Deep Store: An Archival Storage System Architecture

Lawrence L. You, Kristal T. Pollack, and Darrell D. E. Long
University of California, Santa Cruz
Jack Baskin School of Engineering
1156 High Street
Santa Cruz, California 95064

Abstract

We present the Deep Store archival storage architecture, a large-scale storage system that stores immutable data efficiently and reliably for long periods of time. Archived data is stored across a cluster of nodes and recorded to hard disk. The design differentiates itself from traditional file systems by eliminating redundancy within and across files, distributing content for scalability, associating rich metadata with content, and using variable levels of replication based on the importance or degree of dependency of each piece of stored data.

We evaluate the foundations of our design, including PRESIDIO, a virtual content-addressable storage framework with multiple methods for inter-file and intra-file compression that effectively addresses the data-dependent variability of data compression. We measure content and metadata storage efficiency, demonstrate the need for a variable-degree replication model, and provide preliminary results for storage performance.

1. Introduction

The need for large-scale storage systems is becoming obvious; a study estimated that over five exabytes (5×10^{60} bytes) of data was produced [14] in 2002, an increase more than 30% over the previous year. Furthermore, the fraction of data that is *fixed content* or *reference data* continues to increase, accounting for 37% of all stored data, and was expected to surpass mutable data by the end of 2004. This is unsurprising, especially in the face of the over 10,000 legal regulations placed on companies in the U.S. for corporate compliance [25]. The storage for compliance increased by 63% just in 2003, even before some of the most demanding regulations, such as the Sarbanes-Oxley Act, went into effect.

Further compounding the need for archival storage is the increasing volume of material converted into the digital domain. Permanent records archives, in which data is not removed, will only continue to grow. The National Archives and Records Administration (NARA) aim to have 36 petabytes of archival data on-line by the year 2010. As a result of these increased demands for reference storage, for both archival and compliance purposes, it is a rapidly growing area of interest.

Despite the plummeting cost of low-cost consumer storage devices, the cost of managed disk-based storage is high—many times the cost of a storage device itself and higher than tape. As recently as 2002, the cost for enterprise disk storage was over $100 per gigabyte, compared to tape at $10 per gigabyte. A trend for near-line and archival storage is to use cheaper disks, such as ATA devices, instead of SCSI devices, in order to bring down storage cost closer to that of magnetic tape [10].

A new class of storage systems whose purpose is to retain large volumes of immutable data is now evolving. The engineering challenges include: improving scalability, to accommodate growing amounts of archival content; improving space efficiency, to reduce costs; increasing reliability, to preserve data on storage devices with short operational lifetimes and inadequate data integrity for archival storage; and locating and retrieving data from within an archival store. High-performance disk-based storage designs have been evolving to use lower-cost components, but they continue to be expensive to manage.

Metadata will undoubtedly perform an essential role in managing information throughout its lifetime. Data by itself is fragile over the long term because it may be hard to interpret after many years, especially when the systems which created it no longer exist. Future interpretation and presentation of data require rich metadata describing it. Descriptive metadata for each file captures information for enabling diverse search features. Unfortunately, the rich metadata substantially increases file size overhead, when space efficiency is an important goal in reference storage.

The trade-off between space efficiency and redundancy for reliability has always been an issue; however, for archival systems the trade-off is deepened. By the very nature of archives, they will continuously grow over time because data is rarely removed. This creates the need for a space-efficient solution to archival storage. However, the need for a reliable system is also heightened in the case of archival storage since as stored data gets older it is more likely that there will be undetected cases of bit rot, and as devices age the likelihood of their failure grows.

2. System architecture

To address these problems, we propose an archival storage architecture designed to retain large volumes of data efficiently and reliably.

Traditional disk-based file systems, which include direct- or networked-attached storage (DAS/NAS) and storage area networks (SAN), do not have the properties desirable for archival storage. They are designed to have high performance instead of a high level of permanence, to allocate data in blocks instead of maximizing space efficiency, to read and write data instead of storing it immutably, and to provide some security but not to be tamper-resistant. Archival data must be retained for retrieval after a period of time that exceeds the life expectancy of disk-based storage systems hardware and likely to exceed the practical lifetime of the storage system software and their interfaces. Digital data must be stored reliably and automatically managed by the storage system in order to be preserved beyond single failures. In some cases, data lifecycles may include a requirement to destroy certain content after a period of time.

We desire the following properties in an archival storage system that set it apart from file systems: significantly reduced storage cost, immutable properties (write once, read many), cold storage (write once, read rarely), dynamically scalable storage (incremental growth of storage), improved reliability (checksums, active detection, preferential replication), and archival storage compliance (WORM, required duration, lifecycle management).

Additional properties to distinguish Deep Store from other archival storage systems include: much lower latency than the tape systems which it replaces, a simple interface and design, searching capabilities (essential to petabyte-scale storage systems), and accessibility across decades or centuries as well as across local or distributed systems.

2.1. Architectural overview

The Deep Store architecture consists of these primary abstractions: storage objects, physical storage components, a software architecture, and a storage interface. We briefly describe each of these in turn.

The primary storage objects presented to the archival store are the *file* and its *metadata*. A file is a single contiguous stream of binary data. It is identified by its content. A hash function, such as MD5 [22] or SHA-1 [24] digest, is computed over each file to produce the primary portion of its *content address*. Simple metadata associated with the file, such as its filename and length, are contained within a metadata structure. The metadata can also be identified by content address. Content-addressable systems (CAS) systems like EMC Centera store files in this manner [7].

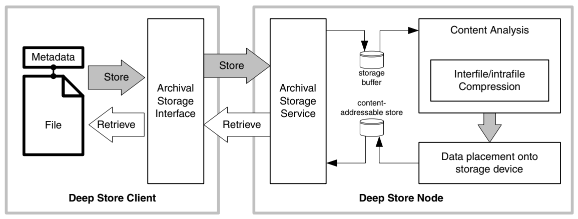


Figure 1. Architectural block diagram

The primary unit for storage is a Deep Store *storage node*. Multiple nodes connect over a low-latency/high-bandwidth network to make up a *storage cluster*. Each node contains a processor, memory, and low-cost disk storage. Content analysis, which is absent from most file systems, includes fingerprinting, compression, and data storage and retrieval; these are necessary operations, but they must have high throughput to be practical. The performance growth rates for silicon-based processors and memories historically have been greater than the capacity and latency improvements of magnetic disk. Considering the ever increasing CPU-I/O gap, this disparity implies a potential performance benefit to effective bandwidth from the reduction of stored content—in other words, compression can benefit I/O performance.

