PRESIDIO: A Framework for Efficient Archival Data Storage

LAWRENCE L. YOU, KRISTAL T. POLLACK, and DARRELL D. E. LONG,
University of California, Santa Cruz
K. GOPINATH, Indian Institute of Science, Bangalore

The ever-increasing volume of archival data that needs to be reliably retained for long periods of time and the decreasing costs of disk storage, memory, and processing have motivated the design of low-cost, high-efficiency disk-based storage systems. However, *managed* disk storage is still expensive. To further lower the cost, redundancy can be eliminated with the use of interfile and intrafile data compression. However, it is not clear what the optimal strategy for compressing data is, given the diverse collections of data.

To create a scalable archival storage system that efficiently stores diverse data, we present PRESIDIO, a framework that selects from different space-reduction efficent storage methods (ESMs) to detect similarity and reduce or eliminate redundancy when storing objects. In addition, the framework uses a virtualized content addressable store (VCAS) that hides from the user the complexity of knowing which space-efficient techniques are used, including chunk-based deduplication or delta compression. Storing and retrieving objects are polymorphic operations independent of their content-based address. A new technique, harmonic super-fingerprinting, is also used for obtaining successively more accurate (but also more costly) measures of similarity to identify the existing objects in a very large data set that are most similar to an incoming new object.

The PRESIDIO design, when reported earlier, had comprehensively introduced for the first time the notion of deduplication, which is now being offered as a service in storage systems by major vendors. As an aid to the design of such systems, we evaluate and present various parameters that affect the efficiency of a storage system using empirical data.

Categories and Subject Descriptors: D.4.2 [**Operating Systems**]: Storage Management; E.2 [**Data**]: Data Storage Representations—*Object representation*; E.5 [**Data**]: Files—*Organization/structure*

General Terms: Algorithms, Design

Additional Key Words and Phrases: Archival storage systems, data compression, content-addressable storage, CAS, progressive compression

ACM Reference Format:
You, L. L., Pollack, K. T., Long, D. D. E., and Gopinath, K. 2011. PRESIDIO: A framework for efficient archival data storage. ACM Trans. Storage 7, 2, Article 6 (July 2011), 60 pages.
DOI = 10.1145/1970348.1970351 http://doi.acm.org/10.1145/1970348.1970351

1. INTRODUCTION

Following on our work to improve network efficiency for backup [Burns and Long 1997a], we started to examine data storage efficiency. In 2002, we examined existing

---

This work was conducted by all the authors at the University of California, Santa Cruz, and includes part of L. L. You's dissertation.

Authors' addresses: L. L. You, Google Inc., Mountain View, CA 94043; K. T. Pollack; email: kristal@cs.ucsc.edu; D. D. E. Long, Storage Systems Research Center, University of California, Santa Cruz, CA 95064; K. Gopinath, Department of Computer Science and Automation, Indian Institute of Science, Bangalore 560 012, India; email: gopi@csa.iisc.ernet.in.

DOI 10.1145/1970348.1970351 http://doi.acm.org/10.1145/1970348.1970351

digital storage systems and discovered that the production of information and system capacities were increasing at a dramatic rate, the per-byte media costs were dropping, and that more of the information being produced, including archival data, was moving into the digital domain [Long 2002]. Users expected more from their online storage systems, including lower costs, higher reliability, and higher accessibility (both in latency and throughput), and to add increasingly diverse data formats. Additionally, addressing data "by content" [EMC Corporation 2002; Quinlan and Dorward 2002] presented new opportunities to identify and store common data. We anticipated these new demands would transfer to archival storage systems, thus motivating our work to develop cheaper, online archival storage systems and started the Deep Store project. Within that project we developed PRESIDIO, a framework to support hybrid data compression techniques in order to optimize reducing redundancy across heterogeneous data.

The amount of data that is created and that must be stored continues to grow [Lyman et al. 2003]. Archival storage systems must retain large volumes of data reliably over long periods of time at a low cost. Archival storage requirements and the type of source material vary widely, with the latter varying from being highly compressed to highly redundant. Achieving regulatory compliance, for example, involves storing many similar textual financial documents or emails, whereas digital audio, video, and image are usually stored in a compressed format. Data compression algorithms vary in the granularity at which redundant data is identified, in the computational complexity of detecting similar or identical data, in the method to encode compressed data, and in the performance with multiple stages of compression. A major challenge when applying these algorithms to heterogenous data is to determine the best data compression, as this is data dependent.

Digital archival storage systems can store data efficiently by eliminating redundancy, but detecting the redundancy, may come at a cost. Data content must be analyzed so that identical or similar data can be identified, and redundancy must be encoded or suppressed. Stored data must also be retrieved reliably and quickly. Content analysis can use significant computing and bandwidth resources. Some compression schemes reduce the storage needed significantly with low time overhead, while others have only moderate benefits. Efficient storage systems generate internal storage metadata that is used to detect similar or identical data. However, if little or no redundancy exists, the additional overhead of producing and searching the metadata is wasteful. The PRESIDIO framework was designed to determine a method of highest compression per file by balancing the tradeoff between reduction of file size and the computation time required to do so based on an exhaustive search across all previously stored files.

Prioritizing space efficiency in an archival storage system affects other areas of design. For instance, reducing redundancy may also reduce reliability due to increased dependence on common data; in contrast, storage system designs that increase data reliability do so by increasing storage requirements through a combination of data replication or encoding. Another area of impact on the design is caused by information retrieval whose performance is directly related to the organization of stored data, which may depend on the type of data compression used. Less tangible qualities are also important, such as the difficulty with which archival storage systems can be engineered to store data today as well as re-engineered to access data archives after many generations of obsoleted systems.

In this article, we investigate the fundamental problem of compressing data within a large-scale repository. Data compression over a large-scale corpus trades-off the cost of detecting similar or identical data at fine or coarse granularity against the effectiveness of the the detection methods and data compression algorithms. Data

compression can be lossless compression within streams, lossy compression for audio or video content, or compression of web/network or disk-based content. Data compression algorithms are as diverse as types of applications and their output data. However, reducing the redundancy existing in a large corpus continues to be a challenging problem. Many storage systems employ data compression [Alvarez 2010; Douglis and Iyengar 2003; EMC Corporation 2002; Kulkarni et al. 2004; Ouyang et al. 2002; Quinlan and Dorward 2002; Trendafilov et al. 2004; Zhu et al. 2008], and more recently introduced interfile compression.

The PRESIDIO framework is founded on the understanding that data compression exists across a spectrum. At one end, if file data entropy is very high, virtually no similarity detection will help; however, redundancy, can still exist when multiple copies of files exist. In this case, large-grained feature extraction results in small numbers of features that can be detected quickly and accurately. At the other end, a file may be a slight variation of an another: content such as human-readable text, computer-generated data, or modifications of previously stored data. In these cases, when data similarity exists, feature selection must be more fine-grained. PRESIDIO uses multiple storage methods to reduce redundancy across several domains: high and low entropy data, identical and similar data, as well as intrafile (within a file) or interfile (across a corpus) compression.

1.1 Contributions

Our main contributions are as follows.

— PRESIDIO, a *Progressive Redundancy Elimination of Similar and Identical Data In Objects* storage framework, that allows multiple different efficient storage methods (ESMs) to be applied to incoming data objects, to find the most efficient way to add them to the archive. This is a polymorphic, class-based approach, which makes the framework extensible.
— The ability of each ESM to compute a measure of how effectively it can reduce the space needed to store the new object, and then have the framework choose the ESM that appears likely to have the most benefit.
— A technique, called *harmonic superfingerprinting*, for using successively more accurate (but also more costly) measures of similarity to identify the existing objects (across a very large data set) that are most similar to an incoming new object.
— A virtualized content addressable store (VCAS) that increases transparency to objects by hiding the complexity of the space-efficient storage encoding method (chunking, delta-compression or something else) was used to store an object, and allows it to be retrieved in full by a single address, that is independent of how it is stored.

Fundamentally, we want to store large volumes of immutable data permanently, efficiently, with high reliability and accessibility. Our original approach for "finding similar data" in a corpus was to find a solution to more general classes of problems: how to store metadata information in a traditional file system to facilitate search; how to identify clusters of data and then find similar clusters; how to index and retrieve similar data using traditional information retrieval and web search techniques. We initially proposed using shingling (fingerprints computed over sliding windows over sequences of bytes) and data clustering methods. Clustering algorithms are numerous [Jain et al. 1999] and have been used in search engines, but in a system where storage frugality and incremental storage are primary goals, the clustering methods available today are not well matched for our problem statement. Although we first envisioned clustered, or "associative" storage to be the solution, we have turned toward heuristic engineering solutions that are low-cost and high-return using approximate metrics. Concepts

such as harmonic superfingerprinting (Section 4.4) were developed after realizing that a generalized indexed retrieval method would be costly and that most of the benefit in data compression comes from highly similar data.

Given the widely divergent approaches of methods such as chunking and delta compression, it is not clear which is better. Chunking is attractive due to its simple data identification model and chunk retrieval model. With chunk hierarchies, similar data can be shared through chunk lists or trees. However, delta compression can express fine-grained differences very efficiently. Experiments to settle the issue were inconclusive however, leading us to embrace not just both the methods but many other possible ones into a common storage framework. But before doing so, we need ways to find similar data regardless of the data compression method being used.

In this article, we explore and develop novel techniques for use as a foundation in the design of large-scale storage systems, especially for archival purposes. We progress through a number of compression schemes to find the most beneficial one with the least cost. Our thesis is that a large-scale scalable archival storage system can efficiently store diverse data by applying data compression algorithms progressively in a single storage framework to provide better space efficiency than any single extant storage compression method. The solution involves a system that identifies similar and identical data, methods to eliminate redundancy using one or more space-efficent storage methods, and a low-level content-addressable storage system into which data is recorded. The degree of similarity between new data requested for storage and previously stored data has a direct relationship on redundancy, and therefore the compression rate. Depending on similarity, different types of compression algorithms are evaluated in a progressive manner by executing each in turn to maximize storage benefit while minimizing computational and I/O cost.

1.2 Outline of the Article

In Section 2, we provide the background to the problem and related work. Section 3 is an overview of our solution, PRESIDIO, which includes a description of a prototype implementation and a summary of our evaluation. Section 4 describes and evaluates content analysis algorithms and data structures in feature selection and for measuring similarity and finding similar data. Section 5 describes redundancy elimination methods. Section 6 describes the unified content-addressable storage subsystem used by PRESIDIO to record and reconstruct data. Section 7 describes the algorithms to compress data progressively using the PRESIDIO algorithm and framework. In conclusion, Section 8 summarizes our work and briefly discusses avenues for future work.

2. BACKGROUND

Current archival storage techniques attempt to exploit the lower costs of disk storage, the increased density of memories, and easy availability of low-cost computing devices.

The storage industry has evolved a new class of storage systems whose purpose is to retain large volumes of immutable data. 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 relatively short operational lifetimes and inadequate data integrity for archival storage; and locating and retrieving data from within an archival store.

When designing storage systems, there is a natural tension between improving space efficiency and improving reliability; however, for archival systems the tradeoffs deepen. The very nature of archives cause them to 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

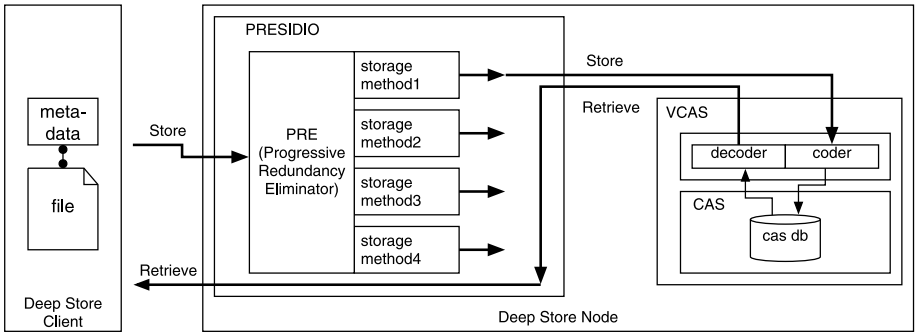


Fig. 1. The Deep Store archival storage system model and PRESIDIO.

storage; 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. Efficient archival storage systems pose a problem: existing reliability models do not consider metrics or have the means to compute the rate of loss of recorded data based on the degree of data dependence.

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, as much as ten times capital costs [Zadok et al. 2003], and higher than tape. A trend for near-line and archival storage is to use cheaper disks. When disk-based storage devices were first considered, ATA (now called Parallel ATA) was used over the higher performance, but more costly, SCSI devices. Now SATA (Serial ATA) is used. These cheaper hard disk technologies are selected in order to bring down the natural storage cost closer to that of magnetic tape [Gray and Shenoy 2000; Gray et al. 2002].

However, 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. To address these problems, we have proposed a new storage architecture for archival storage system, the Deep Store, designed to retain large volumes of data efficiently and reliably.

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).

At the foundation of Deep Store, illustrated in Figure 1, is PRESIDIO, an efficient storage subsystem whose architecture presents a simple content-addressable storage interface, a compression algorithm that applies and evaluates multiple methods progressively for storage efficiency, and a low-level virtual content-addressable storage framework that encodes and decodes content depending on the storage method used. Aspects that 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 availability across decades or centuries as well as across local or distributed systems.

The design of a new storage system is affected by other requirements or considerations that will have impact on the primary goal of reducing the volume of stored data.

Among them are three areas: metadata, reliability, and security, which we discuss briefly in Section 6.

2.1 Space-Efficient Archival Storage Systems

Archival storage systems typically have been at the slowest end of the memory storage hierarchy: *primary storage* in the form of random-access main memory, *secondary storage* in the form of random-access magnetic disk, and finally *tertiary storage* in the form of magnetic tape or optical disk. Hierarchical storage management spans these systems by automatically migrating files through the hierarchy [Gibson 1998].

Efficient archival storage requires tools to help identify and then eliminate redundant data. Traditional tertiary storage media, especially tape, do little to eliminate redundancy due their physical constraints: they are good at sequential access but poor at random access. Storage designers, therefore, had to extract the most out of stream-based compressors. The IBM Tivoli Storage Manager [IBM 2005] pioneered the use of delta compression in tape-based backup; its shortcoming is that redundancy is eliminated for versions of files, and not across the entire dataset being backed up [Burns and Long 1997a]. Furthermore, the high latency for accessing tapes forced delta chains—the length of the dependency graph formed from delta compression operations—to a small length; *version jumping* [Burns and Long 1997a] helped limit chain length at a modest cost in efficiency.

Today, the storage industry aims to further reduce storage resource usage by removing redundant data (duplicates) and have coined the term *data deduplication* to describe it. We next describe major design considerations and techniques that are used to eliminate exact or near duplicates of data.

2.1.1 Identifying Redundant Data. In order to further exploit and eliminate redundancy, we must first be able to identify it. Identification of similar and identical data relies on hashing, fingerprinting, and digest algorithms that make it possible to deterministically compute a unique name for a sequence of data bytes that is much smaller. Using features, data fingerprints, and feature selection, software tools used for information retrieval are used to help identify similar and identical data within a large corpus. We define *feature data* as a substring of data: this may be a whole file, a variable-sized chunk, a fixed-size chunk, or a sliding window. *Storage objects* include files, variable- or fixed-sized blocks from a file, or other content including metadata. From these objects, we compute features to identify them. *Features* are fingerprints of feature data. *Fingerprinting* computes a hash, or fingerprint, over a feature string.

An underlying theme in our experiments has been the wide variety of data that is stored as archival data; but finding a single type of feature that can apply to all data is challenging. While content-specific methods exist, such as those which identify similarity between text-based web pages [Broder et al. 1997; Dean and Henziger 1999], they are not suitable for our storage problem, as they are not designed for nontextual data. Large numbers of smaller features and content-dependent features such as text extraction improve resemblance detection, but may increase storage overhead or may not apply to other classes of data.

2.1.2 Chunking: Dividing Data Streams by Content. Streams of data can be divided into non-overlapping strings of contiguous data, called *chunks*, to identify instances of identical data. For example, the *rsync* program synchronizes specific pairs of remotely located files efficiently [Tridgell 1999]. The receiver computes both strong and fast signatures as a hash over a fixed-size block, such as a MD5 digest, and sends it to the sender. The sender computes *fast signatures* as fingerprints over sliding windows and compares them with those received for a match. A file can be efficiently processed, as

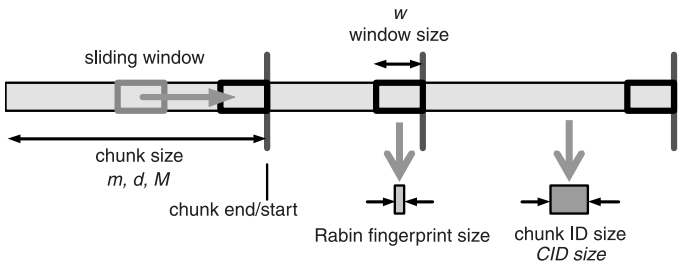


Fig. 2. A graphical representation of chunking parameters. Typically the minimum chunk size m is 64, the maximum M 16384, with expected chunk size d 1024 bytes.

the fast signature can be incrementally computed at each position in the file using only a small number of instructions, and the strong signature can be computed as a digest over a block of hundreds to thousands of bytes.

Our work uses the chunking technique (Figure 2) first used by the Low-Bandwidth Network File System (LBFS) [Muthitacharoen et al. 2001]. in which data chunks are delimited by *breakpoints* located deterministically from the data content. One such method is to compute a hash function over a sliding window of a few tens of bytes and then to select the breakpoint when the integer hash value, or the value modulo an integer constant, or *divisor*, is a specific value. LBFS computes the hash over a window efficiently using Rabin fingerprints of 48-byte length and selecting special values of these fingerprints, say 0, as breakpoints. The data chunks are identified by their SHA-1 hash values.

A shortcoming of many chunking implementations is due to the tradeoff when fixing the parameters so that the chunking algorithm is computed deterministically across all data. This is to ensure that identical chunks can be found when new files are added to a system. Unfortunately, a fixed set of parameters does not compress all data optimally. Furthermore, chunk metadata (chunk lists) increases storage overhead, which makes it less effective than whole-file CAS when unique files are stored.

2.1.3 Stream Compression. Lossless *stream compression* takes an input stream and emits a (generally) smaller output stream by eliminating redundancy that is seen within the stream itself. Repeated strings of input symbols are converted to codes and then output. Codes, which are of varying size, are selected statically or dynamically and assigned to represent symbols, depending on the probability of their occurrence [Nelson and Gailly 1996; Sayood 2003]. Symbol encoding algorithms include *Huffman coding*, in which symbols are assigned codes based on static or dynamic probabilities, and *arithmetic coding*, in which probabilities of the occurrence of symbols are represented by subdivisions of a fractional numerical space. Earlier stream compression algorithms were limited by the size of the window over which they could compute common substrings to eliminate. To address this shortcoming for compressing large files *rzip* [Tridgell 1999] and *lrzip* [Kolivas 2010] use the chunking method employed in *rsync* [Tridgell 1999] as a first pass algorithm before applying stream compression, however this method is still limited to a single stream. Our program, *chc*, described in Section 5.1 also eliminates redundancy across lengthy streams.

Stream compressors can encode identical data with different codes. Static coders, such as Huffman encoding, depend on the input. Dynamic coders, which update encodings as the probability of symbol and substring appearance, adapt to input. In either case, unless a static coder is also using a permanently assigned encoding, the encoded (and compressed) data is not guaranteed to contain identical regions even if

two uncompressed regions are identical. Therefore, stream compression can interfere with interfile compression if it is applied early.

Many data compression implementations exist including the popular programs *zip*, *compress*, *lharc*, *gzip* [Free Software Foundation 2000] (based on Lempel–Ziv compression [Ziv and Lempel 1977]), and *bzip2* [Seward 2002] (based on Burrows–Wheeler compression [Burrows and Wheeler 1994]). We selected *zlib*, the library used in *gzip*, as a reference for several reasons: *zlib* is well known and is a good baseline for comparison, and it provides good compression while using less CPU resources than more space-efficient algorithms like *bzip2*, and therefore is good for experimentation. Finally, it is easily replaced by other stream compressors, and the replacement be trivally implmented to recompress chunks or delta encodings in place if needed.

Structured Data. Structured data, which can be stored into tabular or columnar formats found in serialized records, tables, or databases, may exhibt naturally occurring redundancy. With *a priori* knowledge of such structure, columnar storage can be reduced. Serialized records can similarly be reorganized to re-encode data such as with PZip [Buchsbaum et al. 2000, 2003], Vczip [Vo 2007], and RadixZip [Vo and Manku 2007]. Tabular storage systems, such as Google's Bigtable [Chang et al. 2006] storage system, achieve compression by selecting parts of a two-dimensional storage table for improved locality and then compressing within the local region. Application-level archival storage has been used at the application level to process archival queries effectively; the DNA product [SAND Technology 2009] claims reductions of data feeds up to 98%. The inherent structure of these systems offer application-specific opportunities for compression; however these content-aware systems do not have universal application to arbitrary file data.

Delta Compression (Differential Compression). Our work seeks to improve storage system efficiency by eliminating identical or similar data across files. Since whole files or large contiguous regions of files may only be similar and not identical, we turn to delta compression to express file encodings as differences between two files. *Delta compression*, or *differential compression*, is a method for computing differences between two sources of data in order to produce a small *delta* encoding that represents the changes from one data source to another. Generally, these data streams contain arbitrary content (i.e., not just text) and are files. Methods to compute differences between files are tailored to file contents such as binary data or text. Delta encoding has been used for storage, for in-place updates of data [Burns and Long 1998], and for reducing network bandwidth usage [Mogul et al. 1997]. When chunking indicates a stored file but shows little or no resemblance to an existing file, or when the most space-efficient method is desired, then delta compression should be used. We are further encouraged by the performance of delta compression, whose encoding can be in linear time, using constant space [Ajtai et al 2002; Burns 1996; Burns and Long 1997b].

Early programs to compute differences operated on text files, such as SCCS [Rochkind 1975], to efficiently encode deltas, or changes, between versions. One commonly used program for computing differences still used today is the *diff* tool [Hunt and McIlroy 1976], used in the RCS version control system [Tichy 1985]. Differences were computed at the line level, unlike binary delta that does not have well-defined characters defining units of text. SVN [Apache Subversion 2010], a more recent version control system, expands deltas to encode differences across trees of files.

