive disks, it must consume 198.71 J.

5.4 Dependability and Power Efficiency

We chose the three codes in this example due to similar fault tolerance properties. The (6,2)-MDS is space-optimal and has the ability to tolerate any 2 disk failures. The (5,3)-FLAT code can tolerate all but one 2 disk failure (the failure of D_4 and D_7), thus it is not quite as fault tolerant as the (6,2)-MDS code. Finally, the (4,4,2)-FLAT can tolerate any 2 disk failures, all but four 3 disk failures and all but five 4 disk failures. In this case we see that a system designer must trade both fault tolerance and space for power efficiency.

It is important to point out the tradeoff between power efficiency, fault tolerance and space efficiency. The (6,2)-MDS code provides optimal fault tolerance for 6 data and 2 parity elements, but lacks flexibility in terms of potential power savings; at least 6 devices must be active in order to reconstruct the contents of any inactive device. The (5,3)-FLAT code is not as space efficient or fault tolerant as (6,2)-MDS, but provides more opportunities to recover data off inactive devices. By trading even more space, the (4,4,2)-FLAT code is more fault tolerant than the other codes analyzed and provides many opportunities to exploit redundancy for power savings. While we cannot make any sweeping generalizations, we believe this example motivates further analysis when trading power consumption, fault tolerance and space efficiency.

6 Discussion

Introducing the power-aware coding technique, its terminology and a few motivating examples is only a first step. There are three forms of open problems in this area: determining which environments will benefit from power-aware coding, finding optimal policies and erasure codes for a given environment and developing robust metrics for evaluating the power-reliability-performance tradeoff.

The problem of finding optimal policies and erasure codes is heavily dependent on workload and is very difficult to solve analytically; the optimization must trade power consumption, reliability and performance. In addition, considering reconstruction in terms of both actual failures and a read policy remains an open problem. To gain insight in these areas, we hope to experiment with standard algorithms in the area of constraint satisfaction problems [7] and are currently building a simulation environment to determine the utility of power-aware coding in a variety of settings. Initial results show that power-aware coding may be well suited for the *write-once, read-maybe* workload of long-term archival storage systems.

The metrics used in this paper estimate power consumption and only account for the power consumed by disks. In addition to accounting for other energy consumers, such as CPUs, finding accurate metrics for evaluating the power-reliability-performance tradeoff remains an open problem.

References

[1] B. Battles, C. Belleville, S. Grabau and J. Maurier. Reducing Data Center Power Consumption Through Efficient Storage. NetApp Technical Report, WP-7010-0207, 2007.

[2] D. Colarelli and D. Grunwald. Massive arrays of idle disks for storage archives. In *Supercomputing '02*, pages 1–11, 2002.

[3] J. G. Elerath and M. Pecht. Enhanced reliability modeling of raid storage systems. In *DSN 2007*, pages 175–184. IEEE, June 2007.

[4] J. L. Hafner, V. Deenadhayalan, K. Rao, and J. A. Tomlin. Matrix methods for lost data reconstruction in erasure codes. In *FAST 2005*, pages 183–196, Dec. 2005.

[5] D. Narayanan, A. Donnelly, and A. Rowstron. Write off-loading: Practical power management for enterprise storage. In *FAST 2008*, Feb. 2008.

[6] E. Pinheiro, R. Bianchini, and C. Dubnicki. Exploiting redundancy to conserve energy in storage systems. In *SIGMETRICS 2006*, pages 15–26, 2006.

[7] S. J. Russell and P. Norvig. *Artificial intelligence: a modern approach*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1995.

[8] T. J. E. Schwarz, Q. Xin, E. L. Miller, D. D. E. Long, A. Hospodor, and S. Ng. Disk scrubbing in large archival storage systems. In *MASCOTS 2004*, pages 409–418, 2004.

[9] M. W. Storer, K. M. Greenan, E. L. Miller, and K. Voruganti. Pergamum: Replacing tape with energy efficient, reliable, disk-based archival storage. In *FAST 2008*, Feb. 2008.

[10] R. Tanner. A Recursive Approach to Low Complexity Codes. *IEEE Trans. on Information Theory*, 27(5):533–547, Sep 1981.

[11] J. Wang, H. Zhu, and D. Li. e-RAID: Conserving energy in conventional disk-based RAID system. In *IEEE Transactions on Computers*, 2008.

[12] C. Weddle, M. Oldham, J. Qian, A.-I. A. Wang, P. Reiher, and G. Kuenning. Paraid: A gear-shifting power-aware raid. *Trans. Storage*, 3(3):13, 2007.

[13] J. J. Wylie and R. Swaminathan. Determining fault tolerance of XOR-based erasure codes efficiently. In *DSN 2007*, pages 206–215. IEEE, June 2007.

[14] Q. Zhu, Z. Chen, L. Tan, Y. Zhou, K. Keeton, and J. Wilkes. Hibernator: helping disk arrays sleep through the winter. In *SOSP '05*, pp. 177–190, 2005.

[15] Q. Zhu, F. David, C. Devaraj, Z. Li, Y. Zhou, and P. Cao. Reducing energy consumption of disk storage using power-aware cache management. In *HPCA '04*, 2004.

Building Flexible, Fault-Tolerant Flash-based Storage Systems

Kevin M. Greenan [†] Darrell D.E. Long [†] Ethan L. Miller [†]

Thomas J. E. Schwarz, S.J. [‡] Avani Wildani [†]

Univ. of California, Santa Cruz [†] *Santa Clara University* [‡]

Abstract

Adding flash memory to the storage hierarchy has recently gained a great deal of attention in both industry and academia. Decreasing cost, low power utilization and improved performance has sparked this interest. Flash reliability is a weakness that must be overcome in order for the storage industry to fully adopt flash for persistent storage in mission-critical systems such as high-end storage controllers and low-power storage systems.

We consider the unique reliability properties of NAND flash and present a high-level architecture for a reliable NAND flash memory storage system. The architecture manages erasure-coded stripes to increase reliability and operational lifetime of a flash memory-based storage system, while providing excellent write performance. Our analysis details the tradeoffs such a system can make, enabling the construction of highly reliable flash-based storage systems.

1 Introduction

Recent advances in the performance and density of NAND flash memory devices offer the promise of great performance and low energy consumption. NAND flash is being used to replace DRAM-based disk caches [7], store metadata [5], replace disks in laptops, and even replace high-performance disk arrays [8]. Unfortunately, NAND flash memory has limitations. First, write performance depends on the underlying architecture and does not rival its excellent read performance. For good write performance, each device must have multiple banks of NAND flash to be accessed in parallel [1]. Second, the known limited endurance of NAND flash generally requires wear-leveling algorithms and page-level ECC [2, 6, 7]. Finally, the raw bit error rate (RBER) of NAND flash increases with use [4]. Page-level ECC—which is typically used to provide reliability in NAND flash systems— does not provide protection from whole-page failures, chip failures, or whole-device failures. These types of failures are typically handled using inter-device redundancy, similar to RAID. With the exception of the RAMSAN-500 [8], we are unaware of any RAID-based solutions for fault tolerance across flash devices.

The contributions of this work are as follows. We briefly cover the important reliability and performance characteristics of NAND flash memory. Next, we provide an architecture for flash-based storage systems that uses erasure codes to tolerate failures at various levels from bit-flip to device failure. The level and scope of protection may be changed on-the-fly to adjust gracefully to the degenerative effects of aging. We conduct a preliminary reliability and performance analysis of this architecture. We find that our techniques provide a higher level of protection than existing approaches while maintaining good write performance.

2 Flash Background

Figure 1 shows an example NAND flash chip with 4 banks of flash that may be accessed in parallel. Each bank contains a single page register and 64 blocks. All accesses to a bank are *serialized* at the page register: data to be read or written to/from a bank must be first loaded into the page register. A block contains a total of 64 pages and each page has a 4KB data section and a 128B spare area. The spare area is used to store checksums, ECC bits and page-specific metadata. Individual data bits are stored in cells within the data section of a flash page. There are two primary technologies for storing bits in flash: single-level cell (SLC) and multi-level cell (MLC). SLC technologies store exactly one bit per cell, while MLC technologies store two or more bits per cell.

NAND flash memory has a unique access model. An entire block must be erased (all bits changed to 1) before any data can be programmed (written) into the pages

[†]Supported in part by the Petascale Data Storage Institute under Dept. of Energy award FC02-06ER25768 and by the industrial sponsors of the Storage Systems Research Center.

[‡]Supported in part by a research grant from the Stiles Family Foundation.

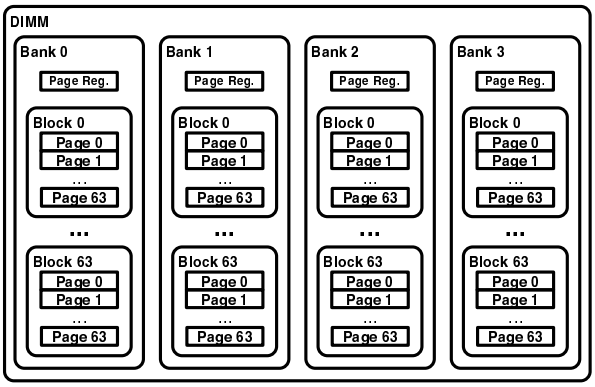


Figure 1: *A* 4 *bank NAND flash chip.*

within the block; thus, the act of writing to a block is generally called the *program-erase cycle*. Manufacturers rate the *endurance* of a block based on the underlying NAND architecture. Single-level cell (SLC) NAND flash blocks are generally rated at 10^5 program-erase cycles, while multi-level cell (MLC) blocks have a much lower endurance. Current two-level cell technologies are typically rated at 10^4 program-erase cycles. In addition to decreasing reliability, block erasures are slow; each block erase takes roughly 1.2 to 2 ms, which is an order of magnitude longer than a program operation.

Flash devices have unique reliability concerns. The most important of these concerns are *read disturb* and *program disturb* [4]. Read disturb errors occur as a side-effect of reading a page, which may induce multiple, permanent bit-flips in the corresponding block. Program disturb causes a program operation on one page to change the value of a bit on another page in the same block. These errors contribute to the raw bit-error rate (RBER), which is typically between 10^{-6} and 10^{-9}, depending on the number of bits per cell. A recent study has shown that program-erase cycles have a significant effect on RBER [4]. In some cases, the RBER increases by more than an order of magnitude as a block's program-erase count approaches its rated endurance. Given these observations, we expect *page errors* to be the primary failure pattern in flash-based systems.

We focus on three notable properties of NAND flash devices when building a flash-based storage system: *program-erase cycling*, *request serialization* at each bank and the *raw-bit error rate*.

3 System Design

Reflecting the needs of an industrial sponsor, we concentrate on a very large flash based design for storage servers where a single host or controller is connected to multiple PCI cards containing flash. We write to the flash devices in a log-structured manner, so that we do not need explicit wear leveling. Our design easily extends to a SSD (or USB) flash device as long as the host / controller has direct access to the pages on the flash device. While other reliable flash-based systems focus on bit-errors and extending flash lifetime using page-based ECCs, our design considers the proper application of erasure codes (or RAID) to a set of flash devices. In particular, we organize the pages into reliability stripes with additional parity pages, similar to what is being done in disk arrays. Because we write in a log-structured manner, the parity pages do not need to be maintained using the read-modify-write operation that is very expensive when used in flash.

There is a subtle difference between our approach and simply applying RAID algorithms to a set of flash devices. An out-of-the-box RAID approach would lead to hotspots on the flash devices and increased contention at each bank due to the traditional read-modify-write operation. We control the way data/parity is written to a set of flash devices to achieve the following goals:

(1) **Tolerate more than bit errors.** Most solutions rely on page-level ECC for reliability. In reality, the system must also tolerate burst, device, and chip-level errors. Our design achieves this goal by adding parity pages.

(2) **Uniform writes across all devices.** The reliability and performance consequences of program-erase cycles necessitates distributing the life-time number of writes uniformly. We manage writes in a log-structured manner that automatically stripes writes across all devices (or chips) and avoids read-modify-write.

(3) **Graceful degradation.** As a system ages, the probability of burst, device, and chip-level error as well as the bit error rate increases [4]. Our design has the ability to increase the fault tolerance for older systems at the cost of slightly lower capacity by changing the parameters of the page-level reliability stripes.

In achieving these goals, we essentially move the flash translation layer (FTL) off of the individual devices and onto the host. In an array of SSDs, this would be equivalent to moving the FTL into the array controller. By moving the FTL out of the individual devices, the host (or RAID controller) can optimize writes in a way that maximizes write performance, while writing uniformly across all of the devices.

3.1 System Architecture

Figure 2 illustrates the basic architecture of the flash-based storage system we are analyzing. The system is composed of PCI cards that contain flash memory chips. We assume a single host will contain multiple PCI buses, allowing the host to address flash on multiple cards. Communication between the host and the card uses a DMA channel. The host has a set of DMA rings, each of which

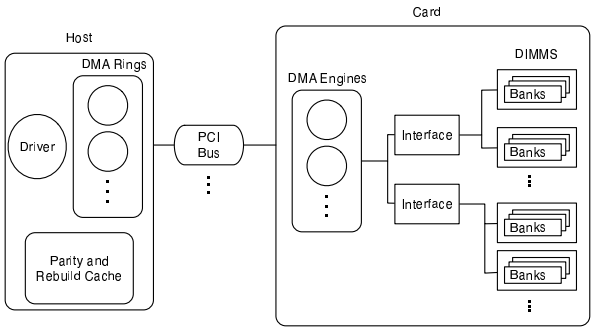


Figure 2: *NAND flash architecture.*

is associated with at most one DMA engine on a card—the pair forms a channel. All commands are issued by a driver at the host, which are queued on one of the host DMA rings. Each DMA ring can only service a single request at a time. Each DMA engine has the ability to send requests to any of the interfaces on the card. Each interface in turn is connected to one or more DIMMs. Each DIMM is connected to one or more banks of flash. The flash banks store the data and are the destination of the request stream. While requests may be parallelized at each interface, the page register requires requests to be serialized at each bank. Each flash bank contains multiple blocks, which can be accessed at page granularity.

The purpose of the driver is to allow a higher level process, such as a file system, to access data pages on the flash devices. The driver maintains the logical to physical page mappings (similar to the flash translation layer), block status and the parity relationship between pages on the devices. The driver exports two functions: read and write. The read function takes the logical page addresses as arguments and returns the requested data. The write function takes a page aligned buffer and length as arguments and returns the logical page addresses.

Each host also contains a cache that is used by the driver to stage parity updates and page rebuild requests. To prevent data loss during power or host failure, we assume the cache may contain non-volatile RAM, such as battery-backed DRAM, MRAM, PRAM or FeRAM. We currently store this cache in DRAM.

Throughout our design, we assume that the system contains C identical cards, each of which contain B banks. Each bank will contain b blocks that have 64 pages of 4K each.

3.2 Erasure Coding Across Devices

An erasure code is made up of codewords that have n symbols. Storage systems typically use systematic erasure codes, where each codeword contains k data symbols and $m = n - k$ parity (redundant) symbols. A code symbol generally refers to a sector or set of sectors on a

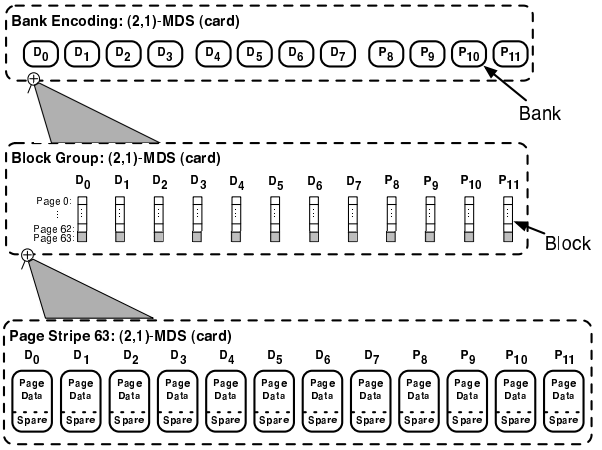


Figure 3: *Example block group.*

single storage device. Storage systems often use a linear, Maximum Distance Separable (MDS) code, denoted (k,m)-MDS, which has m parity symbols, k data symbols, and can correct m erasures, the maximum possible with this many parity symbols. In this sense, an (k,m)-MDS provides optimal storage efficiency. A (k,1)-MDS is usually an XOR parity code, where the one parity symbol is the bitwise XOR of the contents of the data symbols. The main advantage of linear codes in disk-based systems is the ability to calculate new parity symbols when only one data symbol has changed without accessing the other data symbols. Such an update reflects the read-modify-write property of parity-based RAID systems, which is not well suited for similar flash-based systems.

For erasure coding across flash devices, we form a *reliability stripe* out of n pages located in different chips (and *a fortiori* on different blocks). Since write requests to and from a bank are serialized, writing to all pages in a reliability stripe maximizes parallelism. The number of chips, banks per chip, and pages per chips imposes restrictions on the parameters of the erasure correcting code. In general, we want all $C \cdot B \cdot P$ pages be in exactly one reliability group, (unless we contemplate the use of "spare pages" to replace failed pages.) This placement becomes very simple when C divides $k+m$. Defining reliability stripes in our setting is a similar problem to that of laying out a declustered disk array. For example, we want to have an equal proportion of parity and user data on each chip and block in order to balance read load. However, because of the different performance characteristics of disks and flash blocks, these conditions do not need to be stringently enforced.

We show a very simple example encoding in Figure 3. In this example, we have three cards, each with four banks of flash for a total of 12 flash blocks. We use an (2,1)-MDS (similar to 3 disk RAID Level 4), which has relia-

bility stripes consisting of blocks on three different cards. We use the first two cards to store user data and the third to store parity data. In this example, we define reliability stripes for all pages in a block simultaneously. We compute the parities as $P_8 = D_0 \oplus D_4$, $P_9 = D_1 \oplus D_5$, $P_{10} = D_2 \oplus D_6$ and $P_{11} = D_3 \oplus D_7$. This code protects against a single card failure. It involves writing a single parity page with each data page written. As we will describe in more detail below, we use caching in stable memory to minimize actual writes to parity pages.

A bad disk sector in a RAID Level 5 disk array might be discovered during recovery from a failed disk. Since the sector is necessary for reconstructing the contents of the failed disk, the RAID has lost data. In our system, we can protect against the analogous problem by either using an erasure correcting code with two parity symbols corresponding to RAID Level 6, or we can protect against individual page failure by using an erasure code within each block. For instance, we can organize the 64 pages in a block as a $63+1$ reliability stripe. The analogous solution for disks (intra-disk parity) is much less attractive since the disk capacity is much larger than the write cache (so that we have to update parity sectors immediately) and because rotational latency will add to the cost of writing the parity.

In what follows, we restrict ourselves to considering only inter-card codes that place complete blocks into reliability stripes and we consider only the $63+1$ intra-bank code.

3.3 Organizing Pages and Blocks

We use an abstraction for inter-chip reliability stripes, called a *block group*. Figure 3 gives an example block group based on the (2,1)-MDS (card) encoding. If the system contains $C \cdot B$ banks and a bank contains b blocks, then the system has b block groups of size $C \cdot B$. If each bank relatively addresses its blocks as $[0, b-1]$, then the i-th block group is defined as all of the blocks with relative address i from each of the $C \cdot B$ banks.

Each block group has an associated *erasure code instance*. The instance includes a *parity map*, which describes how data is encoded within the block group. Upon initialization, all of the block groups are placed into a queue, where the block group at the head of the queue is called the *current block group*. Similar to a log-structured file system, all writes go to the current block group. Once all of the pages within the current write group have been programmed, the block group is dequeued. While we are writing user data to the current block group, we maintain the parity data in cache. We only write parity data when we dequeue the block. After we dequeue the current block group, the head of the queue becomes the new current block group. We erase the blocks in it before the first write is applied to the group. Thus, the system will only incur the cost of an erase every $\frac{k}{n} \cdot (C \cdot B \cdot 64)$ page writes. Obviously, as the block group queue empties, a cleaner must be invoked to free active block groups. We plan to draw on existing work in log cleaners and leave the design of a block group cleaner to future work.

Block groups are further organized into *page stripes*. Since there are 64 pages in a block, there are 64 page stripes in a block group. As with block groups, the i-th page stripe is defined as all of the pages with a relative address of i in the corresponding block. Writes are applied to page stripes in order from 0 to 63.

Stripe policies determine the order at which data is written to the data pages in an individual page stripe. Currently, the policy is set using a single parameter: *stride*. The i-th write to a page stripe goes to page $i \cdot \text{stride}\%(D-1)$, where D is the number of data pages in the page stripe. A stride of 1 will simply write to the pages in order.

3.4 Changing Encoding On-the-Fly

This organization allows the system to effectively change the level of fault tolerance on a block group basis. For example, if an administrator wishes to perform a system-wide increase in fault tolerance, a new encoding may be chosen for all future block group allocations. Every block group with the old encoding may then be given priority for cleaning, resulting in a slow change in system-wide fault tolerance. Policies for choosing which block groups to clean is an open problem. In practice, changes in the level of fault tolerance would be triggered autonomously based on failure statistics collected similarly to SMART data in current disk drives.

3.5 Updating Parity

All parity updates for a page stripe are staged in the parity cache until all of the dependent data has been written in flash. As an example, assume a page stride of 1 is chosen for the page stripe shown in Figure 3. Each data page within a page stripe is written in order (e.g. $D_0, D_1, \ldots, D_7$) and each page write has an associated parity update (e.g. $P_8 = P_8 \oplus D_0$, $P_9 = P_9 \oplus D_1$, etc.). In this case, P_8 will be staged in cache until the 5-th write to the page stripe has been applied, since P_8 depends on D_0 and D_4. To minimize the time a parity page is staged in cache, the stride may be set to 4. In this case, data page writes are applied in the following order: $D_0, D_4, D_1, D_5, D_2, D_6, D_3, D_7$, allowing the associated parity pages to be hastily flushed from the cache.

Inter-block parity pages are also maintained in the parity cache until all dependent data pages have been added into the parity and written. If a (63,1)-MDS (block) encod-

ing is chosen for each block in Figure 3, then the single parity page in a given block will be flushed from the cache after at most $(62 \cdot 8) + 1 = 435$ data page writes.

3.6 Limitations: Bad Blocks and Scaling

While the block group and page stripe abstractions simplify some of the issues that arise when applying erasure codes to flash, we find these data structures make bad block management and scaling a bit more complicated. One simple solution to the bad block problem is to treat all bad blocks as all-zero data blocks and adjust parity relationships within its parent block group accordingly. Once too many blocks in a single block group have gone bad, the entire block group will be marked as bad. A similar problem arises when incorporating more devices into an existing array. The solution to both of these problems remains open.

4 Evaluation

We evaluate the efficacy of our mechanisms by running microbenchmarks on a flash simulator and estimating the page-error rate under a variety of configurations. Very little is known about the failure rates of whole flash devices or chips, so we leave a device-level reliability estimate to future work. The goal here is to compare the page-error reliability, performance, and space tradeoffs when using our mechanisms in an array of flash devices. While, the results are not intended to entirely prove the effectiveness of these mechanisms in a real system, we believe that these initial results justify further analysis into the reliability mechanisms proposed for arrays of flash devices.

4.1 Setup

All of our measurements were taken on a host computer that is connected to 8 simulated flash cards. Each flash card is simulated using NetApp's Libra card simulator. Each card contains 4 DMA channels and 8 banks of flash. Each bank contains 64 blocks. The access latencies are as follows: 0.2 ms to program, 0.025 ms to read and 1.2 ms to erase. The host is equipped with two 2.74 GHz Pentium 4 processors and 2 GB of RAM. We run microbenchmarks to estimate write and rebuild latency with and without the use of erasure codes. Table 1 lists the erasure code configurations used in our evaluation. The stride in every configuration is set to the number of banks per card so each successive page write is sent to a different card, maximizing parallelism on writes.

4.2 Reliability

We derive the estimated uncorrectable page error rate (UPER) using the cumulative Binomial distribution: $F(k;n,p) = \sum_{i=0}^{k} \binom{n}{i} p^i (1-p)^{n-i}$. The UPER of a page that is encoded with a T bit tolerant ECC is

Code Config	Space Eff.
(7,1)-MDS (card)	87.5%
(6,2)-MDS (card)	75%
(7,1)-MDS (card) + (63,1)-MDS (block)	85.9%

Table 1: *Erasure codes used in our analysis.*

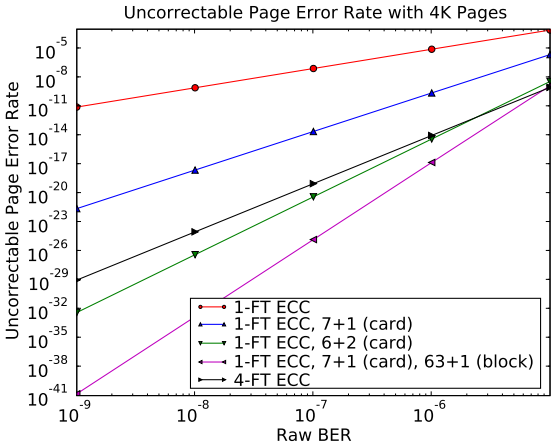


Figure 4: *Estimated Uncorrectable Page Error Rate*

$\text{UPER}_{\text{ECC}}(T) = 1 - F(T;P,\text{RBER})$, where P is the page size in bits and RBER is the raw bit-error rate. Furthermore, given a (k,m)-MDS code, the UPER is computed as $1 - F(m;k+m,\text{UPER}_{\text{ECC}}(T))/(k+m)$. Finally, as long as the corresponding erasure codes only compute a single parity element, the UPER of a two-level encoding with M symbols in the first encoding and N symbols in the second encoding is [†]

$$\frac{\sum_{i=4}^{MN} \beta(M,N,i) \cdot (\text{UPER}_{\text{ECC}}(T))^i \cdot (1-\text{UPER}_{\text{ECC}}(T))^{MN-i}}{MN}$$

Figure 4 illustrates the reliability tradeoffs when using 1 and 4-bit fault tolerant, page-level ECC and 1-bit fault tolerant ECC with the erasure code configurations given in Figure 1. It is important to note that the erasure code configurations not only have the ability to tolerate high level failures (e.g. cards, chips, etc.), they also drastically decrease the UPER. We find that using a 1-bit fault tolerant ECC per page with the (7,1)-MDS (card) + (63,1)-MDS (block) configuration leads to the lowest UPER across the RBER spectrum, because a block group can tolerate the loss of at least 3 pages. The (6,2)-MDS (card) configuration can tolerate many but not all 3 page errors in a block group. Note that if we considered device failures, (6,2)-MDS (card) would have the best reliability.

[†]We calculate $\beta(m,n,i)$ as $\binom{mn}{i} - V_i$, where V_i is an estimate of the number of recoverable erasure patterns of size i [3].

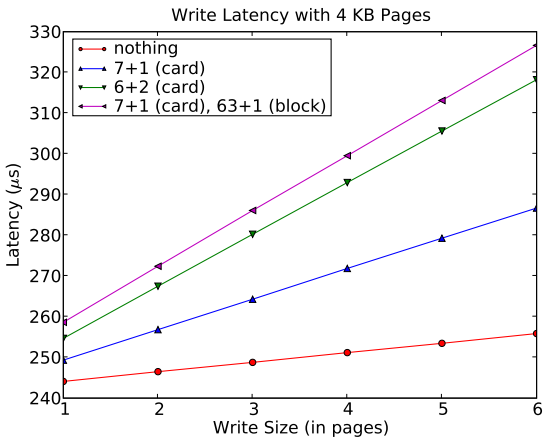


Figure 5: *Write Performance*

4.3 Performance

Figure 5 compares the write latency of no reliability mechanisms and the configurations given in Table 1. To show the effect of parallelism when writing across cards, we varied the size of writes from 1 to 6 pages. We chose the cutoff of 6 because of the contention introduced under this write policy when writing to a new page stripe under the (6,2)-MDS (card) configuration. As expected, the amount of parity written per page determines the performance of each scheme: (7,1)-MDS (card) write 1 parity per page, (6,2)-MDS (card) writes 2 parity per data page and (7,1)-MDS (card) + (63,1)-MDS (block) writes 4 parity per data page.

Another important performance measurement is the page rebuild performance. The time to read a single page is roughly 66.72 μs. Recovering a page under (7,1)-MDS (card) and (6,2)-MDS (card) takes, on average, 100.90 μs and 96.01 μs, respectively. If we try to recover a page from the block parity in the (7,1)-MDS (card) + (63,1)-MDS (block) configuration, a recovery operation takes approximately 4511.12 μs. This type of recovery performs poorly because each page involved in intra-block page recovery must be loaded by the same page register, resulting in 63 serialized reads.

5 Discussion and Open Issues

In this paper, we argue for specific mechanisms that may be used when building highly reliable flash-based storage systems. We also briefly analyzed a few tradeoffs a system designer can make when building such a system: reliability, performance and space efficiency. We found that each choice of erasure code configuration affects all three axes in the tradeoff space. Additionally, as the RBER increases, a reasonable level of reliability can be maintained by adjusting block group encodings on-the-fly.

This work only scratches the surface of flash-based storage system design. In addition to fleshing out how our mechanisms for erasure coding will fit into flash-based systems, there exist many open problems in this area. We must decide how to clean block groups and determine proper cleaning policies when changing erasure code on-the-fly. Additionally, we must figure out how to restructure block groups as new devices are added.

Acknowledgments

We would like to thank our colleagues in the Storage Systems Research Center (SSRC), Kaladhar Voruganti and Garth Goodson, who provided valuable feedback on the ideas in this paper. This research was supported by the Petascale Data Storage Institute, UCSC/LANL Institute for Scalable Scientific Data Management and by SSRC sponsors including Los Alamos National Lab, Livermore National Lab, Sandia National Lab, Digisense, Hewlett-Packard Laboratories, IBM Research, Intel, LSI Logic, Microsoft Research, NetApp, Seagate, Symantec, and Yahoo.

References

[1] N. Agrawal, V. Prabhakaran, T. Wobber, J. Davis, M. Manasse, and R. Panigrahy. Design tradeoffs for ssd performance. In *USENIX Annual Technical Conference*, 2008.

[2] K. R. Fei Sun and T. Zhang. On the use of strong bch codes for improving multilevel nand flash memory storage capacity. Technical report, Rensselaer Polytechnic Institute, 2006.

[3] M. A. Kousa. A novel approach for evlauating the performance of spc product codes under erasure decoding. In *IEEE Transactions on Communications*, 2002.

[4] N. Mielke, T. Marquart, N. Wu, J. Kessenich, H. Belgal, E. Schares, and F. Trivedi. Bit error rate in nand flash memories. In *IEEE International Reliability Physics Symposium*, 2008.

[5] M. W. Storer, K. M. Greenan, E. L. Miller, and K. Voruganti. Pergamum: Replacing tape with energy efficient, reliable, disk-based archival storage. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST)*, Feb. 2008.

[6] F. Sun, S. Devarajan, K. Rose, and T. Zhang. Design of on-chip error correction systems for multilevel nor and nand flash memories. *Circuits, Devices & Systems, IET*, 1(3):241–249, June 2007.

[7] T. M. Taeho Kgil, David Roberts. Improving nand flash based disk caches. In *International Symposium on Computer Architecture*, 2008.

[8] Texas Memory Systems. *An In-depth Look at the RamSan-500 Cached Flash Solid State Disk.*

Published in proceedings of The 2nd Workshop on Hot Topics in Storage
and File Systems (HotStorage '10), June 22, 2010, Boston MA, USA

Mean time to meaningless: MTTDL, Markov models, and storage system reliability

Kevin M. Greenan
ParaScale, Inc.

James S. Plank
University of Tennessee

Jay J. Wylie
HP Labs

Abstract

Mean Time To Data Loss (MTTDL) has been the standard reliability metric in storage systems for more than 20 years. MTTDL represents a simple formula that can be used to compare the reliability of small disk arrays and to perform comparative trending analyses. The MTTDL metric is often misused, with egregious examples relying on the MTTDL to generate reliability estimates that span centuries or millennia. Moving forward, the storage community needs to replace MTTDL with a metric that can be used to accurately compare the reliability of systems in a way that reflects the impact of data loss in the real world.

1 Introduction

"Essentially, all models are wrong, but some are useful"
– George E.P. Box

Since Gibson's original work on RAID [3], the standard metric of storage system reliability has been the *Mean Time To Data Loss* (MTTDL). MTTDL is an estimate of the expected time that it would take a given storage system to exhibit enough failures such that at least one block of data cannot be retrieved or reconstructed.

One of the reasons that MTTDL is so appealing as a metric is that it is easy to construct a Markov model that yields an analytic closed-form equation for MTTDL. Such formulae have been ubiquitous in research and practice due to the ease of estimating reliability by plugging a few numbers into an expression. Given simplistic assumptions about the physical system, such as independent exponential probability distributions for failure and repair, a Markov model can be easily constructed resulting in a nice, closed-form expression.

There are three major problems with using the MTTDL as a measure of storage system reliability. First, the models on which the calculation depends rely on an extremely simplistic view of the storage system. Second, the metric does not reflect the real world, but is often interpreted as a real world estimate. For example, the Pergamum archival storage system estimates a MTTDL of 1400 years [13]. These estimates are based on the assumptions of the underlying Markov models and are typically well beyond the life of any storage system. Finally, MTTDL values tend to be incomparable because each is a function of system scale and omits the (expected) magnitude of data loss.

In this position paper, we argue that MTTDL is a bad reliability metric and that Markov models, the traditional means of determining MTTDL, do a poor job of modeling modern storage system reliability. We then outline properties we believe a good storage system reliability metric should have, and propose a new metric with these properties: NOrmalized Magnitude of Data Loss (NOMDL). Finally, we provide example reliability results using various proposed metrics, including NOMDL, for a simple storage system.

2 The Canonical MTTDL Calculation

In the storage reliability community, the MTTDL is calculated using Continuous-time Markov Chains (a.k.a. Markov model). The canonical Markov model for storage systems is based on RAID4, which tolerates exactly one device failure. Figure 1 shows this model. There are a total of three states. State 0 is the state with all n devices operational. State 1 is the state with one failed device. State 2 is the state with two failed devices, i.e., the data loss state. The model in Figure 1 has two rate parameters: λ, a failure rate and μ, a repair rate. It is assumed that all devices fail at the same rate and repair at the same rate.

At $t = 0$, the system starts pristinely in state 0, and remains in state 0 for an average of $(n \cdot \lambda)^{-1}$ hours (n device failures are exponentially distributed with failure rate λ), when it transitions to state 1. The system is then in state 1 for on average $(((n-1)\cdot\lambda))+\mu)^{-1}$ hours. The system transitions out of state 1 to state 2, which is the data loss state, with probability $\frac{(n-1)\cdot\lambda}{((n-1)\cdot\lambda))+\mu}$. Otherwise, the system transitions back to state 0, where the system

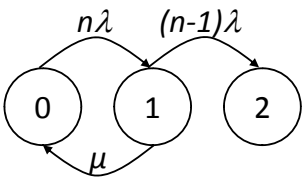


Figure 1: Canonical RAID4/RAID5 Markov model.

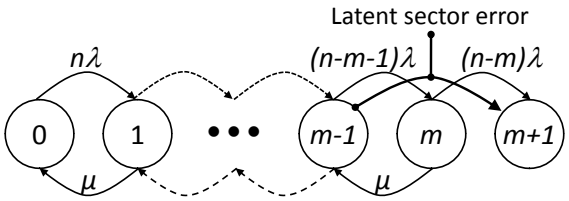


Figure 2: Illustrative multi-disk fault tolerant Markov model with sector errors.

is fully operational and devoid of failures.

The canonical Markov model can be solved analytically for MTTDL, and simplified:

$$MTTDL = \frac{\mu + (2n-1)\lambda}{n(n-1)\lambda^2} \approx \frac{\mu}{n(n-1)\lambda^2}.$$

Such an analytic result is appealing because reassuringly large MTTDL values follow from disk lifetimes measured in hundreds of thousands of hours. Also, the relationship between expected disk lifetime ($1/\lambda$), expected repair time ($1/\mu$), and MTTDL for RAID4 systems is apparent.

3 The MTTDL: Meaningless Values

Siewiorek and Swarz [12] define reliability as follows: "The reliability of a system as a function of time, $R(t)$, is the conditional probability that the system has survived the interval $[0, t]$, given that the system was operational at time $t = 0$." Implicit in the statement that the "system has survived" is that the system is performing its intended function under some operating conditions. For a storage system, this means that the system does not lose data during the expected system lifetime.

An important aspect of the definition of reliability, that is lost in many discussions about storage system reliability, is the notion of *mission lifetime*. MTTDL does not measure reliability directly; it is an expectation based on reliability: $\text{MTTDL} = \int_0^\infty R(t)dt$. MTTDL literally measures the expected time to failure over an infinite interval. This may make the MTTDL useful for quick, relative comparisons, but the absolute measurements are essentially meaningless. For example, an MTTDL measurement of 1400 years tells us very little about the probability of failure during a realistic system mission time. A system designer is most likely interested in the probability and extent of data loss, every year for the first 10 years of a system. The aggregate nature of the MTTDL provides little useful information, and no insight into these calculations.

4 The MTTDL: Unrealistic Models

While Markov Models are appealingly simple, the assumptions that make them convenient also render them inadequate for multi-disk systems In this section, we highlight three ways in which the assumptions significantly do not match modern systems. A more detailed discussion that includes additional concerns may be found in Chapter 4 of Greenan's PhD thesis [4].

For illustration, in Figure 2 we present a Markov Model based on one from [5] that models a n-disk system composed of k disks of data and m disks of parity, in the presence of disk failures and latent sector errors.

4.1 Exponential Distributions

Implicit in the use of Markov models for storage system reliability analysis is the assumption that failure and repair rates follow an exponential distribution and are constant. The exponential distribution is a poor match to observed disk failure rates, latent sector error rates, and disk repairs. Empirical observations have shown that disks do not fail according to independent exponential distributions [2, 6, 10], and that Weibull distributions are more successful in modeling observed disk failure behavior.

Latent sector failures exhibit significant correlation both temporally and spatially within a device [1, 11]. Beyond this, sector failures are highly usage dependent and difficult to quantify [2, 9]. The most recent synthesis of empirical data by Schroeder et al. suggests that Pareto distributions can best capture the burstiness of latent sector errors, as well as spatial and temporal correlations [11].

Disk repair activities such as rebuild and scrubbing tend to require some fixed minimal amount of time to complete. Moreover, in well-engineered systems, there tends to be some fixed upper bound on the time the repair activity may take. Events with lower and upper time bounds are poorly modeled by exponential distributions. Again, Weibull distributions capture reality better [2].

4.2 Memorylessness, Failure & Repair

Exponential distributions and Markov Models are "memoryless." When the system modeled in Figure 2 transitions into a new state, it is as if all the components in the system are reset. Available components' ages are reset to 0 (i.e., brand new), and any repair of failed components is forgotten. Both cases are problematic.

When the system transitions to a new state, it is as if all available disks are refreshed to a "good-as-new" state. In particular, the transition from state 1 to state 0 models a system where the repair of one disk converts all disks into their pristine states. In reality, only the recently repaired component is brand-new, while all the others have a non-zero age.

Now, consider the system under repair in state i such that $1 \leq i < m$. If a failure occurs, moving the system to state $i+1$, any previous rebuilding is assumed to be discarded, and the variable μ governs the transition back to state i. It is as if the *most recent* failure dictates the repair transition. In reality, it is the *earliest* failure, whose rebuild is closest to completion, that governs repair transitions.

These two issues highlight the difficulty with the memorylessness assumption: each transition "forgets" about progress that has been made in a previous state – neither component wear-out nor rebuild progress are modeled. Correctly incorporating these time-dependent properties into such a model is quite difficult. The difficulty lies in the distinction between *absolute time* and *relative time*. Absolute time is the time since the system was generated, while relative time applies to the individual device lifetime and repair clocks. Analytic models operate in absolute time; therefore, there is no reasonable way to determine the values of each individual clock. Simulation methods can track relative time and thus can effectively model reliability of a storage system with time-dependent properties.

4.3 Memorylessness & Sector Errors

In a m-disk fault tolerant system, the storage system enters *critical mode* upon the m-th disk failure. The transition in the Markov model in Figure 2 from the $m-1$ to the $m+1$ state is intended to model data loss due to sector errors in critical mode. In this model, any sector errors or bit errors encountered during rebuild in critical mode lead to data loss. Unfortunately, such a model overestimates the system unreliability. A sector failure only leads to data loss if it occurs in the portion of the failed disk that is critically exposed. For example, in a two-disk fault tolerant system, if the first disk to fail is 90% rebuilt when a second disk fails, only 10% of the disk is critically exposed. Figure 3 illustrates this point in general. This difficulty with Markov models again follows from the memorylessness assumption.

5 Metrics

Many alternatives have been proposed to replace MTTDL as a reliability metric (e.g., [2, 7, 8]). While these metrics have some advantages, none have all the qualities that we believe such a metric must. In our opinion, a storage system reliability metric must have the following properties:

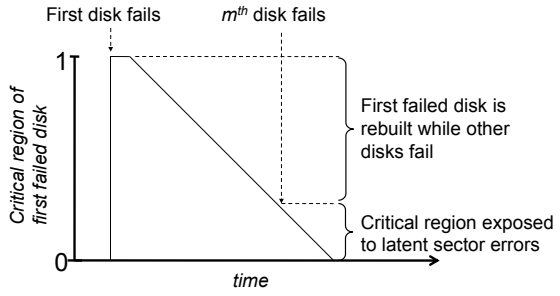


Figure 3: Critical region of first failed disk susceptible to data loss due to latent sector errors.

Calculable. There must exist a reasonable method to calculate the metric. For example, a software package based on well understood simulation or numerical calculation principles can be used.

Meaningful. The metric must relate directly to the reliability of a deployed storage system.

Understandable. The metric, and its units, must be understandable to a broad audience that includes developers, marketers, industry analysts, and researchers.

Comparable. The metric must allow us to compare systems with different scales, architectures, and underlying storage technologies (e.g. solid-state disk).

5.1 NOMDL: A Better Metric?

We start by creating a metric called Magnitude of Data Loss (MDL). Let MDL_t be the expected amount of data lost (in bytes) in a target system within mission time t. There are two advantages to MDL_t. First, like system reliability, the metric deals with arbitrary time periods. This means that a system architect can use MDL_t to estimate the expected number of bytes lost in the first year of deployment, or first ten years of deployment. Second, the units are understandable: bytes and years.

Unfortunately, MDL_t is not a metric that compares well across systems. Consider an 8 disk RAID4 array with intra-disk parity [11] and a 24 disk RAID6 array, both composed of the exact same 1 TB drives. The RAID6 array will result in a higher MDL_t, but has more than 3 times the usable capacity of the RAID4 array.

The MDL can be made comparable by normalizing to the system's usable capacity; doing so yields the NOrmalized Magnitude of Data Loss (NOMDL) metric. NOMDL_t measures expected amount of data lost per usable terabyte within mission time t. For example, the NOMDL_t of a system may be 0.001 bytes lost per usable terabyte in the first 5 years of deployment. Since

NOMDL$_t$ is a normalized version of MDL$_t$, both metrics can be output from the same base calculation.

5.2 Calculating NOMDL$_t$

NOMDL$_t$ is calculable. Markov models can measure the probability of being in a specific state within a mission time. With care, the probabilities can be calculated for all paths in a Markov model and used to derive the number of expected bytes lost. Unfortunately, as we have described above, we do not believe that Markov models accurately capture the behavior of contemporary storage systems. Maybe other modeling paradigms such as Petri Nets and such variants can be used to calculate NOMDL$_t$ while addressing the deficiencies of Markov models.

Our recommendation is to use Monte Carlo simulation to calculate NOMDL$_t$. At a high level, such a simulation can be accomplished as follows. Initially, all devices are assigned failure and repair characteristics. Then, device failures and their repair times are drawn from an appropriate statistical distribution (e.g., Exponential, Weibull, Pareto) for each device. Devices are queued and processed by failure time and the simulation stops at a predefined mission time. Once a device is repaired, another failure and repair time is drawn for that device. Each time a failure occurs, the simulator analyzes the system state to determine if data loss has occurred. Detail of the techniques and overhead associated with simulation is discussed in Greenan's PhD thesis [4].

If the time of data loss is F and the mission time is t, then the simulator implements the function, $I(F < t) = \{0, 1\}$ (0 is no data loss, 1 is data loss). That is, the system either had a data loss event within the mission time or not. Many iterations of the simulator are required to get statistically meaningful results. The standard method of computing system reliability via simulation is to run N iterations (typically chosen experimentally) of the simulator and make the following calculation:

$$R(t) = 1 - \sum_{i=1}^{N} \frac{I(F_i < t)}{N}$$

Since $I(F_i < t)$ evaluates to 1 when there is data loss in iteration i and 0 otherwise, this directly calculates the probability of no data loss in $[0, t]$. Given the magnitude of data loss upon a data loss event, C_i, this standard calculation can produce the MDL$_t$:

$$\text{MDL}_t = \sum_{i=1}^{N} \frac{I(F_i < t) \cdot C_i}{N}.$$

The NOMDL$_t$ is the MDL$_t$ normalized to the usable capacity of the system, D: NOMDL$_t$ = MDL$_t$/D. Using simulation thus produces $\{R(t), \text{MDL}_t, \text{NOMDL}_t\}$ which in our opinion makes NOMDL$_t$ a calculable, meaningful, understandable, and comparable metric.

5.3 Comparison of Metrics

Table 1 provides a high-level comparison of MTTDL and other recently proposed storage reliability metrics. We compare the metrics qualitatively in terms of the aforementioned properties of a good metric. We also perform a sample calculation for each metric of a simple storage system: an 8-disk RAID4 array of terabyte hard drives with periodic scrubbing for a 10 year mission time. The failure/repair/scrub characteristics are taken from Elerath and Pecht [2]. All calculations were performed using the HFRS reliability simulation suite (see §8).

Here we compare MTTDL, *Bit Half-Life* (BHL) [8], *Double-Disk Failures Per 1000 Reliability Groups* (DDF pKRG) [2], *Data Loss events per Petabyte Year* (DALOPY) [5] and NOMDL$_t$. MTTDL and DALOPY are calculated via Markov models. BHL is calculated by finding the time at which a bit has a 0.50 probability of failure, which is difficult to calculate via simulation and can be estimated using a Markov model. For BHL, this time is calculated for the entire system instead of a single bit. DDF pKRG and NOMDL$_t$ are computed using Monte Carlo simulation.

Both calculations for MTTDL and BHL result in reliability metrics that are essentially meaningless. Even in a RAID4 system with the threat of sector errors (2.6% chance when reading an entire disk) both metrics produce numbers that are well beyond the lifetime of most existing systems. In addition, both metrics produce results that are not comparable between systems that differ in terms of technology and scale.

DDF pKRG and DALOPY are interesting alternatives to the original MTTDL calculation. DDF pKRG is not sensitive to technological change, but is bound architecturally to a specific RAID level or erasure-coding scheme (double disk failure is specific to RAID4 or RAID5). DALOPY has most of properties of a good metric, but is not comparable. In particular, it is not comparable across systems based on different technologies or architectures. While DALOPY normalizes the expected number of data loss events to the system size, it does not provide the magnitude of data loss. Without magnitude it is hard to compare DALOPY between systems; data loss event gives no information on what or how much data was lost. In addition, the units of the metric are hard to reason about.

NOMDL$_t$ is not sensitive to technological change, architecture or scale. The metric is normalized to system scale, is comparable between architectures and directly measures the expected magnitude of data loss. As shown in Table 1, the units of NOMDL$_t$—bytes lost per usable TB—are easy to understand. Beyond this, the subscript

	Meaningful	Understandable	Calculable	Comparable	Result
MTTDL			✓		37.60 years
BHL			✓		26.06 years
DDF pKRG	✓	✓	✓		183 DDFs
DALOPY	✓	✓	✓		3.32 DL per (PB*Yr)
NOMDL_{10y}	✓	✓	✓	✓	14.41 bytes lost per usable TB

Table 1: Qualitative comparison of different storage reliability metrics.

$t = 10y$, clearly indicates the mission lifetime and so helps ensure that only numbers based on the same mission lifetime are actually compared.

6 Conclusions

We have argued that MTTDL is essentially a meaningless reliability metric for storage systems and that Markov models, the normal method of calculating MTTDL, is flawed. We are not the first to make this argument (see [2] and [8]) but hope to be the last. We believe NOMDL_t has the desirable features of a good reliability metric, namely that it is calculable, meaningful, understandable, and comparable, and we exhort researchers to exploit it for their future reliability measurements. Currently, we believe that Monte Carlo simulation is the best way to calculate NOMDL_t.

7 Acknowledgments

This material is based upon work supported by the National Science Foundation under grants CNS-0615221.

8 HFRS Availability

The High-Fidelity Reliability (HFR) Simulator is a command line tool written in Python and is available at

`http://users.soe.ucsc.edu/~kmgreen/`.

References

[1] BAIRAVASUNDARAM, L. N., GOODSON, G. R., SCHROEDER, B., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. An Analysis of Data Corruption in the Storage Stack. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST '08)* (San Jose, California, February 2008).

[2] ELERATH, J., AND PECHT, M. Enhanced reliability modeling of raid storage systems. In *Dependable Systems and Networks, 2007. DSN '07. 37th Annual IEEE/IFIP International Conference on* (june 2007), pp. 175–184.

[3] GIBSON, G. A. *Redundant Disk Arrays: Reliable, Parallel Secondary Storage*. PhD thesis, Univeristy of California, Berkeley, December 1990. Technical Report CSD-91-613.

[4] GREENAN, K. M. *Reliability and Power-Efficiency in Erasure-Coded Storage Systems*. PhD thesis, Univeristy of California, Santa Cruz, December 2009. Technical Report UCSC-SSRC-09-08.

[5] HAFNER, J. L., AND RAO, K. Notes on reliability models for non-MDS erasure codes. Tech. Rep. RJ–10391, IBM, October 2006.

[6] PINHEIRO, E., WEBER, W.-D., AND BARROSO, L. A. Failure trends in a large disk drive population. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST '07)* (2007), USENIX Association.

[7] RAO, K. K., HAFNER, J. L., AND GOLDING, R. A. Reliability for networked storage nodes. In *DSN-06: International Conference on Dependable Systems and Networks* (Philadelphia, 2006), IEEE, pp. 237–248.

[8] ROSENTHAL, D. S. H. Bit preservation: A solved problem? In *Proceedings of the Fifth International Conference on Preservation of Digital Objects (iPRES '08)* (London, UK, September 2008).

[9] ROZIER, E., BELLUOMINI, W., DEENADHAYALAN, V., HAFNER, J., RAO, K., AND ZHOU, P. Evaluating the impact of undetected disk errors in raid systems. In *Dependable Systems Networks, 2009. DSN '09. IEEE/IFIP International Conference on* (29 2009-july 2 2009), pp. 83–92.

[10] SCHROEDER, B., AND GIBSON, G. A. Disk failures in the real world: What does an MTTF of 1,000,000 hours mean to you? In *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST '07)* (2007), USENIX Association, pp. 1–16.

[11] SCHROEDER, B., GILL, P., AND DAMOURAS, S. Understanding latent sector errors and how to protect against them. In *Proceedings of the 8th Conference on File and Storage Technologies (FAST '10)* (San Jose, California, February 2010).

[12] SIEWIOREK, D. P., AND SWARZ, R. S. *Reliable computer systems (3rd ed.): design and evaluation*. A. K. Peters, Ltd., Natick, MA, USA, 1998.

[13] STORER, M. W., GREENAN, K., MILLER, E. L., AND VORUGANTI, K. Pergamum: Replacing tape with energy efficient, reliable, disk-based archival storage. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST '08)* (Feb. 2008), pp. 1–16.

Optimizing Galois Field Arithmetic for Diverse Processor Architectures and Applications

Kevin M. Greenan
Univ. of California, Santa Cruz
kmgreen@cs.ucsc.edu

Ethan L. Miller
Univ. of California, Santa Cruz
elm@cs.ucsc.edu

Thomas J. E. Schwarz, S.J.
Santa Clara University
tjschwarz@scu.edu

Abstract

Galois field implementations are central to the design of many reliable and secure systems, with many systems implementing them in software. The two most common Galois field operations are addition and multiplication; typically, multiplication is far more expensive than addition. In software, multiplication is generally done with a look-up to a pre-computed table, limiting the size of the field and resulting in uneven performance across architectures and applications.

In this paper, we first analyze existing table-based implementation and optimization techniques for multiplication in fields of the form $GF(2^l)$. Next, we propose the use of techniques in composite fields*: extensions of $GF(2^l)$ in which multiplications are performed in $GF(2^l)$ and efficiently combined. The composite field technique trades computation for storage space, which prevents eviction of look-up tables from the CPU cache and allows for arbitrarily large fields. Most Galois field optimizations are specific to a particular implementation; our technique is general and may be applied in any scenario requiring Galois fields. A detailed performance study across five architectures shows that the relative performance of each approach varies with architecture, and that CPU, memory limitations and fields size must be considered when selecting an appropriate Galois field implementation. We also find that the use of our composite field implementation is often faster and less memory intensive than traditional algorithms for $GF(2^l)$.*

1. Introduction

The use of Galois fields of the form $GF(2^l)$, called *binary extension fields*, is ubiquitous in a variety of areas ranging from cryptography to storage system reliability. These algebraic structures are used to compute erasure encoded symbols, evaluate and interpolate polynomials in Shamir's secret sharing algorithm [22], compute algebraic signatures over variable-length strings of symbols [21], and encrypt blocks of data in the current NIST advanced encryption standard [14]. Current memory, CPU cache sizes and preferred approaches limit most applications to performing computation in either $GF(2^8)$ or $GF(2^{16})$. The goal of our research is to study the multiplication performance of these common fields, propose an alternate representation for arbitrary-sized fields and compare performance across all representations on different CPU architectures and for different workloads.

Multiplication in $GF(2^l)$ is usually done using precomputed look-up tables, while addition of two elements in $GF(2^l)$ is usually, but not always, carried out using an inexpensive bitwise-XOR of the elements. As a result, multiplication has the greatest effect on overall algorithm performance because a table look-up is more expensive than bit-wise XOR. Due to the restrictions of look-up tables, elements in the implemented field are almost always smaller than a computer word, while bit-wise XOR operates over words by definition. In many applications, the multiplication operation is used just as often as addition; thus, optimizing the multiplication operation will, in turn, lead to much more efficient applications of Galois fields.

The byte-based nature of computer memory motivates the use of $GF(2^8)$: each element represents one byte of storage. This field only has 256 elements, which results in small multiplication tables; however, use of $GF(2^8)$, for example, restricts the size of a Reed-Solomon codeword to no more than 257 elements [13]. The smallest feasible field larger than $GF(2^8)$ is $GF(2^{16})$. Growing to $GF(2^{16})$ and beyond has a significant impact on multiplication and other field operations. A complete lookup table for multiplication in $GF(2^{16})$ requires 8 GB—well beyond the memory capacity of most systems. The standard alternative to a full multiplication table is a logarithm and an antilogarithm table requiring 256 KB of memory, which may fit in the standard L2 cache but will likely be only partially resident in an L1 cache. Using a straightforward table-based multiplication approaches to match word size in a 32-bit system (e.g. $GF(2^{32})$) is impossible, given modern memory sizes.

This paper appeared in the *Proceedings of the 16th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS 2008)*, Baltimore, MD, September 2008.

Composite fields are an alternative to traditional table-based methods. Using this technique, elements of a field $GF(2^n)$ are represented in terms of elements in a sub-field $GF(2^l)$, where $n = l \times k$. The composite field $GF((2^l)^k)$ is a representation of a *k-degree extension* of $GF(2^l)$, where $GF(2^l)$ is called the *ground field*. This technique trades additional computation for a significant decrease in storage space relative to traditional table-based methods. Since the cost of a cache miss is comparable to that of the execution of many instructions, trading additional computation for lower storage requirements can significantly increase performance on modern processors. Additionally, many applications will use Galois fields for different purposes; using the composite field technique, it may be possible to reuse the the ground Galois field tables for several variable-sized Galois field extensions.

The performance of the Galois field implementation is a critical factor in overall system performance. The results presented here show dramatic differences in throughput, but there is no overall algorithmic winner on the various platforms we studied. We conclude that a performance-optimizing software suite needs to tune the Galois field implementations to architecture, CPU speed and cache size. In this paper, we restrict our investigation to the efficient implementation of operations in $GF(2^4)$, $GF(2^8)$, $GF(2^{16})$, and $GF(2^{32})$, postponing an evaluation of implementations for larger size fields for future work.

The contributions of this paper are threefold. First, we present and compare popular table-based binary extension field implementation techniques and some optimizations. Next, we propose the use of software-based composite fields when implementing $GF(2^{16})$ and $GF(2^{32})$. Finally, we show that the performance of different implementations of Galois fields is highly dependent on underlying hardware and workload, and suggest techniques better optimized for distinct architectures. Unlike many hardware and software Galois field optimizations, all of the techniques described are general and may be applied to any application requiring Galois field arithmetic.

2. Applications of Galois Fields

The use of erasure codes in disk arrays, distributed storage systems and content distribution systems has been a common area of research within the systems community over the past few years. Most work is concerned with fault tolerant properties of codes, performance implications of codes, or both. Many of the erasure codes used in storage systems are XOR-based and generally provide limited levels of fault tolerance; a flood of special-purpose, XOR-based codes is the result of a performance-oriented push from the systems community [5, 3, 24]. While these codes perform all encoding and decoding using the XOR operator, they either lack flexibility in the number of tolerated failures or are not maximum distance separable (MDS) and may require additional program complexity.

Linear erasure codes, such as Reed-Solomon [17], are MDS. As a result, Reed-Solomon codes provide flexibility and optimal storage efficiency. A Reed-Solomon codeword is computed by generating m parity symbols from k data symbols such that any m erased symbols may be recovered. Each parity symbol is generated using k Galois field multiplications and $k-1$ additions. The required size of the underlying Galois field is bound by the number of parity and data symbols in a codeword. Linear erasure codes are also used in network coding to maximize information flow in a network, where linear combinations of symbols are computed (mixed) at intermediate nodes within a network [1]. The field size is typically bound by network size and connectivity.

Threshold cryptography algorithms, such as Shamir's secret sharing algorithm [22], also rely on Galois fields for encoding and decoding. The algorithm chooses a random k-degree polynomial over a Galois field; the zero-th coefficient is the secret to be shared among n participants. The polynomial is evaluated over n coordinates (shares), distributed among the participants. Polynomial interpolation is used to reconstruct the zero-th coefficient from any $k+1$ unique shares. The construction, evaluation and interpolation of the polynomial may also be done over Z_p for some prime number p. Unfortunately, when dealing with large fields, the use of a suitable prime number may result in field elements that are not byte-aligned. Using Galois fields allows all of the field elements to be byte aligned.

Another class of algorithms that use Galois field arithmetic is algebraic signatures [21]. Algebraic signatures are *Rabinesque* because of the similarity between signature calculation and the hash function used in the Rabin-Karp string matching algorithm [6]. The algebraic signature of a string $s_0, s_1, \ldots, s_{n-1}$ is the sum $\sum_{i=0}^{n-1} s_i \alpha^i$, where α and the elements of the string are members of the same Galois field. Algebraic signatures are typically used across RAID stripes, where the signature of a parity disk equals the parity of the signatures of the data disks. This property makes the signatures well-suited for efficient, remote data verification and data integrity in distributed storage systems.

While the use of fields larger than $GF(2^8)$ is generally considered overkill, there are practical instances where large fields are required. As long as there are at most 257 symbols in a codeword, Reed-Solomon can be implemented with $GF(2^8)$. Once the number of symbols exceeds this bound, a larger field must be used. For example, the disaster recovery codes described in [9] may require thousands of symbols per codeword; thus, a larger field such as $GF(2^{16})$ must be used. Algebraic signatures have a similar restriction. In this case case, the length of the string is lim-

ited by the size of the underlying field and in many cases a field larger than $GF(2^8)$ is required.

All of these applications make extensive use of Galois field multiplication, which is generally second to disk access as a performance bottleneck in a storage system that uses Galois fields. However, as storage systems begin to use non-volatile memories [8], Galois field performance may begin to dominate storage and retrieval time. We describe methods aimed at improving general multiplication performance in the next two sections.

3. Construction of $GF(2^l)$

The field $GF(2^l)$ is defined by a set of 2^l unique elements that is closed under both addition and multiplication, in which every non-zero element has a multiplicative inverse and every element has an additive inverse. As with any field, addition and multiplication are associative, distributive and commutative [12]. The Galois field $GF(2^l)$ may be represented by the set of all polynomials of degree at most $l-1$, with coefficients from the binary field $GF(2)$—the field defined over the set of elements 0 and 1. Thus, the 4-bit field element $a = 0111$ has the polynomial representation $a(x) = x^2 + x + 1$.

In contrast to finite fields defined over an integer prime, the field $GF(2^l)$ is defined over an *irreducible polynomial* of degree l with coefficients in $GF(2)$. An irreducible polynomial is analogous to a prime number in that it cannot be factored into two non-trivial factors. Addition and subtraction in $GF(2)$ is done with the bitwise XOR operator, and multiplication is the bitwise AND operator. It follows that addition and subtraction in $GF(2^l)$ are also carried out using the bitwise XOR operator; however, multiplication is more complicated. In order to multiply two elements $a, b \in GF(2^l)$, we perform polynomial multiplication of $a(x) \cdot b(x)$ and reduce the product modulo an l-degree irreducible polynomial over $GF(2)$. Division among field elements is computed in a similar fashion using polynomial division. The *order* of a non-zero field element α, $\text{ord}(\alpha)$, is the smallest positive i such that $\alpha^i = 1$. If the order of an element $\alpha \in GF(2^l)$ is $2^l - 1$, then α is *primitive*. In this case, α generates $GF(2^l)$, *i. e.*, all non-zero elements of $GF(2^l)$ are powers of α.

This section describes several approaches to performing multiplication over the fields $GF(2^4)$, $GF(2^8)$, and $GF(2^{16})$ and presents several optimizations. All of the methods described here may be used to perform ground field calculations in the composite field representation.

3.1. Multiplication in $GF(2^l)$

Courses in algebra often define Galois fields as a set of polynomials over a prime field such as $\{0,1\}$ modulo a generator polynomial. While it is possible to calculate in Galois fields performing polynomial multiplication and reduction modulo the generator polynomial, doing so is rarely efficient. Instead, we can make extensive use of precomputed lookup tables. For small Galois fields, it is possible to calculate all possible products between the field elements and store the result in a (full) look-up table. However, this method consumes large amounts of memory—$O(n^2)$ for fields of size n.

Input : $a, b \in GF(2^l)$ and $FLD\_SIZE = 2^l$
if a is 0 OR b is 0 **then**
  return 0
end if
$sum \leftarrow log[a] + log[b]$
if $sum \geq FLD\_SIZE - 1$ **then**
  $sum \leftarrow sum - FLD\_SIZE - 1$
end if
return $antilog[sum]$

Figure 1: Computing the product of a and b using log and antilog tables

Log/antilog tables make up for storage inefficiency by requiring some computation and extra lookups in addition to the single lookup required for a multiplication table. The method requires $O(n)$ space for fields of size n and is based on the existence of a primitive element α. Every non-zero field element $\beta \in GF(2^l)$ is a power $\beta = \alpha^i$ where the *logarithm* is uniquely determined modulo $2^l - 1$. We write $i = \log(\beta)$ and $\beta = \text{antilog}(i)$. As shown in Figure 1, the product of two non-zero elements $a, b \in GF(2^l)$ can be computed as $a \cdot b = \text{antilog}(\log(a) + \log(b)) \mod 2^l - 1$.

3.2. Optimization of the Full Multiplication Table

The size of the full multiplication table is reduced by breaking a multiplier in $GF(2^l)$ into a left and a right part. The result of multiplication by a left and by a right part is stored in two tables, resulting in a significantly smaller multiplication table. Multiplication is performed using a lookup into both tables and an addition to calculate the correct product. Other papers have called this optimization a "double table" [21]; we call it a *left-right table*. To define it formally, we represent the elements of $GF(2^l)$ as polynomials of degree up to $l-1$ over $\{0,1\}$. If we wish to multiply field elements $a(x) = a_1 + a_2 x + \cdots + a_{l-1}x^{l-1}$ and $b(x) = b_1 + b_2 x + \cdots + b_{l-1}x^{l-1}$ in $GF(2^l)$, the product $a(x) \cdot b(x)$ can be arranged into two products and a sum

$$\begin{aligned} & (a_1 + \cdots + a_{l-1}x^{l-1}) \cdot (b_1 + \cdots + b_{l-1}x^{l-1}) \\ = \; & ((a_1 + \cdots + a_{\frac{l}{2}-1}x^{\frac{l}{2}-1}) \cdot (b_1 + \cdots + b_{l-1}x^{l-1})) \\ + \; & ((a_{\frac{l}{2}}x^{\frac{l}{2}} + \cdots + a_{l-1}x^{l-1}) \cdot (b_1 + \cdots + b_{l-1}x^{l-1})). \end{aligned}$$

By breaking the result into two separate products, we can construct two tables having $2^{l/2} \cdot 2^l$ entries each, assuming l is even. This approach, which computes the product of two elements using two lookups, a bit-

Input : $FLD\_SIZE == 2^l$
for $i = 0$ to $FLD\_SIZE \gg \frac{l}{2}$ **do**
  for $j = 0$ to FLD_SIZE **do**
    $mult\_tbl\_left[i][j] \leftarrow gf\_mult(i \ll \frac{l}{2}, j)$
    $mult\_tbl\_right[i][j] \leftarrow gf\_mult(i, j)$
  end for
end for

Figure 2: Precomputing products for the left and right multiplication tables

wise shift, two bitwise ANDs and a bitwise XOR, requires that tables be generated as shown in Figure 2. The product $a \cdot b$, $a, b \in GF(2^l)$ is the sum of $mult\_tbl\_left[a_1 >> l/2][b]$ and $mult\_tbl\_right[a_0][b]$, where a_1 are the $\frac{l}{2}$ most significant bits and a_0 are the $\frac{l}{2}$ least significant bits.

This multiplication table optimization is highly effective for $GF(2^8)$ and $GF(2^{16})$. The standard multiplication table for $GF(2^8)$ requires 64 KB, which typically fits in the L2 cache, but might not fit in the data section of an L1 cache. Using the multiplication table optimization, the table for $GF(2^8)$ is 8 KB, and is much more likely to be fully resident in the L1 cache. The table for $GF(2^{16})$ occupies 8 GB and will not fit in main memory in a majority of systems; however, the optimization shrinks the table from 8 GB to 66 MB, which has a much better chance of fitting into main memory.

3.3. Optimization of the Log/Antilog Method

While our previous optimization traded an additional calculation for space savings, the optimizations for the log/antilog method go in the opposite direction. As shown in Figure 1, the standard log/antilog multiplication algorithm requires two checks for zero, three table-lookups and an addition modulo $2^l - 1$. We can divide two numbers by taking the antilogarithm of the difference between the two logarithms, but this difference has to be taken modulo $2^l - 1$ as well. We can avoid the cumbersome reduction modulo $2^l - 1$ by observing that the logarithm is defined to be a number between 0 and $2^l - 2$, so the sum of two logarithms can be at most $2 \cdot 2^l - 4$ and the difference between two logarithms is larger or equal to $-2^l + 1$. By extending our antilogarithm table to indices between $-2^l + 1$ and $2^l - 2$, our log/antilog multiplication implementation has replaced the addition/subtraction modulo $2^l - 1$ with a normal signed integer addition/subtraction.

If division is a rare operation, we can use an insight by A. Broder [personal communication from Mark Manasse] to speed up multiplication by avoiding the check for the factors being zero. Although the logarithm of zero is undefined, if the logarithm of zero is defined to be a negative number small enough that any addition with a true logarithm still yields a negative number, no explicit zero check is needed. Thus, we set $\log(0) = -2^l$ and then define the antilog of a negative number to be 0. As a result of this redefinition, the antilog table now has to accommodate indices between -2^{l+1} and $2^{l+1} - 2$ and has quintupled in size, but the product of a and b may now be calculated simply as $a \cdot b = \text{antilog}[\log[a] + \log[b]]$. We call this approach the optimized logarithm/antilogarithm or Broder's scheme.

Technique	Space	Complexity
Mult. Table	$2^l \cdot 2^l$	1 LOOK
Log/Antilog	$2^l + 2^l$	3 LOOK, 2 BR, 1 MOD 1 ADD
Log/Antilog Optimized	$5 \cdot 2^l$	3 LOOK, 1 ADD
Huang and Xu	$2^l + 2^l$	3 LOOK, 1 BR, 3 ADD 1 SHIFT, 1 AND
LR Mult. Table	$2^{(3l/2)+1}$	2 LOOK, 2 AND 1 XOR, 1 SHIFT

Table 1: Ground field memory requirements and computation complexity of multiplication in $GF(2^l)$. The operations are abbreviated LOOK for table lookup, BR for branch and MOD for modulus; the rest refer to addition and the corresponding bitwise operations.