The software architecture is embodied in processes that execute on each storage node: an *archival storage service*, a temporary *storage buffer*, a *content analyzer*, and a *content-addressable store*. The archival storage interface accepts input using a common system interface such as function interface, pipes, sockets, or WebDAV, all of which we have found to be suitable for a simple client-server request mechanism. The storage buffer minimizes request latency; our implementation stores the uncompressed content in a CAS. The content analyzer eliminates redundancy and can also be used for extracting metadata. The efficient CAS stores content and metadata alike.

The storage interface consists of operations on objects, such as files or file metadata, which are addressed by content. The storage operations are:

- Store object
- Retrieve object
- Delete object

- Verify object

This simplicity in design is motivated by the need for long-term preservation, but not at the expense of flexibility. For content-addressable storage of disparate types using different efficient storage methods, we required a uniform storage interface. Toward the goal of long-term preservation, a simpler specification and ease of implementation helps ensure that data written today from one client system can still be read from a completely different client system in 1, 10, or even 100 years. These operations form the basis for other storage operations.

2.2. PRESIDIO

Achieving optimal data compression in large storage systems is difficult due to many factors: variability of redundancy in the stored content, tradeoffs in space efficiency versus performance, and selecting the most appropriate efficient storage method. Redundancy exists in many forms: files with low information entropy, files which are similar to others (including themselves), and the exact duplication of data.

Redundancy-elimination techniques include the commonly known *intra-file compression* or stream-based compressors like *gzip* that work well within a file; and *inter-file compression* techniques like single-instance storage in CAS systems, delta compression, and chunk-based compression, all of which work well across files. Compression rates vary with content, and in some cases inter-file and intra-file compression are complementary [27].

Reference data varies widely in its type, content, and redundancy. Although the original input data created (typed or entered) by humans is relatively small, the aggregate volume is significant [13]. Examples include the creation and distribution of email, hypertext markup (HTML), word processing or office documents (Word, Excel, PowerPoint), and paper-replacement files (PDF). Within the context of a file, template data, such as presentation information, fonts, formatting, and repetitious metadata, are included for completeness but are redundant within the scope of an archive. Computer-generated data such as transactions or logs can have high levels of redundancy due to common strings. Delta compression works well on these data, where differences are fine-grained.

Other data, such as digitized content is often created and immediately compressed: *zip*/*gzip* for general-purpose content, JPEG for visual content, and MPEG/MP3 for audio and video content. Even when static compression coding techniques are used, the content-based symbol probabilities differ from file to file. This makes any contiguous subsection of a compressed representation more unique than similar among files, making it difficult to realize any further compression. In these instances, duplicate suppression of identical data—chunks or whole files—can be effective when the storage space overhead is small.

We introduce PRESIDIO (Progressive Redundancy Elimination of Similar and Identical Data In Objects), a storage compression framework which incorporates multiple efficient storage methods to easily and practically reduce redundancy when possible. The main data types in PRESIDIO are illustrated in Figure 2.

Handle: a file handle that contains a *content address* (CA), such as an MD5 or SHA-1 hash. Our prototype stores only an MD5 hash (16 bytes), but we anticipate that we will augment the handle with a small amount of metadata to resolve collisions.

Constant Data Block: a content-addressable data block of variable size containing a string of bits that is stored literally (*i.e.* raw binary data).

Virtual Data Block: a content-addressable data block. Each block contains a type code such as "constant" (K), "concatenation" ("chunk list," Σ), and "differential" ("delta," Δ). Each block has a content address; the contents are reconstructed polymorphically but stored virtually. Handles can be embedded within other blocks of data. The polymorphic behavior is flexible because it allows a single address to map transparently to multiple instances or alternate representations.

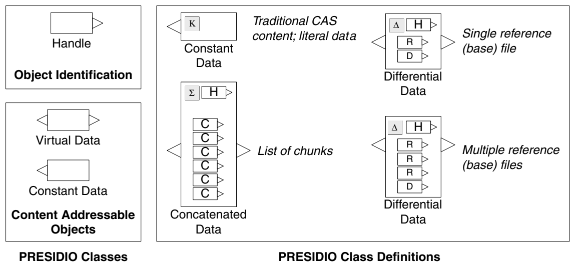


Figure 2. PRESIDIO data classes

Objects are stored by content; each lettered box indicates the content address type: **C** for "chunk," **R** for "reference file" (virtual or real object), and **D** for a "delta file" (also virtual or real). Embedded handles, **H**, contain the hash for the whole file.

2.3. CAS object types

To find a single content-addressable object, first a handle is presented to the Virtual Object Table. The handle's content address is used as a hash key to look up the storage location of the Virtual Data Block that is referenced by the table. The Virtual Data Block is retrieved and the handle is compared for its identity.

Reconstruction of the real image of the data block follows. The framework currently allows for three different types of reconstructible data, or any combination of them.

Constant Data (*K*) If the Virtual Data Block is a Constant Data Block, then the reconstruction is complete and the data block is returned. A variation of this is the Compressed Constant Data Block (*Z* type, not shown), which is compressed using the *zlib* stream compression library.

Concatenated Data (Σ) If the Virtual Data Block is a Concatenated Data Block, reconstruction consists of iterating through the list of handles and retrieving them recursively. The concatenated data is returned serially to the Archival Storage Service.

Differential Data (Δ) If the Virtual Data Block is a Differential Data Block, then reconstruction consists of first reconstructing the Reference Block (R) and the Delta Block (D). A delta reconstruction, using a delta-based compression program such as *xdelta*, applies the delta file to compute the resulting Version Object which is returned to the Archival Storage Service. Note that Version Objects can refer to multiple Reference Blocks.

The framework can be extended to different types of Virtual Data Blocks; for instance, a single version (instance) of a file's metadata can be extracted from a data block storing the entire version history for that file.

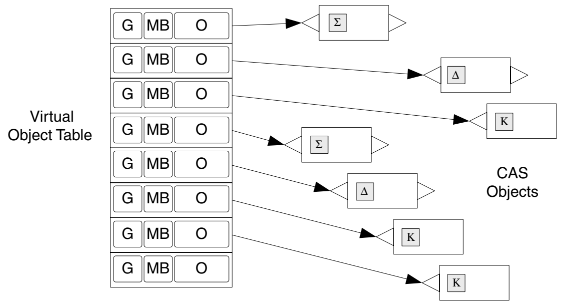


Figure 3. PRESIDIO content-addressable object storage

Figure 3 illustrates the simple relationship between constant data CAS objects and virtual data CAS objects. Object data (files and internal metadata) are stored as small objects. A single handle can be used to store object data. Each entry in the Virtual Object Table consists of a *group* number (**G**), a *megablock* number (**MB**), and an *offset* in the megablock (**O**). (Additional information, such as the block type and length, are not shown.) Our prototype, which uses 16 bit group and megablock identifiers and a 32 bit offset, addresses a maximum of 16 exabytes (18×10^{19} bytes).

