Deduplication: Concepts and Techniques

William Jannen
jannen@cs.williams.edu

April 19, 2020

1 Introduction

This document surveys the deduplication design space. For each design element, the document presents the high-level idea and uses representative systems as case studies. If there are parts of the text that are unclear, please let me know so that I can improve them. I am not aware of any good surveys/introductory texts on deduplication, so this is my attempt to create one for us to use in this class.

Learning objectives:

Definition. At a high level, what are the core processes that deduplication systems perform?

Measurement. How do you quantify the effectiveness of a deduplication system so that we can compare it against other designs?

Timing. When do you perform deduplcation? What is the difference between an inline and an offline deduplication system? How do they trade performance and space consumption?

Data structures. What is a chunk? What is a fingerpint? What is the fingerprint index, and how is the fingerprint index used?

Parameters. What knobs can a system design tune, and how do they affect performance, deduplication ratio, metadata overhead, fragmentation, etc.?

Use cases. In what environments should deduplication techniques be employed? What characteristics of an environment present promising opportunities or difficult challenges for a deduplication system to manage?

2 Deduplication

Data deduplication is a form of compression. The name itself, *deduplication*, hints at a deduplication system's overarching goal: to **identify duplicate data**, and **eliminate repeated instances** of that data. Successfully identifying and eliminating redundant data can drastically decrease the storage requirements for systems that deal with large volumes of data. This is especially true for systems which maintain multiple copies of identical files—such as a system that performs nightly backups—or for systems that store many files with large, overlapping regions—such as boilerplate or configuration files that are part of an OS distribution. Even when storage capacity is not an issue, there are other reasons that deduplication may be valuable; for example, when sending data from a client to a server, a client that knows that the server already has a copy of the data it wishes to send can avoid transfering unnecessary data and avoid network overheads.

Completely eliminating *all* duplicate data is an admirable goal, but a system design may tolerate some replication for a variety of reasons. For example, allowing some redundant data may be necessary in order to complete important tasks in a reasonable amount of time (identifying all duplicates may be expensive!), or tolerate system failures (if we lose the only copy of a file system superblock, our whole system is hosed). Thus, we would like some measure that describes a deduplication system's effective "data savings". For that, we turn to the **deduplication ratio**.

Concretely, if B bytes of data are presented to a storage system S, which unambiguously represents the data using D bytes, then the deduplication ratio of S is defined to be:

$$D_S = \frac{B}{D} \tag{1}$$

A more efficient deduplication system will have a higher deduplication ratio (i.e., it stores more data using fewer bytes). However, D_S is not the only measure of interest. Latency, throughput, scalability, and resiliency are all system design concerns, and the right balance of performance and deduplication efficiency is going to be context dependent.

2.1 Deduplication in a nutshell

Deduplication system designs differ along many dimensions, but at a high level, they all follow the same basic formula. The system receives a stream of data, and it breaks that stream into discrete parts called **chunks**. For each chunk, the system calculates a collision-resistant hash (e.g., SHA-1, MD5). This hash value is commonly

referred to as the chunk's **fingerprint** because the fingerprint uniquely identifies the chunk's contents: two chunks that differ by even a single byte should have unrelated fingerprints. The set of unique chunk hashes (fingerprints) and the locations of the data that those fingerprints describe (chunk pointers), are stored in the system's **fingerprint index**.

After chunking, fingerprinting, and indexing, the deduplication process becomes straightforward: when the system receives a ⟨`fingerprint, chunk pointer`⟩ pair, the system stores the chunk on disk if the fingerprint is absent from the index, and inserts the fingerprint into the index for future data sharing. Otherwise, the system stores a logical pointer to the original chunk, and discards the redundant chunk data.

Restoring deduplicated data is also straightforward: to reassemble the orignal data stream, the stream's logical chunks must be located and combined by looking up the chunk fingerprints in the fingerprint index.

2.2 Design Choices

There are three primary design decisions within the basic deduplication framework:

- when to perform deduplication,
- where to perform deduplication, and
- what to deduplicate

The first choice in the design of any deduplication system is *when* to perform deduplication. The two classes of deduplication systems are **inline** and **post-process** deduplication [12].