2.1.4 Delta Compression and Resemblance Detection. Douglis and Iyengar [2003] studied the efficiency of compression with delta encoding, and the use of shingles (fingeprints of overlapping sequences of bytes) to dynamically detect resemblance across files ("delta

encoding via resemblance detection," DERD) over a number of parameters. The redundancy elimination at the block level (REBL) scheme improves on the performance of DERD by using superfingerprints (fingerprints of features selected from files) over chunks with 1KB to 4 KB average size to detect similar data [Kulkarni et al. 2004]. When superfingerprints are used to detect similar blocks in highly similar files, REBL compresses well, but for larger data sets of less similar files, the benefits are not as evident. The measurements reported focus on resemblance detection; however, computing and selecting a large set of fingerprints for the purpose of fingerprinting must also be taken into account as well as the overhead of file metadata and its efficient storage.

Ouyang et al. [2002] study efficient delta compression across a corpus of data but do not address incremental additions of data to the corpus.

2.1.5 Content-Addressable Storage. Deduplication. Some modern storage systems use *content-addressable storage* (CAS) to identify files by content. *Content-Derived Names* (CDN) was an early example of a CAS-like system using probabilistically unique hashes to identify files for configuration management [Hollingsworth and Miller 1997]. Centera [EMC Corporation 2002] an online network-attached archival storage system uses 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 [Quinlan and Dorward 2002] provides archival storage with write-once characteristics. Venti views files as fixed-size blocks and hierarchies of blocks that can help reconstruct entire files. Similarly, Storage Tank, an IBM SAN file system, employed duplicate data elimination (DDE) over mutable data by coalescing blocks that were identified using content-addressable storage [Hong et al. 2004]. NetApp has also incorporated a similar method into WAFL by using the checksums they store for blocks to detect duplicate data [Alvarez 2010; Lewis 2008].

Content-addressable storage can be extended to data of arbitrary length with chunks. Chunks are also identified and stored by their content address. Data Domain uses this technique in their Deduplication File System [Zhu et al. 2008]. Similarly, HydraFS [Ungureanu et al. 2010] provides deduplication by using a variable-sized chunking strategy to increase the likelihood of duplicate chunks, while implementing a standard file system interface. These chunks are stored in a distributed content-addressable storage system, HYDRAstor [Dubnicki et al. 2009], which uses a distributed hash table strategy for performance and fault-tolerance.

Recent refinements in deduplicating storage systems reduce resource usage and improve throughput. Lillibridge et al. [2009] reduce chunk-indexing RAM usage by defining *segments* of contiguous chunks and sampling chunks to create a *sparse index*. The system deduplicates using heuristic scoring of similar segments. Their design exploits chunk locality to preserve recall. Our harmonic superfingerprinting also prefers recall over precision at a chunk level; and both sparse indexing and superfingerprinting are complementary in their approaches to improving index retrieval. Koller and Rangaswami [2010] consider a storage deduplication optimization that utilizes content similarity for improving I/O performance by eliminating I/O operations and reducing the mechanical delays during I/O operations. This is however different from our work, which is archival storage per se, and which is not especially concerned with making I/O efficient on general workloads. Yang et al. [2009] consider a scalable high-performance deduplication storage system for backup and archival. Building on Data Domain's solution [Zhu et al. 2008], their solution has a two-phase deduplication scheme that exploits memory cache and disk index properties to turn the highly random and small disk I/Os resulting from fingerprint lookups and updates into large sequential disk I/Os, thus achieving a scalable deduplication throughput. Their

techniques are complementary to ours, and hence can be incorporated in our system also, if suitable.

2.1.6 Versioning File Systems; Differencing Directories. Versioning file systems such as the Elephant file system [Santry et al. 1999] and the Write-Anywhere File Layout (WAFL) [Hitz et al. 1994] are similar to archival storage systems in that they retain multiple versions of files and do so by storing data, typically blocks, that are not changed between versions or snapshots. The *rsync* file-copying tool can also improve network transmission efficiency by detecting differences between related files. TAPER [Jain et al. 2005] uses additional methods to synchronize between *hierarchical hash trees* (directories and subdirectories) and detects file similarity using Bloom filters. However, unlike our work, some systems do not automatically detect similar or identical data across an archival data store unless they are stored within a versioned, structured file system; they do not exploit efficiencies due to delta compression.

3. PRESIDIO OVERVIEW

We start by presenting an overview of the solution, which describes the PRESIDIO progressive redundancy elimination and its relationship to the Deep Store architecture. Next, we describe a prototype implementation of PRESIDIO that was used to help validate our design.

3.1 Solution Overview

The best strategy for efficiently storing different forms of compressed data is not directly evident. A wide variance in behavior, for instance, in the average size of a stored file or object, or the amount of redundancy that exists in input files, can have significant impact on the effectiveness of a storage system. When little or no redundancy exists in data, overhead should be minimized; otherwise it makes little sense to use a "compression" scheme that increases storage usage. Likewise, if a storage system does not compress highly redundant data well for a particular application, resource costs remain high, and users of the application will seek customized compression methods like domain-specific methods, including lossy compression for digitized media. The development of a unified object storage mechanism improves the ability of an efficient archival storage system to record data for a wide variety of data types and content.

The PRESIDIO solution strategy is to apply progressively efficient data compression methods that meet performance criteria over large volumes of data, using a single content-addressable storage subsystem. Before proceeding to encode the input data, PRESIDIO uses several heuristic tests to determine the probability of finding exact or similar data. When exact data is not found by using a fast hashing test, then another test is applied. If highly-similar data is not found using a hashing test with fingerprints, then another test is applied. And these tests are continued until they are exhausted or certain performance thresholds are exceeded. In this manner, probabilities for high similarity are progressively evaluated and discharged. Once the tests for probability are evaluated, then coding (redundancy elimination and recoding to disk) is performed.

PRESIDIO uses a data compression model comprised of existing and novel technologies. First, the data compression uses a much broader context, which includes all previously stored data within the archive, a technique now commonplace in the deduplication storage industry. Second, because PRESIDIO is a hybrid compression algorithm, it depends on multiple algorithms whose probabilities of detecting redundant data differ. Third, codes are small compared to the large amounts of data that are stored, leading to high rates of compression when redundancy exists. Fourth, in contrast to many data compression formats, PRESIDIO virtual content-addressable

(VCAS) storage is a nonlinear format that supports hybrid methods for reconstruction. Finally, because our solution uses a hybrid compression scheme, the stages for compression and decompression are defined by the PRESIDIO framework.

3.1.1 Large-Scale Data Compression. We use the term *large-scale data compression* to include both interfile and intrafile data compression. However, interfile compression may affect intrafile compression when used together. For example, when using *chunking* algorithms, which subdivide files based on some selected property of the contents of a a window of the file (see Section 2.1.2), the size of contiguous uncompressed data in a chunk to be compressed is smaller than a file. Because stream compressors work to amortize the cost of collecting and storing internal dictionary data across an entire file, when a file is subdivided, the amortizing efficiency may be diminished. Clever design, such as Data Domain's Stream-Informed Segment Layout [Zhu et al. 2008] exploits on-disk locality by laying out multiple segments in sequence and compressing over them with a Ziv–Lempel [Ziv and Lempel 1977] algorithm. In other instances, such as suppressed storage of identical data or delta-compressed data, intrafile compression is independent of interfile compression.

The general model of data compression consists of a *model* and *coder* [Nelson and Gailly 1996]. The model predicts or defines the probability of the source, and the coder produces the code based on the probabilities. The number of stages for compression in our solution depend on the type:

— *intrafile compression* consists of two stages, the *model* and *coder*; and
— *interfile compression* consists of four stages, the *feature selection* and *resemblance detection* stages, followed by *redundancy elimination* and *recording to disk*.

These two forms of data compression overlap in their operation. The model attempts to identify symbols that can be re-encoded, and then the coder encodes the data. Each method can be used effectively for different types of redundancy. Research in stream compression has produced a wide range of space- and time-efficient algorithms that are effective at reducing stream sizes. These algorithms have tradeoffs in compression and decompression speed as well as the rate of compression [Witten et al. 1999]. Likewise, interfile compression performance—in all dimensions—is subject to tradeoffs. In order to best compress data in a large-scale storage system, we use a combination of these methods to provide the highest reduction of storage with the lowest amount of space and time overhead.

Combining different forms of compression is useful. For instance, stream compression, which is restricted to examining and eliminating redundancy from the symbols seen within a stream, does nothing to eliminate copies of the same stream or file. Likewise, suppressing duplicate storage of files does not compress data within the files. When possible, we maximize complementary compression methods when it improves results. Data compression performance is data-dependent. From an information-theoretic viewpoint, the data presented to a storage system is arbitrary and cn exhibit both high and low entropy; our aim is to use a wider number of tools to find and exploit the spectrum.

3.1.2 Efficient Encoding of Data. Encoding of data is the stage of data compression that reduces the stored data by retaining smaller codes to represent the uncompressed data. Designing space-efficient codes is important for the representation of data in lossless stream compression because they must be included in the output; their size matters significantly.

Our solution uses content-addressable storage with strings of data that range from hundreds of bytes to megabytes in size and are addressable by their content. *Content*

addresses, which are of fixed size in the range of 16 to 20 bytes, are small enough to be used as references and do not require directory organization to disambiguate names. A single content address can address an arbitrary amount of data. Existing content-addressable storage systems suppress storage of duplicates, but our solution also encodes slightly different data with low overhead. Programs to compute a *content address* (CA), a *hash* or *digest* over a variable-length file, can do so at higher throughput than the bandwidth of common magnetic disks. Cryptographic or one-way hash functions improve tamper-resistance because modifications to contents invalidate the address; and SHA-1 [National Institute of Standards and Technology 2008] (20 byte) digests are considered an industry "best practice" for maintaining data integrity for regulatory compliance [Security Innovation 2006]. For large volumes of data, the intentionally random distribution of content addresses digests—an artifact of cryptographic design goals—allows storage system designers to create scalable addressing architectures so that file data is easily distributed across storage nodes or devices.

However, CAS systems also have a few shortcomings. An entire file must be scanned before the address can be computed, thus introducing a serialization dependency. Many cryptographic digest algorithms require chaining, so files must be scanned from start to end. Some CAS systems are designed with the assumption that low probabilities of collision of content addresses (hash values) are acceptable. The collision-avoidance property of hash functions distributes addresses across the entire CA space, but by doing so, it eliminates any locality of name or reference. Hence, a CAS storage organization based purely on content addresses may cause more random accesses than sequential ones, thereby decreasing performance on disk-based systems. Also, CAS space efficiency depends on factors such as the average size of stored objects, the size of metadata specified externally by users or applications as well as internally by the CAS itself, the type of compression methods being used, and most importantly, the level of redundancy in the data being stored.

The main themes employed within our solution are the following: use small content addresses, use low overhead to store data, and store and reconstruct data indirectly through a virtual Content-Addressable Storage (VCAS). Despite some of their shortcomings, we choose content addresses for their ability to self-identify entire files, and have chosen space-efficient design goals over traditional disk-based storage systems that value performance, to use a single storage system that includes (and compresses) metadata as well as content. Simple content-addressable storage, which stores full copies of instances, is not efficient for storing a wide variety of compressed data. The VCAS presents a single CAS programming interface with a framework for storing and retrieving data polymorphically, and recording both small and large content-addressable data blocks into a unified CAS. The following description of the VCAS outlines the virtual data representation and then the storage operations.

Virtual Data Representation. The VCAS stores data internally that is transparent to the larger archival storage system. Data transparency [Rajasekar and Moore 2001] is useful for archival storage systems, either by creating abstraction layers or protocols to hide underlying complexity. The VCAS achieves this by appearing to be a standard CAS, but internally it stores the content-addressable data objects more efficiently. This design solves a number of problems. First, objects stored in the VCAS are addressed by content, as in a regular CAS. Addresses are computed by hashing the contents of the object being stored with a high probability of uniqueness.

Second, objects are stored with little metadata overhead. This permits the CAS to be used with small objects. In some applications, small files will be stored. In other instances, efficient storage methods in PRESIDIO may store sections or encodings that

are smaller than the average file. Low overhead also allows storing completely unique data without incurring the overhead a regular CAS or file system would impose.

Third, the content address (CA) data type is typical, 16 to 20 bytes, and similar to other CAS systems. Because the CAs are used internally and externally, the address size contributes to the storage overhead.

Fourth, objects are stored virtually. The storage and reconstruction processes use simple rules to construct arbitrarily complex internal representations of the stored data. Virtual storage is flexible enough to provide literal storage, that is, traditional CAS operations where single, raw or stream-compressed instances of objects are stored in their entirety, as well as subdivided subfile chunks or delta-encoded storage. Content addresses are embedded within the virtual representations; the VCAS interprets the virtual objects and the content addresses during storage and retrieval operations through polymorphic behavior. During a storage request, the VCAS converts input, or concrete representations, into virtual CAS objects. During a retrieval request, the VCAS converts the virtual representation back into the concrete representation. VCAS also allows us to store more than one copy (useful for caching) and with different coder representations (useful for keeping alternate versions of cached fully reconstructed files to improve performance). The VCAS offers other desirable properties: the content address is used to ensure the integrity of the object, and the polymorphic behavior allows the VCAS to be extensible to use other internal virtual representations. Lastly, internal and external object metadata are stored using the same CAS as the object content metadata. File metadata are stored as VCAS objects, as are internal data to help the large-scale data compression methods.

VCAS Storage Operations. Our VCAS subsystem presents a simple external storage interface so that specification, implementation, verification, and accessibility far in the future are possible. The operations provided should be flexible, to allow a variety of data and file formats to be stored and without limitations such as those placed on metadata like filenames or on structures like singletons of flat files. They should also be *complete*, to provide an end-to-end guarantee that all data can be stored and that all retrieved data is identical to the stored data.

The storage interface consists of operations on objects, such as files or file metadata, that are addressed by content. The storage operations store and retrieve an object. This simplicity in design is motivated by the need for long-term preservation: data written today by a client system should be readable from a completely different client system in one, ten, or even a hundred years. Retrieving data from an archival store can be difficult if the encoding of stored data requires complex interpretation, or if instructions are not self-evident or explicitly defined. Our solution aims to outlive changes in computing and storage technology by combining a simple file storage and identification interface with an efficient storage mechanism that separates the mechanisms to detect and eliminate redundancy from the data format and specifications to retrieve and reconstruct files from their component parts. Although it is not the intent of our project to solve the problem of interpreting application content, our solution aims to ensure that the data stored internally is easily interpreted and can migrate to future systems.

The VCAS storage interface is intentionally designed not to meet certain nongoals. Application- or user-defined interfile relationships are not specified explicitly by any operation; it is the responsibility of the storing application, which includes the filing programs that are the analog of file systems, which provide directories and linkage, to store the relationships. Internally, the VCAS creates interfile relationships in order to reduce storage of redundant data, but these relationships are not made visible to the client of the interface.

Locality of reference is not guaranteed. Content addresses are hash values with the goal of minimizing collisions, so the address by itself indicates no relationship to other stored data. On the other hand, the VCAS interface does not preclude the implementation from inferring locality of reference through caching or temporal relationships, for example, deriving a closeness from the time files stored into the archive.

Operations might be idempotent. Multiple instances of similar or identical files might not be stored multiple times. This is important because applications that create duplicates or backup copies do not assume that storing more copies improves reliability.

3.1.3 PRESIDIO and the Deep Store Architecture. The Deep Store architecture consists of the following primary abstractions: storage objects, physical storage components, a software architecture, and a storage interface. PRESIDIO exists as a space-efficient content-addressable store within the larger system.

The primary storage objects presented to the archival store are the *file* and its *metadata*. File contents are stored as content-addressable objects, devoid of any metadata. Simple metadata associated with the file, such a file's name, length, and content address, are contained within a metadata structure. The metadata can also be identified by content address. Section 6.3 discusses the use of file metadata in detail.

3.1.4 PRESIDIO Hybrid Compression Framework. PRESIDIO is designed as a framework defining a storage model, efficient storage methods, and an algorithm to combine the two. The heart of this storage system is the CAS. Efficient storage methods are object-oriented classes that implement several operations on a single piece of data: feature selection, resemblance detection, and redundancy elimination. The storage methods also perform the two parts to write and then read the data. To record the data, a storage object is represented as a virtual object; the virtual object describes itself as a series of actions to encode itself using the CAS. To reconstruct the data, the self-describing virtual object describes the steps to reconstruction. The instructions are defined as concatenations of other stored CAS objects, delta-compressed CAS objects, stream-compressed data, or just raw bytes. The details of the VCAS encoding are described in Section 6.2.

To determine the most desirable efficient storage method, the PRE algorithm iteratively applies feature selection and resemblance detection to a candidate file to compute a ranking based on the estimated storage efficiency. Once all ranking has been completed, the most efficient method is used to record the object. Because the most efficient storage methods vary based on the content and estimated storage efficiency, more than one method can be applied to a file, thus creating a hybrid compression storage mechanism within the PRESIDIO framework.

4. PRIMITIVES FOR IDENTIFYING SIMILAR DATA

We use the concepts of *features*, *data fingerprints*, and *feature selection* to identify similar data. Features are properties that identify and distinguish data. The main features we use in feature selection are digests, chunk digests, chunk lists, sketches, and superfingerprints. Digests are hashes computed over arbitrary binary strings such as whole files. Chunk digests are the same, but computed over chunks, or subsections, of files. Chunk lists are concatenated identifiers to chunk data; the identifiers are usually chunk digests. Sketches are summary collections of identifying data, like data fingerprints, selected deterministically from a larger set of fingerprints; in addition to identifying data, they can be used to compute a similarity metric. Sketches that are serialized into a stream of data can also be used as input to a fingerprinting function to compute superfingerprints.

The foundations of feature analysis are made up of simple bit-string hashing and data fingerprinting algorithms. Variable-length subsections of files can also be used as content from which to compute features. Shingling methods compute fingerprints from overlapping (sliding) windows over content.

We differentiate between *feature data*, the original binary data, and *(computed) features*, the features computed from the feature data. We use the most general content-independent algorithms on arbitrary binary data whenever possible so as to apply features to a wide range range of problems. When the format and contents of data are known explicitly, applications can extract features that yield higher quality metrics become possible. For example, information retrieval and web search systems use human-readable words extracted from content to determine features made up of words, word bases, sentences, word sequences, as well as numerical or hash representations for those words. Metadata, both manually or automatically generated, are also used as features, including tagged data, filenames, keywords, and timestamps.

If such a wide range of metadata and feature types are available, why do we not use them? Semantic feature computation and selection algorithms that require knowledge of data encoding have several shortcomings for the purpose of storing data efficiently. First, the content types must be well-specified. Data encoding standards exist, but they change or update relatively quickly, making it difficult to predict what features used today will be used in the future [Rothenberg 1995]. Second, interpreting content implies ensuring algorithmic permanence in addition to data permanence—in other words, semantic interpretation of the data must be possible well into the future when data is read, and not at the present when it is written. This is a difficult problem that is well outside the domain of our problem [Lorie 2001, 2004]. Third, searching over the stored data is a different problem than identifying data for data compression. Current indexing and retrieval techniques differ based on content type. For example, text searches that use inverted file indexing [Witten et al. 1999] or PageRank functions [Brin and Page 1998a, 1998b] require low latency not high throughput or space efficiency and serve the purpose of searching for arbitrary user queries.

The goals for storing data efficiently and searching semantic content are different, but not mutually exclusive. Instead of incorporating them into a single system, we defer the design of content-sensitive selection and search mechanisms to higher levels of the system and focus our solution on the problem of storing data efficiently with content-independent feature selection algorithms.

In order to minimize dependencies across a very large system, our design uses strictly deterministic feature computation and selection, that is, computing and selecting features is purely algorithmic and not dependent on data already stored in the system. Independence across a distributed system allows it to operate without interaction, thereby improving opportunities for scalability. Feature selection that is independent of existing data may offer performance benefits within a single storage machine by avoiding extra I/O due to data retrieval before data storage.

4.1 Hashing and Fingerprinting

A hash function produces *hashes* or *hash identifiers*, much smaller bit string representations than their input of arbitrary length s. We use hashes of fixed length r, whose value is typically tens of bytes. Thus, the hash function maps the space of permutations 2^s into 2^r possible values. Because $r \ll s$, not all possible bit strings can be represented unambiguously. But when r is 128 or larger, the range of 2^r hash values is large enough to uniquely identify data with high probability. The properties of hashing functions and their properties are a well-understood area.

Table I. Approximate Collision Probabilities, p, for Specific VCAS Design Values

	f	l_{file}	l_{vcas}	c	m	n	q	r	p	description
1	2^{30}	NA	NA	1	2^{30}	2^{128}	30	128	2^{-69}	MD5, 10^9 files, whole file objs
2	2^{40}	NA	NA	1	2^{40}	2^{128}	30	128	2^{-49}	MD5, 10^{12} files, whole file objs
3	2^{30}	2^{14}	2^{8}	2^{6}	2^{36}	2^{128}	36	128	2^{-57}	MD5, 10^9 files, 256 byte objs
4	2^{40}	2^{20}	2^{7}	2^{13}	2^{53}	2^{128}	53	128	2^{-23}	MD5, 10^{12} files, 128 byte objs
5	2^{40}	2^{20}	2^{7}	2^{13}	2^{53}	2^{160}	53	160	2^{-55}	SHA-1, 10^{12} files, 128 byte objs
6	2^{40}	2^{20}	2^{7}	2^{13}	2^{53}	2^{256}	53	256	2^{-151}	SHA-256, 10^{12} files, 128B objs

Broder narrows the definition of fingerprinting [Broder 1993] from the more general universal hashing. In (universal) hashing, m, the number of distinct objects, is a fraction of the total number of possible fingerprints 2^r, whereas in fingerprinting, $m \ll 2^r$.

4.1.1 Collision Properties. Although probabilistically infrequent, hash collisions can occur. We rely on low collision rates for identifying data; when evaluating content-addressable storage for identifying blocks with probabilistic uniqueness, it is important to consider how it affects the design of the storage system. Let f be the number of files, l_{file} the average file length, in bytes, l_{vcas} the average size of VCAS object (e.g., chunk), $c = l_{file}/l_{vcas}$ (the number of VCAS objects per file), $m = c \times f$ (the number of objects to be stored in the global VCAS), r (number of bits in the content address), $n = 2^r$ (the number of addressable objects in the VCAS), $q = log_2(m)$. Note that r is commonly 128–160 bits in CAS systems.