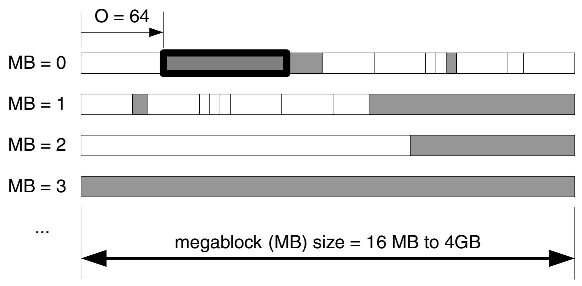


Figure 4. Megablock (MB) storage

Objects are placed within megablocks sequentially to reduce storage overhead from unused portions of blocks and to maximize contiguous writes. Stored data (white boxes) are stored contiguously. Unused data (dark gray boxes) may be present. Megablock fragmentation can be temporary; a periodic *cleaner* operation will reclaim unused space. A fixed megablock size from 16MB to 4GB is selected for uniformity across nodes and the ability of group migration. Compared to file systems, which typically have block sizes in kilobytes, this range of megablock sizes is better matched for large-file storage on cluster file systems such as GPFS [23] and GFS [9]. Files larger than the size of a megablock are divided into chunks smaller than or equal to the size of a megablock and stored as virtually, as a Concatenated Data Block.

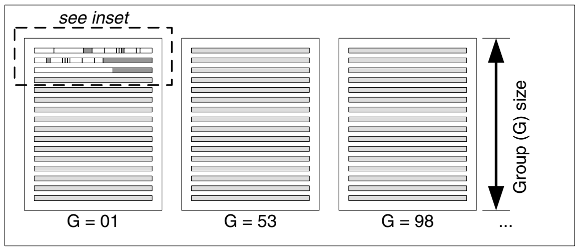


Figure 5. Group (G) and node storage

A storage group, shown in Figure 5, contains a number of megablocks, placed on a recording device using a storage mechanism such as a cluster file system. In this example, the group numbers are 01, 53, and 98. Each group is stored on the node of a server cluster. For reliability, groups can be recorded with varying levels of replication or coding. A small distributed hash table is maintained on each Deep Store node to allow a single node to look-up a node number from the group number. Groups can be migrated from existing nodes to newly added nodes to distribute load. The simple group and megablock structure is easily ported to new large-scale storage systems, and allows group migration to yield efficient storage for a wide distribution of file sizes, including small objects such as file metadata, with very small object-naming overhead. The group also serves to contain a naming space that is unique across a cluster.

2.4. Archival metadata

Metadata for files typically contain fields for information such as create time, ownership, and file size. This information is used by both the file system and users in order to understand the file. In the case of archival storage, archived files must be understood far into the future, by both future file systems and users. An example of file content, metadata, and content addresses is shown in Figure 6.

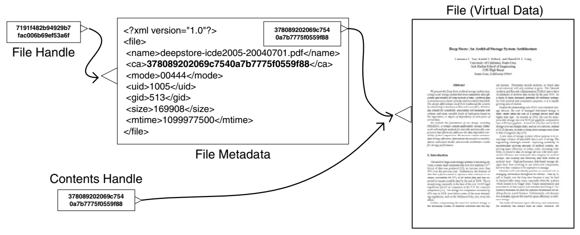

```
<?xml version="1.0"?>
<file>
 <name>deepstore-icde2005-20040701.pdf</name>
 <ca>378089202069c7540a7b7775f0559f88</ca>
 <mode>00444</mode>
 <uid>1005</uid>
 <gid>513</gid>
 <size>169908</size>
 <mtime>1099977500</mtime>
</file>
```
7191f482b94929b7 fac006b69ef53a6f
File Handle
378089202069c754 0a7b7775f0559f88
File Metadata
File (Virtual Data)
Contents Handle
378089202069c754 0a7b7775f0559f88

Figure 6. Metadata and file content

In order to ensure that files can be understood in the future, the metadata must contain a wealth of information about the context, the format, etc. We leave the problem of deciding what fields belong in archival metadata to the archivists. Instead we aim to support archival metadata that is rich and extensible by storing metadata in an XML format. Previous work has also shown this to be an ideal format for archival metadata because of its application independence and widespread use for data exchange [17].

In addition to storing archival metadata in a rich, extensible format we also enable searching and versioning for metadata. Since a large amount of information must be stored in the metadata for future use and understanding, why not take advantage of this information for search and retrieval? Search-enabled storage is especially important for an archival system because it is likely that someone trying to find a file created 100 years previous won't know where to look for it in the system. Some of the most useful information to search over will likely be found in the metadata. For example, a user looking for a file may know the author and roughly what time period it was created in. Versioning of metadata is also useful for archived files, if not essential. Archives are generally concerned with preserving the history and provenance of stored data, so it is consistent to treat metadata with the same consideration. For instance, keywords may be added as new topics become relevant to existing information, or access controls may be changed after some event. Recording these types of changes may prove useful to understanding the context and history of the data even further.

Unfortunately, rich, extensible, versioned metadata opposes the goal of space efficiency. Compressing the metadata seems like a logical solution, but this could add complexity for retrieving files as the metadata would have to be uncompressed before it could be interpreted. Our solution is to store metadata according to its role in the system. First we classify system metadata as the subset of archival metadata that is used for locating and accessing files on the physical storage devices. We then classify some subset of the archival metadata as *search metadata*, metadata that is selected as useful to search across.

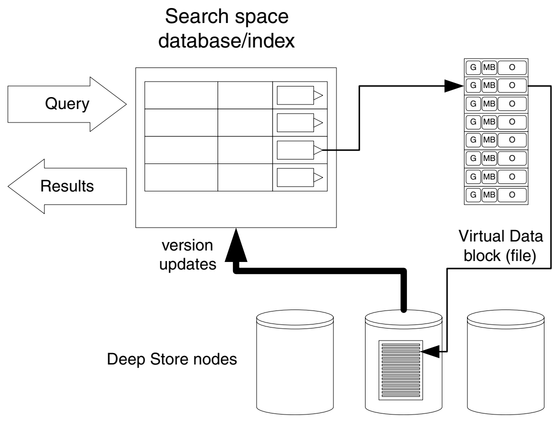


Figure 7. Metadata retrieval and updates

We then store the different types of metadata in structures complementary to their use, as shown in Figure 7. The system metadata is stored in a very fast and efficient look-up structure as it will need to be accessed for every file operation. Search metadata will be stored in a query-enabled structure such as an XML database or search index. Finally the entire set of archival metadata will be stored in the same storage as the archived data itself, versioned and compressed losslessly.