Inline. In an inline deduplication system, the data stream is chunked and hashed before it is written to disk [10]. The main advantage of the inline approach is that duplicate chunks are never written, saving I/O. However, inline deduplication introduces chunking, hashing, and index queries—expensive computations—onto the critical path of each potential write. As a result, inline deduplication systems generally suffer increased latency.

Offline. Post-process (or offline) deduplication eliminates data redundancy *after* the data has been completely committed to disk. This deferred processing keeps file system responsiveness high, since duplicate elimination is done as a background process—possibly on a remote system. Unfortunately, duplicate data consumes temporary space during the period of time between the initial write and the duplicate detection/elimination.

A second choice for deduplication systems is *what* to deduplicate. Deduplication can be performed at the granularity of entire files [4], fixed-size blocks [20, 28], or variable-sized chunks [5, 18]. Hybrid schemes are also possible, where the chunking strategy is flexible. Hybrid schemes often leverage additional information, such as file type, to form heuristics and make decisions on a case-by-case basis about which chunking scheme is most appropriate [17].

Whole file. Whole file deduplication eliminates the runtime "chunking" overheads. (More details about chunking are discussed later.) The downside of this coarse file-level granularity is that there are fewer sharing opportunities. For example, consider two large files which are identical up to the last byte; at file granularity, no data is shared whatsoever. File granularity is inappropriate for deduplicating boilerplate code, configuration files, and iterative file modifications, as no amount of data will be shared despite large regions of commonality.

Yet in some domains whole file granularity is preferable. Compressed files and media files exhibit the property that even small modifications to the underlying data completely transform the final representation. As a consequence, either the whole file is identical or none of it is [17].

Variable-sized chunking. Variable sized chunking lies at the other extreme: data is divided into potentially many, potentially small chunks in an attempt to maximize the sharing opportunities. The locations of chunk boundaries are determined by a computation over the data itself, imparting the name **content defined chunking**.

Content defined chunking often yields the highest deduplication ratios. There are many small segments, so there are many opportunities to share data among the logical objects in the system. Content defined chunking also gracefully handles the case that data is inserted into a stream; only the new data's containing segment and potentially its successor must be re-chunked and re-hashed in the common case [18]. This is discussed later in the LBFS case study.

Despite the higher deduplication ratios, content defined chunking introduces several costs into the end-to-end task of deduplication. First, it takes time to identify chunk boundaries. Rabin fingerprints [21], as used in the two-thresholds two divisors [9] and INC-K [17] chunking algorithms, are computed over a sliding window, so the process incurs many computations even at small average chunk sizes.

The additional costs of storing and querying the fingerprint index are perhaps more subtle. The more chunks there are in the index, the larger the indexing data structure becomes; it is not long until the index exceeds the size of main memory and must spill onto disk. Index queries have almost no locality of reference, which compounds the problem; a collision resistant hash is necessarily independently and uniformly distributed, so data that has

locality in the workload does not have locality in the index. As index sizes grow, each look-up requires one or more disk seeks, introducing the chunk index **disk bottleneck** [29].

Fixed-size chunking. Fixed block sizes present a compromise between file level granularity and variable sized chunking. Chunk boundaries are set at predetermined offsets in the stream instead of at boundaries determined by some computation over the data (often fixed-size chunks are set at 4KiB to match common page/block sizes). Fixed offsets provide fast, computation-free chunking and achieve a finer granularity than the entire file. System rules governing disk layout are much simpler, and an appropriate block size manage internal fragmentation and index size. Unfortunately, data insertion is difficult to handle—the remainder of a file must be rehashed, creating many unreachable blocks as data evolves quickly. Again, data insertion is discussed later, and we will see why variable sized chunking performs well in this scenario.

A third design decision is *where* to perform deduplication. **Source deduplication** describes systems where an index is maintained or queried locally, and data is examined for duplicates before being sent to a remote server for storage. Data transfer is minimized because duplicate data is never transmitted across a network in source deduplication. **Destination deduplication** systems remove the deduplication process from the client; all deduplication is done remotely. Destination deduplication minimizes client computation at the cost of network bandwidth. In some systems [27], a lightweight index is used to check for existence at the source, but location data is maintained at the destination. The choice of source or destination deduplication determines where resource consumption will take place. Location may also be determined by system constraints; general purpose workstations may not be well-equipped to meet deduplication requirements.