Using an analysis similar to that in the *birthday paradox*, the probability of no collisions occurring is given by $e^{-(m(m-1))/2n} = e^{-2^q(2^q-1)/2(2^r)} \approx e^{-2^{2q-(r+1)}}$. Since for $1-e^{-x} \approx x$ for small x, then the chance of failure is $p \approx 2^{2q-(r+1)}$. The different storage scenarios listed in Table I illustrate the effect on collision probability when the number of stored objects, content address size, and other related factors are varied.

We start by designing a system to hold about 2^{30} or about 1.1 billion files. This is approximately equivalent to 2^{10} (or 1,024) storage nodes, each with about 2^{20} (about 1.07 million) files each. We assume the average file size in a file system is less than 2^{14} bytes (16KB) [Tanenbaum et al. 2006] and set a lower practical limit for chunk sizes to $2^7 = 128$ bytes from previous work [You and Karamanolis 2004]. Table I lists parameters, and probabilities of collision. Lines 1 and 2, which can be used for whole-file CAS, provide very low collision probabilities. Lines 4 and 5, which represent MD5 and SHA-1 hashing of chunks, respectively, represent quite different probabilities at a slight 4 bytes per object for SHA-1, at 2^{-23}. SHA-1 will give better range, for example if design assumptions of file counts change, as well as lower probability of collision. The next size of commonly available cryptographic hash, SHA-256, is more costly at 32 bytes, which will have a detrimental effect on storage when the CA is stored as a reference within the encodings.

One criticism of CAS is its lack of strict correctness. Henson argues [Henson 2003] that CAS systems that rely on hash equivalence for object equivalence is fundamentally the same as an error in design or implementation of hash tables in which a hash function is used to compute *hash keys*, but not comparing *values* when there is a collision in the keys.

One last question remains: What probability of collision is satisfactory? A common design philosophy is to match or exceed the undetected disk bit error rate on magnetic storage (in spite of ECC) which is typically 10^{-12} to 10^{-15} [Hitachi Global Storage

Technologies 2004; IBM 1999; Riggle and McCarthy 1998]. One might argue that this is a reasonable error rate, however there is a difference: in magnetic disk devices, undetectable errors occur at the block level. CAS storage systems like ours typically hash entire files and then identify the files by their CA. It would be more appropriate to evaluate subfile hashing, for instance by first subdividing the blocks. Comparing block-level hashing collision rates with undetectable block error rates is perhaps a more appropriate metric. In essence, we argue that hash collision rates should be lower than the error rate of comparing two stored files, which is the acceptable method for determining if a file was recorded to disk or copied correctly. In turn, such guarantees rely on the stated block error rates.

4.2 Hashing Techniques Used by PRESIDIO

PRESIDIO uses two main hash-based algorithms, *message digests* and *Rabin fingerprints* in two distinct ways: *strong hashing* to uniquely identify variable length data across the storage system, and *weak hashing* to identify smaller strings using smaller hash identifiers with lower probability of uniqueness [Tridgell 1999].

Strong hashing is used to identify large blocks of data, including whole files, to produce an identifier that uniquely identifies that object by its content. We define the computed hash of the content as its *content address*. Functions that accept variable-length strings are often called *message digest*. We use cryptograph hash functions like MD5 [Rivest 1992] and SHA-1 [National Institute of Standards and Technology 2008] for their wide availability to improve long-term tranparency (interpretability) into the future, as well as for their cryptographic properties.

For many reasons, weak hashing in the form of fingerprinting is used to identify smaller regions. These hashing regions are smaller than stored objects, and identify data with the assumption that small contiguous regions are more likely to also be present in similar files than not. We will describe how fingerprints over smaller regions are used to identify similar data. Our solutions use weak hashing for shingling (fingerprints over small, overlapping regions in a file), chunking, feature selection and superfingerprints (a fingerprint over multiple fingerprints). We have found Rabin fingerprinting [Broder 1993; Rabin 1981] to be very useful for two reasons: it can be implemented to efficiently compute fingerprints over sliding windows, and can be easily configured to provide a large number of *randomized functions* which provide deterministic results, but may be parameterized to provide unique functions from different irreducible polynomials. We have previously provided details of additional properties and implementations [You 2006]. We have found that 32-bit Rabin fingerprints provide good uniqueness properties, a small size that when combined into a collection of other fingerprints, meets the goal of modest file overhead.

4.3 Feature Selection

Feature selection defines the feature strings to be fingerprinted, and the *feature selection algorithm* for retaining fingerprints. In some cases, the feature set may be small; for instance, a file digest would compute one feature over a feature string that is the whole file. In other cases, the number of features that are computed is very large, as is the case in shingling. Retaining all computed features is not always practical, and as described below, is not necessary for the purpose of approximate similarity metrics; therefore the selection algorithm plays an important role in determining feature set quality.

A feature set may contain multiple instances of features. An instance when this would occur is when randomized functions select two features representing the same feature data twice. We sometimes speak of features as both the feature string, as

well as the feature fingerprint. The literature mixes these terms also, for instance "shingles" sometimes refers to the literal substring and at other times the fingerprint of the shingle.

Feature sets are stored as a representation of stored objects for several reasons. The first is that we assume storage of immutable data, so once the features are computed, there is no need to update them. The second is that computing and selecting features can be I/O- and CPU-intensive. The third is that fingerprints are much smaller than the data they represent, so the additional storage overhead is small.

4.3.1 Whole-File Hashing (Message Digests). Within archival storage, the main goal of whole-file hashing is for storage clients and servers to produce a content address, a unique machine-readable name for a file. Using whole-file hashing conveniently solves several problems: immutable file data is identifiable by content address; content addresses are very small (for instance 16–20 bytes) to reduce identification metadata; and hashing can be used to locate a file object in a flat namespace instead of traversing directories or other search domains, suppressing storage, and minimizing or eliminating data transfer by using content addresses instead of file contents.

Whole-file hashing still exhibits several problems. The first is that file data must be immutable, otherwise the hash is invalidated. Second, hash collisions are possible, making it possible for a file not to be stored because the write operation was suppressed on account of the apparent existence of the file's hash value for a different file [Henson 2003]. Third, hashing requires computation that may affect performance. Furthermore, even when used for improving tamper-resistance, cryptographic hash functions that are designed to provide preimage resistance—a property to make inversion of the hash difficult—may not stand the test of time and attacks. Archival storage systems that assume today's hash functions provide integrity guarantees may eventually become vulnerable.

4.3.2 Chunking. The fine-grained single-instance chunk storage improves on the efficiency of whole-file CAS when files are similar but not identical. Chunking metadata (chunk lists) increases storage overhead with no net benefit over whole-file CAS when unique files are stored. Our prototype computes block hashes using Rabin fingerprints of sufficient degree to satisfy chunk uniqueness within the corpora that were evaluated. To evaluate the storage efficiency using a larger chunk identifier size, we only need to account for the instances of identifiers that were stored and reevaluate the storage used by them.

Fixed-length block storage is a degenerate case of chunk-based storage. The drawback to this method is that data inserted into or deleted from a block (other than the block size itself) will prevent any data following the modification to be identified due to the misalignment of feature data.

4.3.3 Shingling. To compute a complete feature set for a file (a sketch), we compute fingerprints over shingles. Shingles are overlapping substrings, W_i within a file of length s with a fixed length w. Unlike shingles over web corpora that use words as tokens, we shingle binary (as well as text data), resorting to single bytes as the smallest unit of a substring, since binary files do not even necessarily exhibit word (e.g., 2, 4, or 8 byte) alignment. The file length is s bytes in length, resulting in $s - w + 1$ shingles, $0 \leq i < s - w + 1$. The window (shingle) size is fixed, for example 30 bytes. (We used $w = 30$ for some experiments and later determined that, for a text-based corpus, $w = 64$ yielded very slight efficiency [You 2006].) Rabin fingerprinting by random polynomials [Rabin 1981] is commonly used for shingling [Broder et al. 1997; Fetterly et al. 2003], using a fingerprint size of, say, 32 bits to avoid collisions with around a million objects.

Computing, collecting, and creating a subset from all features would use a large amount of temporary space, so we use min-wise independent permutations [Broder et al. 1998; Broder et al. 2000; Fetterly et al. 2003] to incrementally select features, as we fingerprint shingles. This involves the use of a fixed set of k unique and randomly selected irreducible polynomials, each of which is used to fingerprint the shingle fingerprint. After each shingle fingerprint $f_{shingle}(W_i)$ is computed, k Rabin fingerprints are computed, $g_j(f_{shingle}(W_i)), 0 \leq j < k$ using preselected functions g_j. For each j, we retain the minimum $g_j(f_{shingle}(W_i))$ and its corresponding shingle fingerprint, $f_{shingle}(W_i)$. Upon completion, the minima are discarded. The resulting feature set is called a *sketch*. Resemblance [Broder 1998], or the *Jaccard coefficient*, is computed as an overlap of documents by comparing shingles, or more precisely, resemblance r between two files A and B is defined as

$$r(A, B) = \frac{|S(A) \cap S(B)|}{|S(A) \cup S(B)|},$$

where $S(A)$ is a set of features. When two sketches are identical, $S(A) = S(B)$, then $r = 1$, and when no features in sketches $S(A)$ and $S(B)$ match, then $r = 0$. We use ordered feature sets, or *feature vectors* so the resemblance between files A an B is discrete:

$$r(A, B) = \frac{|\{0 \leq i < k : f_i(A) = f_i(B)\}|}{k},$$

where $f_i(F)$ is a feature f of index i extracted from file F, and k is the size of the ordered sketch (array). In other words, each matching pair of elements in the feature vectors are compared and the total matching features are taken as a fraction of the total number of features. The resemblance is in the range of $0 \leq r(A, B) \leq 1$.

One refinement to further summarize the sketch is to compute superfingerprints, or fingerprints of features. Superfingerprints are fingerprints of a fixed number of features [Broder et al. 1997]. We define the *harmonic*, s, as the index, or level, used to partition the sketch, and f_{super_s} as the superfingerpint function for that harmonic substring. Then when $s = 1$, the first harmonic, $S_{0..l-1}$ is the concatenation of $f_{shingle}(W_i), 0 \leq i < l$ to form the harmonic substring itself. Then $f_{super_1}(S_{0..l-1})$ is the first superfingerprint over the first (or fundamental) harmonic, in other words, a fingerprint over a string consisting of all fingerprints.[1] Next, in the second harmonic $s = 2$, there are two substrings, $S_{0..l/2-1}$, $S_{l/2..l-1}$, each a contatenation of half of the fingerprints. The two corresponding harmonic superfingerprints are $f_{super_2}(S_{0..l/2-1})$ and $f_{super_2}(S_{l/2..l-1})$. Figure 3 shows how superfingerprint $S_{0..3}$ is simply a fingerprint with input string of concatenated fingerprints $f_{shingle}(W_i), 0 \leq i < 4$. This reduces the sketch size by a factor of l. Because the features are independent, a single superfingerprint that matches a corresponding superfingerprint in another sketch reduces the probability of locating a sketch with low resemblance, while still being able to detect sketches with high resemblance, [Fetterly et al. 2003]. By permuting any k features and computing a superfingerprint on them, *megashingles* or *super-superfingerprints* have also been defined.

4.3.4 Harmonic Superfingerprints. We further refine superfingerprints to progressively detect similar files with lower resemblance using *harmonic superfingerprints*. For identical, and highly similar documents, a small number of superfingerprints, s, is desirable because it reduces the search space. Earlier work has matched one, two, or

[1]We may visualize harmonic binary substrings as the physical string segments between nodes formed on a musical stringed instrument playing harmonics.

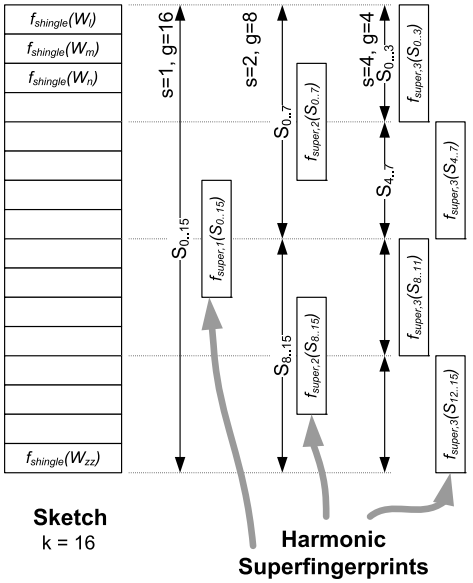


Fig. 3. Sketch, superfingerprints (e.g., $S_{0..3}$) and harmonic superfingerprints.

more superfingerprints to *increase* the specificity of detection, but some data sets can be compressed significantly using small numbers of superfingerprints covering an entire sketch. The result is that we use the superfingerprint covering the entire sketch in lieu of a whole-file digest to perform a hash lookup, and other superfingerprints to search in low-dimensional spaces.

We precompute harmonic superfingerprints at the time we compute a sketch, then progressively search the superfingerprint space to find sketches of decreasing resemblance. Let $s = 1$ and $g = k$, in other words, the superfingerprint is computed over all features in the sketch. Next, compute $s = 2$, $g = k/2$, so that each superfingerprint is computed over half the sketch and continues in this manner, doubling s each time. (For convenience we select $s = 2$, but it can be nonintegral and scale arbitrarily.) Figure 3 illustrates the sketch ($k = 16$), ranges of the superfingerprints and the resulting superfingerprint set.

We can compute the probability that a file matches another file with resemblance at least p. First we divide the sketch of k features into s groups of g. The probability of an individual feature matching the corresponding feature in another sketch is p, the probability of a superfingerprint over g (independently selected) features is p^g. We have s groups, so the probability of one or more groups matching is $1 - (1 - p^g)^s$.

Each curve in Figure 4 shows the probability that a pair of files with a given resemblance (horizontal axis), is detected by a *single* superfingerprint out of s. The sketch size overhead is fixed at k features (in this example, $k = 32$), resulting in k/s features per group. If a single fingerprint is computed over all features in the sketch (rightmost curve), then 100% of the files with high resemblance are detected; it also filters out pairs with less than 90% resemblance, detecting less than 4% of them. (Note that the leftmost curve, $g = 1$, has $k = 32$ features, the same as a feature set. The difference is that a superfingerprint set will contain one fingerprint of a fingerprint. Comparing two sets of superfingerprints still requires $s = 32$ tests, requiring $s = 32$ comparisons between two sketches and a high probability of detecting low as well as highly similar files.)

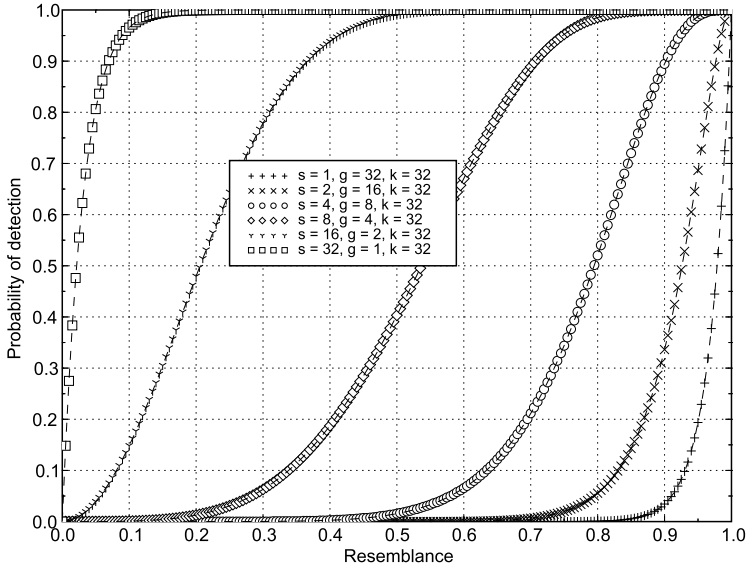


Fig. 4. Probability of detection with one matching superfingerprint, $k = 32$.

For comparison, we evaluate the probabilities that a single superfingerprint will match files above a resemblance for a larger sketch size. When the sketch size is increased to 128 features ($k = 128$), the probability of detection takes on a different shape for the group sizes. With large group sizes ($g = 128$), the probability of finding highly similar files is very high, and it also restricts matching files below 95% resemblance. The computational cost to compute a superfingerprint is linear, $O(k)$, in the size of the sketch. Compare this with the complexity to compute a sketch using min-wise independent permutation feature selection $O(sk)$, where s is approximately the size of the file. Thus for nontrivial files, the computational cost for superfingerprints is negligible.

Searching is straightforward: we first search with the $s = 1$ superfingerprint. Although this is similar to whole-file hashing, it is tolerant of slight variations in the file. Conversely, identical files will produce identical superfingerprints. Next, we search the superfingerprints in the dimensional space $s = 2$, and so on, until we find a sketch. Larger than these trivial cases, the problem returns to a problem similar to an n-dimensional keyword retrieval. Traditional methods using an inverted keyword index [Witten et al. 1999] requires large amounts of memory.

We use harmonic superfingerprints to progressively search the space by finding highly similar files quickly and moderately similar files with more time. The time to find highly similar files in our example is $1/k$, which is significant. If the first harmonic superfingerprint fails, i iterations with $s = 2^{i-1}$ results in $2^i - 1$ superfingerprint lookups. Increasing i by more than one can reduce the total number of tests: for instance, if $s = [1, 4]$, in the best case we speed up by 32 and in the worst case by 5. At higher harmonics, the probability of detection increases, but with diminishing returns. Further experimental analysis of real workload data may provide better insight to the benefit of more exhaustive search, while minding the "curse of dimensionality."

If no sketch is found, we can fall back to slower methods, such as comparing sketches. Alternative methods for detecting near-duplicates use Charikar's hash [Charikar 2002], also called a *simhash* [Manku et al. 2007]. Manku et al. and Henzinger [Henzinger 2006] provide experimental results showing the higher

Table II. Comparing Efficient Storage Methods (ESMs) and Their Similarity Detection Characteristics

	ESM	feature	coverage	fp	size	k	selection
1	Whole File	file	whole file	MD5	16	1	hash file
2	Chunk	chunk	128–8192 var.	RF32	4	m	hash chunk
3	Chunk	chunk	128–8192 var.	RF64	8	m	hash chunk
4	Chunk	chunk	128–8192 var.	MD5	16	m	hash chunk
5	Chunk	chunk	128–8192 var.	SHA-1	20	m	hash chunk
6	Chunk	chunk	4096 fix.	SHA-1	20	K	hash block
7	DCSF-SFP	s.w.	8–32 fix.	RF32	4	$\{2^n,2^m\}$	H-SFP
8	DCSF-SFP	s.w.	8–32 fix.	RF32	4	{84,6}	SFP
9	DCSF-FP	s.w.	8–32 fix.	RF32	4	K	MWP
10	DCSF-FP	s.w.	8–32 ch.	RF32	4	K	MWP

Note: Here fp is fingerprint, s.w. is Rabin fingerprint (RF) sliding window, k the feature set size; in superfingerprint, $\{k, l\}$ represents the cardinality of the initial sketch size k, followed by the cardinality of supersketch size l; RF32/64 is 32/64-bit Rabin fingerprint, H-SFP the harmonic superfingerprint, SFP the superfingerprint, and MWP the min-wise independent permutations.

precision of simhash on a corpus over Broder's method by using iterative comparison methods. Such methods may be a suitable way to further detect similarity at additional computational and storage cost due to the somewhat different construction of *simhash* from shingles. With *simhash*, dissimilarity increases as the bit edit distance between two *simhash*ed objects increases, and with it an increased search space.

4.4 Finding Similar Data Over a Large Corpus

In contrast to stream compression algorithms, commonly used to compress individual files, PRESIDIO operates over a large corpus of data. In Table II, we list some of the similarity detection characteristics for each efficient storage method in approximate order of coverage size, starting with whole file hashing, and ending with delta compression between similar files with fingerprinting. Detecting identical data using probabilistically unique hash functions is straightforward using whole file hashing, chunking or blocking (as a degenerate case of chunking). We now discuss methods to detect similarity among non-identical data.

Files that are added to a system using the store operation can be modifications of an existing file, such as the extension of a system log, or they might be a modification from a common file, such as a document template. In an ever-expanding storage system, files will be added over time. Unlike systems that perform static analysis of the file set to determine similarity across all files [Manber 1993; Douglis and Iyengar 2003], DCSF selects a reference file *incrementally*. DCSF selects the best candidate reference file by using the following criteria, in order: highest resemblance, shortest delta chain, and highest degree of dependence. The delta-dependency graph is a directed acyclic graph, so no cycles may occur. The *delta chain length* is the maximum of all possible paths from version to reference, with shorter ones being beneficial. The DCSF-SFP method (Table II, lines 7,8) depends on the generation of sketch sizes of cardinality k. Features are Rabin fingerprints, computed over data in sliding windows. The sliding window itself is chosen once and fixed for all time, from empirical data. In line 8, by selecting initial sketch size $k = 2^n$, and superfingerprint sketch size $l = 2^m$, $m < n$, the search for matching sketches takes place in a smaller dimensional space. The degenerate case, $m = 0$, or superfingerprint sketch $k = 1$, produces a single superfingerprint that can detect highly similar files with high probability. The parameters in Table II, line 8,

{84,6}, (sketch size and number of superfingerprints, respectively) were used to detect resemblance between web pages [Broder et al. 1997; Manasse 2003] is provided for comparison, but due to the large number of fingerprints (84) and subsequent storage overhead in the initial sketch, we do not consider these parameters for our use because space efficiency is of primary concern.

The DCSF-FP methods (Table II, lines 9,10) are prerequisite computations to DCSF-SFP (7,8), but in practice we compute both at the same time. In the case of DCSF-FP, searching for fingerprints is an indexing problem. Searching multidimensional spaces may take considerable time or space.

The superfingerprint compression rate for a single group of size $s = k$ is the same as the compression for files compressed for resemblance $r = 1.0$ (excepting hash collisions). A simple example using Linux source code versions 2.4.0–2.4.9 resulted in 84% of the files matching exactly by a single harmonic superfingerprint at $s = 1$.

5. EVALUATION OF DATA COMPRESSION

We report the first direct comparison of the two main compression techniques, namely chunking and delta compression, and compare them against intrafile compression. In general, both chunking and delta encoding outperform *gzip*, especially when they are combined with compression of individual chunks and deltas. Our evaluation of resemblance detection and redundancy elimination also shows the effectiveness of different methods across a number of data sets.