In order to enable space efficient versioning we examined two methods, using delta compression (differencing) between successive versions, and merging versions into a single compact XML tree. An interesting approach to the latter method was developed by Buneman *et al.* [5]. Their approach was to merge versions of an XML tree by labeling the differing nodes with timestamps. This allows elegant time travel within a single tree and reduces the overheads of storing multiple files.

Figure 8 shows experimental results for compressing versioned XML metadata using the differencing and merging approaches we discussed. The data set consisted of 5,880 original files, from which nine versions were created by adding, removing or changing 10% of the file between successive versions. Rather than merging the version changes elegantly as previously discussed, we merged the entire version at each step for simplicity, giving us a slightly worse case scenario. We then used *XMill* [12], an intra-file XML compression tool, to compress the single XML tree containing all the versions. For the delta compression approach we used the tool *xdelta* [15] to compute the differences between each successive version. This means that for each file the first version was stored and then only the differences between the current version and the previous were stored in a new file for the current version. Each version file was then compressed using the intra-file compression tool *gzip* [8] rather than *XMill* since the differenced version files were not valid XML trees. We can see from Figure 8 that merging the versions and then compressing the result using *XMill* (xmill-appended-versions) is enormously more space efficient than the differencing approach (delta-chain-versions) as the number of versions increase. We also show the results for just compressing each individual version with *XMill* (xmill-all-versions) to show that the savings are not simply an effect of *XMill* being more efficient than *gzip*. The merging versions method works so well because *XMill* uses dictionary compression, so the compression ratio dramatically increases as the file gets larger since the data added is very similar, and only one dictionary is created for all the versions. By intelligently merging the versions [5] we should be able to achieve even better space efficiency, as well as a simple method for "time travel" between versions. One drawback to this method that may make it infeasible in some cases is that the metadata must be stored on mutable storage in order to keep all the versions in a single XML file. This may be unacceptable for some systems depending on the storage medium and level of security that is required; however, the differencing method could be used in these cases.

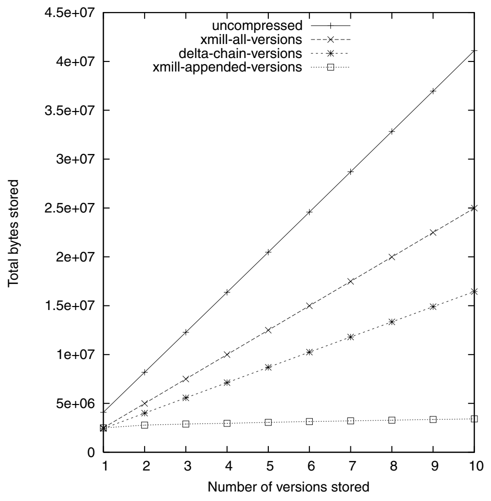


Figure 8. Metadata versions

3. Evaluating the architecture

One of the main goals of the architecture is to allow data to be stored efficiently. The PRESIDIO framework allows multiple lossless data compression methods to use a common object storage mechanism. The immutable property of archived data allows PRESIDIO to address data by content. Efficient storage techniques are variations on a common theme: they first identify data that is identical or similar across files, and then they suppress the storage of redundant information. The scope and resolution of this strategy varies widely, but PRESIDIO ties the different schemes together in a common framework.

We first describe common features of efficient storage methods and then contrast their differences and how they fit into the framework.

3.1. Efficient archival storage methods

A basic trade-off of data compression is increasing computation and I/O against the potential for higher space efficiency. Each method also lies on a spectrum for the resolution at which they can detect similarity. Although these methods do not have any *a priori* knowledge of the data type or format of the files that are stored, customized applications may further expand the spectrum with application-specific knowledge of the content. When storing a file we take advantage of this spectrum, beginning at the computationally inexpensive end and progressively working our way towards the expensive end until we find a method that yields acceptable compression. The methods using little CPU and I/O resources detect large blocks of identical data. If similarity is not found at a coarse level then we must use progressively more expensive techniques to find similarity at a finer grain and eliminate redundancy down to the level of a byte.

The common steps used to efficiently store files in a process employed by the PRESIDIO framework are:

feature selection selecting features (fingerprints or hashes) from a chunk or file

similarity detection locating a similar or identical chunk or file already in storage

redundancy elimination suppressing duplicate storage or compressing data

recording writing the compressed data to storage

All methods perform feature selection as a first step. By first computing a small representation of the entire file, the search space for similar data is dramatically reduced. A small feature that covers a large section of a file is compact and has low overhead but might not be tolerant of small differences between similar files. A large feature set that represents a large number of smaller features from a file may be information that is useful for finding files with some similarity but at a significant amount of indexing space or searching time.

Once a candidate file's feature set has been computed, it can be used to find similar data that has already been stored. The methods vary widely and have different processing and storage requirements.

The compression step includes one or more methods, typically a combination of suppressed storage, stream compression, and delta compression.

Finally, storage is suppressed, reference counts to objects are incremented, and new non-redundant data objects are stored. In some cases computing the feature set is also the step for determining what to store, and in others further computation is necessary.

3.2. Comparison of efficient storage methods

We examine four different efficient storage methods: *whole-file hashing*, *subfile-chunking*, *delta compression between similar files*, and *hybrid chunk and delta techniques*. Each efficient archival storage method has different costs and benefit.

We highlight the significant differences between the different storage methods.

Whole-file hashing A simple content-addressable storage scheme is to use *whole-file hashing*. A hash function or *digest* computes (selects) a single feature from the entire file. Digest functions like MD5 or SHA-1 have good collision-resistant properties (as well as a cryptographic property). The digest can then be used to identify the content-addressable object.

There are several advantages to this method. Computing a digest is fast at 83 MB/s for SHA-1 and 227 MB/s for MD5 on our test hardware (Intel Pentium 4, non-HT 2.66 GHz, 533 MHz FSB, PC2100 ECC memory). Only one feature needs to be computed, and a single hash key can be used to look up a file in constant or nearly constant time. Many files are identical, and so when they are detected, the amount of savings is tremendous. However, the main shortcoming of this method is that there are a significant number of files that differ partially and would not be detected for redundancy elimination.