For the remainder of this section, we discuss some target workloads and the properties of those workloads that lend themselves to different design decisions. The top of Table 1 provides a classification of the systems discussed in this section. Systems below the double line were omitted for brevity.

2.3 Archival

Perhaps the largest demand for data deduplication arises from enterprise backups. Companies are required to maintain diligent and comprehensive records of old data, for many years after its inception, in order to comply with government regulations. The characteristics of such data backups depart from single user workstation use in several ways:

- a backup operation is a large streaming write, with no random accesses or reads
- a backup must complete in its entirety in a given window
- once written, the data will never change (it is immutable)
- data may be read in the future, but rarely (cold storage)
- data must always be accessible in its original form
- the aggregate volume of data will only ever grow — capacity should be incrementally scalable

These fixed-content workloads often prioritize deduplication ratio, favor throughput over latency, require resilience to data loss, and must scale extraordinarily well.

2.3.1 Case Study: Venti

Venti [20] is an early implementation of a content addressable archival repository, sharable amongst multiple clients. Venti implements a **write once** policy — once written, data cannot be modified or deleted by a user or administrator. The decision to archive data is permanent.

Venti writes data to an append-only log, divided into fixed-size, logical containers termed *arenas*. Data blocks may be variable sized, but the append only nature of arenas prevents fragmentation. Each data block is stored with an associated header. The header lists, among other information, whether or not the data was compressed and with what algorithm. A list of the headers for all data the arena contains is replicated at the end of each arena. When an arena is filled, it is sealed — never to be modified again.

An on-disk hash table is maintained separately from the log. Hash buckets, which are used to resolve collisions, occupy an entire disk block, with any excess fingerprints written to subsequent blocks. Thus, every index query requires at least one disk seek, but often only one such seek.

Separating the index from the block store allows Venti to maintain disparate storage policies. The block store is kept on a RAID, providing fault tolerance through parity.

Venti is not a full backup solution — it is merely a back-end block store over which a complete system may be constructed. Mappings from files to their constituent blocks must be maintained externally. We observe this pattern in many CAS systems. A storage backend is optimized for expected access patterns, and overlaid with client file system structures.

2.3.2 Case Study: Deep Store

Deep store is an archival storage system for fixed-content (immutable) reference data (write once, read many) [28]. Deep Store's goals are similar to those of Venti, but Deep

System	Deduplication Location	Chunking	Index	Keywords
Venti [20]	destination	N/A (variable)	on-disk hash table	write-once policy, append-only arena, data compression
Deep Store [28]	*destination*	variable	local hash structure, distributed hash table	rich metadata, delta encoding, compression
Hydrastor [7]	destination	variable (Rabin)	distributed hash table	resiliency classes, erasure codes, continuous operation
LBFS [18]	hybrid	variable (Rabin)	legacy DB	content defined chunking, modified NFS, resource trade-off
Data Domain [29]	destination	variable (Rabin)	tiered: Bloom filter, locality preserving cache, legacy DB	locality-preserving cache, stream informed segment layout, disk bottleneck
Sparse Indexing [14]	destination (hybrid)	variable (two thresholds, two divisors)	in-memory, sampled	sampling, sparse index, chunk locality
PRUNE [17]	destination	variable (INC-K)	partitioned index (tablets)	INC-K, tablet, partitioned index
HydraFS [24]	destination	variable (Rabin)	distributed hash table	familiar file system API over Hydrastor backend
SIS (Windows 2000) [4]	source	N/A (whole-file)	database	file links with copy semantics, copy-on-close

Table 1: A categorization of example deduplication file systems.

Store makes two observations. First, data will be stored over an indefinite time horizon; data must therefore be searchable and comprehensible beyond the lifetimes of those who generated it. Second, the effectiveness of compression is dependent upon the composition of the data; the system should be able to select the compression technique that is best for the data at hand. Deep Store stores versioned XML metadata, and supports chunk-and-hash CAS as well as delta compression to accommodate these requirements.