The functionality and performance of each approach depends on the settings of a number of parameters. As expected, experimental results indicate that no single parameter setting provides optimal results for all data sets. Thus, we first report on parameter-tuning for each approach and different data sets. Then, using optimal parameters for each data set, we compare the overall storage efficiency achieved by each approach. The required storage includes the overhead due to the metadata that needs to be stored. Last, we discuss the performance cost and the design issues of applying the two techniques to an archival storage system.

One problem that comes up in the configuration of the chunking algorithm is setting parameters that maximize storage efficiency (see Figure 2). The size of the hash identifiers computed from the identifying (strong hash) algorithm increases the per-chunk storage. To reconstruct original files from their constituent chunks, the system needs to maintain metadata that maps file identifiers to a list of chunk identifiers. Furthermore, the chunk size to be used is inversely proportional to the chunk list overhead.

An optimistic estimate of the storage overhead comprises a list of chunk identifiers for each file, and for each chunk, the chunk identifier and the chunk size. A production system might incur additional overhead from managing variable-sized blocks, in-memory or on-disk hash tables, as well as file metadata.

5.1 Tools, Datasets and Parameters

5.1.1 The chc *Program.* To measure chunking efficiency, we developed the *chc* program [You and Karamanolis 2004] that compresses an input file to produce a "chunk compressed" output file (Figure 5); an inverse operation reconstructs the original input file. This program emits statistical information (chunk sizes, sharing) with the output file containing all the overhead needed for evaluation of storage efficiency. The actual step of compressing the chunk file was performed by using *gzip* on the output file.

The data objects referenced were simply the stored chunks. Each data set was combined into a single *tar* file and then the file was divided into chunks. (The *tar* file was used as an approximation to chunking each file individually.) Single instances of chunks were stored using the following parameters: min chunk size 64, max chunk

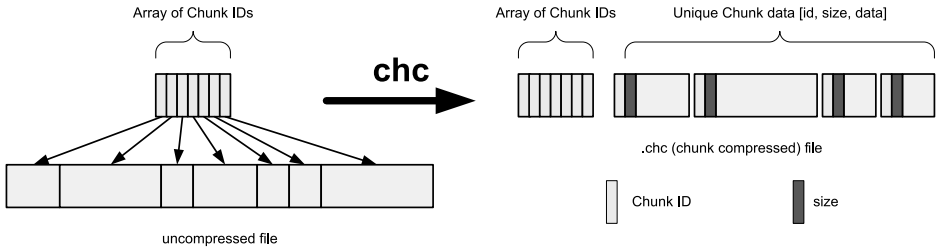


Fig. 5. The *chc* program file format.

size 16384 and window size 32. Every chunk identified in the *tar* file was sent to the CAS to be stored. The first time a chunk was stored, its reference count was set to 1; subsequent store requests were suppressed, but the reference count was incremented. Finally, the CAS tabulated the reference counts over all chunks.

5.1.2 Delta Encoding Similar Files. Several delta programs are available today, including *vcdiff* [Korn and Vo 2002] (an IETF standard [Korn et al. 2002] and successor to *vdiff*), *xdelta* [MacDonald 2000], and *zdelta* (whose report compares several programs for storage efficiency and performance [Trendafilov et al. 2002]). These programs may incorporate a stream-compression stage after computing the delta encoding such that a delta computed between an empty reference file and a version file to produce the output delta file would result in a delta-sized similar to one created by a compression program, such as *gzip*, over the version file alone. Delta encoding tools generally do not have any parameters, even though there can be algorithmic variations.

Detecting similarity requires a number of parameters, such as s, the feature set size (say, 30); w, the shingle window size (24 bytes), the degree of shingle fingerprint (32 bits), the degree of feature fingerprint, maximum length of a delta chain, the delta threshold, and the random irreducible polynomials selected for the window. Each sketch of s fingerprints $feature_i$ of degree l, for example, l = 32 bits, is stored, for a total of $4s$ bytes. We choose two parameters, the feature set size, s = 30, and the window size, w = 24, within ranges that have been shown to provide meaningful resemblance metrics—and more importantly—a strong (inverse) correlation with the size of the delta encoding [Douglis and Iyengar 2003].

In order to determine whether a new file should be added as a new reference file or a delta file as a version of an existing reference, a resemblance is computed using a pairwise comparison of features between two sketches. A delta encoding is computed if the number of matches is greater than a threshold parameter.

Once a reference file within the resemblance threshold is found, we used *xdelta* for calculating the actual delta and then compressed with *gzip* (*zlib*). A pointer to the reference file has to be maintained with every delta in the system. Such identifiers (e.g., SHA digests) as well as file sketches contribute to some storage overhead that has to be taken into account. Our prototype consists of three programs, one for each of the three problems above: feature extraction, resemblance detection, and delta generation.

5.1.3 Data Sets. To establish a baseline for each data set, we created a single *tar* file from the data set (see Table III), and then compressed it with an intrafile compression program, *gzip*. As expected (and as shown by the two first rows of the table), interfile compression improves with larger corpus sizes. This is not the case with *gzip*.

The HP Unix Logs (8,000 files) show very high similarity. Chunk-based compression on this similar data was reduced to 11% of the original data size, and when each chunk is compressed using the *zlib* (similar to *gzip*) compression, it is just 7.7% of the original

Table III. Comparing Storage Efficiencies of Different Compression Methods

Data Set	Size	# Files	tar & gzip	Chunk	Chunk & zlib	Delta	Delta & zlib
HP Unix Logs	824 MB	500	15%	13%	5.0%	3.0%	1.0%
HP Unix Logs	13,664 MB	8,000	14%	11%	7.7%	4.0%	0.94%
Linux 2.2 src (4 vers.)	255 MB	20,400	23%	57%	22%	44%	24%
Email (single user)	549 MB	544	52%	98%	62%	84%	50%
Mailing List (BLU)	45 MB	46	22%	98%	53%	67%	21%
HP ITRC Web Pages	71 MB	4,751	16%	86%	33%	50%	26%
PowerPoint	14 MB	19	67%	55%	46%	38%	31%
Digital raster graphics	430 MB	83	42%	102%	55%	99%	42%

size. Even more impressive are the reductions in size when using delta compression. When delta compression is used alone, the data set is reduced to 4% of the original size, but when combined with *zlib* compression, the compressed data is less than 1% of the original size.

Textual content, such as web pages, can be highly similar. However, in the case of the HP ITRC content, *gzip* compression is more efficient than chunking or delta. More surprisingly, *gzip* is better even when we do additional compression of chunks and deltas. The reason is that *gzip*'s dictionary is more efficient across entire files than within the smaller individual chunks, and chunk IDs appear as random (essentially noncompressible) data. But in the context of an archival storage system, *gzip*'s advantage is not likely to be as effective in practice; this is discussed below.

Nontextual data, such as the PowerPoint files with chunking and delta (especially with *gzip*) achieve better efficiency than *gzip* alone. However, the compression rates achieved are less impressive than those for the log data. For raster graphics, delta encoding with *gzip* achieves modest improvement over *gzip* alone. The single user's email directory and a mailing list archive show little improvement when using delta. Chunking is less effective than *gzip*, although we would expect it to reduce redundancy found across multiple users' data.

In most cases, interfile compression outperforms intrafile compression, especially when individual chunks and deltas are internally compressed. Chunking achieves impressive efficiency for large volumes of very similar data. On the other hand, delta encoding seems better for less similar data. We believe that this is due to the lower storage overhead required for delta metadata. Typical sketch sizes of 80 to 120 bytes (20 to 30 features $\times$ 4 bytes) for a file of any size are significantly smaller than the overhead of chunk-based storage, which is linear with the size of the file.

Although compressing a set of files into a single *gzip* file to establish a baseline measurement helps illustrate how much redundancy might exist within a data set, it is not likely that an archival storage system would reach those levels of efficiency, for several reasons. Most important is that files would be added to an archival system over time and that files would be retrieved individually. If a new file were added to the archival store, it would not be stored as efficiently unless the file could be incorporated into an existing compressed file collection, that is, the new file would need to be added to an existing *tar*/*gzip* file. Likewise, retrieving a file would require first extracting it from a compressed collection and this would require additional time and resources over a chunk or delta-based file retrieval method.

Our experiments measured the size of an entire corpus, in the form of a *tar* file after it has been compressed with *gzip*. Had we compressed each file with *gzip* first and then computed the aggregate size of all compressed files, the sizes for *gzip*-compressed files would have been much larger. For example, in the case of the HP ITRC web

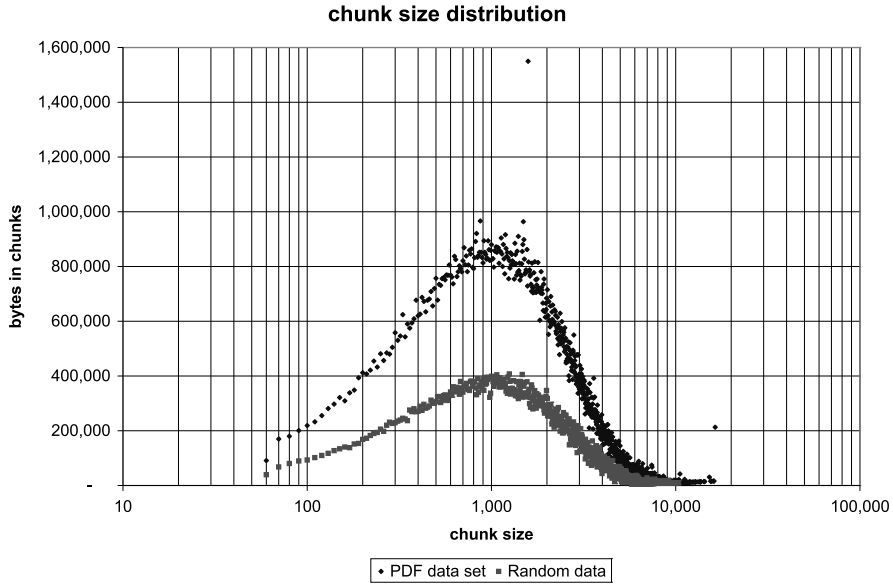


Fig. 6. Chunk size distribution; this depends heavily on d. Parameters $m = 64$, $M = 16{,}384$, $d = 1{,}024$ bytes. Data set is 754 files of total size 238.7 MB with mean 324.1 KB, median 124.4 KB, standard deviation 783.2 KB. The jump at around 1,580 bytes is an artifact of the PDF format. At 16,384, there is a jump due to the max chunk size restriction.

pages, *gzip* efficiency would have been 30% of the original size, much larger than the 16% shown in Table III, and larger than the 26% that can be achieved by using delta compression with *zlib*. When delta compression (or to a lesser extent, chunking) is applied across files first and then an intrafile compression method second, it is more effective than compressing large collections of data because additional redundancy can be eliminated.

Chunking Parameters. Since data that is chunked by one host does not need to see the data from another host in order for common chunks to be identified, we have the constraint that the parameters for chunking must be determined, once and for all, the data that is intended to be stored efficiently.

Typical distribution of chunk sizes varies whether the content is structured content, such as binary content in PDF files, or the data is purely random. We evaluated two different data sets: a single file, 100 MB (100×2^{20}) containing random bytes; and a *tar* file containing mostly unique PDF files. Figure 6 is a histogram showing the distribution of chunks weighted by their sizes and the total number of bytes stored for chunks of that size. The distribution is representative of a wide variety of file types.

Finding universally optimal chunking parameters may not be possible. In one experiment, using a set of highly similar data, "logs-10", a 10-file subset of the HP Log data, we varied the window size w (horizontal axis) and the *divisor* (individual plots) and plotted the results in Figure 7. It may suggest that larger window sizes are more space-efficient, but with small divisors such as 64 bytes, smaller windows are more efficient (23.1% at $w = 8$ compared to 23.5% at $w = 32$). And although smaller divisors appear to be more efficient in this data set, we see later that this may not hold true across data sets. The irregular curvature of the graph is due to the effect of chunking overhead.

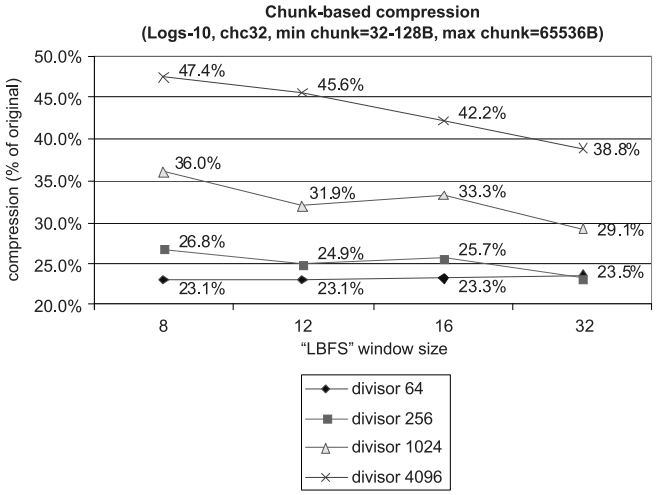


Fig. 7. The space efficiency of chunk-based compression at different window and expected chunk sizes.

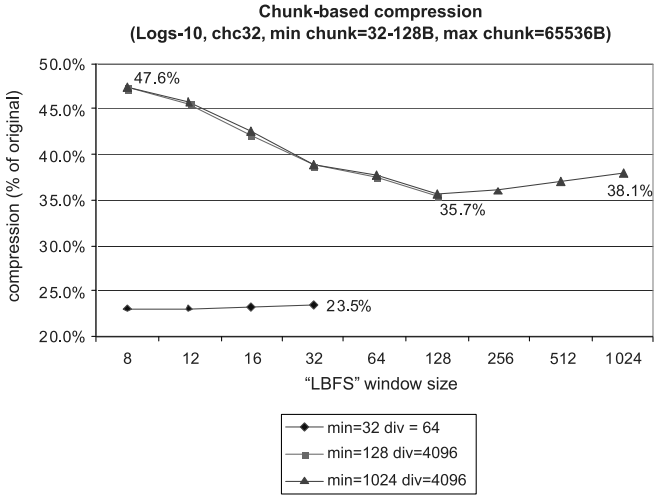


Fig. 8. The space efficiency of chunk-based compression at different window sizes.

The chunking parameters for the effect of the window size on chunking is shown in Figure 8. With larger chunk sizes and larger window sizes, w can improve storage efficiency with no additional cost. In this example of the text-based "log" data, a window size of 128 bytes provided the best efficiency (35.7%) for an expected chunk size of 4,096 bytes.

Chunking Overhead. In the case of chunking, the expected chunk size is a key configuration parameter. It is implicitly set by setting the fingerprint divisor as well as the minimum and maximum allowed chunk size. In general, the smaller it is, the higher the probability of detecting common chunks among files. For data with very high interfile similarity (such as log files), small chunk sizes result in greater storage efficiency. However, for most data this is not the case, because smaller chunks also mean higher metadata overhead. Often, due to this overhead the storage space

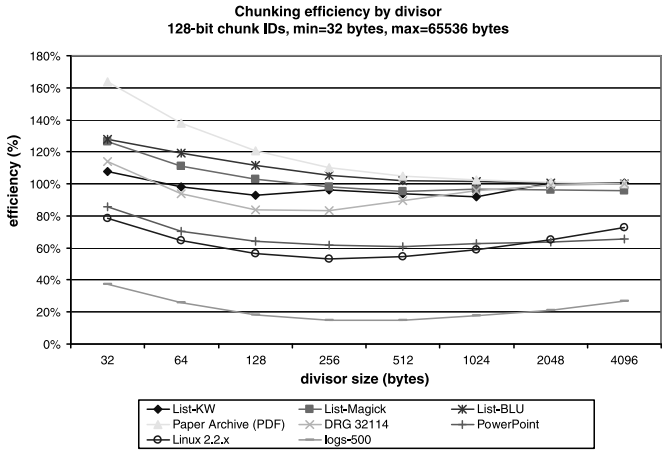


Fig. 9. Chunking efficiency by divisor size.

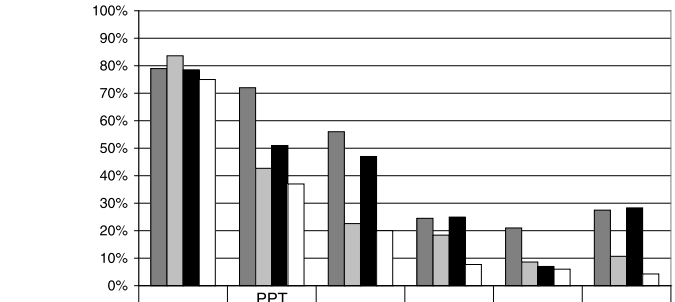
required may be greater than the size of the original corpus. Although Figure 8 suggests that smaller window sizes will improve compression with a limited data set, expanding to larger and more diverse data sets, shown in Figure 9, the optimal expected chunk size depends on the type of data; using 128-bit identifiers, the best efficiencies range from 256 to 512 bytes. The figure also shows that when similarity is low, compression efficiency is poor and the overhead can increase the storage requirements above the size of the original corpus.

Comparing Similarity and Chunking Overhead . For storage efficiency, delta encoded data works well at both ends of the spectrum: when data is highly similar, the total efficiency of delta-encoded storage is nearly the same as chunking when both are combined with *gzip* compression. But for data that is less similar, or even completely dissimilar, delta encoded data does not exhibit nearly as much overhead, since a sketch of 120 bytes for a file of any size is significantly smaller than the overhead of chunk-based storage which is linear with the size of the file. (Furthermore, with a small loss of efficiency, a sketch size of 80 bytes is nearly as effective [Douglis and Iyengar 2003].)

For example, chunks that have an expected size of 1024 bytes (chunking parameter d), a chunk ID size (CID_{size}) of 128 bits will incur approximately ($filesize/1024$) $\times$ 16 bytes, plus additional overhead for storing the chunks themselves of 20 bytes.

5.2 Evaluating Methods for Improving Compression

5.2.1 Delta Compression Between Similar Files. First, features are selected from files in a content-independent and efficient way using the DERD shingling technique (Section 2.1.4). 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 [Broder et al. 2000]) 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, for example, 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



	PDF papers	PPT presentations	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%

Fig. 10. Storage efficiency by method.

Table IV. Data Sets

Name	**size (MB)**	**# files**	**avg. size (B)**	**s.d.**
PDF tech papers	239	754	331,908	802,035
PPT presentations	63	91	722,261	944,486
Mailbox: 3 snapshots	837	383	2,291,208	9,397,150
HTML (f. zdelta benchmark)	545	40,000	14,276	27,317
PDF financial stmts	14	77	186,401	120,146
Linux 2.4.0v-9v	1,028	88,323	12,209	31,528

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). 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. 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 superfingerprints and supershingles [Broder et al. 1997]. The fourth step, common to all efficient storage methods, is storing the compressed data. In DCSF, the *delta file* is recorded to the CAS.

5.2.2 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 10 shows these measurements as a percentage of the original data.

To illustrate the range of redundancy in data, we selected data sets that are likely to be archived, that is, binary and textual data, small and large files, dissimilar as well as highly similar data (Table IV).

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

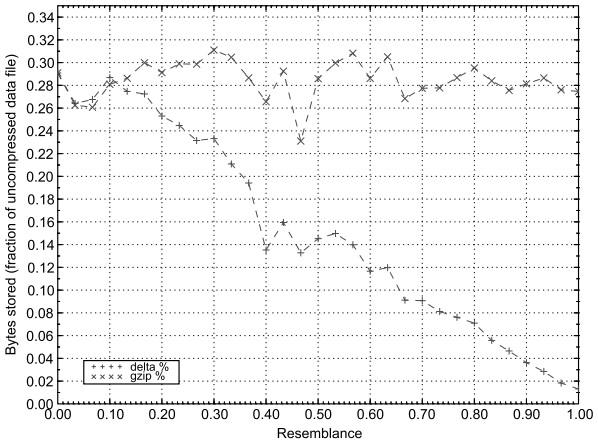


Fig. 11. Storage efficiency of *xdelta* vs. *gzip* for Linux kernel source code, versions 2.4.0-2.4.9, 88,322 files, total 1.00 GB, $w = 30$, $k = 32$, and fingerprint 32 bits long. The x-axis is a measure of the discrete resemblance (n/k) between a new file compared to a stored file with the highest resemblance: n features that matched out of a sketch of size $k = 30$.

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 nonmatching 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, that is, deltas are only computed against reference files to avoid chains of reconstruction, but at the expense of lower space efficiency.

5.2.3 Measuring the Benefit of High Resemblance 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, and so on, evaluating each file individually against previously stored files. The file size after *gzip* (intrafile compression only) was compared against *xdelta* (both intra- and interfile compression) and total storage was tabulated by resemblance (Figure 11).

We also ran the experiment with different sketch size, $k = 16$, and varied the window size, $w = \{8, 16, 32, 64, 128, 256\}$ (Figure 12). Higher commpression efficiency is exhibited at larger window sizes (but see Figure 15 also).

In our experiments we restricted version files to depend on a single reference file; hence the dependency graph is strictly a tree and the length is computed to the one reference file that does not depend on any other files. Another measure of data dependence is the *degree of dependence*, the total number of immediate version files that depend on the reference file itself; a higher number elevates the importance of certain files. Version files can also be used as reference files. The *delta* graph shows the storage efficiency improving 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 intrafile and interfile compression.

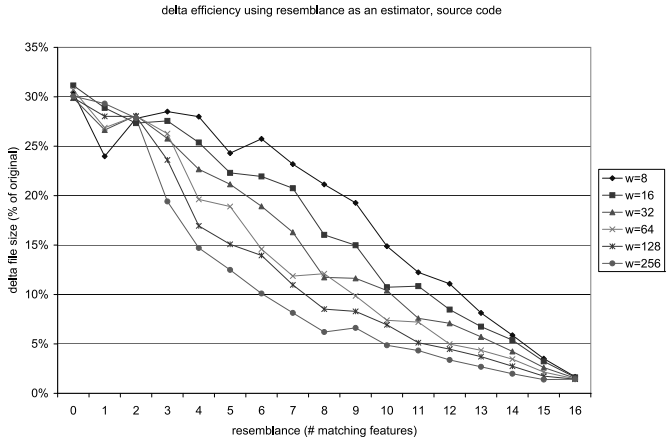


Fig. 12. Storage efficiency for different shingle sizes.