Huang and Xu proposed three improvements to the log/antilog approach [11]. The improvements were compared to the full multiplication table and the unoptimized logarithm/antilogarithm approaches in $GF(2^8)$. The first two improvements optimize the modular reduction operation out and maintain the conditional check for zero, while the third improvement is Broder's scheme. The first improvement replaces the modulus operator by computing the product of two non-zero field elements as

$$\text{antilog}[(\log[a] + \log[b]) \& (2^n - 1) + (\log[a] + \log[b]) \gg n].$$

Due to the similarity between the second and third improvements in Huang and Xu [11] and Broder's scheme, we chose to only include the first improvement (called Huang and Xu) for comparison in our study.

This section has described a variety of techniques and optimizations for multiplication in $GF(2^l)$; Table 1 lists the space and computation requirements for each approach. In the next section, we show that the multiplication techniques in $GF(2^l)$ can be used to efficiently compute products over the field $GF((2^l)^k)$.

4. Using Composite Fields

Many hardware implementations of Galois fields use specialized composite field techniques [15, 16], in which multiplication in a large Galois field is implemented in terms of a smaller Galois field. While only using fields with sizes that are a power of 2, we want to implement multiplication and division in $GF(2^n)$ in terms of $GF(2^l)$, where $n = l \cdot k$. Galois field theory states that $GF(2^n)$ is isomorphic to an extension field of $GF(2^l)$ generated by an irreducible polynomial $f(x)$, of degree k with coefficients in $GF(2^l)$.

In this implementation, elements of $GF(2^n)$, written $GF((2^l)^k)$, are polynomials of degree up to $k-1$ with coefficients in $GF(2^l)$. In our standard representation, each element of $GF((2^l)^k)$ is a bit string of length n, which we now break into k consecutive strings of length l each. For example, if $n = 32$ and $k = 4$, a bit string of length 32 is broken into four pieces of length 8. If the result is (a_3, a_2, a_1, a_0), we identify the $GF(2^{32})$ element with the polynomial $a_3 \cdot x^3 + a_2 \cdot x^2 + a_1 \cdot x + a_0$, with coefficients $a_3, a_2, a_1, a_0 \in GF(2^8)$. This particular representation is denoted $GF((2^8)^4)$.

The product of two $GF((2^l)^k)$ elements is obtained by multiplying the corresponding polynomials $a_{k-1} \cdot x^{k-1} + \ldots + a_2 \cdot x^2 + a_1 \cdot x + a_0$ and $b_{k-1} \cdot x^{k-1} + \ldots + b_2 \cdot x^2 + b_1 \cdot x + b_0$ and reducing the result modulo the irreducible, defining polynomial $f(x)$. Their product is

$$\sum_{i=0}^{2(k-1)} \left(\sum_{\nu+\mu=i} a_\nu \cdot b_\mu \right) x^i$$

For $i > k-1$ in this expression, we replace x^i with $x^i \bmod f(x)$, perform the multiplication, and reorganize by powers of x^i. The result is the product in terms of products of the coefficients of the two factor polynomials multiplied with coefficients of the defining polynomial $f(x)$. In order to do this efficiently, we must search for irreducible polynomials $f(x)$ of degree k over $GF(2^l)$ that have many coefficients equal to zero or to one.

For small field sizes, it is possible to exhaustively search for irreducible polynomials. If $k \leq 3$, an irreducible polynomial is one that has no root (*i. e.*, α such that $f(\alpha) = 0$) and irreducibility testing is simple. Otherwise, the Ben-Or algorithm [7] is an efficient way to find irreducible polynomials.

We have implemented multiplication in $GF((2^l)^2)$, for $l \in \{4, 8, 16\}$ and $GF((2^l)^4)$, for $l \in \{4, 8\}$ using the composite field representation, as described in the remainder of this section We have developed a similar approach for computing the inverse of an element. The composite field inversion techniques and their performance are discussed in the full version of this paper [10].

4.1. Multiplication in $GF((2^l)^2)$

In general, irreducible polynomials over $GF(2^n)$ of degree two must have a linear coefficient. We have found irreducible polynomials of the form $f(x) = x^2 + s \cdot x + 1$ over $GF(2^4)$, $GF(2^8)$, and $GF(2^{16})$ that are well-suited for our purpose. We write any element of $GF((2^l)^2)$ as a linear or constant polynomial over $GF(2^l)$. Multiplying $a_1 \cdot x + a_0$ by $b_1 \cdot x + b_0$ gives the product

$$\begin{aligned} & (a_1 \cdot x + a_0) \cdot (b_1 \cdot x + b_0) \\ = \; & a_1 b_1 x^2 + (a_1 b_0 + a_0 b_1) x + a_0 b_0. \end{aligned}$$

Since $x^2 = sx + 1 \mod f(x)$, this becomes

$$(a_1 b_0 + a_0 b_1 + s a_1 b_1) x + (a_1 b_1 + a_0 b_0).$$

As described above, multiplication in $GF((2^l)^2)$ is done in terms of five multiplications in $GF(2^l)$, of which one is done with a constant element s. If we define $GF(2^8)$ through the binary polynomial $x^8 + x^4 + x^3 + x^2 + 1$, or `0x11D` in hexadecimal notation, we can choose s to be `0x3F`, resulting in an irreducible polynomial that is optimal in the number of resulting multiplications. An irreducible, quadratic polynomial must have three non-zero coefficients since, without a constant coefficient the polynomial has root zero and a polynomial of form $x^2 + a$ always has the square root of a as a root. Since $x^2 + x + 1$ is not irreducible over $GF(2^4)$, $GF(2^8)$ or $GF(2^{16})$, we can do no better than $x^2 + s \cdot x + 1$. The fields $GF((2^4)^2)$, $GF((2^8)^2)$ and $GF((2^{16})^2)$ were implemented in this manner.

4.2. Multiplication in $GF((2^l)^4)$

We can use the composite field technique in two ways to implement $GF((2^l)^4)$. First, we can implement $GF(2^8)$ and $GF(2^{16})$ as $GF((2^4)^2)$ and $GF((2^8)^2)$, respectively, and then implement $GF(2^{32})$ as $GF(((2^8)^2)^2)$. This approach would require that we find an irreducible polynomial over $GF((2^8)^2)$; fortunately, there is one of the same form as in the previous section. *Mutatis mutandis*, our multiplication formula remains valid and we have implemented multiplication in $GF(2^{32})$ using $5 \cdot 5 = 25$ multiplications in $GF(2^8)$. The same approach applies to multiplication in $GF(2^{16})$ over coefficients in $GF((2^4)^2)$.

We can also use a single step to implement $GF(2^{32})$ in terms of $GF(2^8)$, but finding an appropriate irreducible polynomial of degree 4 in $GF(2^8)$ is more involved. After exhaustive searching, we determined that we can do no better than $x^4 + x^2 + sx + t$ ($s, t \notin \{0, 1\}$), for which the resulting implementation uses only 22 multiplications, 16 of which result from multiplying all coefficients with each other and the remaining 6 from multiplying s and t by $a_3 b_3$, $a_3 b_2 + a_2 b_3$, and by $a_3 b_1 + a_2 b_2 + a_1 b_3$, reusing some of the results. For instance, we have an addend of $a_3 b_3 (t+1) x^2$ and of $a_3 b_3 t$, but we can calculate both of them with a single multiplication by t. Again, the same formula works for the $GF((2^4)^4)$ representation of $GF(2^{16})$.

5. Experimental Evaluation

We have written a Galois field library that implements $GF(2^4)$, $GF(2^8)$,$GF(2^{16})$ and $GF(2^{32})$ using the approaches presented in Sections 3 and 4. The core library contains code for finding irreducible polynomials, polynomial operations, and the arithmetic operations over the supported fields. Instances of Shamir's secret sharing, Reed-Solomon and algebraic signatures were also created on top

Processor	L1(data)/L2	Memory
2.4 GHz AMD Opteron	64 KB/1 MB	1 GB
1.33 GHz PowerPC G4	32 KB/512 KB	768 MB
2 GHz Intel Core Duo	32 KB/2 MB[1]	2 GB
2 GHz Intel Pentium 4 M	8 KB/512 KB	1 GB
400 MHz ARM9	32 KB/—	128 MB

Table 2: List of the processors used in our evaluation.

of the library. The entire implementation was written in C and contains roughly 3,000 lines of code. This code will be made available prior to publication as a library that implements Galois fields and the aforementioned applications.

Our experimental evaluation measures the speed of multiplications as well as the throughput of three "higher-level" applications of Galois fields: Shamir secret sharing, Reed Solomon encoding and algebraic signatures. We took our measurements on five machines, whose specifications are listed in Table 2. The remainder of this section focuses on 8 key observations found when studying the effect of Galois field implementation on performance. Due to space, we only present a subset of the results in this section. For more detailed performance numbers, please refer to [10].

5.1. Performance using $GF(2^l)$

Figures 3(a)–3(d) compare multiplication throughput using the table-based techniques discussed in Section 3 over the fields $GF(2^4)$, $GF(2^8)$ and $GF(2^{16})$. These techniques are the full multiplication table (**tbl**), left-right table, (**lr_tbl**), optimized logarithm/ antilogarithm method (**lg**), the optimization chosen from [11] (**huang_lg**), and the unoptimized version of logarithm/antilogarithm (**lg_orig**).

Three distinct workloads were run on four of the architectures. Although they are artificial, the workloads embody typical operations in Galois fields. The UNIFORM workload represents the average-case by computing the product of a randomly-chosen field element (drawn from an array) and a monotonically-increasing value masked to fit the value of a field element. The CONSTANT workload computes the product of a constant value and a randomly chosen field element. The SQUARE workload squares a randomly chosen element, then squares the result and so forth. We expected the UNIFORM workload to essentially use the entire look-up table, while CONSTANT and SQUARE typically utilize only a subset of a table.

Observation #1 : *Workload determines look-up table access pattern, which affects performance.*

As expected, the UNIFORM data set has lower throughput than the other workloads, for two reasons. First, UNIFORM draws random values from an array, competing with the look-up tables for space in the cache. Second, the uniform workload computes the product of two distinct elements at each step, which has a dramatic effect when caching large tables. Unlike UNIFORM , the CONSTANT and SQUARE workloads only access a subset of the tables, which results in reduced cache competition and higher throughput. Our results show that one must be attentive when measuring table-based multiplication performance, since look-up table access patterns have a dramatic effect on performance.

Observation #2 : *Cache size affects performance.*

In addition to cache competition, the actual size of the look-up table relative to cache size also greatly affects performance. For example, the full multiplication table for $GF(2^8)$ generally performs worse than any other algorithm for the average-case workload, since a full multiplication table requires at least 64 KB, and thus does not fit in the L1 cache of most processors.

Observation #3 : *Performance decreases as the cache-resident portion of the table decreases.*

SQUARE and CONSTANT perform much better than UNIFORM when the table size is quite large, since a smaller fraction of the multiplication table is required in cache. When the table size is relatively small, the performance of SQUARE and CONSTANT is closer to UNIFORM , since a larger fraction of the multiplication table will be kept in cache. In addition, any extra computation is overshadowed by cache-related issues in the UNIFORM workload; extra computation in addition to table look-ups has little effect on average-case performance.

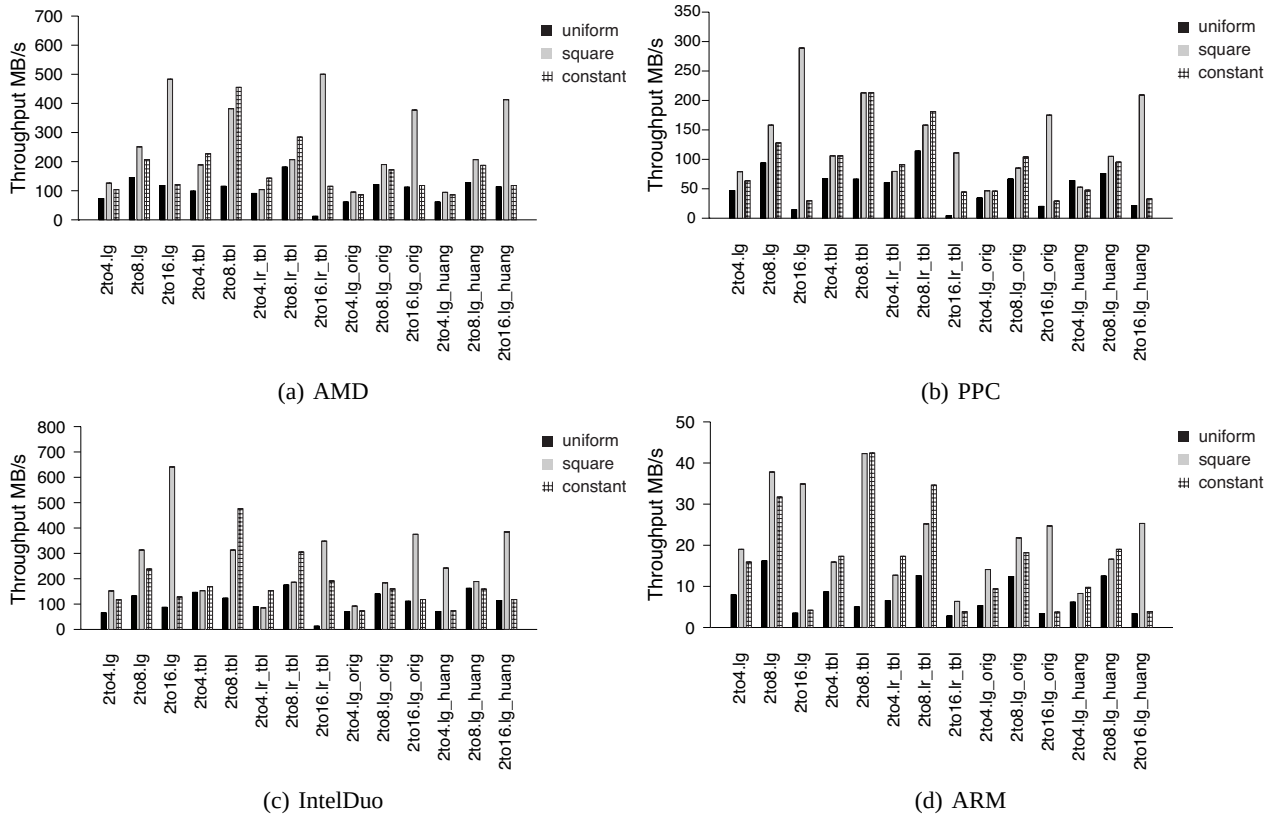
Overall, we found that the left-right implementation of $GF(2^8)$ appears to have the best performance for the UNIFORM workload across three of the four architectures. This optimization appears to provide the best combination of table size and computation for these architectures. We believe that the optimized logarithm/antilogarithm approach for $GF(2^8)$ performs best on ARM due to the computational constraints on that processor and the lack of an L2 cache.

5.2. Performance using $GF((2^l)^k)$

Figure 5 shows the normalized multiplication throughput (in MB/s) of the composite field technique and the traditional look-up table techniques for $GF(2^{16})$ and $GF(2^{32})$ over the UNIFORM workload. In general, the logarithm/antilogarithm approaches performed the best across both composite field and traditional look-up table techniques, thus we omit the other approaches.

Figures 4(b)–4(d) show the multiplication performance in three typical applications: Reed-Solomon encoding, Shamir secret sharing and algebraic signature computation. We present the results for Reed-Solomon encoding with 62 data elements and 2 parity elements in two scenarios. The first reflects basic codeword encoding, in which each symbol in the codeword is an element of the appropriate Galois

[1] Each core has a private 32 KB L1 cache. The L2 cache is shared between the cores.



(a) AMD

(b) PPC

(c) IntelDuo

(d) ARM

Figure 3: Throughput of ground fields using multiplication tables, lg/anilg tables and LR tables. The labels on the x-axis are given as *field.multmethod*, where *field* is the underlying field and *multmethod* is the multiplication algorithm.

field. The second method, called *region multiplication*, performs codeword encoding over 16K symbols, resulting in 16K consecutive multiplications by the same field element. We also perform a $(3,2)$ Shamir secret split, which evaluates a random 2-degree polynomial over each field for the values 1, 2 and 3. In algebraic signatures, the signature is computed by $\{sig_{\alpha^0}(D),\ sig_{\alpha}(D)\}$ and $sig_{\beta}(D) = \sum_{i=0}^{l} d_i \beta^i$ and $|D| = l$ over 4K symbol blocks. Both Shamir and algebraic signatures use Horner's scheme for polynomial evaluation [6], so most of the multiplications are reminiscent of CONSTANT .

We explored an interesting optimization specific to the composite field representation when computing algebraic signatures. In addition to using the multiplication table technique shown in [21], we hand-picked α for the composite field implementations. For instance, if we choose $\alpha = \texttt{0x0101} \in GF((2^8)^2)$, then multiplication by α results in two multiplications by 1 in $GF(2^8)$, which can be optimized out as two copy operations. Note that α is chosen such that $\text{ord}(\alpha) \gg b$, where b is the size of the data blocks. This optimization could also be applied to a Reed-Solomon and Shamir implementation; for brevity, we omit the optimizations.

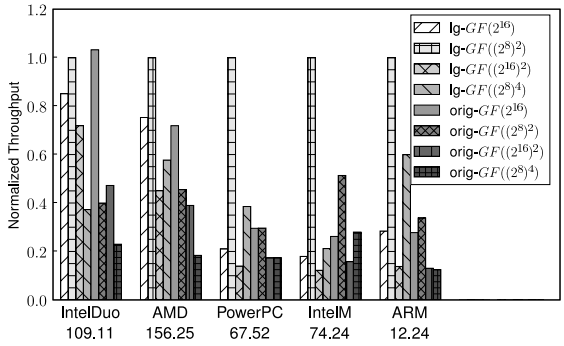
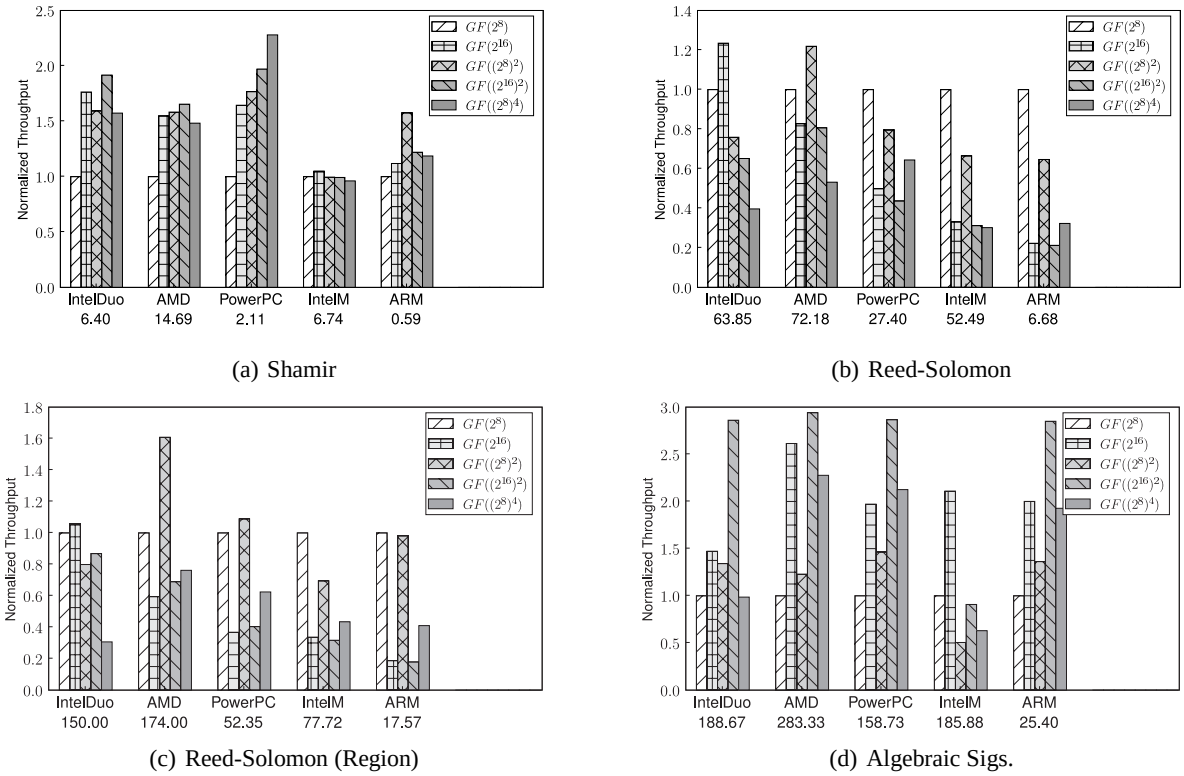


Figure 5: Normalized throughput of the traditional look-up algorithms and the composite field method. All values are normalized to lg-$GF((2^8)^2)$. The actual lg-$GF((2^8)^2)$ throughput numbers (MB/s) are shown on the x-axis along with architecture.

We ran all of the techniques (traditional look-up and composite field representations) for each application. We report the "best" performer for each application and field.

Observation #4 : *The composite field representation is quite effective and in some cases outperforms the traditional look-up table techniques.*



(a) Shamir

(b) Reed-Solomon

(c) Reed-Solomon (Region)

(d) Algebraic Sigs.

Figure 4: Normalized throughput of various applications. All values are normalized to $GF(2^8)$. The actual throughput numbers (MB/s) are shown on the x-axis along with architecture. All of the reported numbers are given for the *best* performing technique for the particular application.

Figure 5 compares the normalized throughput of UNIFORM multiplications in $GF(2^{16})$ and $GF(2^{32})$, where the ground field operations are computed using the unoptimized and optimized logarithm/antilogarithm approaches. We observed very good results when using $GF(2^8)$ as a ground field in the $GF(2^{16})$ and $GF(2^{32})$ composite field implementations. In fact, the $GF((2^8)^2)$ implementation either outperforms or performs comparably to $GF(2^{16})$ on all architectures, showing the effect table size and cache size can have on overall multiplication performance. The effect of cache size is very apparent when comparing IntelDuo and AMD to the other, relatively cache-constrained, architectures. We notice a striking performance improvement when implementing $GF(2^{16})$ as $GF((2^8)^2)$ in the architectures with smaller L1 and L2 caches. This improvement is largest on the IntelM processor, which only has an 8 KB L1 cache.

Observation #5 : *Raw multiplication performance does not always reflect application performance.*

We report the best performing multiplication method for each application in Figure 4. While not shown in Figure 4, the raw multiplication performance does not always reflect the performance across the applications. For instance, as shown in Figure 3, the left-right table results in the highest UNIFORM multiplication throughput for $GF(2^8)$. This is not necessarily the case for applications implemented using $GF(2^8)$ in Figure 4.

Overall, performance varies greatly between implementations and architecture. Interactions within the applications is much more complicated than the raw multiplication experiments; thus, the application must also be considered when choosing an appropriate Galois field representation. For example, the applications in Figure 4 perform multiplication on sizeable data buffers, leading to more frequent cache eviction of the multiplication tables.

Observation #6 : *Composite field implementation is affected by cache size.*

Cache effects are apparent in Figure 4. Due to the relatively small L1 and L2 caches in the PowerPC, IntelM and ARM, the fields implemented over $GF(2^8)$ generally outperform the $GF(2^{16})$ implementations in both Reed-Solomon encoding algorithms. The multiplication workload of algebraic signatures and Shamir's secret sharing algorithm is similar to CONSTANT , thus computation becomes a limiting factor.

Observation #7 : *No technique is best for every architecture and application.*

There exists no clear winner for $GF(2^{16})$ or $GF(2^{32})$ across architectures or techniques. However, it is important to note that performance degrades quickly as the size of the extension field grows from 2 to 4 degrees because of the exponential increase in the number of multiplications and the choice of irreducible polynomial. While there is variance in the performance across architecture and application, the optimized logarithm/antilogarithm scheme and the full multiplication table technique for $GF(2^8)$ seem to perform very well in most cases.

Observation #8 : *Application-specific optimizations for composite fields can further improve performance.*

As shown in Figure 4(d), hand-picking field elements when computing algebraic signatures in a composite field can further improve performance. As an example, we find that the $GF((2^{16})^2)$ implementation with $\alpha = \texttt{0x00010001}$ outperforms all others. Given a large ground field (*i. e.*, $GF(2^{16})$ or larger), where the elements are expected to have higher order, we should be able to perform a similar optimization for a Reed-Solomon implementation.

5.3. Discussion

While we cannot state a sweeping conclusion based on our results, there are a few interesting points worth mentioning when considering composite fields. First, one advantage to using composite fields is the ability to optimize for an application based on the underlying field representation. We have given an example of how this is done for algebraic signatures, noting that similar optimizations are possible for Reed-Solomon and Shamir's secret sharing algorithm. Second, the composite field representation has utility even when the application only requires $GF(2^8)$. As we have shown in our analysis, there exist cases where $GF((2^8)^2)$ outperforms $GF(2^8)$. This is quite evident when using the AMD processor, since it has the fastest clock speed and largest L1 cache. Finally, if an implementation requires a field larger than $GF(2^8)$, the composite field representation may lead to better performance. In many cases, $GF((2^8)^2)$ tends to outperform $GF(2^{16})$ and $GF((2^{16})^2)$ is expected to outperform many software-based $GF(2^{32})$ implementations.

6. Related Work

Plank has recently released a fast Galois field library [18], tailored for arithmetic in $GF(2^8)$, $GF(2^{16})$ and $GF(2^{32})$. Multiplication in $GF(2^8)$ and $GF(2^{16})$ is implemented using the full multiplication table and a log/antilog approach that maintains the check for zero, but optimizes the modulus operation out. Multiplication in $GF(2^{32})$ is implemented as either the field-element-to-bit-matrix approach [19] or a split-multiplication that uses seven full multiplication tables for multiplication of two 32-bit words. Our composite field approach for $GF(2^{16})$ and $GF(2^{32})$ requires less space, since we only require a single multiplication table. In addition, any ground field multiplication technique may be used in the the composite field representation.

A great deal of effort has gone into alternative Galois field representations for Reed-Solomon codes. One popular optimization uses the bit-matrix representation, and is used in Reed-Solomon erasure coding [19, 4], where each multiplication over the field $GF(2^l)$ is transformed into a product of an $l \times l$ bit matrix and an $l \times 1$ bit vector. This scheme has the advantage of computing all codewords using nothing more than word-sized XORs, by trading a multiplication for at least l XOR operations. The bit-matrix representation is typically stored as a look-up. For Reed-Solomon, $k \times m$ matrices must be stored for a code that computes m parity elements from k data elements, where $k \times m$ is generally less than 256. This optimization works well for many Reed-Solomon implementations, but it may be difficult or impossible to map the optimization to other applications. Additionally, the bit-matrix scheme may not work well in cases where all field elements are represented or there is relatively little data to encode.

Huang and Xu presented three optimizations for the logarithm/antilogarithm approach in $GF(2^8)$ [11]. The authors show improvements to the default logarithm/antilogarithm multiplication technique that result in a 67% improvement in execution time for multiplication and a $3\times$ encoding improvement for Reed-Solomon. In contrast, our study evaluates a wide range of fields and techniques that may be used for multiplication in a variety of applications.

The emergence of elliptic curve cryptography has motivated the need for efficient field operations in extension fields [23, 2, 20]. The security of this encryption scheme is dependent on the size of the field; thus the implementations focus on large fields (*i. e.*, of size 2^{160}). DeWin, *et al.* [23] and Savas, *et al.* [20] focus on $GF((2^l)^k)$, where $gcd(l,k) = 1$. Baily and Paar [2] propose a scheme, called an Optimal Extension Field, where the field polynomial is an irreducible binomial and field has prime characteristic other than 2, leading to elements that may not be byte-aligned. In other work, Paar, *et al.* describe hardware-based composite field arithmetic [15, 16].

7. Conclusions

We have presented and evaluated a variety of ways to perform arithmetic operations in Galois fields, comparing multiplication, and application performance on five distinct architectures and six workloads: three artificial workloads and three actual Galois field-based coding applications. We found that the composite field representation requires less

memory and, in many cases, leads to higher throughput than a binary extension field of the same size. Performance for both raw multiplications and applications shows that both CPU speed and cache size have a dramatic effect on performance, potentially leading to high variation in throughput across architectures, especially for larger fields such as $GF(2^{16})$ and $GF(2^{32})$. Additionally, our results show that schemes performing well in isolation (*i. e.*, measuring multiplication throughput) may not perform as well when used in an application or to perform ground computation in a composite field.

The choice of Galois field calculation method for a particular application resulted in differences as high as a factor of three for different approaches on the same architecture; moreover, no single approach worked best for any given architecture across applications. Using the approaches and evaluation techniques we have described, implementers of systems that use Galois fields for erasure code generation, secret sharing, algebraic signatures, or other techniques can increase overall system performance by selecting the best approach based on the characteristics of the hardware on which the system will run and the application-generated Galois field arithmetic workload.

References

[1] R. Ahlswede, N. Cai, S.-Y. R. Li, and R. W. Yeung. Network information flow. *IEEE Transactions on Information Theory*, 46(4):1204–1216, Jul 2000.

[2] D. V. Bailey and C. Paar. Optimal extension fields for fast arithmetic in public-key algorithms. *Lecture Notes in Computer Science*, 1462, 1998.

[3] M. Blaum, J. Brady, J. Bruck, and J. Menon. EVENODD: An efficient scheme for tolerating double disk failures in RAID architectures. *IEEE Transactions on Computers*, 44(2):192–202, 1995.

[4] J. Blomer, M. Kalfane, R. Karp, M. Karpinski, M. Luby, and D. Zuckerman. An XOR-based erasure-resilient coding scheme. Technical report, ICSI, UC-Berkeley, 1995.

[5] P. Corbett, B. English, A. Goel, T. Grcanac, S. Kleiman, J. Leong, and S. Sankar. Row-diagonal parity for double disk failure correction. In *Proceedings of the Third USENIX Conference on File and Storage Technologies (FAST)*, pages 1–14, 2004.

[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Second Edition*. MIT Press, Cambridge, Massachusetts, 2001.

[7] S. Gao and D. Panario. Tests and constructions of irreducible polynomials over finite fields. In *Foundations of Computational Mathematics*, 1997.

[8] K. M. Greenan and E. L. Miller. PRIMS: Making NVRAM suitable for extremely reliable storage. In *Proceedings of the Third Workshop on Hot Topics in System Dependability (HotDep '07)*, June 2007.

[9] K. M. Greenan, E. L. Miller, T. J. E. Schwarz, and D. D. Long. Disaster recovery codes: increasing reliability with large-stripe erasure correcting codes. In *StorageSS '07*, pages 31–36, New York, NY, USA, 2007. ACM.

[10] K. M. Greenan, E. L. Miller, and T. J. E. Schwarz, S.J. Analysis and construction of Galois fields for efficient storage reliability. Technical report, UC-Santa Cruz, 2007.

[11] C. Huang and L. Xu. Fast software implementation of finite field operations. Technical report, Washington Univ., 2003.

[12] R. Lidl and H. Niederreiter. *Introduction to Finite Fields and Their Applications*. Cambridge Univ. Press, New York, USA, 1986.

[13] F. J. MacWilliams and N. J. Sloane. *The Theory of Error Correcting Codes*. Elsevier Science B.V., 1983.

[14] National Institute of Standards and Technology. *FIPS Publication 197 : Advanced Encryption Standard*.

[15] C. Paar. A new architecture for a parallel finite field multiplier with low complexity based on composite fields. *IEEE Transactions on Computers*, 45, July 1996.

[16] C. Paar, P. Fleischmann, and P. Roelse. Efficient multiplier architectures for Galois fields $GF(2^{4n})$. *IEEE Transactions on Computers*, 47, Feb 1998.

[17] J. S. Plank. A tutorial on Reed-Solomon coding for fault-tolerance in RAID-like systems. *Software—Practice and Experience (SPE)*, 27(9):995–1012, Sept. 1997. Correction in James S. Plank and Ying Ding, Technical Report UT-CS-03-504, U Tennessee, 2003.

[18] J. S. Plank. Fast Galois field arithmetic library in C/C++, April 2007.

[19] J. S. Plank and L. Xu. Optimizing Cauchy Reed-Solomon codes for fault-tolerant network storage applications. In *IEEE Interantional Symposium on Network Computing and Applications*, 2006.

[20] E. Savas and C. K. Koc. Efficient methods for composite field arithmetic. Technical report, Oregon St. Univ., 1999.

[21] T. Schwarz, S. J. and E. L. Miller. Store, forget, and check: Using algebraic signatures to check remotely administered storage. In *Proceedings of the 26th International Conference on Distributed Computing Systems (ICDCS '06)*, Lisboa, Portugal, July 2006. IEEE.

[22] A. Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, Nov. 1979.

[23] D. Win, Bosselaers, Vandenberghe, D. Gersem, and Vandewalle. A fast software implementation for arithmetic operations in $GF(2^n)$. In *ASIACRYPT: International Conference on the Theory and Application of Cryptology*, 1996.

[24] L. Xu and J. Bruck. X-code : MDS array codes with optimal encoding. In *IEEE Transactions on Information Theory*, 1999.

Flat XOR-based erasure codes in storage systems: Constructions, efficient recovery, and tradeoffs

Kevin M. Greenan
ParaScale
kgreenan@parascale.com

Xiaozhou Li
HP Labs
xiaozhou.li@hp.com

Jay J. Wylie
HP Labs
jay.wylie@hp.com

***Abstract*—Large scale storage systems require multi-disk fault tolerant erasure codes. Replication and RAID extensions that protect against two- and three-disk failures offer a stark tradeoff between how much data must be stored, and how much data must be read to recover a failed disk. *Flat* XOR-*codes*—erasure codes in which parity disks are calculated as the XOR of some subset of data disks—offer a tradeoff between these extremes. In this paper, we describe constructions of two novel flat XOR-code, *Stepped Combination* and *HD-Combination* codes. We describe an algorithm for flat XOR-codes that enumerates *recovery equations*, i.e., sets of disks that can recover a failed disk. We also describe two algorithms for flat XOR-codes that generate *recovery schedules*, i.e., sets of recovery equations that can be used in concert to achieve efficient recovery. Finally, we analyze the key storage properties of many flat XOR-codes and of MDS codes such as replication and RAID 6 to show the cost-benefit tradeoff gap that flat XOR-codes can fill.**

I. Introduction

Erasure codes such as replication, RAID 5, and other Reed-Solomon codes, are the traditional means by which storage systems are typically made reliable. Such codes are Maximum Distance Separable (MDS). This means that they offer optimal space-efficiency for a given fault tolerance: each additional disk of redundancy allows the system to tolerate an additional disk (or sector) failure. As big data systems become more prevalent, more systems are comprised of huge populations of disks, and the commodity-based architecture leads to higher disk failure rates. Indeed, there are many alarming trends in disk failure rates [1], [2], [3], [4], [5]. These storage trends motivate the move towards two- and three-disk fault tolerant storage systems for big data, archival, and cloud storage.

As we move towards more and more fault tolerant storage systems, we believe that non-MDS codes ought to be considered for deployment. We have studied *flat* XOR-*codes* and identified features of such codes that may benefit a wide range of large-scale storage systems. Flat XOR-codes are erasure codes in which each parity disk is the XOR of a distinct subset of data disks. Such codes are non-MDS and so incur additional storage overhead, and, in some cases, additional small write costs relative to MDS codes. In return for these costs, such codes offer short, diverse *recovery equations*, i.e., many distinct sets of disks that can recover a specific disk.

We believe that the recovery equations of flat XOR-codes can be exploited in many different ways. First, short, diverse recovery equations offers the potential for more efficient recovery of failed disks relative to MDS codes. By efficient, we mean fewer bytes need to be read and the read recovery load is spread evenly over most disks in the stripe. Efficient disk recovery of a flat XOR-code can be faster and can interfere less with the foreground workload than for a similarly fault tolerant MDS code. For flat XOR-codes with a *static* layout (like a RAID 4 layout with specific parity disks), a *simultaneous recovery schedule* uses multiple, distinct recovery equations to achieve efficient recovery. For flat XOR-codes with a *rotated* layout (like a RAID 5 layout with data and parity on each disk), a *rotated recovery schedule* uses a different recovery equation for each failed stripe to achieve efficient recovery. Our proposed *intra-stripe* recovery schedules complement techniques such as distributed sparing [6] and parity declustering [7].

We also believe there are opportunities in low-power (archival) storage and storage-efficient big data computing. Flat XOR-codes may offer a richer set of tradeoffs for power-aware storage systems that leverage redundancy to schedule disk power-downs: EERAID [8], PARRAID [9], eRAID [10], and Power-Aware Coding [11]. Big data computing systems that use triple-replication (e.g., Google File System [12] and the Hadoop File System [13]) may be able to achieve significantly higher storage-efficiency with nominal additional read costs incurred by computation scheduled on parity nodes.

We make the following contributions in this paper. First, we propose the following novel flat XOR-code *Combination* code constructions: *Stepped Combination* codes and *HD-Combination* codes. Both code constructions have two- and three-disk fault tolerant variants. We also describe a prior flat XOR-code construction, *Chain* code, and a *flattening* method to convert previously known array codes such as EVENODD into a flat XOR-code. Second, we develop the following algorithms for *flat* XOR-*codes*: the Recovery Equations (RE) Algorithm, the Simultaneous Recovery Schedule (SRS) Algorithm, and the Rotated Recovery Schedule (RRS) Algorithm. Third, we analyze key properties of flat XOR-codes and MDS codes for storage: storage overhead, small write costs, recovery equation size, recovery read load, and fault tolerance.

Even though replication is an MDS code, there is a gap between the properties of replicated storage (high space overhead, small recovery equations) and other traditional MDS codes such as RAID 6 (low space overhead, long recovery equations). Our analysis leads us to believe that the Combi-

978-1-4244-7153-9/10/ $26.00

nation codes and Chain codes delineate the possible tradeoff space that flat XOR-codes can achieve, and that this tradeoff space fills a large part of the gap between replicated storage and other MDS storage.

In Section II, we provide an extensive background on erasure-coded storage. We describe all of the flat XOR-code constructions, including our novel constructions, in Section III. In Section IV, we describe our algorithms for producing recovery equations and schedules for flat XOR-codes. We analyze key storage properties of two- and three-disk fault tolerant MDS and flat XOR-codes in Section V.

II. Background

Over a decade ago, a class of non-MDS erasure codes, low-density parity-check (LDPC) codes, were discovered [14] (actually, re-discovered [15]). This class of erasure code has had a significant impact in networked and communication systems, starting with Digital Fountain content streaming [16], and now widely adopted in modern IEEE standards such as 10GBase-T Ethernet [17] and WiMAX [18]. The appeal of LDPC codes is that, for large such codes, a small amount of space-efficiency can be sacrificed to significantly reduce the computation costs required to encode and decode data over lossy channels. Networked systems can take advantage of the asymptotic nature of LDPC codes to great effect.

To date, the only non-MDS codes that have been used in practice in storage systems are replicated RAID configurations such as RAID 10, RAID 50, and RAID 60. Storage systems have not taken advantage of LDPC style codes. There are many reasons for this. First, storage systems only use *systematic* codes—erasure codes in which the stored data is striped across the first k disks. This allows *small reads* to retrieve a portion of some stored data by reading a minimal subset of the data disks. This constraint distinguishes storage systems from network systems. Second, though storage systems continue to get larger and larger, they do not take advantage of the asymptotic properties of LDPC codes because any specific stored object is striped over a relatively small number of disks (e.g., 8–20, maybe 30). This leads to the final reason: "small" LDPC codes are not well understood. There are not many well-known methods of constructing small LDPC codes for storage, though some specific constructions are known (e.g., [19], [20], [21], [22], [23]). The performance, fault tolerance, and reliability of small LDPC codes is also not well understood, though there is much progress (e.g., [24], [25], [26], [27], [28], [29], [11]). In this paper we provide many constructions of flat XOR-codes, which in some sense are small LDPC codes, and an analysis to put the storage properties of such constructions in perspective relative to traditional MDS codes.

A. Terminology

An erasure code consists of n symbols, k of which are *data symbols*, and m of which are *redundant symbols*. For XOR-codes, we refer to redundant symbols as *parity symbols*. We only consider *systematic* erasure codes: codes that store the data symbols along with the parity symbols. Whereas coding theorists consider individual bits as symbols, symbols in storage systems correspond to device blocks. To capture this difference in usage of erasure codes by storage systems from elsewhere, we refer to *elements* rather than symbols. We use the terms *erase* and *fail* synonymously: an element is erased if the disk on which it is stored fails.

The fault tolerance of an erasure code is defined by d its *Hamming distance*. An erasure code of Hamming distance d tolerates all failures of fewer than d elements, data or parity. In storage systems, erasure codes are normally described as being one-, two-, or three-disk fault tolerant. These respectively correspond to Hamming distances of two, three, and four.

MDS codes use m redundant elements to tolerate any m erasures. MDS codes are optimally space-efficient: every redundant element is necessary to achieve the Hamming distance. Replication, RAID 5, RAID 6, and Reed-Solomon codes are all examples of MDS codes. To be specific, for replication $k = m = 1$, for RAID 5 $m = 1$, and for RAID 6 $m = 2$. Reed-Solomon codes can be instantiated for any value of k and m [30] and can be implemented in a computationally efficient manner (e.g., Liberation [31] and Liber8Tion codes [32]).

Still, much work has gone into determining constructions of MDS codes that use only XOR operations to generate redundant elements. Such constructions are based on parity techniques developed by Park [33]. We refer to this class of erasure code constructions as *parity-check array codes*. Examples of such codes include EVENODD [34], generalized EVENODD [35], Row-Diagonal Parity (RDP) [36], X-Code [37], Star [38], and P-Code [39]. Parity-check array codes have a two-dimensional structure in which some number of elements are placed together "vertically" in a *strip* on each disk in the *stripe*. This two-dimensional structure allows such codes to be MDS and so does not offer interesting recovery equation possibilities. In Section III-B though, we describe a method of converting parity-check array codes into non-MDS flat XOR-codes.

B. Flat XOR-codes

Flat XOR-codes are small low-density parity-check (LDPC) codes [14]. Because such codes are flat—each strip consists of a single element—and each parity element is simply the XOR of a subset of data elements, they cannot be MDS. The singular exception to this rule is RAID 5 which is a one-disk fault tolerant MDS flat XOR-code with $m = 1$.

A concise way of describing an LDPC code is with a Tanner graph: a bipartite graph with data elements on one side, and parity elements on the other. A parity element is calculated by XORing each data element to which it is connected together. Figure 1 illustrates a simple Tanner graph with the corresponding parity equations. The Tanner graph can also be used for efficient decoding of LDPC codes[1].

[1]A technique called *iterative decoding* based on the Tanner graph can efficiently decode with high probability. *Stopping sets* [40], specific sets of failures, can prevent iterative decoding from successfully decoding even though a more expensive matrix-based decode operation could succeed. In storage systems, we expect the expensive matrix-based decode method to be used in such cases.

When k is very large, LDPC codes can achieve near MDS level space-efficiency for a given fault tolerance, and significantly lower computational complexity for both encoding and decoding. In storage systems, these asymptotic properties do not apply since k tends not to be sufficiently large. Traditionally, LDPC codes are constructed by randomly generating a Tanner graph based on some distributions of the expected edge count for data and parity elements. Unfortunately, such techniques only work (with high probability) when k is sufficiently large. Therefore, storage systems practitioners need constructions of flat XOR-codes with well understood properties.

C. Related work

Parity declustering [7] (and chained declustering [41]) distribute the recovery read load among many disks. Parity declustering lays out many stripes on distinct sets of devices such that all available devices participate equally in reconstruction when some device fails; it needs to be used with more disks than the stripe width to distribute the recovery read load. Distributed sparing allows spare capacity on each available disk in the system to recover a portion of a failed device and so distributes the recovery write load [6]. The recovery schedules we propose are complementary to these techniques, and differ from these techniques in that they operate wholly within a stripe.

Recently, non-MDS self-adaptive parity-check array codes have been proposed [23]. Such codes allow disks to be rebuilt within the stripe so as to preserve two-disk fault tolerance, until fewer than two redundant disks are available. In some sense, distributed sparing is built into the array code itself. SSPiRAL codes also include techniques for rebuilding elements within a stripe as disks fail [22], [42]. Our techniques do not yet do anything analogous to distributed sparing within a stripe for flat XOR-codes. Pyramid codes [43] are non-MDS codes constructed from traditional MDS codes; we are interested in comparing flat XOR-codes to Pyramid constructions in the future. Weaver codes are non-MDS vertical (i.e., not flat) XOR-codes that Hafner discovered via computational methods [20]. Weaver codes have a regular, symmetric structure which limits the recovery read load after a failure to nearby "neighbor" disks. We are interested in distributing the recovery read load widely rather than localizing it.

There is much prior work on evaluating erasure codes in storage systems. Hafner et al. [44] outlined a comprehensive performance evaluation regime for erasure-coded storage. There is a long history of evaluating the reliability of RAID variants in storage systems (e.g., [45], [46], [47], [48], [1]). Some aspects of small LDPC code instances suitable for storage systems have been previously studied, e.g., read performance (e.g., [24], [25]), fault tolerance (e.g., [28], [27]), and reliability (e.g., [26], [29], [11]). For additional background on erasure coding in storage, see Plank's tutorial slides [49].

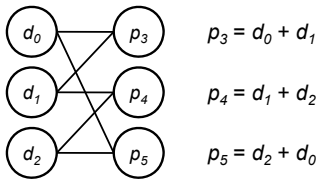


Fig. 1. Two-disk fault tolerant Chain code construction ($k = m = d = 3$).

III. Constructions

In this section, we describe Chain codes a variant of a previously known flat XOR-code and *flattening* a technique to convert any parity-array check code into a flat XOR-code. We describe these variants of previously known codes in detail so that others can easily reproduce these constructions, and because we include these constructions in our analysis. We then describe our novel combination constructions for Stepped Combination and HD-Combination codes.

A. Chain codes

For a code to tolerate at least two disk failures, it must have a Hamming distance d of at least three. For a systematic XOR-based code, this means that each data element must be in at least two parity equations. Starting with this minimal constraint, two-disk fault tolerant *Chain* codes[2] can be constructed [19].

A Chain code construction requires that $k = m$. As originally proposed, a Chain code can be thought of as alternating data elements and parity elements in a row, with each parity element being the XOR of the data element on either side of it. Figure 1 illustrates our Chain code construction for $k = m = d = 3$ with the Tanner graph on the left and parity equations on the right. Let d_i be data element i, where $0 \leq i < k$, and let p_j be parity element j, where $0 \leq j < m$. For Hamming distance d, the parity equation for p_j is as follows:

$$p_j = \sum_{i=j}^{j+d-1} d_{i \bmod k}.$$

Simply put, each parity equation is the XOR sum of $d-1$ contiguous data elements, wrapping around appropriately after the k^{th} data element.

The Chain code construction requires that $k = m \geq d = 3$. If k is less than d, then all of the parity equations end up being the same (the XOR sum of all data elements) and the constructed Chain code does not achieve a Hamming distance of d. If $d = 2$, then the Chain code construction replicates each data element once (i.e., RAID 10), and if $d = 1$, then the Chain code construction stripes the data (i.e., RAID 0). The Chain code construction can be extended to three-disk fault tolerance: if $k = m \geq d = 4$, then connect each parity element to three contiguous data elements.

[2] "Chain code" is our name for these codes. The patent by Wilner does not explicitly name these codes [19].

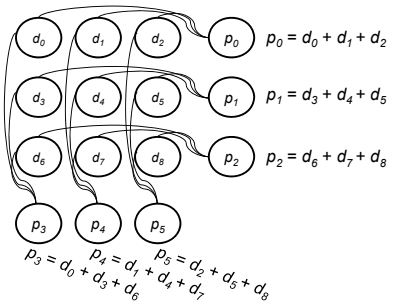


Fig. 2. SPC with $k = 9$ and $m = 6$. Hamming distance is three when flattened.

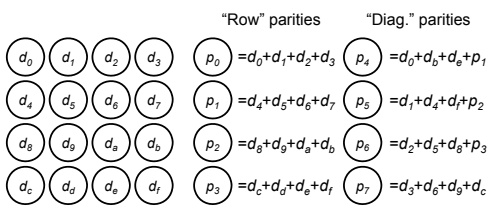


Fig. 3. Flattened RDP code for prime $p = 5$ yields $k = 16$, $m = 8$, and $d = 3$.

Unfortunately, we have not found such simple Chain code construction variants for a Hamming distance greater than four. Note that the Weaver$(n, 2, 2)$ codes [20] discovered by Hafner can be converted into the two-disk fault tolerant Chain code by flattening (discussed below). We believe that flattening the other Weaver(n, t, t) code constructions may produce other flat XOR-codes with greater Hamming distance, though the construction will not be as simple as the Chain codes for $d = 3$ and $d = 4$. Note that the SSPiRAL codes are another variant/extension of the Chain code construction [22].

B. Flattened parity-check array codes

Parity-check array code constructions can be *flattened* to produce a flat XOR-code with the same Hamming distance as the original array code. By flattening, we mean taking some two (or more) dimensional code and placing each element on a distinct device. Whereas the original parity-check array code construction produces an MDS code, flattening the code yields a non-MDS code.

To illustrate the flattening technique, we apply it to two parity-check array code constructions: Simple Product Code (SPC) and Row-Diagonal Parity (RDP). Flattening can be applied to other two-disk fault tolerant array codes such as EVENODD, X-CODE, P-CODE, and so on. Flattening can also be applied to three-disk fault tolerant array codes such as STAR codes and generalized EVENODD codes, or even to non-MDS parity-check array codes such as Weaver and HoVer [21] codes.

1) Simple Product Code (SPC): A Simple Product Code (SPC) [50] is constructed by grouping data elements into a rectangle and then performing RAID 4 on each row and each column. Figure 2 illustrates a specific SPC construction in which nine data elements are layed out in a square pattern. There is a parity element for each row and for each column of data elements. When flattened, this code uses $k = 9$ data disks and $m = 6$ parity disks to tolerate any two element failures (i.e., $d = 3$).

We only consider flattened *square* SPC constructions (i.e., those with the same number of rows as columns) for the purposes of this paper, even though any rectangular layout of data elements can be used. For an SPC code with q rows/columns, $k = q^2$ and $m = 2q$. The storage overhead is therefore $(q + 2)/q$, and each data element participates in two parity equations each of which is of length q elements. Note that the *full-2* and *full-3* codes proposed by Gibson et al. [51] are similar to the SPC construction, and to the HoVer$^2_{1,1}$ construction given by Hafner [21].

2) Row-Diagonal Parity (RDP): Figure 3 illustrates a flattened RDP code. This figure is based on Figure 1 from the RDP paper [36] which illustrates the RDP construction for prime $p = 5$. The figure is setup to illustrate the columns (or strips) of elements that would be on each disk of an RDP instance. For example the first data disk would store elements d_0, d_4, d_8, d_c, and the first parity disk would store elements p_0, p_1, p_2, and p_3. Once flattened, each element of the RDP code is on a distinct disk producing a non-MDS flat XOR-code with $k = 16$, $m = 8$, and $d = 3$.

In general, for prime p, a flattened RDP code has $k = (p-1)^2$ and $m = 2(p-1)$. The space overhead is therefore $(p+1)/(p-1)$. Note that some of the "diagonal" parity equations are given in terms of "row" parity equations in the figure. This makes it look like each parity equation has $p-1$ elements. In fact, p parity equations are $p-1$ elements in length, and $p-2$ parity equations are $2p-3$ elements in length.

C. Combination codes

In this section, we present four novel Combination code constructions. We present two- and three-disk fault tolerant *Stepped Combination* codes. We then describe a variant of each of these codes called *HD-Combination* codes (where HD means Hamming Distance). We refer to these code constructions as Combination codes because they construct a Tanner graph by assigning distinct combinations of parity elements to each data element. The difference between the two constructions is in the size of the combinations assigned. We believe Combination constructions minimize the storage overhead for flat XOR-codes. Thus far, we have only developed constructions for $d = 3$ and $d = 4$. We expect that constructions for greater Hamming distances do exist, but have not yet found them. We include a proof sketch that combination codes achieve the specified Hamming distance in Appendix I.

1) Two-disk fault tolerant Stepped Combination: Stepped Combination codes are constructed by assigning all combinations of parity elements that are of size $d-1$ to a different data element, then all combinations of parity elements that are of size d, and so on, up until the one combination of all m parity elements is assigned. Figure 4 lists the procedure for constructing a Stepped Combination code with $d = 3$. Given

100: **for** $i := 2, 3, \ldots, m$ **do**
101: **for** each m-choose-i combination of parities, P **do**
102: **for** next unconnected data element D **do**
103: connect D to the i parities in P

Fig. 4. Stepped Combination code construction for $d = 3$.

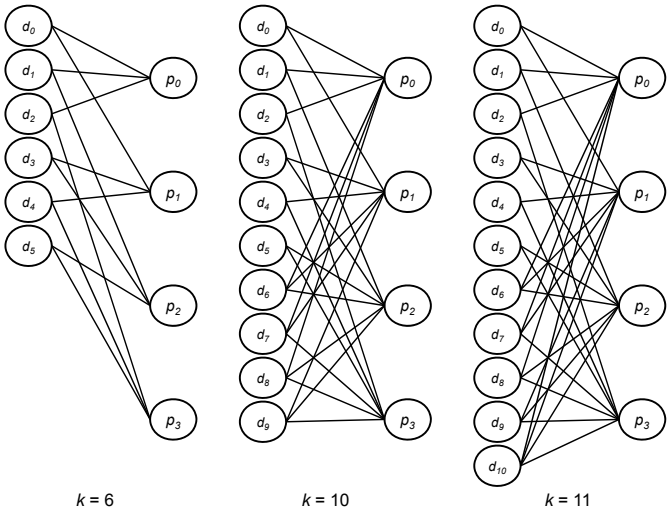


Fig. 5. Two-disk fault tolerant Stepped Combination constructions for $d = 3$ and $m = 4$.

the Hamming distance d and a number of parity elements m, the construction assigns combinations of increasing size to the available data elements. There is a maximum number of data elements for which this construction works. If $d = 3$ and $m = 4$, then there are $\binom{4}{2} + \binom{4}{3} + \binom{4}{4} = 6 + 4 + 1 = 11$ distinct combinations of parity elements that can be assigned to data elements. In general, $k \leq \sum_{i=2}^{m} \binom{m}{i} = 2^m - m - 1$. In some sense, this also implies a lower bound on k: if a Stepped Combination code can be constructed for the desired value of k with a smaller m, then that smaller value of m should be used. Therefore, $2^{m-1} - m < k$.

Figure 5 illustrates the Tanner graphs of three Stepped Combination code constructions with $d = 3$ and $m = 4$. The values of k are chosen based on the three different sizes of combinations: 6 of size two, $6 + 4 = 10$ of size two or three, and $6 + 4 + 1 = 11$ of size two through four. In the Tanner graph for $k = 6$, notice that each data element participates in two distinct parity equations (connects to two parity elements). For the Tanner graph with $k = 10$ the first six data elements are the same as for $k = 6$. The next four data elements each participate in three parity equations. Finally, the Tanner graph with $k = 11$ has the same first ten data elements as for $k = 10$ and then one additional data element that connects to all four parity elements. These Tanner graphs illustrate the key idea behind the construction of Stepped Combination codes.

2) Three-disk fault tolerant Stepped Combination: The Stepped Combination code construction for Hamming distance four is quite similar to that for three, but with one key difference. Consider the construction algorithm for the two-disk fault tolerant Stepped Combination code in Figure 4. The first difference for the three-disk fault tolerant construction is that variable i on line 100 starts at 3: to achieve a Hamming distance of four, all data elements must be connected to at least three parity elements. The second difference is that variable i increments by two each iteration, i.e., $i := 3, 5, 7, \ldots, m$. This difference is not intuitive, though the proof sketch in Appendix I provides some explanation.

Given some number of parity elements m, the three-disk fault tolerant Stepped Combination code has an upper and lower limit on k. For example, if $m = 5$, then $k \leq \binom{5}{3} + \binom{5}{5} = 10 + 1 = 11$. In general, $k \leq \sum_{3 \leq i \leq m, i \text{ odd}} \binom{m}{i} = 2^{m-1} - m$. The lower bound is based on $m - 1$ and so $2^{m-2} - m + 1 < k$. Compared with the two-disk fault tolerant construction, the maximum value of k drops by about half: $2^m - m - 1$ vs. $2^{m-1} - m$.

3) HD-Combination codes: HD-Combination codes are a variant of Stepped Combination codes. As with Stepped Combination codes, we have developed two- and three-disk fault tolerant constructions. Whereas the Stepped Combination construction steps up the size of parity combinations, the HD-Combination construction only uses parity combinations of the minimum size necessary to achieve the Hamming distance. As with the Stepped Combination code, we have not found HD-Combination constructions for larger Hamming distances yet.

The HD-Combination construction code differs from the algorithm described in Figure 4 only on line 100. For an HD-Combination construction with either $d = 3$ or $d = 4$, this line is simply $i = d - 1$. This difference leads to different bounds on k. For the two-disk fault tolerant HD-Combination code construction, $\binom{m-1}{2} < k \leq \binom{m}{2}$. Whereas for the three-disk fault tolerant construction, $\binom{m-1}{3} < k \leq \binom{m}{3}$.

IV. Recovery equations and schedules

In this section, we describe *recovery equations* (i.e., sets of elements that can be used to recover a specific other element) and *recovery schedules* (i.e., sets of recovery equations that can be used in concert to make recovery more efficient). To illustrate the basic ideas, we first consider multi-disk fault tolerant MDS codes. We then describe examples of recovery equations and schedules for flat XOR-codes that illustrate some key differences with MDS codes. We then present algorithms to determine recovery equations and schedules for flat XOR-codes. Finally, we discuss some limitations of our current algorithms.

A. MDS Examples

For MDS codes, any combination of k elements can be used to recovery any other element. Therefore, each element in an MDS code has exactly $\binom{k+m}{m}$ recovery equations, each consisting of k elements. For two-disk fault tolerant codes like RAID 6, this is only $k + m - 1$ recovery equations.

Figure 6 illustrates the recovery equations for element e_0 of a RAID 6 MDS code with $k = 4$ and $m = 2$. In this example, the code is static (not rotated), so disks 0 through 3 are data disks, and disks 4 and 5 are parity disks. If disk 0 were inaccessible, and an element e_0 from some stripe needs

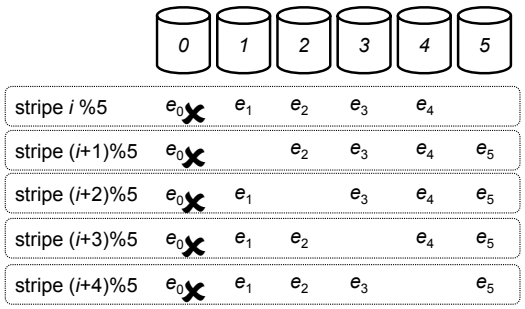


Fig. 6. Recovery equations for RAID 6.

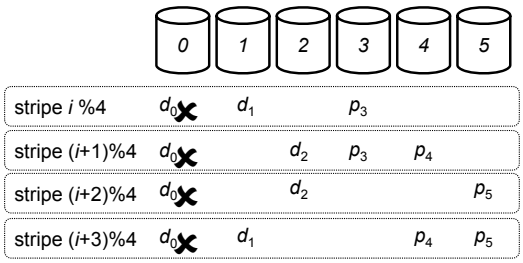


Fig. 7. Simultaneous recovery schedule for Chain code.

to be read, any of the five sets of four other elements in that stripe can be read and decoded instead to recover e_0.

Now, consider if disk 0 fails, as is illustrated in Figure 6. A *recovery schedule* is a set of recovery equations that can be used in concert to make recovery more efficient. Since the code is static, we want a *simultaneous recovery schedule* that uses many different recovery equations for the same element in concert. This figure illustrates such a schedule: each of the five recovery equations is used to recover one fifth of the stripes on the failed disk. This simultaneous recovery schedule distributes the recovery read load evenly over all the available disks.

If the RAID 6 code in Figure 6 was rotated, then elements e_0 through e_5 would be stored in the various stripes on disk 0. For this layout, a *rotated recovery schedule* is required. For MDS codes, a simultaneous recovery schedule is easily converted to a rotated recovery schedule since any k elements can recover *any* element. I.e., the recovery schedule is the same, but the element recovered on disk 0 depends on which stripe is being recovered.

B. Flat XOR-code examples

Recovery equations for flat XOR-codes are more complicated than for MDS codes. Each element of a flat XOR-code may have a different number of recovery equations, and each of these may differ in size. For example, consider the recovery equations of the the two-disk fault tolerant Chain code illustrated in Figure 1 with $k = m = 3$:

$$d_0 = d_1 \oplus p_3 = d_2 \oplus p_3 \oplus p_4 = d_2 \oplus p_5 = d_1 \oplus p_4 \oplus p_5$$
$$d_1 = d_0 \oplus p_3 = d_2 \oplus p_4 = d_2 \oplus p_3 \oplus p_5 = d_0 \oplus p_4 \oplus p_5$$
$$d_2 = d_1 \oplus p_4 = d_0 \oplus p_3 \oplus p_4 = d_0 \oplus p_5 = d_1 \oplus p_3 \oplus p_5$$
$$p_3 = d_0 \oplus d_1 = d_0 \oplus d_2 \oplus p_4 = d_1 \oplus d_2 \oplus p_5 = p_4 \oplus p_5$$
$$p_4 = d_1 \oplus d_2 = d_0 \oplus d_2 \oplus p_3 = d_0 \oplus d_1 \oplus p_5 = p_3 \oplus p_5$$
$$p_5 = d_0 \oplus d_2 = d_1 \oplus d_2 \oplus p_3 = d_0 \oplus d_1 \oplus p_4 = p_3 \oplus p_4$$

We can list these recovery equations exhaustively because the code is small and simple. Contrast these recovery equations with those of the similarly sized MDS code discussed above. Some of these recovery equations consist of only two elements, which is less than $k = 3$. This difference is more pronounced in flat XOR-codes with larger k. Another property that emerges with larger k is that there are more distinct recovery equations to choose from than for a similarly sized MDS code.

Figure 7 illustrates a simultaneous recovery schedule for the two-disk fault tolerant Chain code discussed previously. If disk 0 fails, and the Chain code has a static layout, then each of the four recovery equations for d_0 can be used in a simultaneous recovery schedule. Such a schedule has two interesting features. First, 2.5 disk's worth of data are read to recover the failed disk. This is less than $k = 3$, the minimum amount of data required to be read by an MDS code with $k = 3$. Second, each available disk reads only one half a disk's worth of data, and the recovery read load is evenly distributed among available disks. These two properties, reducing the minimum total amount of data required to be read for recovery, and distributing that recovery read load evenly among available devices, are the properties we mean by efficient recovery.

If distributing the recovery read load evenly among available disks is not a priority, then just the shortest recovery equations can be used: $d_0 = d_1 \oplus p_3 = d_2 \oplus p_5$. This simultaneous recovery schedule requires two disk's worth of data to be read, less than the above schedule. It also requires four of the five available disks to read half a disk's worth of data, and the one remaining available disk, disk 4, to read none.

Figure 8 illustrates a rotated recovery schedule for the two-disk fault tolerant Chain code discussed previously. Six rotated stripes are illustrated with all elements labeled. In the figure, disk 0 is failed and bold-faced elements are in the recovery equation used to recover the failed element, whereas grayed out elements are not. A total of two disk's worth of data are read to recover failed disk 0. Of the five available disks, two read half a disk's worth of data (disks 1 and 5) and three read one third of a disk's worth of data (disks 2, 3, and 4). Disks 1 and 5 could thus be a bottleneck in this recovery.

C. Recovery Equations (RE) Algorithm

Before describing the RE Algorithm, consider a simplistic, brute-force algorithm which enumerates the powerset of all possible recovery equations (i.e., all other elements) for each element in the code. Each possible recovery equation is tested to see if it recovers the desired element; if it does so, then it is retained as a recovery equation. (By tested, we mean decoding would be attempted which is essentially a matrix rank test.) Such a brute force algorithm is very expensive since it evaluates $(k + m) \cdot 2^{k+m-1}$ possible recovery equations.

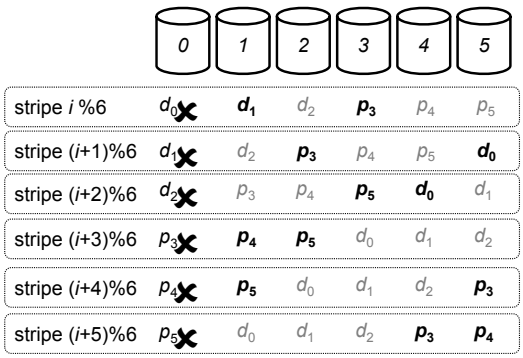


Fig. 8. Rotated Recovery for Chain code.

Figure 9 illustrates the pseudocode for our current RE Algorithm. The RE Algorithm first generates *base recovery equations* for each element (line 201). The Tanner graph is used to generate the base recovery equations. Each parity element has exactly one base recovery equation: the parity equation itself. Each data element has at least one base recovery equation per parity element to which it is connected. Each parity equation in which the data element appears is rearranged to become a base recovery equation for that data element. Note that if any *odd set* of recovery equations for some element e_i (i.e., any $3, 5, \ldots$ equations) are added together (via XOR), then the resulting recovery equation also solves for e_i. So, if a data element is connected to three or more parity elements, the RE Algorithm includes all odd sets of the initial base recovery equations in that data element's base recovery equations.

The RE Algorithm generates all of the other recovery equations for each element in turn by invoking **re_element**. The set of recovery equations $RE_i$for element e_i is initialized with its base recovery equations (line 300). Every recovery equation re in $RE_i$is used to generate additional recovery equations to add to RE_i(line 301). Each element e_j of re is substituted by every one of its base recovery equations that does not contain element e_i (lines 302–305). The XOR notation as applied to the sets on line 305 means that only the unique elements are retained (i.e., $(re \cup bre) \setminus (re \cap bre)$). If a new recovery equation is found, it is added to the set RE_i(lines 306–307). Note that this new recovery equation will eventually also be processed by this loop (i.e., by line 300).

To make this algorithm a bit more concrete, consider the recovery equations we list above for the two-disk fault tolerant Chain code. For d_0, the base recovery equations are $d_1 \oplus p_3$ and $d_2 \oplus p_5$. For d_1, the base recovery equations are $d_0 \oplus p_3$ and $d_2 \oplus p_4$. In the first iteration of the loop in **re_element** for $i = 0$, the element d_1 in the first base recovery equation of d_0 is substituted. The second base recovery equation of d_1 is substituted because the first includes d_0. This results in recovery equation $d_2 \oplus p_3 \oplus p_4$ being added to RE_i.

D. Recovery schedule algorithms

The Simultaneous Recovery Schedule (SRS) Algorithm uses the recovery equations enumerated by the RE Algorithm to generate simultaneous recovery schedules. Remember that a recovery schedule is simply a set of recovery equations. The SRS Algorithm works similarly to an iterative, depth-first search of possible schedules, where "depth" is the number of recovery equations in which each disk participates in a schedule. Given the recovery equations and some maximum depth $D_{\max}$, the SRS Algorithm enumerates all the feasible schedules for each depth up to $D_{\max}$. A *feasible* schedule for depth D being one in which no disk participates in more than D distinct recovery equations. The SRS Algorithm post-processes the feasible schedules for all depths to find the most efficient.

```
re_algorithm ()
200: for i ← 0, ..., k + m − 1 do
201:     BRE[i] ← base recovery equations for e_i
202: for i ← 0, ..., k + m − 1 do
203:     RE[i] ← re_element(i, BRE)
204: return RE
re_element(i, BRE)
300: RE_i ← BRE[i]
301: for all re ∈ RE_i do
302:     for all e_j ∈ re do
303:         for all bre ∈ BRE[j] do
304:             if e_i ∉ bre then
305:                 re' ← re ⊕ bre
306:                 if re' ∉ RE_i then
307:                     RE_i ← RE_i ∪ re'
308: return RE_i
```

Fig. 9. Recovery Equations (RE) Algorithm pseudocode.

```
fsrs(srs, SRS, re_off, RE_i, D)
400: for re_ndx ∈ (re_off, ..., |RE_i|) do
401:     srs_nxt ← srs ∪ {RE_i[re_ndx]}
402:     if max_depth(srs_nxt) ≤ D then
403:         SRS ← fsrs(srs_nxt, SRS, re_ndx + 1, RE_i, D)
404: return SRS ∪ {srs}
```

Fig. 10. Part of the Simultaneous Recovery Schedule (SRS) Algorithm pseudocode that enumerates feasible schedules for element e_i with depth D.