Subfile chunking Files are divided into variable-sized chunks with a minimum size and a maximum size. Blocks are divided deterministically by sliding a window and creating a division when the sliding window's hash value matches a criteria, so as to meet a desired distribution of block sizes [18]. In many cases the following criteria is sufficient: compute the Rabin fingerprint [21] $fp = f(A)$ (f is a Rabin fingerprint function using a preselected irreducible polynomial of a fixed-size window of size w over a string A) and evaluating when the integer expression $fp \bmod D$ is equal to a constant R. By selecting a divisor D, the criteria will be met approximately one in D times. Lower and upper limits on the block size can be set to reduce the variance in the block size.

As chunk boundaries are determined, the chunk's content address is computed at the same time, *i.e.* per increment of the sliding window. The feature set is made up of a list of chunk identifiers. Using the same hardware as above, we measured our chunking program *chc* [27] for feature selection alone (determining boundaries and then computing MD5 over the chunk) at 36 MB/s using 32-bit Rabin fingerprints, $D = 1,024$, $R = 0$, $w = 32$, with minimum and maximum chunk sizes set at 64 and 16,384 bytes, respectively.

Each chunk is then stored individually, in a manner similar to whole file hashing. Individual chunks are stored with low overhead in the PRESIDIO content-addressable object store. Identical chunks are identified in time that is linear with the number of chunks. While the finer granularity is better than whole-file hashing, two sets of hashes must be computed at the same time, and data within a block must still be identical in order for hash values to match.

Fixed-sized blocks are a degenerate case of chunking when the minimum and maximum chunk size are equal. While this condition eliminates the need for a chunk-dividing algorithm, resemblance detection of identical blocks will be lower [19]. Shortcomings of chunking are that chunk identifier lists must be stored, even when no redundancy exists, adding unnecessary overhead; this storage overhead is small, but when the average chunk size is large (1 kilobyte and larger), efficiency degrades [27].

Delta compression between similar files Delta encoding [1], an excellent inter-file compression method, is a compelling mechanism for reducing storage within a corpus that stores similar files. Delta compression is used to compute a delta encoding between a new *version file* and a *reference file* already stored in the system. When resemblance, an estimate of the similarity between two files, is above a predetermined threshold, a delta is calculated and only that is stored in the system. There are three key steps that need to be designed and implemented for efficiency to delta compress between similar files (DCSF).

First, features are selected from files in a content-independent and efficient way. We use the shingling technique (DERD) by Douglis *et al.* [6], which calculates Rabin fingerprints [2] over a sliding window on byte boundaries along an entire file. The window size, w is a preselected parameter. The number of intermediate fingerprints produced is proportional to the file size. To reduce it to a manageable size, a deterministic *feature selection* algorithm selects a fixed size (k) subset of those fingerprints (using *approximate min-wise independent permutations* [3]) into a *sketch*, which is retained and later used to compute an estimate of the *resemblance* between two files by comparing two sketches. This estimate of similarity is computed between two files by counting the number of matching pairs of features between two sketches. Douglis has shown that even small sketches, *e.g.* sets of 20 features, capture sufficient degrees of resemblance. Our experiments also show sketch sizes between 16 and 64 features using 32-bit fingerprints to produce nearly identical compression efficiency. Using the same hardware as above, we measured our feature selection program at 19.7 MB/s, $k = 16$, reading a 100 MB input file.

Second, when new data needs to be stored, the system finds an appropriate reference file in the system: a file exhibiting a high degree of resemblance with the new data. In general, this is a computationally intensive task (especially given the expected size of archival data repositories). We are currently investigating feature-based hashing to reduce the search complexity. Our method differs from DERD by allowing delta chains of length greater than one, by storing and detecting similar files incrementally to more closely match a growing archive. We used sketch sizes of 16 features ($k = 16$) and sliding window size of 24 bytes ($w = 24$).

Third, deltas are computed between similar files. Storage efficiency from delta is directly related to the resemblance. Highly similar files are more effective at reducing storage usage than mildly similar ones. Fortunately, it is possible to reduce the comparison between pairs of fingerprints in feature sets (16 or more fingerprints each) down to a smaller number of features that are combined into "super" features, or superfingerprints/supershingles [4]. Each superfingerprint is a hash of a subset of the features. If one or more superfingprints match, there is high probability of a high similarity. REBL [11], by Kulkarni *et al.*, computed superfingerprints over chunks with 1 KB to 4 KB average size to yield large numbers of superfingerprints to detect files with high resemblance. Because our design constraints such as the number of expected files and real memory prevent this method from being directly applicable, we are evaluating a solution to meet those requirements. We measured *xdelta* at 8.7 MB/s (20 MB total input size, 2.29 seconds) for worst case performance: a 10 MB input file of zeros and a 10 MB file of random data, producing a 10 MB output file. Delta compression programs with higher performance are known to exist [26] and can be used in practice.

The fourth step, common to all efficient storage methods, is storing the compressed data. In DCSF, the *delta file* is recorded to the CAS.

3.3. Measurements

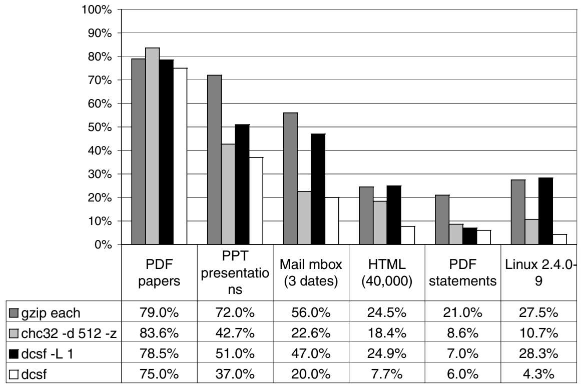
To evaluate our expected storage efficiency, we compressed six data sets using stream compression (*gzip*), chunking (*chc32*), and delta compression between similar files (*dcsf*), measuring the total (not just incremental) compressed size. Figure 9 shows these measurements as a percentage of the original data.

To illustrate the range of redundancy in data, we selected data sets which are likely to be archived, binary and textual data, and small and large files, and dissimilar as well as highly similar data. Table 1 list for each set its total size in megabytes, number of files, and average file size in bytes.