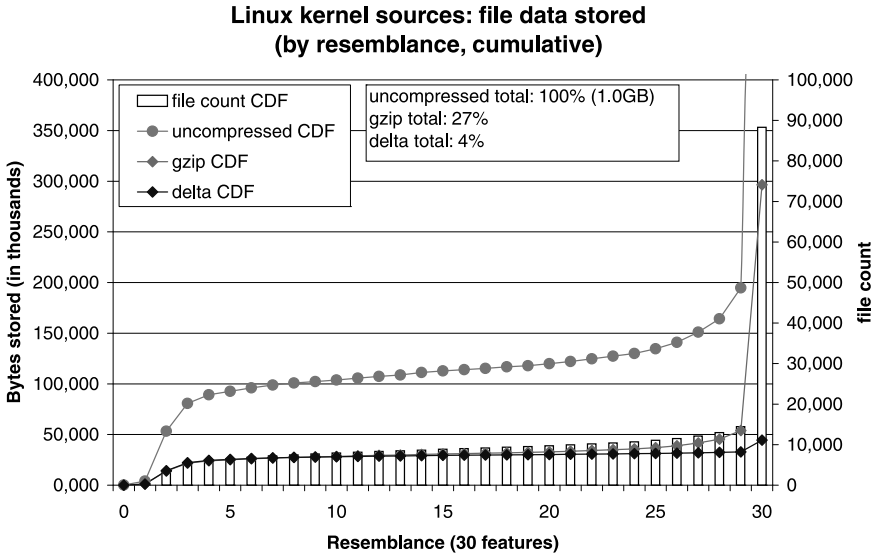


Fig. 13. Overlay of cumulative number of bytes (left vertical axis) and files (right vertical axis), for files with a given maximum resemblance to all previously stored files in the data set. Compares uncompressed data (1.00 GB) to *gzip* (0.296 GB) and delta compression/dcsf (0.044 GB).

Using the same data, Figure 13 shows a different view of the storage efficiency which 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, nearly 30 of 30 features,

Table V. Cumulative Data Stored by DCSF. One Version of Source vs. Ten Versions

method	bytes		size of 2.4.0	% of uncompressed	
	2.4.0	2.4.0–2.4.9	vs. 2.4.0–2.4.9	2.4.0	2.4.0–2.4.9
uncompressed	99,944,270	1,078,316,273	9.27%	100.00%	100.00%
gzip	27,527,744	296,475,233	9.29%	27.54%	27.49%
dcsf	26,005,285	44,446,197	58.51%	26.02%	4.12%

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 pairwise feature resemblance across a large set of files.

Another experiment compares DCSF on a single set of source code, Linux 2.4.0, against DCSF on ten sets of source code, versions 2.4.0 through 2.4.9. This experiment used a sketch size of $k = 32$ and produced nearly identical compression results to the previous experiment, $k = 30$ for ten versions. The results are shown in Table V. As would be expected of incremental source code development, the uncompressed data size for one version is 9.27% or about one-tenth of ten versions. Likewise, the total size of all files in a single version individually compressed by *gzip* is 9.29%, also close to one-tenth of all ten versions of source files gzipped individually.

This experiment reveals discriminating behavior when DCSF is applied to both dissimilar and highly similar data. While a single version of source files are individually compressed with *gzip* to 27.54% of their total uncompressed size, DCSF compresses slightly better, to 26.02% of total size. The small improvement is due to the slight favor of delta compression to *gzip* between files with nonzero resemblance. The more dramatic result is that storing ten versions of source requires 44.4 million bytes versus 26.0 million bytes—an increase of 70.91% to store an additional 979% more data.

Figure 14 illustrates the effect of resemblance on compression, comparing graphed points that compare a single set of source code (version 2.4.0) stored once, followed by ten versions of source code (2.4.0 through 2.4.9). Solid data points indicate a single version; outlined data points are for ten versions. For each of these, three sets of points are plotted for measurements taken over uncompressed data, *gzip* compressed data, and DCSF compressed data. As in Figure 13, the number of bytes is the cumulative total for all files for files with a given maximum resemlance to all previously stored files in the data set. The methodology follows: over three passes, files are added sequentially to the stored set; the maximum resemblance of each new file is computed to another file previously stored; in the first pass, files are added individually without compression; in the second, files are added but compressed with *gzip*; and in the third, files are added and delta compressed against a previously stored file arbitrarily selected from the set of files with the highest resemblance. The shape of uncompressed and *gzip*-compressed curves are similar, but reduced by a nearly constant factor for *gzip*. The difference between uncompressed (circles) and DCSF (squares) is very different for all ten (2.4.0–2.4.9) versions of source: most data in uncompressed form is highly similar, as indicated by the data point at 1,078,316,273 bytes, off the chart. However, for both single-version and ten-version curves, the additional storage required to store highly similar data is very small due to the high efficiency of delta compression.

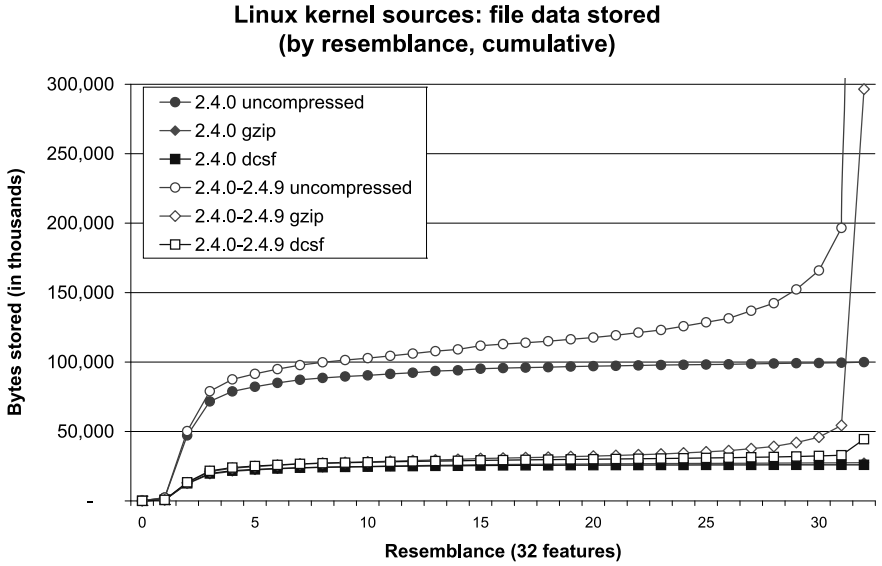


Fig. 14. Cumulative data, by resemblance; Linux 2.4.0 vs. 2.4.0–2.4.10; 1.00 GB cumulative uncompressed size.

By experimenting with DCSF and varied parameters, we have measured a number of data sets that have high variation in the amount of interfile and intrafile redundancy. High levels of compression are possible, including computer-generated data that was measured to less than 1% of the original size [You and Karamanolis 2004]. 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.

Size of Sliding Window. The most efficient shingling parameters are a tradeoff, and depend on the type of data. We experimented with window size, w, from 8 bytes to 256 bytes, and determined that the overall compressed size using DCSF varied slightly, but for each data set there was a minimum. Figure 15 displays the number of files matched by DCSF for each number of matching features up to 16 on 10 versions of the Linux kernel source code, versions 2.4.0 through 2.4.9, 88,322 files, 1.0 GB in size. The "bathtub curve" indicates that many files exhibited high resemblance and also that many other files had little, but nonzero, resemblance. Also, the total storage across various values of w is nearly the same (Table VI).

Other data sets also exhibit similar variations, with better compression ratios on windows of 16–128 bytes. As is the case with selecting chunk-size divisors, no single parameter works best across all data sources. More importantly, a single window size must be selected for all data in order to compute features deterministically; fortunately, for window sizes within this range, the data compression rates do not vary significantly.

5.2.4 Computational Performance. We measured the hashing performance of selected digest and fingerprinting functions. Table VII lists the functions, the size of the fingerprint, and the computational throughput; file sizes were 100 MB and 256 MB, precached in file cache; and time was measured with 10 microsecond resolution.

Whole-file hashing using MD5 and SHA-1 is much faster than disk bandwidth, suggesting that I/O bandwidth is the limiting factor. Rabin fingerprinting for a single

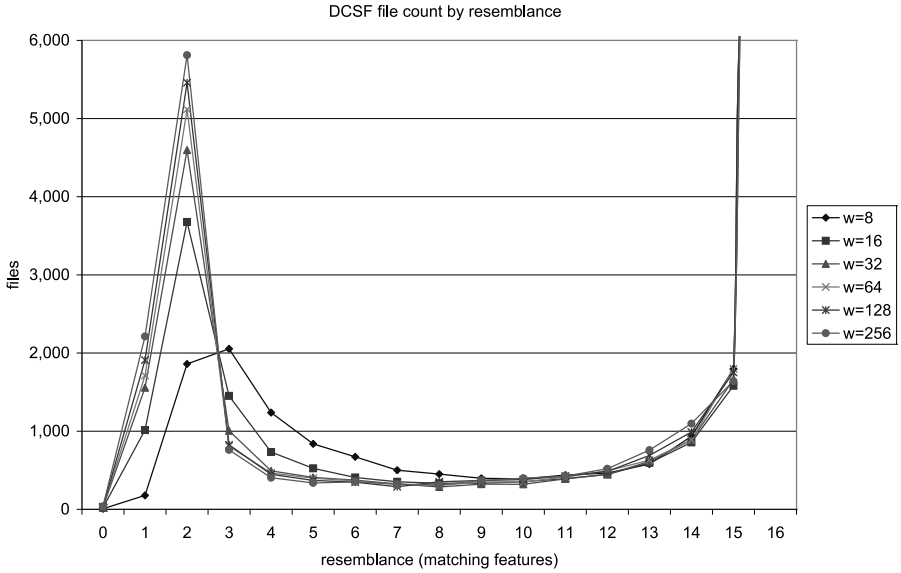


Fig. 15. DCSF filecount by resemblance and window size of highly similar Linux 2.4.0–2.4.9 kernel sources. There is one peak with two matching features and a much sharper one at 16 (out of scale).

Table VI. DCSF Compression Based on Window Size (1,078,315,981 bytes input)

window size (bytes)	$w = 8$	$w = 16$	$w = 32$	$w = 64$	$w = 128$	$w = 256$
bytes (1000s)	47,729	47,298	47,015	46,864	47,016	47,575
% over best	1.845%	0.925%	0.321%	0.000%	0.324%	1.517%
compression efficiency	4.426%	4.386%	4.360%	4.346%	4.360%	4.412%

fingerprint over the entire data file is significantly faster, due to the use of a precomputed table lookup. The chc32 program computed both strong and weak hashes (one for the chunk fingerprint and one for the sliding window to determine the division point) and also stored chunk data for its output, written to the null device. The shingle32 program computed a feature vector of 20 bytes.

Programs computing shingle feature sets using the min-wise permutation algorithms are computationally expensive. Our initial implementation reached 0.5 MB/second, but we were able to improve the throughput dramatically by more than 42 times through careful programming. Such improvements are necessary if resemblance detection is to be practical in an environment that evaluates different compression algorithms.

Delta Chain Length. The cost of retrieving a file by reconstructing data from reference and delta files is directly related to the length of the delta chains. We have measured the effect of short and long chain lengths. Delta chains of length greater than one benefit compression significantly. Previous work using delta compression to store data efficiently did not use delta chains [Douglis and Iyengar 2003; Kulkarni et al. 2004], and only used a single delta file to compress against a reference file and not another version file, that is, chain length $L \leq 1$. The measurements listed in Figure 10 show the potential difference between bounded and unbounded delta chains. When the maximum chain length is unbounded, storage can be reduced by nearly seven times, down to a storage rate of 4.3% (in the case of Linux 2.4.0–2.4.9).

Table VII. Feature Selection Performance

	hash size	MB/s	Notes
MD5	128b	174.8	GNU *md5sum* (coreutils) 4.5.3
SHA-1	160b	79.5	GNU *sha1sum* (coreutils) 4.5.3
Rabin fingerprint (by bytes)	32b	272.5	
Rabin fingerprint (by 32b words)	32b	1,528.0	
Rabin fingerprint (by bytes)	64b	75.2	
Rabin fingerprint (by 64b words)	64b	1,514.6	
chc32	32b	36.4	Chunk Compression program, 32-bit
shingle32 (before optimization)	32b	0.5	Shingle and superfingerprinting 20 features
shingle32 (after optimization)	32b	19.7	Shingle and superfingerprinting 20 features
cat > /dev/null	N/A	45.0	

Note: Feature selection performance for selected digest and fingerprinting functions on a Pentium IV 2.66 GHz machine with Red Hat Linux 9.

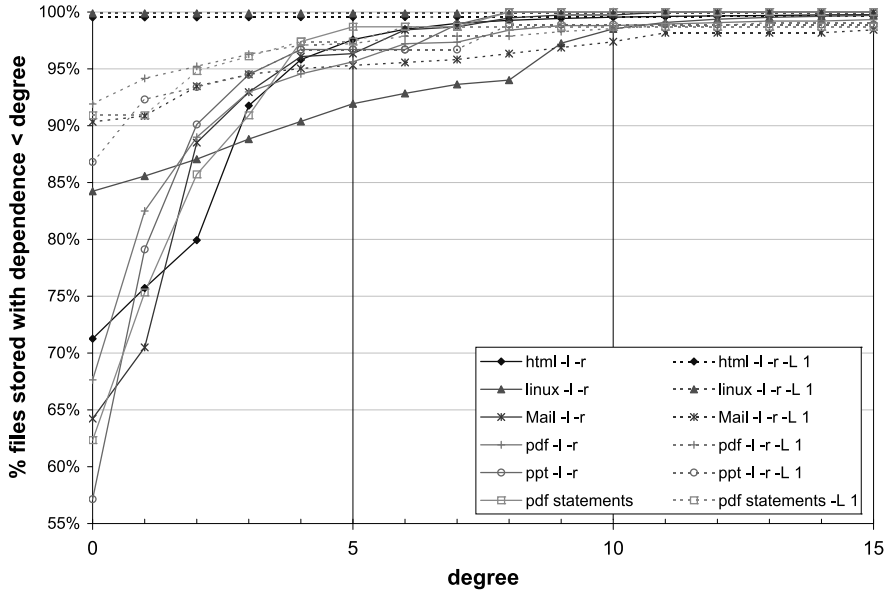


Fig. 16. Cumulative distribution of files stored by degree of dependence.

Figure 16 shows the cumulative distribution of files against the degree of dependence on base files. Each data set is graphed twice, once using delta chains of maximum length 1, and again using delta chains of unbounded length. Limiting the delta chain length also limits the degree of dependence. Higher numbers of files with lower degrees of dependence indicate lower instances of delta compression, which is in turn reflected by lower storage efficiency. In contrast, unbounded delta chains allow higher levels of data dependence. One reason for this effect is that files are introduced into the system one at a time.

The Linux data set contains ten similar versions of source code. The lowered data points below 95% for degrees less than 9 indicate the high level of delta encoding against similar files, including high degrees of dependence as high as 8. We can picture a single file (version 2.4.0) being introduced to the system, followed 9 identical or

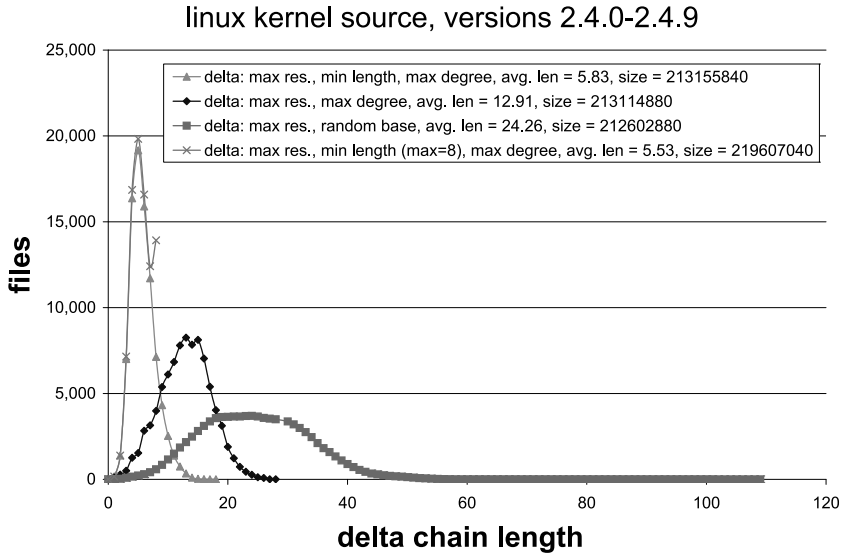


Fig. 17. Delta chain length and storage efficiency.

similar files being detected (2.4.1 through 2.4.9). Delta files are computed against the original.

But this restriction is a simplifying assumption, and relaxing it to allow arbitrary length increases storage efficiently significantly. This is in part due to the problem that a file can only be a reference file or version file. Once a file is committed to storage as a version file (i.e., the file is considered a version file and a delta is stored in its place), it can no longer be used as a reference. A low resemblance threshold compounds this problem by allowing more files to be considered as versions of existing reference files than when a high threshold is used. When no *a priori* knowledge of future files is known, highly similar files might become version files. Our measurements show this is true, rendering a highly similar set of file data to compress is no better than *gzip* on a per-file basis.

Delta chains can be detrimental to storage performance and resource usage. Reference files can be virtual, meaning that they must first be reconstructed. When reference files, or version files that are used as delta references, must be retrieved or reconstructed, disk and computation are used first to reassemble and then to store the fully instantiated reference file.

We ameliorate the reconstruction cost from delta chains in a number of ways. The first and most important step is to reduce the delta chains when possible. If the reference file is selected arbitrarily, long delta chains can form. A simple change to associate a single delta chain length remedies the situation significantly, as seen in Figure 17. In this example, the reference file with the highest resemblance is selected. The first line listed in the legend minimizes the delta chain in selection; all things being equal, a reference file with the lowest delta chain length is selected. This resulted in an average chain length of 5.83. The second line attempts to compute delta against the maximum degree, but produced an average chain length over double (12.91) the previous method. The third method selected an arbitrary reference file, with an average chain length of 24.26. The last case is the same as the first, with the exception that the maximum chain length was 8 (-L 8). This may be useful for an analysis of reliability models at a small expense of storage space. With high levels of

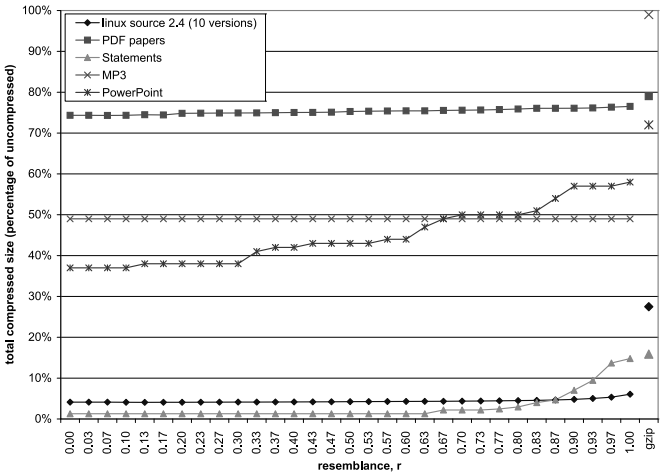


Fig. 18. Storage used when delta is applied: resemblance $(F1, F2) < r$ vs. gzip.

resemblance seen in this data set, careful base file selection can reduce chain length or increase the degree of dependence with low cost. The difference between minimum and maximum compression between these methods is approximately 3%, which is negligible compared to the 23.5:1 compression ratio.

A possible improvement is to rewrite some version files as reference files. When the system detects a large degree of dependence on a version file, the version file can be retrieved and stored as a reference.

Caching might further mitigate the effect of delta chains on retrieval performance. The issues in delta chains are largely related to performance, so file caching techniques, including predictive prefetching, may be useful. To date, we know of no research that has been conducted on caching in delta compressed archival storage systems.

Data-Dependent Compression. We measured the storage efficiency of *dcsf* over a number of data sets. In Figure 18 we show the amount of storage required to compress several data sets. The vertical axis indicates the total amount of space used when delta compression is applied above a threshold, indicated on the horizontal axis. (For illustration, the last data point on the data graph indicates the cumulative storage when each file is compressed with *gzip*.) For example, the Linux source data set (ten versions) shows on the far right that if delta compression were applied to files showing perfect resemblance of two sketches (at $r = 1.00$, equivalent to a single superfingerprint when $s = 1$), the compression reduces storage to 6% of the total, a saving which would be realized by one supershingle. The data set, labeled Statements, would be reduced to 14.8% of uncompressed data ($r = 1.00$), but more significantly, down to 2.5% ($r = 0.77$). With high probability, the harmonic superfingerprints at $s = 8$ would be able to detect those matching pairs. Finally, the PowerPoint data set shows a range of 58% compressed size for $r = 1.00$ down to 37% for $r > 0$, indicating that superfingerprinting would not detect redundancy, which is still significant.

5.2.5 Performance. In practice, space efficiency is not the only factor used to choose a compression technique; we briefly discuss some other important system issues such as computation and I/O performance. The chunking approach requires less computation

than delta encoding. It requires two hashing operations per byte in the input file: one fingerprint calculation and one digest calculation. (In practice, the digest calculation is deferred until a breakpoint is reached and all previously scanned bytes are used as input.) In contrast, delta encoding requires $s + 1$ fingerprint calculations per byte, where s is the sketch size. It also requires calculating the deltas, even though this can be performed efficiently, in linear time with respect to the size of the inputs. Additional issues with delta encoding include efficient file reconstruction and resemblance detection in large repositories.

To write chunks to the CAS, a hash table or distributed hash table must be first examined to determine placement of a chunk based on its chunk ID. Unlike traditional file systems, which can place object data based on directory, file name, or block number within a file, the chunk ID is intended to be globally addressable. During the file store operation, a write can be placed anywhere according to the mapping function provided by the CAS. However, during retrieval, chunks must be collected and retrieved. Over a distributed store, the random distribution of the chunk IDs lowers the probability that any single storage node contains all the data. However, the small sizes of chunks can make performance similar to a completely fragmented traditional file system. Wise engineering, such as clustering or ordering requests for a file, can improve performance.