Figure 10 illustrates **fsrs**, the greedy and recursive manner in which feasible schedules are enumerated for depth D. For the initial invocation of **fsrs**, both srs and SRS are $\emptyset$. The greedy and recursive parts of this algorithm are lines 400 and 403 respectively. Together, these steps ensure that all feasible schedules are found and returned in variable SRS. The method **max_depth** returns the maximum depth across all elements for the schedule $srs \cup \{RE_i\}$.

The SRS Algorithm finds the most efficient schedule by post-processing all of the feasible schedules found up to $D_{\max}$. An efficient schedule primarily minimizes the amount of data read by any one disk and secondarily minimizes the overall amount of data read. For the two-disk fault tolerant Chain code discussed above, the best simultaneous recovery schedule for d_0, $\{d_1 \oplus p_3,\ d_2 \oplus p_5\}$, would be found by **fsrs** at $D = 1$. The schedule illustrated in Figure 7 would be found by **fsrs** at $D = 2$.

The Rotated Recovery Schedule (RRS) Algorithm is a

straight forward variant of the SRS Algorithm: whereas the SRS Algorithm consider all recovery equations for a single element at a time, the RRS Algorithm considers all recovery equations for all elements. Unfortunately, this makes the RRS Algorithm much slower than the SRS Algorithm. There are two reasons for this slowdown. First, the RRS Algorithm must consider many more recovery equations. Second, the RRS Algorithm needs to run to a greater depth to find a schedule which recovers all of the elements.

E. Discussion

The number of recovery equations can grow exponentially. The RE Algorithm can be configured to terminate after some amount of time, or some number of recovery equations have been found. Developing more efficient versions of the RE Algorithm, SRS Algorithm, and the RRS Algorithm is an open question.

Hafner et al. have developed heuristic methods of constructing a pseudoinverse of a parity-check array code that determines whether a specific configuration of disk and sector failures is recoverable [52], and if so, provides a short recovery equation to recover the lost data. We believe that an algorithm based on Hafner's could more effectively find "short" recovery equations than the RE Algorithm. We believe that stochastic optimization techniques could be incorporated into the SRS Algorithm or RRS Algorithm to tame some of the state space explosion. We also believe that recovery schedules for flat XOR-codes are related to network coding for erasure-coded storage systems [53], [54].

We have only presented recovery schedules for single disk failures even though the example Chain code is two-disk fault tolerant. The algorithms we have developed easily extend to recovering a single disk in the face of a multi-disk failure: recovery equations that include failed elements are not passed into the scheduling algorithms. We have also developed extensions to the SRS Algorithm to produce recovery schedules for static layouts that re-use elements within a strip to recover multiple failed elements in that stripe. Note that MDS codes are much simpler in this regard: any k elements can recover any number of other failed elements.

V. Analysis

In our analysis, we focus on two- and three-disk fault tolerant codes. We compare flat XOR-codes to MDS codes for stripes based on $1 \leq k \leq 30$. When $k = 1$, MDS codes are three- and four-fold replication (with $m = 2$ and $m = 3$ respectively). When $k > 1$, the MDS codes are RAID 6 ($m = 2$) and beyond ($m = 3$). Table I lists the codes we analyze. Checkmarks indicate which codes are included in two- and three-disk fault tolerant analysis. Pointers to the appropriate sections of this paper and to the original description of the code are listed as well. Flat parity-check array code constructions depend on some prime number or other constant. When we analyze such codes for a given k, we select the most space-efficient construction that is large enough to store k data elements. I.e., we use the smallest prime number or other constant that generates a code that we can then shorten to the appropriate k value. For SPC codes, we only analyze "square" constructions.

Code	2DFT	3DFT	Comment
MDS	✓	✓	Replication, RAID 6, and beyond.
StepComb	✓	✓	Section III-C
HDComb	✓	✓	Section III-C
Chain	✓	✓	Section III-A
Flat (square) SPC	✓		Section III-B and [50]
Flat RDP	✓		Section III-B and [36]
Flat XCODE	✓		Section III-B and [37]
Flat STAR		✓	Section III-B and [38]

TABLE I
ERASURE CODES ANALYZED.

We analyze the following storage properties of the erasure code constructions:

1) Relative storage overhead (Section V-B): The amount of data stored relative to a single replica. I.e., the cost of a full stripe write.
2) Small write cost (Section V-C): The average number of parity elements that need to be updated when a single data element is updated.
3) Average size of shortest recovery equation (Section V-D): The fewest number of disks that must be accessed to recover an element. I.e., the number of disk's worth of data that must be read to recover one failed disk.
4) Average recovery read load (Section V-E): The amount of data relative to an entire disk's worth of data, that an available disk must read to recover one failed disk within the stripe.
5) Fault tolerance at the Hamming distance (Section V-F): The fraction of three- and four-disk failures that lead to data loss for two- and three-disk fault tolerant codes respectively.

We summarize all of our analyses for a specific number of data disks ($k = 15$) in Section V-G.

A. Computation

Even though computation costs of erasure codes receive the bulk of attention from coding theorists, we believe that they are rarely the bottleneck in storage systems. Whether the CPU that performs erasure coding is firmware in a RAID controller or a library on a general-purpose CPU, we believe that bandwidth constraints in cache, the memory bus, on the network, or from the storage devices dictate performance. Beyond this, most reads in erasure-coded storage systems read data elements directly and so require no computation to decode.

To provide a sense of the computational demands of the various codes being analyzed, we counted the minimum number of distinct mathematical operations that must be performed to encode an entire stripe. For a given d, both the computational cost of MDS and non-MDS codes both increase linearly with k and are within a small constant factor of one another. In

practice, the non-MDS codes are likely to be computationally faster to encode (or decode) for two main reasons: one, many of the XOR computations are common across parity elements and so XOR-*Scheduling* can be used [55]; two, the actual mathematical operations performed while encoding an MDS code are heavier weight than the simple XOR performed in the non-MDS codes [30], [56], [57]. In summary, there are moderate differences between computational costs among the codes we are analyzing, but we do not believe these costs affect the decision of which codes to employ in a given storage system.

B. Relative storage overhead

Figure 11 and Figure 12 show the relative storage overhead for two- and three-disk fault-tolerant codes respectively. The y axis on each graph goes up to the relative storage overhead of d-fold replication. A key property of MDS codes is optimal storage overhead. As k increases, the relative storage overhead of MDS codes decreases monotonically.

The Chain code construction has a fixed relative storage overhead of 2. This is because it is constructed with $m = k$. I.e., the Chain code has the same relative storage overhead as two-fold replication, but can achieve either two- or three-disk fault tolerance (depending on the construction).

As the stripe widens (k increases), the relative storage overhead of all the other flat XOR-code constructions reduces. The Stepped Combination construction provides the best relative storage overhead. We believe the Stepped Combination construction uses the minimal number of parity elements possible to achieve the desired fault tolerance using only simple XOR operations upon data elements to produce parity elements. If k gets large enough, then a construction like the Stepped Combination code exhibits storage overhead like LDPC codes [14]. At $k = 30$, the three-disk fault tolerant StepComb code has 12.4% more storage overhead than the MDS code, and the four-disk fault tolerant StepComb has 11.8% more.

The storage overhead of the flattened parity-check array codes and combination codes do not decrease monotonically. This is because we plot the most space-efficient construction for each of these specific code constructions that can store k data elements.

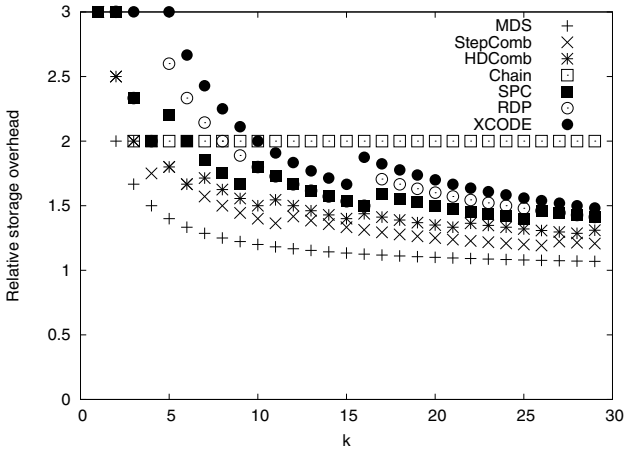


Fig. 11. Relative storage overhead for $d = 3$.

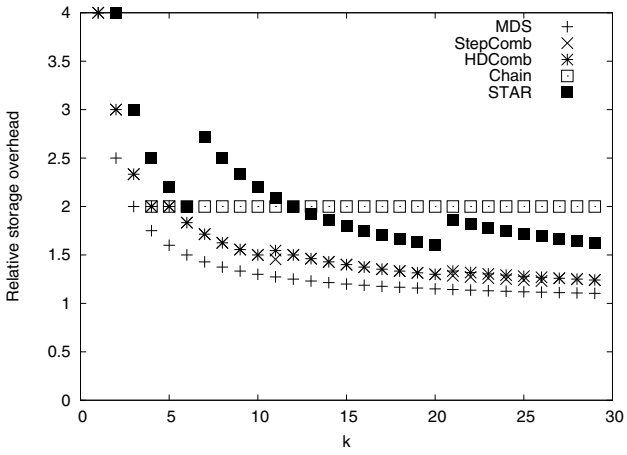


Fig. 12. Relative storage overhead for $d = 4$.

C. Small write cost

Two key performance metrics of erasure-coded storage are small write costs and strip write costs [44]. A small write is a write that updates a single data element and all necessary parity elements. A strip write is a write that updates a column of (data) elements and all necessary parity elements. In traditional parity-check array codes, these performance metrics are different. For flat XOR-based codes and MDS codes, these performance metrics are identical because each strip consists of a single element.

To calculate the small write cost, we count the number of parity elements that each data element must update if it is updated. An average is reported because for flat XOR-codes, different data elements have different small write costs. Figure 13 and Figure 14 show the average small write cost for two- and three-disk fault-tolerant codes respectively.

As with relative storage overhead, MDS codes are optimal in this property: exactly $d-1$ parity elements must be updated whenever a data element is updated. Many of the flat XOR-codes are also optimal: HDComb, Chain, SPC, and XCODE. Each of these XOR-codes was constructed to achieve optimal update complexity and so this result is not surprising. For some values of k, the remaining XOR-codes are also optimal: StepComb, RDP, and Star.

For values of k that the StepComb construction is identical to the HDComb construction, it achieves optimal small write costs. Over the range of k values analyzed, three-disk fault tolerant StepComb is always less than 50% worse than optimal. Whereas four-disk fault tolerant StepComb is always less than 17% worse than optimal.

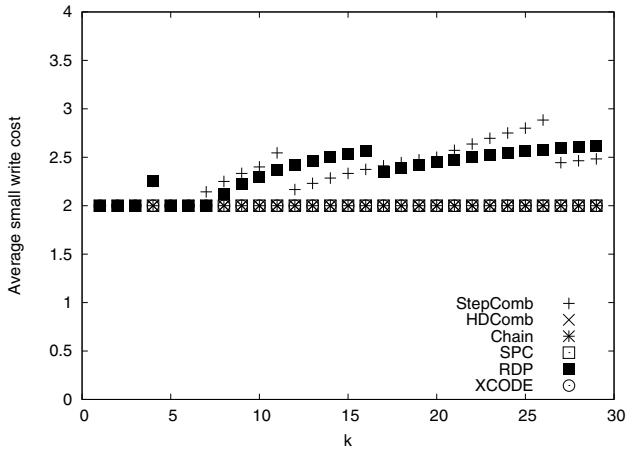


Fig. 13. Small write costs for $d = 3$.

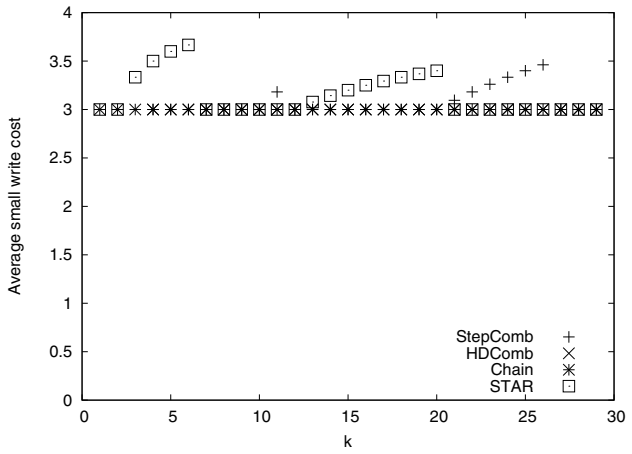


Fig. 14. Small write costs for $d = 4$.

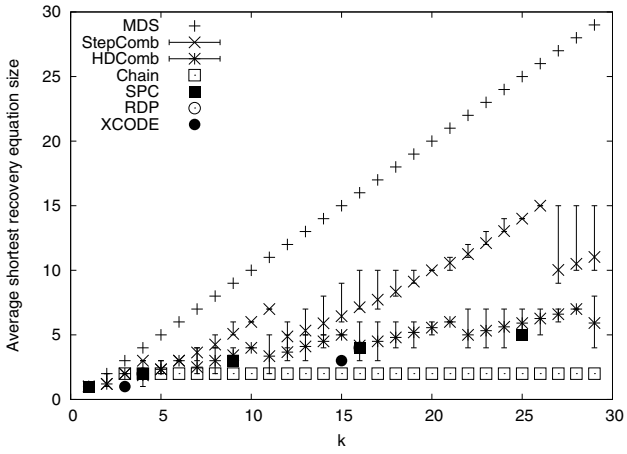


Fig. 15. Average shortest recovery equation size $d = 3$.

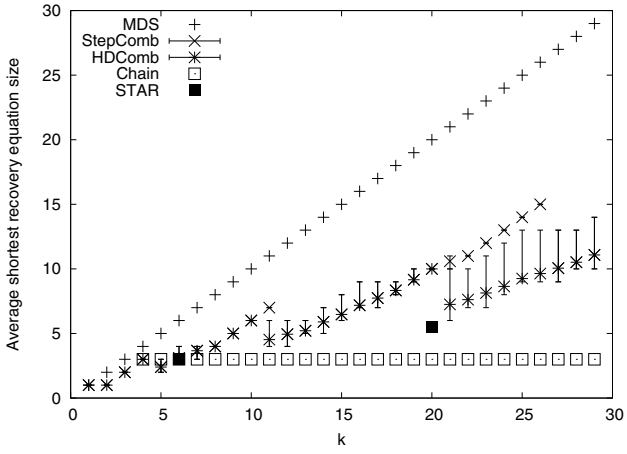


Fig. 16. Average shortest recovery equation size for $d = 4$.

D. Recovery equation size

We use the RE Algorithm to analyze all the flat XOR-codes to find all of the recovery equations for each element. We then calculate the average size of the smallest recovery equation. We are interested in the size of the smallest recovery equations because they indicate the potential for choosing a recovery equation over a small read, and are the best-case for how many disk's of data must be read to recover a failed disk. To put this in context, consider replication: each element in x-fold replication has $x - 1$ recovery equations of size one. I.e., one disk's worth of data needs to be read to recover a failed disk. Replication has the smallest recovery equations of any code.

Figure 15 and Figure 16 show the results for two- and three-disk fault tolerant codes respectively. The shortest recovery equation for MDS codes increases linearly with k since k elements are required to recover any element. The Chain code has the shortest recovery equations of $d - 1$ since all parity elements are connected to exactly $d - 1$ data elements, and vice versa. Of course, this is only possible due to the storage overhead incurred by the Chain code construction. The flat parity-check array code constructions also offer quite short recovery equations also due to the storage overhead cost.

For most of the codes, the minimum, average, and maximum size of shortest recovery equations across all data elements are the same. For the combination codes, different data elements have different shortest recovery equations and so we use error bars to indicate the size of the minimum and maximum values of the shortest recovery equation across all k data elements. The inverse relationship between storage overhead and shortest recovery equation size continues with the combination codes falling between MDS codes and the other flat XOR-codes, and with HDComb codes having shorter average recovery equations than StepComb codes.

All of these results are based on exactly one element being inaccessible—if multiple elements are inaccessible, then the average shortest recovery equation size will likely increase for flat XOR-codes.

E. Recovery read load

The recovery read load is the amount of data relative to an entire disk's worth of data, that an available disk must

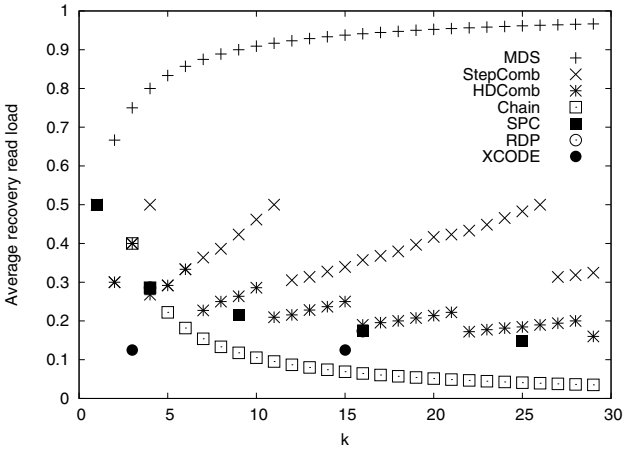


Fig. 17. Average recovery read load for $d = 3$.

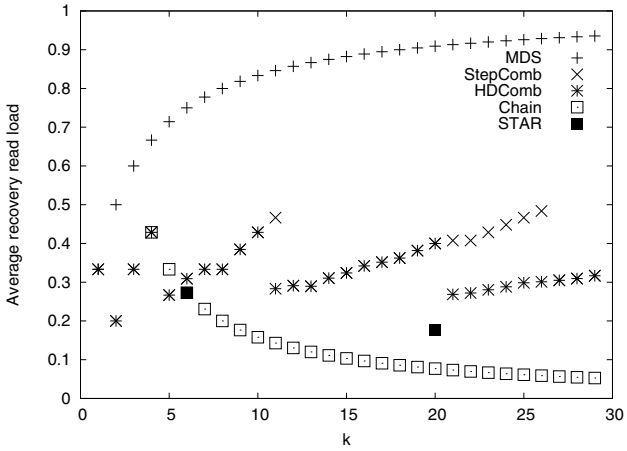


Fig. 18. Average recovery read load for $d = 4$.

read to recover a failed disk within the stripe. The smaller this value is, the quicker recovery can complete, or the less impact recovery has on foreground workload, or a combination of both. Figure 17 and Figure 18 shows the results for two- and three-disk fault tolerant codes respectively. This analysis is again based on a single disk failure. Beyond this, these results assume that code rotation perfectly distributes the recovery read load across available devices (i.e., that the recovery schedule takes full advantage of the average smallest recovery equation).

As k increases, MDS codes approach the case of each available disk having to be read in its entirety to recover a single failed disk. The flat XOR-codes exhibit radically different behavior! Storage overhead has a compound affect: it creates the opportunity for small recovery equations *and* allows the recovery read load to be spread over more disks. The end result is that Chain codes approach the case of each available disk within a stripe having to read a nominal amount of data to recover a single failed disk. The flat parity-check array codes exhibit the same trend, though in a less pronounced manner.

The combination codes are again in between the MDS codes and the other flat XOR-codes. At two-disk fault tolerance, HDComb codes range from 18% to 40% and StepComb codes range from 30% to 50% of a disk's worth of data being read by each available disk. At three-disk fault tolerance, the combination codes range from 25% to 50% along this metric.

Consider each available disk having to read 25% of a disk's worth of data to recover an element. If recovery read disk bandwidth is the bottleneck for recovering the failed disk, then the failed disk can be recovered $4\times$ faster. Alternately, for a given recovery time objective, the recovery workload can be reduced by this same factor.

Note that for d-fold replication ($k = 1$) each available device must read $1/(d-1)$ of a disk's worth of data. For three-fold replication, this is 0.5 and for four-fold replication, this is 0.33.

We have used the RRS Algorithm and SRS Algorithm to produce simultaneous and rotated recovery schedules for many of the codes we analyze in this section. The rotated schedules we have produced thus far have matched the analysis in this section. The simultaneous schedules we have produced thus far have yielded two distinct recovery equations (or the equivalent schedule) for all the flat XOR-codes. Because our current algorithms are computationally intensive, we are still working on producing complete recovery schedules for all of the codes for stripes with k up to 30.

F. Fault tolerance

Flat XOR-codes offer fault tolerance at and beyond the Hamming distance. By this, we mean that a two-disk fault tolerant code may be able to tolerate *some* triple-disk failures (but not all). MDS codes do not tolerate any failures at or beyond the Hamming distance. This is because such codes are optimally space-efficient. The storage overhead of non-MDS codes is what makes fault tolerance at and beyond the Hamming distance possible

Figure 19 and Figure 20 show the fault tolerance, at the Hamming distance, for two- and three-disk fault tolerant codes respectively. We plot the fraction of failures at the Hamming distance that lead to data loss. Notice that the y axes of these graphs are logarithmic and so these graphs are somewhat like the "nines" graphs used to illustrate availability. Results for the flat parity-check array codes are only plotted for constructions that are not shortened (i.e., for the largest possible value of k given the prime used to construct the code).

The fault tolerance at the Hamming distance is inversely correlated with the storage overhead: the more storage overhead of the code construction, the fewer faults at the Hamming distance that lead to data loss. In particular, the Chain code, with the highest storage overhead, offers the most fault tolerance at the Hamming distance.

We do not include fault tolerance beyond the Hamming distance in our analysis because additional fault tolerance at the Hamming distance has the largest affect on overall reliability. Translating fault tolerance of codes into reliability is

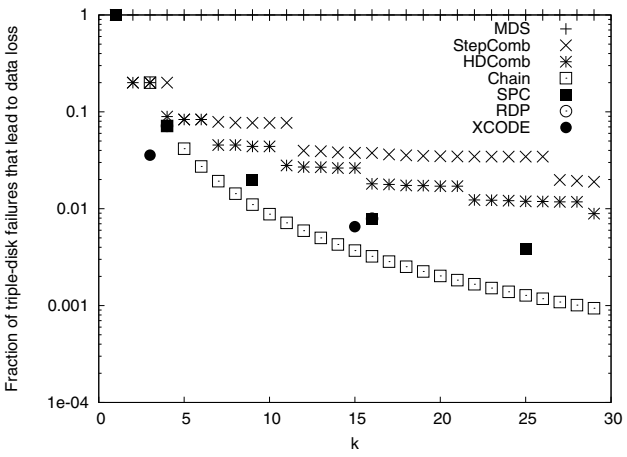


Fig. 19. Fraction of triple-disk failures that lead to data loss.

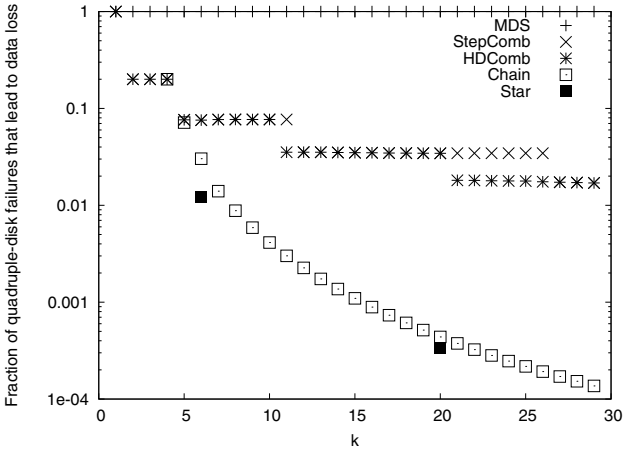


Fig. 20. Fraction of quadruple-disk failures that lead to data loss.

beyond the scope of this paper (though there is prior work [26], [29], [11]).

G. Results summary

At a high level, flat XOR-codes pay some cost in storage overhead, small write, or both relative to MDS codes. By incurring this cost, flat XOR-codes are able to achieve shorter recovery equations, reduced recovery read load, and increased fault tolerance at the Hamming distance. Table II summarizes our analysis for specific codes with $k = 15$: MDS codes, StepComb, HDComb, and Chain codes. We also include Replication (Rep) on a distinct line from MDS in the table. For Replication, we use $k = 1$ which is a much smaller stripe than for the other codes. This set of codes illustrates the various tradeoffs possible between MDS codes and flat XOR-codes. We believe that the properties demonstrated by the StepComb, HDComb, and Chain constructions delineate the possible tradeoff space for flat XOR-codes. I.e., we believe that there exists other flat XOR-codes we did not analyze, and which no one may yet know how to construct, that achieve tradeoff points between these three code constructions.

Code	d	Stor. over.	Small write	Min. RE	Read load	% fail at d
MDS	3	1.13	2.00	15.00	0.94	100
StepComb	3	1.33	2.33	6.45	0.34	3.8
HDComb	3	1.40	2.00	5.00	0.25	2.6
Chain	3	2.00	2.00	2.00	0.069	0.37
Rep ($k = 1$)	3	3.00	2.00	1.00	0.50	100
MDS	4	1.20	3.00	15.00	0.88	100
StepComb	4	1.40	3.00	6.48	0.32	3.5
HDComb	4	1.40	3.00	6.48	0.32	3.5
Chain	4	2.00	3.00	3.00	0.10	1.1
Rep ($k = 1$)	4	4.00	3.00	1.00	0.33	100

TABLE II
SUMMARY OF RESULTS FOR $k = 15$.

Compare two-disk fault tolerant StepComb to MDS: an 18% additional storage overhead and a 17% small write overhead provides recovery equations that are on average 43% the size, a 64% reduction in per-disk read load during recovery, and the ability to tolerate 96.2% of all triple-disk failures. A slightly larger storage overhead allows the HDComb code to achieve optimal small write costs, even shorter recovery equations, further reduced read load during recovery, and slightly better fault tolerance. The Chain code takes this tradeoff to the limit: for the storage overhead of two-fold replication, the Chain code achieves optimal small write costs, and other fantastic properties. Compared to the MDS code, the Chain code decreases recovery equation size 87%, per-disk read load 93%, and tolerates 99.6% of all triple-disk failures.

The comparison of these codes at $d = 4$ demonstrate similar tradeoffs. Though the HDComb and StepComb constructions for this specific value of k are identical.

The replication results for read load and failure are somewhat misleading because of the smaller stripe width. Consider if parity declustering [7] is used with replication to take advantage of 30 disks, the number of disks that the Chain code uses when $k = 15$. The read load for both two- and three-disk fault tolerant replication is then 0.034, even less than for the Chain code. Again considering 30 disks, it is possible to setup 10 distinct three-fold replication stripes. Such a layout only loses data in 0.15% of the possible three disk failures. Again, this is better than the Chain code. Note how the replica layouts for these two 30-disk comparisons differ: replication does not achieve both properties simultaneously like Chain code does.

VI. CONCLUSIONS

As the scale of storage systems increases, two- and three-disk fault tolerant protection, or more, is needed. There is a pronounced gap between the cost-benefit propositions of replication and other MDS codes such as RAID 6 in multi-disk fault tolerant systems. In such systems, flat XOR-codes offer cost-benefit tradeoffs that cover a large portion of the gap between replication and other MDS codes.

In this paper, we described a number of two- and three-disk fault tolerant non-MDS flat XOR-code constructions: Chain

codes, flattened parity-check array codes, Stepped Combination codes, and HD-Combination codes. Chain codes are based on previously known constructions, and flattening is a technique that applies to previously known codes. The combination codes are novel constructions. We expect additional flat XOR-code constructions will be discovered that fill in more of the tradeoff space between replication and other MDS codes.

We introduced recovery equations, recovery schedules and algorithms to determine such equations and schedules for flat XOR-codes. We expect that the initial algorithms we describe can be improved significantly.

A key property of flat XOR-codes is that they trade an increased storage overhead for shorter recovery equations which can be leveraged to create efficient recovery schedules. We analyzed key storage properties of these flat XOR-codes and MDS codes: storage overhead, small write cost, size of shortest recovery equation, recovery read load, and fault tolerance at the Hamming distance.

MDS codes achieve optimal storage overhead and small write costs. Recovery equations for such codes though increase in size with stripe width. The Stepped Combination code achieves minimal storage overhead for a flat XOR-code construction at a given fault tolerance. Whereas, the HD-Combination code achieves minimal storage overhead *and* optimal small write costs for a flat XOR-code construction at a given fault tolerance. The recovery read load is the fraction of a disk's worth of data each available disk in a stripe must read to recover a single failed disk. Combination codes, depending on stripe width and fault tolerance, have a recovery read load of between 0.2 and 0.5. These values are competitive with replication and much better than other MDS codes. Chain codes provide optimal small write costs and optimal shortest recovery equation size, but require storage overhead equivalent to two-fold replication. This storage overhead cost enables Chain codes to achieve a recovery read load that decreases with stripe width and so is better than replication and significantly better than other MDS codes.

REFERENCES

[1] M. Baker, M. A. Shah, D. S. H. Rosenthal, M. Roussopoulos, P. Maniatis, T. Giuli, and P. Bungale, "A fresh look at the reliability of long-term digital storage," in *EuroSys-2006: 1st EuroSys Conference*. ACM, April 2006, pp. 221–234.

[2] B. Schroeder and G. A. Gibson, "Disk failures in the real world: What does an MTTF of 1,000,000 hours mean to you?" in *FAST-2007: 5th USENIX Conference on File and Storage Technologies*. USENIX Association, 2007, pp. 1–16.

[3] E. Pinheiro, W.-D. Weber, and L. A. Barroso, "Failure trends in a large disk drive population," in *FAST-2007: 5th USENIX Conference on File and Storage Technologies*. USENIX Association, 2007.

[4] L. N. Bairavasundaram, G. R. Goodson, S. Pasupathy, and J. Schindler, "An analysis of latent sector errors in disk drives," *SIGMETRICS Perform. Eval. Rev.*, vol. 35, no. 1, pp. 289–300, 2007.

[5] J. Elerath, "Hard-disk drives: The good, the bad, and the ugly," *ACM Queue*, 2009.

[6] J. Menon and D. Mattson, "Distributed sparing in disk arrays," in *COMPCON '92: Proceedings of the thirty-seventh international conference on COMPCON*. IEEE Computer Society Press, 1992, pp. 410–421.

[7] M. Holland and G. Gibson, "Parity declustering for continuous operation in redundant disk arrays," in *Architectural Support for Programming Languages and Operating Systems*, 1992, pp. 23–25.

[8] D. Li and J. Wang, "EERAID: Energy Efficient Redundant and Inexpensive Disk array," in *EW11: Proceedings of the 11th workshop on ACM SIGOPS European workshop*. New York, NY, USA: ACM, 2004, p. 29.

[9] C. Weddle, M. Oldham, J. Qian, A.-I. A. Wang, P. Reiher, and G. Kuenning, "PARAID: A gear-shifting power-aware RAID," *Trans. Storage*, vol. 3, no. 3, p. 13, 2007.

[10] J. Wang, H. Zhu, and D. Li, "e-RAID: Conserving energy in conventional disk-based RAID system," in *IEEE Transactions on Computers*, 2008.

[11] K. Greenan, "Reliability and power-efficiency in erasure-coded storage systems," UC Santa Cruz, Tech. Rep. UCSC–SSRC–09–08, 2009.

[12] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google File System," in *SOSP '03: Proceedings of the 19th ACM Symposium on Operating Systems Principles"*, October 2003, pp. 29–43.

[13] The Apache Software Foundation, "Hadoop File System (HDFS))," http://hadoop.apache.org/hdfs/.

[14] M. G. Luby, M. Mitzenmacher, M. A. Shokrollahi, D. A. Spielman, and V. Stemann, "Practical loss-resilient codes," in *STOC-1997*. ACM Press, 1997, pp. 150–159.

[15] R. G. Gallager, *Low density parity-check codes*. MIT Press, 1963.

[16] J. W. Byers, M. Luby, M. Mitzenmacher, and A. Rege, "A digital fountain approach to reliable distribution of bulk data," in *SIGCOMM '98*. ACM, 1998, pp. 56–67.

[17] "IEEE standard for local and metropolitan area networks part 3: Carrier sense multiple access with collision detection (CSMA/CD) access method and physical layer specifications - section one," *IEEE Std 802.3-2008 (Revision of IEEE Std 802.3-2005)*, pp. c1–597, 2008.

[18] "IEEE standard for local and metropolitan area networks part 16: Air interface for broadband wireless access systems," *IEEE Std 802.16-2009 (Revision of IEEE Std 802.16-2004)*, pp. C1–2004, 2009.

[19] A. Wilner, "Multiple drive failure tolerant RAID system," United States Trademark and Patent Office, December 2001, patent number 6,327,627 B1.

[20] J. L. Hafner, "WEAVER Codes: Highly fault tolerant erasure codes for storage systems," in *FAST-2005*. USENIX Association, December 2005, pp. 212–224.

[21] ——, "HoVer erasure codes for disk arrays," in *DSN-2006: The International Conference on Dependable Systems and Networks*. IEEE, June 2006, pp. 217–226.

[22] A. Amer, J.-F. Pâris, and T. Schwarz, "Outshining mirrors: MTTDL of fixed-order spiral layouts," in *Storage Network Architecture and Parallel I/Os, 2007. SNAPI.International Workshop on*, Sept. 2007, pp. 11–16.

[23] J.-F. Pâris, T. J. E. Schwarz, and D. D. E. Long, "Self-adaptive two-dimensional raid arrays," in *International Performance, Computing, and Communications Conference (IPCCC)*. IEEE, April 2007, pp. 246–253.

[24] J. S. Plank and M. G. Thomason, "A practical analysis of low-density parity-check erasure codes for wide-area storage applications," in *DSN-2004*. IEEE, June 2004, pp. 115–124.

[25] J. S. Plank, A. L. Buchsbaum, R. L. Collins, and M. G. Thomason, "Small parity-check erasure codes - exploration and observations," in *DSN-2005*. IEEE, July 2005.

[26] J. L. Hafner and K. Rao, "Notes on reliability models for non-MDS erasure codes," IBM, Tech. Rep. RJ–10391, October 2006.

[27] M. Woitaszek and H. M. Tufo, "Tornado codes for MAID archival storage," in *24th IEEE Conference on Mass Storage Systems and Technologies*, 2007, pp. 221–226.

[28] J. J. Wylie and R. Swaminathan, "Determining fault tolerance of XOR-based erasure codes efficiently," in *DSN-2007*. IEEE, June 2007, pp. 206–215.

[29] K. M. Greenan, E. L. Miller, and J. J. Wylie, "Reliability of flat XOR-based erasure codes on heterogeneous devices," in *DSN-2008*. IEEE, June 2008.

[30] I. S. Reed and G. Solomon, "Polynomial codes over certain finite fields," *Journal of the Society for Industrial and Applied Mathematics*, vol. 8, no. 2, pp. 300–304, 1960.

[31] J. S. Plank, "The RAID-6 Liberation codes," in *FAST-2008: 6th Usenix Conference on File and Storage Technologies*, February 2008.

[32] ——, "The RAID-6 Liber8Tion code," *International Journal of High Performance Computing Applications*, vol. 23, no. 3, pp. 242–251, 2009.

[33] C.-I. Park, "Efficient placement of parity and data to tolerate two disk failures in disk array systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 6, no. 11, pp. 1177–1184, 1995.

[34] M. Blaum, J. Brady, J. Bruck, and J. Menon, "EVENODD: An efficient scheme for tolerating double disk failures in RAID architectures," *IEEE Trans. Comput.*, vol. 44, no. 2, pp. 192–202, 1995.
[35] M. Blaum, J. Bruck, and A. Vardy, "MDS array codes with independent parity symbols," *Information Theory, IEEE Transactions on*, vol. 42, no. 2, pp. 529–542, Mar 1996.
[36] P. Corbett, B. English, A. Goel, T. Grcanac, S. Kleiman, J. Leong, and S. Sankar, "Row-diagonal parity for double disk failure correction," in *FAST-2004: 3rd USENIX Conference on File and Storage Technologies*. USENIX Association, March 2004, pp. 1–14.
[37] L. Xu and J. Bruck, "X-Code: MDS array codes with optimal encoding," *IEEE Transactions on Information Theory*, vol. 45, no. 1, pp. 272–276, 1999.
[38] C. Huang and L. Xu, "Star: an efficient coding scheme for correcting triple storage node failures," in *FAST'05: Proceedings of the 4th conference on File and Storage Technologies*. Berkeley, CA, USA: USENIX Association, 2005, pp. 15–15.
[39] C. Jin, H. Jiang, D. Feng, and L. Tian, "P-Code: a new RAID-6 code with optimal properties," in *ICS '09: Proceedings of the 23rd International Conference on Supercomputing*. New York, NY, USA: ACM, 2009, pp. 360–369.
[40] M. Schwartz and A. Vardy, "On the stopping distance and the stopping redundancy of codes," *IEEE Trans. on Inf. Theory*, vol. 52, no. 3, pp. 922–932, 2006.
[41] H.-I. Hsiao and D. DeWitt, "Chained Declustering: A new availability strategy for multiprocessor database machines," in *Proceedings of 6th International Data Engineering Conference*, 1990, pp. 456–465.
[42] A. Amer, J.-F. Pâris, D. Long, and T. Schwarz, "Progressive parity-based hardening of data stores," in *Performance, Computing and Communications Conference, 2008. IPCCC 2008. IEEE International*, Dec. 2008, pp. 34–42.
[43] C. Huang, M. Chen, and J. Li, "Pyramid codes: Flexible schemes to trade space for access efficiency in reliable data storage systems," in *Network Computing and Applications, 2007. NCA 2007. Sixth IEEE International Symposium on*, July 2007, pp. 79–86.
[44] J. L. Hafner, V. Deenadhayalan, T. Kanungo, and K. Rao, "Performance metrics for erasure codes in storage systems," IBM, Tech. Rep. RJ–10321, 2004.
[45] D. A. Patterson, G. Gibson, and R. H. Katz, "A case for Redundant Arrays of Inexpensive Disks (RAID)," in *ACM SIGMOD International Conference on Management of Data*, June 1988, pp. 109–116.
[46] W. A. Burkhard and J. Menon, "Disk array storage system reliability," in *Symposium on Fault-Tolerant Computing*, 1993, pp. 432–441. [Online]. Available: citeseer.ist.psu.edu/burkhard93disk.html
[47] J. F. Elerath and M. Pecht, "Enhanced reliability modeling of raid storage systems," in *DSN-2007*. IEEE, June 2007, pp. 175–184.
[48] K. Rao, J. L. Hafner, and R. A. Golding, "Reliability for networked storage nodes," in *DSN-2006: The International Conference on Dependable Systems and Networks*. IEEE, June 2006, pp. 237–248.
[49] J. S. Plank, "Erasure codes for storage applications," Tutorial slides, presented at *FAST-2005: 4th Usenix Conference on File and Storage Technologies*, December 2005.
[50] P. Elias, "Error free coding," *IRE Trans. Inform. Theory*, vol. IT-44, pp. 29–37, Sept. 1954.
[51] G. A. Gibson, L. Hellerstein, R. M. Karp, and D. A. Patterson, "Failure correction techniques for large disk arrays," in *ASPLOS-III: Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*. New York, NY, USA: ACM, 1989, pp. 123–132.
[52] J. L. Hafner, V. Deenadhayalan, K. Rao, and J. A. Tomlin, "Matrix methods for lost data reconstruction in erasure codes," in *FAST-2005*. USENIX Association, December 2005, pp. 183–196.
[53] C. Fragouli, J.-Y. Le Boudec, and J. Widmer, "Network coding: An instant primer," *SIGCOMM Comput. Commun. Rev.*, vol. 36, no. 1, pp. 63–68, 2006.
[54] A. Dimakis, V. Prabhakaran, and K. Ramchandran, "Decentralized erasure codes for distributed networked storage," *Information Theory, IEEE Transactions on*, vol. 52, no. 6, pp. 2809–2816, June 2006.
[55] J. Luo, L. Xu, and J. S. Plank, "An efficient XOR-Scheduling algorithm for erasure codes encoding," in *DSN-2009: The International Conference on Dependable Systems and Networks*. Lisbon, Portugal: IEEE, June 2009.
[56] K. Greenan, E. L. Miller, and T. Schwarz, "Optimizing Galois Field arithmetic for diverse processor architectures," September 2008.
[57] J. S. Plank and L. Xu, "Optimizing Cauchy Reed-Solomon codes for fault-tolerant network storage applications," in *NCA-2006: 5th IEEE International Symposium on Network Computing Applications*. IEEE, July 2006, pp. 173–180.

APPENDIX I
CORRECTNESS SKETCHES FOR COMBINATION CODES

Below we outline correctness sketches for the two- and three-disk fault tolerant Stepped Combination code constructions. These sketches also apply to the corresponding HD-Combination code constructions.

A. Stepped Combination - Hamming distance 3

Consider the following three scenarios of disk failures: (1) Two parity disks fail. In this case, since no data disks fail, all parity disks can be easily recovered. (2) One data disk fails and one parity disk fails. Since every data disk is connected to at least two parity disks, there exists a working parity disk that is connected to the failed data disk. Therefore, the failed data disk can be recovered, and then so can the failed parity disk. (3) Two data disks fail. Since no two data disks are connected to the same set of parity disks, there exists a parity disk that is connected to exactly one of the failed disks. Therefore, that disk can be recovered, and then so can the other one.

This construction is storage-efficient optimal in the sense that it achieves the maximum value of k. The reason is that in order to recover two disk failures, all data disks have to be connected to at least two parity disks and each data disk must be connected to a a unique combination of parity disks. The above construction includes all such possible connections.

B. Stepped Combination - Hamming distance 4

Again, we consider all the disk failure scenarios: (1) Three parity disks fail. Since no data disks fail, all parity disks can be easily recovered. (2) Two parity disks and one data disk fail. Since every data disk is connected to at least three parity disks, there exists a working parity disk that is connected to the failed data disk. Therefore, that data disk can be recovered, and then so can the two failed parity disks. (3) One parity disk and two data disks fail. Since the two sets of parity disks connected to any two data disks differ by at least two parity disks, there exists a working parity disk that is connected to exactly one of the failed data disk. Therefore, that data disk can be recovered, and then so can the other failed data disk and the failed parity disk. (4) Three data disks fail. Since all three data disks are connected to an odd number of parity disks, there exists a parity disk that is connected to exactly an odd number (i.e., one or three) of these three failed data disks. We further consider two cases: (4.1) Suppose no parity disks are connected to exactly one failed data disk. Then there exists a parity disk that is connected to all three failed data disks, and there exists another parity disk that is connected to exactly two of the three failed data disks (but not to the third one). In this case, the third data disk can be first recovered, and then so can the other two. (4.2) Suppose there exists a parity disk that is connected to exactly one of the failed data disks. In this case, that data disk can be recovered first, and then so can the other two.

POTSHARDS : Storing Data for the Long-term Without Encryption

Kevin Greenan Mark Storer Ethan L. Miller Carlos Maltzahn

Storage Systems Research Center
Computer Science Department
University of California, Santa Cruz

Abstract

Many archival storage systems rely on keyed encryption to ensure privacy. A data object in such a system is exposed once the key used to encrypt the data is compromised. When storing data for as long as a few decades or centuries, the use of keyed encryption becomes a real concern. The exposure of a key is bounded by computation effort and management of encryption keys becomes as much of a problem as the management of the data the key is protecting. POTSHARDS is a secure, distributed, very long-term archival storage system that eliminates the use of keyed encryption through the use of unconditionally secure secret sharing. A (m,n) unconditionally secure secret sharing scheme splits an object up into n shares, which provably gives no information about the object, unless m of the shares collaborate.

POTSHARDS separates security and redundancy by utilizing two levels of secret sharing. This allows for secure reconstruction upon failure and more flexible storage patterns. The data structures used in POTSHARDS are organized in such a way that an unauthorized user attempting to collect shares will not go unnoticed since it is very difficult to launch a targeted attack on the system. A malicious user would have a difficult time finding the shares for a particular file in a timely or efficient manner. Since POTSHARDS provides secure storage for arbitrarily long periods of time, its data structures include built-in support for consistency checking and data migration. This enables reliable data churning and the movement of data between storage devices.

Keywords : Data Security, Distributed Storage, Secure Storage, Survivable Storage

1 Introduction

In today's computing environment, more and more data is being migrated from hard copy to digital form. This trend is catalyzed by a number of motivations revolving around economic efficiency. Digital data offers many economic advantages compared to traditional, tangible publishing methods such as books, magazines, and films. Digital data is often much easier and cheaper to transport and store in comparison to traditional mediums such as paper and ink based printing. In many cases, digital content is also cheaper to produce. Many forms of traditional content delivery such as books and magazines are already created and edited as digital data; thus the production of a hard-copy simply adds additional costs.

Despite the advantages of digital content, tangible hard-copies of data do offer at least one major advantage over digital data. Archivists have, over many years, developed a diverse set of strategies for preserving hard-copies of data. Additionally, archivists have become very adept at judging degradation of hard-copy media through empirical observation and testing. The relative newness of computer data presents many challenges as digital archivists develop the tools and techniques to preserve digital data.

The access patterns of archival storage are distinctly different from general purpose storage. Archival storage is heavily write-centric—information is written to an archive and it may be a long time, if ever, before that data is accessed from the archive. An example of this would be a collection of business documents that must be preserved for legal reasons even though they are rarely, if ever, requested. This is the exact opposite of the model of shared storage for a distributed application or content distribution in which the access patterns would be heavily skewed towards reading or editing data For example, a web page may be written once to a storage system but read many times by many different clients. Additionally, archival storage is less concerned with throughput and latency than it is with ensuring data persistence, integrity and security.

This paper introduces the POTSHARDS (Protection Over Time, Securely Harboring And Reliably Distributing Stuff) project, an archival storage system designed for a computing environment where relatively static data must be preserved for an indefinite period of time. POTSHARDS separates data redundancy and data secrecy and utilizes a geographically distributed array of network at-

tached storage devices, called *archives*. The first phase of data storage involves the use of a secret sharing algorithm to ensure data secrecy. This avoids the problem introduced by keyed cryptography where the key represents a single point of failure which could render data recovery infeasible. This also helps avoid the problem of preserving historic keys associated with an archive of encrypted files. The second phase of data storage in POTSHARDS utilizes a data redundancy algorithm to ensure data persistence. The longevity of the data within the system is ensured through the redundancy inherent to secret sharing schemes and to aggressive consistency checking.

In order to understand the methods we propose to use in POTSHARDS, an elementary understanding of secret sharing is necessary. Although we chose to not bound POTSHARDS to use any single secret sharing algorithm, two popular algorithms will be quickly explained. We assume that POTSHARDS will be equipped to use any secret sharing scheme.

A rather simple approach to sharing a b-bit secret data block is to generate $n-1$ random b-bit blocks and XOR the blocks to the secret data block as follows: $P = rand_1 \bigoplus rand_2 \bigoplus \cdots \bigoplus rand_{n-1} \bigoplus secret$. The $n-1$ *rand* blocks and the result P could be distributed among n participants and the *secret* block could be tossed away. In this case, an attacker would need all n blocks in order to reconstruct the secret. Any number of the blocks less than n will not reveal anything about the secret. This scheme works very well for security-centered storage.

Shamir's secret sharing scheme is often called an (m,n)-threshold scheme [12, 8], since $m \leq n$ of the original n shares are needed to reconstruct the secret, where m is chosen when the shares are created. Shamir's scheme is based on polynomial generation and interpolation. First, a random polynomial of degree $m-1$ is created by generating $m-1$ random coefficients $c_1, c_2, \ldots, c_{m-1}$ and placing the secret at coefficient c_0 in the polynomial. m participants can collaborate to generate the interpolation polynomial $P_{m-1}(x)$. The secret is revealed by evaluating $P_{m-1}(0)$. If fewer than m participants collaborate, then the secret will not be revealed, since at least m of the shares are required to reconstruct the secret.

As of now, the first level of splitting requires a secret sharing scheme similar to the two schemes covered. The second level of splitting can be use any form of redundancy, such as Reed-Solomon encoding or Shamir's scheme. Obviously, using the XOR-based secret sharing scheme would not be sufficient for the second level of splitting. As we will show, each object written to POTSHARDS is subject to two levels of splitting, where *fragments* are created at the first, secure split and *shards* are a product of splitting *fragments* for redundancy.

2 Related Work

The design concepts and motivation of the POTSHARDS project borrow from various research projects. These projects range from general purpose distributed storage systems, to distributed content delivery systems, to archival systems designed for very specific uses.

A number of systems such as OceanStore [5], FarSite [1], and PAST [9] rely on the explicit use of keyed encryption to provide file secrecy. While this may work reasonably well for short-term file secrecy it is less than ideal for the very long-term storage problem that POTSHARDS is addressing. Further evidence that POTSHARDS is designed for a different application can be found in the design choices made by the authors of the systems mentioned previously. For example, in OceanStore straight replication was chosen in favor of erasure coding in order to provide for better read performance. In contrast, the design emphasis on POTSHARDS is reliability for very long-term storage.

Another class of storage projects that use distributed storage techniques but rely on keyed encryption for file secrecy do not provide any method for insuring long-term file persistence. These systems, such as Glacier [4] and Freenet [3] are designed to deal with the specific needs of content delivery as opposed to to the requirements of long-term storage. An archival storage system must explicitly address the problem of insuring the persistence of the system's contents.

Another class of systems is aimed at long-term storage but with the explicit goal of open content. Systems such as LOCKSS [7], and Intermemory [2] are designed around preserving digital data for libraries and archive where file consistency and accessibility are paramount. These systems are developed around the central idea of very long-term access for public information and thus file secrecy is explicitly not part of the design.

The PASIS architecture [13] and the work of Subbiah and Blough [11] avoids the use of keyed encryption by using secret sharing threshold schemes. While this prevents the introduction of the singular point of failure that keyed encryption introduces to a system, the design of these system only use one level of secret sharing. In effect this combines the secrecy and redundancy aspects of the systems. While related, these two elements of security are, in many respects, orthogonal to one another. Combining the secrecy and redundancy aspects of the system also has the possible effect of introducing compromises into the system by restricting the choices of secret sharing schemes. By separating secrecy and redundancy, an implementation of POTSHARDS is able to utilize a security mechanism optimized for redundancy or secrecy.

3 Design Goals

3.1 Assumptions

One of the motivating ideas of the POTSHARDS project is the need for secure, very long-term storage. To this end certain assumptions are made out of the understanding that very long-term storage must take into account advances in computing technology. These advances in technology are difficult to predict, but POTSHARDS is made immune to them by specifying policy as opposed to mechanism. Five key policies are outlined below along with the assumptions related to the mechanisms that enforce the policy.

The first policy is related to authentication. POTSHARDS assumes that authentication is provided by the host system. The mechanism for this policy may be as simple as a security guard that verifies the identity of a user or it may be a much more advanced authentication system using cryptographic primitives. By specifying policy instead of mechanism this detail is left to the implementation. Part of the advantage of assuming that the system includes correct authentication is that, in a very long-term storage system, the file lifetimes may be far longer than the effective lifetime of a user account. For example, an employee of a company may store an important document in POTSHARDS, but it is unlikely that the user will still be a valid employee decades later. In this scenario, file ownership runs the risk of becoming little more than a historical side-note of file origins. The contents of POTSHARDS are designed to be protected through security policies that designate security clearance. This allows much of the problem of file access to encapsulated in the authentication layer.

The second policy is related to network traffic. POTSHARDS assumes that all communication between nodes in the system is secure. While keyed encryption is a weakness for long-term file storage, encryption is very effective for securing network traffic; network traffic might be secured through the use of session keys as is done in SSL. The nature of keyed encryption makes it very useful for short-term security of replaceable data. For example, if the session keys of a secured communication are lost, it is a relatively straightforward procedure to generate new keys and restart the communication. In contrast, if the encryption keys for an encrypted file are lost it may not be possible to recover the file in a timely manner.

The third assumption is based on the nature of computing technology. POTSHARDS assumes that failures will occur in the system. While most systems take into consideration some level of disaster recovery, the very long-term nature of POTSHARDS and unpredictable growth of technology dictate that the system must be designed to accommodate the failure of any of its subsystems. Failures might include catastrophic failure of part of the system or Byzantine failures caused by a comprised component of the system.

The fourth assumption is that POTSHARDS is designed expressively for use as an archival storage system and thus places its design emphasis upon longevity and security. It is not designed for interactive use as a low-latency file server. The design of POTSHARDS largely considers performance in the interactive time-scale a moot point. A likely usage scenario could include a user requesting a file to be delivered at a later time when processing has completed. When faced with a design compromise POTSHARDS will opt in favor or longevity and security over throughput speed.

Finally, data may be exposed if all or a subset of the archives collude. In the case that all of the archives collude, it is possible to expose all of the information stored by POTSHARDS. The archive collusion property is both necessary and potentially dangerous. When an authorized subject requests a particular object, the archives holding the shards for that object must collude, which must result in a properly reconstructed object. On the other hand, it would be unfavorable to have archives unexpectedly collude. In the case of unexpected collusion, we assume an archives main interest is colluding only when an authorized subject requests an object. In other words, an archive would gain very little by fulfilling unauthorized requests.

3.2 Security and Replication

Storing data securely is one of the most important aspects of a long-term archival storage system, and keyed encryption is a common method of storing data securely. Unfortunately, given the lifetime of the data being stored in an archival storage system, keyed encryption may not be sufficient, due to the single point of failure introduced when encrypting with a key. Keyed encryption relies on the computational effort required to determine the key. Given enough time and computing power, an adversary might be able to compute the key for a given set of data. Often times advances in technology drastically reduce the time it takes to obtain the encryption key. For example while the DES standard using a 56-bit key was considered secure in 1977 it was only 22 years later that a cooperative effort managed to locate a decryption key in less than twenty-three hours. [10] Even 128–256 bit symmetric keys might, at some future time, be trivial to break using as-yet undiscovered algorithms, quantum computers, or biologically-based computers, among other possibilities.

Some may argue that new encryption algorithms may be applied as the old ones are broken. Unfortunately, every time a new algorithm is applied, all data in the system

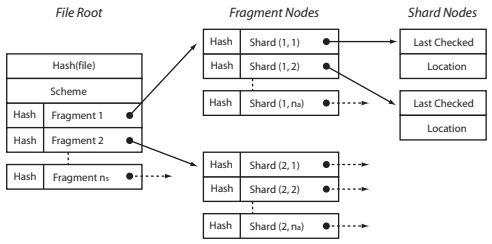


Figure 1: Shard information such as integrity data and location stored in a tree data-structure.

must be re-encrypted, which is a potential housekeeping nightmare.

In contrast, it is possible to provably store the data securely using secret sharing. Instead of relying on computational effort and the latest encryption algorithms, we can rely on the fact that an adversary would need to collect all of (or a subset of) the shares. In addition, shares can be distributed such that an adversary would have trouble effectively finding all of the shares and would not go unnoticed while launching an attack on the storage system.

Systems such as PASIS [13] combine security and redundancy using general threshold schemes. In this case, an object will be split into *n* shares, which are then given out to *n* shareholders. If any *m* shares are recovered, the original object can be reconstructed. Thus, security is accomplished by handing the shares to *n* trusted shareholders and data redundancy is accomplished by requiring only *m* of the *n* shares for reconstruction. In contrast, we aim to separate security and redundancy into two separate steps. Such a scheme will allow for the reconstruction of an object during failure without requiring knowledge of all shares created while encoding for security. Such separation also enables a system to parameterize the threshold for security and redundancy independently.

3.3 Data Structures

The shares of an object can be organized hierarchically in a tree-like structure, as shown in figure 1, which fits naturally into a two level splitting scheme. However on closer inspection this introduces several problems. For example, in a tree-like structure an object can link to its secrecy-centric fragments, which are in turn each linked to their redundancy-centric shards. In this scheme, the amount of information obtained is dependent on what level is compromised. If a fragment is compromised, then all of its shards are compromised. If the object root is compromised, then it is very possible that an adversary can reconstruct the original object.

We would like to organize the data such that its position in the data structure will not expose any information about its origin. This can be easily accomplished using a linear structure such as the one shown in figure 3. Each node in the list would have two outgoing links and two incoming links, which connect to two neighbors. Thus, if a node in the structure is compromised, then the only information exposed is that of its two neighbors. With this structure, it would be beneficial to enforce a policy that neighbor nodes not be related. An object can be reassembled by assigning a name to each shard, which allows an authorized entity to collect the correct shards and reconstruct the object.

3.4 Data Migration

We assume directed attacks will eventually occur within a long-term archival storage system. To make the task of reassembling the shards more difficult, we can make use of our data structures to churn the shards. Since the physical location of the shards does not affect object reconstruction, we can randomly migrate shards throughout the list. Such migration not only creates difficulties for directed attacks, but it can also be used as a load balancing mechanism. Migrating shards such that no single point of failure exists for an object would also be beneficial, but may be difficult to accomplish without exposing too much information about the shards. A possible solution would be to assign a system-wide failure group to each shard, which is checked upon insertion into the system's data structures. These problems will be covered in subsequent sections.

In addition to slowly churning the shards, such a system would also need to support the migration of data from one form of storage to another. As previously stated, migration between different storage devices is pivotal in POTSHARDS. We assume that failures will eventually occur and current storage technologies will one day be replaced, thus it would be to our advantage to ensure that data can be moved between any storage medium.

3.5 Malicious Attack Survivability

A key element to POTSHARDS survivability is its distributed nature. To ensure the survivability of the contents of the system, two key design elements related to the distributed nature of POTSHARDS must be enforced. These two design features relate to malicious attacks on the system.

The first design feature that must be present is that it must be very difficult to launch a targeted attack against POTSHARDS. This feature entails several aspects and is important as protection from unauthorized data access. An assumption made in this area is that activity in the system is being monitored and strange behavior can be detected and acted upon. If a malicious user is attempting to access data for which they are not authorized, the attack strategies available to the attacker should take a suffi-

ciently long time that an alarm would be raised. For example, a brute force attack where the malicious user attacks each storage node could be detected and the attacker isolated before sufficient shares are obtained to reconstruct data. Thus, the design of POTSHARDS should make it difficult to launch a time-efficient targeted attack on the system.

The second design feature that must be present is that the distributed nature of POTSHARDS must not introduce a single point of failure into the system. Any such single element would introduce a vulnerability to denial of service attacks into the system. Each of POTSHARDS subsystems and the system as a whole must be robust in design and implementation to resist an attack on any single point which could prevent access to the contents of the system.

4 Preliminary Design

The POTSHARDS design is still in the early stages of development. Even though we have not thoroughly covered all design aspects of the system, we have some idea of how the data will be organized. This section will cover techniques for splitting the data into storage units called shards, writing objects to the system, retrieving objects from the system and some of the basic data structures used for data management.

4.1 Securely Splitting the Data

The POTSHARDS system stores files as a series of fixed sized data blocks. These blocks of data are produced through two levels of data processing: the first tuned for security and the second for redundancy. The product of the first level of splitting is a set of *fragments*. These fragments are hashed to form a unique fragment identifier. Each fragment is then split into a set of *shards*, with each shard holding its source fragment's identifier. Thus, a set of shards can be used to reconstruct a fragment when a failure occurs. This process is illustrated in Figure 2.

Using two levels of splitting provides a number of interesting and useful properties. First, each object to be stored by the system can be tuned for a particular storage strategy. For example, a file that can be reproduced with relative ease but contains content that must be secure can be tuned for maximum secrecy while saving space by sacrificing a bit of redundancy. Second, in the event of a failure, an individual fragment for a file can be reconstructed without exposing any additional information about the original file. This can be very useful for online consistency checkers. If a number of shards are found to be corrupt, the remaining shards can be used to regenerate the fragment. This fragment does not expose any

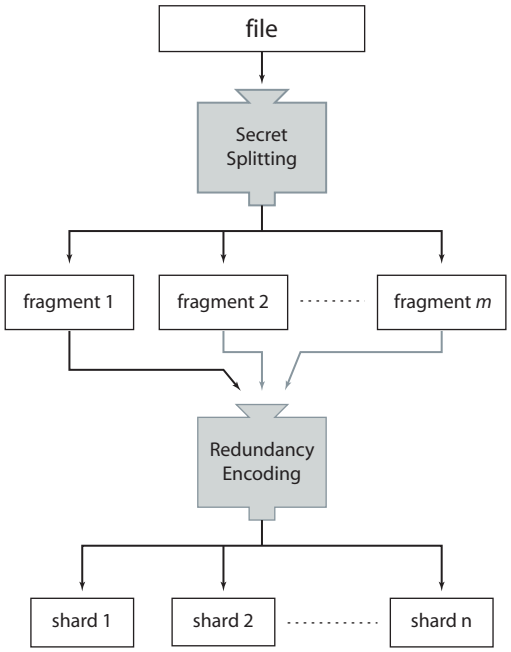


Figure 2: Splitting an object into fragments using secret splitting and fragments into shards using redundancy encoding.

information about the contents of the original file and can be used to regenerate and redistribute the shards. Since the contents of the file are not exposed in this process it could potentially be done automatically without the need for user intervention.

4.2 Fragment Identifier Lists

A list of fragment identifiers is created when an object is added to the system. This fragment identifier list is constructed during the first level of splitting. A fragment identifier is added to an object's fragment identifier list when a fragment is created for the object. In addition, as shown in Figure 2, a fragment identifier is concatenated with a shard when the shard is created during the second level of splitting. Such placement of the fragment identifiers allows one to identify the shards needed to reconstruct the fragments for a given object. The use of the concatenated fragment identifier is explained in the next section.

4.3 Storing the Shards

We propose to organize data in a distributed, circular, doubly-linked list. An example of the basic structure is given in Figure 3. As shown in Figure 4, each node in the list contains a pointer to its predecessor, a pointer to its successor, a unique identifier, a shard, and a list of fragment identifiers representing fragments constructed using

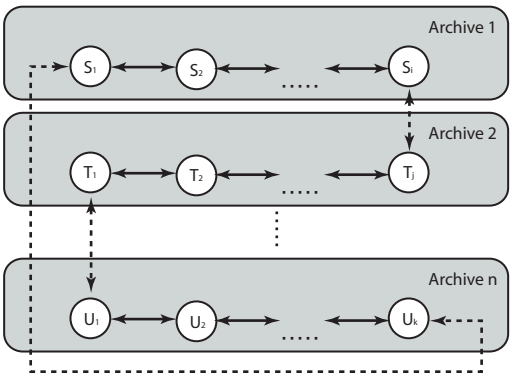


Figure 3: Data structure for organizing shares on a set of archives.

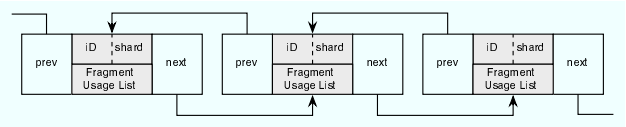


Figure 4: Individual nodes of the list containing the shards.

the contained shard. Each node in the list contains a list of fragment identifiers, because we would like to reuse shards to conserve space. Thus, it is possible for a shard to be shared between two distinct fragments. Remember to note that the shards are essentially random, thus sharing shards between fragments does not reveal any information about the object. The unique identifier is currently a cryptographic hash of the shard. Note that the cryptographic hash is used for verification at the shard level.

As shown in Figure 3, each archive holds a fraction of the list locally, where the last node on the local list points to another list on a different archive. Each node in the list contains a shard generated through the two-level splitting process, a unique identifier and a list of fragment identifiers. The contents of the list are probabilistically churned periodically to ensure that the nodes are distributed as uniformly as possible throughout the list. The contents of the nodes can be churned by specialized processes and during normal operations (i.e. searches, inserts, etc.). It is assumed that all operations performed on an archive are properly authenticated. Thus, in order to traverse the entire list, the subject traversing the list will have to authenticate with every archive in the system multiple times, once for each shard. Since the shards are periodically churned, it would be very difficult for an adversary to efficiently reconstruct any of the objects. The details involving the functionality of each archive is left to future work.

Given the structure of the data within the system, there are currently two methods of verification. First, since a node's identifier is a hash of the shard, it can be used to verify the contents of each node in the list. A more powerful method of verification involves the concatenation of the fragment identifier with the fragment before performing the second level of splitting. If the shards are created with the fragment identifier embedded, verification can occur on the fragment level. A process can collect the shards for a randomly chosen fragment by searching the list for a particular fragment identifier. The fragment identifier constructed after combining the shards would then simply be compared to the search key used when collecting the shards. Note that by randomly reconstructing fragments for verification purposes, no information about any of the objects is revealed, since a fragment only represents a single piece of many needed to actually reconstruct the object.

4.4 Creating Objects in the System

As stated in the previous section, two levels of splitting will break an object up into many pieces, called shards. We must now concern ourselves with how to store these pieces in an efficient manner. Simply storing the shards created in the second level of splitting can provide an efficient and straightforward storage solution. With reference to Figure 2, the fragments are thrown away and the shards and their respective fragment identifiers are handed off to the storage layer of the system. The shards are then inserted into the distributed, doubly-linked list in a probabilistic manner.

Access to the original contents of an object will require all of the fragments generated in the first level of splitting. Since only the fragment identifiers and shards are stored, the fragments must be reconstructed from the shards. Thus, authorization to first obtain the fragment identifier list of an object is necessary along with the authorization for the system to reconstruct a fragment using each fragment identifier. Without authorization, an adversary can only guess which fragments are used to create an object. Such a search of the system would not only require a great deal of time and computing power, but it would also be easy to detect.

4.5 Retrieving Objects from the System

As implied from the structure illustrated in the previous sections, the only piece of information necessary for immediate object reconstruction is the fragment identifier list. As of now, we are unsure how the actual fragment identifier list will be stored. We assume a subject will request an object, which will require some form of authentication. If the subject is authorized to access the object, the system will retrieve the fragment identifier list and issue a fragment reconstruction request for each fragment identifier. Since each shard is stored with a list of the fragment identifiers that use the shard, a traversal of the entire distributed list is required for each fragment identifier.

Depending on the chosen storage policies for an object, a fragment is reconstructed after all ((n,n)-scheme) or a subset ((m,n)-scheme) of the shards for the given fragment are found. The fragments are then used to reconstruct the object.

5 Open Problems and Future Work

The POTSHARDS project is still at a relatively early stage and thus many of its design elements are still in their formative stages. In fact, none of the aspects of the POTSHARDS are at a stage where their design can be considered finalized for even the initial implementation. There are still many questions that, while identified, have yet to be examined in greater detail. Some of the more pressing issues that we identified are listed below.

5.1 Data Structures

We expect POTSHARDS data structures to change as our design of the system as a whole matures. Currently, we are designing the system with doubly-linked lists in mind. An early sketch of the system used tree structures which may have improved performance but presented too much of a security risk.

Aside from the fundamental data structures, the contents of each node are subject to change as well. Some possibilities for changes to the structure of the nodes include fields to indicate status of the node. This might include fields used by garbage collection or fields which identify the nodes as being of a particular data type. The latter example could be useful if multiple types of data are stored in the list besides shards such as naming information or object metadata.

5.2 Consistency Checking

A critical aspect of very long-term storage is insuring that file consistency is maintained. Malicious users, degradation and faulty writes can all cause trouble for a system aimed at maintaining data for an extended period of time; such a system must provide a proactive solution to insuring that the integrity of its contents is protected. One method of ensuring this in POTSHARDS is through the use of active consistency checking.

One straightforward method of active consistency checking would be to check the integrity of the system contents at regular times. This would require securely recording consistency information, such as a hash value, for each shard. This method could be optimized by throttling these integrity operations to reduce overhead during times of high activity. Further optimization might involve smart checking that does not check each shard but rather checks enough shards so that the fragment could be regenerated.

A promising area of consistency checking could be the use of algebraic signatures such as those described by Litwin and Schwarz [6]. These structures could be used to optimize the consistency checking within the system compared to traditional hashing algorithms.

5.3 Archive Recovery

The distributed nature of POTSHARDS introduces the possibility of a failed storage device. Since the system is designed to provide storage for decades or longer, the failure of one storage devices or even an entire archive is inevitable. Further pressing the need for reliable disaster recovery is the doubly linked list structure used in POTSHARDS as shown in Figure 3. Since all the storage devices are connected, the loss of one device must not render the list irreparable. POTSHARDS must have a reliable way to recover from the simultaneous failures of multiple storage archives.

In addition to straightforward storage device failures, the POTSHARDS system should be able to deal with Byzantine failures where a device may be acting maliciously. While the projected implementation of POTSHARDS would consist of a controlled network of distributed devices, as opposed to a federated storage system such as FARSITE [1], the possibility still exists that a compromised storage device could be acting maliciously.