Table 1. Data sets

Name	size (MB)	# files	avg. size (B)
PDF papers	239	754	331,908
PPT presentations	63	91	722,261
Mail mbox (3 dates)	837	383	2,291,208
HTML	545	40,000	14,276
PDF statements	14	77	186,401
Linux 2.4.0-9	1,028	88,323	12,209

We briefly describe the content in these data sets. *PDF papers* are PDF documents of technical papers (conference proceedings, journal articles, and technical reports) in a bibliographic research archive. *PPT presentations* are Microsoft PowerPoint presentation files of related work. *Mail mbox* are email folder snapshots for a single user on three separate dates. The *HTML* set comes from the *zdelta* benchmark [26]. *PDF statements* are monthly financial statements from investment institutions. And *Linux 2.4.0-9* is ten versions of the Linux kernel source code, 2.4.0 through 2.4.9.



	PDF papers	PPT presentatio ns	Mail mbox (3 dates)	HTML (40,000)	PDF statements	Linux 2.4.0-9
▨ gzip each	79.0%	72.0%	56.0%	24.5%	21.0%	27.5%
▨ chc32 -d 512 -z	83.6%	42.7%	22.6%	18.4%	8.6%	10.7%
■ dcsf -L 1	78.5%	51.0%	47.0%	24.9%	7.0%	28.3%
□ dcsf	75.0%	37.0%	20.0%	7.7%	6.0%	4.3%

Figure 9. Storage efficiency by method

Sizes were measured in the following manner. The *gzip* compressor was applied on each file with default parameters, and all file sizes were added to produce the compressed size. The *chc* program read a *tar* file containing all input files and produced a single chunk archive, using a divisor of 512 bytes ($D = 512$), compressing each chunk with the *zlib* stream compressor; the total size is a sum of the single instances of compressed chunks and a chunk list. The *dcsf* method computed delta using *xdelta* (version 1.1.3 with standard options that use the *zlib* compressor), selecting the best file with a threshold of at least one matching fingerprint in the sketch; reference and non-matching files were compressed with *gzip* and the measured size was the sum of all of these files. The `-L 1` option sets a maximum delta chain length of one, *i.e.* deltas are only computed against reference files. This avoids chains of reconstruction, but at the expense of lower space efficiency.

3.4. Measuring the benefit of high resemblance data

To help understand the importance of file similarity on the storage efficiency, we conducted experiments to compute the relationship between resemblance (an estimate of file similarity) and the actual amount of storage that is used. The amount of data similarity varied widely, and so did the amount of compression. However, some behaviors were common, such as the relationship of storage resemblance and the inter-file compressibility of data. Using the Linux source code data set (ten versions, 2.4.0–2.4.9), we ran *dcsf* to assess the importance of high resemblance data. Our experiments would store all files from version 2.4.0 first in order, then 2.4.1, etc. evaluating each file individually against previously stored files. The file size after *gzip* (intra-file compression only) was compared against *xdelta* (both intra- and inter-file compression) and total storage was tabulated by resemblance.

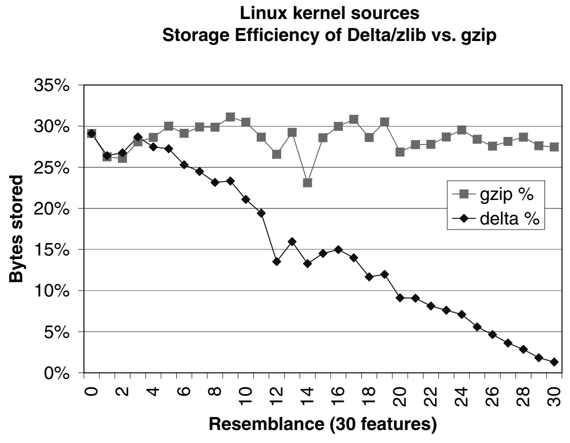


Figure 10. Delta efficiency by resemblance

In Figure 10, the horizontal axis lists the (discrete) resemblance: a number of features that were matched and for each corresponding number of matching features (out of 30). On the vertical axis, the amount of space that was necessary to store the files with that resemblance. The *delta* graph shows the storage efficiency improving (decreasing) as the resemblance increases. This confirms the relationship between resemblance (an estimate) and delta (a computed difference between two files). By comparison, the *gzip* graph is relatively flat, ranging from approximately 25% to 30%. The reduction in storage illustrates the complementary benefit of intra-file and inter-file compression.

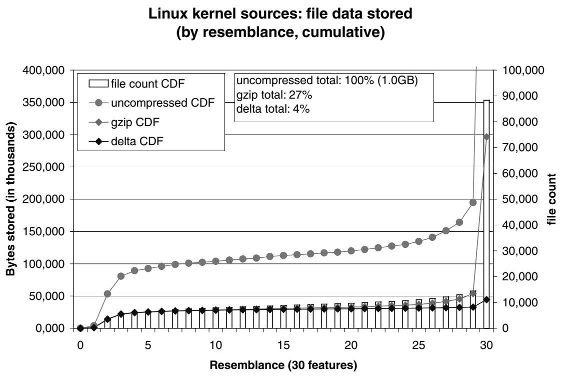


Figure 11. Cumulative data stored, by resemblance

Using the same data, Figure 11 shows a different view of the storage efficiency that demonstrates the importance of finding identical or highly similar data. The resemblance is still on the horizontal axis, but two sets of data are superimposed. The file count (bar graph) shows the number of files that are in the workload with a given resemblance. The other lines show both uncompressed size, the size of the data set when each file is compressed with *gzip*, and finally the delta-compressed size using *xdelta*. File counts and sizes are cumulative of all files with lower resemblance.

With 88,323 files and one gigabyte of data, a significant number of files have very high similarity, in fact many are identical. The amount of storage required for *gzip* is only 27% but with *delta*, the total amount of storage is 4% of the original, uncompressed source code. The relative flatness of the *delta* plot near 30 of 30 features shows only a slight increase in storage space despite the large numbers of copies that were stored.

What is important to note is that the major benefit comes from files that have high resemblance (30 out of 30 matching features, which is equivalent to all superfingerprints matching). PRESIDIO's progressive feature matching process would first attempt to match identical files, then highly similar files, and then finally somewhat similar files. The time to search the first two categories is relatively fast and requires direct lookup of whole files or chunks instead of a full pair-wise feature resemblance across a large set of files.

Using these techniques, we have measured a number of data sets which have high variation in the amount of inter-file and intra-file redundancy. High levels of compression are possible, including computer-generated data that was measured to less than 1% of the original size [27]. By not attempting to fit a single compression scheme to all data, and providing a framework for one or more schemes, the Deep Store architecture benefits a wide range of data types.