The delta encoding method requires more computing resources than chunking, and is made up of three main phases: computing a file sketch, determining which file is similar, and computing a delta encoding. Currently, the most costly phase is computing the file sketch due to the large number of fingerprints that are generated. For each byte in a file, one fingerprint is computed for the sliding window and another 30 fingerprints are computed for feature selection. The shingling performance of our prototype on a Pentium 2.66GHz with 512KB L2, after many coding optimizations, is 19.7MB per second, but this can be increased by parallelization, for example.

The second operation, locating similar files, is more difficult. Our prototype implementation for this phase of the experiments, which preceded our discovery of the harmonic superfingerprinting technique, was not scalable since it compares a new file against all existing files that have already been stored; the search was exhaustive, over a moderately-sized data set. Fortunately, the discovery that the most significant storage benefits would come with highly similar data motivated our work to develop the harmonic superfingerprint, which provides additional compression benefit, with limited and constant cost. We are also optimistic that large-scale searches can be further improved given the existence of web-scale search engines that index the web using similar resemblance techniques [Broder et al. 1997].

The two techniques exhibit different I/O patterns. Chunks can be stored on the basis of their identifiers using a (potentially distributed) hash table. There is no need for maintaining placement metadata, and hashing may work well in distributed environments. However, reconstructing files may involve random I/O. In contrast, delta-encoded objects are whole reference files or smaller delta files, which can be stored and accessed efficiently in a sequential manner, though the placement in a distributed infrastructure is more involved.

Several additional issues exist for delta encoding that are not present with chunking. Because delta encodings imply dependency, a number of dependent files must first be reconstructed before a requested file can be retrieved. Limiting the number of revisions can bound the number of reconstructions at a potential reduction in space efficiency. Another concern that might be raised is the issue of intermediate memory requirements; however, in-place reconstruction of delta files can be performed, minimizing transient resources [Burns et al. 2002]. At first glance, it would appear that the dependency chain and reconstruction performance of delta files might be lower than reconstruction of chunked files, but since reference and delta files are stored as

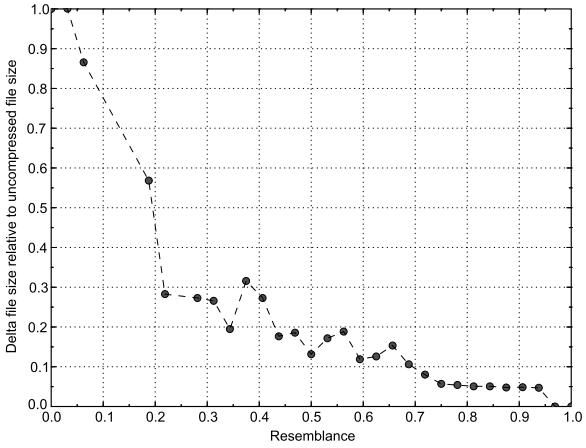


Fig. 19. Relationship between resemblance and delta compression against uncompressed data. A synthetic dataset is created using the code for evaluating the *zdelta* delta compression tool [Trendafilov et al. 2002]. Random data was written into two starter files, f_0 and f_1, each of size 1,048,576 bytes (1 MB), to ensure zero resemblance. Next, intermediate files, also 1 MB in size, were "morphed" by "blending" data from each of the two starter files using a simple Markov process parameterized by q and forming nonlinear similarities. The Markov process has two states, s_0, where we copy a character from f_0, and s_1, where we copy a character from f_1, and two parameters, p, the probability of staying in s_0, and q, the probability of staying in s_1. Here, $q = 0.5$ and p varies from 0 to 1 in 0.005 increments.

a single file stream and chunking may require retrieval of scattered data—especially in a populated chunk CAS—it is unclear at this point which method would produce worse throughput.

5.2.6 Relationship Between Resemblance and Storage Efficiency. We validate our hypothesis that using interfile compression between highly similar files is more efficient than compression between dissimilar files. Since mid- to low-similarity files were not easy to discover, we performed an experiment on synthetic data. First, we manufactured a data set from two random starter files and computed intermediates that were partially different between the two files. With this set of files, we ran the *dcsf* algorithm and computed the amount of storage each file would take if it were to be compressed using delta compression against the most similar file already stored. We repeated the experiment with user data.

Figure 19 illustrates the direct relationship between delta compression on similar files. Using a synthetic data set of 201 files, we used *dcsf* to first detect similar files and then used *xdelta* to compute against the most similar file detected. The resulting delta file size on the vertical axis is plotted as a fraction of the size of the selected reference file. We observe that the compression benefit is not directly proportional to the resemblance, which helps support the utility of the harmonic superfingerprints (where high resemblance detection provides disproportionately higher compression benefit over low resemblance data).

6. STORING DATA WITH PRESIDIO

Once redundancy has been eliminated, data is ultimately stored in a unified content-addressable storage, or a CAS, subsystem. Extending the CAS model, we describe our PRESIDIO Virtual CAS, or VCAS, which presents a unified storage interface externally, but stores and reconstructs data using virtual content-addressable objects. We

also present a low-overhead VCAS storage implementation that also preserves some of the temporal locality.

6.1 Content-Addressable Storage

We start with an abstract model of content-addressable storage. File content is identified by its content address (CA). A client program makes a *store* request to the CAS and is returned its CA. Later, another program sends a retrieve request with the CA and the original contents are returned.

We simplify the data storage model by separating file metadata from its contents. File metadata, which can include filename, ownership, location, and other, is serialized into a single stream. The metadata is also stored in the CAS and is identified by its CA. To improve the ease of implementation, we use the same content address namespace for both file content and file metadata.

Two other operations that would complete the set are Delete object and Verify object. These were not implemented. Deletion of VCAS objects might not cause data to be deleted from the store if files are still referenced. Hence, referenced file handles must be stored separately and the file content (CAS objects) must be reference-counted. The verification operation, had it been implemented, would test objects by first reconstructing them and then checking them against their content addresses.

6.1.1 Addressing Objects. CAS systems share common addressing properties. Content stored in CAS are immutable. This permits the system to address by content instead of location. Variable-length content can be reduced to a probabilistically unique identifier or address by hashing its contents using functions with very low collision rates. In many systems, one-way hashing functions are used in order to prevent intentional collisions of hash values.

Content addresses might refer directly or indirectly to a stored block of data. In other words, a hashing function like MD5 might compute its digest just over the block of data that is stored, or it might be computed over a handle that incorporates metadata that includes a content address within its structure that refers to raw content.

In order for a content address type to be reusable across a large storage corpus, its size, the function that computes values, and the bit representation of both the content as well as the address must be determined once and fixed for all time. Typical content address sizes start at 128 bits using cryptographic hash (digest) functions like MD5, which take variable-length byte strings as input and produce a small fixed-length hash value.

Because addresses can only be computed once the entire contents of a file are known, the performance of the hashing function is a matter of practical importance. Although some fast hashing functions are known to produce low probability of collision with arbitrary input, they might not satisfy a system design requirement to prevent tampering of data by substituting stored contents with manufactured data that hash to the same address. In other words, cryptographic hashes are often a design requirement, but they can reduce performance, as we have seen in Table VII.

Hash Collisions. Collisions are a metric of key collisions for CAS objects of arbitrary size, and not the error rate for the recovery for a single bit. If the storage of a CAS object was suppressed due to a collision, then retrieval would likely be incorrect for a whole block, whereas undetected ECC failure may return only slightly corrupted data. Furthermore, when we seek to eliminate redundancy by sharing common data through chunks or delta storage, the effect of a single error is magnified by the number of CAS objects (files) that depend on it. The failure modes between CAS and device are different, so comparing probabilities of error is difficult.

Due to the exceptional nature of collisions, we believe collision resolution can be handled by detecting collisions, definitively determining whether data is different, and then storing additional metadata in CAS subsystems (or in our case, CAS storage groups) to indicate objects with colliding addresses.

6.2 Implementation

We implemented a CAS server on a single host aimed at storing a large number of variable-length data blocks. Primary considerations were to index each block by its content address, to minimize storage overhead per block, to present a small set of functions, and to improve performance with easily implemented modifications.

We now give a summary of our block storage strategy. First, the CAS divides an input file to create smaller pieces of data to identify duplicate or similar data. This data may potentially be shared with other files, so a store of previously written data is suppressed. Second, the CAS data is written in order to storage segments, called *megablocks*. Third, the megablocks are indexed to internally address the VCAS storage. Fourth, the indexed megablocks form groups, which can be distributed across the storage system. The implementation uses these principles to record variable-length block data using flat files and databases storing key-value pairs.

6.2.1 Design Overview. Reducing storage overhead in an immutable content-addressable store is a simpler problem than in read-write file systems. In a fixed-content CAS, objects are written but never modified; this immutability property allows a system to organize data in a way such that the written data does not need to be moved or resized to accommodate append or truncate operations. In contrast, read-write file systems need to allocate extra space when files are appended, and most designs use fixed-sized blocks, which exhibit internal fragmentation when not completely full. Furthermore, file system performance goals typically require extents or other sequential grouping strategies to better utilize the highly beneficial sequential throughput that hard disks exhibit as opposed to high latency when random block placement is used.

Like many storage systems, our design assumptions were aimed at balancing space efficiency against read and write performance. To reduce per-file and per-block overheads that are commonly found in storage systems using fixed block sizes, we used variable-length blocks. To facilitate high write throughput, we used a lazy write placement model by writing sequential data. To help reduce read latency, we indexed our data by using two levels of indexing, the first containing primary indexes to record offsets similar to the *indexed-sequential files* [Wiederhold 1983] or the *indexed sequential access method* (ISAM), and second, to use a hash table to map content addresses to the primary indexes.

Sequential Block Storage. File systems exhibit behaviors that can be used to improve storage read and write performance. Files can be related by their address (namespace), or they can be related by time. Files or directories may be placed together when they are created or dynamically due to the write patterns. In contrast, content-addressable storage is designed to use addresses with random distribution of values, making locality of reference by address impossible.

Fortunately, file access patterns are typically related by time and ordering, making it possible to improve performance by using temporal locality. The Log-structured File System (LFS) [Rosenblum and Ousterhout 1992] writes blocks linearly onto a storage device as they arrive, having the effect of grouping temporally related blocks together. Results from HighLight, a *hierarchical storage management* (HSM) system extending LFS, suggest that by using a migration policy to group by access time instead of

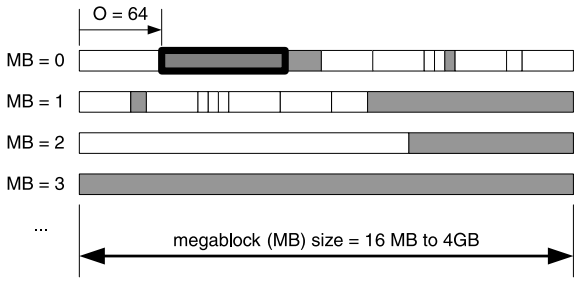


Fig. 20. Megablock (MB) storage.

namespace, the storage system can achieve high bandwidth and low latency by using secondary caching with tertiary storage [Kohl et al. 1993].

Megablocks. Megablocks are user-level container files containing temporally-grouped variable-length data. The underlying block storage implementation is similar to *chunks* used in the Google File System (GFS) [Ghemawat et al. 2003], large flat files for containing parts of other files. In our implementation, the large container files are also flat files with no special metadata attributes, and range in size from 16MB (represented by 24 bits of offset) to 4GB (represented by 32 bits of offset). We selected this range for the following reasons. Small numbers of large file sizes require less overhead per file than larger numbers of small files. Clustered file systems like GFS and GPFS [Schmuck and Haskin 2002] improve access by multiple clients when using larger block allocation sizes larger than 64KB (versus file system allocations typically at 4KB) due to the lower per-block overhead for managing the block storage and associated file allocation tables across multiple storage nodes. Very large files would not be limited to these boundaries due to the nature of the VCAS storage object model, which supports arbitrarily large files through composition. In other words, a very large original file could be stored and reconstructed from concatenated blocks.

In Figure 20, megablocks from zero (MB = 0) through three (MB = 3) are shown as horizontal rectangles representing flat files. File content is shown as dark gray rectangles, which represent single VCAS objects. The object with the heavy outline is located with an offset of 64 bytes from the beginning of the file (O = 64).

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 (not yet implemented) will reclaim unused space. A fixed megablock size from 16MB to 4GB is selected for uniformity across nodes and the ability for 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 and GFS. 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.

Group Storage. Collections of megablocks are combined into *groups*. The grouping offers distribution across nodes to distribute load and to improve reliability. A storage group contains a number of megablocks, placed on a recording device using a storage mechanism such as a cluster file system. 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

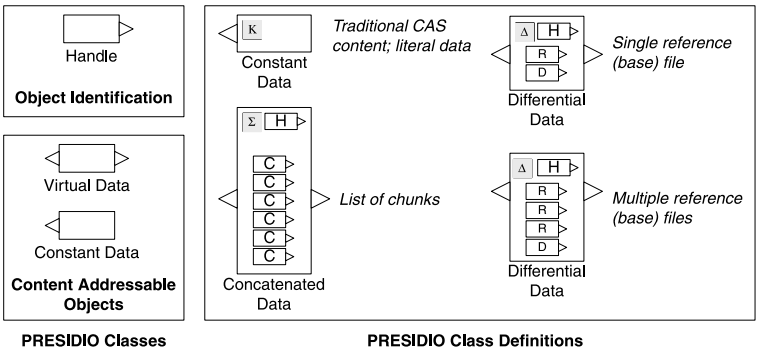


Fig. 21. PRESIDIO data classes.

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.

VCAS and Object Types. The main data types in PRESIDIO are illustrated in Figure 21. Handle is 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. The Handles are in-memory and on-disk objects that are used throughout PRESIDIO as a representation of a content address. Next, two basic classes represent objects that are also used in-memory and on disk: Constant Data Block is a content-addressable data block of variable size containing a string of bits that is stored literally (i.e., raw binary data), and Virtual Data Block is a content-addressable data block. In turn, virtual data blocks are defined having one of the following polymorphic blocks. Each block contains a type code such as "constant" (signified by K), "concatenation" or "chunk list" (Σ), and "differential" or "delta" (Δ); and a combination of raw data and handles. Each block can be referenced by Handle (CA); 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.

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.

CAS objects are untyped data made up of a sequence of bytes. These objects are used to persistently store a wide range of data including whole files, partial files, delta files, chunks, sketch data, metadata, and so on. Because our CAS uses only one type of storage object, storage operations are easily specified and implemented.

To locate a single content-addressable object, a handle is first 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. 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.

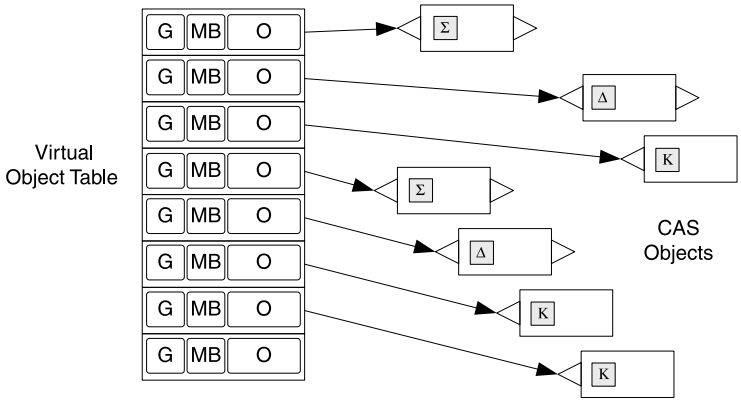


Fig. 22. PRESIDIO content-addressable object storage.

Figure 22 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).

6.2.2 Indexing. Once data is recorded, we provide an index to map content addresses to megablock positions. Our implementation uses two-level indexing: a simple Berkeley DB *Hash* (B-tree hash table) database that maps the CA to the sequential index (an integer record number) and then a Berkeley DB *Recno* database [Oracle Berkeley DB 2010] (record number as keys) that maps a sequential index to a variable-length block (record) location. The *Recno* database is effectively an array whose elements containing the sequential block positions are accessible by the sequential index. These two levels of indirection achieve two goals: to allow sequential data to be written quickly, and for CA retrieval to be fast by using very small hash entries consisting of CA and *recno*.

Using just a single-level hash table (a Berkeley DB Hash table of key and value pairs) has several problems: the number of keys stored in memory is low (as Berkeley DB stores both the keys and values pairs in memory) and lack of locality of reference makes reads slow, while writes can thrash the database once hash tables exceed available memory.

Since the Berkeley DB Hash storage method gives the best random access performance if all of the keys and values are in memory [Seltzer and Yigit 1991], one design change was simply to store the content address as the key and a secondary index (a 32- or 64-bit ID) to a separate storage database. This maximizes the number of CA that can be in memory; otherwise, disk reads may be necessary.

Next, block contents were organized by writes. Just as in LFS, we appended blocks as they arrived into a megablock file; when each megablock file reached a limit, it was closed and a new megablock was started. Our experimental results show that the 2-level scheme does not suffer ftom thrashing effects that the 1-level scheme does when Berkeley DB is used.

Although our initial prototype is sufficient for investigation and experimentation, we believe there is room for improvement. What traditional file systems do better than our system is to exploit locality of reference, whether it be from storing files in

directories together, sharing i-nodes, using block extents, or using temporally-related or access-dependent caching or prefetching. In contrast, our goal for improving storage space efficiency improves *locality of associativity*, which does not clearly align with high performance. As such, it will be important for future work to incorporate inherent locality, even when the content addressable architecture does not directly reveal such relationships.

6.3 Metadata

While the somewhat minimal design of a content addressable store simplifies an application programming interface, it lacks feature parity with traditional file systems that support metadata such as a file system object's name, the object's location within a storage organization (directories or linkage), permissions and ownership (and by design, operations to allow file mutation). Such metadata, while not counted in measurements of file size, are necessary overhead and must be included as a design consideration when describing data objects, as they may increase the overall storage requirements. Furthermore, archival metadata often provides not only a description of the data, but also how to interpret it beyond the lifetime of the storage medium, making it especially verbose.

The PRESIDIO CAS stores metadata which can be either file metadata, primitive file system-level information about the raw content of the file (such as size of file and creation date), or rich metadata, application-level information (such as keywords, format, thumbnails). Our storage system is designed to store the most primitive metadata necessary to access files, while allowing rich metadata associated with files to be stored in a flexible format within the PRESIDIO VCAS (along with the file itself), reducing storage overhead.

6.3.1 Archival Metadata Formats. File archives contain both file content as well as file metadata. We examined some Unix file formats for their size overhead in order to assemble a simple metadata format that could be used to archive and restore commonly used files.

Two Unix uncompressed archiving programs, *ar* and *tar*, and two compression programs, *gzip* and *xdelta*, are common examples of programs that retain file metadata. We measured files with one byte length and then disassembled the file format to determine what would constitute a minimal set of file attributes. In Table VIII, we list two files, a and b, to show the relative file overhead for each single-byte file. Because the *ar* file format was small, we extracted the fields that would commonly be used. Next, we converted each field into a corresponding XML entity using a representation in text, thus producing a suitable replacement that is easily interpreted. The metadata was then separated from file content and then converted into a text representation that was then stored as another content-addressable object.

6.3.2 Rich Metadata. Metadata will undoubtedly play 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 that 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 capabilities [Crespo and Garcia-Molina 1998; Mahalingam et al. 2002]. Unfortunately, storing the rich metadata substantially increases file size overhead, when space efficiency is an important goal in reference storage.

Rich metadata storage may have different storage requirements than the archival data it references [You et al. 2005]. It may be desirable that the metadata not have the same immutability requirement, since metadata content such as keywords or access

Table VIII. Archive File Metadata Sizes

filename	size (bytes)	command to create file
a	1	`echo -n a > a`
b	1	`echo -n b > b`
a.a	70	`ar -q a.a a`
b.a	70	`ar -q b.a b`
a.a.gz	69	`gzip a.a`
b.a.gz	69	`gzip b.a`
a.tar	10,240	`tar -cf a.tar a`
a.tar.gz	122	`tar -zcf a.tar.gz a`
a.tar.gz	128	`tar -cf a.tar a; gzip a.tar`
ab.tar.gz	143	`tar -zcf ab.tar.gz a b`
b.delta	209	`xdelta delta a.a b.a b.delta`

control parameters may be updated based on changing criteria. In an immutable storage system this requires updated versions of metadata to be stored. Using PRESIDIO, this rich metadata can be compressed against previous versions using the best efficient storage method (ESM) available. Archival metadata is often stored in a representation such as XML for which increases in size due to versioning can cause a linear increase in total file storage size when delta compression is used, while other content-specific compression mechanisms, like XMill for XML [Liefke and Suciu 2000], have much higher space efficiency [Buneman et al. 2002]. For archival systems requiring versioning of rich metadata, content-specific compression techniques can be easily integrated as another ESM within PRESIDIO.

6.4 Reliability

If files share data due to interfile 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.

In previous work [You 2006], we examined the effect of data dependencies on the delta chain length. The loss of highly dependent files used in delta compression, which are necessary for reconstructing other files, will have a more adverse and disproportionate effect on data loss than independent data files. Subfile chunking also introduces interfile dependencies. If a large set of files all contained the same chunk, for example, which is 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. A new reliability model for chunk- or delta-based compression would place increased value on highly referenced chunks or files. 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. In our work on reliability [Bhagwat et al. 2006], we have shown how a simple strategy may be used to increase the robustness of data, controlling the balance between space efficiency and reliability by a choice of heuristics and parameter variation.

Another issue that must be considered for preventing catastrophic failure is that of data distribution. 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.

Storage systems that distribute data for security or for reliability exist, but they assume data has unit cost. Distributed storage systems such as Petal [Lee and Thekkath 1996], OceanStore [Kubiatowicz et al. 2000], Interarchival Memory [Chen et al 1999; Goldberg and Yianilos 1998], FARSITE [Adya et al. 2002], Pangaea [Saito and Karamanolis 2002; Saito et al. 2002], and GFS [Ghemawat et al. 2003] address the problem of recovering data from a subset of all stored copies. In these systems, both plaintext and encrypted data are distributed, and recorded with simple replication or error-correcting coding schemes. However, they do not distinguish between degrees of dependent data.

It is clear that reliability guarantees for a system storing data with interfile dependencies is a particularly difficult problem. Building a model that derives the value of data and developing a strategy for data placement are the subjects of future work.