5.4 Naming

At the present time the naming of shards within the system has not been finalized. There are a number of possibilities that are being examined ranging from randomly generated names, names based on cryptographic hashes of the shards contents, or magic numbers generated by some deterministic process. Hash based naming has the advantage that the naming and consistency information is one and the same. If a malicious user has the ability to change shard data then there are two possible attack scenarios. In the first, a malicious user changes the data but not the name. In this case regenerating the name reveals the change. In the second scenario a malicious user changes the data and the name. In this case a search for a shard by name then the search simply returns a negative result.

The issue of naming extends beyond shards to other elements in the system. The methods and role of naming in dealing with fragments and objects is another area to be examined. The distributed nature of POTSHARDS also suggests that the naming of components is an important issue as well.

5.5 Storage Protocol

In the current discussion of POTSHARDS, the actual storage devices are referenced in rather general terms. As mentioned previously, the system is designed to be hardware agnostic so that it can accommodate future advances in computing technology. None the less, the capabilities and high level storage protocols must be defined so that an implementation can make adequate hardware decisions.

5.6 Migration

As POTSHARDS is designed for very long-term storage, migration is an important consideration. The distributed nature of the system suggests that archives will be coming on-line as well as leaving the system. A method of safely moving shards off of the archive that is scheduled to be removed from the system is therefore very important. Related to graceful removal of archives from the system is a method of dealing with catastrophic failure of a archive.

For archives that come on line after the initial start of the system, a method of normalizing population across all the archives in the system will be important. This may involve placing more shards on new archives, moving existing shards to new archives or a hybrid of both approaches.

Another area of migration that is being considered is the idea of data churning. Data churning would involve the automatic movement of shards within the system. This has the possible benefit of making targeted attacks more difficult. One possible difficulty is that any churning strategy would need to ensure that all of the shards needed to reconstruct an object are not inadvertently moved to the same archive. If the churning strategy is based on an even probability distribution, the chances of this occurring should be acceptably low. Even with a low probability of shard consolidation within a single archive, we would still prefer to not take any chances. We are considering policies to bound the number of single-object shards placed on an archive. For instance, some fraction f of the total number of shards for an object will place a firm upper bound on shard consolidation within a single archive.

5.7 Managing Storage Growth

If the growth of storage is not properly addressed, it is obvious that POTSHARDS may incur a great deal of storage overhead, which is the cost for long-term data protection and integrity. While the POTSHARDS system's distributed nature allows for easy installation of additional storage, efficient use of existing storage might be achieved through the use of garbage collection and shard reuse.

A possible problem with garbage collection would be designing a strategy that does not violate any of the POTSHARDS design principles. Specifically, any garbage collection strategy would have to be secure against a malicious user. For example, if a straightforward strategy of reference counting was used, the data structures must be secure from a malicious user that attempts to artificially reduce the reference count in order to trigger a clean-up. If certain shards are used for the regeneration of more than one file, as previously suggested, this sort of attack could be very damaging.

Storage efficiency might also be accomplished by reusing preexisting shards to limit the amount of storage overhead needed. In this strategy, instead of generating all the shards for a given fragment, it might be possible to use a mixture of pre-existing and randomly generated shards. The implications of this strategy would be that some shards would be used in the regeneration of more than one file. This could have the desired effect of reducing the total amount of storage overhead imposed by the secret splitting and redundancy encoding.

6 Conclusions

Current systems do not address the needs of a system that must store files securely for a very long period of time. When storing files for spans of time measured in decades, many of the common conventions used in storage introduce unacceptable weaknesses. Keyed encryption introduces a single point of failure and is only computationally bound; history has shown that this approach often fails over time. Similarly, user accounts and file ownership may be shorter lived than the files when dealing with data that must survive longer than the users that created it.

The POTSHARDS project aims to provide file security for very long-term storage through the use of secret sharing. Objects that are to be stored in the system are split into fragments in a security layer and shards in the redundancy layer. These shards are stored within a distributed storage environment in a linked list structure. Since computing technology has the potential to drastically change over several decades the POTSHARDS system is designed with a modular structure that allows for components to be upgraded over time.

Moving forward, the primary focus for the POTSHARDS system is to continue to revise and formalize the design, with the goal of producing a working prototype. A testable prototype should further force the revision of the designs described in this paper and provide some evidence for the scalability and feasibility of the system.

Acknowledgments

The authors would like to thank Owen Hofmann, Carl Lischeske, Kristal Pollack, Deepavali Bhagwat, Lawrence

You, and Darrell Long for their help in early discussions on the POTSHARDS design. Other members of the Storage Systems Research Center were also helpful. We would also like to thank the industrial sponsors of the SSRC, including Hewlett Packard Laboratories, Hitachi Global Storage Technologies, IBM Research, Intel, Microsoft Research, Network Appliance, Rocksoft, Veritas, and Yahoo for their generous support.

References

[1] A. Adya, W. J. Bolosky, M. Castro, R. Chaiken, G. Cermak, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. Wattenhofer. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*, Boston, MA, Dec. 2002. USENIX.

[2] Y. Chen, J. Edler, A. Goldberg, A. Gottlieb, S. Sobti, and P. Yianilos. A prototype implementation of archival intermemory. In *Proceedings of the Fourth ACM International Conference on Digital Libraries*, 1999.

[3] I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong. Freenet: A distributed anonymous information storage and retrieval system. *Lecture Notes in Computer Science*, 2009:46+, 2001.

[4] A. Haeberlen, A. Mislove, and P. Druschel. Glacier: Highly durable, decentralized storage despite massive correlated failures. In *2nd Symposium on Networked Systems Design and Implementation (NSDI '05)*, Boston, MA, USA, May 2005.

[5] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 190–201, Nov 2000.

[6] W. Litwin and T. Schwarz. Algebraic signatures for scalable distributed data structures. Technical Report CERIA Technical Report, Université Paris 9 Dauphine, Sept. 2002.

[7] P. Maniatis, M. Roussopoulos, T. Giuli, D. S. H. Rosenthal, M. Baker, and Y. Muliandi. Preserving peer replicas by rate-limited sampled voting. In *Proceedings of the Symposium on Operating Systems Principles*, Bolton Landing, NY, Oct 2003. ACM.

[8] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 2001.

[9] 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*, Chateau Lake Louise, Banff, Alberta, Canada, Oct 2001.

[10] D. R. Stinson. *Cryptography Theory and Practice*. Chapman & Hall/CRC, Boca Raton, FL, second edition, 2002.

[11] A. Subbiah and D. M. Blough. An approach for fault tolerant and secure data storage in collaborative work environements. In *Proceedings of the 2005 ACM Workshop on Storage Security and Survivabiilty*, pages 84–93, Fairfax, VA, Nov. 2005.

[12] T. M. Wong, C. Wang, and J. M. Wing. Verifiable secret redistribution for threshold sharing schemes. Technical Report CMU-CS-02-114-R, Carnegie Mellon University, Oct. 2002.

[13] J. J. Wylie, M. W. Bigrigg, J. D. Strunk, G. R. Ganger, H. Kiliççöte, and P. K. Khosla. Survivable storage systems. *IEEE Computer*, pages 61–68, Aug. 2000.

Disaster Recovery Codes: Increasing Reliability with Large-Stripe Erasure Correcting Codes

Kevin M. Greenan
Computer Science Dept.
Univ. of California, Santa Cruz
Santa Cruz, CA, USA
kmgreen@cs.ucsc.edu

Ethan L. Miller
Computer Science Dept.
Univ. of California, Santa Cruz
Santa Cruz, CA, USA
elm@cs.ucsc.edu

Thomas J. E. Schwarz
Computer Engineering Dept.
Santa Clara University
Santa Clara, CA, USA
tjschwarz@scu.edu

Darrell D. E. Long
Computer Science Dept.
Univ. of California, Santa Cruz
Santa Cruz, CA, USA
darrell@cs.ucsc.edu

ABSTRACT

Large-scale storage systems need to provide the right amount of redundancy in their storage scheme to protect client data. In particular, many high-performance systems require data protection that imposes minimal impact on performance; thus, such systems use mirroring to guard against data loss. Unfortunately, as the number of copies increases, mirroring becomes costly and contributes relatively little to the overall system reliability. Compared to mirroring, parity-based schemes are space-efficient, but incur greater update and degraded-mode read costs. An ideal data protection scheme should perform similarly to mirroring, while providing the space efficiency of a parity-based erasure code.

Our goal is to increase the reliability of systems that currently mirror data for protection without impacting performance or space overhead. To this end, we propose the use of large parity codes across two-way mirrored reliability groups. The secondary reliability groups are defined across an arbitrarily large set of mirrored groups, necessitating a small amount of non-volatile RAM for parity. Since each parity element is stored in non-volatile RAM, our scheme drastically increases the mean time to data loss without impacting overall system performance.

Categories and Subject Descriptors

D.4.2 [**Software**]: Storage Management—*Secondary storage*; D.4.5 [**Software**]: Operating Systems—*reliability*

General Terms

Reliability

Keywords

erasure coding, storage reliability

StorageSS'07, October 29, 2007, Alexandria, Virginia, USA.

1. INTRODUCTION

Storage systems use redundancy in order to store data reliably. High performance storage systems, such as Ceph [12] and GPFS [9] use mirroring to store client data over possibly thousands of disks. While mirroring is well suited for high-performance workloads, full data replication results in a great deal of overhead and contributes little to the overall probability of not losing data during the economic lifespan of the file system [7]. For instance, 2-way mirroring could yield two or three nines of data survivability, but pushing this number to four or five nines requires much higher redundancy. Systems such as Oceanstore [6] and FAB [8] have the ability to provide higher redundancy and pay less in terms of space consumption using parity-based erasure correcting codes for data protection. These systems trade space efficiency for the read-write-modify update and degraded-mode read penalties incurred when using parity-based protection schemes.

One quality most mirrored and erasure-encoded systems share is a single level of replication; trading off either space efficiency or performance in the process. Our aim is to maintain the performance advantages of a mirrored system while providing very high reliability. Instead of providing additional redundancy using a single parity-based encoding, we propose a two-level scheme that adds a second layer of redundancy; resulting in data protection that can handle correlated and massive failures without incurring unacceptable space and performance overhead. Our scheme creates primary redundancy groups using two-way mirroring, which enables fast recovery in the case of single disk failure. A second, erasure-encoded redundancy group is computed across a large set of primary copies of mirrored data. The second layer of redundancy augments a traditional mirroring scheme with additional protection; making recovery attainable in the case of massive or correlated failures.

In an effort to minimize the parity update costs across the second layer of reliability groups, we store all parity data in non-volatile RAM (NVRAM), which is distributed throughout the storage devices. By creating very large stripes in the second layer of redundancy, our solution only requires a small fraction of NVRAM. For example, suppose a single data strip encompasses 1 GB and a single parity strip is computed from 1000 data strips. Such a configuration would result in 1 MB of parity for every gigabyte of data, or $\frac{1}{1000}$ parity overhead. We chose this number because currently, the price of NVRAM, such as compact flash, is less than 1000 times that of disk measured in $/GB. Since NVRAM can be updated at a much higher rate, our solution is cheaper and faster than adding

disk space to store additional parity data for a much smaller reliability stripe. In addition, we find that data reliability increases substantially even when using single parity across mirrored groups.

The main contribution of this paper is the addition and analysis of large parity groups across mirrored reliability groups. We believe that high-performance storage systems that currently rely on mirroring for reliability will benefit from the extra redundancy, without compromising system performance. As our preliminary results show, our scheme results in much higher reliability than 2-way mirroring. Since this work concentrates on reliability, we postpone a detailed performance analysis.

2. OVERVIEW

Our scheme is not specific to any single system; however, for clarity, we present our analysis in a system similar to Ceph [12]. The system stores client data in fixed-sized blocks, called *buckets*. The buckets themselves are objects assigned to object-based storage devices (OSD), which are assumed to contain at least a disk and some NVRAM. All client data is stored in *data buckets* on disk, while parity data is placed in *parity buckets* on NVRAM.

Our storage system stores client data in buckets of size approximately 1 GB, so each device stores about $1,000$ data buckets. The storage system consists of about $10,000$ devices and hence stores about 10^7 buckets, for a total of 10 PB of raw storage. In the future, we expect the size of storage installations to increase and the capacity of disk drives to increase, though perhaps not at the historical rate of 60% to 80%. Our purpose in giving these numbers is to give a more concrete picture of the system, but we expect the utility of our approach to increase with larger systems.

For our analysis, we assume the system uses mirroring to protect buckets against device failure. Each bucket is stored on two devices selected in a pseudo-random manner using algorithms similar to RUSH [5]. A bucket is considered lost if the two devices are lost or sector failures corrupt both copies of the bucket. Buckets are addressed by a simple number and might migrate during the lifetime of the system as new devices are added to the system and old ones are removed because of failure or obsolescence.

Assume that a disk fails every 5 years. Because of the distributed nature of our storage devices, the system is reconfigured and protected much faster than the several hours it takes to read a disk completely. In such a system 30 minutes is a very conservative estimate for reconfiguration time, even including time to detect the failure. This means that each bucket is vulnerable to single OSD failure for 30 minutes / 5 years = 10^{-5} of its life. Thus, a bucket is lost at a probability of 10^{-10} times the lifespan of its data. Unfortunately, there are many buckets, leading to a much greater likelihood of data loss.

We increase the already excellent survival chances of mirrored data by maintaining additional parity data stored in a NVRAM calculated over a reliability stripe of thousands of disks. In the rare case of a bucket annihilation on disk (e.g. loss of both replicas), we shut down the system and use the NVRAM based parity together with the other buckets in the same reliability group to reconstruct the missing buckets. The calculation is tedious (involving reading thousands of disks) but highly likely to succeed.

We propose a technique that calculates NVRAM parity data over a large set of data buckets. Each bucket in the system is assigned to a single reliability stripe with R buckets total to which we add 1, 2 or 3 NVRAM parity buckets. In each of the three scenarios, we calculate the parity using a maximum distance separable (MDS) code. A (n,k) MDS code that computes $n-k$ parity buckets over k data buckets can sustain the failure of any $n-k$ buckets. We assume that an XOR-parity scheme is used to calculate single parity, while

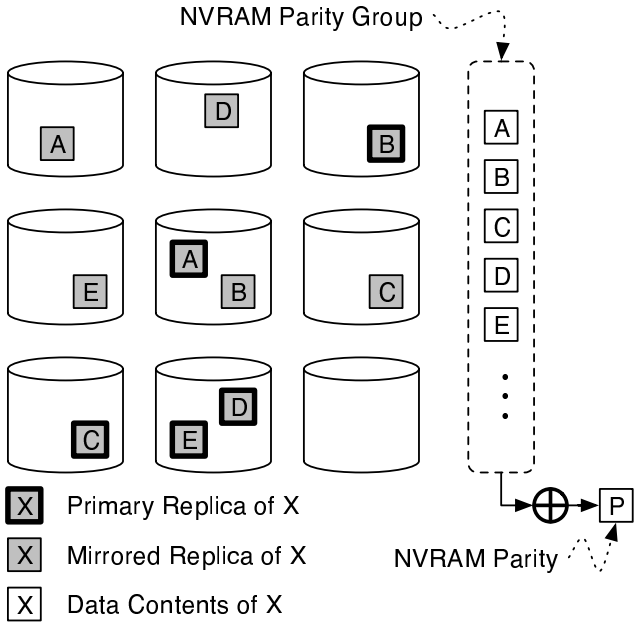


Figure 1: *The layout of our disaster recovery encoding scheme. Primary and mirrored replicas are distributed such that no two replicas of a single data bucket are stored on the same disk. The data contents of each object contribute to the parity of a single parity group, achieving another level of redundancy. We assume that at least one physical copy of each data element in a parity group exists on a distinct disk.*

	Disk Overhead	NVRAM Overhead
Mirror	$\frac{N}{2}$ *TB*	0
Mirror+P	$\frac{N}{2}$ *TB*	$\left(P \cdot \frac{1}{R} \cdot \frac{N}{2} \cdot B\right)$ *GB*

Table 1: *Storage overhead for each reliability scheme given N 1 TB OSDs, B 1 GB buckets per OSD, secondary reliability groups with R data buckets and P parity elements per group.*

Reed-Solomon codes are used for double and triple parity.

Figure 1 shows how both mirrored replicas and parity groups are defined in our system. Each *data bucket* is stored in two physical locations, as the *primary replica* and *mirrored replica*. Each replica must be stored on two distinct OSDs. The contents of each data bucket also contribute to a single *parity group*, with the constraint that each data element in a parity group must exist on a distinct OSD. This constraint is easily maintained if we consider the physical location of the primary and mirrored replica of a data bucket. The data elements of a parity group are used to compute a parity element, which is stored in distributed NVRAM banks across the OSDs themselves.

Each parity group protects R data buckets from data loss. If a data bucket's corresponding primary and mirrored replicas are lost due to disk failure, the remaining data buckets and parity buckets in the parity group are used to reconstruct the lost bucket. Once the lost data bucket is reconstructed, the primary and mirrored replica are redistributed. Reconstruction during whole OSD failure, though more complicated and time consuming, works in a similar fashion. As long as each parity group uses an (n,k) MDS code, our scheme can tolerate any $n-k$ lost data buckets.

Figure 2 shows the overhead associated with 2-way mirroring and parity groups over the mirrored buckets, assuming $1,000$ (1 GB) buckets per OSD, $10,000$ OSDs and $3,000$ buckets per parity group.

If the parity groups utilize an encoding that adds a single parity element, then the total storage overhead is roughly 5,000 TB of disk redundancy and 1.63 TB of NVRAM redundancy. Since the NVRAM storage is distributed throughout the system, placing 1 GB of NVRAM on each OSD should be sufficient. Also note that the storage overhead in our example (excluding 2-way mirroring overhead) is 1.63 TB for every 5,000 TB of data, or 0.03%.

An issue arises when using a Reed-Solomon code to calculate the additional parity. In most fast software implementations of Reed-Solomon, a Galois field with 256 elements is used to calculate parity. Unfortunately, this field will only support 257 buckets per reliability group. Following byte boundaries, the next largest field, $GF(2^{16})$, contains $65,536$ elements, which is sufficient for our purposes. We have developed an efficient implementation of $GF(2^{16})$ and larger fields [4].

3. RELIABILITY ANALYSIS

Intuitively, the addition of NVRAM parity would be expected to lead to much higher data reliability than standalone mirroring. We quantify and compare the reliability of the mirrored and NVRAM parity schemes using both probabilistic and stochastic analysis. The analysis shows that augmenting a mirrored system with NVRAM parity results in a substantial increase in reliability.

The target storage system in our analysis contains a total of N OSDs, though strictly speaking, N varies due to failure and replacement. We assume that B buckets are stored on each device, resulting in $N \cdot B$ buckets of storage in the entire system. Each data bucket is mirrored; thus, the system stores $\frac{N \cdot B}{2}$ buckets of data. We distinguish between physical buckets and data buckets, where the latter is stored in two physical buckets. In the following, we simply say bucket for a data bucket.

3.1 Probabilistic Analysis

By design, we never store the two physical buckets of a (data) bucket on the same OSD. Assume now that k OSDs have failed in a Ceph-like system that relies on 2-way mirroring for data reliability. With probability $p(k) = \frac{\binom{k}{2}}{\binom{N}{2}}$, a given bucket is located on these k OSDs and hence lost. Then, with probability $q(k) = 1 - p(k)$, the failures have not led to data loss. Given k failures and no NVRAM parity, the system survival probability is $Q_{0\mathrm{P}}(k) = q(k)^{\frac{N \cdot B}{2}}$; the failure probability is $1 - Q_{0\mathrm{P}}(k)$.

Suppose a 2-way mirrored scheme is augmented with the NVRAM parity scheme by adding n parity buckets to each group containing R data buckets. This allows the whole group to survive up to n bucket failures—situations where *both* mirrors of a bucket are lost. Given k failures, the probability that $l \leq k$ of the n NVRAM parity buckets in a single group are located on a failed OSD is $\frac{\binom{n}{l}\binom{N-n}{k-l}}{\binom{N}{k}}$. Since there are $n-l$ available NVRAM parity buckets in the group, the chance of data survival is the probability that at most $n-l$ of the R data buckets are unavailable; given by the cumulative binomial distribution. The survival probability of a single parity group is calculated as

$$Q(n,R,k) = \sum_{l=0}^{n} \left(\frac{\binom{n}{l} \cdot \binom{N-n}{k-l}}{\binom{N}{k}} \sum_{v=0}^{n-l} \binom{R}{v} p(k)^v q(k)^{R-v} \right).$$

Assuming k failures, the survival probability of a system with n NVRAM parity buckets per R bucket group is $Q_{\mathrm{nP}}(k) = Q(n,R,k)^{\frac{N \cdot B}{2 \cdot R}}$, since there are $\frac{N \cdot B}{2 \cdot R}$ parity groups in the system.

Figure 2 shows the effect of computing NVRAM parity across data buckets. These results show that using of parity groups with just a single parity element has a dramatic effect on reducing the probability of data loss given multiple OSD failures. We find that calculating a single NVRAM parity across every $1,000$ buckets gives even odds to survive roughly 35 OSD failures, while mirroring alone has the same odds to survive about 4 OSD failures.

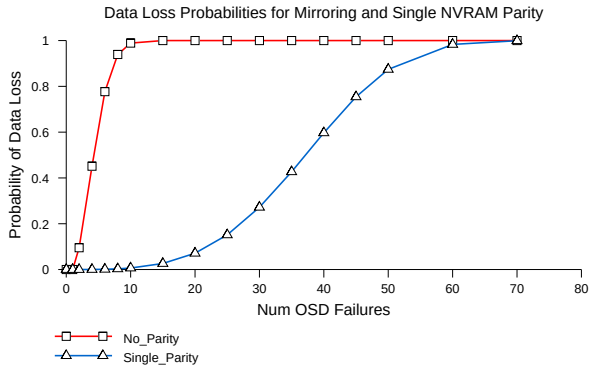


Figure 2: *Data loss probabilities for an increasing number of OSD failures, across mirroring and the single parity reliability schemes. Both schemes use* 2*-way mirroring as the primary redundancy scheme. The single parity scheme computes parity across groups of* $1,000$ *data buckets.*

Figure 3 gives the data loss probabilities for double and triple parity NVRAM layouts. First, notice that adding more NVRAM protection per group greatly increases the number of tolerated failures. This figure also shows the effect of group size on the probability of data loss, which is lower for smaller R. This observation is rather intuitive because the probability of two or more OSD failures occurring in the same group decreases with group size. The reliability benefit associated with group size introduces a cost/reliability tradeoff; decreasing the size of the NVRAM parity groups will increase NVRAM utilization.

3.2 Calculating Mean Time to Data Loss

The probability calculations given above demonstrate that computing large NVRAM parity groups across mirrored data results in very high data survival probabilities. We now use the probabilistic analysis to derive and compare the expected mean time to data loss.

Figure 4 shows our generic Markov model with which we calculate Mean Time to Data Loss (MTTDL). The system is in a state S_k, where k denotes the number of OSDs currently unavailable. The initial state is S_0. For accuracy, the model presumes that we replace failed OSDs with new OSDs, whereas in reality, the OSDs are not replaced; instead, the system is replenished from time to time with new batches of OSDs. While these OSDs would probably also have a higher capacity and carry more buckets, modeling these details is quite difficult and would not yield additional insights.

Since the baseline redundancy scheme is 2-way mirroring, the failure of two or more OSDs may result in data loss. If $Q_{\mathrm{nP}}(k)$ denotes the chances of survival when k disks have failed, then we calculate the probability that the additional failure caused data loss as the conditional probability

$$P_k = \Pr(DL_k | NDL_{k-1}) = 1 - \frac{Q_{\mathrm{nP}}(k)}{Q_{\mathrm{nP}}(k-1)},$$

where DL_k is the event of data loss after k OSD failures and NDL_{k-1} is the event of no data loss after $k-1$ OSD failures. Since

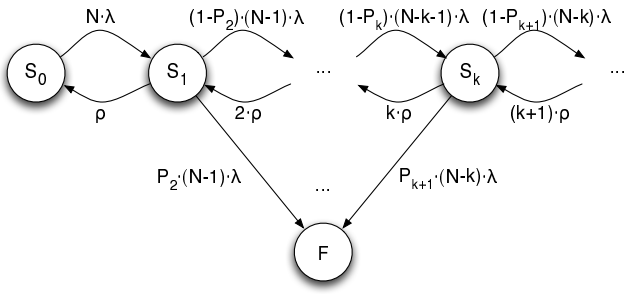


Figure 4: *Our Markov model resembles a standard Birth-Death model with an additional absorbing failure state (F). Transitions between the non-absorbing states are OSD failures that do not result in data loss and OSD repairs. A transition to the failure state is a data loss event.*

all buckets are mirrored across distinct OSDs, we set $P_0 = 0$ and $P_1 = 0$.

Our Markov model is closely related to a standard birth and death model, but with additional failure transitions. The model assumes that disk drives fail at a constant rate λ and "repair" replicates all buckets from the failed OSD elsewhere and that the rate of repair is ρ.

As shown in Figure 4, an OSD failure corresponds to a birth and a repair to a death. An OSD failure without data loss is the transition from state S_k going to S_{k+1} with rate $(1-P_{k+1})(N-k)\lambda$. If an OSD failure leads to data loss, then the model transitions to the failure state F, taken with frequency $P_{k+1}(N-k)\lambda$. The repair transition from state S_k to S_{k-1} is taken with rate $k\rho$, where the multiplier, k, accounts for concurrent OSD recovery. We cannot expect numerically reliable answers for large numbers of state and we cut off our Markov model at a state S_S where $p_S \approx 1$ and then set $P_S = 1$.

Figure 5 displays the system MTTDL for a mirrored system without NVRAM parity and a system with single NVRAM parity. We measure system MTTDL as a function of OSD mean time to failure (MTTF) and repair rate. The analysis assumes repair rates of $\frac{1}{4}$, $\frac{1}{2}$ and 1 hour. Due to the seemingly random allocation of buckets to disks, these repair times are sufficient to "copy" all of the failed buckets on a OSD to other OSDs in the system. The MTTDL numbers indicate that the use of single NVRAM parity per group results in a much more reliable system than standalone mirroring. We also find that repair rate has a non-trivial effect on system reliability. Even though the use of a single NVRAM parity bucket per group results in a $4,000$ fold increase in MTTDL, applying these results to the provisioning of an actual storage system needs to be done with caution because the basic assumption-the Markov property- makes MTTDL values (millions of years) very hard to comprehend.

Unfortunately, we could not numerically handle the large number of required states for double and triple parity with any confidence. In light of this, we still believe the analysis as a whole justifies the use of NVRAM parity in a mirrored system.

4. RELATED WORK

Most large-scale distributed systems rely on redundancy for data availability and reliability. High performance systems such as GPFS [9] (which also supports RAID 5) and Ceph [12] utilize mirroring for redundancy. FAB [8] has the ability to use either erasure codes or mirroring for redundancy, while FARSITE [1] uses mirroring instead of erasure codes. All of these systems use a single level of redundancy, which may fall prey to correlated failures or failure during rebuild. We believe that our techniques could be incorporated into these systems resulting in much higher reliability.

Other systems use multiple levels of redundancy for greater fault-tolerance and availability. POTSHARDS [11] performs a single availability-centric split using threshold cryptography before storing data across archives using secure distributed RAID. Oceanstore [6] replicates so-called active data and disperses copies of this data into deep archival data using erasure codes. Our scheme independently generates two-levels of redundancy in a way that has a minimal impact on average-case performance, while dramatically improving data reliability.

A variety of studies analyze ways to further improve storage system reliability. Xin *et al.* [13] propose mechanisms for mirrored and mirrored RAID 5 systems that increase system recovery rate and recover from nonrecoverable read errors. While Xin *et al.* analyzed mirrored and mirrored-RAID 5 configurations, we propose and analyze large parity codes over mirrored groups.

Schwarz *et al.* [10] propose and analyze a mechanism, called disk scrubbing, used to actively check data integrity in large storage systems. Baker *et al.* [3] and Bairavasundaram *et al.* [2] further validate the importance of active data scrubbing (or auditing) for detecting and recovering from latent faults as quickly as possible. Although we did not consider active checking in our analysis, we expect disk scrubbing to be integral in future analysis—especially when considering latent errors.

5. CONCLUSION AND FUTURE WORK

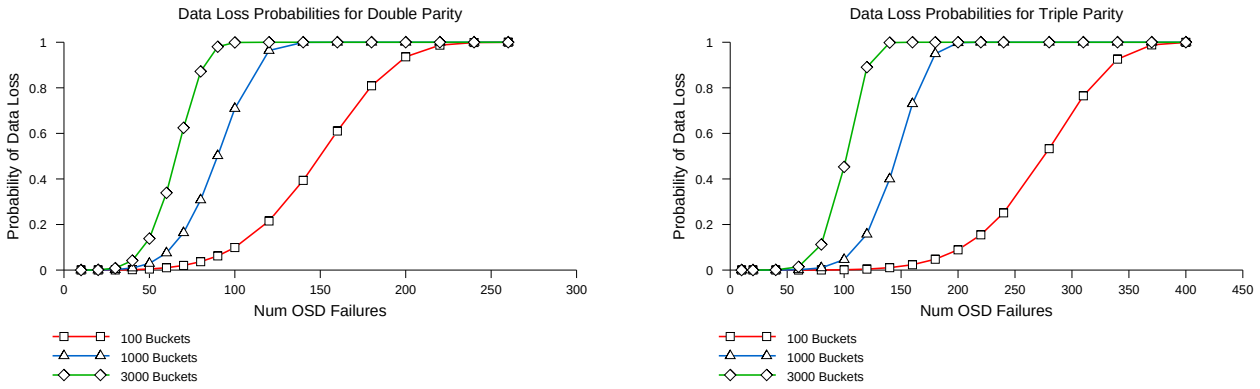
There exists a great deal of work to be completed in our disaster recovery encoding scheme. Our preliminary analysis does not include a performance evaluation. In the future, we plan to compare the update, degraded-mode read and rebuild overhead of our two-level scheme to a one-level mirroring scheme. In addition, we plan to consider the effect of correlated failures, latent sector faults and "bad" batches of disks. Finally, this scheme may be extended to a multi-level hierarchy of arbitrary erasure encoding schemes. We plan to investigate the utility and reliability of this and other hierarchical encoding schemes.

We have presented a method that drastically increases the reliability of mirrored systems, while imposing relatively small space and expected performance overhead. Our scheme generates an extra level of redundancy, using large parity-based erasure codes, computed across mirrored groups. All parity is stored in distributed NVRAM banks, resulting in a faster implementation compared to storing additional redundancy on disk.

Our preliminary analysis shows that even the use of a parity group with a single parity element contributes greatly to the overall reliability of the system. Our probabilistic analysis shows that the NVRAM parity scheme increases resilience against multiple OSD failures that might occur, for example, as the result of a bad batch of disks. System MTTDL (defined as loss of a single data bucket), also increases, about 4000-fold by introducing a single RAM parity per 1000 buckets. As the standalone mirroring results show, recourse to the NVRAM parity is rare, but may be necessary since such events are quite traumatic.

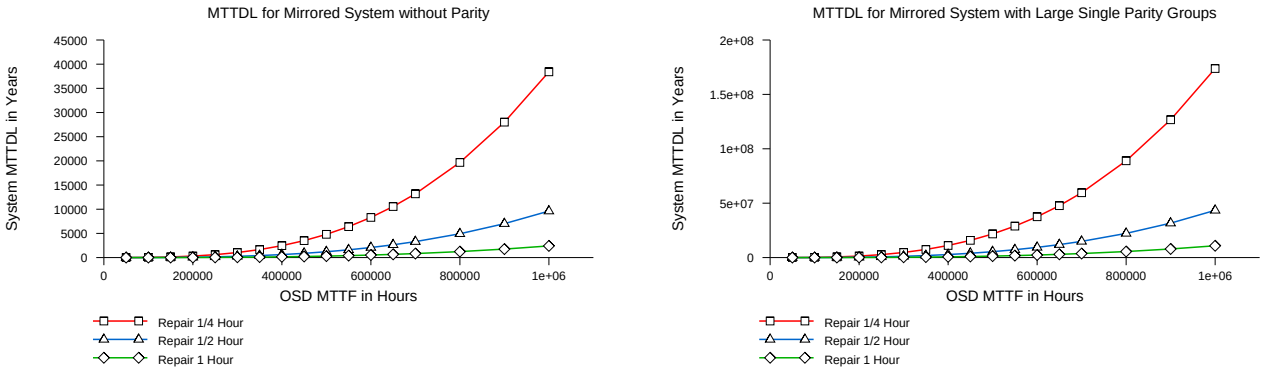
Acknowledgments

We would like to thank our colleagues in the Storage Systems Research Center (SSRC) who provided valuable feedback on the ideas in this paper. This research was supported by the Petascale Data Storage Institute, UCSC/LANL Institute for Scalable Scientific Data Management and by SSRC sponsors including Los Alamos Na-



(a) (b)

Figure 3: *Data loss probabilities for an increasing number of OSD failures and three parity group sizes, across double (a) and triple (b) parity groups. Both schemes use* 2*-way mirroring as the primary redundancy scheme.*



(a) (b)

Figure 5: *Mean Time to Data Loss (years) with* 2*-way mirroring (a) and with one NVRAM parity per 3000 bucket group (b) for a system with 10,000 disks, 1000 blocks per disk, and a repair times of 1/4, 1/2 and 1 hour. We find that repair repair time has a non-trivial effect on reliability. MTTF is in hours.*

tional Lab, Livermore National Lab, Sandia National Lab, Digisense, Hewlett-Packard Laboratories, IBM Research, Intel, LSI Logic, Microsoft Research, Network Appliance, Seagate, Symantec, and Yahoo.

6. REFERENCES

[1] A. Adya, W. J. Bolosky, M. Castro, R. Chaiken, G. Cermak, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. Wattenhofer. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*, Boston, MA, Dec. 2002. USENIX.

[2] L. N. Bairavasundaram, G. R. Goodson, S. Pasupathy, and J. Schindler. An analysis of latent sector errors in disk drives. In *Proceedings of the 2007 SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, June 2007.

[3] M. Baker, M. Shah, D. S. H. Rosenthal, M. Roussopoulos, P. Maniatis, T. Giuli, and P. Bungale. A fresh look at the reliability of long-term digital storage. In *Proceedings of EuroSys 2006*, pages 221–234, Apr. 2006.

[4] K. M. Greenan. Efficient Galois field and data encoding library. http://www.soe.ucsc.edu/ kmgreen/html/sw.html, May 2007.

[5] 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 & Distributed Processing Symposium (IPDPS 2004)*, Santa Fe, NM, Apr. 2004. IEEE.

[6] J. Kubiatowicz, D. Bindel, Y. Chen, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. OceanStore: An architecture for global-scale persistent storage. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Cambridge, MA, Nov. 2000. ACM.

[7] S. Nath, H. Yu, P. B. Gibbons, and S. Seshan. Subtleties in tolerating correlated failures in wide-area storage systems. In *Proceedings of the 3rd Symposium on Networked Systems Design and Implementation (NSDI)*, 2006.

[8] Y. Saito, S. Frølund, A. Veitch, A. Merchant, and S. Spence. FAB: Building distributed enterprise disk arrays from commodity components. In *Proceedings of the 11th*

International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pages 48–58, 2004.

[9] F. Schmuck and R. Haskin. GPFS: A shared-disk file system for large computing clusters. In *Proceedings of the 2002 Conference on File and Storage Technologies (FAST)*, pages 231–244. USENIX, Jan. 2002.

[10] T. J. E. Schwarz, Q. Xin, E. L. Miller, D. D. E. Long, A. Hospodor, and S. Ng. Disk scrubbing in large archival storage systems. In *Proceedings of the 12th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS '04)*, pages 409–418. IEEE, Oct. 2004.

[11] M. W. Storer, K. M. Greenan, E. L. Miller, and K. Voruganti. POTSHARDS: secure long-term storage without encryption. In *Proceedings of the 2007 USENIX Annual Technical Conference*, pages 143–156, June 2007.

[12] 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 Symposium on Operating Systems Design and Implementation (OSDI)*, Seattle, WA, Nov. 2006. USENIX.

[13] Q. Xin, E. L. Miller, T. J. Schwarz, D. D. E. Long, S. A. Brandt, and W. Litwin. Reliability mechanisms for very large storage systems. In *Proceedings of the 20th IEEE / 11th NASA Goddard Conference on Mass Storage Systems and Technologies*, pages 146–156, Apr. 2003.

Power Consumption in Enterprise-Scale Backup Storage Systems

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

Abstract

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

1 Introduction

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

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

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

2 Related Work

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

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

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

3 Methodology

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

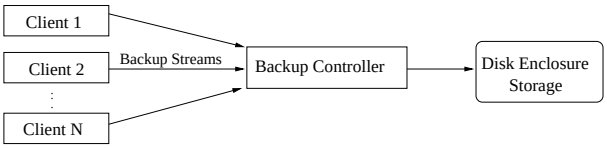


Figure 1: Backup system architecture

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

Table 1: Controller hardware summary

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

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

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

Table 2: Enclosure hardware summary

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

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

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

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

Table 3: Backup workloads used

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

4 Discussion

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

4.1 Controller Measurements

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

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

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

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

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

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

Table 4: Idle power consumptions for storage controllers

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

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

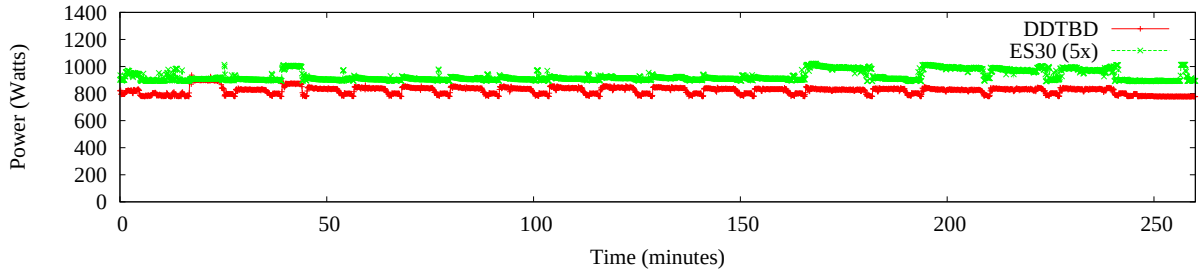
Controller under load. We measured the power consumption of each controller while running the aforementioned workloads. Each controller ran the DDFS deduplicating file system [35] and all required software services. Services such as replication were disabled. The power consumed under load approximates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

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

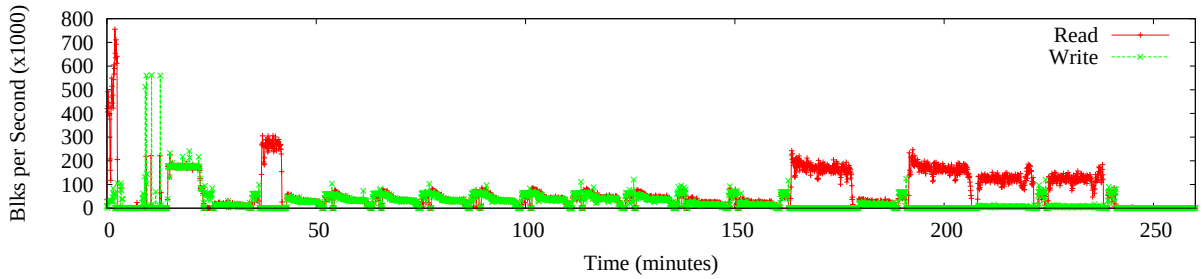
	DD880	DD670	DD860	DDTBD
WL-A	44%	24%	58%	20%
WL-B	58%	29%	61%	36%
WL-C	56%	28%	57%	23%

Table 5: Power increase ratios from idle to loaded system

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



(a) Power consumption



(b) I/O statistics

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

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

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

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

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

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

4.2 Enclosure Measurements

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

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

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

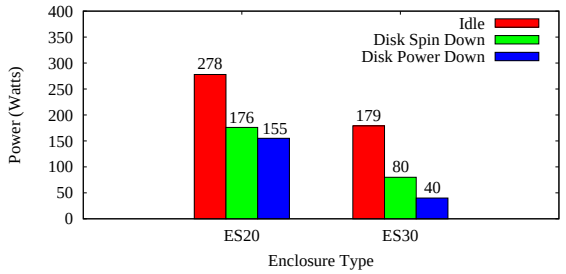


Figure 3: Disk power down vs. spin down. ES20 and ES30 are specified as in Table 2.

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

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

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

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

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

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

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

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

Table 6: Max power for enclosures ES20 and ES30

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

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

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

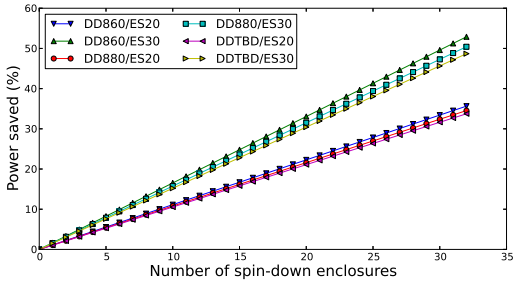
■ **Observation 8:** *Disk enclosures may consume more power than the drives they house. As a result, effective power management of the storage subsystem may require more than just disk-based power-management.*

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

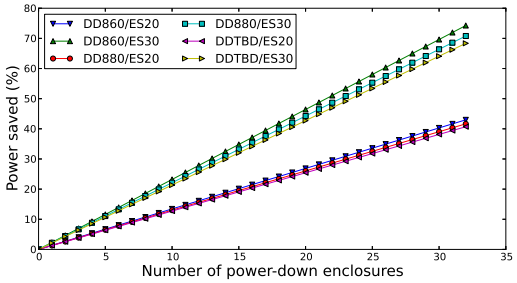
4.3 System-Level Measurements

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

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



(a) Disk Spin Down vs. Power Savings Percentage



(b) Disk Power Down vs. Power Savings Percentage

Figure 4: Total system power savings using disk power management

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

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

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

5 Conclusions

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

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

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

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

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

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

References

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[31] G. Wallace, F. Douglis, H. Qian, P. Shilane, S. Smaldone, M. Chamness, and W. Hsu. Characteristics of backup workloads in production systems. In *Proceedings of the Tenth USENIX Conference on File and Storage Technologies (FAST '12)*, San Jose, CA, February 2012. USENIX Association.

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

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

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

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

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

Power Consumption in Enterprise-Scale Backup Storage Systems

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

Abstract

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

1 Introduction

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

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

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

2 Related Work

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

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

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

3 Methodology

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

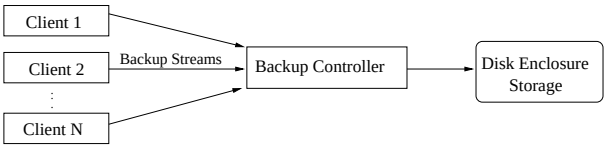


Figure 1: Backup system architecture

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

Table 1: Controller hardware summary

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

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

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

Table 2: Enclosure hardware summary

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

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

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

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

Table 3: Backup workloads used

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

4 Discussion

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

4.1 Controller Measurements

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

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

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

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

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

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

Table 4: Idle power consumptions for storage controllers

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

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

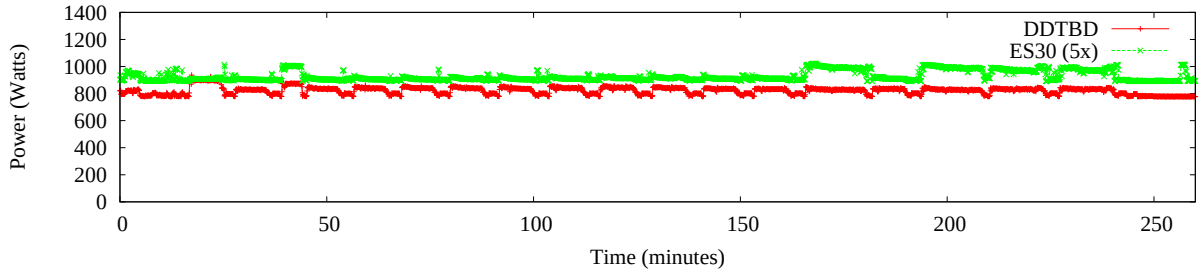
Controller under load. We measured the power consumption of each controller while running the aforementioned workloads. Each controller ran the DDFS deduplicating file system [35] and all required software services. Services such as replication were disabled. The power consumed under load approximates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

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

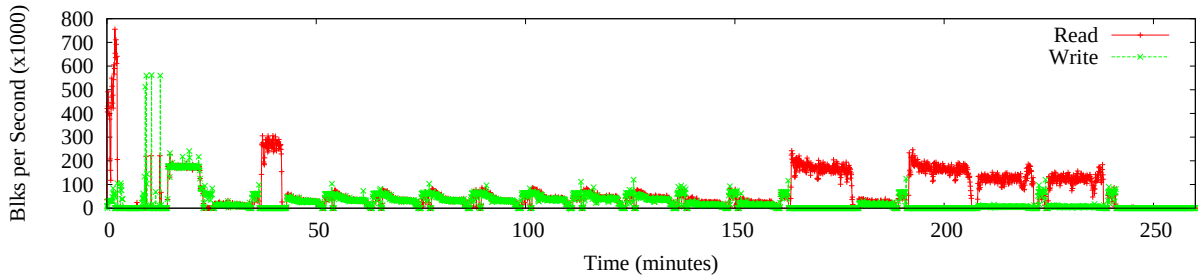
	DD880	DD670	DD860	DDTBD
WL-A	44%	24%	58%	20%
WL-B	58%	29%	61%	36%
WL-C	56%	28%	57%	23%

Table 5: Power increase ratios from idle to loaded system

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



(a) Power consumption



(b) I/O statistics

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

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

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

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

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

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

4.2 Enclosure Measurements

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

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

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

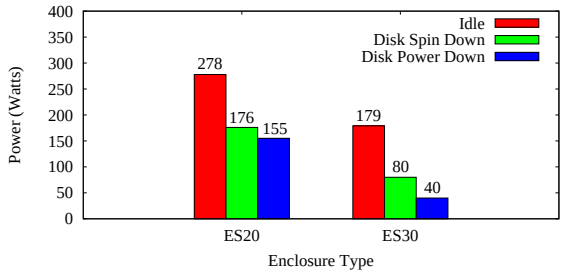


Figure 3: Disk power down vs. spin down. ES20 and ES30 are specified as in Table 2.

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

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

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

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

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

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

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

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

Table 6: Max power for enclosures ES20 and ES30

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

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

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

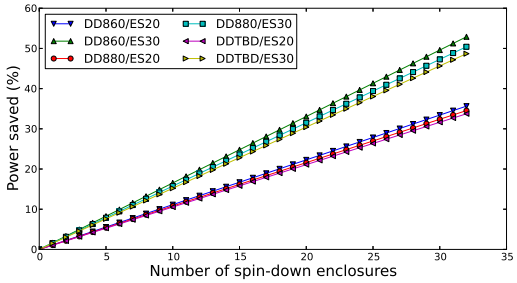
■ **Observation 8:** *Disk enclosures may consume more power than the drives they house. As a result, effective power management of the storage subsystem may require more than just disk-based power-management.*

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

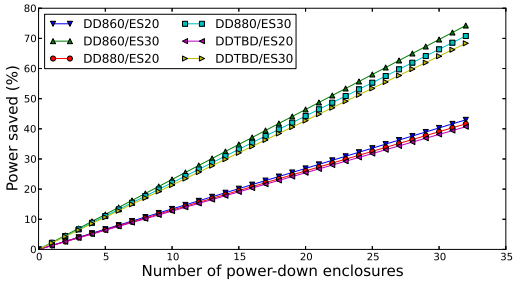
4.3 System-Level Measurements

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

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



(a) Disk Spin Down vs. Power Savings Percentage



(b) Disk Power Down vs. Power Savings Percentage

Figure 4: Total system power savings using disk power management

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

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

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

5 Conclusions

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

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

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

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

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

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

References

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[31] G. Wallace, F. Douglis, H. Qian, P. Shilane, S. Smaldone, M. Chamness, and W. Hsu. Characteristics of backup workloads in production systems. In *Proceedings of the Tenth USENIX Conference on File and Storage Technologies (FAST '12)*, San Jose, CA, February 2012. USENIX Association.

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

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

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

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

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

Screaming Fast Galois Field Arithmetic Using Intel SIMD Instructions

James S. Plank
EECS Department
University of Tennessee
plank@cs.utk.edu

Kevin M. Greenan
EMC Backup Recovery
Systems Division
Kevin.Greenan@emc.com

Ethan L. Miller
Computer Science Department
UC Santa Cruz
elm@cs.ucsc.edu

Appearing in FAST 2013:
11th USENIX Conference on File and Storage Technologies
February, 2013

http://web.eecs.utk.edu/~plank/plank/papers/FAST-2013-GF.html

Screaming Fast Galois Field Arithmetic Using Intel SIMD Instructions

James S. Plank
EECS Department
University of Tennessee

Kevin M. Greenan
EMC Backup Recovery
Systems Division

Ethan L. Miller
Computer Science Department
UC Santa Cruz

Abstract

Galois Field arithmetic forms the basis of Reed-Solomon and other erasure coding techniques to protect storage systems from failures. Most implementations of Galois Field arithmetic rely on multiplication tables or discrete logarithms to perform this operation. However, the advent of 128-bit instructions, such as Intel's Streaming SIMD Extensions, allows us to perform Galois Field arithmetic much faster. This short paper details how to leverage these instructions for various field sizes, and demonstrates the significant performance improvements on commodity microprocessors. The techniques that we describe are available as open source software.

1 Introduction

Storage systems rely on erasure codes to protect themselves from data loss. Erasure codes provide the basic underlying technology for RAID-6 systems that can tolerate the failures of any two disks [1, 7], cloud systems that tolerate larger numbers of failures [6, 12, 19, 21], and archival systems that tolerate catastrophic situations [18, 29, 31, 32]. The canonical erasure code is the Reed-Solomon code [27], which organizes a storage system as a set of linear equations. The arithmetic of these linear equations is *Galois Field arithmetic*, termed $GF(2^w)$, defined as arithmetic over w-bit words. When applied to a storage system, encoding and decoding compute these linear equations by multiplying large regions of bytes by various w-bit contants in $GF(2^w)$ and then combining the products using bitwise exclusive-or (XOR). Thus, the two fundamental operations are performing multiplication of a region of bytes by a constant, and performing the XOR of two regions of bytes.

Historically, implementations of Galois Field arithmetic use multiplication tables for small values of w, logarithm tables for larger values, and incremental shifters for even larger values [9, 21, 22]. In practice, the performance of multiplication is at least four times slower than XOR [26].

In recent years, processors that implement Intel's Streaming SIMD Extensions instruction set have become ubiquitous. The SIMD instructions allow 128-bit numbers to be manipulated in the CPU, and their ramifications for multiplying regions of numbers by constants in $GF(2^w)$ are significant. Anecdotes of performing Galois Field arithmetic "at cache line speeds" have become commonplace at meetings such as Usenix FAST. However, as these anecdotes have typically come from those working at storage companies, the exact mechanics of using the SIMD instructions have been guarded.

This short paper details the SIMD instructions to multiply regions of bytes by constants in $GF(2^w)$ for $w \in \{4, 8, 16, 32\}$. These are the most common values of w in storage installations (please see Section 2 for a discussion of why w matters in a storage system). Each value of w requires different implementation techniques, some of which are subtle. We present each technique in enough detail for a reader to implement it in his or her own storage system, and we detail an open source implementation. The performance of these implementations is 2.7 to 12 times faster than other implementations of Galois Field arithmetic.

2 Erasure Codes and Galois Fields

Erasure codes that are based on Galois Field arithmetic are defined by a set of linear equations. A storage system composed of n disks is partitioned into k that hold data and m that hold coding information that is calculated from the data. For the purposes of the code, each disk logically stores one w-bit word. Suppose the words on the data disks are labeled $d_0, \ldots, d_{k-1}$ and the words on the coding disks are labeled $c_0, \ldots, c_{m-1}$. Then creating the coding words from the data words may be ex-

pressed by m equations (where arithmetic is in $GF(2^w)$:

$$\text{For } 0 \leq i < m: \quad c_i = \sum_{j=0}^{k-1} a_{i,j} d_j. \tag{1}$$

For example, the RAID-6 installation in the Linux kernel has two coding disks, c_0 and c_1, which are created by two equations [1]:

$$\begin{aligned} c_0 &= d_0 + d_1 + d_2 + \ldots + d_{k-1} \\ c_1 &= d_0 + 2d_1 + 4d_2 \ldots + 2^{k-1} d_{k-1} \end{aligned}$$

When up to m disks fail, we are left with m equations with up to m unknown variables. We decode by solving those equations with Gaussian elimination or matrix inversion.

There are a variety of erasure codes that are organized in this fashion. The most prevalent one is the Reed-Solomon code [27], which employs a generator matrix to define the above equations for any value of k and m, so long as $k + m \leq 2^w$. Reed-Solomon codes are *Maximum Distance Separable (MDS)*, which means that they tolerate the loss of any m disks. There is a detailed tutorial by Plank that spells out exactly how to implement Reed-Solomon codes in storage systems [22, 24].

More recently, other erasure codes have been developed that are based on the above methodology. Pyramid codes [11], LRC codes [12] and F-MSR codes [10] are all based on Galois Field arithmetic to achieve improved encoding performance, tolerance to sector failures, and improved decoding performance. LRC codes are the erasure code currently employed by the Microsoft Azure cloud storage system [12].

When implementing an erasure code, a value of w must be selected. This has three main impacts. First, it impacts the size of the storage system. For example, when using Reed-Solomon codes, a value of $w = 4$ can be used on systems up to 16 disks in size. A value of $w = 8$ expands that to 256 disks, and a value of $w = 16$ expands that to 65,536 disks. Second, it impacts layout. Implementors typically choose values of w that are factors of two so that w-bit words in the coding system fit precisely into machine words. Finally, it impacts performance. We will explore performance below, but the rule of thumb is that larger values of w perform more slowly than smaller values. For that reason, one typically chooses the smallest value of w such that w is a power of two and has the properties needed for the size of one's storage system. For example, Azure uses $w = 4$ [12], while Cleversafe uses $w = 8$ [28]. There are benefits to larger w that justify the extra complexity of implementation. For example, HAIL blends security and erasure coding, using large values of w such as 32 and 64 [21].

3 XOR's and Region Multiplication

Although coding equations such as (1) work on single words, in reality, the data and coding words are larger regions of bytes, such as disk sectors in RAID systems, or very large blocks of sectors in cloud systems [12, 17]. The reason is that when one partitions a region of bytes into multiple words, one can perform operations on them in parallel.

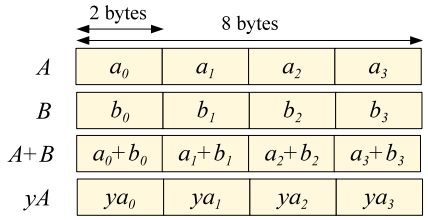


Figure 1: Two 8-byte regions of memory A and B are partitioned into four 16-bit words. The last two lines show how addition and multiplication are mapped to the individual words of the regions.

We depict an example in Figure 1. Here we have two 8-byte regions of memory, A and B, each of which is partitioned into four 16-bit words, $a_0, \ldots, a_3$ and $b_0, \ldots, b_3$. With Galois Field arithmetic, addition is equivalent to XOR; thus adding the region A and B is equivalent to adding each a_i with each b_i, and it may be implemented with a single 64-bit XOR operation. One may view multiplication of A by a constant y as multiplying each a_i by y. Performing this operation fast is the subject of this paper.

4 Streaming SIMD Instructions

Intel's Streaming SIMD Instructions [15] have has become ubiquitous in today's commodity microprocessors. They are supported in CPUs sold by Intel, AMD, Transmeta and VIA. Compiler support for these instructions have been developed as well; however, leveraging these instructions for Galois Field arithmetic requires too much application-specific knowledge for the compilers.

The basic data type of the SIMD instructions is a 128-bit word, and we leverage the following instructions in our implementations:

- *mm_set1_epi8(b)* creates a 128-bit variable by replicating the byte b sixteen times. *mm_set1_epi16(b)*, *mm_set1_epi32(b)* and *mm_set1_epi64(b)* set the variable by replicating 2-byte, 4-byte and 8-byte words respectively.
- *mm_and_si128(a, b)* and *mm_xor_si128(a, b)* perform bitwise AND and bitwise XOR on 128-bit words a and b respectively.

byte	f	e	d	c	b	a	9	8	7	6	5	4	3	2	1	0
table1:	0b	0c	05	02	04	03	0a	0d	06	01	08	0f	09	0e	07	00
table2 = mm_slli_epi64(table1, 4):	b0	c0	50	20	40	30	a0	d0	60	10	80	f0	90	e0	70	00
mask1 = mm_set1_epi8(0xf):	0f	0f	0f	0f	0f	0f	0f	0f	0f	0f	0f	0f	0f	0f	0f	0f
mask2 = mm_set1_epi8(0xf0):	f0	f0	f0	f0	f0	f0	f0	f0	f0	f0	f0	f0	f0	f0	f0	f0
A:	39	1d	9f	5a	aa	ab	15	c3	63	e0	7c	43	fb	83	16	23
l = mm_and_si128(A, mask1):	09	0d	0f	0a	0a	0b	05	03	03	00	0c	03	0b	03	06	03
l = mm_shuffle_epi8(l, table1):	0a	05	0b	03	03	04	08	09	09	00	02	09	04	09	01	09
h = mm_and_si128(A, mask2):	30	10	90	50	a0	a0	10	c0	60	e0	70	40	f0	80	10	20
h =mm_srli_epi64(h, 4):	03	01	09	05	0a	0a	01	0c	06	0e	07	04	0f	08	01	02
h = mm_shuffle_epi8(h, table2):	90	70	a0	80	30	30	70	20	10	c0	60	f0	b0	d0	70	e0
yA = mm_xor_si128(h, l):	9a	75	ab	83	33	34	78	29	19	c0	62	f9	b4	d9	71	e9

Figure 2: Multiplying a 128-bit region A by $y = 7$ in $GF(2^4)$. The first few instructions show variables that are set up before performing the multiplication. The last six perform the 32 multiplications using two table lookups.

- *mm_srli_epi64(a, b)* treats a as two 64-bit words, and right shifts each by b bits. *mm_slli_epi64(a, b)* performs left shifts instead.
- *mm_shuffle_epi8(a, b)* is the real enabling SIMD instruction for Galois Fields. Both a and b are 128-bit variables partitioned into sixteen individual bytes. The operation treats a as a 16-element table of bytes, and b as 16 indices, and it returns a 128-bit vector composed of 16 simultaneous table lookups, one for each index in b.

5 Calculating yA in $GF(2^4)$

When $w = 4$, there are only 16 values that a word may have. Thus, employing a 16×16 multiplication table requires very little memory, and such a table may be precomputed very quickly. While that suffices to perform multiplication of single values, it is less efficient when performing yA where y is a 4-bit word and A is a region of bytes, because a separate table lookup is required for every four bits in A.

The *mm_shuffle_epi8(a, b)* instruction may be leveraged so that yA may be performed 128-bits at a time with just two table lookup operations. We give a detailed example in Figure 2. In this example, we are multiplying a 16-byte region, A, by $y = 7$ in $GF(2^4)$. In the figure, we show 128-bit variables on the right and the instructions that create them on the left. The variables are presented as 16 two-digit numbers in hexadecimal. The figure is broken into three parts. The first part displays two multiplication tables that are created from y, and two masks. The second part shows the 16 bytes of A. The third part shows how to implement the multiplication. The low-order four bit words of each byte of A are isolated using the first mask, and they are used to perform 16 simultaneous table lookups that multiply those words by 7. The result is put into l. The high-order four bit words of each byte of A are then isolated using the second mask, right shifted by four bits and then used to perform table lookup in the second table. The result is put into h, which is combined with l to create the product. Thus, six instructions suffice for the 32 multiplications.

6 Calculating yA in $GF(2^8)$

As with $GF(2^4)$, one can implement multiplication in $GF(2^8)$ with table lookup. For a given value of y, one needs a 256-element table of bytes to look up each potential value of ya_i. However, *mm_shuffle_epi8()* only works on 16-element tables. To keep the prose clean, let us first drop the "i" subscript of a_i. To leverage *mm_shuffle_epi8()*, we observe that we can split a into two four-bit words, a_l and a_r, and then perform the multiplication with two table lookups. This is called a "left-right table" by Greenan *et al* [9].

To be precise, let a be an 8-bit word and let a_l and a_r be four-bit words so that:

$$a = (a_l \ll 4) \oplus a_r.$$

Then:

$$ya = y(a_l \ll 4) \oplus ya_r.$$

Thus, to implement multiplication, we create *table1* and *table2* as in Section 5. *Table1* contains the product of y with all four-bit words, a_r, and *table2* contains the product of y with all eight-bit words whose last four bits are zeros, $(a_l \ll 4)$, indexed by a_l. We give an example of these tables when $y = 7$ in Figure 3. Unlike Figure 2,

these tables are not four-bit shifts of each other. Instead, each table contains the eight bit product of 7 with another element of $GF(2^8)$ whose first or last four bits happen to equal zero. For example, in $GF(2^8)$, the product of 7 and 0xa is 0x36, and the product of 7 and 0xa0 is 0x47.

byte	f	e	d	c	b	a	9	8	7	6	5	4	3	2	1	0
table1:	2d	2a	23	24	31	36	3f	38	15	12	1b	1c	09	0e	07	00
table2:	ea	9a	0a	7a	37	47	d7	a7	4d	3d	ad	dd	90	e0	70	00

Figure 3: The two tables for multiplying 128-bit regions by $y = 7$ in $GF(2^8)$.

After the two tables are constructed, the product yA is calculated with the exact same six instructions as in Figure 2. Since the tables are small, and there are only 256 potential values of y, they may be precomputed.

It should be noted that this implementation technique was documented in assembly code by Anvin in his explanation of RAID 6 decoding in the Linux kernel [1].

7 Calculating yA in $GF(2^{16})$

A similar technique may be employed for $GF(2^{16})$; however, there are quite a few subtleties. As above, so that we can use *mm_shuffle_epi8()*, we split each a into subwords whose sizes are four bits. Let those words be a_0 through a_3. Then we calculate the product with the equation:

$$ya_i = y(a_3 \ll 12) + y(a_2 \ll 8) + y(a_1 \ll 4) + ya_0.$$

Unfortunately, we cannot map each subproduct to a single *mm_shuffle_epi8()* instruction, because each subproduct is a 16-bit word, and *mm_shuffle_epi8()* only performs table lookup on bytes. Thus, we need to use two tables per subproduct: one that holds the low-order bytes of each product, and one that holds the high-order bytes. This is a total of eight tables, which we label Ti_{high}, for the tables that calculate the high product bytes using a_i, and Ti_{low}, for the tables that calculate the low bytes using a_i.

To fully utilize *mm_shuffle_epi8()'s* ability to perform 16 simultaneous table lookups, it is best *not* to map words to contiguous memory locations as in Figure 1. Instead, every set of 16 words is mapped to two 128-bit variables, where the high byte (a_3 and a_2) of each word is in the first variable, and the low byte (a_1 and a_0) is in the second variable.

Because this can be quite confusing, we show a picture in Figure 4, which shows the first six bytes of two 128-bit variables, A_{high} and A_{low}. The variables together hold sixteen 16-bit words, of which we show three, b, c and d, in the picture. The high bytes of b, c and d are held in A_{high} and the low bytes are held in A_{low}.

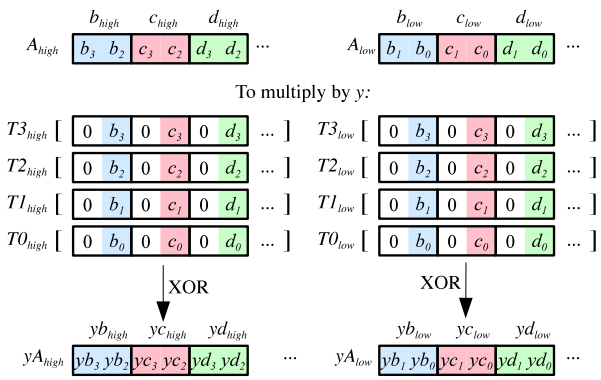


Figure 4: Using eight lookup tables and an alternate mapping of words to memory, to multiply a region A of sixteen 16-bit words in $GF(2^{16})$. Each rectangle is a byte, and only three words, b, c and d are shown. Each word is split into two bytes, one of which is stored in A_{high} and one in A_{low}. Each product yA_{high} and yA_{low} requires four *mm_shuffle_epi8()* operations, plus some bit masks and shifts.

Calculating the products requires four *mm_shuffle_epi8()* operations for each of yA_{high} and yA_{low}. For yA_{high}, we use the tables Ti_{high} and for yA_{low}, we use the tables Ti_{low}. By splitting each 16-bit word into two 128-bit variables, we can fully utilize *mm_shuffle_epi8()*'s ability to perform 16 simultaneous table lookups.

We call this technique *Altmap*, because it maps words to memory differently than the standard mapping of Figure 1. The down sides to *Altmap* are that memory regions are constrained to be multiples of 32 bytes, and that it is more confusing than the standard mapping. For the purposes of erasure coding, however, memory regions are only multiplied by constants and XOR'd together, so one only needs to actually extract the 16-bit values from the memory regions when debugging.

The eight tables consume a total of 128 bytes, and there are 64K potential values of y, so it is conceivable that the tables may be precomputed and stored. However, computing them at the beginning of the multiplication operation takes under 200 instructions, which means that table creation may be amortized by multiplying larger regions of memory by the same constant.

7.1 Using the Standard Mapping

If the standard mapping must be employed, for example to be interoperable with other erasure coding implementations that employ the standard mapping, then

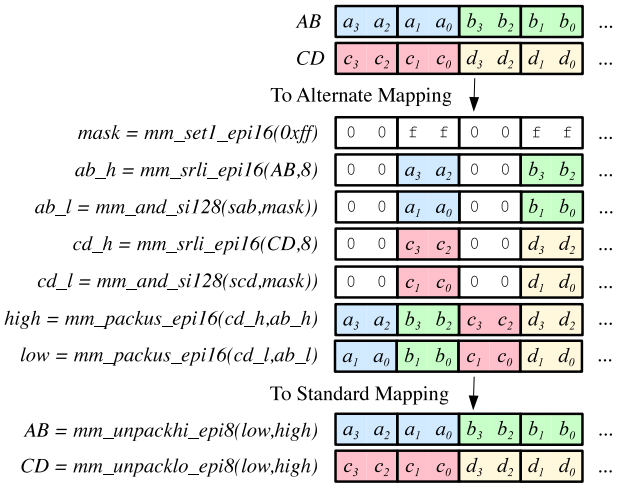


Figure 5: The SIMD instructions for converting two 128-bit variables to the alternate mapping and back again.

one may convert the standard mapping to the alternate mapping before multiplying each set of 16 words, and then convert back. The conversions leverage the instructions *mm_packus_epi16()*, *mm_unpackhi_epi8()* and *mm_unpacklo_epi8()*, which pack two 128-bit variables into one, and unpack one 128-bit variable into two. We demonstrate the operations in Figure 5, which pictures the first four bytes of each 128-bit variable.

8 Calculating yA in $GF(2^{32})$

The same technique for $w = 16$ generalizes to larger values of w. For $w = 32$, we partition each word a into eight four-bit words which are be used as indices for *mm_shuffle_epi8()* calls. Since the products are 32-bit words, there must be four tables for each four-bit word, for a total of 32 different lookup tables. As with $w = 16$, we employ the alternate mapping so that each set of sixteen 32-bit words is held in four 128-bit vectors in such a way that each vector holds one byte from each word. We may convert the standard mapping to the alternate mapping and back again using a similar technique as with $w = 16$.

9 Performance

We have released an open source library in C called GF-Complete [25]. The library implements all documented techniques for performing Galois Field arithmetic, including logarithm and multiplication tables from standard libraries like **jerasure** [26] and Rizzo's FEC library [30], split multiplication tables [26, 9], composite fields [9, 21], bit-grouping tables [21] and Anvin's technique based on fast multiplication by two, which is included in the Linux kernel and has received further attention that leverages GPU co-processors [1, 16].

Our performance evalution briefly demonstrates the dramatic speed improvements due to these techniques. GF-Complete includes a performance testing module which fills regions of bytes with random values, and then multiplies those regions by contants in $GF(2^w)$. We show results of the performance tester in Figure 6. The testing machine is a 3.4 GHz Intel Core i7-3770 with 16 GB of DRAM, a 256 KB L2 cache and an 8 MB L3 cache. In the tests, we vary the region sizes from 1 KB to 1 GB in multiples of four and encode a total of 5 GB.