4. Reliability

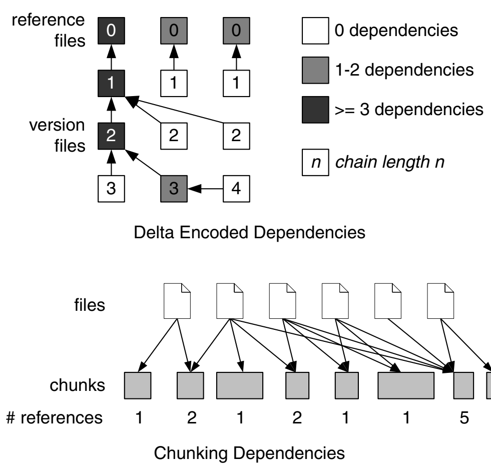


Figure 12. Dependency graphs

The problem of reliability becomes much more interesting when data is stored in a compressed format. How do we reintroduce redundancy (for reliability) after we have worked so hard to remove it? Now that files share data due to inter-file compression, a small device failure may result in a disproportionately large data loss if a heavily shared piece of data is on the failed component. This makes some pieces of data inherently more valuable than others. In the case of using delta compression for stored files, a file may be the reference file for a number of files, and in turn those may be a reference file for another set of files. A single file may quickly become the root of a large dependency graph, as shown in Figure 12. If a file for which other files depended on was lost, all the files with a path to it in the dependency graph would be lost as well. Figure 13 shows the chain lengths for a real set of highly similar data (ten Linux kernel sources) stored using PRESIDIO with only delta compression. The average chain length for the set of files was 5.83, but more interestingly, there were only five reference files (chain length zero) for this file set. In other words, in the set of 88,323 files, all of the files depended on five files.

Sub-file chunking also forms inter-file dependencies. The dependencies created when using this method are

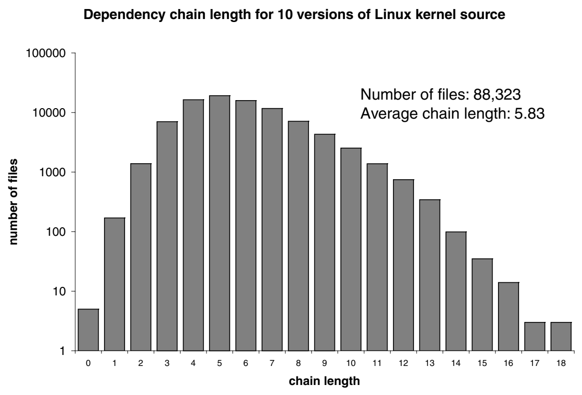


Figure 13. Delta chain length

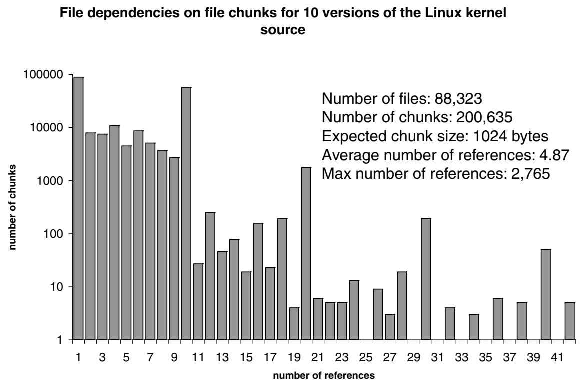


Figure 14. Chunk dependencies

shown in Figure 12. If a large set of files all contained the same chunk, for example a regularly occurring sequence in a set of log files, the loss of this small chunk would result in the loss of a large set of files. Figure 14 shows the number of files dependent on shared chunks for the same set of files used in Figure 13, stored using PRESIDIO with only chunk-based compression. The large peak for chunks with one reference shows the number of unique chunks, and the peaks seen at 10, 20, 30 and 40 occur since the files are 10 versions of the Linux kernel source. Of the 200,635 chunks stored, only 89,181 were unique, with an average of five files referencing a chunk. The data extends out to 2,765 references to a single chunk, however, it was truncated to 41 for the graph in the interest of space. There were 72 chunks not shown with more than 41 references, 17 of which were over 100.

For our reliability model we place a value on a chunk or a file based on how many files would be affected if it were lost. So we would consider the five reference files in our first example and our chunk with 2,765 references from our second example to be more valuable than less referenced data. To protect the more valuable data we would like to store it with a higher level of redundancy than less valuable data in order to preserve space efficiency while minimizing the risk of a catastrophic failure. Another issue that must be considered for preventing catastrophic failure is that of data distribution. Say we were to distribute files and/or chunks randomly across devices. If we were to lose an entire device the effect would likely be devastating. Depending on the number of devices and the degree of interdependence of the data, it would be likely that a file in the system would have a chunk of data lost, or a missing file in its delta chain, preventing future reconstruction. It is clear that reliability guarantees for a system storing data with inter-file dependencies is a particularly difficult problem. Building a model that derives the value of data and developing a strategy for data placement are the subject of future work.

5. Status and future work

We have investigated and experimented with a number of archival storage problems that are detailed above. Armed with initial experimental data that shows the potential for addressing some of the archival storage problems, our goal is to prototype the system with a single node and then expand it to multiple nodes. We have evaluated delta compression for its storage efficiency and are working to first detect very similar data and then low similarity data with low time and space overhead. Because content analysis can be both CPU and I/O intensive, but not necessarily at the same time, we have determined process scheduling and data pipelining will be necessary for high average throughput. The megablock CAS implementation is under experimentation using a database to map handles to stored data and simple files on a cluster file system to contain the data. The large-block storage, content-addressable storage, and distribution of work sets the groundwork for scalability on a much larger scale.

We continue to evaluate the problems with data reliability, including using both erasure coding and mirroring depending on the degree and performance requirements for the types of data. We are also looking at providing security for a system that is extended temporally. This is also a difficult problem because the common security designs used in systems today may not extend over the decades that are necessary for the lifetime of archival data storage.

6. Related work

Current archival storage systems are commercially available and under research, typically configured as network attached systems that are used in a manner similar to more traditional network file systems. EMC Corporation's Centera on-line archival storage system [7] uses *content-addressable storage* (CAS), identifying files with 128-bit hash values. Each file with identical hash is stored just once (plus a mirror copy for fault tolerance). Venti [20] provides archival storage with write-once characteristics. Venti views files as fixed-size blocks and hierarchies of blocks that can help reconstruct entire files. Versions of files are stored at fixed locations and common data is not shared unless they are identical on fixed block boundaries. In order to detect redundancies at a finer grain we divide files into variable size chunks as was done in LBFS [18]. Chunks can be uniquely identified by a hash of their contents, allowing only duplicate chunks to have the same address. LBFS used this idea to eliminate unnecessary network transmission, both client and server avoid sending data chunks if they are already found in a cache on the other end.