6.5 Deletion

We designed Deep Store as a permanent archival storage system, in which files are written once and retained forever. However, commercial applications may require that files are stored with limited retention periods to meet "information lifecycle management," legal, or regulatory compliance goals, in turn requiring file deletion or data migration. Content-addressable storage systems face a traditional garbage collection problem: objects are referenced, but once all references to those objects are eliminated, then the object should no longer be retained. These circumstances present several challenges to data deletion. First, data retention is not an immutable property of the stored data; retention periods may change over time due to changing regulations or requirements such as legal data preservation imposed after the data is written. Changing requirements suggest metadata is needed to manage virtual copies of addressable content. Second, content address references must be managed. In our system, references to permanent data allow a client to rely on the data indefinitely. To change the semantics of held references as an ownership to an object, that is, a file cannot be deleted as long as a reference is held, the content address references must be managed such that file deletion could invalidate client references. Such invalidations suggest that the client (content address) references must be held in a closed system, rather than allowed to be copied freely by the clients. Third, systems that share common content, whether through chunks or by delta compression, require that the data dependencies can be reversed before an interdependent file object is deleted, since regulations may require absolute deletion, as opposed to achieving inaccessibility by letting references dangle. Such multistep deletion operations are best handled as offline operations, removing files and rewriting interdependent files back to the archival store. This work is beyond the scope of this article.

7. PROGRESSIVE COMPRESSION

The Progressive Redundancy Elimination of Similar and Identical Data In Objects framework, or PRESIDIO, is a combination of an algorithm and framework to compress data across files in an archival store using progressively improved methods. We now bring together the compression framework with the object storage framework, and an adaptive algorithm to select compression methods to yield high compression of highly redundant data and low overhead for highly unique data.

As the space efficiency of lossless data compression is highly data-dependent, a practical solution is to progressively apply data compression algorithms which yield high efficiency and high performance first, and only when they do not provide a satisfactory result, apply lower efficiency and lower performance algorithms. As we have shown, even lower resemblance data may still provide a benefit of reducing storage,

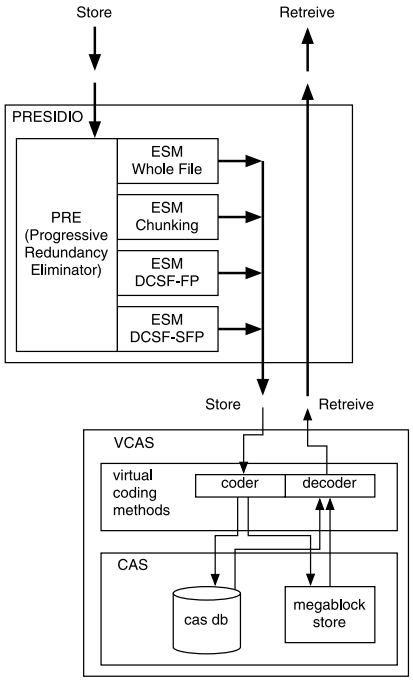


Fig. 23. PRESIDIO ESM and CAS.

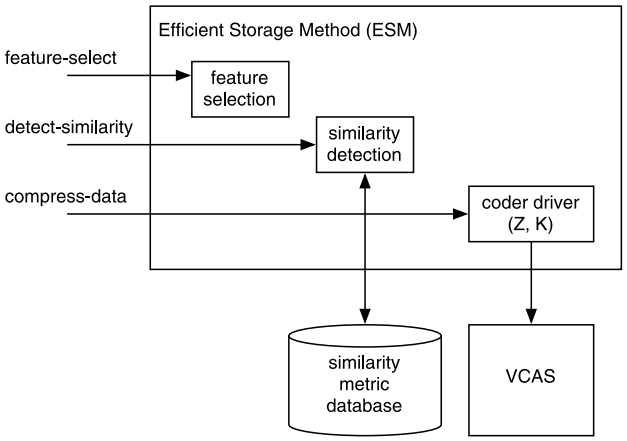
but with diminishing returns. However, the tradeoffs are dependent on workloads, application requirements, and computing resources that are available for storing as well as retrieving. In PRESIDIO, each data compression algorithm is modeled to provide an approximate data compression efficiency yield as a function of the resemblance of input data. The properties of the model include expected data compression storage efficiency, the input read rate, and the output write rate. When a candidate file is presented to PRESIDIO, it evaluates the data and compression models to compute scores for each algorithm. The algorithm with the best score is applied. If the result of the data compression achieves the compression threshold for the model for both storage space and performance, the algorithm is used, otherwise the next-highest algorithm is applied. The evaluation process is repeated until the available compression algorithms are exhausted.

7.1 The PRESIDIO Storage Framework

The PRESIDIO storage framework consists of a layered architecture with the following major components:

— the *progressive redundancy eliminator* (PRE) compression algorithm;
— *efficient storage methods* (ESMs);
— *virtual coding methods* (VCMs); and
— a *virtual content-addressable store* (VCAS)

Figure 23 illustrates the main storage components, the mid-level PRESIDIO framework, and the low-level VCAS and CAS storage subsystem. PRESIDIO allows its storage interface to detect, select, and employ the most space-efficient storage method. Its progressive redundancy eliminator algorithm (PRE) links against a set of known ESM classes; from each class and object instance that is created. PRE calls methods in each




```
feature-set = feature-select(file, content-address)
(r, resemblance-state) = detect-similarity(file, content-address, feature-set)
compress-file(file, content-address, feature-set)
```

Fig. 24. Efficient storage method operations. Coder driver marshals parameters that will be passed to the VCAS to encode a file.

ESM object in turn, using polymorphic object-orientation to select features and to detect resemblance. Once PRE selects an ESM, it is called to compress a file into the VCAS.

The VCAS storage subsystem is made up of two parts: *virtual coding methods*, which use content addresses to represent virtual but not internal representation, and the CAS, which uses content addresses to represent real content. A low-level VCAS interface is used to write objects into a content-addressable store by encoding them using a small number of virtual coding methods. Each method is made up of a coder and decoder; the virtual coding methods are codecs. The CAS implementation we use consists of a simple CAS database and megablock log file store, as described in Section 6.2.

There are some notable differences in this design when compared to existing CAS systems. The first is the use of multiple methods for detecting similarity and eliminating redundancy, unlike pure CAS systems like Centera that use a single method such as whole-file hashing or chunk-based hashing. The second is the use of a virtual content-addressable store instead of a traditional CAS. The third is the use of an algorithm to detect and use efficient storage in a progressive manner. The last is the use of a storage system design that combines multiple compression methods into a hybrid archival storage system.

7.2 Efficient Storage Methods

The PRESIDIO architecture defines ESMs as a combination of data and algorithms or simply, a class of objects whose purpose is to select features from a file, determine the effectiveness of a specific compression method for a given file, and then to marshal the parameters used to drive the input to a VCAS coder. ESMs are expressed as class definitions with the objective of compressing a single type of CAS compression.

Each ESM implementation (Figure 24) uses the same common programming interface made up of three algorithms, implemented as independent or inherited functions. Additionally, some state may be shared between multiple ESMs.

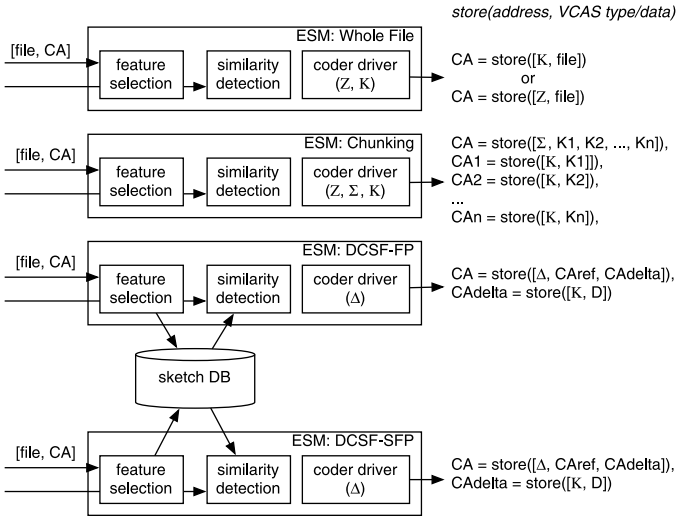


Fig. 25. PRESIDIO efficient storage methods.

Figure 25 illustrates the different ESMs we have developed. Each ESM box represents a different implementation. Some ESMs may share state, for example in the case of DCSF-FP and DCSF-SFP, the sketch database contains fingerprints per file. On the right, the coder driver will call the VCAS with specific parameters. We describe the ESM interface and then each of the ESMs in turn.

7.2.1 ESM Interface. The ESM interface contains two types of information: a pointer to a polymorphic implementation object and PRE evaluation state. The internal state consists of the following. The *virtual length*, l_v, is equal to the length of the file in bytes presented in a *store* request. The *real length*, l_r, is the approximate or exact storage size in bytes of the file, including storage overhead for intermediate VCAS structures.

We define *efficiency* as the incremental storage size of a newly presented storage file as a fraction of the virtual storage size, where 0 is perfect efficiency, no additional bytes stored; and 1 is no efficiency, all input data is stored uncompressed. PRE scoring is computed from real storage efficiency, $u_{real} = l_r/l_v$, and biased storage efficiency, $u_{biased} = b\,u_{real}$ where bias, b, is a coefficient to modify the efficiency score.

Bias is a factor to help offset the introduction of storage overhead for ESMs that have a potential benefit, but not during initial evaluation. For example, the whole-file hashing ESM will introduce no virtual storage overhead in the VCAS because the entire file can be stored as a single CAS object. A chunk-based ESM would introduce a list of chunk CAs, at a slight increase in storage. One reason to use bias is when chunks were never stored in the first place, then there would be no opportunity to share chunks later. By introducing bias, slightly inefficient ESMs, for example, chunking, can be selected opportunistically despite that method's higher, nonbiased u. Our implementation uses a constant bias, but adjusting bias dynamically is a potential area to further improve storage efficiency.

7.2.2 ESM: Whole-File. The *Whole-File* ESM implements the standard content-addressable store, corresponding to the schematic in the upper box in Figure 25. This ESM has the highest possible compression when content addresses match exactly. When identical files are presented to the ESM, it can easily and quickly detect the existence of a file through a hash lookup in the VCAS database. Compression is

actually implemented as suppression of any storage operation to the VCAS. The feature selection algorithm returns the hash of the file; our prototype uses the MD5 hashing function.

The similarity detection method is implemented by testing for the file existing in the VCAS using the content address. Efficiency u_{real} is 0 when the file exists or 1 if not. (Resemblance r is binary: $r = 0$ when the file is not found in the VCAS, or $r = 1$, when the hash for the file already exists in the VCAS.) The coder driver function marshals the "constant data" operation, K, and the file data and the VCAS operation "Zlib compression" is used.

7.2.3 ESM: Chunking CAS. The chunking algorithm uses the following parameters: window size of 32, the min/max/expected chunk size 64/4096/1024B, the Rabin polynomial is of degree 32, and, if coefficients are encoded big-endian, of value `0x1A8948691`, the breakpoint residue is 0, and chunk ID hash (also CA size) is 128bits. The feature set that is returned is a sequence of content addresses $CA_1, CA_2, \ldots, CA_n$ corresponding to the chunks $K_1, K_2, \ldots K_n$ such that the concatenation of chunks K_i is the input, file.

Similarity detection computes the potential real storage size by iterating over the candidate chunks in two steps. First, for each chunk, it determines whether the chunk has already been stored, in which case the incremental real chunk size is zero, otherwise the real chunk size is computed from the similarity detection using PRE. In other words, an entire chunk itself is evaluated recursively for its potential real size. Second, each chunk K_i is added to the chunk lists and then that list is evaluated for its size, using recursive PRE. The recursive method for similarity detection has one major benefit: if a chunk list has already been stored, or if resemblance can be detected from the chunk list, then it too will be stored efficiently.

Our current recursive implementation statically computes the intermediate results, in effect performing many of the compression steps to compute the actual VCAS real (compressed) object size. Heuristic evaluations of the virtual content may be able to yield qualitatively similar results without incurring the read/write or computational overhead.

The coder driver marshals the parameters: a concatenated "chunklist" (Σ) of chunks. Each chunk is submitted back to PRE so that it can also be stored efficiently, generally either as constant chunks (K) or compressed chunks (Z). In practice, only compressed chunks are written. To record the files, the chunklist is written to the VCAS followed by chunks that have not already been written.

7.2.4 ESM: DCSF-FP. Both the *delta compression of similar files using fingerprints* (DCSF-FP) and *delta compression of similar files using superfingerprints* (DCSF-SFP) start with identical feature sets, but use slightly different similarity detection algorithms. Computing DCSF feature sets is a computationally expensive operation, so it is only computed once. However, once it is determined, it is easy to compute additional features, for example, harmonic superfingerprints, that make similarity detection faster due to a smaller number of feature lookups. Separating the DCSF-FP and DCSF-SFP algorithms into two ESMs makes it possible to share common data and to find highly similar data quickly.

Feature selection computes a sketch of fingerprints. Each sketch is a vector of $k = 32$ features selected from the file using k functions to independently select features from a sliding window of size w in the file using the min-wise independent permutations. Similarity detection is a comparison between a file with a sketch and all sketches in the database. The comparison function returns resemblance r ($r = max(n/k)$ for all sketches, where n is the number of features that match).

Our implementation does not currently use information retrieval (IR) techniques to retrieve from feature vectors of size k. The implementation can be performed using methods such as inverted word lists, where each word is represented by a fingerprint or by using Bloom filters. In both cases, the in-memory and on-disk usage are high, and such overhead may not offset potential compression gains.

The coder driver writes a delta (Δ) between a reference file (R) and a delta file (Δ); a delta file is also written as constant data (K) and the contents of the delta file itself (D).

7.2.5 ESM: DCSF-SFP. At the time a sketch is computed, the harmonic superfingerprints are also easily computed. Superfingerprint features give high probability of matching sketches with high resemblance using a much smaller set of fingerprints. DCSF-SFP similarity detection is a comparison between one or more superfingerprints. Instead of comparing entire feature sets (or feature vectors), one or more superfingerprints are compared. When harmonic superfingerprints are used, one superfingerprint covering all features $f_i, 0 \leq i \leq k$ can be indexed directly. Retrieval and comparison of harmonic superfingerprints covering a subset of sketch $S(A)$ require additional indexing and retrieval. The coder driver for the DCSF-SFP is identical to DCFS-FP.

7.3 Virtual CAS (VCAS)

The architecture of the VCAS incorporates a set of *codecs*, or coder/decoder pairs. The VCAS is a low-overhead virtual object (VO) storage mechanism. The absence of block allocation eliminates internal fragmentation common in many file systems. VCAS object encodings (VOEs), the metadata that specifies how to reconstruct files, are stored within the VCAS itself. Each VOE is written as a single serialized stream of bytes that can be stored in a CAS and addressed by its CA. VOEs can also be stored as virtual objects themselves, but in practice they are small (size $\ll$ 1 KB) and stored simply as constant (literal) data.

Polymorphic object storage lets us reconstruct based on methods that are stored with the data. Each VOE is a single method that can be used within a recursive VO definition. VO reconstruction is a recursive reconstruction of the underlying data.

VOE coder and decoder methods are separated from the efficient storage methods (ESMs). Once files have been stored, only the decoder implementations are required. To help ensure permanence by lowering software maintenance, the more complex similarity detection, redundancy elimination methods, and efficient storage methods are not needed once the data has been recorded. To further simplify the need for maintenance, only the VOE decoder implementation is needed.

7.3.1 Progressive Redundancy Elimination. PRESIDIO incorporates the *progressive redundancy eliminator* (PRE) algorithm. Its purpose is to use a collection of ESMs to first determine the best method for eliminating redundancy, and then to encode the data into virtual object encodings. Next, the encodings and original file content are passed to the VCAS where the data is written.

The delta encoding method requires more computing resources than chunking and is made up of three main phases: computing a file sketch, determining which file is similar, and computing a delta encoding. Currently, the most costly phase is computing the file sketch due to the large number of fingerprints that are generated. For each byte in a file, one fingerprint is computed for the sliding window and another 20 fingerprints are computed for feature selection.

The second operation, locating similar files, is more difficult. Our current implementation is not scalable since it compares a new file against all existing files that have already been stored. We are optimistic that large-scale searches are possible given the

existence of web-scale search engines that index the web using similar resemblance techniques [Broder et al. 1997].

PRE determimes the most space-efficient ESM, selects its encoding, then commits the encoded data to permanent storage. For the first file being stored, a chunking storage with stream compression may be selected. The second, but slightly different, file may be encoded against the (previously encoded) chunking instance using DCSF-SFP.

Chunk storage and delta compressed storage exhibit different I/O patterns. Chunks can be stored on the basis of their identifiers using a (potentially distributed) hash table. There is no need for maintaining placement metadata, and hashing may work well in distributed environments. However, reconstructing files may involve random I/O. In contrast, delta-encoded objects are whole reference files or smaller delta files, which can be stored and accessed efficiently in a sequential manner. But placement in a distributed infrastructure is more involved.

In order to select the best compression, PRE simply iterates through each ESM, computing the net change to stored data and then selects the smallest value. An exact whole-file match, or $u_{real} = 0$, is ideal. Clearly the exhaustive evaluation is costly. Future work may better exploit the relationship between heuristic resemblance and the predicted reduced size.

A couple of additional issues exist for delta encoding that are not present with chunking. Because delta encodings imply dependency, a number of dependent files must first be reconstructed before a requested file can be retrieved. Limiting the number of revisions can bound the number of reconstructions at a potential reduction in efficiency. Another concern that might be raised is the intermediate memory requirements; however, in-place reconstruction of delta files can be performed, minimizing transient resources [Burns et al. 2002]. At first glance, it would appear that the dependency chain and reconstruction performance of delta files might be lower than reconstruction of chunked files, but since reference and delta files are stored as a single file stream and chunking may require retrieval of scattered data especially in a populated chunk CAS it is unclear at this point which method would produce worse throughput.

7.4 Prototype Implementation

We have implemented a prototype of PRESIDIO that demonstrates the ability of our solution to eliminate redundancy progressively within an experimental VCAS and efficient storage method framework. Feature selection is an important part of our prototype. We developed a flexible Rabin fingerprinting library in C++ for this purpose, with high throughput. In addition, we used open-sourced libraries such as *md5sum* and *libopenssl* to compute whole file hashes. We implemented a hash-based storage prototype using Berkeley DB Database [Oracle Berkeley DB 2010; Seltzer and Yigit 1991], a multipurpose embedded database storing key-value pairs in linear *hash* and *recno* (record number) databases. The Virtual CAS was implemented on top of the database as a C++ framework with multiple object-recording implementations. Each virtual coding method was encoded numerically, each representing an instance of object-oriented classes; polymorphic Store and Retrieve functions performed the actual work of storing and retrieving the data. Additional implementation details may be found in You [2006].

8. CONCLUSION

The main contribution of this article is to unify archival storage solutions to maximize space efficiency. To achieve this goal, we have described the design and implementation of the Progressive Redundancy Elimination of Similar and Identical Data in Objects (PRESIDIO). The premise of the development system is threefold: the storage

architecture presents a simple content-addressable storage interface; a unified virtual content-addressable store encodes data using polymorphic methods that use multiple storage methods; and an internal storage framework in which to express and evaluate multiple efficient storage methods.

In addition, we have also constructed prototype systems, and present experimental results as lesser contributions:

— Simulation and analysis that provide empirical results on choosing the best size for the sliding window and the expected chunk size in chunking algorithms, based on different kinds of datasets.
— Simulation and analysis that provide empirical results on choosing the best shingle size and sliding window size for the shingling technique used in similarity detection.
— Simulation and analysis of the relationship between interchunk and interfile dependencies and their effect on storage efficiency.

Whereas file systems do not typically analyze files for content, archival storage systems in particular, efficient archival storage systems require multiple types of content analysis to detect and eliminate redundancy. Another significant difference in our work is to abstract a file's contents from the underlying mechanisms from which it is stored. While content may not be the only way to identify data, automatically selecting features and performing similarity detection across many files of arbitrary types creates the opportunity to think about file storage in other ways. Additionally, our design advances the idea storage and data retrieval should be easily separated by creating self-descriptive data structures from which contents can be retrieved.

Extensive data dependencies, not common in files in traditional file systems, are prevalent in an environment where inter-file data compression is a means to the goal. While aggregate statistics are useful, many types of investigative questions are not easily answered by viewing numbers or tables; we found two- and three-dimensional representations of dependency graphs and graphs showing similarity or resemblance invaluable. However, with large numbers of data points, many tools did not scale well enough to give both "big picture" and detailed information.

The future may be brighter for data storage with high dependency chains with the use of low latency solid state memories due to the lower random access costs. With this, PRESIDIO would offer higher reconstruction throughput and potentially higher performance for retrieval of feature data that are used for similarity detection. In this case, low latency could open further opportunity for additional gains in storage efficiency.

8.1 Future Work

One main concern for the Deep Store was to store large amounts of data reliably over a large number of storage nodes while focusing on space efficiency and realizing scalable storage. In particular, the flat namespace of a CAS system would permit data distribution easily by partitioning a hash space, for example by using distributed hashing like LH [Litwin and Neimat 1996; Litwin et al. 1993, 1996] to partition and distribute the tables. We have not looked seriously at aspects relating to distributed storage, its replication, scalability or reliability, though we believe that our solution offers a foundation. For example, our architecture decouples naming and addresses from content, making it possible for data to be distributed widely. In addition, the architecture is agnostic to file content and metadata format, allowing different system and programming interfaces to be layered on top of the system. Distributed file storage, using distributed hashing or peer-to-peer architectures, can be built on top of PRESIDIO.

Reliability models that assume probabilities of failure per byte or per block do not address changes when data dependency in shared chunks or delta compressed data. When compounded with distributed data storage, the reliability models may be difficult to describe. We can also improve compression efficiency if properties of content type that is being compressed is known. Our system aims to treat all binary data without discrimination, and doing so may yield further storage benefits.

Although we have tried to address one problem, that of space efficiency, many hard problems still exist. Most data today is encoded in one form or another; our virtual encoding is no exception. Without effective mechanisms to ensure correct interpretation of bits in perpetuity, the bits themselves will become meaningless. A shortcoming of an extensible system such as PRESIDIO is that the ability to decompress programs in the future also depends on the existence of the software, operating environment, and hardware required to execute the decompressors. This is a very difficult problem that needs to be solved if long-term digital archival is to become practical.