On the rightmost graph, we include baseline performance numbers for reference: the speed of **memcpy()**, the speed of *mm_xor_si128()*, and the speed of multiplying regions by two using Anvin's SIMD optimization from the Linux kernel [1], which is independent of w.

For controls, we include historically "standard" techniques as implemented by **jerasure**/Rizzo [26, 30]: Multiplication tables for $w \in \{4, 8\}$, logarithm tables for $w = 16$ and seven "split" multiplication tables, each of size 256x256 for $w = 32$. For $w \in \{4, 8, 16\}$, we also include "16-bit tables." These break up the regions into 16-bit words and use them as indices into a table that is created at the beginning of multiplication (except for $w = 4$, where the tables are precomputed). This improves upon the standard multiplication tables for $w \in \{4, 8\}$ because 16 bits of table-lookup are performed at a time, as opposed to four table lookups for $w = 4$ and two for $w = 8$. We also include "By-two," which structures multiplication by selectively multiplying 128-bit regions by two using Anvin's optimization [1, 16].

The SIMD implementations and the baselines follow similar trajectories which are cache dependent. As the region sizes grow, performance improves as setup costs are amortized. Performance reaches a peak as caches are saturated: **Memcpy()** and XOR are limited by the L1 cache; while the SIMD techniques and Anvin's optimization are limited by the L2 cache. When the L2 cache is saturated (at a region size of 256 KB), performance drops slightly. When the L3 cache is saturated (at a region size of 4 MB), the performance drops dramatically. In fact, then the L3 cache is saturated, the SIMD performance, even at $w = 32$, matches that of XOR.

Because they perform roughly the same operations, $w = 4$ and $w = 8$ perform identically. $w = 16$ performs slightly slower, and $w = 32$ slower still. With respect to the controls, at their peak, the SIMD implementations perform 2.7 to to 12 times faster than the other implementations. When $w \in \{16, 32\}$, the alternate mappings perform 48 and 33 percent faster than the standard mappings, which require conversion to the alternate mapping and back again.

The base conclusion to draw from the tests in Figure 6

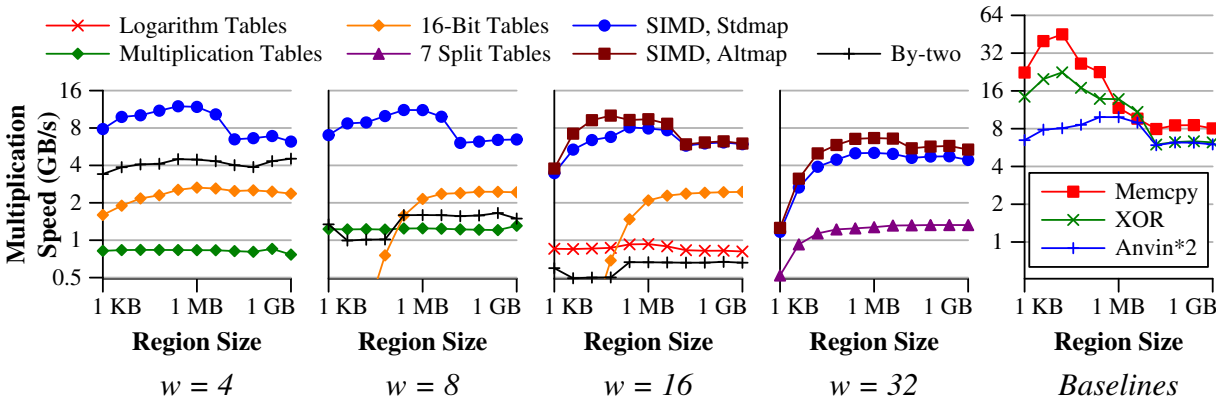


Figure 6: The performance of multiplying regions of bytes by constants in $GF(2^w)$.

is that with the SIMD instructions, the cache becomes the limiting factor of multiplication, and I/O becomes the dominant concern with erasure coding.

10 Related Work

There has been quite a lot of work on implementing Galois Field arithmetic in software. The various implementation techniques detailed in Section 9 are explained in papers by Anvin [1], Greenan [9], Kalcher [16], Lopez [20], Luo [21] and Plank [22]. The implications on erasure coding systems has been demonstrated to varying degrees by Greenan [9], Hu [10], Huang [12], Khan [17] and Luo [21]. The techniques in this paper have been leveraged to sustain throughputs of over 4 GB/s in recent tests of Reed-Solomon coding [23].

Of particular interest is the judicious design of RAID 6 in the Linux kernel [1]. By employing the encoding equations c_0 and c_1 from Section 2, one coding disk requires only XOR calculations, and the second requires XOR's and multiplications by two using the baseline implementation from Figure 6. General-purpose multiplication is only required upon decoding, and the kernel employs a technique which is identical to our technique for $GF(2^8)$, albeit implemented directly in assembly code. As such, the CPU performance of encoding and decoding is thoroughly optimized.

This paper focuses solely on multiplying regions of bytes by constants. Its results do not apply to optimizing single multiplication operations. For these, the standard tables work the best for $w \leq 16$, and the grouping techniques of Luo perform the best for $w = 32$ [21].

While we only employ Intel's SIMD instructions, other instructions sets like Altivec and AVX2 feature permutation instructions that may be leveraged similarly. The Intel Performance Primitives library [14] has deprecated (as of revision 7.1) Galois Field operations; however, these only support single multiplications for $GF(2^8)$ and cannot be leveraged in a SIMD fashion.

11 Conclusion

In this paper, we demonstrate how to leverage 128-bit SIMD instructions to perform multiplication of a region of bytes by a constant in Galois Fields $GF(2^w)$, where $w \in \{4, 8, 16, 32\}$. For $w \in \{16, 32\}$, an alternate mapping of words to memory allows us to further optimize performance. For a small penalty, one may convert this mapping back to the standard mapping.

We have implemented these techniques in an open-source library whose performance we have tested and compared to other Galois Field implementations. The improvement ranges from 2.7 to 12 times faster than the traditional implementations, and helps to perpetuate a trend of worrying more about I/O than CPU performance in erasure coding settings.

The speed of multiplication using these techniques is so much faster than previous implementations that it has implications on the design of erasure codes. A previous assumption in erasure code design has been that erasure codes based on XOR operations are significantly faster than those based on Galois Field arithmetic. This assumption has led to the design of many XOR-only erasure codes [2, 4, 5, 7, 13, 33]. When Galois Field multiplication is cache-limited, erasure codes based on Galois Field arithmetic become viable alternatives to XOR codes. Recent examples of codes based on Galois Field arithmetic are LRC and Rotated Reed-Solomon codes [12, 17] for improved recovery performance in cloud storage systems, regenerating codes to lower network bandwidth in decentralized storage settings [8, 10], and PMDS/SD codes to improve storage efficiency in disk arrays [3, 23]. The applicability of these codes is heightened by the techniques in this paper, and we anticipate that future code design will rely more on Galois Field arithmetic than on XOR's.

12 Acknowledgements

This material is based upon work supported by the National Science Foundation under grants CNS-0917396, IIP-0934401 and CSR-1016636. The authors would like to thank shepherd Joseph Tucek and the FAST referees for constructive comments on the paper.

References

[1] H. P. Anvin. The mathematics of RAID-6. `http://kernel.org/pub/linux/kernel/people/hpa/raid6.pdf`, 2011.

[2] M. Blaum, J. Brady, J. Bruck, and J. Menon. EVENODD: An efficient scheme for tolerating double disk failures in RAID architectures. *IEEE Transactions on Computing*, 44(2):192– 202, February 1995.

[3] M. Blaum, J. L. Hafner, and S. Hetzler. Partail-MDS codes and their application to RAID type of architectures. IBM Research Report RJ10498 (ALM1202-001), February 2012.

[4] M. Blaum and R. M. Roth. On lowest density MDS codes. *IEEE Transactions on Information Theory*, 45(1):46–59, January 1999.

[5] J. Blomer, M. Kalfane, M. Karpinski, R. Karp, M. Luby, and D. Zuckerman. An XOR-based erasure-resilient coding scheme. Technical Report TR-95-048, International Computer Science Institute, August 1995.

[6] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, J. Haridas, C. Uddaraju, H. Khatri, A. Edwards, V. Bedekar, S. Mainali, R. Abbasi, A. Agarwal, M. Fahim ul Haq, M. Ikram ul Haq, D. Bhardwaj, S. Dayanand, A. Adusumilli, M. McNett, S. Sankaran, K. Manivannan, and L. Rigas. Windows Azure Storage: A highly available cloud storage service with strong consistency. In *23rd ACM Symposium on Operating Systems Principles*, October 2011.

[7] P. Corbett, B. English, A. Goel, T. Grcanac, S. Kleiman, J. Leong, and S. Sankar. Row diagonal parity for double disk failure correction. In *3rd Usenix Conference on File and Storage Technologies*, San Francisco, CA, March 2004.

[8] A. G. Dimakis, K. Ramchandran, Y. Wu, and C. Suh. A survey on network codes for distributed storage. *Proceedings of the IEEE*, 99(3), March 2011.

[9] K. Greenan, E. Miller, and T. J. Schwartz. 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.

[10] Y. Hu, H. C. H. Chen, P. P. C. Lee, and Y. Tang. NCCloud: Applying network coding for the storage repair in a cloud-of-clouds. In *FAST-2012: 10th Usenix Conference on File and Storage Technologies*, San Jose, February 2012.

[11] C. Huang, M. Chen, and J. Li. Pyramid codes: Flexible schemes to trade space for access efficienty in reliable data storage systems. In *NCA-07: 6th IEEE International Symposium on Network Computing Applications*, Cambridge, MA, July 2007.

[12] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, and S. Yekhanin. Erasure coding in Windows Azure storage. In *USENIX Annual Technical Conference*, Boston, June 2012.

[13] C. Huang and L. Xu. STAR: An efficient coding scheme for correcting triple storage node failures. *IEEE Transactions on Computers*, 57(7):889–901, July 2008.

[14] Intel Corporation. Intel Integrated Performance Primitives for Intel architecture, reference manual IPP 7.1. `http://software.intel.com`, 2000-2012.

[15] Intel Corporation. Intel 64 and IA-32 architectures software developers manual, combined volumes: 1, 2A, 2B, 2C, 3A, 3B and 3C. Order Number: 325462-044US, August 2012.

[16] S. Kalcher and V. Lindenstruth. Accelerating Galois Field arithmetic for Reed-Solomon erasure codes in storage applications. In *IEEE International Conference on Cluster Computing*, 2011.

[17] O. Khan, R. Burns, J. S. Plank, W. Pierce, and C. Huang. Rethinking erasure codes for cloud file systems: Minimizing I/O for recovery and degraded reads. In *FAST-2012: 10th Usenix Conference on File and Storage Technologies*, San Jose, February 2012.

[18] A. Leventhal. Triple-parity RAID and beyond. *Communications of the ACM*, 53(1):58–63, January 2010.

[19] X. Li, A. Marchant, M. A. Shah, K. Smathers, J. Tucek, M. Uysal, and J. J. Wylie. Efficient eventual consistency in Pahoehoe, an erasure-coded key-blob archive. In *DSN-10: International Conference on Dependable Systems and Networks*, Chicago, 2010. IEEE.

[20] J. Lopez and R. Dahab. High-speed software multiplication in f_{2^m}. In *Annual International Conference on Cryptology in India*, 2000.

[21] J. Luo, K. D. Bowers, A. Oprea, and L. Xu. Efficient software implementations of large finite fields $GF(2^n)$ for secure storage applications. *ACM Transactions on Storage*, 8(2), February 2012.

[22] J. S. Plank. A tutorial on Reed-Solomon coding for fault-tolerance in RAID-like systems. *Software – Practice & Experience*, 27(9):995–1012, September 1997.

[23] J. S. Plank, M. Blaum, and J. L. Hafner. SD codes: Erasure codes designed for how storage systems really fail. In *FAST-2013: 11th Usenix Conference on File and Storage Technologies*, San Jose, February 2013.

[24] J. S. Plank and Y. Ding. Note: Correction to the 1997 tutorial on reed-solomon coding. Technical Report CS-03-504, University of Tennessee, April 2003.

[25] J. S. Plank, K. M. Greenan, E. L. Miller, and W. B. Houston. GF-Complete: A comprehensive open source library for Galois Field arithmetic. Technical Report UT-CS-13-703, University of Tennessee, January 2013.

[26] J. S. Plank, J. Luo, C. D. Schuman, L. Xu, and Z. Wilcox-O'Hearn. A performance evaluation and examination of open-source erasure coding libraries for storage. In *FAST-2009: 7th Usenix Conference on File and Storage Technologies*, pages 253–265, February 2009.

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

[28] J. K. Resch and J. S. Plank. AONT-RS: blending security and performance in dispersed storage systems. In *FAST-2011: 9th Usenix Conference on File and Storage Technologies*, pages 191–202, San Jose, February 2011.

[29] S. Rhea, P. Eaton, D. Geels, H. Weatherspoon, B. Zhao, and J. Kubiatowitz. Pond: The OceanStore prototype. In *FAST-2003: 2nd Usenix Conference on File and Storage Technologies*, San Francisco, January 2003.

[30] L. Rizzo. Effective erasure codes for reliable computer communication protocols. *ACM SIGCOMM Computer Communication Review*, 27(2):24–36, 1997.

[31] M. W. Storer, K. M. Greenan, E. L. Miller, and K. Voruganti. Pergamum: Replacing tape with energy efficient, reliable, disk-based archival storage. In *FAST-2008: 6th Usenix Conference on File and Storage Technologies*, pages 1–16, San Jose, February 2008.

[32] M. W. Storer, K. M. Greenan, E. L. Miller, and K. Voruganti. POTSHARDS – a secure, long-term storage system. *ACM Transactions on Storage*, 5(2), June 2009.

[33] L. Xu and J. Bruck. X-Code: MDS array codes with optimal encoding. *IEEE Transactions on Information Theory*, 45(1):272–276, January 1999.

Managing Flocking Objects with an Octree Spanning a Parallel Message-Passing Computer Cluster

Thomas E. Portegys, Kevin M. Greenan
Illinois State University
portegys@ilstu.edu, kmgreen2@ilstu.edu

Abstract

We investigate the management of flocking mobile objects using a parallel message-passing computer cluster. An octree, a data structure well-known for use in managing a 3D space, is adapted to "span" the cluster. Objects are distributed in the tree, and partitions of the tree are distributed among the processors in such a way that a minimum of global information is required to be shared by the processors. When objects move, the tree is modified accordingly; this in turn may cause partitions to migrate processors. Two constraints drive the distribution algorithm: (1) minimizing message traffic by clustering nearby objects on the same processor, and (2) processor load-balancing. Boids, flocking artificial life forms, embody the objects in this study. The performance of the system is measured in terms of the inter-processor message traffic as a function of the number, interactivity, and mobility of objects. An application of the scheme allows external clients to view objects in specified spatial loci.

Keywords: message-passing parallel computer, octree, flocking behavior, boids.

1. Introduction

Many systems involve large numbers of mobile and spatially related components. Examples include simulating molecular changes in chemical reactions, weather modeling, air and fluid dynamics, population modeling, forest fire simulation, and networked gaming. These systems require the sort of massive computational power that parallel processors can provide in an economical fashion.

We chose a message-passing parallel computer cluster as our platform. Message-passing machines are typically less efficient than shared-memory processors for tasks involving relatively small numbers of tightly connected objects, and in turn show superior performance and scaling for tasks involving large numbers of loosely coupled objects [8]. One of the aims of this project is to investigate the conditions under which the use of a message-passing system is effective. To this end, we measure the performance of the system in terms of the inter-processor message traffic as a function of the number of objects and their mobility.

A number of investigations of N-body tasks on message-passing parallel clusters appear in the literature [3,4,7]. In our study, boids [6], flocking artificial life forms, are used as test objects. A boid moves about in a swarming fashion that requires knowledge other boids in its vicinity. This produces group flocking movements that are characteristic of animal and human behavior in contrast to typical N-body movements caused by force-fields. The movement of flocks across processor boundaries provides an opportunity to study the processing and message-passing capabilities of the system, as well as to investigate the effectiveness of load-balancing.

1.1. Spanning octree

An octree, a data structure well-known for use in managing 3D space, is adapted to "span" the cluster. In our octree, space is recursively divided into smaller and cubic partitions such that terminal cubes usually contain a single object. The tree is partitioned by orthogonally bisecting space and assigning volumes to each processor in the form of bounds. Figure 1 depicts a 2D (quadtree) view of a tree spanning a space containing a number of objects.

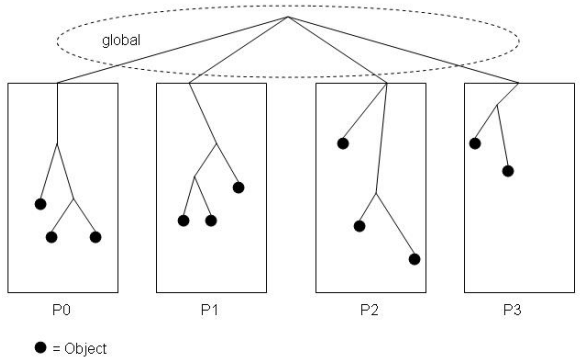


Figure 1. Initial Spanning Tree

In the figure, the partition bounds are globally known to processors P0 through P3. This is accomplished by replicating the bounds so that each processor has its own copy. The goal is to provide a fast determination of which partitions of the octree are managed by which processor so that objects in the corresponding volumes of space can be accessed. Each processor manages a single volume of space and the objects that it contains. The specific details of the objects in these local spaces, including the configurations of the sub-trees that track them, are known only to each processor.

1.2. Migrating objects

Figure 2 shows objects O1 and O2 migrating to the space owned by processor P3. This involves at a minimum messages from P2 to P3 to insert the migrating objects in the target space. P2 then deletes its copy of the objects.

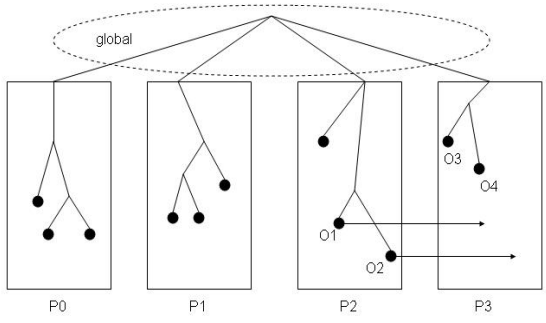


Figure 2. Migrating objects

1.3. Load-balancing

Two constraints drive the distribution algorithm: (1) minimizing message-passing by clustering nearby objects on the same processor, and (2) processor load-balancing. Figure 3 shows the result of load-balancing that has caused the global space to be repartitioned as a result of the migration of objects O1 and O2. This has caused the redistribution of objects O3 and O4 to P2, and forced an update to the global bounds. We use a variation of an orthogonal recursive bisection algorithm [5] to partition space.

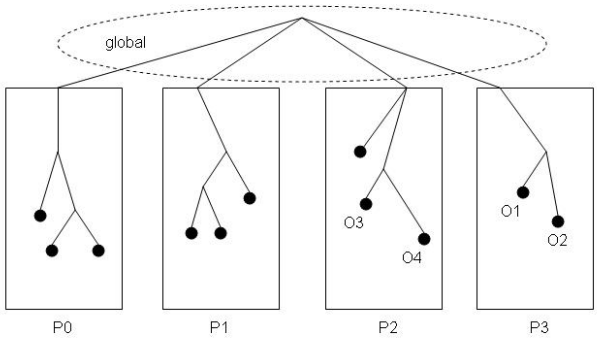


Figure 3. Repartitioned space

1.4. Object interaction

In contrast with typical N-body systems, a boid does not often interact with every other boid. A boid's movement depends only on the movements and positions of nearby boids. Because of this, a straightforward octree search is sufficient to implement efficient proximity checking, as opposed to using a cumulative scheme such as the multipole method [1,2].

Figure 4 illustrates a situation in which objects O1 and O3 reside in space managed by processor P0, and object O2 resides in space managed by P1. Consider proximity checking for O1. In the case of an O1-O3 interaction, the checking can be entirely local to P0. However, since O1 extends into space owned by P1, and the contents of this space are unknown to P0, a message to P1 must be sent containing O1's position so that P1 may conduct the proximity checking in its local space.

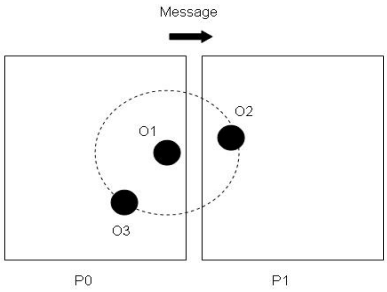


Figure 4. Proximity checking

1.5. Viewports

An application of the scheme allows external clients to view objects in specified spatial loci. This is accomplished by designating one of the processors as a viewport gateway. Alternatively, a dedicated machine may perform the gateway function. Clients connecting to the gateway specify a viewing frustum, which is essentially a bounding box. The gateway then makes appropriate searches on the various processors to obtain information about objects contained in the viewing frustum. A list of these is returned to the client, allowing a graphical view of a volume of space to be rendered.

2. Procedure

Our platform is the Applied Computer Science Department's Beowulf machine, a cluster of 16 SUN Ultra 10 workstations running SuSe Linux, connected by a 10mbps Ethernet. The software is the C++ programming language and PVM (Parallel Virtual Machine) to provide the message-passing infrastructure. The OpenGL graphics language is used to exercise the viewport feature.

The partition algorithm requires that the number of processors be a power of 8, yet for obvious reasons this is not a practically achievable machine configuration. The solution was to implement processors as virtual entities and distribute processors among physical machines in a clustered manner.

The boids code was initially obtained from an internet source. After a measure of re-writing and parameter tuning we obtained satisfactory flocking behavior: a variety of dynamically changing flock sizes.

A PVM program is a master-slave process configuration. Our master process spawned slave processes representing virtual processors on specified machines, initializing them with their bounds and a random distribution of boids.

Updating processor bounds for load-balancing is done by the master using information from the slave processors. Each processor reports its current load (number of boids) and the position of the median boid. The master then re-partitions based on these weighted boid positions.

After initialization, the master enters a loop, an iteration of which constitutes the following cycle:

1. Broadcast an *aim* message to slave processors causing them to determine the next position of each boid. This entails intra-processor and cross-processor searches not involving the master.
2. Broadcast a *move* message, causing processors to move boids to their new positions, possibly involving cross-processor insertions and deletions, also not involving the master. Insertions are unacknowledged for efficiency reasons.
3. If load-balancing, broadcast a *report* message, causing the processors to send back their load and median information. The master computes new bounds and broadcasts them in a *balance* message.
4. If gathering statistics, broadcast a *stats* messages and gather results.
5. If in viewing mode, broadcast the viewing planes in a *view* message and gather the results. Each processor determines which searches its octree for boids falling within the viewing box.

The cycle steps are synchronized; each step is completed before moving to the next. This means that the master waits for all processors to respond before issuing the next command. This can only happen after all searching and insertion activity for a particular step is completed by the slaves.

The viewing capability is implemented as a separate thread within the master process. This allows a user to navigate through space, selectively viewing boids, or to run in non-interfering "blind" mode.

3. Results

Figure 5 is a graphical depiction of a simulation of the system. Here, a number of mobile point objects are shown in their octree volumes.

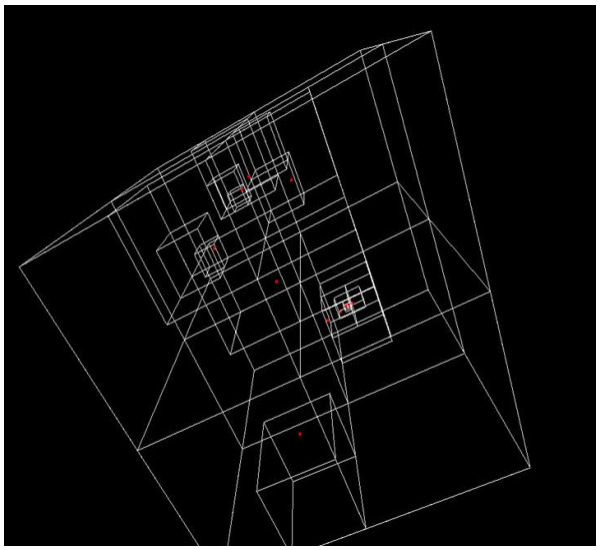


Figure 5. Octree simulation

The independent variables were: number of boids (25, 50, 100, 200), number of machines (4, 8), and load-balancing (on/off). Unfortunately, due to hardware problems we were not able to use the entire 16 processor cluster. The dependent variables for which data was gathered were: load (boids) per machine and message traffic. Each trial was run for 1000 cycles. The boids had an interaction range of 5 units. As the number of boids increased, the spatial dimension were increased: 25 boids in a 15x15x15 volume, 50 in 20x, 100 in 25x, and 200 in 30x.

Figures 6 and 7 show the average and standard deviation boid load for 8 and 4 machine configurations, respectively, under no load-balancing (NLB), and load-balancing (LB) conditions. The flocking aspect of the boids can be seen in the non-uniform distribution indicated by the standard deviation.

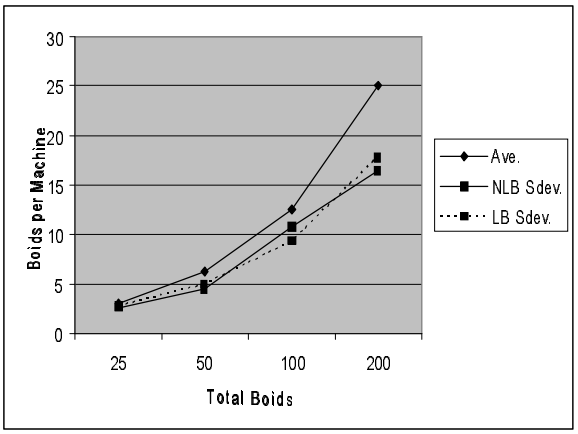


Figure 6. Load for 8 machines

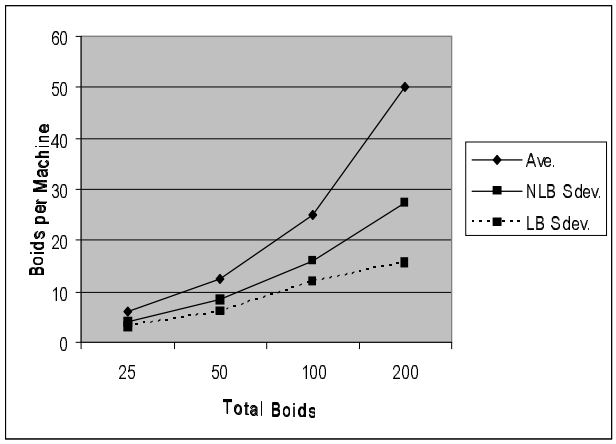


Figure 7. Load for 4 machines

Figures 8 and 9 show the message traffic for the 8 and 4 machine configurations. Notable here is the effect of load-balancing, which causes a significant decrease in message traffic, although not a correspondingly large decrease in burstiness as indicated by the standard deviation.

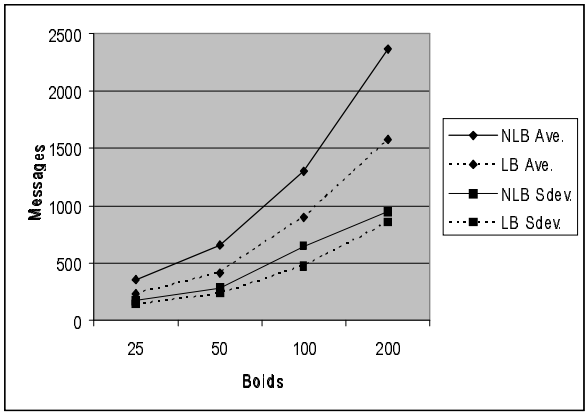


Figure 8. Message traffic for 8 machines

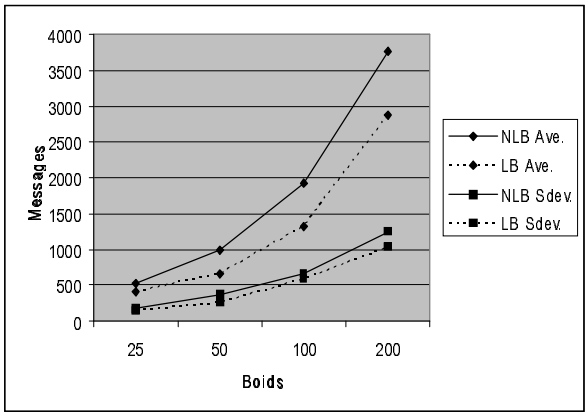


Figure 8. Message traffic for 4 machines

4. Conclusion

The overall observation is that load-balancing can be effective in reducing message traffic for flocking objects. The cost for this improvement in our scheme is an additional 2 steps in the processing cycle.

For future work, we propose to investigate decentralized load-balancing schemes to avoid the overhead cost. In addition, processor load could consist of factors other than simple numbers of objects. For example, the state of objects may be a viable factor. In a forest fire simulation, burning areas would take more computation resources, and thus these objects might "weigh" more heavily. In addition, in a cluster of heterogeneous processors, the resources of each processor could be taken into account for load-balancing.

The code is available at:
www.acs.ilstu.edu/faculty/portegys/research.html

5. Acknowledgements

The authors wish to especially thank Chris McBride for many valuable ideas and contributions. Thanks also to Andy Thayer and Tesh Shah for their insights.

6. References

[1] Anderson, C. R., "An implementation of the fast multipole method without multipoles", SIAM J. Sci. Stat. Comput. 13 (1992) 923.

[2] Greengard, L. Gropp, W.L., "A Parallel version of the Fast Multipole Method", Parallel Processing for Scientific Computing SIAM Conference proceedings, p213-222, 1988.

[3] Hariharan, B. and Aluru, S., "Efficient Parallel Algorithms and Software for Compressed Octrees with Applications to Hierarchical Methods", High Performance Computing - HiPC 2001 8th International Conference, Hyderabad, India, December, 17-20, 2001. Proceedings.

[4] Hu, Y., "Implementing O(N) N-body algorithms efficiently in data parallel languages (High Performance Fortran)", Journal of Scientific Programming, 1994.

[5] Salmon, J.K., "Parallel Hierarchical N-Body Methods", Ph.D. thesis, California Institute of Technology, 1990.

[6] Scientific American: Feature Article: "Boids of a Feather Flock Together", November 2000.

[7] Sun Y., Liang, Z., and Wang, C-L, "A Distributed Object-Oriented Method for Particle Simulations on Clusters", Proceedings of the 7th International Conference on High Performance Computing and Networking (HPCH Europe 1999), April 12-14, 1999, Amsterdam, The Netherlands, 1999.

[8] Wilkinson, B. and Allen M., "Parallel Programming, Techniques and Applications Using Networked Workstations and Parallel Computers", Prentice-Hall, Inc., 1999.

This paper will be published in the 6th *Conference on File and Storage Technologies (FAST '08)*, San Jose, California, February 26–29, 2008.

Pergamum: Replacing Tape with Energy Efficient, Reliable, Disk-Based Archival Storage

Mark W. Storer Kevin M. Greenan Ethan L. Miller Kaladhar Voruganti
University of California, Santa Cruz *Network Appliance*

Abstract

As the world moves to digital storage for archival purposes, there is an increasing demand for reliable, low-power, cost-effective, easy-to-maintain storage that can still provide adequate performance for information retrieval and auditing purposes. Unfortunately, no current archival system adequately fulfills all of these requirements. Tape-based archival systems suffer from poor random access performance, which prevents the use of inter-media redundancy techniques and auditing, and requires the preservation of legacy hardware. Many disk-based systems are ill-suited for long-term storage because their high energy demands and management requirements make them cost-ineffective for archival purposes.

Our solution, Pergamum, is a distributed network of intelligent, disk-based, storage appliances that stores data reliably and energy-efficiently. While existing MAID systems keep disks idle to save energy, Pergamum adds NVRAM at each node to store data signatures, metadata, and other small items, allowing deferred writes, metadata requests and inter-disk data verification to be performed while the disk is powered off. Pergamum uses both intra-disk and inter-disk redundancy to guard against data loss, relying on hash tree-like structures of algebraic signatures to efficiently verify the correctness of stored data. If failures occur, Pergamum uses staggered rebuild to reduce peak energy usage while rebuilding large redundancy stripes. We show that our approach is comparable in both startup and ongoing costs to other archival technologies and provides very high reliability. An evaluation of our implementation of Pergamum shows that it provides adequate performance.

1 Introduction

Businesses and consumers are becoming increasingly conscious of the value of archival data. In the business arena, data preservation is often mandated by law, and data mining has proven to be a boon in shaping business strategy. For individuals, archival storage is being called upon to preserve sentimental and historical artifacts such as photos, movies and personal documents. In both of these areas, archival systems must keep pace with a growing need for efficient, reliable, long-term storage.

Many storage systems designed for long-term data preservation rely on sequential-access technologies, such as tapes, that decouple media from its access hardware. While effective for back-up workloads (write-once, read-rarely, newer writes supersede old), such systems are poorly suited to archival workloads (write-once, read-maybe, new writes unrelated to old writes). With as many as 50–100 tapes per drive, a requirement to keep tapes running at full speed, and a linear media-access model, random-access performance with tape-media is relatively poor. This conspires against many archival storage operations — such as auditing, searching, consistency checking and inter-media reliability operations — that rely on relatively fast random-access performance. This is especially important in light of the preservation and retrieval demands of recent legislation [23, 30]. Further, many data retention policies include the notion of a limited lifetime, after which data is securely deleted; selective deletion is difficult and inefficient in linear media. Finally, the separation of media and access hardware introduces the need to preserve complex chains of hardware; reading an old tape requires a compatible reader, controller and software.

Recently, hard drives have dropped in price relative to tape, making them a potential alternative for archival storage [33]. The availability of high-performance, low-power CPUs [4] and inexpensive, high-speed networks have made it possible to produce a self-contained, network-attached storage device [16] with reasonable performance and low power utilization: as little as 500 mW when both the CPU and disk are idle. The use of disks instead of tapes means that heads are packaged

with media, removing the need for robotics and reducing physical movement and system complexity. Using standardized communication interfaces, such as TCP/IP over Ethernet, also helps simplify technology migration and long-term maintenance. By using randomly-accessible disks instead of linear tapes, systems can take advantage of inter-media redundancy schemes. Unfortunately, many existing disk-based systems incur high costs associated with power, cooling and administration because of design approaches that favor performance over energy efficiency. However, recent work on MAIDs (Massive Arrays of Idle Disks) has demonstrated that considerable energy-based cost savings can be realized while maintaining high levels of performance [10, 32, 45], though such systems often favor performance over even greater energy savings.

Our design differs from that of existing MAID systems, which still have centralized controllers. Instead, our system, Pergamum, takes an approach similar to that used in high-performance scalable storage systems [36, 46, 48], and is built from thousands of intelligent storage appliances connected by high-speed networks that cooperatively provide reliable, efficient, long-term storage. Each appliance, called a Pergamum *tome*, is composed of four hardware components: a commodity hard drive for persistent, large-capacity storage; on-board flash memory for persistent, low-latency, metadata storage; a low-power CPU; and a network port. Each appliance runs its own copy of the Pergamum software, allowing it to manage its own consistency checking, disk scrubbing and redundancy group responsibilities. Additionally, the CPU and extensible software layer enables disk-level processing, such as compression and virus checking. Finally, the use of standardized networking interfaces and protocols greatly reduces the problem of maintaining complex chains of dependent hardware.

Pergamum introduces several new techniques to disk-based archival storage. First, our system distributes control to the individual devices, rather than centralizing it, by including a low-power CPU and network interface on each disk; this approach reduces power consumption by eliminating the need for power-hungry servers and RAID controllers. Systems such as TickerTAIP [8] used distributed control in a RAID, but did not include reliability checking and power management. Second, Pergamum aggressively ensures data reliability using two forms of redundancy: intra-disk and inter-disk. In the former, each disk stores a small number of redundancy blocks with each set of data blocks, providing a self-sufficient way of recovering from latent sector errors [6]. In the latter, Pergamum computes redundancy information across multiple disks to guard against whole disk failure. However, unlike existing RAID systems, Pergamum can stagger inter-disk activity during data recovery, minimizing peak energy consumption during rebuilding. Third, energy-efficient decentralized integrity verification is enabled by storing data signatures for disk contents in NVRAM. Thus, using just the signatures, Pergamum tomes can verify the integrity of their local contents and, by exchanging signatures with other Pergamum tomes, verify the integrity of distributed data without incurring any spin up costs. Finally, the Pergamum architecture allows disk-based archives to look like tape: an individual Pergamum tome may be pulled out of the system and read independently; the remaining Pergamum tomes will eventually treat this event like a disk failure and rebuild the "missing" data in a new location.

The goal of Pergamum, is to realize significant cost savings by keeping the vast majority, as many as 95%, of the disks spun down while still providing reasonable performance and excellent reliability. Our techniques allow us to greatly reduce energy usage, as compared to traditional hard drive based systems, making it suitable for archival storage. The use of signatures to verify data reduces the need to power disks on, as does the reduced scrubbing frequency made possible by the extra safety provided by intra-disk parity. Similarly, staggering disk rebuilds reduces peak power load, again allowing Pergamum to reduce the maximum number of disks that must be active at the same time. While we believe these techniques are best realized in a distributed system such as Pergamum—the use of many low-power CPUs is more efficient than a few high-power servers—they are also suitable for use in more conventional MAID architectures, and could be used to reduce power consumption in them as well.

The remainder of this paper is organized as follows. Section 2, places Pergamum within the context of existing research. Following that, Section 3 a detailed discussion of the systems components, including a discussion of the components in each Pergamum tome. Then, Section 4 details the system's design, including its redundancy and power management approaches. Section 5 contains our evaluation of Pergamum in terms of cost, long-term reliability and performance. Finally, following a discussion of future work in Section 6, we conclude the paper in Section 7.

2 Related Work

In designing Pergamum to meet the goals of energy-efficient, reliable, archival storage [7], we used concepts from various systems. These projects can be distinguished from Pergamum by identifying their intended workload, cost strategy and redundancy strategy. As Table 1 illustrates, many existing systems fulfill some of the goals of Pergamum, but none adequately address all of its concerns.

System	Media	Workload	Redundancy	Consistency	Power Aware
PARAID	disk	server clusters	RAID		Yes
Nomad FS	disk	server clusters	none		Yes
Google File System	disk	data-intensive apps	replicas	relaxed	No
EMC Centera	disk	archival	mirroring or parity	WORM media	No
Venti	disk	archival	RAID 5	content-based naming, type ids	No
Deep Store	disk	archival	selectable replication	content-based naming	No
Copan Revolution 220A	disk	archival	RAID 5	SHA 256	Yes
Sun StorageTek SL8500	tape	backup	N+1	WORM media	No
RAIL	optical	backup, archival	RAID 4	optional write verification	No
Pergamum	disk	archival	2-level erasure coding	algebraic signatures	Yes

Table 1: Overview of storage systems described in Section 2.

A number of systems have also sought to achieve cost savings through the use of commodity hardware [14, 45]. Typically, this strategy assumes that cheaper SATA drives will fail more often than server-class hardware, requiring that the solution utilize additional redundancy techniques. Recent studies, however, call this assumption into question, showing that SATA drives often exhibit the same replacement rate as SCSI and FC disks [37].

Energy efficiency is an area that many designs have explored in pursuit of cost savings. Some reports state that commonly used power supplies operate at only 65–75% efficiency, representing one of the primary culprits of excess heat production, and contributing to cooling demands that account for up to 60% of data-center energy usage [17]. The development of Massive Arrays of Idle Disks (MAIDs) generated large cost savings by leaving the majority of a system's disks spun down [10]. Further work has expanded on the idea by incorporating strategies such as data migration, the use of drives that can spin at different speeds, and power-aware redundancy techniques [31, 32, 45, 49, 51]. While these systems realize energy savings, they are not designed specifically for archival workloads, instead attempting to provide performance comparable to "full-power" disk arrays at reduced power. Thus, they do not consider approaches that could save even more power at the expense of high performance. For example, some MAID systems, such as those built by Copan Systems [19], use a relatively small number of server-class CPUs and controllers that can control dozens of disks. However, this approach is still relatively power-hungry because the CPU and controllers are always drawing power, reducing energy efficiency. A Copan MAID system in normal use consumes 11 W/TB [19]; as shown in Table 2, this is comparable to the 11–13 W required by a spun-up Pergamum tome with a 1 TB drive. However, it is much higher than the 2–3 W/TB that Pergamum can achieve with 95% of the disks powered off.

Another class of systems relies on media such as tape or optical media rather than hard drives [41, 43]; such systems are typically used for archival or back-up workloads. While the raw media cost may be somewhat lower than that of disk, the cost savings of such media are often offset by the need for additional hardware, *e. g.*, extra drive heads and robotic arms. Additionally, the random access performance of these systems is often quite poor, introducing a number of correlated side effects such as limitations on the system's choice of redundancy schemes. For example, RAIL stores data on optical disks and utilizes RAID 4 redundancy, but only at a very high level: for every five DVD libraries, a sixth library is solely devoted to storing parity [43]. Other systems have used striped tape to increase performance [13]; later systems used extra tapes in the stripe to add parity for reliability [24].

A final class of storage systems is designed for an archival workload, but lacks a specific cost-saving strategy [2, 20, 34, 50]. Like many systems designed for primary tier storage, these systems favor performance over power-efficiency and cost savings. While they may offer fast random access performance, their lack of cost efficiency makes them ill-suited for the long-term preservation of large corpora of data. Other wide-area long-term storage systems, such as SafeStore [26], OceanStore [35] and Glacier [21], can provide data longevity, but do not take energy consumption into account. For example, SafeStore uses multiple remote storage systems to ensure data safety, but does not address the issue of reducing power consumption on the remote servers.

Pergamum also expands upon techniques found in systems spanning various usage models and cost strategies. Many systems have used hierarchical hashing as a means of ensuring file integrity [1, 2, 25, 27, 28, 34, 35]; Pergamum extends this technique by utilizing hash trees of algebraic signatures [39]. Additionally, we extend the use of hierarchical hashing to the power-efficient auditing and consistency checking of inter-device replication. Intra-disk redundancy strategies were first suggested for use in full-power RAID systems to avoid loss of data due to disk failure and simultaneous latent sector errors on a surviving disk [11, 12]. Finally, a number of hybrid

drives that combine flash memory with a hard drive have come to market [40]. However, in such units, the flash is used primarily as a read and write cache, in contrast to Pergamum, which uses flash memory on each Pergamum tome for metadata consistency, and indexing information, allowing Pergamum to reduce disk spin-ups while preserving high levels of functionality.

3 System Components

The design of Pergamum was driven by a workload that exhibits read, write and delete behavior that differs from typical disk-based workloads, providing both challenges and opportunities. The workload is write-heavy, motivated by regulatory compliance and the desire to save any data that *might* be valuable at a later date. Reads, while relatively infrequent, are often part of a query or audit and thus are likely to be temporally related. Deletes are also likely to exhibit a temporal relationship as retention policies often specify a maximum data lifetime. This workload resembles traditional archival storage workloads [34,50], adding deletion for regulatory compliance.

The Pergamum system is structured as a distributed network of independent storage appliances, as shown in Figure 1. Alone, each Pergamum tome acts as an intelligent storage device, utilizing block-level erasure coding to survive media faults and algebraic signatures to verify block integrity. Collectively, the storage appliances provide data reliability through distributed RAID techniques that allow the system to recover from the loss of a device, and inter-disk data integrity by efficiently exchanging hash trees of algebraic signatures. As we will show, this approach is so reliable that disk scrubbing [38] need not be done more frequently than annually. In addition, lost data can be rebuilt with lower peak energy consumption by staggering disk activity; this approach is slower, but reduces peak power consumption.

The next two sections discuss the design and implementation of Pergamum and implementation of these techniques. This section describes an individual Pergamum appliance, or *tome*, including its components, intra-appliance redundancy strategy, interconnection network, and interface. Section 4 then describes how multiple storage appliances work together to provide reliable, distributed, archival storage, including a description of the system's inter-appliance redundancy and consistency checking strategy.

3.1 Pergamum Tomes

A Pergamum tome is a storage appliance made up of four main components: a low-power processor, a commodity hard drive, non-volatile flash memory and an ethernet

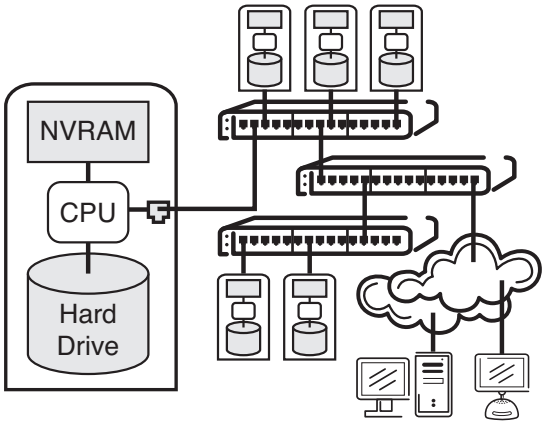


Figure 1: High-level system design of Pergamum. Individual Pergamum tomes, described in Section 3.1 are connected by a commodity network built from off-the-shelf switches.

Component	Power
SATA Hard Drive [47]	7.5 W
ARM-based board (w/ NIC) [4]	3.5 W
NVRAM	< 0.6 W

Table 2: Active power consumption (in watts) of the four primary components that make up a Pergamum tome.

controller. To protect against media errors, erasure coding techniques are used on both the hard drive and flash memory.

Each Pergamum tome is managed by an on-board, low-power CPU; a modern ARM-based single board computer consumes 2–3 W when active (using a 400 MHz CPU) and less than 300 mW when inactive [4]. The processor handles the usual roles required of a network-attached storage device [15,16] such as network communications, request handling, metadata management, and caching. In addition, each Pergamum tome's CPU manages consistency checking and parity operations for the local drive, responds to search requests, and initiates communications with other disks to provide inter-disk reliability. The processor can also be used to handle other operations at the device level, such as virus checking and compression.

Persistent storage is provided through the unit's SATA-class hard drive. The use of commodity hardware offers cost savings over more costly SCSI and FC drives while providing acceptable performance for archival workloads. By using both intra-disk redundancy and distributed redundancy groups, commodity SATA-class drives can provide excellent reliability for long-term archival storage [37].

While a single processor could manage multiple hard drives, Pergamum pairs each processor with a single hard

drive. This is done for performance matching, power savings, and ease of maintenance. As Section 5 details, low-power processors are not fast enough to run even a single disk at full speed, so there is little incentive to control multiple disks with a single CPU. Power savings is another issue: a faster CPU and multi-disk controller would consume more power than multiple individual low-power CPUs (cutting processor voltage in half results in half the clock speed but one fourth the power consumption). Finally, the pairing of a CPU with a single disk and network connection makes it simpler to replace a failed Pergamum tome. If any part of the Pergamum tome fails, the entire Pergamum tome is discarded and replaced, rather than trying to diagnose which part of the Pergamum tome failed to "save" working hard drives. The system then heals itself by rebuilding the data from the failed device elsewhere in the system. By reducing the complexity of routine maintenance, Pergamum reduces ongoing costs.

In addition to a hard drive, each Pergamum tome includes a pool of on-board NVRAM for storing metadata such as the device's index, data signatures and information about pending writes. The purpose of the NVRAM is to provide low-power, persistent storage; operations such as metadata searches and signature requests do not require the unit's drive to be spun up. While the use of flash-type NVRAM provides better persistency and energy-efficiency compared to DRAM, it does raise two issues: reliability and durability. Our system protects the flash memory from erroneous writes and media errors through the use of page-level protection and consistency checking [18], ensuring memory reliability. Flash memory is also limited in that the memory must be written in blocks, and each block may only be rewritten a finite number of times, typically 10^4–10^5 times. However, since the NVRAM primarily holds metadata such as algebraic signatures and index information, flash writes are relatively rare; flash writes coincide with disk writes. Because this typically occurs fewer than 1000 times per year, or 8000 times during the lifetime of a disk, even if the flash memory is totally overwritten each time, such activity will still be below the 10,000 write cycles that flash memory can support. Additionally, while the current implementation uses NAND flash memory, other technologies such as MRAM [44] and phase change RAM [9] could be used as they become available and price-competitive, further reducing or eliminating the rewrite issue.

Finally, each Pergamum tome includes an Ethernet controller and network port, providing a number of important advantages. First, a network connection is a standardized interface that changes very slowly—modern Ethernet-based systems can interoperate with systems that are more than fifteen years old. In contrast, tape-based systems require a unique head unit for each tape format, and each of those devices may require a different interface; supporting legacy tapes could require the preservation of lengthy hardware chains. The use of a network also eliminates the need for robotics hardware (or humans) to load and unload media; such robots might need to be modified for different generations of tape media and must be maintained. Instead, the system can use commodity network interconnects, leaving all media permanently connected and always available for messaging.

3.2 Interconnection Network

Since Pergamum must contain thousands of disks to contain the petabytes of data that long-term archives must hold, its network must scale to such sizes. However, throughput is not a major issue for such a network—a modern tape silo with 6,000 tapes typically has fewer than one hundred tape drives, each of which can read or write at about 50 MB/s, for an aggregate throughput of 5 GB/s. Scaling a gigabit Ethernet network to support comparable bandwidth can be done using a star-type network with commodity switches at the "leaves" of the network and, potentially, higher-performance switches in the core. For example, a system built from 48-port gigabit Ethernet switches could use two switches as hubs for 48 switches, each of which supports 46 disks, with the remaining two connections going to each of the two hubs. This approach would support over 2200 disks at minimal cost; if the central hubs each had a few 10 Gb/s uplinks, a single client could easily achieve bandwidth above 5 GB/s. This structure could then be replicated and interconnected using a more expensive 10 Gb/s switch, allowing reasonable-speed access to any one of tens of thousands of drives, with the vast majority remaining asleep to conserve power.

The interconnection network must allow any disk to connect with any network-connected client. By using a standard Ethernet-based network running IP, Pergamum ensures that *any* disk can communicate with any other disk, allowing the system to both detect newly-connected disks and allowing them to communicate with existing disks to "back up" their own data.

The approach described above is highly scalable, with minimal "startup cost" and low incremental cost for adding additional disks. Further efficiencies could be achieved by pairing the Ethernet cable with a higher-gauge wire capable of distributing the 14–18 W that a spun-up disk and processor combination requires. Alternatively, the system could use disks that can spin at variable speeds as low as 5400 RPM [47], reducing disk power requirements to 7.5 W and overall system power needs to below 11 W, sufficiently low to use standard power-over-Ethernet. Central distribution of power has

several advantages, including lower hardware cost and lower cabling cost. Additionally, distributing power via Ethernet greatly simplifies maintenance—adding a new drive simply requires plugging it into an Ethernet cable. While the disks in the system will work to keep average power load below 5% utilization, a central power distribution system will allow the network switches themselves to guarantee that a particular power load will never be exceeded by restricting power distributed by the switch.

3.3 Pergamum Tome Interface

There are two distinct data views in Pergamum: a file-centric view and a block-centric view. Clients utilize the file-centric view, submitting requests to a Pergamum tome through traditional read and write operations. In contrast, requests from one Pergamum tome to another utilize the block-centric view of data based on redundancy group identifiers and offsets.

Clients access data on a Pergamum tome using a set of simple commands and a connection-oriented request and response protocol. Currently, clients address their commands to a specific device, although future versions of Pergamum will include a self-routing communications mechanism. Internally, files are named by a file identifier that is unique within the scope of a single Pergamum tome. The `new` command allocates an unused file identifier and maps it to a filename supplied by the user. This mapping is used by the `open` command to provide the file's unique identifier to a client. This file id, the device's `read` and `write` commands, and a byte offset are then used by the client to access their data.

Requests between Pergamum tomes primarily utilize a data view based on segment identifiers and block offsets, as opposed to files. There are four main operations that take place between Pergamum tomes. First, external parity update requests provide the a Pergamum tome storing parity with the delta and metadata needed to update its external redundancy data. Second, signature requests are used to confirm data integrity. Third, token passing operations assist in determining which devices to spin up. Finally, there are commands for the deferred (*foster*) write operations discussed in Section 4.3.1.

Management of Pergamum tomes can be done either with a centralized "console" to which each Pergamum tome reports its status, or in a distributed fashion where individual Pergamum tomes report their health via LED. For example, each Pergamum tome could have a small green LED that is on when the appliance is working correctly, and off when it is not. An operator would then replace Pergamum tomes whose light is off; this approach is simple and requires little operator skill. Alternatively, a central console could report "Pergamum tome 53 has failed," triggering a human to replace the failed unit. The Pergamum design permits both approaches; however, we do not discuss the tradeoffs between them in this paper.

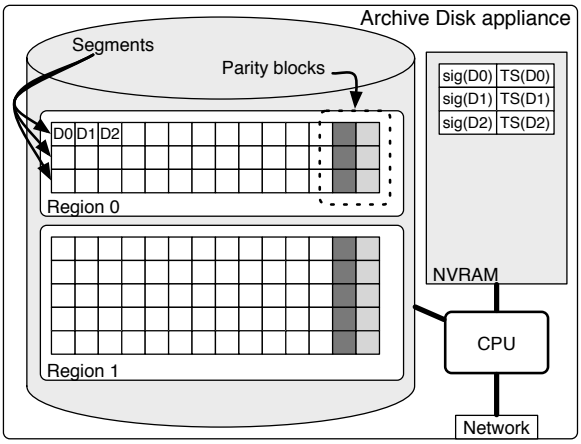


Figure 2: Layout of data on a single Pergamum tome. Data on the disk is divided into blocks and grouped into segments and regions. Data validity is maintained using signatures, and parity blocks are available to rebuild lost or corrupted data.

4 Pergamum Algorithms and Operation

A Pergamum system, deployed as described in Section 3 is highly decentralized, relying upon individual disks to each manage their own behavior and their own data. Each disk is responsible for ensuring the reliability of the data it stores, using both local redundancy information and storage on other nodes.

4.1 Intra-Disk Storage and Redundancy

The basic unit of storage in a Pergamum tome are fixed-size blocks grouped into fixed-size *segments*, as shown in Figure 2. Together, blocks and segments form the basic units of the system's two levels of redundancy encoding: intra-disk and inter-disk. Since the system is designed for archival storage, blocks are relatively large—128 KB–1 MB or larger—reducing the metadata overhead necessary to store and index them. This approach mirrors that of tape-based systems, which typically require data to be stored in large blocks to ensure high efficiency and reasonable performance.

The validity of individual blocks is checked using hashes; if a block's content does not match its hash, it can be identified as incorrect; this approach has been used in other file systems [27, 42]. Disks themselves maintain error-correcting codes, but such codes are insufficiently accurate for long-term archival storage because they have a silent failure rate of about 10^{-14}, a rate sufficiently high

to cause data corruption in large-scale long-term storage. To avoid this problem, each disk appliance stores both a hash value and a timestamp for each block on disk. Assuming a 64-bit hash value and a 32-bit timestamp, a 1 TB disk will require 96 MB of flash memory to maintain this data for 128 KB blocks. Keeping this information in flash memory has several advantages. First, it ensures that block validity information has a different failure mode from the data itself, reducing the likelihood that both data and signature will be corrupted. More importantly, however, it allows the Pergamum tome to access the signatures and timestamps without powering on the disk, enabling Pergamum to conduct inter-disk consistency checks without powering on individual disks.

The hash values used in Pergamum are *algebraic signatures*—hash values that are highly sensitive to small changes in data, but, unlike SHA-1 and RIPEMD, are not cryptographically secure. Algebraic signatures are ideally suited to use in Pergamum because, for many redundancy codes, they exhibit the same relationships that the underlying data does. For example, for simple parity:

$$d_0 \oplus d_1 \cdots \oplus d_{n-1} = p \Longrightarrow$$
$$\text{sig}(d_0) \oplus \text{sig}(d_1) \cdots \oplus \text{sig}(d_{n-1}) = \text{sig}(p) \quad (1)$$

While 64-bit algebraic signatures are sufficiently long to reduce the likelihood of "silent" errors to zero; they are ineffective against malicious intruders, though there are approaches to verifying erasure-coded data using signatures or fingerprints that can be used to defeat such attacks [22, 39].

As Figure 2 illustrates, each segment is protected by one or more parity blocks, providing two important protections to improve data survivability. First, the extra parity data provides protection against latent sector errors [6]. If periodic scrubbing reveals unreadable blocks within a segment, the unreadable data can be rebuilt and written to a new block using only the parity on the local disk. Second, while simple scrubbing merely determines whether the block is readable, the use of algebraic signatures and parity blocks allows a disk to determine whether a particular block has been read back properly, catching errors that the disk drive itself cannot [22, 39] and correcting the error without the need to spin up additional disks.

4.2 Inter-Disk Redundancy

While intra-disk parity guards against latent sector errors, Pergamum can survive the loss of an entire Pergamum tome through the use of inter-tome redundancy encoding. Segments on a single disk are grouped into *regions*, and a *redundancy group* is built from regions of identical sizes on multiple disks. To ensure data survival,

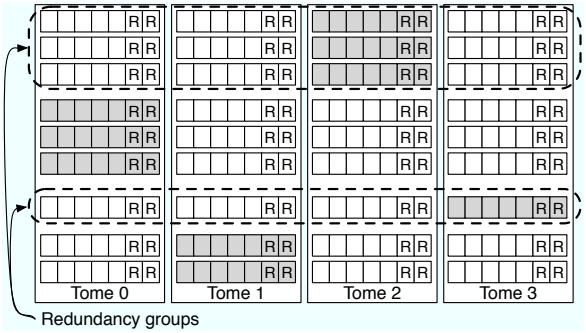


Figure 3: Two levels of redundancy in Pergamum. Individual segments are protected with redundant blocks on the same disk—those labeled with an **R**. Redundancy groups are protected by the shaded segments, which contain erasure correcting codes for the other segments in the redundancy group. Note that segments used for redundancy still contain intra-disk redundant blocks to protect them from latent sector errors.

each redundancy group also includes extra regions on additional disks that contain erasure correction information to allow data to be rebuilt if any disks fail. These *redundancy regions* are stored in the same way as data regions: they have parity blocks to guard against individual block failure and the disk appliances that host them store their algebraic signatures in NVRAM.

The naïve approach to verifying the consistency of a redundancy group would require spinning up all the disks in the group, either simultaneously or in sequence, and verifying that the data in the segments that make up the regions in the group is consistent. Pergamum dramatically reduces this overhead in two ways. First, the algebraic signatures stored in NVRAM can be exchanged between disks in a redundancy group and verified for consistency as described in Section 4.1. Since the signatures are retrieved from NVRAM, the disk need not be spun up during this process as long as changes to on-disk data are reflected in NVRAM. If inconsistencies are found, the timestamps may be used to decide on the appropriate fix. For example, if a set of segments is inconsistent and a data segment is "newer" than the newest parity segment, the problem is likely that the write was not applied properly; depending on how writes have been applied and whether the "old" data is available, the parity may be fixed without powering up the whole set of segments.

While this approach only requires that signatures, rather than data, be transmitted, it is still very inefficient, requiring the transmission of nearly 100 MB of signatures for each disk to verify a redundancy group's consistency. To further reduce the amount of data and computation that must be done, Pergamum uses hash trees [29] built from algebraic signatures, as shown in Figure 4. Using signatures of blocks as d_i in Equation 1 shows that

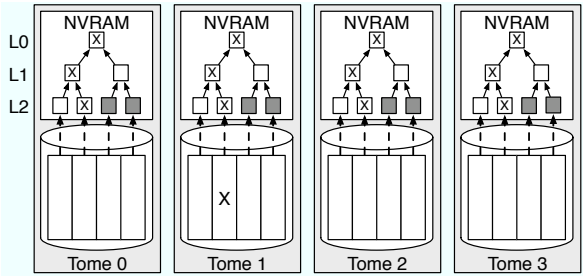


Figure 4: Trees of algebraic signatures. Tomes in a redundancy group exchange the roots of their trees to verify consistency; in this diagram, the signatures marked with an X are inconsistent. The roots (L0) are exchanged; since they do not match, the nodes recurse down the tree to L1 and then L2 to find the source of the inconsistency. "Children" of consistent signatures (signatures shaded in gray at L2) are not fetched, saving transmission and processing time. The inconsistent block on tome 1 is found by checking the intra-segment signatures on each block; only those on tome 1 were inconsistent. Note that only tome 1's disk need be spun up to identify and correct the error if it is localized.

signatures of sets of signatures follow the same relationships as the underlying data; this property is maintained all the way up to the root of the tree. Thus, the signatures at the roots of each disk's hash tree for the region should yield a valid erasure code word when combined together. If they do not, some block in the redundancy group is invalid, and the disks recurse down the hash tree to find the bad block, exchanging the contents at each level to narrow the location of the "bad" block. This approach requires $O(k)$ computation and communication when the group is correct—the normal case—and $O(k \log n)$ computation and communication to find an error in a redundancy group with a total of n blocks across k disks. Since redundancy groups are not large ($k \leq 50$, typically), high-level redundancy group verifications can be done quickly and efficiently.

4.3 Disk Power Management

Reducing power consumption is a key goal of Pergamum; since spinning disks are by far the largest consumer of power in a disk appliance, keeping the disk powered off ("spun down") dramatically reduces power consumption. In contrast to earlier systems that aim to keep 75% of the disks inactive [19], Pergamum tries to keep 95% or more of the disks inactive all of the time, reducing disk power consumption by a factor of five or more over existing MAID approaches. This goal is achieved with several strategies: sequentially activating disks to update redundancy information on writes, low-frequency scrubbing, and sequentially rebuilding regions on failed disks.

To guard against too many disks being spun up at once, Pergamum uses *spin-up tokens*, which are passed from one node to another to allow spin-up. If multiple nodes require a token simultaneously, the node currently holding the token (which may or may not be spun up at the time) calculate need based on factors such as a unit's oldest pending request, the types of requests it has pending, the number of pending requests and the last time the disk was spun up.

4.3.1 Reading and Writing Data

When a client requests a data read, the device from which data is to be read is spun up. This process takes a few seconds, after which data can be read at full speed. While a Pergamum tome is somewhat slower than a high-power network-attached disk, its performance, discussed in Section 5, is sufficient for archival storage retrieval. Moreover, since the data is stored on a disk rather than a tape, random access performance is significantly better than that of a tape-based system.

As with reads, archive writes require a spun-up disk. Pergamum clients choose the disks to which they write data; Pergamum does not impose a choice on users. This is done because some clients may want to group particular data on specific disks: for example, a company might choose to archive email for an individual user on one drive. On the other hand, a storage client may query Pergamum nodes to identify spun-up nodes, allowing it to select a disk that is already spun up.

Since writes require the eventual update of distributed data, they are more involved than reads. First, the target disk is spun up if it is not already active. Next, data is written to blocks on the local disk. However, existing data blocks are not overwritten in place; instead, data is written to a new data block, allowing the Pergamum tome to calculate "deltas" based on the old and new block. These deltas are then sent to the Pergamum tomes storing the redundancy regions for the old block's segment. On the local device, the segment mapping is updated to replace the old block with the new block. It is important to note, however, that the old block is retained until it has been confirmed that all external parity has been updated.

On the Pergamum tomes storing the redundancy information, the deltas arrive as a parity update request. Since the redundancy update destination knows how the erasure correcting code is calculated, it can use the delta from the data target disk to update its own redundancy information; it does not need both the old and new data block, only the delta. Because the delta may be different for different parity disks, however, the Pergamum tome that received the original write request must keep both old and new data until all of the parity segments have

been updated. However, doing updates this way ensures that a write requires no more than two disks to be active at any time; while the total energy to write the data is unchanged—a write to an (m,n) redundancy group must still update $n-m+1$ disks—the peak energy is dramatically reduced from $n-m+1$ disks active to 2 disks active, resulting in an improvement for any code that can correct more than one erasure.

One problem with allowing writes directed to a specific Pergamum tome is that the disk may not be spun up when the write is issued. While the destination disk may be activated, an alternate approach is to write the data to *any* currently active disk and later copy the data to the "correct" destination. This approach is called *surrogate writing*, and is used in Pergamum to avoid spinning disks up too frequently. Instead, writes are directed to an already-active disk, and the Pergamum tome to which data will eventually be sent is also notified. The data can the be transferred to the correct destination lazily.

4.3.2 Scrubbing and Recovering Data

To ensure reliability, disks in Pergamum are occasionally scrubbed: every block on the disk is read and checked for agreement with the signature stored in NVRAM. This procedure is relatively time-consuming; even at 10 MB/s, a 1 TB disk requires more than a day to check. However, Pergamum tome's use of on-disk redundancy to guard the data in a segment, described in Section 4.1, greatly reduces the danger of data loss from latent sector errors, so the system can reduce the frequency with which it performs full-disk scrubs. Instead, a Pergamum tome performs a "limited scrub" each time it is spun up, either during idle periods or immediately before the disk is spun down. This limited scrub checks a few hundred randomly-chosen locations on the disk for correctness and examines the drive's SMART status [5], ensuring that the disk is basically operating correctly. If the drive passes this check, the major concern is total drive failure, either during operation or during spin-up, as Section 5.2 describes.

Complete drive failures are handled by rebuilding the data on the lost drive in a new location. However, since fewer than 5% of the disks in Pergamum may be on at any given time and redundancy groups that may contain data and parity on 15–40 disks for maximal storage efficiency, it is impractical to spin up all of the disks in a redundancy group to rebuild it. Instead, Pergamum uses techniques similar to those used in writing data to recover data lost when a disk fails. The rebuilding algorithm begins by choosing a new location for the data that has been lost; this may be on an existing disk (as long as it is not already part of the redundancy region), or it may be on a newly-added disk. Pergamum then spins up the disks in the redundancy region one by one, with each disk sending its data to the node on which data is being rebuilt. The node doing the rebuilding folds the incoming data into the data already written using the redundancy algorithm; thus, it must write each location in the region m times and read it $m-1$ times (the first "read" would result in all zeros, and is skipped).

5 Experimental Evaluation

Our experiments with the current implementation of Pergamum were designed to measure several things. First, we wanted to evaluate the cost of our system in order to ensure that our solution was economically feasible. Second, we wanted to confirm that Pergamum can provide long-term reliability through a strategy of multiple levels of parity and consistency checking using algebraic signatures. Finally, we wanted to measure the performance of our implementation to show that Pergamum is suitable for archival workloads and to identify potential bottlenecks.

The remainder of this section proceeds as follows. First, we first present an analytical evaluation of the system's cost. Then, using a series of simulations, we examine the system's long-term reliability. Finally, we present the results of our performance tests with the current implementation of Pergamum.

5.1 Cost

An archival system's cost can be broken down into two primary areas: static (initial costs) and operational. The first figure describes the cost to acquire the system, and the second figure quantifies the cost to run the system. Examining both costs together is important because low static costs can be overshadowed by the total cost of operating and maintaining a system over its lifetime.

We do not consider personnel costs in any of the systems we describe; we assume that all of the systems are sufficiently well automated that human maintenance costs are relatively low. However, this assumption is somewhat optimistic, especially for large tape-based systems that use complex hardware that may require repair. In contrast, Pergamum is built from simple, disposable components—a failed Pergamum tome or network switch may simply be thrown out rather than repaired, reducing the time and personnel effort required to maintain the system.

Static costs reflect the expenses associated with acquiring an archival storage solution, and can be calculated by totaling a number of individual costs. One is the system expense, which totals the base hardware and software costs of a storage system with a given capacity for storage media. This cost is paid at least once per

storage system, regardless of how much storage is actually required. Media cost, in dollars per terabyte, is a second expense. Large archival storage systems may require several "base" systems; for example, an archival system that uses tape silos and robots might require one silo per 6,000 tape cartridges, even if the silo will not be filled initially.

Operational costs reflect those costs incurred by day to day operation of an archival storage system. This cost can be measured using a dollars per operational period figure, normalized to the amount of storage being managed. Some of the primary contributors to a system's total operational expenses include power, cooling, floor space and management. As described above, we omit management cost, both because we assume it will be similar for different storage technologies, and because it is extremely difficult to quantify. We also omit the cost of floor space since it is highly variable depending on the location of the data center. However, an important, but often omitted, aspect of operational costs includes the expenses related to reliability: expected replacement costs for failed media and the operational cost associated with parity operations. This cost, along with power and cooling, forms the basis of our comparison of operational costs.

The static and operational costs must include the cost for any redundant hardware or storage. However, since existing solutions vary in their reliability, even within a particular technology, we have not attempted to quantify the interplay between capacity and reliability. Instead, we assume that a system that requires mirroring simply costs twice as much to purchase and run per byte as a non-redundant system. In this respect, Pergamum is very low cost: the storage overhead for a system with segments using 62 data and 2 parity blocks and redundancy groups with 13 data disks and 3 parity disks is $\frac{64}{62} \times \frac{16}{13} - 1 = 0.27$ times usable data capacity. In such a system, 1 TB of raw storage can hold 787 GB of user data.