Our similarity detection for delta compression builds off a large body of previous work. Initially Manber used fingerprints to determine file similarity on a single volume, computing the entire contents of the volume as a single operation or incrementally as new files are changed [16]. Later Douglis and Iyengar evaluated the efficiency of storage by using delta encoding via resemblance detection (DERD) [6] over a number of parameters, showing the relationship between similarity and delta encoding sizes, and establishing a number of parameters such as resemblance thresholds, feature set sizes, and the number of delta encodings to compute to determine the best match.

7. Conclusions

We have presented the basis for a new storage architecture capable of storing large volumes of immutable data efficiently and reliably. The Deep Store model for storage uses a new architecture consisting of abstractions for data objects, a content analysis phase that includes PRESIDIO, a new lossless data compression framework that incorporates inter-file compression with a content-addressable object storage mechanism. We presented a method for metadata storage that provides extensibility, versioning and searching while still maintaining the goal of overall space-efficiency. We also proposed a model for reliability of files with shared data using variable levels of redundancy. We then presented data that helps to show the feasibility of a scalable storage system that eliminates redundancy in stored files.

8. Acknowledgements

This research was supported by a grant from Hewlett-Packard Laboratories (via CITRIS), Microsoft Research with a matching grant from University of California MICRO, Veritas, and by National Science Foundation Grant CCR-0310888. Additional support for the Storage Systems Research Center was provided by Engenio, Hitachi Global

Storage Technologies, IBM Research, Intel, and Network Appliance.

Hewlett-Packard Laboratories has provided a generous gift of a ProLiant server cluster used for our experiments. IBM Almaden Research provided their GPFS cluster file system software.

We thank Thomas Schwarz for his insight into reliability issues in large storage systems and Svetlana Kagan for her help on block-based file storage and client interfaces. We are also grateful to members of the Storage Systems Research Center at the University of California, Santa Cruz for their discussions and comments.

References

[1] M. Ajtai, R. Burns, R. Fagin, D. D. E. Long, and L. Stockmeyer. Compactly encoding unstructured inputs with differential compression. *Journal of the ACM*, 49(3):318–367, May 2002.

[2] A. Z. Broder. Some applications of Rabin's fingerprinting method. In R. Capocelli, A. D. Santis, and U. Vaccaro, editors, *Sequences II: Methods in Communications, Security, and Computer Science*, pages 143–152. Springer-Verlag, 1993.

[3] A. Z. Broder, M. Charikar, A. M. Frieze, and M. Mitzenmacher. Min-wise independent permutations. *Journal of Computer and Systems Sciences*, 60(3):630–659, 2000.

[4] A. Z. Broder, S. C. Glassman, M. S. Manasse, and G. Zweig. Syntactic clustering of the web. In *Proceedings of the 6th International World Wide Web Conference*, pages 391–404, Santa Clara, California, Apr. 1997.

[5] P. Buneman, S. Khanna, K. Tajima, and W. C. Tan. Archiving scientific data. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, Madison, Wisconsin, 2002.

[6] F. Douglis and A. Iyengar. Application-specific delta-encoding via resemblance detection. In *Proceedings of the 2003 USENIX Annual Technical Conference*, San Antonio, Texas, June 2003.

[7] EMC Corporation. EMC Centera: Content Addressed Storage System, Data Sheet. `http://www.emc.com/pdf/products/centera/centera\_ds.pdf`, Apr. 2002.

[8] Free Software Foundation. `http://www.gnu.org/software/gzip/gzip.html`, 2000.

[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, New York, Oct. 2003. ACM.

[10] J. Gray and P. Shenoy. Rules of thumb in data engineering. In *Proceedings of the 16th International Conference on Data Engineering (ICDE '00)*, pages 3–12, San Diego, California, Mar. 2000. IEEE.

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

[12] H. Liefke and D. Suciu. XMill: An efficient compressor for XML data. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, pages 153–164, May 2000.

[13] P. Lyman, H. R. Varian, J. Dunn, A. Strygin, and K. Searingen. How much information? 2000. `http://www.sims.berkeley.edu/research/projects/how-much-info/`, Oct. 2000.

[14] P. Lyman, H. R. Varian, K. Searingen, P. Charles, N. Good, L. L. Jordan, and J. Pal. How much information? 2003. `http://www.sims.berkeley.edu/research/projects/how-much-info-2003/`, Oct. 2003.

[15] J. P. MacDonald. xdelta 1.1.3. `http://sourceforge.net/projects/xdelta/`.

[16] U. Manber. Finding similar files in a large file system. Technical Report TR93-33, Department of Computer Science, The University of Arizona, Tucson, Arizona, Oct. 1993.

[17] R. Moore, C. Barua, A. Rajasekar, B. Ludaescher, R. Marciano, M. Wan, W. Schroeder, and A. Gupta. Collection-based persistent digital archives - part 1. `http://www.dlib.org/dlib/march00/moore/03moore-pt1.html`, Mar. 2000.

[18] 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, Lake Louise, Alberta, Canada, Oct. 2001.

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

[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)*, pages 89–101, Monterey, California, USA, 2002. USENIX.

[21] M. O. Rabin. Fingerprinting by random polynomials. Technical Report TR-15-81, Center for Research in Computing Technology, Harvard University, 1981.

[22] R. Rivest. The MD5 message-digest algorithm. Request For Comments (RFC) 1321, IETF, Apr. 1992.

[23] 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.

[24] Secure hash standard. FIPS 180-2, National Institute of Standards and Technology, Aug. 2002.

[25] The Enterprise Storage Group. Compliance: The effect on information management and the storage industry. `http://www.enterprisestoragegroup.com/`, 2003.

[26] D. Trendafilov, N. Memon, and T. Suel. zdelta: An efficient delta compression tool. Technical Report TR-CIS-2002-02, Polytechnic University, June 2002.

[27] L. L. You and C. Karamanolis. Evaluation of efficient archival storage techniques. In *Proceedings of the 21st IEEE / 12th NASA Goddard Conference on Mass Storage Systems and Technologies*, pages 227–232, College Park, Maryland, Apr. 2004.