Storage performance is measured by the rate of ingest, and retrieval performance is measured by both latency and read bandwidth. Thus, the PRESIDIO steps to identify and detect similar data must also satsify the high throughput requirement. Searching for data may be resource-intensive, either computationally or in memory usage, when high dimensionality searches are used. Indexes and inverted indexes are memory-inefficient. Our initial solution was to use low-dimensional spaces—for example, through harmonic superfingerprinting—to keep search complexity low. Other methods may help reduce search, for example with multidimensional extendible hashing [Otoo 1986; Ouksel and Scheuermann 1983], or with approximate nearest neighbors [Indyk and Motwani 1998].

File reconstruction requires that interior data is first reconstructed. As we have shown, efficiently stored delta-compressed data form dependency chains can be longer than one. However, due to caching of commonly linked data, reconstruction may be much faster; furthermore, techniques of prefetching and aging are applicable in the context of archival systems.

ACKNOWLEDGMENTS

We thank Christos Karamonolis, Kave Eshghi and George Forman of Hewlett-Packard Laboratories for their help and insight into the behavior of file chunking. 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

ADYA, A., BOLOSKY, W. J., CASTRO, M., CHAIKEN, R., CERMAK, G., DOUCEUR, J. R., HOWELL, J., LORCH, J. R., THEIMER, M., AND WATTENHOFER, R. 2002. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, Berkeley, CA.

AJTAI, M., BURNS, R., FAGIN, R., LONG, D. D. E., AND STOCKMEYER, L. 2002. Compactly encoding unstructured inputs with differential compression. *J. ACM 49*, 3, 318–367.

ALVAREZ, C. 2010. NetApp deduplication for FAS and V-Series deployment and implementation guide. Tech. rep. TR-3505, NetApp.

APACHE SUBVERSION. 2010. http://subversion.apache.org/.

BHAGWAT, D., POLLACK, K., LONG, D. D. E., SCHWARZ, T., MILLER, E. L., AND PARIS, J.-F. 2006. Providing high reliability in a minimum redundancy archival storage system. In *Proceedings of the 14th IEEE International Symposium on Modeling, Analysis, and Simulation*. IEEE, Los Alamitos, CA, 413–421.

BRIN, S. AND PAGE, L. 1998a. The anatomy of a large-scale hypertextual web search engine. In *Proceedings of the 7th International World Wide Web Conference*. 107–117.

BRIN, S. AND PAGE, L. 1998b. The anatomy of a large-scale hypertextual web search engine. http://www-db.stanford.edu/_backrub/google.html.

BRODER, A. Z. 1993. Some applications of Rabin's fingerprinting method. In *Sequences* II: *Methods in Communications, Security, and Computer Science*, R. Capocelli et al. Eds., Springer, Berlin, 143–152.

BRODER, A. Z., GLASSMAN, S. C., MANASSE, M. S., AND ZWEIG, G. 1997. Syntactic clustering of the web. In *Proceedings of the 6th International World Wide Web Conference*. 391–404.

BRODER, A. Z. 1998. On the resemblance and containment of documents. In *Proceedings of Compression and Complexity of Sequences (SEQUENCES'97)*. IEEE, Los Alamitos, CA, 21–29.

BRODER, A. Z., CHARIKAR, M., FRIEZE, A. M., AND MITZENMACHER, M. 1998. Min-wise independent permutations. In *Proceedings of the 30th Annual ACM Symposium on Theory of Computing (STOC'98)*. 327–336.

BRODER, A. Z., CHARIKAR, M., FRIEZE, A. M., AND MITZENMACHER, M. 2000. Min-wise independent permutations. *J. Comput. Syst. Sci. 60*, 3, 630–659.

BUCHSBAUM, A. L., CALDWELL, D. F., CHURCH, K. W., FOWLER, G. S., AND MUTHUKRISHNAN, S. 2000. Engineering the compression of massive tables: An experimental approach. In *Proceedings of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'00)*. ACM, New York, 175–184.

BUCHSBAUM, A. L., FOWLER, G. S., AND GIANCARLO, R. 2003. Improving table compression with combinatorial optimization. *J. ACM 50*, 6, 825–851.

BUNEMAN, P., KHANNA, S., TAJIMA, K., AND TAN, W. C. 2002. Archiving scientific data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. ACM, New York.

BURNS, R. 1996. BURNS, R. 1996. Differential compression: A generalized solution for binary files. M.S. thesis, University of California, Santa Cruz.

BURNS, R. AND LONG, D. D. E. 1997a. Efficient distributed back-up with delta compression. In *Proceedings of the IO Conference on Parallel and Distributed Systems (IOPADS'97)*. ACM, New York, 27–36.

BURNS, R. AND LONG, D. D. E. 1997b. A linear time, constant space differencing algorithm. In *Proceedings of the 16th IEEE International Performance, Computing and Communications Conference (IPCCC'97)*. IEEE, Los Alamitos, CA, 429–436.

BURNS, R. AND LONG, D. D. E. 1998. In-place reconstruction of delta compressed files. In *Proceedings of the 17th ACM Symposium on Principles of Distributed Computing (PODC'98)*. ACM, New York, 267–275.

BURNS, R., STOCKMEYER, L., AND LONG, D. D. E. 2002. Experimentally evaluating in-place delta reconstruction. In *Proceedings of the 19th IEEE Symposium on Mass Storage Systems and Technologies*. IEEE, Los Alamitos, CA.

BURROWS, M. AND WHEELER, D. J. 1994. A block-sorting lossless data compression algorithm. Tech. rep. 124, Digital Systems Research Center.

CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. E. 2006. Bigtable: A distributed storage system for structured data. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, Berkeley, CA, 205–215.

CHARIKAR, M. S. 2002. Similarity estimation techniques from rounding algorithms. In *Proceedings of the 34th Annual ACM Symposium on Theory of Computing (STOC'02)*. ACM, New York, 380–388.

CHEN, Y., EDLER, J., GOLDBERG, A., GOTTLIEB, A., SOBTI, S., AND YIANILOS, P. 1999. A prototype implementation of archival intermemory. In *Proceedings of the 4th ACM International Conference on Digital Libraries (DL'99)*. ACM, New York, 28–37.

CRESPO, A. AND GARCIA-MOLINA, H. 1998. Archival storage for digital libraries. In *Proceedings of the 3rd ACM International Conference on Digital Libraries (DL'98)*. ACM, New York, 69–78.

DEAN, J. AND HENZIGER, M. R. 1999. Finding related pages in the World Wide Web. In *Proceedings of the 8th International World Wide Web Conference*.

DOUGLIS, F. AND IYENGAR, A. 2003. Application-specific delta-encoding via resemblance detection. In *Proceedings of the USENIX Annual Technical Conference*. USENIX Association, Berkeley, CA.

DUBNICKI, C., GRYZ, L., HELDT, L., KACZMARCZYK, M., KILIAN, W., STRZELCZAK, P., SZCZEPKOWSKI, J., UNGUREANU, C., AND WELNICKI, M. 2009. HYDRAstor: A scalable secondary storage. In *Proceedings of the 7th USENIX Conference on File and Storage Technologies (FAST)*. USENIX Association, Berkeley, CA, 197–210.

EMC CORPORATION. 2002. EMC Centera: Content addressed storage system, data sheet. http://www.emc.com/pdf/products/centera/centera_ds.pdf.

FETTERLY, D., MANASSE, M., NAJORK, M., AND WIENER, J. 2003. A large-scale study of the evolution of web pages. In *Proceedings of the 12th International World Wide Web Conference*. ACM, New York, 669–678.

FREE SOFTWARE FOUNDATION. 2000. The gzip data compression program. http://www.gnu.org/software/gzip/gzip.html.

GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. 2003. The Google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03)*. ACM, New York.

GIBSON, T. J. 1998. Long-term UNIX file system activity and the efficacy of automatic file migration. Ph.D. dissertation, University of Maryland, Baltimore.

GOLDBERG, A. V. AND YIANILOS, P. N. 1998. Towards an archival intermemory. In *Proceedings of the IEEE Advances in Digital Libraries (ADL'98)*. IEEE, Los Alamitos, CA, 147–156.

GRAY, J. AND SHENOY, P. 2000. Rules of thumb in data engineering. In *Proceedings of the 16th International Conference on Data Engineering (ICDE'00)*. IEEE, Los Alamitos, CA, 3–12.

GRAY, J., CHONG, W., BARCLAY, T., SZALAY, A., AND VANDENBERG, J. 2002. TeraScale SneakerNet: Using inexpensive disks for backup, archiving, and data exchange. Tech. rep. MS-TR-02-54, Microsoft Research, Redmond, WA.

HENSON, V. 2003. An analysis of compare-by-hash. In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems (HotOS-IX)*. USENIX Association, Berkeley, CA.

HENZINGER, M. 2006. Finding near-duplicate web pages: A large-scale evaluation of algorithms. In *Proceedings of the 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR'06)*. ACM, New York, 284–291.

HITACHI GLOBAL STORAGE TECHNOLOGIES. 2004. Hitachi hard disk drive specification: Deskstar 7K400 3.5-inch Ultra ATA/133 and 3.5-inch Serial ATA hard disk drives, ver. 1.4.

HITZ, D., LAU, J., AND MALCOM, M. 1994. File system design for an NFS file server appliance. In *Proceedings of the Winter USENIX Technical Conference*. USENIX Association, Berkeley, CA, 235–246.

HOLLINGSWORTH, J. AND MILLER, E. L. 1997. Using content-derived names for configuration management. In *Proceedings of the Symposium on Software Reusability (SSR'97)*, 104–109.

HONG, B., PLANTENBERG, D., LONG, D. D. E., AND SIVAN-ZIMET, M. 2004. Duplicate data elimination in a SAN file system. In *Proceedings of the 21st IEEE/12th NASA Goddard Conference on Mass Storage Systems and Technologies*. IEEE, Los Alamitos, CA.

HUNT, J. W. AND MCILROY, M. D. 1976. An algorithm for differential file comparison. Tech. rep. CSTR 41, Bell Laboratories, Murray Hill, NJ.

IBM. 1999. IBM OEM hard disk drive specification for DPTA-3xxxxx 37.5 GB13.6 GB 3.5-inch hard disk drive with ATA interface, revision (2.1).Deskstar 34GXP and 37GP hard disk drives.

IBM. 2005. IBM Tivoli Storage Manager. http://www.tivoli.com/products/solutions/storage/.

INDYK, P. AND MOTWANI, R. 1998. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *Proceedings of the 30th Annual ACM Symposium on Theory of Computing (STOC'98)*. ACM, New York, 604–613.

JAIN, A. K., MURTY, M. N., AND FLYNN, P. J. 1999. Data clustering: A review. *ACM Comput. Surv. 31*, 3, 264–323.

JAIN, N., DAHLIN, M., AND TEWARI, R. 2005. TAPER: Tiered approach for eliminating redundancy in replica synchronization. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*. USENIX Association, Berkeley, CA, 281–294.

KOHL, J. T., STAELIN, C., AND STONEBRAKER, M. 1993. HighLight: Using a log-structured file system for tertiary storage management. In *Proceedings of the Winter USENIX Technical Conference*. USENIX Association, Berkeley, CA, 435–447.

KOLIVAS, C. 2010. lrzip v0.46. http://ck.kolivas.org/apps/lrzip/README.

KOLLER, R. AND RANGASWAMI, R. 2010. I/O deduplication: Utilizing content similarity to improve I/O performance. *ACM Trans. Storage 6*, 3, 1–26.

KORN, D. G. AND VO, K.-P. 2002. Engineering a differencing and compression data format. In *Proceedings of the USENIX Annual Technical Conference*. USENIX Association, Berkeley, CA, 219–228.

KORN, D. G., MACDONALD, J., MOGUL, J., AND VO, K. 2002. The VCDIFF generic differencing and compression data format. Request For Comments (RFC) 3284, IETF.

KUBIATOWICZ, J., BINDEL, D., CHEN, Y., EATON, P., GEELS, D., GUMMADI, R., RHEA, S., WEATHERSPOON, H., WEIMER, W., WELLS, C., AND ZHAO, B. 2000. 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)*. ACM, New York.

KULKARNI, P., DOUGLIS, F., LAVOIE, J., AND TRACEY, J. M. 2004. Redundancy elimination within large collections of files. In *Proceedings of the USENIX Annual Technical Conference*. USENIX Association, Berkeley, CA, 59–72.

LEE, E. K. AND THEKKATH, C. A. 1996. Petal: Distributed virtual disks. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, New York, 84–92.

LEWIS, B. 2008. Deduplication comes of age. http://www.netapp.com/us/communities/tech-ontap/dedupe-0708.html.

LIEFKE, H. AND SUCIU, D. 2000. XMill: An efficient compressor for XML data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. ACM, New York, 153–164.

LILLIBRIDGE, M., ESHGHI, K., BHAGWAT, D., DEOLALIKAR, V., TREZISE, G., AND CAMBLE, P. 2009. Sparse indexing: Large scale, inline deduplication using sampling and locality. In *Proceedings of the 7th USENIX Conference on File and Storage Technologies (FAST)*. USENIX Association, Berkeley, CA, 111–123.

LITWIN, W. AND NEIMAT, M.-A. 1996. High-availability LH* schemes with mirroring. In *Proceedings of the Conference on Cooperative Information Systems*. 196–205.

LITWIN, W., NEIMAT, M.-A., AND SCHNEIDER, D. A. 1993. LH*—Linear hashing for distributed files. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. ACM, New York, 327–336.

LITWIN, W., NEIMAT, M.-A., AND SCHNEIDER, D. A. 1996. LH*—A scalable, distributed data structure. *ACM Trans. Datab. Syst. 21*, 4, 480–525.

LONG, D. D. E. 2002. A scalable on-line associative deep store. NSF Grant Proposal, Award Number 0310888.

LORIE, R. A. 2001. A project on preservation of digital data. http://www.rlg.org/preserv/diginews/diginews5-3.html. Vol. 5, No. 3.

LORIE, R. A. 2004. Long-term archival of digital information. Storage Systems Research Center Seminar, University of California, Santa Cruz.

LYMAN, P., VARIAN, H. R., SEARINGEN, K., CHARLES, P., GOOD, N., JORDAN, L. L., AND PAL, J. 2003. How much information? 2003. http://www.sims.berkeley.edu/research/projects/how-much-info-2003/.

MACDONALD, J. P. 2000. File system support for delta compression. M.S. thesis, University of California, Berkeley.

MAHALINGAM, M., TANG, C., AND XU, Z. 2002. Towards a semantic, deep archival file system. Tech. rep. HPL-2002- 199, HP Laboratories, Palo Alto, CA.

MANASSE, M. 2003. Finding similar things quickly in large collections. http://research.microsoft.com/research/sv/PageTurner/similarity.htm.

MANBER, U. 1993. Finding similar files in a large file system. Tech. rep. TR 93-33, Department of Computer Science, The University of Arizona, Tucson, AZ.

MANKU, G. S., JAIN, A., AND DAS SARMA, A. 2007. Detecting near-duplicates for web crawling. In *Proceedings of the 16th International Conference on World Wide Web (WWW'07)*. ACM, New York, 141–150.

MOGUL, J., DOUGLIS, F., FELDMANN, A., AND KRISHNAMURTHY, B. 1997. Potential benefits of delta-encoding and data compression for HTTP. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM'97)*. ACM, New York.

MUTHITACHAROEN, A., CHEN, B., AND MAZIERES, D. 2001. A low-bandwidth network file system. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP'01)*, ACM, New York, 174–187.

NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY. 2008. FIPS PUB 180-3: Secure Hash Standard (SHS). Gaithersburg, MD 20899-8900.

NELSON, M. AND GAILLY, J.-L. 1996. *The Data Compression Book* 2nd Ed. M&T Books, New York.

ORACLE BERKELEY DB. 2010. Berkeley DB Database. http://www.oracle.com/database/berkeley-db/index.html.

OTOO, E. J. 1986. Balanced multidimensional extendible hash tree. In *Proceedings of the 5th ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*. ACM, New York, 100–113.

OUKSEL, M. AND SCHEUERMANN, P. 1983. Storage mappings for multidimensional linear dynamic hashing. In *Proceedings of the 2nd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*. ACM, New York, 90–105.

OUYANG, Z., MEMON, N., SUEL, T., AND TRENDAFILOV, D. 2002. Cluster-based delta compression of a collection of files. In *Proceedings of the International Conference on Web Information Systems Engineering (WISE'02)*. IEEE, Los Alamitos, CA, 257–266.

QUINLAN, S. AND DORWARD, S. 2002. Venti: A new approach to archival storage. In *Proceedings of the Conference on File and Storage Technologies (FAST)*. USENIX Association, Berkeley, CA, 89–101.

RABIN, M. O. 1981. Fingerprinting by random polynomials. Tech. rep. TR-15-81, Center for Research in Computing Technology, Harvard University.

RAJASEKAR, A. AND MOORE, R. 2001. Data and metadata collections for scientific applications. In *Proceedings of the 9th International Conference on High-Performance Computing and Networking (HPCN Europe'01)*. Springer, Berlin, 72–80.

RIGGLE, C. M. AND MCCARTHY, S. G. 1998. Design of error correction systems for disk drives. *IEEE Transactions on Magnetics 34*, 4, 2362–2371.

RIVEST, R. 1992. The MD5 message-digest algorithm. Request for comments (RFC) 1321, IETF.

ROCHKIND, M. J. 1975. The source code control system. *IEEE Transactions on Software Engineering SE-1, 4*, 364–370.

ROSENBLUM, M. AND OUSTERHOUT, J. K. 1992. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst. 10*, 1, 26–52.

ROTHENBERG, J. 1995. Ensuring the longevity of digital documents. *Sci. American 272*, 1, 42–47.

SAITO, Y. AND KARAMANOLIS, C. 2002. Pangaea: A symbiotic wide-area file system. In *Proceedings of the ACM SIGOPS European Workshop.* ACM Press.

SAITO, Y., KARAMANOLIS, C., KARLSSON, M., AND MAHALINGAM, M. 2002. Taming aggressive replication in the Pangaea wide-area file system. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, Berkeley, CA.

SAND TECHNOLOGY. 2009. SAND/DNA. http://www.sand.com/dna/compress/index.html/.

SANTRY, D. S., FEELEY, M. J., HUTCHINSON, N. C., VEITCH, A. C., CARTON, R. W., AND OFIR, J. 1999. Deciding when to forget in the Elephant file system. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP'99)*. 110–123.

SAYOOD, K., Ed. 2003. *Lossless Compression Handbook*. Academic Press.

SCHMUCK, F. AND HASKIN, R. 2002. GPFS: A shared-disk file system for large computing clusters. In *Proceedings of the Conference on File and Storage Technologies (FAST)*. USENIX Association, 231–244.

SECURITY INNOVATION, I. 2006. Regulatory compliance demystified: An introduction to compliance for developers. http://msdn.microsoft.com/en-us/library/aa480484.aspx.

SELTZER, M. I. AND YIGIT, O. 1991. A new hashing package for UNIX. In *Proceedings of the Winter USENIX Technical Conference.* USENIX Association, Berkeley, CA, 173–184.

SEWARD, J. 2002. http://sources.redhat.com/bzip2/.

TANENBAUM, A. S., HERDER, J. N., AND BOS, H. 2006. File size distribution on UNIX systems: Then and now. *ACM SIGOPS Operating Systems Review 40*, 1, 100–104.

TICHY, W. F. 1985. RCS—A system for version control. *Softw. Pract. Exper. 15*, 7, 637–654.

TRENDAFILOV, D., MEMON, N., AND SUEL, T. 2002. Zdelta: An efficient delta compression tool. Tech. rep. TR-CIS- 2002-02, Polytechnic University.

TRENDAFILOV, D., MEMON, N., AND SUEL, T. 2004. Compression file collections with a TSP-based approach. Tech. rep. TR-CIS-2004-02, Polytechnic University.

TRIDGELL, A. 1999. Efficient algorithms for sorting and synchronization. Ph.D. thesis, Australian National University.

UNGUREANU, C., ATKIN, B., ARANYA, A., GOKHALE, S., RAGO, S., CAŁKOWSKI, G., DUBNICKI, C., AND BOHRA, A. 2010. HydraFS: A high-throughput file system for the HYDRAstor content-addressable storage system. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies (FAST)*. USENIX Association, Berkeley, CA, 17–17.

VO, B. D. AND MANKU, G. S. 2007. RadixZip: Linear time compression of token streams. In *Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB)*. VLDB Endowment, 1162–1172.

VO, K.-P. 2007. Vcodex: A data compression platform. In *Proceedings of the International Conference on Software and Data Technologies (ICSOFT'07)*. Springer, Berlin, 201–212.

WIEDERHOLD, G. 1983. *Database Design* 2nd Ed. McGraw-Hill, New York.

WITTEN, I. H., MOFFAT, A., AND BELL, T. C. 1999. *Managing Gigabytes: Compressing and Indexing Documents and Images* 2nd Ed. Morgan Kaufmann.

YANG, T., JIANG, H., FENG, D., AND NIU, Z. 2009. DEBAR: A scalable high-performance de-duplication storage system for backup and archiving. Tech. rep. TR-UNL-CSE-2009-0004, University of Nebraska-Lincoln.

YOU, L. L. 2006. Efficient archival data storage. Ph.D. dissertation, University of California, Santa Cruz.

YOU, L. L. AND KARAMANOLIS, C. 2004. Evaluation of efficient archival storage techniques. In *Proceedings of the 21st IEEE/12th NASA Goddard Conference on Mass Storage Systems and Technologies*. IEEE, Los Alamitos, CA, 227–232.

YOU, L. L., POLLACK, K. T., AND LONG, D. D. E. 2005. Deep Store: An archival storage system architecture. In *Proceedings of the 21st International Conference on Data Engineering (ICDE'05)*. IEEE, Los Alamitos, CA.

ZADOK, E., OSBORN, J., SHATER, A., WRIGHT, C. P., MUNISWAMY-REDDY, K.-K., AND NIEH, J. 2003. Reducing storage management costs via informed user-based policies. Tech. rep. FSL-03-01, Computer Science Department, SUNY, Stony Brook. http://www.ncl.cs.columbia.edu/publications/sunysb-fsl-03-01.pdf.

ZHU, B., LI, K., AND PATTERSON, H. 2008. Avoiding the disk bottleneck in the data domain deduplication file system. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST'08)*. USENIX Association, Berkeley, CA, 18:1–18:14.

ZIV, J. AND LEMPEL, A. 1977. A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theory IT-23*, 3, 337–343.

Received February 2010; revised November 2010; accepted January 2011