All of these factors—static cost, operational cost, and redundancy overhead—are summarized in Table 3. Static costs are approximations based on publically available hardware prices. For operational costs, We have used a constant rate of $0.20/kWh for electricity to cover both the direct cost of power and the cost of cooling. Table 3 shows the costs for a 10 PB archive for each technology, including sufficient base systems to reach this capacity. While the costs reflected in the table are approximate, they are useful for comparative purposes. Also, we note that some systems have ranges for redundancy overhead because they can be configured in several ways to ensure sufficient reliability; we chose the least expensive reliability option for each technology. For example, the EMC Centera [20] can be used with mirroring; doing so might increase reliability, but will certainly increase total cost.

The results summarized in Table 3 illustrate a number of cost-related archival storage issues. First, as shown by PARAID, even energy-efficient, non-archival systems are too expensive for archival scenarios. Second, media with low storage densities can become expensive very quickly because they require a large amount of hardware to manage the high numbers of media. For example, RAIL uses UDO2 optical media that only offers 60 GB per disk and thus the system requires numerous cabinets and drives to handle the volume of media. Using off-the-shelf dual-layer DVDs, with capacity under 10 GB per disk, would reduce the media cost, but would increase the hardware cost by a factor of six because of the added media; such an approach would require 100 DVDs per terabyte, making the cost prohibitive. Third, the Copan and Centera demonstrate two different strategies for cost effective storage: lower initial costs versus lower runtime costs. Finally, we see that Pergamum is competitive in cost to Sun's StorageTek SL8500 system while providing functionality, such as inter-archive redundancy, that tape-based systems are unable to provide.

An understanding of the costs associated with reliability is important because it assists in matching the data to be protected with an economically efficient reliability strategy. Unfortunately, because it is largely dependent on the data itself, the economic impact of lost data is difficult to calculate. Moreover, many of the costs resulting from data loss are, at best, difficult to quantify. For example, the cost to replace data can vary from zero (don't replace it) to nearly priceless (how much is bank account data worth?). Another factor, opportunity costs, expresses the cost of lost time; every hour spent dealing with data loss is an hour that is not spent doing something else. In a professional setting, data loss may also involve mandatory disclosures that could introduce costs associated with bad publicity and fines. While we do not quantify these costs, we note that long-term archive reliability is a serious issue [7].

5.2 Reliability

There are many tradeoffs that influence the reliability of an archival storage system. Factors such as stripe size, both on an individual disk and between disks, disk failure rate, disk rebuild time and the expected rate of latent sector errors must be considered when building a long-term archival system. Our analysis considers these factors along with strict power-management constraints to compute the expected mean time to data loss (MTTDL) of a deployed Pergamum system.

Table 4 shows the parameters used in our analysis. In the absence of an archival workload for our reliability

System	Media	Static cost	Oper. cost	Redundancy
Sun StorageTek SL8500	T10000 tape	$4,250	$60	None
EMC Centera	SATA HD	$6,600	$1,800	parity
PARAID	SCSI HD	$37,800	$1,200	RAID
Copan Revolution	SATA HD	$19,000	$250	RAID-5
RAIL	UDO2	$57,000	$225	RAID-4 (5+1)
Pergamum	SATA HD	$4,700	$50	2-level

Table 3: Comparison of system and operational costs for 10 PB of storage. All costs are in thousands of dollars and reflect common configurations. Operational costs were calculated assuming energy costs of $0.20/kWh (including cooling costs).

Parameter	Value
Disk Fail Rate (λ_D)	$1/100000$ hours
Disk Repair Rate (μ_D)	$1/100$ hours
Latent Sector Fault Rate (λ_S)	1/13245 hours
Scrub Rate (μ_S)	$1/8640$ hours

Table 4: Simulator and model parameters.

analysis, we assume that each active device transfers a constant 2 MB/s, on average. Given a byte error rate of 1×10^{-14}, the on-disk sector error rate is approximately $1/13245$ hours. Due to the incremental nature of our rebuild algorithm, we approximate the time to rebuild a single device in our system to be roughly 100 hours, or 3 MB/s. Finally each disk in the system fails at a rate of $1/100000$ hours and is subject to a full scrub every year or 8640 hours. We consider these estimates to be liberal and provide a near-worst-case MTTDL of our system.

In order to determine the reliability of our system, we developed a discrete event simulator in Python using the SimPy module. There are four core events in our simulator: `DiskFail`, `DiskRebuild`, `SectorFail` and `Scrub`. Values for disk failure time, sector failure time and disk scrub are all drawn from an exponential distribution, while disk rebuild takes place in simulation time at 3 MB/s. We model the effects of disk spin-up by subtracting 10 hours from the life of a disk every time it is spun up [38]; this may well be pessimistic, resulting in an MTTDL that is shorter than in a real system. Each iteration of the simulator runs until a data loss event is reached and the current time is recorded. Although we found that around 100 iterations is sufficient, we calculate the MTTDL of a single configuration by running 1000 iterations with that configuration.

We also use Markov models to compute the reliability of single, double and triple disk fault tolerant codes. These models only capture disk failure and rebuild, and thus serve two purposes. First, the models give us a straightforward way to verify the behavior of the simulator. Most importantly, the MTTDL computed from each model serves as an approximation to a system that has

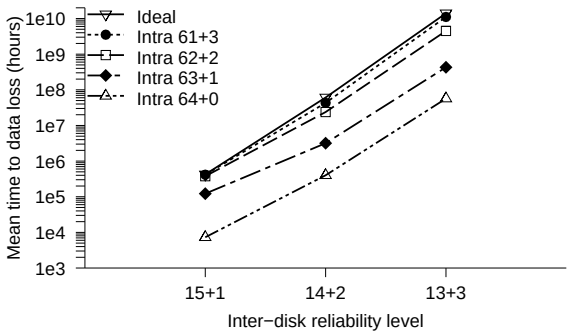


Figure 5: Mean time to data loss in hours for a single 16 disk group. 61+3 intra-disk parity is nearly equivalent to the "ideal" system, in which latent sector errors never occur. Note that MTTDL of 10^{10} hours for 16 disks corresponds to a 1000 year MTTDL for a 10 PB Pergamum system.

the ability to handle any number of latent sector errors.

Recent work has shown that latent sector errors make a non-trivial contribution to system reliability [6]. We thus modeled data loss in our system for configurations with 1, 2, and 3 parity segments per redundancy group under several different assumptions: levels of intra-disk parity protection ranging from 0–3 parity blocks per segment, and an "ideal" analytical model which assumed *no* latent sector errors occurred and considered only whole-disk failures. The results of our modeling using a scrub rate of once per year for each disk, shown in Figure 5, indicate that latent sector errors do indeed cause data loss if nothing is done to guard against them. The distance between the top curve ("ideal" MTTDL without latent sector errors) and bottom curve (no intra-disk parity) is approximately two orders of magnitude, showing that Pergamum must guard against data loss from latent sector errors. However, by using intra-disk erasure coding, the effect of latent sectors on MTTDL is nearly eliminated. In essence, we are trading disk space for a longer scrub interval, saving power in the process. Figure 5 shows that a configuration of 3 intra-disk parity blocks per 64 block segment provides nearly two or-

ders of magnitude longer MTTDL than no protection at all, approaching the "ideal" situation where latent sector errors never exist. With the exception of the configurations with 3 inter-disk parity elements and the "ideal" case, all of the MTTDL values in the graph are based on 1000 iterations of the simulator; we were only able to capture tens of MTTDL numbers for the configurations involving 3 inter-disk parity elements and 1 or 2 intra-parity elements, and the simulation for 3 inter-disk parity and 3 intra-disk parity elements took a great deal of time to run and only resulted in a few data points. This lack of data is due to the extremely high reliability of these configurations—the simulator modeled many failures, but so few caused data loss that the simulation ran very slowly. This behavior is precisely what we want from an archival storage system: it can gracefully handle many failure events without losing data. Even though we captured fewer data points for the triple inter-parity configuration, we believe the reported MTTDL is a reasonable approximation. As we see in the graph, the use of 3 intra-disk parity elements is close to the "ideal" situation across all inter-parity configurations.

Our simulation and modeling show that a configuration of 3 inter-disk parity segments per 16-disk reliability group and 3 intra-disk parity blocks per segment will result in an MTTDL of approximately 10^{10} hours. If each disk has a raw capacity of 1 TB, a Pergamum system capable of storing 10 PB of user data will require about 800 such groups, resulting an MTTDL of 1.25×10^7 hours, or about 1,400 years. Should this MTTDL for an entire archive be too low, we would recommend using more inter-disk parity—3 parity blocks per 64 block segment can correct most of the latent sector errors.

5.3 Performance

The current Pergamum prototype system consists of approximately 1,400 lines of Python 2.5 code, with an additional 300 lines of C code that were used to implement performance-sensitive operations such as data encoding and low-level disk operations. Our implementation includes the core system functionality, including internal redundancy, external redundancy, and a client interface that allows for basic I/O interactions. In its current state however, the implementation relies upon statically assigned redundancy groups and it does not include scrubbing or consistency checking.

For testing, all systems were located on the same gigabit Ethernet switch with little outside contention for computing or network resources. Communication between the Pergamum tome and the client used standard TCP/IP sockets in Python. For maximum compatibility, we utilized an MTU size of 1500 B.

Each Pergamum tome was equipped with an ARM 9 CPU running at 400 MHz, 128 MB of DDR2 SDRAM and Linux version 2.6.12.6. The client was equipped with an Intel Core Duo processor running at 2 GHz, 2 GB of DDR2 SDRAM and OS X version 10.4.10. The primary storage on each Pergamum tome was provided by a 7200 RPM SATA drive formatted with XFS. For read and write performance experiments, we utilized block sizes of 1 MB and 64 blocks per segment. Persistent metadata storage utilized a 1 GB USB flash drive and Berkeley DB version 4.4. The workload consisted of randomly generated files, all several megabytes in size.

Test	Client	Server
Raw Data Transfer	20.02	20.96
Raw Data Write	9.33	9.98
Unsafe Pergamum Write	4.74	4.74
XOR Parity Pergamum Write	4.72	3.25
Reed Solomon Pergamum Write	4.25	1.67
Fully Protected Pergamum Write	3.66	0.75
Pergamum Read	5.77	5.78

Table 5: Read and write performance for a single Pergamum tome to client connection. XOR parity writes used 63 data blocks to one parity block segments. Reed Solomon writes used 62 data blocks to two parity block segments. Fully protected writes utilize two level of Reed Solomon encoding and the server throughput reflects time to fully encode and commit internal and external parity updates.

5.3.1 Read and Write Throughput

Our first experiment with the Pergamum implementation was an evaluation of the device's raw data transfer performance. As Table 5 shows, the maximum throughput of a single TCP/IP stream to a Pergamum tome is 20 MB/s at the device. Further tests showed that, the device could copy data from a network buffer to an on-disk file at about 10 MB/s. Together, these values serve as an upper limit for the write performance that could be expected from a single client connection over TCP/IP.

Write throughput using the Pergamum software layer was tested at varying levels of write safety. The first write test was conducted with no internal or external parity updates. As shown in Table 5, writes without data protection ran at 4.74 MB/s. While no redundancy encoding was performed in the unsafe write, the system did incur the overhead of updating segment metadata and dividing the incoming data into fixed-size blocks.

Testing with internal parity updates enabled was performed using both simple XOR-based parity and more advanced Reed Solomon encoding. In these tests, the client-side and server-side throughput differ, as Pergamum utilizes parity logging during writes. Thus, while

the client views throughput as the time taken to simply ingest the data, the Pergamum tome's throughput includes the time to ingest the data and update the redundancy information. The first test utilized simple XOR-based parity in a 63+1 (63 data blocks and 1 parity block) configuration. This arrangement achieved a client-side write throughput of 4.72 MB/s and a Pergamum tome-side throughput of 3.25 MB/s. As Table 5 shows, using Reed Solomon in a 62+2 configuration results in similar client side throughput, 4.25 MB/s. However, the extra processing and parity block updates results in a server throughput of 1.67 MB/s.

The final write test, fully protected Pergamum tome writes, utilizes both inter- and intra-disk redundancy. Internal parity utilized Reed Solomon encoding in a 62+2 configuration. External redundancy utilized Reed Solomon with 3 data regions to 2 parity regions. In this configuration, client throughput is reduced to 3.66 MB/s as the CPU is taxed with both internal and external parity calculations. This is evident in the server throughput which is reduced to 0.75 MB/s. However, this does reflect the time required to update both internal and external parity and thus reflects the rate at which a single Pergamum tome can protect data with full internal and external parity.

Profile data obtained from the test runs indicates the system is CPU-bound. The performance penalty for the Pergamum tome writes appears to be based largely on two factors. First, as shown in the difference between a raw write and an unsafe Pergamum tome write in Table 5, Python's buffer management imposes a performance penalty, an issue that could be remedied with an optimized, native implementation. Second, as seen in the difference between the XOR Pergamum tome write and the Reed Solomon write, data encoding imposes a significant penalty for lower power processors. This is further evident by the results of our read throughput tests. Since a read operation to the Pergamum tome involves less buffer management and parity operations, throughput is correspondingly faster. We were able to achieve sustained read rates of 5.78 MB/s.

While the performance numbers in Table 5 would be inadequate for most high-performance workloads, even our current, prototype implementation of Pergamum is capable of supporting archival workloads. For example, 1000 Pergamum tomes and a spin-up rate of only 5% can provide a system-level ingestion throughput in excess of 175 MB/s, ingesting a terabyte in 90 minutes and fully protecting it in 8 hours. At this rate such an archive built from 1 TB disks could be filled in a year.

Encode Operations	ARM9	Core Duo
XOR parity	20.02	201.41
Reed Solomon; 5 data, 2 parity	3.13	33.68
Data signature (64-bit)	57.44	533.33

Table 6: Throughput, in MB/sec, to encode 50 MB of data using the Pergamum tome's 400 MHz ARM9 board drawing 2-3 W and a desktop class 2 GHz Intel Core Duo drawing 31 W.

5.3.2 Data Encoding

One of the primary functions of each Pergamum tome's processor is data encoding for redundancy and signature generation. Thus, we wanted to confirm that the low-power CPUs used by Pergamum to save energy are actually capable of meeting the encoding demands of archival workloads.

In our first data encoding test, we measured the throughput of the XOR operation by updating parity for 50 MB of data. We were able to achieve an average encoding rate of 20.79 MB/s on the tome's CPU. For reference, a desktop class processor using the same library was able to encode data at 201.41 MB/s. However, this performance increase comes at the cost of power consumption; the Intel Core Duo processor consumes 31 W compared to the tome's ARM-based processor which consumes roughly 2.5 W for the entire board.

A similar result was achieved when updating parity for 50 MB of data protected by a 5+2 Reed Solomon configuration. As Table 6 summarizes, the processor on the Pergamum tome was able to encode the new parity blocks at a rate of 3.13 MB/s. For reference, the desktop processor could encode at average rate of 33.68 MB/s. Again, we notice an order of magnitude throughput increase at the cost of over an order of magnitude power consumption increase.

Our final encoding experiment involved the generation of data signatures. Our current implementation of Pergamum generates data signatures using $GF(2^{32})$ arithmetic in an optimized C-based library. Generating 64 bit signatures over 32 bit symbols, we achieved an average signature generation throughput of 57.44 MB/s. For reference, the same library on the desktop-class client achieved a rate of 533.33 MB/s.

Our results indicate that the low-power processor on the Pergamum tome is capable of encoding data at a rate comparable to its power consumption. Additionally, we believe that is capable of adequately encoding data for an archival system's write-once, read maybe usage model. While our current performance numbers are reasonable, our experience in designing and implementing the Pergamum prototype has shown that low-power processors greatly benefit from carefully optimized code.

Our early implementations provided more than adequate performance on a desktop class computer but were somewhat slow on the Pergamum tome's low-power CPU.

6 Future Work

While Pergamum demonstrates some of the features needed in an archival storage system, work remains to turn it into a fully effective, evolving, long-term storage system. In addition to the engineering tasks associated with optimizing the Pergamum implementation for low-power CPUs, there are a number of important research areas to examine.

Storage management in Pergamum, and archival storage in general, is an open area with a number of interesting problems. Management strategies play a large part in cost efficiency; many believe that management costs eclipse hardware costs [3]. As a long-term data repository, the effectiveness of archival storage is increased as management overhead is decreased and, ideally, automated; the easier it is to store long-term data, the more ubiquitous it will become. Thus, Pergamum must address how extensibility can be handled in an automatic way, without sacrificing its distributed nature, or the independence of its Pergamum tomes. This overall question includes a number of facets. How are redundancy groups populated? How does the system know if a Pergamum tome is nonfunctional as opposed to temporarily off-line? How can the system's capacity be expanded while still providing adequate reliability?

In our current implementation, users interact with Pergamum by submitting requests to specific Pergamum tomes using a connection-oriented protocol. In future versions, the use of a simple, standardized `put` and `get` style protocol, such as that provided by HTTP, could allow storage to be more evolvable and permit the use of standard tools for storing and retrieving information. Further, techniques such as distributed searching that take into account data movement and migration could greatly simplify how users interact with the system.

While the trade-off between redundancy and storage usage is well acknowledged, there is still work to be done in understanding the interplay of redundancy, storage overhead and power consumption. We have chosen a relatively small set of points in this space; future work could explore this space more completely. This could include an examination of which redundancy codes are best suited to the unique demands and usage model of archival storage.

Long-term storage systems must assume that no single device will serve as the storage appliance for the data's entire lifetime. Thus, data migration in a secure and power-efficient manner is another requirement for Pergamum, and is a critical area for research. This research direction also has implications for reliability; a policy of device refreshment could be an integral part of a long-term reliability strategy.

The optimality of the choice of one CPU and network connection per disk is also an open question; our choice is based on both quantitative and qualitative factors, but other arrangements are certainly possible. Additionally, it has always been assumed that client machines would include modern desktop level CPUs that could be leveraged for pre-processing. Similarly, determining the best network to use to connect thousands of (mostly idle) devices is an interesting problem to consider.

7 Conclusions

We have developed Pergamum, a system designed to provide reliable, cost-effective archival storage using low-power, network-attached disk appliances. Reliability is provided through two levels of redundancy encoding: intra-disk redundancy allows an individual device to automatically rebuild data in the event of small-scale data corruption, while inter-disk redundancy provides protection from the loss of an entire device. Fixed costs are kept low through the use of a standardized network interface, and commodity hardware such as SATA drives; since each Pergamum tome is essentially "disposable", a system operator can simply throw away faulty nodes. Operational costs are controlled by utilizing ultra-low-power CPUs, power-managed disks and new techniques such as local NVRAM for caching metadata and redundancy information to avoid disk spin-ups, intra-disk redundancy, staggered data rebuilding, and hash trees of algebraic signatures for distributed consistency checking. Finally, Pergamum's performance is acceptable for archival storage: the use of many low-power CPUs instead of a few server-class CPUs results in disks that can transfer data at 3–5 MB/s, with faster performance possible through the use of optimized code.

At 2–3 W/TB and under $0.50/GB for a full system, Pergamum is far cheaper and more reliable than existing MAID systems, though the techniques we have developed may be applied to more conventional MAID designs as well. Moreover, a Pergamum system is comparable in cost and energy consumption to a large-scale tape archive, while providing much higher reliability, faster random access performance and better manageability. The combination of low power usage, low hardware cost, very high reliability, simpler management, and excellent long-term upgradability make Pergamum a strong choice for storage in long-term data archives.

Acknowledgments

We would like to thank our colleagues in the Storage Systems Research Center (SSRC) who provided valuable feedback on the ideas in this paper, helping us to refine them. We would also like to thank our shepherd Doug Terry and the anonymous reviewers for their insightful comments that helped us improve the paper.

This research was supported by the Petascale Data Storage Institute under Department of Energy award DE-FC02-06ER25768, and by the industrial sponsors of the SSRC, including Los Alamos National Lab, Livermore National Lab, Sandia National Lab, Agami Systems, Data Domain, Digisense, Hewlett-Packard Laboratories, IBM Research, LSI Logic, Network Appliance, Seagate, Symantec, and Yahoo!.

References

[1] ADYA, A., BOLOSKY, W. J., CASTRO, M., CHAIKEN, R., CERMAK, G., DOUCEUR, J. R., HOWELL, J., LORCH, J. R., THEIMER, M., AND WATTENHOFER, R. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)* (Boston, MA, Dec. 2002), USENIX.

[2] AGARWALA, S., PAULY, A., RAMACHANDRAN, U., AND SCHWAN, K. e-SAFE: An extensible, secure and fault tolerant storage system. In *Proceedings of the First International Conference on Self-Adaptive and Self-Organizing Systems (SASO 2007)* (2007), pp. 257–268.

[3] ANDERSON, E., HOBBS, M., KEETON, K., SPENCE, S., UYSAL, M., AND VEITCH, A. Hippodrome: running circles around storage administration. In *Proceedings of the 2002 Conference on File and Storage Technologies (FAST)* (Monterey, CA, Jan. 2002).

[4] ARCOM, INC. `http://www.arcom.com/`, Aug. 2007.

[5] ATA SMART feature set commands. Small Form Factors Committee SFF-8035. http://www.t13.org.

[6] BAIRAVASUNDARAM, L. N., GOODSON, G. R., PASUPATHY, S., AND SCHINDLER, J. An analysis of latent sector errors in disk drives. In *Proceedings of the 2007 SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (June 2007).

[7] BAKER, M., SHAH, M., ROSENTHAL, D. S. H., ROUSSOPOULOS, M., MANIATIS, P., GIULI, T., AND BUNGALE, P. A fresh look at the reliability of long-term digital storage. In *Proceedings of EuroSys 2006* (Apr. 2006), pp. 221–234.

[8] CAO, P., LIN, S. B., VENKATARAMAN, S., AND WILKES, J. The TickerTAIP parallel RAID architecture. *ACM Transactions on Computer Systems 12*, 3 (1994), 236–269.

[9] CHEN, Y. C., RETTNER, C. T., RAOUX, S., BURR, G. W., CHEN, S. H., SHELBY, R. M., SALINGA, M., RISK, W. P., HAPP, T. D., MCCLELLAND, G. M., BREITWISCH, M., SCHROTT, A., PHILIPP, J. B., LEE, M. H., CHEEK, R., NIRSCHL, T., LAMOREY, M., CHEN, C. F., JOSEPH, E., ZAIDI, S., YEE, B., LUNG, H. L., BERGMANN, R., AND LAM, C. Ultra-thin phase-change bridge memory device using GeSb. In *International Electron Devices Meeting (IEDM '06)* (Dec. 2006), pp. 1–4.

[10] COLARELLI, D., AND GRUNWALD, D. Massive arrays of idle disks for storage archives. In *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing (SC '02)* (Nov. 2002).

[11] DHOLAKIA, A., ELEFTHERIOU, E., HU, X.-Y., ILIADIS, I., MENON, J., AND RAO, K. Analysis of a new intra-disk redundancy scheme for high-reliability RAID storage systems in the presence of unrecoverable errors. In *Proceedings of the 2006 SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (2006), pp. 373–374.

[12] DHOLAKIA, A., ELEFTHERIOU, E., HU, X.-Y., ILIADIS, I., MENON, J., AND RAO, K. Analysis of a new intra-disk redundancy scheme for high-reliability RAID storage systems in the presence of unrecoverable errors. Tech. Rep. RZ 3652, IBM Research, Mar. 2006.

[13] DRAPEAU, A. L., AND KATZ, R. H. Striped tape arrays. Tech. Rep. CSD-93-730, University of California, Berkeley, 1993.

[14] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. The Google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)* (Bolton Landing, NY, Oct. 2003), ACM.

[15] GIBSON, G. A., NAGLE, D. F., AMIRI, K., BUTLER, J., CHANG, F. W., GOBIOFF, H., HARDIN, C., RIEDEL, E., ROCHBERG, D., AND ZELENKA, J. A cost-effective, high-bandwidth storage architecture. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (San Jose, CA, Oct. 1998), pp. 92–103.

[16] GIBSON, G. A., AND VAN METER, R. Network attached storage architecture. *Communications of the ACM 43*, 11 (2000), 37–45.

[17] GREEN GRID CONSORTIUM. The green grid opportunity, decreasing datacenter and other IT energy usage patterns. http://www.thegreengrid.org, Feb 2007.

[18] GREENAN, K. M., AND MILLER, E. L. Reliability mechanisms for file systems using non-volatile memory as a metadata store. In *6th ACM & IEEE Conference on Embedded Software (EMSOFT '06)* (Seoul, Korea, Oct. 2006), ACM.

[19] GUHA, A. Solving the energy crisis in the data center using COPAN Systems' enhanced MAID storage platform. Copan Systems white paper, Dec. 2006.

[20] GUNAWI, H. S., AGRAWAL, N., ARPACI-DUSSEAU, A. C., ARPACI-DUSSEAU, R. H., AND SCHINDLER, J. Deconstructing commodity storage clusters. In *Proceedings of the 32nd Int'l Symposium on Computer Architecture* (June 2005), pp. 60–71.

[21] HAEBERLEN, A., MISLOVE, A., AND DRUSCHEL, P. Glacier: Highly durable, decentralized storage despite massive correlated failures. In *Proceedings of the 2nd Symposium on Networked Systems Design and Implementation (NSDI)* (Boston, MA, May 2005), USENIX.

[22] HENDRICKS, J., GANGER, G. R., AND REITER, M. K. Verifying distributed erasure-coded data. In *Proceedings of the 26th ACM Symposium on Principles of Distributed Computing (PODC 2007)* (Aug. 2007).

[23] Health Information Portability and Accountability Act, Oct. 1996.

[24] HUGHES, J., MILLIGAN, C., AND DEBIEZ, J. High performance RAIT. In *Proceedings of the 19th IEEE Symposium on Mass Storage Systems and Technologies* (Apr. 2002), pp. 65–73.

[25] KALLAHALLA, M., RIEDEL, E., SWAMINATHAN, R., WANG, Q., AND FU, K. Plutus: scalable secure file sharing on untrusted storage. In *Proceedings of the Second USENIX Conference on File and Storage Technologies (FAST)* (San Francisco, CA, Mar. 2003), USENIX, pp. 29–42.

[26] KOTLA, R., ALVISI, L., AND DAHLIN, M. SafeStore: a durable and practical storage system. In *Proceedings of the 2007 USENIX Annual Technical Conference* (June 2007), pp. 129–142.

[27] LI, J., KROHN, M., MAZIÈRES, D., AND SHASHA, D. Secure untrusted data repository (SUNDR). In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI)* (San Francisco, CA, Dec. 2004).

[28] MANIATIS, P., ROUSSOPOULOS, M., GIULI, T. J., ROSENTHAL, D. S. H., AND BAKER, M. The LOCKSS peer-to-peer digital preservation system. *ACM Transactions on Computer Systems 23*, 1 (2005), 2–50.

[29] MERKLE, R. C. A digital signature based on a conventional encryption function. In *Advances in Cryptology - Crypto '87* (Berlin, 1987), Springer-Verlag, pp. 369–378.

[30] OXLEY, M. G. (H.R.3763) Sarbanes-Oxley Act of 2002, Feb. 2002.

[31] PINHEIRO, E., AND BIANCHINI, R. Energy conservation techniques for disk array-based servers. In *Proceedings of the 18th International Conference on Supercomputing* (June 2004).

[32] PINHEIRO, E., BIANCHINI, R., AND DUBNICKI, C. Exploiting redundancy to conserve energy in storage systems. In *Proceedings of the 2006 SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (Saint Malo, France, June 2006).

[33] PRESTON, W. C., AND DIDIO, G. Disk at the price of tape? an in-depth examination. Copan Systems white paper, 2004.

[34] QUINLAN, S., AND DORWARD, S. Venti: A new approach to archival storage. In *Proceedings of the 2002 Conference on File and Storage Technologies (FAST)* (Monterey, California, USA, 2002), USENIX, pp. 89–101.

[35] RHEA, S., EATON, P., GEELS, D., WEATHERSPOON, H., ZHAO, B., AND KUBIATOWICZ, J. Pond: the OceanStore prototype. In *Proceedings of the Second USENIX Conference on File and Storage Technologies (FAST)* (Mar. 2003), pp. 1–14.

[36] SAITO, Y., FRØLUND, S., VEITCH, A., MERCHANT, A., AND SPENCE, S. FAB: Building distributed enterprise disk arrays from commodity components. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2004), pp. 48–58.

[37] SCHROEDER, B., AND GIBSON, G. A. Disk failures in the real world: What does an MTTF of 1,000,000 hours mean to you? In *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST)* (Feb. 2007), pp. 1–16.

[38] SCHWARZ, T. J. E., XIN, Q., MILLER, E. L., LONG, D. D. E., HOSPODOR, A., AND NG, S. Disk scrubbing in large archival storage systems. In *Proceedings of the 12th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS '04)* (Oct. 2004), IEEE, pp. 409–418.

[39] SCHWARZ, S. J., T., AND MILLER, E. L. Store, forget, and check: Using algebraic signatures to check remotely administered storage. In *Proceedings of the 26th International Conference on Distributed Computing Systems (ICDCS '06)* (Lisboa, Portugal, July 2006), IEEE.

[40] SEAGATE TECHNOLOGY LLC. Momentus 5400 psd. `http://www.seagate.com/docs/pdf/marketing/ds_momentus_5400_psd.pdf`, Aug 2007.

[41] SUN MICROSYSTEMS. Sun StorageTek SL8500 modular library system. `http://www.sun.com/storagetek/tape_storage/tape_libraries/sl8500/`.

[42] SUN MICROSYSTEMS. Solaris ZFS and Red Hat Enterprise Linux EXT3 file system performance. `http://www.sun.com/software/whitepapers/solaris10/zfs_linux.pdf`, June 2007.

[43] TANABE, T., TAKAYANAGI, M., TATEMITI, H., URA, T., AND YAMAMOTO, M. Redundant optical storage system using DVD-RAM library. In *Proceedings of the 16th IEEE Symposium on Mass Storage Systems and Technologies* (Mar. 1999), pp. 80–87.

[44] TEHRANI, S., SLAUGHTER, J. M., CHEN, E., DURLAM, M., SHI, J., AND DEHERRERA, M. Progress and outlook for MRAM technology. *IEEE Transactions on Magnetics 35*, 5 (Sept. 1999), 2814–2819.

[45] WEDDLE, C., OLDHAM, M., QIAN, J., WANG, A.-I. A., REIHER, P., AND KUENNING, G. PARAID : A gear-shifting power-aware RAID. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST)* (Feb. 2007).

[46] WEIL, S. A., BRANDT, S. A., MILLER, E. L., LONG, D. D. E., AND MALTZAHN, C. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)* (Seattle, WA, Nov. 2006), USENIX.

[47] WESTERN DIGITAL. WD Caviar GP 1 TB SATA hard drives. `http://www.westerndigital.com/en/library/sata/2879-701229.pdf`, Aug. 2007.

[48] WILCKE, W. W., GARNER, R. B., FLEINER, C., FREITAS, R. F., GOLDING, R. A., GLIDER, J. S., KENCHAMMANA-HOSEKOTE, D. R., HAFNER, J. L., MOHIUDDIN, K. M., RAO, K., BECKER-SZENDY, R. A., WONG, T. M., ZAKI, O. A., HERNANDEZ, M., FERNANDEZ, K. R., HUELS, H., LENK, H., SMOLIN, K., RIES, M., GOETTERT, C., PICUNKO, T., RUBIN, B. J., KAHN, H., AND LOO, T. IBM Intelligent Bricks project—petabytes and beyond. *IBM Journal of Research and Development 50*, 2/3 (2006), 181–197.

[49] YAO, X., AND WANG, J. RIMAC: a novel redundancy-based hierarchical cache architecture for energy efficient, high performance storage systems. In *Proceedings of EuroSys 2006* (Oct. 2006), pp. 249–262.

[50] YOU, L. L., POLLACK, K. T., AND LONG, D. D. E. Deep Store: An archival storage system architecture. In *Proceedings of the 21st International Conference on Data Engineering (ICDE '05)* (Tokyo, Japan, Apr. 2005), IEEE.

[51] ZHU, Q., CHEN, Z., TAN, L., ZHOU, Y., KEETON, K., AND WILKES, J. Hibernator: Helping disk arrays sleep through the winter. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)* (Brighton, UK, Oct. 2005), ACM.

Logan: Automatic Management for Evolvable, Large-Scale, Archival Storage

Mark W. Storer, Kevin M. Greenan, Ian F. Adams
Ethan L. Miller, Darrell D.E. Long
Storage System Research Center
University of California, Santa Cruz
{mstorer, kmgreen, iadams, elm, darrell}@cs.ucsc.edu

Kaladhar Voruganti
NetApp
Kaladhar.Voruganti@netapp.com

***Abstract*—Archival storage systems designed to preserve scientific data, business data, and consumer data must maintain and safeguard tens to hundreds of petabytes of data on tens of thousands of media for decades. Such systems are currently designed in the same way as higher-performance, shorter-term storage systems, which have a useful lifetime but must be replaced in their entirety via a "fork-lift" upgrade. Thus, while existing solutions can provide good energy efficiency and relatively low cost, they do not adapt well to continuous improvements in technology, becoming less efficient relative to current technology as they age. In an archival storage environment, this paradigm implies an endless series of wholesale migrations and upgrades to remain efficient and up to date.**

Our approach, Logan, manages node addition, removal, and failure on a distributed network of intelligent storage appliances, allowing the system to gradually evolve as device technology advances. By automatically handling most of the common administration chores—integrating new devices into the system, managing groups of devices that work together to provide redundancy, and recovering from failed devices—Logan reduces management overhead and thus cost. Logan can also improve cost and space efficiency by identifying and decommissioning outdated devices, thus reducing space and power requirements for the archival storage system.

I. Introduction

The ability to store and maintain massive quantities of data is becoming increasingly important, as scientists, businesses, and consumers are increasingly aware of the value of archival data. Scientists have long attempted to preserve data archivally, though such efforts have sometimes fallen short. In the business arena, data preservation is often mandated by law [1, 17], and data mining has proven to be a boon in shaping business strategy. For individual consumers, archival storage is being called upon to preserve sentimental and historical artifacts such as photos, movies and personal documents. Unfortunately, traditional storage systems are not designed to meet the needs of long-term, archival data [6]. Paradoxically, despite the increasing value of archival data, high cost is one of the biggest obstacles to applying traditional storage techniques to design systems to house archival data. Long-term storage systems should be inexpensive enough to allow the preservation of all data that *might* eventually prove useful.

In contrast to traditional storage systems, which are typically more concerned with scalability in performance and capacity, an archival storage system designed for long-lived data must scale over many dimensions, including time, vendors and technologies [6]. The goal therefore, is to move away from an endless series of migrations and "fork-lift" upgrades, to a continuously evolving system. To realize this goal, we are developing Pergamum, a system that consists of a distributed network of up to 10^5–10^6 energy-efficient storage devices, called *tomes*, that communicate over a commodity Ethernet backplane [26]. While these devices can operate independently, their full potential is realized when they cooperate in inter-device redundancy groups to provide data reliability and ensure data longevity. Since the storage nodes are intelligent, each device contains specialized software that acts as an abstraction layer between the system, and the device's underlying hardware. This flexibility provides the potential for an adaptable system that changes gradually with technology; while individual components may change, the overall system evolves gracefully.

Although a fully distributed system is well-suited to an evolvable design, it introduces the problem of managing the global state of a fully decentralized system. Recent work has made great strides towards efficiently aggregating data over very large networks of distributed nodes [31, 32]; however, these approaches may be insufficient in a system that could easily encompass millions of nodes. A million-node distributed system must facilitate nodes joining the system, manage placement of data and redundancy information, handle node failure, and gracefully phase out nodes as they age. All of

this must be done with no central point of failure. However, it is impractical for each node to maintain global knowledge—keeping just 10 KB per node for each of a million nodes would require 10 GB of storage per node. Moreover, keeping that information current would require far too many messages to be exchanged between nodes. Instead, an archival system must allow nodes to operate with partial knowledge of the whole system and more complete knowledge of a small part.

Archival systems become more useful as their cost decreases, making energy efficiency an important aspect of long-term storage. However, while some earlier systems have addressed energy efficiency [8, 11, 18], none have examined how opportunity costs affect a system over its lifetime. Since drive capacity, real estate values and power costs are always increasing, system efficiency must be measured against what is currently achievable, not simply what was *once* achievable. For example, most storage systems assume that drives are replaced due to failure or wholesale system upgrades, suggesting that drives may remain in use well past the point of being economically efficient. Further, proactive decommissioning could also improve system reliability; earlier work has shown that the previously-held bathtub failure model for hard drives may not be valid [24], and that even small numbers of sector failures can presage overall drive failure [5].

Because our overall goal is to reduce the cost of long-term archival storage by reducing management cost, we designed Logan to address several major challenges. First, Logan must automatically integrate new nodes into an existing system. This includes both making the system aware of the new device, and integrating the new device's storage into redundancy groups to protect the data it stores against device failure. Second, Logan must ensure the correctness and longevity of data stored redundantly across nodes in the face of changing device membership and failures. Third, Logan should strive to keep power and other ongoing costs as low as possible by monitoring a device's *usefulness*—the utility it provides compared to the resources it consumes—and decommissioning the device when it has outlived its useful lifespan.

The remainder of the paper is organized as follows. Section II places Logan within the context of existing research. Next, Section III provides an overview of our current design. Finally, in Section IV we discuss where we plan to focus our future efforts; we conclude the paper in Section V.

II. Related Work

In designing Logan to meet the goals of energy-efficient, reliable, archival storage [6], we used concepts from a variety of projects. In this section, we place our work in the context of existing work by presenting an overview of relevant existing storage systems, and distinguishing them from Logan by identifying their architecture, intended workload, and cost strategy.

A. Distributed Communication

Because distributed systems are composed of loosely coupled, independent devices, system-wide communication can be challenging. An extreme approach is the pursuit of global awareness, in which a fully connected graph allows one-hop communications between any two nodes [2, 12, 14]. Unfortunately, the per node storage overhead, and proliferation of messages with these strategies make them suitable for only relatively small systems. Some systems have sought efficiency gains through the use of randomization, but even these approaches incur relatively heavy costs [7, 28].

Information management systems attempt to provide system level awareness, while still maintaining a decentralized, distributed architecture. Some, such as SDIMS and Shruti, utilize DHT algorithms as part of their foundation [31, 32]. These information management systems aggregate information about a distributed system's state, and make it available in a way that does not collect all of the information in a central point of failure. Another approach to data aggregation is seen in Astrolabe, which eschews DHTs in favor of a gossip-based approach [21]. Unfortunately, this approach focuses more on data summaries that data aggregation and can be inefficient for some workloads.

B. Storage Systems

Distributed, peer to peer architectures have been used in storage systems geared to a variety of workloads. Logan, which is designed to run on a Pergamum-style archival storage architecture [26], relaxes some performance criteria in exchange for economic efficiency, unlike many existing systems that attempt to achieve performance levels on par with traditional, centralized storage [22, 23].

Logan is designed for a fully distributed architecture with no need for specialized nodes and central repositories of information. In long-term storage scenarios, centralized points of failure can compromise data longevity. For example, Venti, which can store archival data on removable media, utilizes a centralized index [20]. Unfortunately, while this index can be rebuilt by reading in partial indices stored on media, the bottleneck of media readers makes this a prohibitively lengthy procedure as overall data size increases to hundreds of petabytes and

beyond. Similarly, the Google File System is a semi-distributed system that utilizes master nodes. However, as it was not designed for long-term archival data, it avoids the recovery problem by relaxing consistency and longevity constraints [9].

Energy efficiency is an area that many designs have explored in pursuit of cost savings. Some reports state that commonly used power supplies operate at only 65–75% efficiency, representing one of the primary culprits of excess heat production, and contributing to cooling demands that account for up to 60% of data-center energy usage [10]. The development of Massive Arrays of Idle Disks (MAIDs) generated large cost savings by leaving the majority of a system's disks spun down [8]. Further work has expanded on the idea by incorporating strategies such as data migration, the use of drives that can spin at different speeds, and power-aware redundancy techniques [16, 18, 19, 29, 33, 35].

Finally, a number of projects have sought to yield cost savings from improved management techniques [3]. IBM's Intelligent Bricks project utilizes intelligent storage appliances in liquid cooled rack designed for very high device density [30]. Unfortunately, the design intentionally forgoes accessibility of nodes as a trade off for higher density. In a long-term scenario, this implies that eventually failed nodes will consume floor space while providing no utility. Others, such as Sun's Honeycomb project, trade energy efficiency for management gains [27]; in some scenarios, as many as sixty disks could be involved with a single write.

III. Proposed Design

The system we envision is driven by a workload that exhibits read, write and delete behavior that differs from typical disk-based workloads, providing both challenges and opportunities. The workload is write-heavy, motivated by regulatory compliance and the desire to save any data that *might* be valuable at a later date. Reads, while relatively infrequent, are often part of a query or audit and thus are likely to refer to data that is temporally correlated. Similarly, data being deleted are likely to exhibit a temporal relationship since retention policies often specify a maximum data lifetime. This workload resembles traditional archival storage workloads [20, 34], adding deletion for regulatory compliance.

We are currently developing Logan, a management layer that runs atop the distributed network of inexpensive, independent storage appliances provided by Pergamum [26]. This architecture provides a number of advantages in a long-term archival scenario. First, each device acts as an abstraction layer to the underlying media, easing transitions to new technologies. Second, using a high number of low powered processors yields energy savings versus a few high powered processors (cutting processor voltage in half results in half the clock speed but one fourth the power consumption). Third, an inexpensive node can be treated as an indivisible entity; if any part of the node fails, the entire node is discarded and replaced. This reduces the management overheard associated with locating and replacing individual components.

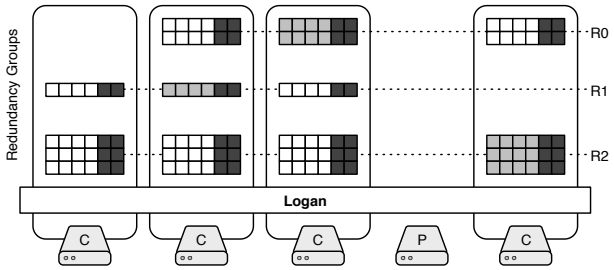


Fig. 1. Overview of Logan running a distributed network of devices. One device, *P*, is in the PENDING state, while the others, *C*, are CONTRIBUTING in redundancy groups. Data blocks (white) are protected with internal parity (dark grey) and external parity.

While each device is independent and actively ensures the longevity of its own data, nodes also cooperate in *redundancy groups* to provide system wide data reliability. Data in each device is divided into fixed sized *blocks*, and blocks are grouped into fixed sized *segments*. In a large archival system, data failures will be quite common, and thus a "one size fits all" approach to reliability is excessively wasteful. Figure 1 illustrates the two-level reliability model used in Pergamum [26], the system on which Logan runs. First, each device survives media faults by utilizing block-level erasure coding over segments [4]. Second, distributed RAID techniques are used over groups of segments to survive the loss of a device [25]. Since heterogeneity is inevitable in an evolvable system, device capacity will vary between appliances. Thus, unlike a simple RAID system where all drives are the same size, a device can contribute segments to more than one redundancy group in order to utilize all of its local storage capacity.

A. Management Groups

Because global knowledge becomes increasingly infeasible as the number of nodes increases, devices are arranged into *management groups*. Each management group chooses one or more group leaders that oversees administration for a given term. At a system wide level, the management groups are arranged in a logical hypercube because this topology offers a number of benefits. First, it offers efficient communications routing. As shown in Figure 2, message routing occurs in a

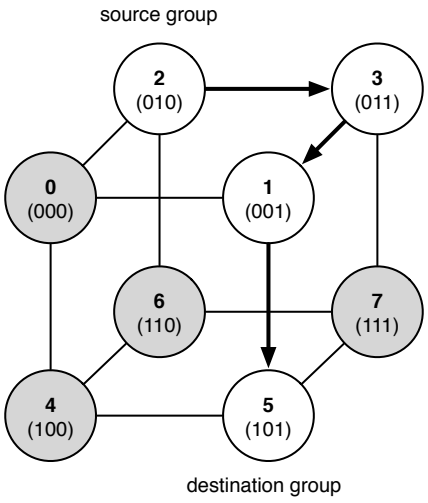


Fig. 2. Eight management groups arranged in a hypercube of dimension 3. Nodes involved in routing from group 2 to 5 shown in white. Routing is done in $O(\lg n)$ time, since each hop brings the message one bit closer to its destination.

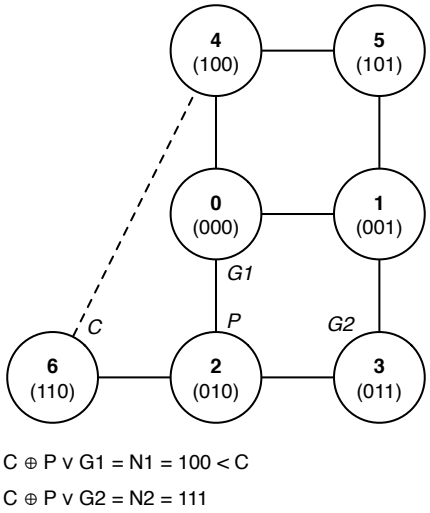


Fig. 3. When parent *P* (node 2), produces a child *C* (node 6), it provides the child with a list of its grandparents *G*1 and *G*2 (nodes 0 and 3). The child then calculates which, if any of its bitwise neighbors it needs an introduction to from its grandparent.

hierarchical fashion, with messages first routed to the destination group, and finally routed within the destination group. Second, management groups grow until they reach a certain size, and then are split into two groups. This technique for incremental growth in the number of management groups is well suited to the construction of hypercubes.

The system begins with a single management group. Nodes entering the system are added to this group until it reaches a predetermined saturation point, at which point it splits into two groups with an average of half the membership each. Membership and splitting is based on the LH* family of distributed data structures [15]. This approach offers several benefits. First, these data structures do not require a globally consistent view in order to function properly. This property is especially important because, in a system of the scale we envision, tight consistency expectations are unrealistic. Second, they allow the system to gracefully scale from a small system of just a single management group to a large system with thousands. Third, since group membership is calculated instead of statically assigned, a node can be located from its name alone.

LH* utilizes two variables, *n* and *i*, to coordinate all of operations. First, system routing utilizes these variables in the hash function used to determine which management group a node belongs to. In the event that a node is routed using an older version of *n* and *i*, the selected bucket will route the message to the correct bucket, as well as update the source with the up to date variables values. Second, *n* acts a token allowing distributed splitting; when bucket *n* splits, it passes the token to bucket $n+1 \bmod 2^i$.

When a group splits, it must establish connections with its bitwise neighbors, *N*, in order to preserve the logical hypercube. One such connection is from the child, *C*, to its parent, *P*. This connection is easy to make. Additionally, it must establish a connection with each existing group whose name is exactly one bit different than its own name. To perform this, the parent supplies the child with a list of its grandparents, *G*. For each grandparent, the child calculates $C \oplus P \vee G = N$. If $N < C$, then *C* asks *G* to make an introduction. If $N \geq C$, then that bitwise neighbor has not yet been formed and no further action is required. This process is illustrated in Figure 3.

B. Device Lifetimes

When a new node enters the system, it performs a broadcast to nearby nodes to locate other Logan devices; there is no need for the new node to broadcast to all (or even most) of the nodes in the system. The devices that respond are able to provide the new node with their view of the global system state, which in turn is used by the new node to calculate its management group. Once a new node has calculated its management group, it asks for further assistance in locating a fellow member of its group. Eventually the new node is introduced to the current group leader.

Once a node has become member of the system, it is managed through its entire lifespan, progressing through three states: PENDING, CONTRIBUTING, and EXPIRED.

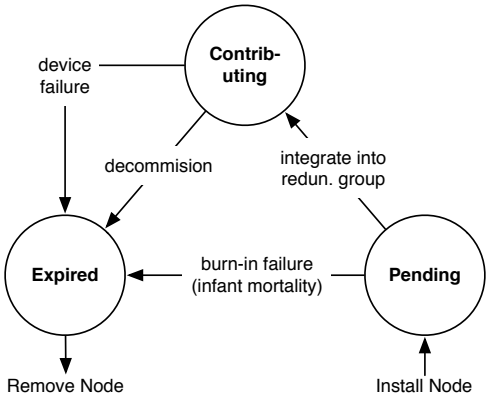


Fig. 4. Nodes in the system exist in one of three states. A functioning node that is not yet part of any redundancy groups is in the initial, PENDING state. CONTRIBUTING nodes have been integrated into redundancy groups. EXPIRED nodes have either failed or been decommissioned, and can be removed from the system.

The first state, PENDING, indicates a node that is alive on and known to the system, but is not yet a member of any redundancy groups. The second state, CONTRIBUTING, denotes a live node that is member of one or more redundancy groups. The third state, EXPIRED, indicates that a node has failed or been decommissioned. EXPIRED nodes can be physically removed from the system; if other nodes are removed, the system handles the removal as a failure. Figure 4 illustrates these three states and the transitions between them.

A newly installed node is not immediately integrated into redundancy groups, but rather is placed into the PENDING state. This design provides a number of benefits. First, this allows the node to undergo a self-check and burn-in period in order to reduce the impact of infant mortality and batch correlated failures. Second, when it is time to expand the available storage in the system, Logan is able to make smarter management decisions by utilizing the devices in the PENDING pool, as compared to an approach that immediately integrates every node as soon as it arrives.

When Logan integrates a device into into one or more redundancy groups, that node enters the CONTRIBUTING state. In this mode, nodes store data, participate in redundancy group activities, and answer read and write requests. Eventually, as the node ages and its usefulness decreases relative to newer nodes, it will increasingly become a candidate for decommissioning.

C. Management Tasks

Management groups are tasked with a number of administrative duties. The first of these, *scale out*, deals with expanding the capacity of the system. It involves the creation of redundancy groups, and the assignment of segments to those groups. The second area, *recovery*, determines where data will be recovered to when a node is lost. The final area, *maintenance*, monitors the health of the system and actively identifies nodes that are ready to be decommissioned.

Each device in Logan maintains a list of named attributes that describe that device. In addition to querying the device for values, the system can also update the values. This can be used to reflect usage effects such as the accelerated wear caused by drive spin-ups, or the effects of batch correlated failures.

In order to make good management decisions, we are exploring the use of heuristic algorithms such as simulated annealing [13]. These algorithms attempt to solve an optimization problem by utilizing heuristics to repeatedly perform minor modifications to a partial solution. To this end, these algorithms utilize three main components. First, the solution space, X is the space of all possible solutions from which the answer will be drawn. Second, the neighbor function, N, heuristically chooses a new solution that is "close" to the current solution in the solution space. Finally, a objective function, P, measures the "goodness" of a solution, and is the value that the heuristic algorithm attempts to minimize or maximize.

Management group leaders collect the attribute lists from the members in their groups in order to develop a statistical understanding about the group's devices. This information is used during the administrative functions to identify expensive devices, in terms of utility versus resources consumed, without requiring administrator input. For example, this approach can identify a group's most power-hungry device. Further, this approach can determine how power-hungry that device is in comparison to the average of the group's devices.

1) Scale Out: For scale-out operations, each management group maintains a list of its redundancy groups and the devices assigned to those groups. This list is consulted and updated based on two redundancy group operations. First, Logan can form a new redundancy group. Second, Logan can expand an existing redundancy group. The latter strategy is possible because redundancy groups have a population range. Logan does not always fully populate new redundancy groups. Rather, it creates partially populated groups that still meet the system's reliability criteria, thus allowing the system to expand capacity, even when there are insufficient devices to create an entirely new redundancy group. For example, the system might require parity groups to be of the form $n+3$ disks, where $6 \leq n \leq 13$. This would mean that a redundancy group would have a minimum

of 9 disks and a maximum of 16 disks, and be able to grow from 9 to 16 gradually over time if needed.

At the device level, each management group maintains a list of of its devices and their unassigned, or free, segments. From this pool, Logan can assign device segments to redundancy groups from two primary sources. First, Logan can utilize previously unassigned segments from a device in the CONTRIBUTING state. Second, it can utilize segments from a PENDING device. Naturally, this would cause the device to transition to the CONTRIBUTING state.

Logan monitors the system, and performs a scale out operation when it detects that available free space in a management group has dropped below a predetermined low water mark. When this occurs, the management group performs a number of scale-out initialization steps. First, it determines which redundancy group are less than fully populated. Second, it determines if the existing groups offer sufficient scale-out room, or if new redundancy groups must be created.

2) Recovery: As with any storage system, and especially a long-term archival system, failure is inevitable. Additionally, since the system must be cost efficient, it is not enough to simply recover data to the first available free space. To address this problem, Logan uses similar heuristic search techniques to determine where data should be recovered to in the event of a device failure.

An instance of solution space is a mapping of segments to redundancy groups. At each iteration of the algorithm, some subset of free segments are mapped to the segments of the failed device. The primary constraint to enforce during recovery is that each member of a redundancy group is a different device.

3) Maintenance: The goal of maintenance is to determine if there is a management group configuration that can offer better service for the same or lower resource consumption.

As in previous management group operations, the state of the system consists of a mapping of device free segments to redundancy groups. However, in the case of maintenance, the redundancy group list consists of all the existing redundancy groups. At each iteration, devices that are likely to be decommissioned based on their expected lifetime or high energy costs per segment are randomly swapped with available segments. For this operation, a valid solution enforces the constraint that a device can only be decommissioned if all of its committed segments have a replacement, and that those replacements conform to the standard redundancy group constraints.

Unlike recovery and scale-out which are performed as soon as the heuristic completes, maintenance chores can handled opportunistically. A device that has been identified for decommissioning can wait until a scrubbing event or recovery event occurs in order to defray the power costs associated with a wholesale migration of a nodes complete contents. An important factor that enables this opportunistic approach is that the optimizations that maintenance seeks to achieve are not critical to data safety. When the unit being decommissioned activates, it can check to see if the units slated to takes its place in redundancy groups are still available. If they are not, the decommissioning can be cancelled, or new replacements can be chosen.

IV. Future Work

Current effort on Logan is focused on refining the skeleton, described in the previous section. Much of this work is directed towards exploring the use of heuristic algorithms in making sound management decisions. Additionally, we are exploring the behavior of the system to help determine the correct size of management groups; too large and the group leaders are overwhelmed, too small and the resulting splitting results in unnecessary management overhead. Finally, we are examining the boot-strapping problem; while the system is designed to scale up to hundreds of thousands of nodes, it must inevitably start with one.

Further along in our research plans, we plan on examining how best to deal with large-scale disasters and network partitions. In a long-term storage system, these sorts of events are inevitable, and must be survived gracefully, and with a minimum of needless energy expenditures. Many such events, such as a failed switch causing a network partition, are benign in the sense that data may still be safe, it is simply unreachable. However, the system's reaction in such a scenario could inadvertently cause more harm than good; the system may try and immediately rebuild all data that it could not contact.

As previously discussed, large archival systems are well suited to recovery procedures that allow the response to be scaled to the size of the problem. Currently, we utilize a two level scheme of intra-device and inter-device reliability. A third level, across geographically diverse sites, would be useful in order to protect data from natural disasters or other other "act of god" failures.

The dependency list of a given device describes the nodes that contribute to the reliability of a given node's data. Put another way, if a device fails, all of the device's in the failed device's adjacency list will need to contribute data during the recovery process. Thus, the size of the dependency list could have considerable

impact on data reliability, and during recovery, energy consumption. A large redundancy group allows greater parallelization during recovery, and implies greater diversity in the redundancy group's devices. In contrast a smaller adjacency list requires less devices to spin up during recovery. Considering these and other potential tradeoffs, an understanding of how adjacency affects reliability and power consumption could allow us to tailor our optimization methods to their ideal size.

Another intersection of reliability and power can be seen in a failed devices recovery schedule. That is, the amount and ordering of parallelization that occurs during rebuild. With a fuller understanding of power use during rebuild Logan could determine not only the placement of recovered data, but also the order that recovery should proceed. This area is complicated by the affect of very transient system states. For example, device population changes much slower than the list of currently spun up devices.

V. Conclusions

While archival systems are well served by a distributed architecture, such a design introduces the management challenges of heterogeneity in an evolving and aging system. Further, as part of a comprehensive cost strategy, such a system should continuously seek ways to maximize the utility it offers for the resources it is consuming.

To this end, we are developing Logan, a management layer that runs atop, a distributed network of energy-efficient, intelligent storage appliances [26]. Nodes are arranged in redundancy groups which allows data to be recovered from a lost node. To manage redundancy groups, and to facilitate system-wide communication, Logan arranges devices into management groups. Further, Logan collects information about the nodes in each management group and uses this data to make intelligent management decisions. Logan helps control archival storage costs by automating a number of common administrative tasks, and opportunistically decommissioning of old hardware.

Acknowledgments

We would like to thank our colleagues in the Storage Systems Research Center (SSRC) who provided valuable feedback. This research was supported by the Petascale Data Storage Institute under Dept. of Energy award DE-FC02-06ER25768, and by the industrial sponsors of the SSRC, including Los Alamos National Lab, Lawrence Livermore National Lab, Sandia National Lab, Data Domain, Hewlett-Packard Laboratories, IBM Research, LSI, NetApp, Seagate, and Symantec.

References

[1] "Health Information Portability and Accountability Act," 104th Congress, Oct. 1996.

[2] I. Abraham and D. Dolev, "Asynchronous resource discovery," in *Proceedings of the Twenty-Second Annual Symposium on Principles of Distributed Computing (PODC 2003)*, Boston, MA, Jul. 2003, pp. 143–150.

[3] E. Anderson, M. Hobbs, K. Keeton, S. Spence, M. Uysal, and A. Veitch, "Hippodrome: running circles around storage administration," in *Proceedings of the 2002 Conference on File and Storage Technologies (FAST)*, Monterey, CA, Jan. 2002.

[4] L. N. Bairavasundaram, G. R. Goodson, S. Pasupathy, and J. Schindler, "An analysis of latent sector errors in disk drives," in *Proceedings of the 2007 SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, Jun. 2007.

[5] L. N. Bairavasundaram, G. R. Goodson, B. Schroeder, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "An analysis of data corruption in the storage stack," in *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST)*, Feb. 2008, pp. 223–238.

[6] M. Baker, M. Shah, D. S. H. Rosenthal, M. Roussopoulos, P. Maniatis, T. Giuli, and P. Bungale, "A fresh look at the reliability of long-term digital storage," in *Proceedings of EuroSys 2006*, Apr. 2006, pp. 221–234.

[7] B. S. Chlebus and D. R. Kowalski, "Gossiping to reach consensus," in *Proceedings of the 14th ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, Winnipeg, Manitoba, Aug. 2002, pp. 220–229.

[8] D. Colarelli and D. Grunwald, "Massive arrays of idle disks for storage archives," in *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing (SC '02)*, Nov. 2002.

[9] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google file system," in *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*. Bolton Landing, NY: ACM, Oct. 2003.

[10] Green Grid Consortium, "The green grid opportunity, decreasing datacenter and other IT energy usage patterns," http://www.thegreengrid.org, The Green Grid, Feb 2007.

[11] A. Guha, "Solving the energy crisis in the data center using COPAN Systems' enhanced MAID storage platform," Copan Systems white paper, Dec. 2006.

[12] M. Harchol-Balter, T. Leighton, and D. Lewin, "Resource discovery in distributed networks," in *Proceedings of the Eighteenth ACM Symposium on Principles of Distributed Computing (PODC 1999)*, Atlanta, GA, May 1999, pp. 229–237.

[13] S. Kirkpatrick, C.D. Gelatt, Jr., and M. Vecchi, "Optimization by simulated annealing," *Science*, vol. 220, no. 4598, pp. 671–680, 1983.

[14] S. Kutten, D. Peleg, and U. Vishkin, "Deterministic resource discovery in distributed networks," in *Proceedings of the 13th ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, Crete Island, Greece, Jul. 2001, pp. 77–83.

[15] W. Litwin, M.-A. Neimat, and D. A. Schneider, "LH*—a scalable, distributed data structure," *ACM Transactions on Database Systems*, vol. 21, no. 4, pp. 480–525, 1996.

[16] D. Narayanan, A. Donnelly, and A. Rowstron, "Write off-loading: Practical power management for enterprise storage," in *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST)*, Feb. 2008, pp. 253–267.

[17] M. G. Oxley, "(H.R.3763) Sarbanes-Oxley Act of 2002," Feb. 2002.

[18] E. Pinheiro and R. Bianchini, "Energy conservation techniques for disk array-based servers," in *Proceedings of the 18th International Conference on Supercomputing*, Jun. 2004.

[19] E. Pinheiro, R. Bianchini, and C. Dubnicki, "Exploiting redundancy to conserve energy in storage systems," in *Proceedings of the 2006 SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, Saint Malo, France, Jun. 2006.

[20] S. Quinlan and S. Dorward, "Venti: A new approach to archival storage," in *Proceedings of the 2002 Conference on File and Storage Technologies (FAST)*. Monterey, California, USA: USENIX, 2002, pp. 89–101.

[21] R. V. Renesse, K. P. Birman, and W. Vogels, "Astrolabe: A robust and scalable technology for distributed system monitoring, mangement, and data mining," *ACM Transactions on Computer Systems*, vol. 21, no. 2, pp. 164–206, May 2003.

[22] 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: ACM, Oct. 2001, pp. 188–201.

[23] Y. Saito, S. Frølund, A. Veitch, A. Merchant, and S. Spence, "FAB: Building distributed enterprise disk arrays from commodity components," in *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2004, pp. 48–58.

[24] B. Schroeder and G. A. Gibson, "Disk failures in the real world: What does an MTTF of 1,000,000 hours mean to you?" in *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST)*, Feb. 2007, pp. 1–16.

[25] M. Stonebraker and G. A. Schloss, "Distributed RAID—a new multiple copy algorithm," in *Proceedings of the 6th International Conference on Data Engineering (ICDE '90)*, Feb. 1990, pp. 430–437.

[26] M. W. Storer, K. M. Greenan, E. L. Miller, and K. Voruganti, "Pergamum: Replacing tape with energy efficient, reliable, disk-based archival storage," in *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST)*, Feb. 2008.

[27] Sun Microsystems, "Sun StorageTek 5800 system architecture," White paper, Dec. 2007.

[28] D. J. Watts and S. H. Strogatz, "Collective dynamics of 'small-world' networks," *Nature*, vol. 393, no. 6684, pp. 440–442, Jun. 1998.

[29] C. Weddle, M. Oldham, J. Qian, A.-I. A. Wang, P. Reiher, and G. Kuenning, "PARAID : A gear-shifting power-aware RAID," in *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST)*, Feb. 2007.

[30] W. W. Wilcke, R. B. Garner, C. Fleiner, R. F. Freitas, R. A. Golding, J. S. Glider, D. R. Kenchammana-Hosekote, J. L. Hafner, K. M. Mohiuddin, K. Rao, R. A. Becker-Szendy, T. M. Wong, O. A. Zaki, M. Hernandez, K. R. Fernandez, H. Huels, H. Lenk, K. Smolin, M. Ries, C. Goettert, T. Picunko, B. J. Rubin, H. Kahn, and T. Loo, "IBM Intelligent Bricks project—petabytes and beyond," *IBM Journal of Research and Development*, vol. 50, no. 2/3, pp. 181–197, 2006.

[31] P. Yalagandula and M. Dahlin, "A scalable distributed information management system," in *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM '04)*. Portland, OR: ACM Press, Aug. 2004, pp. 379–390.

[32] P. Yalagundula and M. Dahlin, "Shruti: A self-tuning hierarchical aggregation system," in *Proceedings of the First International Conferece on Self-Adaptive and Self-Organizing Systems (SASO 2007)*, Boston, MA, Jul. 2007, pp. 141–150.

[33] X. Yao and J. Wang, "RIMAC: a novel redundancy-based hierarchical cache architecture for energy efficient, high performance storage systems," in *Proceedings of EuroSys 2006*, Oct. 2006, pp. 249–262.

[34] L. L. You, K. T. Pollack, and D. D. E. Long, "Deep Store: An archival storage system architecture," in *Proceedings of the 21st International Conference on Data Engineering (ICDE '05)*. Tokyo, Japan: IEEE, Apr. 2005.

[35] Q. Zhu, Z. Chen, L. Tan, Y. Zhou, K. Keeton, and J. Wilkes, "Hibernator: Helping disk arrays sleep through the winter," in *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)*. Brighton, UK: ACM, Oct. 2005.

Long-Term Threats to Secure Archives

Mark W. Storer Kevin Greenan Ethan L. Miller
Storage Systems Research Center
University of California, Santa Cruz
mstorer@cs.ucsc.edu kmgreen@cs.ucsc.edu elm@cs.ucsc.edu

ABSTRACT

Archival storage systems are designed for a write-once, read-maybe usage model which places an emphasis on the long-term preservation of their data contents. In contrast to traditional storage systems in which data lifetimes are measured in months or possibly years, data lifetimes in an archival system are measured in decades. Secure archival storage has the added goal of providing controlled access to its long-term contents. In contrast, public archival systems aim to ensure that their contents are available to anyone.

Since secure archival storage systems must store data over much longer periods of time, new threats emerge that affect the security landscape in many novel, subtle ways. These security threats endanger the secrecy, availability and integrity of the archival storage contents. Adequate understanding of these threats is essential to effectively devise new policies and mechanisms to guard against them. We discuss many of these threats in this new context to fill this gap, and show how existing systems meet (or fail to meet) these threats.

Categories and Subject Descriptors

D.4.6 [**Security and Protection**]: Access controls; H.3 [**Information Systems**]: Information Storage and Retrieval

General Terms

Design, Security

Keywords

secure storage, survivable storage, archival storage, secret splitting, encryption, cryptography, threat modeling

1. INTRODUCTION

The drive to archive information in digital form brings new challenges. Recent legislation, such as Sarbanes-Oxley, have placed strict demands on the preservation and retrieval properties of long-term storage systems. Archival storage systems must meet specific demands inherent to the usage model of write-once, read-maybe and long data lifetimes. In addition, there are novel security threats that are unique to long-term storage.

Traditionally, the adversaries to storage security have been roughly classified into two categories, passive and active. The active adversary is a malicious entity that actively attempts to break in and compromise an archive. In contrast, a passive adversary's actions are restricted to eavesdropping on communication channels. In the realm of long-term storage, the long data lifetimes of the storage contents suggest that time itself becomes an active adversary. It is constantly working against the system and threatening to compromise data. Throughout this discussion we will see that threats to traditional data storage systems take on a different meaning when viewed with the long view of the future that archival storage demands.

Archival storage systems attempt to address the issues of longevity that traditional storage systems have long ignored. As a result, some have come to refer to our current era as the digital dark ages. Danny Hillis noted [3], "For example, when we finally shut down the old PDP-10 at the MIT Artificial Intelligence Lab, there was no place to put files except onto mag tapes that are by now unreadable. So we lost the world's first text editor, the first vision and language programs, and the early correspondence of the founders of artificial intelligence." As our collective knowledge and artistic heritage becomes increasingly digital, the need to preserve data indefinitely becomes critical. For archival storage to succeed it is vital that we have an understanding of the security threats it faces.

2. SECURITY THREATS

When dealing with archival storage, the time frames for data lifetimes extend from the rather short-term scale of months or years to the longer-term scale of decades or even indefinite. The presence of long data lifetimes introduces some unique security threats. Some of these threats are variations on common concerns that take on new meaning in the area of archival storage while others are new threats that traditional storage system are largely unaffected by.

2.1 Long-Term Secrecy

Secrecy in long-term storage presents a complex challenge. Often, systems that provide file secrecy do so through the use of encryption [8, 12, 14]. The use of encryption within a storage system can be described in terms of the encrypted data's relevant lifetime. In long-lived encryption, such as for an encrypted file in an archival storage system, the data is persistent and the key must be preserved for an indefinite period of time. In contrast, an example of short-lived encryption would be a communication channel that is secured through the use of encryption and a key that is relevant only for the duration of the session. Long-lived encryption introduces a number of security threats.

StorageSS'06, October 30, 2006, Alexandria, Virginia, USA.

In an archival storage system, data can be very difficult to reproduce. The software, hardware and even users that produced the data may no longer be available. Encryption keys are a single point of failure and key loss is effectively equivalent to data deletion.

With the long-data lifetimes of archival storage, the use of encryption usually introduces the related problem of re-encryption at a future date. A number of scenarios could introduce the need to re-encrypt the contents of long-term file storage, including key rotations, compromised keys, compromised encryption algorithms and the need for access revocation. In some cases, re-encryption may only be needed for a few files, but in other cases, many petabytes of data might need re-encryption.

The re-encryption of large amounts of data must be done in a timely manner, especially if required for a key compromise or algorithm exploit. However, the time to apply the new encryption technique to a large amount of data will probably not be the limiting factor; rather, issues with maintaining keys and actually reading all of the old data quickly will be critical. Further, optimizations to speed the migration likely come with an associated management cost. For example, if a system chooses to save time by encrypting over the old algorithm, it must have a way of dealing with key histories and key distribution. In contrast, if the system chooses to decrypt the data before applying the new algorithm then it must have access to the users' encryption keys. Further, this access must must prevent a malicious user with system-level access from accessing data she is not authorized to view.

Even the traditional security threats and precautions associated with keyed encryption are made more difficult by the long data lifetimes found in archival storage. For example, cryptography is only computationally bound and it is very difficult to predict the future of cryptography and cryptanalysis. Technology such as quantum computing could usher in drastic changes to cryptography, and a long-term archival system which provides secrecy through encryption must be capable of handling those changes.

2.2 Locating Data

While the archival model of storage is write-once, read-maybe, users must still be able to find the data they stored should they desire to read it. Perhaps due to the fact that archival storage is not intended to be a user's primary data storage solution, it suffers from the potential problem of users forgetting where their data is stored. If users are unable to locate their data within an archival system, the availability aspect of security has been violated. For example, the Sarbanes-Oxley act specifies not only that data must be retained for a given period of time but also that it be retrievable in a timely manner.

One aspect of this problem lies in the location of data indices. A centralized index, maintained alongside the storage contents, has the benefit of relieving the user of remembering the details associated with storing their data. The centralized index is similar to a client that knows which bank their safety deposit box is in but not which box is theirs. This client might rely on a bank-maintained index which maps clients to their safety deposit box. One possible danger with this strategy is that an attacker that can compromise the centralized index is able to perform a much more targeted attack.

Another option, in contrast to the central index, places the onus of maintaining an index upon the client. In the safety deposit box analogy, the client would know which bank the safety deposit box is in as well as the location or ID of the safety deposit box itself. In this manner, even if a malicious person gained access to the vault, he would still have a brute-force problem of locating a specific box. With the personal index strategy, an attacker that compromises one user's index learns very little about the other users' data. This strategy suffers in that the client then has more long-term responsibility.

2.3 Authentication and User Accounts

In long-term secure storage, authentication must cope with a few scenarios not usually encountered in other storage arenas. If the storage system plans on providing file secrecy as part of security then it follows that the users must be able to authenticate themselves to the system as a first step in authorization. In other words, the users must show who they are before the system can determine what they are allowed to do.

A challenge unique to long-term secure storage is the problem that the user primarily attached to the data may no longer be available—for example, the user may be dead. Suppose a user wants to securely store their will in an archival storage system. The subsequent read may take place decades later by the owner's next of kin after the owner had passed away. A secure, archival storage system must thus be able to authenticate new users and establish their relationship to resources attached to existing users.

Additionally, if an archival storage system is able to authenticate a user to the system and determine her access permissions, it must be able to provide access to the entitled data in a meaningful fashion. For example, suppose a system utilizes encryption in order to provide file secrecy. If a user with rights to the data can view the file but has no access to the encryption key, she still has effectively no access. To continue the previous example, if a deceased user's next of kin proves his legal right to the data, the secrecy mechanism must function in the complete absence of the user that wrote the file.

2.4 Integrity Guarantees

Due to the rather short lifetime and limited reliability of traditional storage components, data begins to degrade as soon as it is placed on media. In short-term systems, this threat is relatively low because a great deal of frequently used data is accessed and updated on a regular basis. Archival storage assumes a write-once, read-maybe access pattern, thus the integrity of the data in the system must be actively checked at regular intervals.

Integrity checks often come in the form of *disk scrubbing* (also called *auditing*) procedures. A scrubbing procedure periodically scans sections of the data and uses strong hashes to detect corruption, recovering data that is damaged. A variety of problems can arise related to these scrubbing procedures. First, an overactive scrubbing procedure can contribute to media failures. In addition, an under-active scrubbing procedure may provide no utility over the absence of disk scrubbing. Xin, *et al.* analyzed these problems [23], showing that opportunistic scrubbing provides a good balance between overactive and under-active techniques. Unfortunately, opportunistic disk scrubbing policies require scrubbing requests to piggy-back on regular access requests. In a write-once, read-maybe system such policies may not be sufficient.

In the long term, cryptographic hashes may not be sufficient for integrity checking procedures. Given a reasonable amount of time, it is possible for an adversary to find collisions in the hash used for disk scrubbing. The adversary can update the original data with incorrect data, thus fooling the integrity checking procedure.

Compared to the internal data integrity problem of data within an archive becoming corrupted, distributed storage systems have an additional external integrity challenge. These systems must have a method of ensuring that the other archives in the distributed system are behaving properly. In such systems, one popular method of monitoring external integrity is through the use of a challenge-

response protocol. Of course, a secure system cannot rely on protocols that compare data in-the-clear.

Some have suggested the use of algebraic signatures as a way of performing integrity checks without exposing too much information [16]. Algebraic signatures have the property that the signatures of the parity equals the parity of the data signatures. For example, suppose a RAID stripe uses single parity as an erasure encoding, where the data symbols $d_1, d_2, ..., d_m$ compute the parity p. The XOR-sum of the algebraic signatures of the data, $sig(d_1) \oplus sig(d_2) \oplus ... \oplus sig(d_m)$, equals the signature of the parity element p. This scheme can also be extended to multiple erasure correcting codes, such as XOR-based Reed-Solomon.

As long as data is striped across the archives into reliability groups, a restricted distributed disk scrubbing algorithm can be used without disclosing too much information. If the disk scrubbing algorithm runs unrestricted, an adversary has the potential to extract information without proper authentication. For example, suppose that any party that can authenticate with an archive can submit an unbounded number of signature requests. If an adversary constructs enough sets of overlapping signature requests to an archive, then the sets form a system of linear equations that can be solved to reveal data.

2.5 Slow Attacks

When dealing with archival storage, the time frames for data lifetimes extend from the rather short-term scale of months and years to the longer-term scale of decades. This long lifetime gives attackers a much larger window within which they can attempt to compromise a security system. With archival storage an assailant might have several decades of time to conduct an attack.

One difficulty with slow attacks is intrusion detection. If the attack is methodical enough to make only the slightest of changes at any one time and each step was spaced far enough apart, it would be difficult to detect by traditional signature matching algorithms. This technique compares audit data and network activity to a database of known attacks. Thus, if the audit data contains a slight enough anomaly, it may not be enough to trigger a match with a known attack signature.

Another difficulty with detecting slow attacks is the problem of maintaining attack history. Security logging is especially important for systems that utilize secret-sharing algorithms such as PASIS [22], POTSHARDS [17] and others [18, 24]. Secret-sharing algorithms, while provably secure, rely on keeping a sufficient number of secret shares secure. This method of insuring file secrecy thus necessitates the maintenance of a history of compromises.

An example of the threat that slow attack presents is as follows. Suppose a file is protected using a 3/5 scheme in which the file is split into five pieces, three of which are required to rebuild the file. Now suppose that an attacker has compromised the system and obtained one of the shares. A decade later that same attacker obtains a second share. Due to the provably secure nature of the secret-sharing algorithms it can be shown that the attacker can gain no information about the data. However the system must deal with the fact that the attacker is making progress. With one more share the file would be revealed. Thus, the system must either immediately deal with any compromise or maintain a history of compromises in order to intelligently schedule corrective action.

2.6 Migration and Recovery

Due to the long data lifetimes of archival data, long-term storage systems will witness events that require data to be moved between archives. Reasons for this change include the inevitable failure or obsolesce of hardware, system updates and possibly even data movement as a security factor.

In archival systems as in more traditional storage, hardware failure is inevitable [23]. A long-term system must thus be immune to the failure of any given component. Additionally, the act of recovery and migration of effected data to new hardware should not compromise the availability aspect of security.

Even if hardware failure were not a factor, storage technology changes at a rapid pace. Storage systems today are quite different from those of decades past in terms of performance, capacity, media and interfaces. It is not a stretch of the imagination to assume that decades from now, today's technology will seem just as anachronistic as a punch card or drum storage device looks today. A storage system that is intended to keep data secure for several decades must thus be able to adapt to these changes by incorporating new technology and migrating data from outdated components. However, it must do so without allowing any party to actually recover data.

Another possibility for the role of data migration in a secure long-term storage system is to use migration to effect greater security. Moving data would create a moving target that could help to limit an adversary's ability to launch a targeted attack. This strategy might help to mitigate the effectiveness of the types of slow attacks discussed in section 2.5.

3. STORAGE SYSTEMS

In this section we examine a number of storage systems and how they deal with the security threats that have been outlined in the previous section. The sampling of systems presented below should not be considered an exhaustive list but rather a representation of the diversity within the storage field. Most of these systems are not specifically archival systems but they provide a useful perspective on the shortcomings of traditional storage systems in long-term roles. Examining the appropriateness of each system for the task of long-term secure storage illustrates the need to design a storage systems expressively for the purpose of archival storage. The capabilities of each system with respect to the discussion presented in Section 2 are summarized in Table 1.

3.1 FreeNet

FreeNet [4] is a peer to peer distribution system which in many ways is a stark contrast to archival storage. While archival storage is concerned with the long-term persistence of data, FreeNet is expressively created for distributing content and makes little effort to maintain its contents' persistence.

One of the primary goals of FreeNet is to allow anonymous distribution of data. It utilizes encryption over all stored files but this is primarily to provide a host with plausible deniability over their contents and not for secrecy.

FreeNet relies on local data-stores and dynamic routing tables to locate data. File migration is handled in a similar fashion. As requests travel back through the system the requested data is copied to each host along the request path. Thus hot data is quickly replicated while the space occupied by cold data is eventually reclaimed. This is clearly not conducive to long-term storage. Additionally, integrity in FreeNet is only assured through hashing of short strings which accompany the data. These are susceptible to dictionary attacks due to their short length.

3.2 OceanStore

Oceanstore [9] is a global, persistent distributed storage system, which stores data across sets of untrusted nodes. The system's objective is to provide an all-in-one, secure, highly- available, global

	Secrecy	Authorization	Integrity	Slow Attacks	Migration
FreeNet	encryption	none	hashing		access based
OceanStore	encryption	signatures	versioning		access based
FarSite	encryption	certificates	merkle trees		continuous relocation
PAST	encryption	smart-cards	immutable files		
Publius	encryption	password (delete)	retrieval based		
SNAD / Plutus	encryption	encryption	hashing		
GridSharing	secret sharing		replication		
PASIS	secret sharing		repair agents, auditing		
CleverSafe	secret sharing		high replication degree		
POTSHARDS	secret sharing	pluggable	algebraic signatures		
LOCKSS	none		vote based checking		site crawling
Glacier		node auth.	signatures		
Venti			retrieval		

Table 1: Capability overview of a variety of storage systems. Each of the systems discussed is listed along with a brief description of the mechanism used to provide the stated aspect of long-term security.

storage architecture. Oceanstore is designed around the assumption that all of the nodes are untrusted and failure is inevitable. Data protection is achieved through encryption and replication.

3.3 FarSite

Farsite [1] was designed to serve the same function as a centralized file server. It utilizes a distributed architecture that attempts to utilize the unused resources across a network of loosely coupled, insecure and unreliable machines.

Secrecy in Farsite is accomplished through the explicit use of encryption. At file creation a symmetric key is generated and encrypted with the public-keys of users authorized to access the file.

Locating data involves communicating with a directory group member. These nodes maintain the directory structure for a virtual hierarchy. In Farsite, files are presented in a hierarchical view but there can be multiple views of the data. Each view has its own root maintained by a set of directory root members. For long-term storage this makes locating data vulnerable to the failure of the directory group members. If there is an unlimited allowable number of roots however, it may be possible for each client to maintain its own view of the system.

Reliability and integrity come from replication and the use of Merkle trees respectively. File migration is also provided by the replication mechanism.

3.4 PAST

The PAST [6] system utilizes a overlay network to connect peers across the Internet with the aim of providing persistent storage with strong security. PAST achieves a high level of secrecy through the use of encryption on the client and authorization is achieved through judicious use of certificates. To facilitate this, PAST utilizes smart-cards to assist with the certificate operations. In addition to the usual concerns with long-lived encryption, the reliance on specialized hardware raises other concerns for long-term preservation. The authors do state that the role of smart-cards could be performed by trusted services.

Read requests are based on fileIds and routed using the Pastry [15] routing and location scheme. The smart-card is used in many aspects of this process. From naming integrity and quota management the smart-card is an essential token for system interaction. While convenient, for the long-term viability of this solution there must be a procedure for dealing with a lost smart-card.

Integrity and persistence in PAST is achieved through a two level approach. The first level attempts to prevent unwanted changes by making all data in the system immutable. The second level is through the randomized replication of data.

3.5 Publius

Publius [20] is another publishing system that gives the user a familiar URL based interface to system contents. While it is used to publish content it still utilizes encryption over the contents of the file as well secret splitting to manage the keys. Passwords are used to control the deletion and updating of contents. As discussed earlier, this reliance on encryption and passwords is a source of concern for long-term storage. The concern is somewhat mitigated as retrieval is limited to parsing a URL and the password is only used for deleting and changing contents. If the password is lost, the worst case scenario is that the data becomes effectively immutable.

In the Publius system, integrity is checked during retrieval. With a long-term archival system there are no guarantees for how frequently data will be accessed so this would be another area of concern. One possible solution would be to have an automated system which requests data for the sole purpose of insuring data integrity.

3.6 SNAD and Plutus

Secure Network-Attached Disk (SNAD) [12] was designed for secure storage. As such, it does not include any facilities for long-term archival usage beyond those present in many workstation-type file systems. Data location is straightforward; SNAD encrypts files individually, so any system that can store directory information in files, such as FFS [11] and ext3 [19] has location facilities.

Secrecy in SNAD is ensured using strong symmetric cryptography, with authorization accomplished using a public key infrastructure. Integrity in SNAD is ensured when the data is read through the use of hashes. Only a user that can decrypt the data can verify the hash, however, making it difficult to use SNAD for archival storage.

Since SNAD is intended for workstation file system use, it has few mechanisms aimed to ensure long-term data survival. For example, there is no mechanism to recover a lost private key. Similarly, there is no defense against “slow attacks;” however, it is unlikely that a slow attack would succeed because SNAD is only vulnerable to discovery of users’ private keys.

Other workstation-type secure file systems, such as Plutus [8], have similar properties with respect to long-term data survival. They ensure that data will never be revealed without the appropriate key, but are not appropriate for long-term data storage because they do

not address issues of key loss, algorithm compromise, and data longevity.

3.7 GridSharing and CleverSafe

The work of Subbiah and Bough [18] utilizes secret sharing to build a secure and fault tolerant data storage. While their system will function with a variety of secret sharing algorithms, the GridSharing system is designed for low-latency access and thus their testing shows that XOR secret sharing is the only viable algorithm. While the use of secret sharing provides secrecy without the need for encryption, the GridSharing system shows that its use may be limited in low-latency storage. The XOR algorithm in its basic form is an *m* of *m* scheme. This may be a problem in long-term storage since the data could not survive the loss of any of the secret shares.

Integrity in GridSharing is achieved through the use of replication. Using XOR based secret splitting would require very high levels of replication as each secret share is required for accurate reconstruction and thus the loss or corruption of any single share could compromise the entire file.

CleverSafe [5] has similar goals for data storage, protecting data using a custom-designed information dispersal algorithm to generate shares. As with GridSharing, CleverSafe provides little functionality to rebuild lost shares. Thus, CleverSafe cannot store data for long periods of time without extensive user intervention to ensure that sufficient data shares survive for long periods of time.

Both GridSharing and CleverSafe suffer from the rebuilding problem that, in order to recover lost shares, the system must first rebuild the data that contains the missing shares. While this makes system implementation easier, it reduces security and longevity by exposing the system to long-term decay.

3.8 PASIS

The PASIS [22] architecture is centered around providing a highly-available, fault-tolerant, secure storage system. Secrecy and redundancy is provided using a general threshold sharing scheme such as Shamir's secret sharing scheme. Since the shares are placed across storage nodes, an intruder would have to compromise several storage nodes in order to compromise data secrecy. PASIS relies on a directory service to translate file objects into the object shares and their respective location.

Integrity checking and correction is provided in PASIS through the use of a repair agent on each storage node. The status of each archive is actively monitored as part of the system's aggressive self-maintenance features. Additionally, as each request is considered suspect, PASIS also employs a system of versioning and request auditing. Audit logs allow the system to roll back any changes committed by a malicious user within a given window. This still raises the issue of how long to maintain the audit logs. In long-term archival storage an intruder may be able to space out the changes in a such a way that by the time the change is detected, the audit logs required to undo the damage are no longer available.

Unlike CleverSafe and GridSharing, PASIS can rebuild lost shares in a secure way [21]. However, this approach is computationally intensive and still may leak some information if sufficiently many servers are compromised; the number of servers that must be compromised is lower than the number of servers required to rebuild the data.

3.9 POTSHARDS

The POTSHARDS [17] system shifts data secrecy from encryption to authentication by using secret splitting to store secret shares across multiple authentication domains. Additionally, with proper authorization it is possible for the archives to collude and reconstruct any user's data or even all data stored in the system.

Before distributing shares across a set of archives, files are transformed by splitting the file into fragments using secret sharing algorithms. User- level redundancy is provided by a second level of secret-splitting optimized for redundancy. The result is a set of shards which are stored across a disjoint set of erasure encoded failure domains providing long-term, system-level redundancy. The shards in a lost archive can be rebuilt without revealing data because there is no way for any archive to discover which shards make up which objects.

Each client is responsible for storing and preserving an index over its own files. The user index is stored in the system so that it can be reconstructed given proper authorization should it become lost or corrupted. A system-level index holds a shard to archive mapping along with the information needed to reconstruct failed archives.

Each archive in the POTSHARDS system is responsible for maintaining the integrity of its own data. Archives check the integrity of their data by storing a cryptographic hash over sets of shards. In order to ensure each archive is actively performing integrity checks, an outside party can perform distributed integrity checks using algebraic signatures.

3.10 LOCKSS

LOCKSS [10] is a content distribution system designed for libraries that mimics the behavior of a web cache. As the purpose of LOCKSS is to ensure public access to data, it specifically does not include a secrecy aspect for securing the contents of files.

The data location aspects of LOCKSS are in keeping with its web-cache like behavior. Since the data appears to originate from its originally published source, the system does not present its own index to the user. Instead, it relies upon existing indexing systems and web content directories.

Integrity guarantees are provided through a voting mechanism. The system's contents are organized in archival units and a policy of voting uses cooperating systems to check the integrity of its AUs and repair any damage that may occur.

As with integrity checking, file replication is an active process within LOCKSS. In a three step process, nodes collect newly published data from journal websites, distribute the data by acting as a proxy cache for local requests and preserve their contents though the voting procedure with other LOCKSS systems.

3.11 Glacier

Glacier [7] is a decentralized storage system that relies on a large number of replicas to insure availability. It was designed based on for environment described by Bolosky [2] which found an expected node lifetime of 290 days.

There is no specific encryption mechanism for file secrecy in Glacier, so it avoids the problems discussed earlier. However is does utilize signed manifests for insuring integrity and thus still uses cryptographic primitives. In effect, while there is no specific secrecy policy, it still suffers from the problems introduced by long-term encryption.

Data location in Glacier is based on a circular naming structure and data location algorithm. This has the advantage of relieving the client from the onus of maintaining an index. The system was designed with large-scale correlated failure in mind and thus this technique may be more effective than the use of indices.

3.12 Venti

Venti [13] is a system specifically designed for archival storage. Key to the system is content-addressable storage. This provides built-in consistency checking information and enforces an immutable data policy that fits into the archival model. One concern with content-addressable storage is that users must have a way of remembering exactly what they are looking for. Put another way, content-addressable storage is great for filing but bad for finding. Some, such as Howard Besser, may argue that searching on metadata is a better fit for archival storage.

Integrity in Venti is performed primarily at retrieval time. Both the client and server are able to perform an integrity check as they are able to compute the fingerprint of the data and compare it to the request fingerprint.

Venti's properties make a it good candidate to act as the storage layer in a archival system, but as a stand-alone system it does not address the full needs of long-term storage. Functionality that deals with archive loss, and data migration might be implemented in higher layers.

4. CONCLUSION

Long-term archival storage systems introduce integrity, authentication and privacy threats that do not generally exist in non-archival storage systems. We have presented a general model for secure long-term storage systems and a set of threats that may lead to system compromise. The focus of this paper was not to solve the many problems that may arise in long-term system, but rather enumerate potential threats. We have presented new threats specific to the long-term storage problem and existing threats from the perspective of long-term storage.

In an effort to motivate the need for storage systems specifically tailored for the archival storage model, we have examined how a variety of existing systems deal with the threats to long-term data survivability. While many of these systems are still being developed and modified none of them specifically addressed all of the security concerns. Of greater concern is that there are threats which none of the system addressed. In particular the risk from slow compromises is an area that must be addressed in future archival storage systems. To deal with hardware changes, migration will also be an area that a long-term system must be able to accommodate. This demonstrates the need for storage systems that are specifically designed for the write-once, read-maybe usage model and long-data lifetimes found in archival storage.

An important aspect of documenting and discussing these threats to long term secure storage is that is has assisted us in the design of our own archival storage system. Our hope is that by listing the threats that such a system must contend with, future efforts to build secure archival storage systems will be more focused and complete.

Acknowledgments

We thank Kaladhar Voruganti and the other members of the Storage Systems Research Center (SSRC) for spirited discussions that helped focus the content of this paper. We also thank the sponsors of the SSRC, including Los Alamos National Lab, Livermore National Lab, Sandia National Lab, Hewlett-Packard Laboratories, IBM Research, Intel, Microsoft Research, Network Appliance, Rocksoft, Symantec, and Yahoo.

5. REFERENCES

[1] ADYA, A., BOLOSKY, W. J., CASTRO, M., CHAIKEN, R., CERMAK, G., DOUCEUR, J. R., HOWELL, J., LORCH, J. R., THEIMER, M., AND WATTENHOFER, R. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)* (Boston, MA, Dec. 2002), USENIX.

[2] BOLOSKY, W. J., DOUCEUR, J. R., ELY, D., AND THEIMER, M. Feasibility of a serverless distributed file system deployed on an existing set of desktop pcs. In *sigmetrics00* (2002), pp. 33 –43.

[3] BRAND, S. *The Clock of the Long Now*, new york, ny ed. Basic Books, 1999.

[4] CLARKE, I., SANDBERG, O., WILEY, B., AND HONG, T. W. Freenet: A distributed anonymous information storage and retrieval system. *Lecture Notes in Computer Science 2009* (2001), 46+.

[5] CLEVERSAFE. Highly secure, highly reliable, open source storage solution. Available from http://www.cleversafe.org/, June 2006.

[6] DRUSCHEL, P., AND ROWSTRON, A. PAST: A large-scale, persistent peer-to-peer storage utility. In *Proceedings of the 8th IEEE Workshop on Hot Topics in Operating Systems (HotOS-VIII)* (2001), pp. 75–80.

[7] HAEBERLEN, A., MISLOVE, A., AND DRUSCHEL, P. Glacier: Highly durable, decentralized storage despite massive correlated failures. In *Proceedings of the 2nd Symposium on Networked Systems Design and Implementation (NSDI)* (Boston, MA, May 2005), USENIX.

[8] KALLAHALLA, M., RIEDEL, E., SWAMINATHAN, R., WANG, Q., AND FU, K. Plutus: scalable secure file sharing on untrusted storage. In *Proceedings of the Second USENIX Conference on File and Storage Technologies (FAST)* (San Francisco, CA, Mar. 2003), USENIX, pp. 29–42.

[9] KUBIATOWICZ, J., BINDEL, D., CHEN, Y., EATON, P., GEELS, D., GUMMADI, R., RHEA, S., WEATHERSPOON, H., WEIMER, W., WELLS, C., AND ZHAO, B. OceanStore: An architecture for global-scale persistent storage. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Cambridge, MA, Nov. 2000), ACM.

[10] MANIATIS, P., ROUSSOPOULOS, M., GIULI, T. J., ROSENTHAL, D. S. H., AND BAKER, M. The LOCKSS peer-to-peer digital preservation system. *ACM Transactions on Computer Systems 23*, 1 (2005), 2–50.

[11] MCKUSICK, M. K., JOY, W. N., LEFFLER, S. J., AND FABRY, R. S. A fast file system for UNIX. *ACM Transactions on Computer Systems 2*, 3 (Aug. 1984), 181–197.

[12] MILLER, E. L., LONG, D. D. E., FREEMAN, W. E., AND REED, B. C. Strong security for network-attached storage. In *Proceedings of the 2002 Conference on File and Storage Technologies (FAST)* (Monterey, CA, Jan. 2002), pp. 1–13.

[13] QUINLAN, S., AND DORWARD, S. Venti: A new approach to archival storage. In *Proceedings of the 2002 Conference on File and Storage Technologies (FAST)* (Monterey, California, USA, 2002), USENIX, pp. 89–101.

[14] RIEDEL, E., KALLAHALLA, M., AND SWAMINATHAN, R. A framework for evaluating storage system security. In *Proceedings of the 2002 Conference on File and Storage Technologies (FAST)* (Monterey, CA, Jan. 2002).

[15] ROWSTRON, A., AND DRUSCHEL, P. Pastry: Scalable, distributed object location and routing for large-scale

peer-to-peer systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms* (Heidelberg, Germany, Nov 2001), pp. 329–350.

[16] SCHWARZ, S. J., T., AND MILLER, E. L. Store, forget, and check: Using algebraic signatures to check remotely administered storage. In *Proceedings of the 26th International Conference on Distributed Computing Systems (ICDCS '06)* (Lisboa, Portugal, July 2006), IEEE.

[17] STORER, M., GREENAN, K., MILLER, E. L., AND MALTZAHN, C. POTSHARDS: Storing data for the long-term without encryption. In *Proceedings of the 3rd International IEEE Security in Storage Workshop* (Dec. 2005).

[18] SUBBIAH, A., AND BLOUGH, D. M. An approach for fault tolerant and secure data storage in collaborative work environements. In *Proceedings of the 2005 ACM Workshop on Storage Security and Survivability* (Fairfax, VA, Nov. 2005), pp. 84–93.

[19] TWEEDIE, S. EXT3, journaling file system, July 2000.

[20] WALDMAN, M., RUBIN, A. D., AND CRANOR, L. F. Publius: A robust, tamper-evident, censorship-resistant, web publishing system. In *Proceedings of the 9th USENIX Security Symposium* (Aug 2000), pp. 59–72.

[21] WONG, T. M., WANG, C., AND WING, J. M. Verifiable secret redistribution for threshold sharing schemes. Tech. Rep. CMU-CS-02-114-R, Carnegie Mellon University, Oct. 2002.

[22] WYLIE, J. J., BIGRIGG, M. W., STRUNK, J. D., GANGER, G. R., KILIÇÇÖTE, H., AND KHOSLA, P. K. Survivable storage systems. *IEEE Computer* (Aug. 2000), 61–68.

[23] XIN, Q., SCHWARZ, T. J. E., AND MILLER, E. L. Disk infant mortality in large storage systems. In *Proceedings of the 13th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS '05)* (Atlanta, GA, Sept. 2005), IEEE.

[24] ZANIN, G., MEI, A., AND MANCINI, L. V. A secure and efficient large scale distributed system for data sharing. In *Proceedings of the 26th International Conference on Distributed Computing Systems (ICDCS '06)* (Lisboa, Portugal, July 2006), IEEE.

Secure Data Deduplication

Mark W. Storer Kevin Greenan Darrell D. E. Long Ethan L. Miller
Storage Systems Research Center
University of California, Santa Cruz
{mstorer,kmgreen,darrell,elm}@cs.ucsc.edu

ABSTRACT

As the world moves to digital storage for archival purposes, there is an increasing demand for systems that can provide secure data storage in a cost-effective manner. By identifying common chunks of data both within and between files and storing them only once, deduplication can yield cost savings by increasing the utility of a given amount of storage. Unfortunately, deduplication exploits identical content, while encryption attempts to make all content appear random; the same content encrypted with two different keys results in very different ciphertext. Thus, combining the space efficiency of deduplication with the secrecy aspects of encryption is problematic.

We have developed a solution that provides both data security and space efficiency in single-server storage and distributed storage systems. Encryption keys are generated in a consistent manner from the chunk data; thus, identical chunks will *always* encrypt to the same ciphertext. Furthermore, the keys cannot be deduced from the encrypted chunk data. Since the information each user needs to access and decrypt the chunks that make up a file is encrypted using a key known only to the user, even a full compromise of the system cannot reveal which chunks are used by which users.

Categories and Subject Descriptors

D.4.6 [**Security and Protection**]: Access controls; H.3 [**Information Systems**]: Information Storage and Retrieval

General Terms

Design, Security

Keywords

secure storage, encryption, cryptography, deduplication, capacity optimization, single-instance storage

1. INTRODUCTION

Businesses and consumers are becoming increasingly conscious of the value of secure, archival data storage. In the business arena, data preservation is often mandated by law [16, 26], and data mining has proven to be a boon in shaping business strategy. For individuals, archival storage is being called upon to preserve sentimental and historical artifacts such as photos, movies and personal documents. Further, while few would argue that business data calls for security, privacy is equally important for individuals; data such as medical records and legal documents must be kept for long periods of time but must not be publicly accessible.

Paradoxically, the increasing value of archival data is driving the need for cost-efficient storage; inexpensive storage allows the preservation of all data that *might* eventually prove useful. To that end, deduplication, also known as single-instance storage, has been utilized as a method for maximizing the utility of a given amount of storage [4, 38, 5]. Deduplication identifies common sequences of bytes both within and between files ("chunks"), and only stores a single instance of each chunk regardless of the number of times it occurs. By doing so, deduplication can dramatically reduce the space needed to store a large data set.

Data security is another area of increasing importance in modern storage systems and, unfortunately, deduplication and encryption are, to a great extent, diametrically opposed to one another. Deduplication takes advantage of data similarity in order to achieve a reduction in storage space. In contrast, the goal of cryptography is to make ciphertext indistinguishable from theoretically random data. Thus, the goal of a secure deduplication system is to provide data security, against both inside and outside adversaries, without compromising the space efficiency achievable through single-instance storage techniques.

To this end, we present two approaches to secure deduplication: authenticated and anonymous. While the two models are similar, they each offer slightly different security properties. Both can be applied to single server storage as well as distributed storage. In the former, single server storage, clients interact with a single file server that stores both data and metadata. In the later, metadata is stored on an independent metadata server, and data is stored on a series of object-based storage devices (OSDs).

Both models of our secure deduplication strategy rely on a number of basic security techniques. First, we utilize convergent encryption [10] to enable encryption while still allowing deduplication on common chunks. Convergent encryption uses a function of the hash of the *plaintext* of a chunk as the encryption key: any client encrypting a given chunk will use the same key to do so, so identical plaintext values will encrypt to identical ciphertext values, regardless of who encrypts them. While this technique does leak knowledge that a particular ciphertext, and thus plaintext, already exists, an adversary with no knowledge of the plaintext cannot deduce the key from the encrypted chunk. Second, all data chunking and encryption occurs on the client; plaintext data is never trans-

StorageSS'08, October 31, 2008, Fairfax, Virginia, USA.

mitted, strengthening the system against both internal and external adversaries. Finally, the map that associates chunks to a given file is encrypted using a unique key, limiting the effect of a key compromise to a single file. Further, the keys are stored within the system in such a way that users only need to maintain a single private key regardless of the number of files to which they have access.

The remainder of this paper is organized as follows. In Section 2, we place our system within the context of the field's related work. Section 3 describes the threat model, which forms the basis of our design. In addition, Section 3 defines the assumptions, storage model, notation, and players in our secure, deduplication system. Section 4 provides a detailed description of how our system achieves improved storage utilization through deduplication, while providing data security. Section 5 provides an analytical examination of our system, including an evaluation of its security in a variety of scenarios. Finally, we conclude in Sections 6 and 7 with our future plans for this system and a short summary of our work.

2. RELATED WORK

Current systems that utilize single instance storage rely upon one of three primary deduplication strategies: whole file, fixed-sized chunks, and variable-sized chunks. The first, whole file, typically utilizes a file's hash value as its identifier. Thus, if two or more files hash to the same value, they are assumed to have identical contents and only stored once (not including redundant copies). This form of *content addressable storage* (CAS) is used in the EMC Centera system [14]. Farsite [10] and the Windows Single Instance Store [6] also perform deduplication on a per-file bases, though both use traditional identifiers and handle deduplication using a separate data structure. The second type of deduplication, per-block deduplication, is exemplified by the Venti archival storage system [27]. In Venti, files are broken into fixed sized blocks before deduplication, so files that share some identical contents (but not all), may still yield storage savings. The third, and most flexible form, breaks files into variable-length "chunks" using a hash value on a sliding window; by using techniques such as Rabin fingerprints [28], chunking can be done very efficiently. Variable-length chunks are used in LBFS [25], Shark [4], and Deep Store [38].

Many distributed file systems, such as OceanStore [29], SNAD [24], Plutus [20], and e-Vault [19], address file secrecy through the use of keyed encryption. The use of cryptographic techniques in these systems range from the assumption that all incoming data is already encrypted, to central architecture elements that define the system. However, none of these systems attempt to achieve the storage efficiency that is possible through deduplication. High-performance distributed file systems such as the Panasas Parallel File System [37], Ceph [35], and Lustre [7] typically have much less security than "standard" distributed file systems, trading higher performance for lower security. While there is an effort to add greater security to Ceph [21], this effort only involves authentication, not encryption.

At the opposite end of systems that provide secure, deduplicated storage efficiency, some systems utilize security models that incur a storage overhead. For example, PASIS [13] and POTSHARDS [33], achieve secure storage through secret sharing [31]. While well suited to long-term security, this technique incurs a very high storage overhead. Similarly, steganographic systems, such as the Steganographic File System [3] and Mnemosyne [15], provide plausible deniability over storage contents through the use of random data blocks. In both secret sharing and steganography, the storage overhead can be many times the size of the plaintext data.

In addition to data secrecy, several systems have addressed the demand for anonymity. Especially in the area of content distribution, there is a desire for systems that can hide the identity of data hosts, publishers and readers. For example, Publius uses encrypted data and secret sharing over keys to provide a censorship resistant web publishing platform that provides a high degree of writer anonymity [34]. Data encryption also provides the storage host with a level of plausible deniability; as there is no clear owner, a node's operator can claim that they have no knowledge about the plain-text data stored on their node [9].

Of all of these file systems, only Farsite combined deduplication with security. In its original design, its goal was to harness the unused disk space in a network of desktop-class computers, and present it as though it were a central file server [1]. In the original implementation, security was provided through file encryption where each user utilized a combination of symmetric and asymmetric keys. An extension of the work was an attempt to achieve better space efficiency through duplicate file coalescing [10]. To this end, the authors developed *convergent encryption*, in which the hash of the data is used as the encryption key. This allows users to independently encrypt identical plaintexts to the same ciphertext. However, unlike our work, Farsite only coalesces at the level of entire files. Our system coalesces data at a sub-file level, thus achieving space savings with files that are merely similar, as opposed to identical. Additionally, in the Farsite design, the client generates the encrypted value and its identifier. We show that if the key/value store for deduplicated data is not verified, its contents may be susceptible to targeted collision attacks. Finally, we present a model that allows secure, deduplicated storage in an anonymous user scenario.

3. THREAT MODEL

In order to properly design and evaluate a secure storage system, the threat model must be clearly laid out. In this section we identify the adversaries present in our model, our assumptions, and the attacks that must be considered in a secure deduplication system.

As part of establishing our security model, it is important to establish a consistent notation. There are two primary cryptographic functions that we utilize: encryption and hashing. An encryption function takes two parameters and is denoted $e(K,P) = C$, where K is the encryption key, P is the plaintext and C is the ciphertext. An encryption function has a corresponding decryption function that uses a key and ciphertext to recover the original plaintext, expressed as $d(K,C) = P$.

Hashing is expressed in a manner similar to encryption. Simple hashing that does not utilize an encryption key is expressed as the single parameter function, $hash(P) = H$. Generating a hash of plain-text P using a hash function and the encryption key K_i, known as an HMAC (keyed-Hash Message Authentication Code), provides integrity as well as message authentication. We express this function with the following notation: $HMAC_i(P)$.

3.1 Assumptions

One of the most fundamental assumptions that we make is that encrypted data is effectively random. The implication, in regards to deduplication, is that random data yields very low storage gains. We support this claim by examining the storage utilization of a system where each user encrypts data with their own distinct keys and the system deduplicates the encrypted data. For brevity, we assume the encrypted data is divided into fixed-sized chunks, though a similar argument can be made for variable-length chunks. At any point in time, the system is storing k logical chunks of of length l. Each

chunk can take on any one of $2^l = m$ values. We recursively derive the physical storage utilization in chunks, s_k, as

$$
\begin{aligned}
s_1 &= 1 && (1)\\
s_k &= s_{k-1} + \left(1 - \frac{s_{k-1}}{m}\right) && (2)\\
&= s_{k-1}\left(1 - \frac{1}{m}\right) + 1 && (3)\\
&= \sum_{i=0}^{k-1}\left(1 - \frac{1}{m}\right)^i && (4)\\
&= m^{1-k}(m^k - (m-1)^k) && (5)
\end{aligned}
$$

If a single chunk exists in the system, then the number of logical and physical chunks is equal, thus our base case is $s_1 = 1$. Assuming a uniform distribution over encrypted chunks, the second chunk will match with probability $1/m$. In general, the k-th chunk matches with probability s_{k-1}/m. If $k \gg m$ then s_k is very close to m, which results in a great deal of deduplication. Unfortunately, when $k \gg m$, performance may suffer and the size of file indices can become unwieldy. In practice, we find that $m \gg k$ and the utility of deduplication is extremely small because $s_k \approx k$. From this, we must conclude that deduplication of traditionally-encrypted data is largely ineffective. Since we are assuming that encrypted data is random, we can model deduplication of random data using the equation above. The chunk signature will operate on this random data and so should, if it is a good hashing function, be uniformly distributed as well. For chunks of non-trivial size—more than a few bytes—the likelihood of a match is extremely small and so, with extremely rare exceptions, every chunk will be unique and must be stored in its entirety.

Our second assumption is that encryption provides an adequate level of security for relatively short archival scenarios: if the data's lifetime is on the order of a few years, an attacker with access to ciphertext generated by a modern cryptosystem will be unable to determine the encryption key or derive the corresponding plaintext value. We recognize that in very long-term scenarios, on the order of decades, this assumption may not hold [32]. Extending this work into the secure, long-term area may be pursued as future work.

Next, we assume that an adversary that can sufficiently imitate a user has access to that user's data. In other words, if a malicious user has acquired enough information about the user—user names and passwords, for example—to participate in the system's protocols, then that user will obtain the standard results of that protocol. This scenario holds true in almost every secure system.

Because our solution utilizes hash functions in the generation of key material, we assume that they are cryptographically secure. More specifically, we assume that they are both weakly and strongly collision resistant. The former states that finding two input values that hash to the same output value is an intractable problem. The later, states that given a hash value, finding a value that hashes to the same output value is intractable.

Archival storage is typically used as a write-once, read-maybe store; thus, it stresses throughput rather than low-latency performance. Most existing large-scale deduplication systems are used as archival stores [27, 38]; the systems that are not often exhibit infrequent writes because they are well-suited for read-heavy workloads such as software distribution [4]. This usage pattern is quite different from the top storage tier of a hierarchical storage solution that stresses low-latency access and frequent writes, and also differs from backup solutions whose sole goal is high throughput writes, with reads an option of last resort. We assume that this emphasis on throughput allows the system to accommodate a reasonable latency penalty.

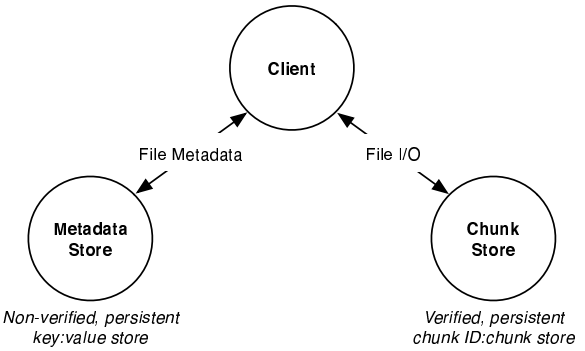


Figure 1: The three primary players in the storage model and their interactions. In a distributed storage system, the responsibilities of the metadata server and chunk server are handled by separate clusters of systems. In a single-server model, the metadata and chunk server are on the same system.

The data lifetimes we are considering are assumed to be on the order of years, not decades. While this is longer than the file lifetime often encountered in front-line storage [2, 22, 30], it is not as long as the indefinite lifetimes that other secure, archival systems are designed to support [33].

3.2 Players

As Figure 1 shows, at the protocol level, there are three primary players in our storage model: the client, metadata store, and chunk store. This arrangement maps to both single-server and distributed storage architectures. In a single-server architecture, the metadata store and chunk store are located on the same system, while a distributed storage system might choose to disconnect the metadata store and chunk server, handling the duties of each in separate clusters [35, 37].

Users interact with the system through the *client*, which is the starting point for both ingestion and extraction. As Figure 1 illustrates, it is the central contact point between the other components in the storage model. Unlike the other components in the storage model, the client does not have any persistent storage requirements, though the system assumes that users have reliable, secure access to their keys.

The *metadata store* is responsible for maintaining the information that users require in order to rebuild files from chunks — such as maps and encryption keys. We model this persistent storage using a simple, unverified key:value architecture. In such a system, when the user submits a key:value pair to the metadata server, the server does not need to verify that the key correctly corresponds to the value. For example, if the key is the hash of the value, the server does not need to verify that the hash of the value is the same as the key that the user submitted.

The role of the third player, the *chunk store*, is to persistently store data chunks, and to fulfill requests for chunks based on their ID. The chunk store is also modeled as a key:value store, however, unlike the metadata store, the chunk store must be able to verify the correctness of a the key with regards to the value. This is due to the possibility of targeted-collision attacks, as described below, that are possible within the chunk store.

In a deduplicated chunk store, a targeted-collision attack could be used to associate a false value with a given key. The pivotal difference between random collisions and targeted collisions is that a

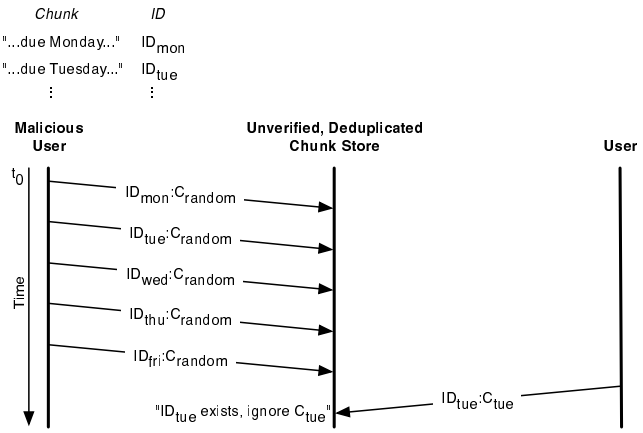


Figure 2: Targeted-collision attack in which a malicious user exploits predictable data (in this example, a form letter with a due date) to generate valid chunk IDs, and associate those IDs with invalid chunks. If the user is the first to submit the ID, subsequent chunks will be deduplicated to a garbage value.

user can exploit the predictable content of some data — in Fig 2 the malicious user utilizes similarities in form letters — to generate valid chunk identifiers. If an adversary can be the first to submit those identifiers with a garbage chunk, and if the chunk store cannot verify the correctness of the identifiers, subsequent submissions that have the same identifier will be deduplicated to the garbage chunk.

In addition to the three players of the storage model, our system identifies two adversaries, identified by their relationship to the system: external and internal. The external attacker exists outside of the system. This adversary does not have even simple insider access, such as a user account, and can only intercept messages or attempt to compromise a user's account.

In contrast, the internal attacker, or malicious insider, does have at least limited inside access. These attackers are further defined by their level of access, ranging from a simple user account access, to privileged root level access (as might be held by a malicious administrator). The existence of internal attackers implies that neither the metadata store, nor the chunk store are assumed to be trustworthy. Section 5 explores this implication by examining the security threat posed by internal adversaries.

The goal in both of the security models we present is to provide the users with a level of data protection from both external and internal attackers, regardless of the adversaries access level (or in the very least reduce the amount of data lost in a compromise). Each model provides an additional set of security features.

4. SYSTEM DESIGN

In this section we describe our two primary secure deduplication models: authenticated and anonymous. While similar, as Table 1 summarizes, each model offers a slightly feature set. We start by describing the security features, and then proceed to introduce the basic design of our secure deduplication techniques. Finally, we present the specifics of the two models. In particular we describe the contents of the metadata and chunk store for each, as well as their respective ingestion and extraction procedures.

The security property most associated with encryption is *secrecy*, which states that only authorized users are able to read plaintext data. Often, authorization is handled through key distribution; if a user is able to legitimately acquire the proper keys, she can access and decrypt the data. Both of the models that we present offer data secrecy against both external and internal adversaries.

	Authenticated	Anonymous
Data secrecy	Yes	Yes
Anonymity	No	Yes
Per user revocation	Yes	No
Storage mode	Mutable	Immutable

Table 1: Security feature-set offered by our two secure deduplication models: anonymous, and authenticated. Note that anonymity is incompatible with per-user revocation.

Anonymity allows the identity of a user to be hidden. This feature has two facets. The first is anonymity with respect to users submitting requests, *i. e.*, read and write requests cannot be attributed to a particular user. The second facet is anonymity with respect to storage contents, which states that the system is unable to determine which data is owned or accessible by a particular account.

Revocation is the ability to remove a user's access to a given file. In order for this to be done at a fine granularity, as in a per-user revocation, the system must include authentication; per-user revocation obviously cannot exist in a system without knowledge of a user's identity. Revocation schemes can be described by the action that takes place at the time of the revocation. In active revocation, access is immediately removed. This is often expensive, and may involve a fair amount of cryptographic computation. In lazy revocation, access is only removed when the data is changed. Thus, the user is unable to see any changes that occur after the revocation, but may have continuing access to what they were previously entitled to view.

4.1 Secure Deduplication Overview

In both the anonymous and authenticated models, clients begin the ingestion process by transforming a file into a set of chunks. This is often accomplished using a content-based chunking procedure which produces chunks based on the contents of the file. The advantage of this approach is that it can match shared content across files even if that content does not exist at the multiple of a given, fixed offset [25]. The algorithm selects chunks based on a threshold value A and a sliding window of width w that is moved over the file. At each position k in the file, a fingerprint, $F_{k,k+w-1}$, of the window's contents is calculated [28]. If $F_{k,k+w-1} > A$, then k is selected as a chunk boundary. The result is a set of variable sized chunks, where the boundary between chunks is based on the content of the data.

Both file chunking and encryption occur on the client. There are a number of benefits to performing these tasks on the client, as opposed to the server. First, it reduces the amount of processing that must occur on the server. Second, by encrypting chunks on the client, data is never sent in the clear, reducing the effectiveness of many passive, external attacks. Third, a privileged, malicious insider would not have access to the data's plaintext because the server does not need to hold the encryption keys.

Clients encrypt chunks using *convergent encryption*, which was introduced in the Farsite system [10]. Using this approach, clients use an encryption key deterministically derived from the plaintext content to be encrypted; both Farsite and our system use a cryptographic hash of the plaintext as the key. Since identical plaintexts result in the use of identical keys, regardless of who does the encryption, a given plaintext always results in the same ciphertext.

$$K = hash(\text{chunk}) \tag{6}$$

Compared to other approaches, this strategy offers a number of advantages. As we have shown in Section 3, if each user encrypted using his own key, the amount of storage space saved through deduplication would be greatly reduced because the same chunk encrypted using two different keys would be would result in different ciphertext (with very high probability). Second, attempting to share a random key across several user accounts introduces a key sharing problem. Third, a user that does not know the data plaintext value cannot generate the key, and therefore cannot obtain the plaintext from the ciphertext. This point is especially important since, in contrast to an approach where the server encrypts the data, even a root level administrator does not have access to a chunk's plaintext value without the key.

The primary security disadvantage of this approach, as identified in its original description [10], is that it leaks some information. In particular, convergent encryption reveals if two ciphertext strings decrypt to the same plaintext value. However, this behavior is necessary in systems that use deduplication, since it allows a system to remove duplicate plaintext data chunks while only observing the ciphertext; information leakage is part of the compromise needed to achieve space-efficiency through deduplication.

Each ciphertext chunk must be assigned an identifier. In our system, each chunk in the system is identified using the encrypted chunk's hash value, a technique sometimes referred to as content-based naming.

$$\text{chunk\_id} = hash(e(hash(\text{chunk}), \text{chunk})) \tag{7}$$

An alternative to using the hash of the encrypted chunk is to use the hash of the hash of the plain-text chunk, *i. e.*, the hash of the encryption key is the chunk identifier. This approach offers a number of attractive qualities. First, performance is improved. In both approaches the user performs two hashes: a key generation hash, and an identifier generation hash. Assuming that key lengths are smaller than chunk lengths, performing two chunk hashes will be more expensive than a chunk hash and a key hash. Second, if the identifier can be derived from the key, then the file to chunk map only needs to preserve the key, as opposed to the key and the identifier. However, there is a large drawback of using the hash of the key as the identifier: the chunk store cannot verify that the chunk's content-based identifier is correct. As Section 3.2 explained, unverified chunk signatures permit the use of targeted collision attacks.

The encrypted chunks themselves are stored within the chunk store. In a distributed storage model, where there may be multiple chunk stores, the chunk list can also include the information needed to locate the correct storage device. Alternatively, deterministic placement algorithms can be used to locate the correct storage devices based on the chunk's identifier [18, 36, 8].

4.2 Authenticated Model

The authenticated model is the most similar to the original design of convergent encryption as it is utilized in Farsite [10]. As with their design, our authenticated model makes a number of assumptions regarding encryption keys and the key management techniques available to users. First, we assume that each user has a symmetric key that is private to that user. Second, we assume that each user also has an asymmetric key pair. Third, it is also assumed that a certificate authority exists to facilitate the trusted distribution of public keys. Finally, it is assumed that users are able to generate cryptographically sound encryption keys.

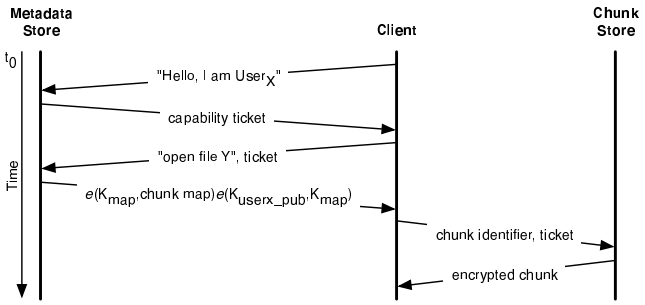


Figure 3: Extraction in the authenticated model begins with the client contacting the metadata store for the secure chunk map, and the chunk map's encryption key. From there, subsequent communication involves requesting chunks from the chunk store.

Ingestion begins with the client identifying the chunks and then encrypting them using convergent encryption. Following this, the information needed to rebuild the files, including chunk locations, names and encryption keys, is stored within a chunk map. As Table 2 illustrates, this map is stored in the metadata store in a map entry and accessed through a file's inode number. Additionally, it is encrypted using a dedicated map key. To allow authorized users to decrypt the map, the map key is encrypted using the authorized users' public keys. These encrypted keys are identified via a user identifier, and appended to the end of the encrypted chunk map; this technique is similar to the widely used "lockbox" approach to encrypting files [24, 20]. As more users are granted access to the file, additional encrypted keys can be appended to the map entry. The final step of ingestion is to submit the encrypted chunks to the chunk store. As Subsection 3.2 discussed, the chunk store is capable of generating the chunk IDs, so the client is not required to submit an identifier along with each chunk.

Extraction, as Figure 3 illustrates, follows a communications path similar to that of ingestion. The process starts with the client authenticating to the metadata store and submitting an `open()` request. As shown in Table 2, the metadata store can use the file's inode number to locate the encrypted chunk map and list of encrypted map keys. Rather than return the chunk map and the entire list of keys, the metadata store will only return the chunk map and the key that corresponds to the user, resulting in less information to transmit, and not leaking to the user the list of all users that have access to the file. Finally, the client decrypts the map key and subsequently the chunk map, and directs chunk requests to the appropriate chunk store.

In addition to allowing multiple users access to a single chunk map, the list of encrypted map keys also plays a central role in revocation. If access to a file needs to be revoked for a specific user, a new chunk key can be generated, the chunk map is encrypted using the new key, and the list of encrypted keys is updated for the users that still have access.

4.3 Anonymous Model

The goal of the anonymous model is to hide the identities of both authors and readers. This model operates under the assumption that encrypted data is secure against an adversary that does not posses the correct encryption key; thus, authentication is unnecessary.

One of the drawbacks of an anonymous data-store is that both well-behaved and malicious users are anonymous. This opens the door to attacks in which authorized users perform malicious acts,

Metadata Store	Key	Value
File inode	file name	inode number
Map entry	inode	$e(K_{map}, \text{chunk map})[(\text{uid}, e(K_{user\_pub}, K_{map}))]$

Chunk Store	Chunk ID	Encrypted Chunk
	$hash(\text{encrypted chunk})$	$e(hash(\text{chunk}), \text{chunk})$

Table 2: Authenticated model persistent storage details. The map entry stores a chunk map (an ordered list of the data needed to request and decrypt chunks) and is encrypted using a dedicated map key. This key is then encrypted using the public key of users that are authorized to access the file and appended to the encrypted chunk map.

Metadata Store	Key	Value
File entry	file name	inode number
Map reference	$HMAC_{user}(\text{inode})$	$e(K_{user}, K_{map})$
Map entry$_i$	$HMAC_{map}(\text{inode}, \text{map}_{i-1})$	$e(K_{map}, [\text{chunk map}])$

Chunk Store	Chunk ID	Encrypted Chunk
	$hash(\text{encrypted chunk})$	$e(hash(\text{chunk}), \text{chunk})$

Table 3: Details of the anonymous model for a persistent chunk store. The model makes exclusive use of symmetric keys. Map references are used to store the map encryption key in a manner that allows users to see changes made by other authorized members.

such as deleting or changing data, under the assumption that they cannot be definitively identified. One of the ways to guard against such an attack is to make both the metadata and chunk stores immutable. This approach thus implies that file changes are reflected in a versioned history of chunk maps, similar to the mechanism used in WAFL to transition to a new version of the file system [17]. Thus, as in systems such as SUNDR [23], malicious changes are isolated in a branch of the original file.

As Table 3 illustrates, the system makes exclusive use of symmetric keys; thus, the model must make several assumptions regarding encryption keys. First, it is assumed that each user has a symmetric encryption key that is private to the user. The use of a symmetric key for each user does not compromise anonymity because the key is used in HMAC procedures and, therefore, only the owner can confirm that the hash was created with their key. Second, it is assumed that users have the ability to generate cryptographically sound keys. Third, it is assumed that users are able to communicate off-line (relative to the chunk store) for the purposes of sharing keys.

In the anonymous model, as in the authenticated model, ingestion begins with the client identifying and encrypting chunks. The information needed to reconstruct the file from chunks is written to a map entry and encrypted using a map key that is generated when the file is created. This map entry is then written to the metadata store.

As Figure 4 shows, the system utilizes a map reference, specific to each user, that holds the map key to allow multiple users to see file changes made by other authorized users. Thus, file access is granted outside of the system by sharing the map key. When the user is given a map key, they create and store a map reference in the metadata store. In this manner, the only key the client is required to remember is their private symmetric key.

If changes are made to a file, the new map entry is written to the metadata store as a linked list. As Figure 4 illustrates, each user's map reference is used to locate the root of this linked list through an HMAC keyed with the map key. Each time a client commits a write

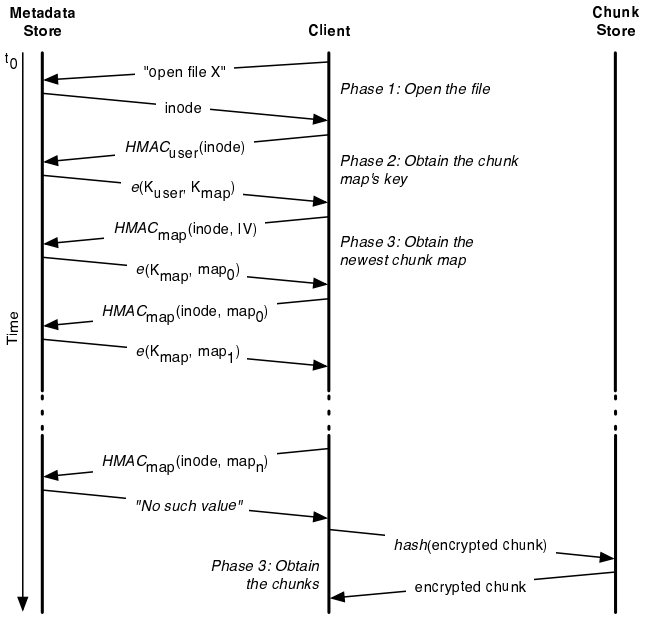


Figure 5: Extraction in the anonymous model begins with an `open()` request which returns an inode number. Using the file's inode number and the user's symmetric key, a user can obtain the chunk list through his chunk list reference. Requests for chunks are then directed to the chunk store.

to the system, a new node is appended to the list. Traversal of the list is accomplished through an HMAC, keyed with the map key, of the inode, and the previous map entry. As with the authenticated model, the client submits chunks to the chunk store in the final stage of ingestion.

Sharing files could also be accomplished by using the authorized user's symmetric key to encrypt the map key, and appending this encrypted key to the chunk map. While similar to the authenticated model's strategy, this approach suffers from a number of disadvantages. First, any information that identifies the user's key in the list is breaking anonymity. Second, even if an HMAC of the inode was used to hide the user's identity, the list would still leak the number of users that have access to the file. Third, the use of map references provides a level of coarse grained revocation. A chunk map can be created and encrypted with a new chunk map key. This new key would then need to be distributed to authorized users, who in turn can create new map references.

File extraction in the anonymous model, illustrated in Figure 5, proceeds in four phases. First, the client contacts the metadata store to issue an `open()` request and obtain the file's inode. Second,

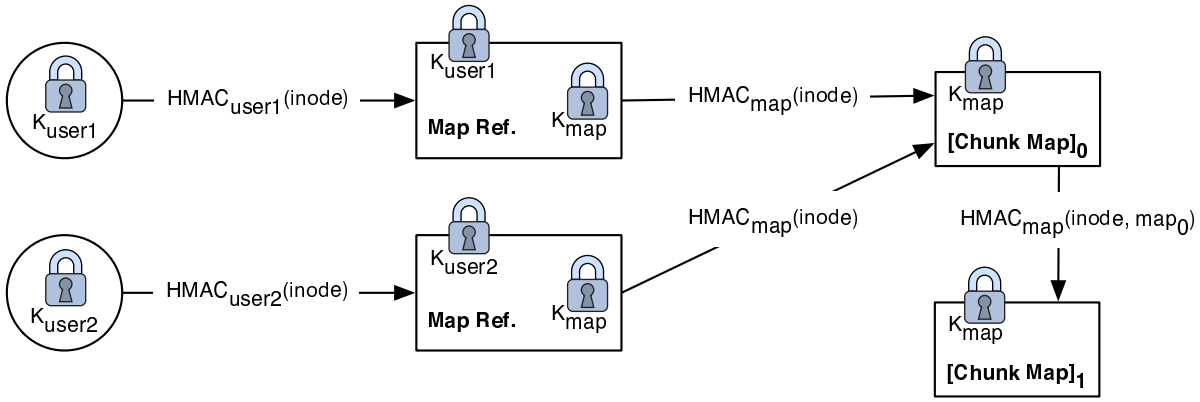


Figure 4: The information needed to reconstruct files is stored as a linked list of immutable chunk maps that are encrypted using a dedicated key, K_{map}. Each user creates a map reference, protected by their unique symmetric key, to store the map key. This allows users to see changes created by other authorized users, while only requiring them to remember one key (their unique user key).

the client obtains the map's key by utilizing the map reference, as shown in Figure 4. In the third phase, the client traverses the linked list of map entries by issuing map requests until they arrive at the version they want, or the map request fails, indicating that the end of the list has ben reached. In the fourth and final phase, the client utilizes the map entry and map key to determine which chunks to request from the chunk store.

5. SECURITY ANALYSIS

The evaluation of the two secure deduplication models that we have presented is intended to demonstrate that the system is secure in the face of a variety of foreseeable scenarios. First, we examine the attacks that an external adversary could inflict upon the system. Second, we examine the security leaks possible when faced with a malicious insider who might have access to all of the raw data, such as system administrator with root-level access. Third, we examine the security implications involved when the keys in the system become compromised.

5.1 External Adversaries

For a system to be considered secure, it must be able to prevent information from leaking to an external attacker. A passive example of such an adversary would be an attacker that intercepts messages sent between players in the system. An active example is an adversary that changes or transmits messages.

In both the authenticated and external model, the passive attacker problem is largely ameliorated by having the client perform the chunking and encryption. Thus, plaintext data is never transmitted in the clear. However, the anonymous model assumes that the keys can be exchanged in a secure manner but does not explicitly state how this is accomplished. A potential area of future work could be to define a secure protocol for this procedure.

Since data transmitted between players is always encrypted, the danger from an active adversary is one of messages being changed. For example, in the basic models we have presented, a chunk could be intercepted en route to the chunk store and modified. While our design does not explicitly address such scenarios, these attacks can be largely mitigated through the use of transport layer security (TLS) approaches such as Secure Sockets Layer.

As the anonymous model includes the goal of hiding the user's identity, an external adversary can gain some information by identifying where requests originate from. As with the man-in-the-middle type attacks previously discussed, our system does not directly deal with the issue, however solutions such as onion routing have addressed this concern, and are compatible with our design [12].

5.2 Internal Adversaries

As discussed in Section 3, a secure system must also provide protection from internal attackers. To this end, we analyze the ability of an inside adversary to launch attacks based on their location within the system and across their potential access levels.

As in most systems, a malicious insider with full access can change or delete any information he chooses, resulting in a denial of service attack. From a security standpoint, our goal is, therefore, to limit an insider's ability to make targeted changes. There are two facets to limiting such changes. First, we would like to limit an insider's ability to target specific files. Second, we would like to limit an adversary's ability to make undetectable changes; overwriting a value with garbage is generally more detectable that overwriting it with a semantically valid, but incorrect value.

5.2.1 Authenticated Model

In the authenticated model, the metadata server does leak some information to an internal adversary. First, an insider has access to the file name to inode mapping. Second, the inode number to encrypted map entry is also available to an internal adversary. Finally, a malicious insider can determine the files to which a user has access, and the users that have access to a specific file.

Using the information available, a inside attacker at the metadata server is able to launch a variety of attacks. First, an inside adversary can delete metadata and revoke access for specific users. If the client is not knowledgeable about which files it should be able to access, this attack is undetectable. Second, when a client requests a file, the map entry of a different file accessible by the client could be returned. Whether or not this attack is detected would rely upon the client's understanding of the file's contents.

Targeted changes to file contents, however, require the adversary to obtain the map key. In the current design, users grant access by submitting map keys encrypted using the authorized user's public key. In this way, a malicious insider is never exposed to the plaintext key needed to access a map entry's details. If the system were to encrypt map keys, a malicious insider could change the contents of map entries. One way to further strengthen the system, then, would be to hide the map entry from an inside attacker. This could be accomplished using a technique such as the anonymous model's

map references, which, as shown in Figure 4, requires the map key in order to locate the map entry.

Finally, if a malicious insider at the metadata store also distributes capability tickets, as is done in some systems, then it can be assumed that the adversary also has access to chunks; a malicious metadata store can simply issue itself a valid capability. However, without access to the map key, the adversary would not know which chunks correspond to a give file, and would lack the key needed to decrypt a chunk.

5.2.2 Anonymous Model

In both the authenticated and anonymous model, an inside adversary at the chunk store would be unable to modify data without being detected. Since the name of the chunk is based on the content, a user would not be able to request the modified chunk, or at the very least could tell that the chunk they requested is different from the chunk that was returned to them. An insider at the chunk store could, of course, delete chunks or refuse to fulfill chunk requests.

In the anonymous model, the metadata store does leak some information to an internal adversary. First, an insider can deduce which inode numbers map to which files. This is not a serious issue because the user's symmetric key is needed to map inodes to map references. More importantly, however, an insider could deduce which entries are map references, as they will all be the same length. This is due to the fact that their payload is always one key, as opposed to a variable list of chunk metadata. One way to avoid leaking the fact that an entry is a map-key is to append some amount of random data to the entry.

5.3 Key Compromises

Any system that utilizes cryptographic primitives is highly dependent on the controlled access of encryption keys for the security of the system. As Kerckhoff's principle states, the security of the system comes from an adversary not knowing the encryption key; it is assumed that the adversary knows the protocols and cryptosystems. Thus, one way to analyze a security system is to examine the effects of compromised keys.

5.3.1 Authenticated Model

In the authenticated model, the user's identity is tied to their asymmetric key pair. Further, if an adversary learns a users private key, it is assumed they have the users complete key pair; the public key can easily be acquired from a certificate server. In this scenario, a malicious user may be able to fully impersonate the key's rightful owner, and obtain all the abilities of that user. As a safeguard against this possibility, it is recommended that authentication require more information than the user's key, but this approach is outside the scope of our model.

A compromise of the other metadata key in the authenticated model, the map key, results in a less drastic information leak. If an adversary learns the map key, the problem of authenticating to the metadata store still exists. Finally, the revocation process can be used to generate a new map key, making the old key invalid. Thus, the system is relatively safe in the event of a compromised map key.

Similarly, if the last key of the authenticated model, the chunk key, is compromised, the information leak is rather small. This is due to the fact that an adversary with the chunk key would still need to know the chunk identifier, and be able to authenticate to the chunk server in order to obtain plaintext data.

5.3.2 Anonymous Model

In the anonymous model, the user's private, symmetric key is very important to the security of the system. If a malicious user obtains the user's key, it can be safely assumed that they can access any file that the user has stored a map reference for. Another potential attack they can issue in this scenario is to extend the length of the linked list of map entries indefinitely. However, since the anonymous model uses immutable chunks, a new key could be generated, and the file branched.

If an adversary obtains the map key, the adversary will only need the inode number of a the file to obtain plaintext data. Assuming that the number of inodes is relatively small, this can be accomplished using a brute force attack. Additionally, as the system is immutable, even generating a new map key will result in the original file being compromised.

As in the authenticated model, an adversary with the chunk's encryption key, would still need to know the chunk identifier in order to obtain plaintext data.

6. FUTURE WORK

While the models we have presented demonstrate some of the ways that security and deduplication can coexist, works remains to create a fully realized, secure, space efficient storage system. Open areas for exploration exist in both security, as well as deduplication.

Storage efficiency can be increased in a number of ways through intelligent chunking procedures. For example, the size of the file may be used to determine the average chunk size, potentially yielding greater deduplication in data such as media files, which tend to be large and exhibit an "all or nothing" level of similarity with other files. However, since some large files, such as mail archives or `tar` files, may be aggregations of smaller files, another possibility would be to adjust chunking parameters based on file types. Since chunking is done at the clients rather than at the servers, this approach only requires that clients agree on the way they divide files into chunks. Moreover, taking this approach does not increase the likelihood of collision, which remains very small for chunk identifiers of 160 bits or longer.

Unfortunately, techniques such as delta compression on files or chunks [11, 38], while they have proven effective for standard deduplicated storage systems, may not work well with encryption because clients cannot access encrypted chunks for any files but their own, limiting the source material for deduplication. Moreover, approaches that try to locate similar chunks in a chunk store will likely be ineffective because they require plaintext data for indexing; similar, but not identical, plaintext chunks will result in ciphertext chunks that have no similarity.

Another way to increase storage efficiency would be to provide deletion and garbage collection. While these are straightforward to implement in many systems, storage reclamation can be difficult in a system that uses deduplication because a single chunk may be referenced by many different files. Thus, removing a chunk requires an understanding of how many files reference the chunk. A common approach to deletion in a deduplicated file system uses reference counts to track the number of files that use a particular chunk. Of course, such a system would need to ensure that a malicious user could not launch a denial of service type attack simply by deleting chunks or modifying reference counts.

Currently, our model provides an all or nothing level of access; if a user has the map key, they have access to the file. Future designs could utilize multiple levels of permissions. Thus, a user could be allowed to read a file, but the system would prevent them from deleting information.

Finally, even in the anonymous model, a secure capability from the metadata store could be used to implement file locking. Currently, there is no guard against multiple users writing to the same chunk map. While the anonymous model is immutable and therefore all versions of a file are present, locking would still allow for changes to appear at the correct order in the list of maps. This would, of course compromise some of the users anonymity, as their requests would form a distinct session.

7. CONCLUSION

We have developed two models for secure deduplicated storage: authenticated and anonymous. These two designs demonstrate that security can be combined with deduplication in a way that provides a diverse range of security characteristics. In the models we present, security is provided through the use of convergent encryption. This technique, first introduced in the context of the Farsite system [1, 10], provides a deterministic way of generating an encryption key, such that two different users can encrypt data to the same ciphertext. In both the authenticated and anonymous models, a map is created for each file that describes how to reconstruct a file from chunks. This file is itself encrypted using a unique key. In the authenticated model, sharing of this key is managed through the use of asymmetric key pairs. In the anonymous model, storage is immutable, and file sharing is conducted by sharing the map key offline and creating a map reference for each authorized user.

In our evaluation, we have analyzed the security of each model with regard to a number of security compromises. We found that the system is mostly secure against external attackers. Further, the security threats that our models do not explicitly guard against can be addressed through the addition of standard secure communications techniques such as transport later security. Security compromises by a malicious insider are largely mitigated from the design's avoidance of server side encryption. Since insiders are never exposed to plain-text or encrypted keys, their ability to change metadata values in an undetectable way is greatly diminished. Security is even more apparent in the chunk store where the content addressed nature of secure chunks intrinsically makes the detection of malicious changes quite noticeable. Finally, we examined the information leaks resulting from key compromises and found that the most severe security breaches result from the loss of the client's key. The damage in the event of such a key loss is confined, however, to the user's files. Moreover, the breach of client's keys is a serious threat in most secure systems.

Acknowledgments

We thank the members of the Storage Systems Research Center (SSRC) for spirited discussions that helped focus the content of this paper. This work was supported in part by the Department of Energy under award DE-FC02-06ER25768 and industrial sponsors of the Storage Systems Research Center at UC Santa Cruz, including Agami Systems, Data Domain, Hitachi, LSI Logic, NetApp, Seagate, and Symantec.

8. REFERENCES

[1] A. Adya, W. J. Bolosky, M. Castro, R. Chaiken, G. Cermak, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. Wattenhofer. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*, Boston, MA, Dec. 2002. USENIX.

[2] 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)*, pages 31–45, Feb. 2007.

[3] R. Anderson, R. Needham, and A. Shamir. The steganographic file system. In *Proceedings of the International Workshop on Information Hiding (IWIH 1998)*, pages 73–82, Portland, OR, Apr. 1998.

[4] S. Annapureddy, M. J. Freedman, and D. Mazières. Shark: Scaling file servers via cooperative caching. In *Proceedings of the 2nd Symposium on Networked Systems Design and Implementation (NSDI)*, pages 129–142, 2005.

[5] D. Bhagwat, K. Pollack, D. D. E. Long, E. L. Miller, J.-F. Pâris, and T. Schwarz, S. J. Providing high reliability in a minimum redundancy archival storage system. In *Proceedings of the 14th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS '06)*, Monterey, CA, Sept. 2006.

[6] W. J. Bolosky, S. Corbin, D. Goebel, and J. R. Douceur. Single instance storage in Windows 2000. In *Proceedings of the 4th USENIX Windows Systems Symposium*, pages 13–24. USENIX, Aug. 2000.

[7] P. J. Braam. The Lustre storage architecture. http://www.lustre.org/documentation.html, Cluster File Systems, Inc., Aug. 2004.

[8] A. Brinkmann, S. Effert, F. Meyer auf der Heide, and C. Scheideler. Dynamic and redundant data placement. In *Proceedings of the 27th International Conference on Distributed Computing Systems (ICDCS '07)*, 2007.

[9] I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong. Freenet: A distributed anonymous information storage and retrieval system. *Lecture Notes in Computer Science*, 2009:46–66, 2001.

[10] J. R. Douceur, A. Adya, W. J. Bolosky, D. Simon, and M. Theimer. Reclaiming space from duplicate files in a serverless distributed file system. In *Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS '02)*, pages 617–624, Vienna, Austria, July 2002.

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

[12] D. Goldschlag, M. Reed, and P. Syverson. Onion routing. *Communications of the ACM*, 1999.

[13] G. R. Goodson, J. J. Wylie, G. R. Ganger, and M. K. Reiter. Efficient Byzantine-tolerant erasure-coded storage. In *Proceedings of the 2004 Int'l Conference on Dependable Systems and Networking (DSN 2004)*, June 2004.

[14] H. S. Gunawi, N. Agrawal, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and J. Schindler. Deconstructing commodity storage clusters. In *Proceedings of the 32nd Int'l Symposium on Computer Architecture*, pages 60–71, June 2005.

[15] S. Hand and T. Roscoe. Mnemosyne: Peer-to-peer steganographic storage. *Lecture Notes in Computer Science*, 2429:130–140, Mar. 2002.

[16] Health Information Portability and Accountability Act, Oct. 1996.

[17] D. Hitz, J. Lau, and M. Malcom. File system design for an NFS file server appliance. In *Proceedings of the Winter 1994*

USENIX Technical Conference, pages 235–246, San Francisco, CA, Jan. 1994.

[18] 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 & Distributed Processing Symposium (IPDPS 2004)*, Santa Fe, NM, Apr. 2004. IEEE.

[19] A. Iyengar, R. Cahn, J. A. Garay, and C. Jutla. Design and implementation of a secure distributed data repository. In *Proceedings of the 14th IFIP International Information Security Conference (SEC '98)*, pages 123–135, Sept. 1998.

[20] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, and K. Fu. Plutus: scalable secure file sharing on untrusted storage. In *Proceedings of the Second USENIX Conference on File and Storage Technologies (FAST)*, pages 29–42, San Francisco, CA, Mar. 2003. USENIX.

[21] A. W. Leung, E. L. Miller, and S. Jones. Scalable security for petascale parallel file systems. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing (SC '07)*, Nov. 2007.

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

[23] J. Li, M. Krohn, D. Mazières, and D. Shasha. Secure untrusted data repository (SUNDR). In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI)*, San Francisco, CA, Dec. 2004.

[24] E. L. Miller, D. D. E. Long, W. E. Freeman, and B. C. Reed. Strong security for network-attached storage. In *Proceedings of the 2002 Conference on File and Storage Technologies (FAST)*, pages 1–13, Monterey, CA, Jan. 2002.

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

[26] M. G. Oxley. (H.R.3763) Sarbanes-Oxley Act of 2002, Feb. 2002.

[27] S. Quinlan and S. Dorward. Venti: A new approach to archival storage. In *Proceedings of the 2002 Conference on File and Storage Technologies (FAST)*, pages 89–101, Monterey, California, USA, 2002. USENIX.

[28] M. O. Rabin. Fingerprinting by random polynomials. Technical Report TR-15-81, Center for Research in Computing Technology, Harvard University, 1981.

[29] S. Rhea, P. Eaton, D. Geels, H. Weatherspoon, B. Zhao, and J. Kubiatowicz. Pond: the OceanStore prototype. In *Proceedings of the Second USENIX Conference on File and Storage Technologies (FAST)*, pages 1–14, Mar. 2003.

[30] D. Roselli, J. Lorch, and T. Anderson. A comparison of file system workloads. In *Proceedings of the 2000 USENIX Annual Technical Conference*, pages 41–54, San Diego, CA, June 2000. USENIX Association.

[31] A. Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, Nov. 1979.

[32] M. W. Storer, K. M. Greenan, and E. L. Miller. Long-term threats to secure archives. In *Proceedings of the 2006 ACM Workshop on Storage Security and Survivability*, Alexandria, VA, Oct. 2006.

[33] M. W. Storer, K. M. Greenan, E. L. Miller, and K. Voruganti. POTSHARDS: secure long-term storage without encryption. In *Proceedings of the 2007 USENIX Annual Technical Conference*, pages 143–156, June 2007.

[34] M. Waldman, A. D. Rubin, and L. F. Cranor. Publius: A robust, tamper-evident, censorship-resistant web publishing system. In *Proceedings of the 9th USENIX Security Symposium*, pages 59–72, Denver, CO, Aug. 2000.

[35] 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 Symposium on Operating Systems Design and Implementation (OSDI)*, Seattle, WA, Nov. 2006. USENIX.

[36] S. A. Weil, S. A. Brandt, E. L. Miller, and C. Maltzahn. CRUSH: Controlled, scalable, decentralized placement of replicated data. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing (SC '06)*, Tampa, FL, Nov. 2006. ACM.

[37] 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)*, pages 17–33, Feb. 2008.

[38] L. L. You, K. T. Pollack, and D. D. E. Long. Deep Store: An archival storage system architecture. In *Proceedings of the 21st International Conference on Data Engineering (ICDE '05)*, Tokyo, Japan, Apr. 2005. IEEE.

POTSHARDS: Secure Long-Term Storage Without Encryption

Mark W. Storer Kevin M. Greenan Ethan L. Miller
University of California, Santa Cruz
Kaladhar Voruganti
Network Appliance[†]

Abstract

Users are storing ever-increasing amounts of information digitally, driven by many factors including government regulations and the public's desire to digitally record their personal histories. Unfortunately, many of the security mechanisms that modern systems rely upon, such as encryption, are poorly suited for storing data for indefinitely long periods of time—it is very difficult to manage keys and update cryptosystems to provide secrecy through encryption over periods of decades. Worse, an adversary who can compromise an archive need only wait for cryptanalysis techniques to catch up to the encryption algorithm used at the time of the compromise in order to obtain "secure" data.

To address these concerns, we have developed POTSHARDS, an archival storage system that provides long-term security for data with very long lifetimes without using encryption. Secrecy is achieved by using provably secure secret splitting and spreading the resulting shares across separately-managed archives. Providing availability and data recovery in such a system can be difficult; thus, we use a new technique, approximate pointers, in conjunction with secure distributed RAID techniques to provide availability and reliability across independent archives. To validate our design, we developed a prototype POTSHARDS implementation, which has demonstrated "normal" storage and retrieval of user data using indexes, the recovery of user data using only the pieces a user has stored across the archives and the reconstruction of an entire failed archive.

1 Introduction

Many factors motivate the need for secure long-term archives, ranging from the relatively short-term (for archival purposes) requirements on preservation, retrieval and security properties demanded by recent legislation [1, 20] to the indefinite lifetimes of cultural and family heritage data. As users increasingly create and store images, video, family documents, medical records and legal records digitally, the need to securely preserve this data for future generations grows correspondingly. This information often needs to be stored securely; data such as medical records and legal documents that could be important to future generations must be kept indefinitely but must not be publicly accessible.

The goal of a secure, long-term archive is to provide security for relatively static data with an indefinite lifetime. There are three primary security properties that such archives aim to provide. First, the data stored must only be viewable by authorized readers. Second, the data must be available and accessible to authorized users within a reasonable amount of time, even to those who might lack a specific key. Third, there must be a way to confirm the integrity of the data so that a reader can be reasonably assured that the data that is read is the same as the data that was written.

The usage model of secure, long-term archival storage is write-once, read-maybe, and thus stresses throughput over low-latency performance. This is quite different from the top storage tier of a hierarchical storage solution that stresses low-latency access or even bottom-tier backup storage. The usage model of long-term archives also has the unique property that the reader may have little knowledge of the system's contents and no contact with the original writer; while file lifetimes may be indefinite, user lifetimes certainly are not. For digital "time capsules" that must last for decades or even centuries, the writer is assumed to be gone soon after the data has been written.

There are many novel storage problems [3, 32] that result from the potentially indefinite data lifetimes found in long-term storage. This is partially due to mechanisms such as cryptography that work well in the short-term but are less effective in the long-term. In long-term applications, encryption introduces the problems of lost

[†]Work performed while a member of IBM Almaden Research

keys, compromised keys and even compromised cryptosystems. Additionally, the management of keys becomes difficult because data will experience many key rotations and cryptosystem migrations over the course of several decades; this must all be done without user intervention because the user who stored the data may be unavailable. Thus, security for archival storage must be designed explicitly for the unique demands of long-term storage.

To address the many security requirements for long-term archival storage, we designed and implemented POTSHARDS (Protection Over Time, Securely Harboring And Reliably Distributing Stuff), which uses three primary techniques to provide security for long-term storage. The first technique is secret splitting [28], which is used to provide secrecy for the system's contents. Secret splitting breaks a block into n pieces, m of which must be obtained to reconstitute the block; it can be proven that any set of fewer than m pieces contains *no* information about the original block. As a result, secret splitting does not require the same updating as encryption, which is only computationally secure. By providing data secrecy without the use of encryption, POTSHARDS is able to move security from encryption to the more flexible and secure authentication realm; unlike encryption, authentication need not be done by computer, and authentication schemes can be easily changed in response to new vulnerabilities. Our second technique, *approximate pointers*, makes it possible to reconstitute the data in a reasonable time even if all indices over a user's data have been lost. This is achieved without sacrificing the secrecy property provided by the secret splitting. The third technique is the use of secure, distributed RAID techniques across multiple independent archives. In the event that an archive fails, the data it stored can be recovered without the need for other archives to reveal their own data.

We implemented a prototype of POTSHARDS and conducted several experiments to test its performance and resistance to failure. The current, CPU-bound implementation of POTSHARDS can read and write data at 2.5–5 MB/s on commodity hardware but is highly parallelizable. It also survives the failure of an entire archive with no data loss and little effect seen by users. In addition, we demonstrated the ability to rebuild a user's data from all of the user's stored shares without the use of a user index. These experiments demonstrate the system's suitability to the unique usage model of long-term archival storage.

2 Background

Since POTSHARDS was designed specifically for secure, long-term storage, we identified three basic design tenets to help focus our efforts. First, we assumed that encrypted data could be read by anyone given sufficient CPU cycles and advances in cryptanalysis. This means that, if all of an archives encrypted contents are obtained, an attacker can recover the original information. Second, data must be recoverable without any information from outside the set of archives. Thus, fulfilling requests in a reasonable time cannot require anything stored outside the archives, including external indexes or encryption keys. If this assumption is violated, there is a high risk that data will be unrecoverable after sufficient time has passed because the needed external information has been lost. Third, we assume that individuals are more likely to be malicious than an aggregate. In other words, our system can trust a group of archives even though it may not trust an individual archive. The chances of every archive in the system colluding maliciously is small; thus, we designed the system to allow rebuilding of stored data if all archives cooperate.

In designing POTSHARDS to meet these goals, we used concepts from various research projects and developed additional techniques. There are many existing storage systems that satisfy some of the design tenets discussed above, ranging from general-purpose distributed storage systems to distributed content delivery systems, to archival systems designed for short-term storage and archival systems designed for very specific uses such as public content delivery. A representative sample of these systems is summarized in Table 1. The remainder of this section discusses each of these primary tenets within the context of the related systems. Since these existing systems were not designed with secure, archival storage in mind, none has the combination of long-term data security and proof against obsolescence that POTSHARDS provides.

2.1 Archival Storage Models

Storage systems such as Venti [23] and Elephant [26] are concerned with archival storage, but tend to focus on the near-term time scale. Both systems are based on the philosophy that inexpensive storage makes it feasible to store many versions of data. Other systems, such as Glacier [13], are designed to take advantage of the underutilized client storage of a local network. These systems, and others that employ "checkpoint-style" backups, address neither the security concerns of the data content nor the needs of long-term archival storage. Venti and commercial systems such as the EMC Centera [12] use content-based storage techniques to achieve their goals, naming blocks based on a secure hash of their data. This approach increases reliability by providing an easy way to verify the content of a block against its name. As with the short-term storage systems described above, security

System	Secrecy	Authorization	Integrity	Blocks for Compromise	Migration
FreeNet	encryption	none	hashing	1	access based
OceanStore	encryption	signatures	versioning	m (out of n)	access based
FarSite	encryption	certificates	Merkle trees	1	continuous relocation
Publius	encryption	password (delete)	retrieval based	m (out of n)	
SNAD / Plutus	encryption	encryption	hashing	1	
GridSharing	secret splitting		replication	1	
PASIS	secret splitting		repair agents, auditing	m (out of n)	
CleverSafe	information dispersal	unknown	hashing	m (out of n)	none
Glacier	user encryption	node auth.	signatures	n/a	
Venti	none		retrieval	n/a	
LOCKSS	none		vote based checking	n/a	site crawling
POTSHARDS	secret splitting	pluggable	algebraic signatures	$O(R^{m-1})$	device refresh

Table 1: *Capability overview of the storage systems described in Section 2. "Blocks to compromise" lists the number of data blocks needed to brute-force recover data given advanced cryptanalysis; for POTSHARDS, we assume that an approximate pointer points to R shard identifiers. "Migration" is the mechanism for automatic replication or movement of data between nodes in the system.*

is ensured by encrypting data using standard encryption algorithms.

Some systems, such as LOCKSS [18] and Intermemory [10], are aimed at long-term storage of open content, preserving digital data for libraries and archives where file consistency and accessibility are paramount. These systems are developed around the core idea of very long-term access for public information; thus file secrecy is explicitly not part of the design. Rather, the systems exchange information about their own copies of each document to obtain consensus between archives, ensuring that a rogue archive does not "alter history" by changing the content of a document that it holds.

2.2 Storage Security

Many storage systems seek to enforce a policy of secrecy for their contents. Two common mechanisms for enforcing data secrecy are encryption and secret splitting.

2.2.1 Secrecy via Encryption

Many systems such as OceanStore [25], FARSITE [2], SNAD [19], Plutus [16], and e-Vault [15] address file secrecy but rely on the explicit use of keyed encryption. While this may work reasonably well for short-term secrecy needs, it is less than ideal for the very long-term security problem that POTSHARDS is addressing. Encryption is only computationally secure and the struggle between cryptography and cryptanalysis can be viewed as an arms race. For example, a DES encrypted message was considered secure in 1976; just 23 years later, in 1999, the same DES message could be cracked in under a day [29]; future advances in quantum computing have the potential to make many modern cryptographic algorithms obsolete.

The use of long-lived encryption implies that re-encryption must occur to keep pace with advances in cryptanalysis in order to ensure secrecy. To prevent a single archive from obtaining the unencrypted data, re-encryption must occur over the old encryption, resulting in a long key history for each file. Since these keys are all external data, a problem with any of the keys in the key history can render the data inaccessible when it is requested.

Keyed cryptography is only computationally secure, so compromise of an archive of encrypted data is a potential problem regardless of the encryption algorithm that is used. An adversary who compromises an encrypted archive need only wait for cryptanalysis techniques to catch up to the encryption used at the time of the compromise. If an insider at a given archive gains access to all of its data, he can decrypt any desired information even if the data is subsequently re-encrypted by the archive, since the insider will have access to the new key by virtue of his internal access. This is unacceptable, since the data's existence on a secure, long-term archive suggests that data will still be valuable even if the malicious user must wait several years to read it.

Some content publishing systems utilize encryption, but its use is not motivated solely by secrecy. Publius [34] utilizes encryption for write-level access control. Freenet [6] is designed for anonymous publication and encryption is used for plausible deniability over the contents of a users local store. As with secrecy, the use of encryption to enforce long-lived policy is problematic due to the mechanism's computationally secure nature.

2.2.2 Secrecy via Splitting

To address the issues resulting from the use of encryption, several recent systems including PASIS [11, 36] and GridSharing [33] have used or suggested the use [31] of *secret splitting* schemes [5, 22, 24, 28]; a related approach used by Mnemosyne [14] and CleverSafe [7] uses encryption followed by information dispersal (IDA) to attempt to gain the same security. In secret splitting, a secret is distributed by splitting it into a set number n of

shares such that no group of k shares ($k < m \leq n$) reveals any information about the secret; this approach is called an (m,n) threshold scheme. In such a scheme, any m of the n shares can be combined to reconstitute the secret; combining fewer than m shares reveals *no* information. A simple example of an (n,n) secret splitting scheme for a block B is to randomly generate $X_0,\ldots,X_{n-2}$, where $|X_i| = |B|$, and choose X_{n-1} so that $X_0 \oplus \cdots \oplus X_{n-2} \oplus X_{n-1} = B$. Secret splitting satisfies the second of our three tenets—data can be rebuilt without external information—but it can have the undesirable side-effect of combining the secrecy and redundancy aspects of the systems. Although related, these two elements of security are, in many respects, orthogonal to one another. Combining these elements also risks introducing compromises into the system by restricting the choices of secret splitting schemes.

To ensure that our third design tenet is satisfied, a secure long-term storage system must ensure that an attempt to breach security will be noticed by *somebody*, ensuring that the trust placed in the collection of archives can be upheld. Existing systems do not meet this goal because the secret splitting and data layout schemes they use are minimally effective against an inside attacker that knows the location of each of the secret shares. None of PASIS, CleverSafe, or GridSharing are designed to prevent attacks by insiders at one or more sites who can determine which pieces they need from other sites and steal those specific blocks of data, enabling a breach of secrecy with relatively minor effort. This problem is particularly difficult given the long time that data must remain secret, since such breaches could occur over years, making detection of small-scale intrusions nearly impossible. PASIS addressed the issue of refactoring secret shares [35]; however, this approach could compromise data in the system because the refactoring process may reveal information during the reconstruction process that a malicious archive could use to recover user data. By keeping this on separate nodes, the PASIS designers hoped to avoid information leakage. Mnemosyne used a local steganographic file system to hide chunks of data, but this approach is still vulnerable to rapid information leakage if the encryption algorithm is compromised because the IDA provides no additional protection to the distributed pieces.

2.3 Disaster Recovery

With long data lifetimes, hardware failure is a given; thus, dealing with a failed archive is inevitable. In addition, a long-term archival storage solution that relies upon multiple archives must be able to survive the loss of an archive for other reasons, such as business failure. Recovering from such large-scale disasters has long been a concern for storage systems [17]. To address this issue, systems such as distributed RAID [30], Myriad [4] and OceanStore [25] use RAID-style algorithms or more general redundancy techniques including (m,n) error correcting codes along with geographic distribution to guard against individual site failure. Secure, long-term storage adds the requirement that the secrecy of the distributed data must be ensured at all times, including during disaster recovery scenarios.

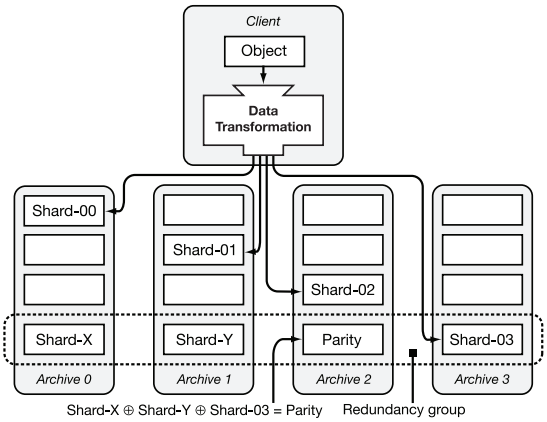


Figure 1: *An overview of POTSHARDS showing the data transformation component producing shards from objects and distributing them to independent archives. The archives utilize distributed RAID algorithms to securely recover shards in the event of a failure.*

3 System Overview

POTSHARDS is structured as a client communicating with a number of independent archives. Though the archives are independent, they assist each other through distributed RAID techniques to protect the system from archive loss. POTSHARDS stores user data by first splitting it into secure *shards*. These shards are then distributed to a number of archives, where each archive exists within its own security domain. The read procedure is similar but reversed; a client requests shards from archives and reconstitutes the data.

Data is prepared for storage during ingestion by a *data transformation* component that transforms *objects* into a set of secure shards which are distributed to the archives, as shown in Figure 1; similarly, this component is also responsible for reconstituting objects from shards during extraction. The data transformation component runs on a system **separate** from the archives on which the shards reside, and can fulfill requests from either a single client or many clients, depending on the implementation. This approach provides two benefits: the data never reaches an archive in an unsecured form; and multiple CPU-bound data transformation processes can generate shards in parallel for a single set of physical archives.

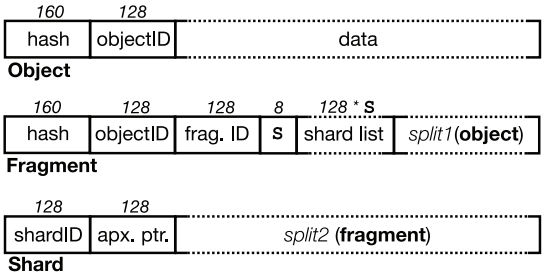


Figure 2: *Data entities in POTSHARDS, with size (in bits) indicated above each field. Note that entities are not shown to scale relative to one another. is the number of shards that the fragment produces. split1 is an XOR secret split and split2 is a Shamir secret split in POTSHARDS.*

The archives operate in a manner similar to financial banks in that they are relatively stable and they have a incentive (financial or otherwise) to monitor and maintain the security of their contents. Additionally, the barrier to entry for a new archive should be relatively high (although POTSHARDS does takes precautions against a malicious insider); security is strengthened by distributing shards amongst the archives, so it is important that each archive can demonstrate an ability to protect its data. Other benefits of archive independence include reducing the effectiveness of insider attacks and making it easier to exploit the benefits of geographic diversity in physical archive locations. For these reasons, a single entity, such as a multinational company, should still maintain multiple independent archives to gain these security and reliability benefits.

3.1 Data Entities and Naming

There are three main data objects in POTSHARDS: *objects*, *fragments* and *shards*. As Figure 1 shows, objects contain the data that users submit to the system at the top level. Fragments are used within the data transformation component during the production of shards, which are the pieces actually stored on the archives. The details of these data entities can be seen in Figure 2.

All data entities in the current implementation of POTSHARDS are given unique 128-bit identifiers. The first 40 bits of the name uniquely identify the client in the same manner as a bank account is identified by an account number. The remaining 88 bits are used to identify the data entity. The length of the identifier could be extended relatively easily in future implementations. The names for entities that do not directly contribute to security within POTSHARDS, such as those for objects, can be generated in any way desired. However, the security and recovery time for a set of shards is directly related to the shards' names; thus, shards' IDs must be chosen with great care to ensure a proper density of names, providing sufficient security.

In addition to uniquely identifying data entities within the system, IDs play an important role in the secret splitting algorithms used in POTSHARDS. For secret splitting techniques that rely on linear interpolation [28], the reconstitution algorithm must know the original order of the secret shares. Knowing the order of the shards in a shard tuple can greatly reduce the time taken to reconstitute the data by avoiding the need to try each permutation of share ordering. Currently, this ordering is done by ensuring that the numerical ordering of the shard IDs reflects the input order to the reconstitution algorithm.

3.2 Secrecy and Reliability Techniques

POTSHARDS utilizes three primary techniques in the creation and long-term storage of shards. First, secret splitting algorithms provide file secrecy without the need to periodically update the algorithm. This is due to the fact that perfect secret splitting is information-theoretically secure as opposed to only computationally secure. Second, approximate pointers between shards allow objects to be recovered from only the shards themselves. Thus, even if all indices over a user's shards are lost, their data can be recovered in a reasonable amount of time. Third, secure, distributed RAID techniques across multiple independent archives allow data to be recovered in the event of an archive failure without exposing the original data during archive reconstruction.

POTSHARDS provides data secrecy through the use of secret splitting algorithms; thus, there is no need to maintain key history because POTSHARDS does not use traditional encryption keys.Additionally, POTSHARDS utilizes secret splitting in a way that does not combine the secrecy and redundancy parameters. Storage of the secret shares is also handled in a manner that dramatically reduces the effectiveness of insider attacks. By using secret splitting techniques, the secrecy in POTSHARDS has a degree of future-proofing built into it—it can be proven that an adversary with infinite computational power cannot gain any of the original data, even if an entire archive is compromised. While not strictly necessary, the introduction of a small amount of redundancy at the secret splitting layer allows POTSHARDS to handle transient archive unavailability by not requiring that a reader obtain *all* of the shards for an object; however, redundancy at this level is used primarily for short-term failures.

POTSHARDS provides approximate pointers to enable the reasonably quick reconstitution of user data without any information that exists outside of the shards themselves. POTSHARDS users normally keep indexes allowing them to quickly locate the shards that they need to reconstitute a particular object, as described in Section 4.3, so normal shard retrieval consists of asking

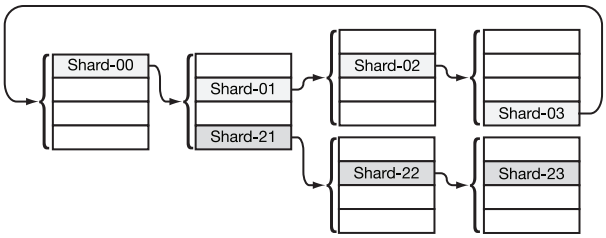
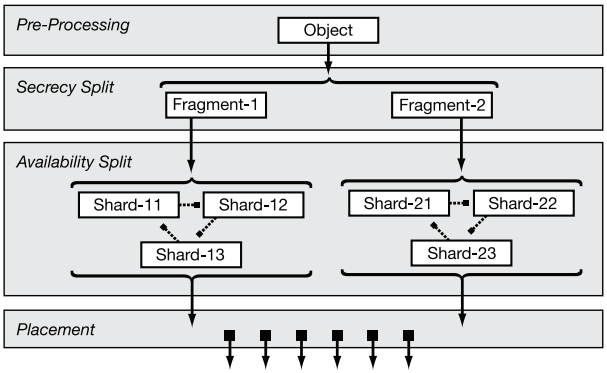


Figure 3: *Approximate pointers point to R "candidate" shards ($R = 4$ in this example) that might be next in a valid shard tuple. $Shards_{0X}$ make up a valid shard tuple. If an intruder mistakenly picks $shard_{21}$, he will not discover his error until he has retrieved sufficient shards and validation fails on the reassembled data.*

archives for the specific shards that make up an object, and is relatively fast. Approximate pointers are used when these user indexes are lost or otherwise unavailable. Since POTSHARDS can be used as a time capsule to secure data, it is foreseeable that a future user may be able to access the shards that they have a legal right to but have no idea how to combine them. The shards that can be combined together to reconstitute data form a *shard tuple*; an approximate pointer indicates the region in the user's private namespace where the next shard in the shard tuple exists, as shown in Figure 3. An approximate pointer has the benefit of making emergency data regeneration tractable while still making it difficult for an adversary to launch a targeted attack. If exact pointers were used, an adversary would know exactly which shards to target to rebuild an object. On the other hand, keeping no pointer at all makes it intractable to combine the correct shards without outside knowledge of which shards to combine. With approximate pointers, an attacker with one shard would only know the *region* where the next shard exists. Thus, a brute force attack requesting *every* shard in the region would be quite noticeable because the POTSHARDS namespace is intentionally kept sparse and an attack would result in requests for shards that do not exist. Unlike an index relating shards to objects that users would keep (and not store in the clear on an archive), an approximate pointer is part of the shard and is stored on the archive.

The archive layer in which the shards are stored consists of independent archives utilizing secure, distributed RAID techniques to provide reliability. As Figure 1 shows, archive-level redundancy is computed across sets of *unrelated* shards, so redundancy groups provide no insight into shard reassembly. POTSHARDS includes two novel modifications beyond the distributed redundancy explored earlier [4, 30]. The first is a secure reconstruction procedure, described in Section 4.2.1, that allows a failed archive's data to be regenerated in a manner that prevents archives from obtaining additional shards during the reconstruction; shards from the failed archive



(a) Four data transformation layers in POTSHARDS.

Module	*Input*	*Output*
Pre-processing	file	object
Secrecy split	object	set of fragments
Availability split	fragment	set of shards
Placement	set of shards	msgs for archives

(b) Inputs and outputs for each transformation layer.

Figure 4: *The transformation component consists of four levels. Approximate pointers are utilized at the second secret split. Note that locating one shard tuple provides no information about locating the shards from other tuples.*

are rebuilt only at the new archive that is replacing it. Second, POTSHARDS uses algebraic signatures [27] to ensure intra-archive integrity as well as inter-archive integrity. Algebraic signatures have the desirable property that the parity of a signature is the same as the signature of the parity, which can be used to prove the existence of data on other archives without revealing the data.

4 Implementation Details

This section details the components of POTSHARDS and how each contributes to providing long-term, secure storage. We first describe the transformation that POTSHARDS performs to ensure data secrecy. Next, we detail the inter-archive techniques POTSHARDS uses to provide long-term reliability. We then describe index construction; the use of indices makes "normal" data retrieval much simpler. Finally, we describe how we use approximate pointers to recover data with no additional information beyond the shards themselves, thus ensuring that POTSHARDS archives will be readable by future generations.

4.1 Data Transformation: Secrecy

Before being stored at the archive layer, user data travels through the data transformation component of POTSHARDS. This component is made up of four layers as shown in Figure 4.

1. **The pre-processing layer** divides files into fixed-sized,

b-byte objects. Additionally, objects include a hash that is used to confirm correct reconstitution.

2. **A secret splitting layer tuned for secrecy** takes an object and produces a set of fragments.
3. **A secret splitting layer tuned for availability** takes a fragment and produces a tuple of shards. It is also at this layer that the approximate pointers between the shards are created.
4. **The placement layer** determines how to distribute the shards to the archives.

4.1.1 Secret Splitting Layers

Fragments are generated at the first level of secret splitting, which is tuned for secrecy. Currently we use an XOR-based algorithm that produces n fragments from an object. To ensure security, the random data required for XOR splitting can be obtained through a physical process such as radio-active decay or thermal noise. As Figure 2 illustrates, fragments also contain metadata including a hash of the fragment's data which can be used to confirm a successful reconstitution.

A tuple of shards is produced from a fragment using another layer of secret splitting. This second split is tuned for availability which allows reconstitution in the event that an archive is down or unavailable when a request is made. In this version of POTSHARDS, shards are generated from a fragment using an (m,n) secret splitting algorithm [24, 28]. As the Figure 2 shows, shards contain no information about the fragments that they make up.

The two levels of secret splitting provide three important security advantages. First, as Figure 4 illustrates, the two-levels of splitting can be viewed as a tree with an increased fan out compared to one level of splitting. Thus, even if an attacker is able to locate all of the members of a shard tuple they can only rebuild a fragment and they have no information to help them find shards for the other fragments. Second, it separates the secrecy and availability aspects of the system. With two levels of secret splitting we do not need to compromise one aspect for the other. Third, it allows useful metadata to be stored with the fragments as this data will be kept secret by the second level of splitting. The details of shards and fragments are shown in Figure 2.

One cost of two-level secret splitting is that the overall storage requirements for the system are increased. A two-way XOR split followed by a $(2,3)$ secret split increases storage requirements by a factor of six; distributed RAID further increases the overhead. If a user desires to offset this cost, data can be submitted in a compressed archival form [37]; compressed data is handled just like any other type of data.

4.1.2 Placement Layer

The placement layer determines which archive will store each shard. The decision takes into account which shards belong in the same tuple and ensures that no single archive is given enough shards to recover data.

This layer contributes to security in POTSHARDS in four ways. First, since it is part of the data transformation component, no knowledge of which shards belong to an object need exist outside of the component. Second, the effectiveness of a insider attack at the archives is reduced because no single archive contains enough shards to reconstitute any data. Third, the effectiveness of an external attack is decreased because shards are distributed to multiple archives, each of which can exist in their own security domain. Fourth, the placement layer can take into account the geographic location of archives in order to maximize the availability of data.

4.2 Archive Design: Reliability

Storage in POTSHARDS is handled by a set of independent archives that store shards, actively monitor their own security and actively question the security of the other archives. The archives do not know which shards form a tuple, nor do they have any information about fragments or object reconstitution. A compromised archive does not provide an adversary with enough shards to rebuild user data. Nor does it provide an adversary with enough information to know where to find the appropriate shards needed to rebuild user data. Absent such precautions, the archive model would likely weaken the strong security properties provided by the other system components.

Since POTSHARDS is designed for long-term storage, it is inevitable that disasters will occur and archive membership will change over time. To deal with the threat of data loss from these events, POTSHARDS utilizes distributed RAID techniques. This is accomplished by dividing each archive into fixed-sized blocks and requiring all archives to agree on distributed, RAID-based methods over these blocks. Each block on the archive holds either shards or redundancy data.

When shards arrive at an archive for storage, ingestion occurs in three steps. First, a random block is chosen as the storage location of the shard. Next, the shard is placed in the last available slot in the the block. Finally, the corresponding parity updates are sent to the proper archives. Each parity update contains the data stored in the block and the appropriate parity block location. The failure of any parity update will result in a roll-back of the parity updates and re-placement of the shard into another block. Although it is assumed that all of the archives are trusted, we are currently analyz-

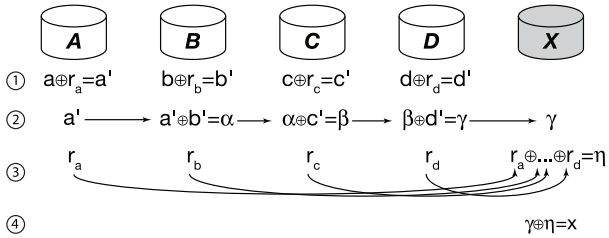


Figure 5: *A single round of archive recovery in a RAID 5 redundancy group. Each round consists of multiple steps. Archive N contains data n and generates random blocks r_n.*

ing the security effects of passing shard data between the archives during parity updates and exploring techniques for preventing archives from maliciously accumulating shards.

The distributed RAID techniques used in POTSHARDS are based on those from existing systems [4, 30]. In such systems, cost-effective, fault-tolerant, distributed storage is achieved by computing parity across unrelated data in wide area redundancy groups. Given an (n,k) erasure code, a redundancy group is an ordered set of k data blocks and $n-k$ parity blocks where each block resides on one of n distinct archives. The redundancy group can survive the loss of up to $n-k$ archives with no data loss. The current implementation of POTSHARDS has the ability to use Reed-Solomon codes or single parity to provide flexible and space-efficient redundancy across the archives.

POTSHARDS enhances the security of existing distributed RAID techniques through two important additions. First, the risk of information leakage during archive recovery is greatly mitigated through secure reconstruction techniques. Second, POTSHARDS utilizes algebraic signatures [27] to implement a secure protocol for both storage verification and data integrity checking.

4.2.1 Secure Archive Reconstruction

Reconstruction of data can pose a significant security risk because it can involve many archives and considerable amounts of data passing between archives. The secure recovery algorithm implemented within POTSHARDS exploits the independence of the archives participating in a redundancy group and the commutativity of evaluating the parity. Our reconstruction algorithm permits each archive to independently reconstruct a block of failed data without revealing any information about its data. The commutativity of the reconstruction procedure results in a reconstruction protocol that can occur in permutations, which greatly decreases the likelihood of successful collusion during archive recovery.

The recovery protocol begins with the confirmation of a partial or whole archive failure and, since each archive is a member of one or more redundancy groups, proceeds one redundancy group at a time. If a failure is confirmed, the archives in the system must agree on the destination of recovered data. A fail-over archive is chosen based on two criteria: the fail-over archive must not be a member of the redundancy group being recovered and it must have the capacity to store the recovered data. Due to these constraints multiple fail-over archives may be needed to perform reconstruction and redistribution. Future work will include ensuring that the choice of fail-over archives prevent any archive from acquiring enough shards to reconstruct user data.

Once the fail-over archive is selected, recovery occurs in multiple rounds. A single round of our secure recovery protocol over a single redundancy group is illustrated in Figure 5. In this example, the available members of a redundancy group collaborate to reconstruct the data from a failed archive onto a chosen archive X. An archive (which cannot be the fail-over and cannot be one of the collaborating archives) is appointed to manage the protocol by rebuilding one block at a time through multiple rounds of chained requests. A request contains an ordered list of archives, corresponding block identifiers and a data buffer and proceeds as follows at each archive in the chain:

1. Request α involving local block n arrives at archive N.
2. The archive creates a random block r_n and computes $n \oplus r_n = n'$.
3. The archive computes $\beta = \alpha \oplus n'$ and removes its entry from the request
4. The archive sends r_n directly to archive X.
5. β is sent to the next archive in the list.

This continues at each archive until the chain ends at archive X and the block is reconstructed. The commutativity of the rebuild process allows us to decrease the likelihood of data exposure by permuting the order of the chain in each round. This procedure is easily parallelized and continues until all of the failed blocks for the redundancy group are reconstructed. Additionally, this approach can be generalized to any linear erasure code; as long as the generator matrix for the code is known, the protocol remains unchanged.

4.2.2 Secure Integrity Checking

Preserving data integrity is a critical task in all long-term archives. POTSHARDS actively verifies the integrity of data using two different forms of integrity checking. The first technique requires each of the archives to periodically check its data for integrity violations using a hash stored in the header of each block on disk. The second technique is a form of inter-archive integrity checking that utilizes algebraic signatures [27] across the redundancy groups. Algebraic signatures have the prop-

erty that the signatures of the parity equals the parity of the signatures. This property is used to verify that the archives in a given redundancy group are properly storing data and are performing the required internal checks [27].

Secure, inter-archive integrity checking is achieved through algebraic signature requests over a specific interval of data. A check begins when an archive asks the members of a redundancy group for an algebraic signature over a specified interval of data. The algebraic signature forms a codeword in the erasure code used by the redundancy group and integrity over the interval of data is checked by comparing the parity of the data signatures to the signature of the parity. If the comparison check fails, then the archive(s) in violation may be found as long as the number of incorrect signatures is within the error-correction capability of the code. In general, a small signature (typically 4 bytes) is computed from a few megabytes of data. This results in very little information leakage. If necessary, restrictions may be placed on algebraic signature requests to ensure that no data is exposed during the integrity check process.

4.3 User Indexes

When shards are created, the *exact* names of the shards are returned to the user along with their archive placement locations; however, these exact pointers are *not* stored in the shards themselves, so they are not available to someone attacking the archives. Typically, a user maintains this information and the relationship between shards, fragments, objects, and files in an index to allow for fast retrieval. In the general case, the user consults her index and requests specific shards from the system. This index can, in turn, be stored within POTSHARDS, resulting in an index that can be rebuilt from a users shards with no outside information.

The index for each user can be stored in POTSHARDS as a linked list of index pages with new pages inserted at the head of the list, as shown in Figure 6. Since the index pages are designed to be stored within POTSHARDS, each page is immutable. When a user submits a file to the system, a list of mappings from the file to its shards is returned. This data is recorded in a new index page, along with a list of shards corresponding to the previous head of the index list. This new page is then submitted to the system and the shard list returned is maintained as the new head of the index list. These index root-shards can be maintained by the client application or even on a physical token, such as a flash drive or smart card.

This approach of each user maintaining their own private index has three advantages. First, since each user maintains his own index, the compromise of a user index does not affect the security of other users' data. Second,

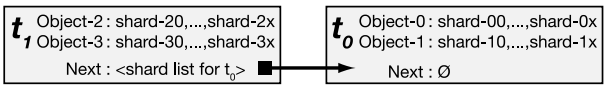


Figure 6: *User index made up of two pages. One page was created at time t_0 and the other at time t_1.*

the index for one user can be recovered with no effect on other users. Third, the system does not know about the relationship between a user's shards and their data.

In some ways, the index over a user's shards can be compared to an encryption key because it contains the information needed to rebuild a user's data. However, the user's index is different from an encryption key in two important ways. First, the user's index is not a single point of failure like an encryption key. If the index is lost or damaged, it can be recovered from the data without any input from the owner of the index. Second, full archive collusion can rebuild the index. If a user can prove a legal right to data, such as by a court subpoena, than the archives can provide all of the user's shards and allow the reconstitution of the data. If the data was encrypted, the files without the encryption key might not be accessible in a reasonable period of time.

4.4 Approximate Pointers and Recovery

Approximate pointers are used to relate shards in the same shard tuple to one another in a manner that allows recovery while still reducing an adversary's ability to launch a targeted attack. Each shard has an approximate pointer to the next shard in the fragment, with the last shard pointing back to the first and completing the cycle, as shown in Figure 3. This allows a user to recover data from their shards even if all other outside information, such as the index, is lost.

There are two ways that approximate pointers can be implemented: randomly picking a value within $R/2$ above or below the next shard's identifier, or masking off the low-order r bits ($R = 2^r$) of the next shard's identifier, hiding the true value. Currently, POTSHARDS uses the latter approach; we are investigating the tradeoffs between the two approaches. One benefit to using the $R/2$ approach is that it allows a finer-grained level of adjustment compared to the relatively coarse-grained bitmask approach.

The use of approximate pointers provides a great deal of security by preventing an intruder who compromises an archive or an inside attacker from knowing exactly which shards to steal from other archives. An intruder would have to steal *all* of the shards an approximate pointer could refer to, and would have to steal all of the shards they refer to, and so on. All of this would have to bypass the authentication mechanisms of each archive, and archives would be able to identify the access pattern of a thief, who would be attempting to obtains shards that

may not exist. Since partially reconstituted fragments cannot be verified, the intruder might have to steal *all* of the potential shards to ensure that he was able to reconstitute the fragment. For example, if an approximate pointer points to R shards and a fragment is split using (m,n) secret splitting, an intruder would have to steal, on average, $R^{m-1}/2$ shards to decode the fragment.

In contrast to a malicious user, a legitimate user with access to all of his shards can easily rebuild the fragments and, from them, the objects and files they comprise. Suppose this user created shards from fragments using an (m,n) secret splitting algorithm. A user would start by obtaining all of her shards which, in the case of recoveries, might require additional authentication steps. Once she obtains all of her shards from the archives, there are two approaches to regenerating those fragments. First, she could try every possible chain of length m, rebuilding the fragment and attempting to verify it. Second, she could narrow the list of possible chains by only attempting to verify chains of length n that represented cycles, an approach we call the *ring heuristic*. As Figure 2 illustrates, fragments include a hash that is used to confirm successful reconstitution. Fragments also include the identifier for the object from which they are derived, making the combination of fragments into objects a straightforward process.

Because the Shamir secret splitting algorithm is computationally expensive, even when combining shards that do not generate valid fragments, we use the ring heuristic to reduce the number of failed reconstitution attempts in two ways. First, the number of cycles of length n is lower than the number of paths of length m since many paths of length n do not make cycles. Second, reconstitution using the Shamir secret splitting algorithm requires that the shares be properly ordered and positioned within the share list. Though the shard ID provides a natural ordering for shards, it does not assist with positioning. For example, suppose the shards were produced with a 3 of 5 split. A chain of three shards, $\langle s_1, s_2, s_3 \rangle$, would potentially need to be submitted to the secret splitting algorithm three times to test each possible order: $\langle s_1, s_2, s_3, \phi, \phi \rangle$, $\langle \phi, s_1, s_2, s_3, \phi \rangle$, and $\langle \phi, \phi, s_1, s_2, s_3 \rangle$.

5 Experimental Evaluation

Our experiments using the current implementation of POTSHARDS were designed to measure several things. First, we wanted to evaluate the performance of the system and identify any bottlenecks. Next, we compared the behavior of the system in an environment with heavy contention for processing and network resources against that in a dedicated, lightly loaded environment. Finally, we evaluated POTSHARDS' ability to recover from the loss of an archive as well as the loss of a user index.

During our experiments, the data transformation component was run from the client's system using object sizes of 750 KB. The first layer of secret splitting used an XOR based algorithm and produced two fragments per object, and the second layer utilized a $(2,3)$ Shamir threshold scheme. The workloads contained a mixture of PDF, Postscript files, and images. These files are representative of the content that a long-term archive might contain, although it is important to note that POTSHARDS sees all objects as the same regardless the objects' origin or content. File sizes ranged from about half a megabyte to several megabytes in size; thus, most were ingested and extracted as multiple objects.

For the local experiments, all systems were located on the same 1 Gbps network with little outside contention for computing or network resources. The client computers were equipped with two 2.74 GHz Pentium 4 processors, 2 GB of RAM and Linux version 2.6.9-22.01.1. Each of the sixteen archives were equipped with two 2.74 GHz Pentium 4 processors, 3 GB of RAM, 7.3 GB of available local hard drive space and Linux version 2.6.9-34. In contrast to the local experiments, the global-scale experiments were conducted using PlanetLab [21], resulting in considerable contention for shared resources. For these experiments, both the clients and archives were run in a slice that contained twelve PlanetLab nodes (eight archives and four clients) distributed across the globe.

The POTSHARDS prototype system itself consists of roughly 15,000 lines of Java 5.0 code. Communications between layers used Java sockets over standard TCP/IP, and the archives used Sleepycat Software's BerkeleyDB version 3.0 for persistent storage of shards.

5.1 Read and Write Performance

Our first set of experiments evaluated the performance of ingestion and extraction on a dedicated set of systems and on PlanetLab. Table 2 profiles the ingestion and extraction of one block of data. comparing the time taken on an unloaded local cluster of machines and the heavily loaded, global scale PlanetLab. In addition to the time, the table details the number of messages exchanged during the request.

As Table 2 shows, most of the time on the local cluster is spent in the transformation layer. This is to be expected as Shamir secret-splitting algorithm is compute-intensive. While slower than many encryption algorithms, such secret-splitting algorithms do not suffer from the problems discussed earlier with long-term encryption and are fast enough for archival storage. The compute-intensive nature of secret-splitting is further highlighted in the local experiments due to the local cluster's dedicated network with almost no outside cross-

Ingestion Profile		Cluster	PlanetLab
Secret Splitting	time (ms)	1509	2276
Layers	msgs in	1	1
Request	msgs out	1	1
Placement	time (ms)	37	30606
Layer	msgs in	1	1
Request	msgs out	6	6
Archive	time (ms)	67	39109
Layer	msgs in	6	6
Request	msgs out	6	6
Response Trip	time (ms)	88	54271
Total Round Trip	time (ms)	1731	95952

Extraction Profile		Cluster	PlanetLab
Request Trip	time (ms)	28	6493
Shard	time (ms)	832	29666
Acquisition	msgs	34	34
Transformation	time (ms)	1009	1698
Layer	msgs in	1	1
Response	msgs out	1	1
Total Round Trip	time (ms)	1843	31410

Table 2: *Profile of the ingestion and extraction of one object, comparing trials run on a lightly-loaded local cluster with the global-scale PlanetLab. Results are the average of 3 runs of 36 blocks per run using a* $(2,2)$ *XOR split to generate fragments and a* $(2,3)$ *Shamir split to generate shards.*

traffic. The transformation time for ingestion is greater than for extraction for two reasons. First, during ingestion, the transformation must generate many random values. Second, during extraction, the transformation layer performs linear interpolation using only those shards that are necessary. That is, given an (m,n) secret split, all n are retrieved but calculation is only done on the first m shards; the minimum required to rebuild the data. During extraction, the speed improvements in the transformation layer are balanced by the time required to collect the requested shards from the archive layer.

In a congested, heavily loaded system, the time to move data through the system begins to dominate the transformation time as the PlanetLab performance figures in Table 2 show. This is evident in the comparable times spent in the transformation layers in the two environments contrasted with the very divergent times spent on requests and responses in the two environments. For example, the extraction request trip took only 28 ms on the local cluster but required about 6.5 seconds on the PlanetLab trials. Since request messages are quite small, the difference is even more dramatic in the shard acquisition times for extraction. Here, moving the shards from the archives to the transformation layer took only 832 ms on the local cluster but over 29.5 seconds on PlanetLab.

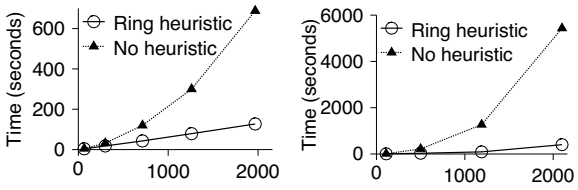
The measurements per object represent two distinct scenarios. The cluster numbers are from a lightly-loaded,

Figure 7: *System throughput with sixteen archives and a workload of 100 MB per client using the same system parameters as in Table 2.*

well-equipped and homogeneous network with unsaturated communication channels. In contrast, the PlanetLab numbers feature far more congestion and resource demands as POTSHARDS contended with other processes for both host and network facilities. However, in archival storage, latency is not as important as throughput. Thus, while these times are not adequate for low-latency applications, they are acceptable for archival storage.

The results from local tests show a per client throughput of 0.50 MB/s extraction and 0.43 MB/s ingestion—per-client performance is largely limited by the current design of the data transformation layer. In the current version, both XOR splitting and linear interpolation splitting is performed in a Java-based implementation; future versions will use $GF(2^{16})$ arithmetic in an optimized C based library. Additionally, clients currently submit objects to the data transformation component and synchronously await a response from the system before submitting the next object. In contrast, the remainder of the system is highly asynchronous. The high level of parallelism in the lower layer is demonstrated in the throughput as the number of clients increases. As Figure 7 shows, the read and write throughput scales as the number of clients increases. With a low number of clients, much of the system's time is spent waiting for a request from the secret splitting layers. As the number of clients increases, however, the system is able to take advantage of the increased aggregate requests of the clients to achieve system throughput of 4.66 MB/s for extraction and 2.86 MB/s for ingestion. Write performance is further improved through the use of asynchronous parity updates. While an ingestion response waits for the archive to write the data before being sent, it does not need to wait for the parity updates.

An additional factor to consider in measuring throughput is the storage blow-up introduced by the two levels of secret splitting. Using parameters of $(2,2)$ XOR splitting and $(2,3)$ shard splitting requires six bytes to be stored for every byte of user data. In our experiments, system throughput is measured from the client perspective even though demands inside the system are six times those



(a) 2 of 3 split.

(b) 3 of 5 split.

Figure 8: *Brute force recovery time for an increasing number of shards generated using different secret splitting parameters.*

Name Space	Shards	False Rings	Time
16 bits	4190	24451	6715 sec
32 bits	4190	0	225 sec

Table 3: *Recovery time in a name space with 5447 allocated names for two different name space sizes. For larger systems, this time increases approximately linearly with system size; name density and secret splitting parameters determine the slope of the line.*

seen by the client. Nonetheless, one goal for future work is to improve system throughput by implementing asynchronous communication in the client.

5.2 User Data Recovery

In the event that the index over a user's shards is lost or damaged, user data, including the index, if it was stored in POTSHARDS, can be recovered from the shards themselves. To begin the procedure, the user authenticates herself to each of the individual archives and obtains all of her shards. The user then applies the algorithm described in Section 4.4 to rebuild the fragments and the objects that make up her data.

We ran experiments to measure the speed of the recovery process for both algorithm options. While the recovery process is not fast enough to use as the sole extraction method, it is fast enough for use as a recovery tool. Figure 8 shows the recovery times for two different secret splitting parameters. Using the ring heuristic provides a near-linear recovery time as the number of shards increases, and is much faster than the naïve approach. In contrast, recovery without using the ring heuristic results in an exponential growth. This is very apparent in Figure 8(b), which must potentially try each path three times. The ring heuristic provides an additional layer of security because a user that can properly authenticate to all of the archives and acquire all of their shards can recover their data very quickly. In contrast, an intruder that cannot acquire all of the needed shards must search in exponential time.

The density of the name space has a large effect on the time required to recover the shards. As shown in Table 3, a sparse name space results in fewer false shard rings (none in this experiment) and is almost 30 times faster than a densely packed name space. An area of future research is to design name allocation policies that balance the recovery times with the security of the shards. One simple option would be to utilize a sliding window into the name space from which names are drawn. As the current window becomes saturated it moves within the name space. This would ensure adequate density for both new names and existing names.

5.3 Archive Reconstruction

The archive recovery mechanisms were run on our local system using eight 1.5 GB archives. Each redundancy group in the experiment contained eight archives encoded using RAID 5. A 25 MB client workload was ingested into the system using $(2,2)$ XOR splitting and $(2,3)$ Shamir splitting, resulting in 150 MB of client shards, excluding the appropriate parity. After the workload was ingested, an archive was failed. We then used a static recovery manager that sent reconstruction requests to all of the available archives and waited for successful responses from a fail-over archive. Once the procedure completed, the contents of the failed archive and the reconstructed archive were compared. This procedure was run three times, recovering at 14.5 MB/s, with the verification proving successful on each trial. The procedure was also run with faults injected into the recovery process to ensure that the verification process was correct.

6 Discussion

While we have designed and implemented an infrastructure that supports secure long-term archival storage without the use of encryption, there are still some outstanding issues. POTSHARDS assumes that individual archives are relatively reliable; however, automated maintenance of large-scale archival storage remains challenging [3]. We plan to explore the construction of archives from autonomous power-managed disk arrays as an alternative to tape [8]. The goal would be devices that can distribute and replicate storage amongst themselves, reducing the level of human intervention to replacing disks when sufficiently many have failed.

A secure, archival system must deal with the often conflicting requirements of maintaining the secrecy of data while also providing a degree of redundancy. To this end, further work will explore the contention between these two demands in such areas as parity building. In future versions, we hope to improve the security of parity updates in which sensitive data must be passed between archives.

Currently, POTSHARDS depends on strong authentication and intrusion detection to keep data safe, but it is not clear how to defend against intrusions that may occur

over many years, even if such attacks are detected. We are exploring approaches that can refactor the data [35] so that partial progress in an intrusion can be erased by making new shards "incompatible" with old shards. Unlike the failure of an encryption algorithm, which would necessitate wholesale re-encryption, refactoring for security could be done over time to limit the window over which a slow attack could succeed. Refactoring could also be applicable to secure migration of data to new storage devices.

We have introduced the approximate pointer mechanism as a means of making data recovery more tractable while maintaining security. While we believe they are useful in this capacity, we admit that there is more work to be done in understanding their nature. Specifically, we plan on exploring the relationship between the ID namespace and approximate pointer parameters.

We would also like to reduce the storage overhead in POTSHARDS, and are considering several approaches to do so. Some information dispersal algorithms may have lower overheads than Shamir secret splitting; we plan to explore their use, assuming that they maintain the information-theoretic security provided by our current algorithm.

The research in POTSHARDS is only concerned with preserving the bits that make up files; understanding the bits is an orthogonal problem that must also be solved. Others have begun to address this problem [9], but maintaining the semantic meanings of bits over decades-long periods may prove to be an even more difficult problem than securely maintaining the bits themselves.

7 Conclusions

This paper introduced POTSHARDS, a system designed to provide secure long-term archival storage to address the new challenges and new security threats posed by archives that must securely preserve data for decades or longer.

In developing POTSHARDS, we made several key contributions to secure long-term data archival. First, we use multiple layers of secret splitting, approximate pointers, and archives located in independent authorization domains to ensure secrecy, shifting security of long-lived data away from a reliance on encryption. The combination of secret splitting and approximate pointers forces an attacker to steal an exponential number of shares in order to reconstitute a single fragment of user data; because he does not know which particular shares are needed, he must obtain *all* of the possibly-required shares. Second, we demonstrated that a user's data can be rebuilt in a relatively short time from the stored shares *only* if sufficiently many pieces can be acquired. Even a sizable (but incomplete) fraction of the stored pieces from a subset of the archives will not leak information, ensuring that data stored in POTSHARDS will remain secret. Third, we made intrusion detection easier by dramatically increasing the amount of information that an attacker would have to steal and requiring a relatively unusual access pattern to mount the attack. Fourth, we ensure long-term data integrity through the use of RAID algorithms across multiple archives, allowing POTSHARDS to utilize heterogeneous storage systems with the ability to recover from failed or defunct archives and a facility to migrate data to newer storage devices.

Our experiments show that the current prototype implementation can store user data at nearly 3 MB/s and retrieve user data at 5 MB/s. Since POTSHARDS is an archival storage system, throughput is more of a concern than latency, and these throughputs exceed typical long-term data creation rates for most environments. Since the storage process is parallelizable, additional clients increase throughput until the archives' maximum throughput is reached; similarly, additional archives linearly increase maximum system throughput.

By addressing the long-term threats to archival data while providing reasonable performance, POTSHARDS provides reliable data protection specifically designed for the unique challenges of secure archival storage. Storing data in POTSHARDS ensures not only that it will remain available for decades to come, but also that it will remain secure and can be recovered by authorized users even if all indexing is lost.

Acknowledgments

We would like to thank our colleagues in the Storage Systems Research Center (SSRC) who provided valuable feedback on the ideas in this paper. This research was supported by the Petascale Data Storage Institute, UCSC/LANL Institute for Scalable Scientific Data Management and by SSRC sponsors including Los Alamos National Lab, Livermore National Lab, Sandia National Lab, Digisense, Hewlett-Packard Laboratories, IBM Research, Intel, LSI Logic, Microsoft Research, Network Appliance, Seagate, Symantec, and Yahoo.

References

[1] Health Information Portability and Accountability act, Oct. 1996.

[2] ADYA, A., BOLOSKY, W. J., CASTRO, M., CHAIKEN, R., CERMAK, G., DOUCEUR, J. R., HOWELL, J., LORCH, J. R., THEIMER, M., AND WATTENHOFER, R. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)* (Boston, MA, Dec. 2002), USENIX.

[3] BAKER, M., SHAH, M., ROSENTHAL, D. S. H., ROUSSOPOULOS, M., MANIATIS, P., GIULI, T., AND BUNGALE, P. A fresh

look at the reliability of long-term digital storage. In *Proceedings of EuroSys 2006* (Apr. 2006), pp. 221–234.

[4] CHANG, F., JI, M., LEUNG, S.-T. A., MACCORMICK, J., PERL, S. E., AND ZHANG, L. Myriad: Cost-effective disaster tolerance. In *Proceedings of the 2002 Conference on File and Storage Technologies (FAST)* (San Francisco, CA, Jan. 2002).

[5] CHOI, S. J., YOUN, H. Y., AND LEE, B. K. An efficient dispersal and encryption scheme for secure distributed information storage. *Lecture Notes in Computer Science 2660* (Jan. 2003), 958–967.

[6] CLARKE, I., SANDBERG, O., WILEY, B., AND HONG, T. W. Freenet: A distributed anonymous information storage and retrieval system. *Lecture Notes in Computer Science 2009* (2001), 46+.

[7] CLEVERSAFE. Highly secure, highly reliable, open source storage solution. Available from http://www.cleversafe.org/, June 2006.

[8] COLARELLI, D., AND GRUNWALD, D. Massive arrays of idle disks for storage archives. In *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing (SC '02)* (Nov. 2002).

[9] GLADNEY, H. M., AND LORIE, R. A. Trustworthy 100-year digital objects: Durable encoding for when it's too late to ask. *ACM Transactions on Information Systems 23*, 3 (July 2005), 299–324.

[10] GOLDBERG, A. V., AND YIANILOS, P. N. Towards an archival intermemory. In *Advances in Digital Libraries ADL'98* (April 1998), pp. 1–9.

[11] GOODSON, G. R., WYLIE, J. J., GANGER, G. R., AND REITER, M. K. Efficient Byzantine-tolerant erasure-coded storage. In *Proceedings of the 2004 Int'l Conference on Dependable Systems and Networking (DSN 2004)* (June 2004).

[12] GUNAWI, H. S., AGRAWAL, N., ARPACI-DUSSEAU, A. C., ARPACI-DUSSEAU, R. H., AND SCHINDLER, J. Deconstructing commodity storage clusters. In *Proceedings of the 32nd Int'l Symposium on Computer Architecture* (June 2005), pp. 60–71.

[13] HAEBERLEN, A., MISLOVE, A., AND DRUSCHEL, P. Glacier: Highly durable, decentralized storage despite massive correlated failures. In *Proceedings of the 2nd Symposium on Networked Systems Design and Implementation (NSDI)* (May 2005).

[14] HAND, S., AND ROSCOE, T. Mnemosyne: Peer-to-peer steganographic storage. *Lecture Notes in Computer Science 2429* (2002), 130–140.

[15] IYENGAR, A., CAHN, R., GARAY, J. A., AND JUTLA, C. Design and implementation of a secure distributed data repository. In *Proceedings of the 14th IFIP International Information Security Conference (SEC '98)* (Sept. 1998), pp. 123–135.

[16] KALLAHALLA, M., RIEDEL, E., SWAMINATHAN, R., WANG, Q., AND FU, K. Plutus: scalable secure file sharing on untrusted storage. In *Proceedings of the Second USENIX Conference on File and Storage Technologies (FAST)* (San Francisco, CA, Mar. 2003), USENIX, pp. 29–42.

[17] KEETON, K., SANTOS, C., BEYER, D., CHASE, J., AND WILKES, J. Designing for disasters. In *Proceedings of the Third USENIX Conference on File and Storage Technologies (FAST)* (San Francisco, CA, Apr. 2004).

[18] MANIATIS, P., ROUSSOPOULOS, M., GIULI, T. J., ROSENTHAL, D. S. H., AND BAKER, M. The LOCKSS peer-to-peer digital preservation system. *ACM Transactions on Computer Systems 23*, 1 (2005), 2–50.

[19] MILLER, E. L., LONG, D. D. E., FREEMAN, W. E., AND REED, B. C. Strong security for network-attached storage. In *Proceedings of the 2002 Conference on File and Storage Technologies (FAST)* (Monterey, CA, Jan. 2002), pp. 1–13.

[20] OXLEY, M. G. (H.R.3763) Sarbanes-Oxley Act of 2002, Feb. 2002.

[21] PETERSON, L., MUIR, S., ROSCOE, T., AND KLINGAMAN, A. PlanetLab Architecture: An Overview. Tech. Rep. PDN–06–031, PlanetLab Consortium, May 2006.

[22] PLANK, J. S. A tutorial on Reed-Solomon coding for fault-tolerance in RAID-like systems. *Software—Practice and Experience (SPE) 27*, 9 (Sept. 1997), 995–1012. Correction in James S. Plank and Ying Ding, Technical Report UT-CS-03-504, U Tennessee, 2003.

[23] QUINLAN, S., AND DORWARD, S. Venti: A new approach to archival storage. In *Proceedings of the 2002 Conference on File and Storage Technologies (FAST)* (Monterey, California, USA, 2002), USENIX, pp. 89–101.

[24] RABIN, M. O. Efficient dispersal of information for security, load balancing, and fault tolerance. *Journal of the ACM 36* (1989), 335–348.

[25] RHEA, S., EATON, P., GEELS, D., WEATHERSPOON, H., ZHAO, B., AND KUBIATOWICZ, J. Pond: the OceanStore prototype. In *Proceedings of the Second USENIX Conference on File and Storage Technologies (FAST)* (Mar. 2003), pp. 1–14.

[26] SANTRY, D. S., FEELEY, M. J., HUTCHINSON, N. C., VEITCH, A. C., CARTON, R. W., AND OFIR, J. Deciding when to forget in the Elephant file system. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP '99)* (Dec. 1999), pp. 110–123.

[27] SCHWARZ, S. J., T., AND MILLER, E. L. Store, forget, and check: Using algebraic signatures to check remotely administered storage. In *Proceedings of the 26th International Conference on Distributed Computing Systems (ICDCS '06)* (Lisboa, Portugal, July 2006), IEEE.

[28] SHAMIR, A. How to share a secret. *Communications of the ACM 22*, 11 (Nov. 1979), 612–613.

[29] STINSON, D. R. *Cryptography Theory and Practice*, 2nd ed. Chapman & Hall/CRC, Boca Raton, FL, 2002.

[30] STONEBRAKER, M., AND SCHLOSS, G. A. Distributed RAID—a new multiple copy algorithm. In *Proceedings of the 6th International Conference on Data Engineering (ICDE '90)* (Feb. 1990), pp. 430–437.

[31] STORER, M., GREENAN, K., MILLER, E. L., AND MALTZAHN, C. POTSHARDS: Storing data for the long-term without encryption. In *Proceedings of the 3rd International IEEE Security in Storage Workshop* (Dec. 2005).

[32] STORER, M. W., GREENAN, K. M., AND MILLER, E. L. Long-term threats to secure archives. In *Proceedings of the 2006 ACM Workshop on Storage Security and Survivability* (Oct. 2006).

[33] SUBBIAH, A., AND BLOUGH, D. M. An approach for fault tolerant and secure data storage in collaborative work environements. In *Proceedings of the 2005 ACM Workshop on Storage Security and Survivability* (Fairfax, VA, Nov. 2005), pp. 84–93.

[34] WALDMAN, M., RUBIN, A. D., AND CRANOR, L. F. Publius: A robust, tamper-evident, censorship-resistant web publishing system. In *Proceedings of the 9th USENIX Security Symposium* (Aug. 2000).

[35] WONG, T. M., WANG, C., AND WING, J. M. Verifiable secret redistribution for threshold sharing schemes. Tech. Rep. CMU-CS-02-114-R, Carnegie Mellon University, Oct. 2002.

[36] WYLIE, J. J., BIGRIGG, M. W., STRUNK, J. D., GANGER, G. R., KILIÇÇÖTE, H., AND KHOSLA, P. K. Survivable storage systems. *IEEE Computer* (Aug. 2000), 61–68.

[37] YOU, L. L., POLLACK, K. T., AND LONG, D. D. E. Deep Store: An archival storage system architecture. In *Proceedings of the 21st International Conference on Data Engineering (ICDE '05)* (Tokyo, Japan, Apr. 2005), IEEE.