The Deep Store index is called the *virtual object table*, a hash structure storing location information as a {16-bit *group* number, 16-bit *megablock* number, and 32-bit *offset*}. To locate a file, a distributed hash table maps the *group* number to a storage node, where the *megablock* number and *offset* address the virtual block on disk. It is called a virtual block because it can be one of three types:

K block contains actual block data

Σ block contains a list of handles that, when themselves resolved, are concatenated to form the final file

Δ block contains the handle list of one or more reference files filed by a delta file

Files longer than a megablock (4GB) are split amongst multiple megablocks and stored as Σ blocks (concatenations).

Deep Store metadata is intended to be rich: both searchable and versioned. However, such rich metadata conflicts with Deep Store's emphasis on compression. The Deep Store solution is to store metadata in a representation appropriate for its use. Metadata is maintained as versioned XML, but storage is as follows:

system metadata is needed on all file operations and is stored in an efficient lookup structure

search metadata is kept in an XML database

archival metadata is stored alongside regular data — it is both indexed and losslessly compressed

Deep Store is the prototype of a complete backup solution, but it leaves several questions open further exploration. One question is how to make a system resilient to data loss. The authors identify the ideological contradiction of adding redundancy to a system designed to eliminate redundancy. Yet if a CAS block shared by many files is lost, or the base file of a chain of delta versions is corrupted, entire subtrees of files dependent on that data are ruined. Deep Store does not solve this problem, but the authors conclude that the degree of replication should be proportional to the number of files dependent on a data instance.

Deep Store also notes that content analysis is both CPU and I/O intensive, but not necessarily at the same time. Deep Store leaves an analysis of these trade-offs, data structure optimization, and pipelining as future work.

It is also worth discussing what the authors termed the CPU-I/O gap. The historical growth rates in the performance of silicon based components (processors/memory) have always outstripped the performance gains in capacity and latency of magnetic media (disks). An implication of this gap is that any system designed to leverage computations that consume memory and CPU resources in order to reduce traffic to and from disk, will get progressively faster, assuming technology trends continue. Archival systems are designed to operate in the very long run, so this trade-off seems important.

2.3.3 Case Study: Hydrastor

Hydrastor is another archival system which utilizes CAS for data deduplication [7]. Hydrastor's primary deployment target is the data center. As such, Hydrastor's goals are continuous operation — in the face of hardware failures as well as upgrades; reliability; integrity; availability; and configurability. We focus on Hydrastor's methods for providing of reliability in this subsection.

Hydrastor provides the abstraction of an infinite store of content-addressed, immutable, variable-sized blocks. Each block is stored at a user-defined **resiliency level**, which determines the number of concurrent node failures the block can survive. Resiliency is provided at the block level via erasure codes [6].

Erasure codes break a chunk of data into a set of fragments in such a way that the original chunk can be reconstructed from *any* appropriately sized subset of those fragments. Formally, a message of m symbols is transformed into a code word of $n > m$ symbols. The message can be recovered from any set of $m' \leq n$ symbols, as long as $m' \geq k$. In Hydrastor, n is set to the supernode cardinality. The resiliency level dictates the choice of k.

Heterogeneous resiliency levels introduce an additional requirement for block writes. An incoming block determined to be a duplicate can only be discarded if the previously existing copy of that block has an equivalent or higher resiliency level. The system must also handle the case where blocks are in the process of being recovered after failure or migration to a new peer node. Hydrastor manages this case by storing the additional block whenever Hydrastor cannot conclusively determine the status of previous block copies.

When a failure occurs, the resiliency levels of all still-readable blocks are automatically scheduled for rebuilding. This requires block reassembly, recoding, and rewriting of fragments to an entirely new syncrun.

An evolution of this project is HydraFS, which builds a familiar file system API over a Hydrastor back-end [24].

2.4 Minimizing data transfer

Persistent storage is not always the most constrained system resource. This subsection studies the application of CAS and deduplication techniques to data written onto the network, as opposed to a rotating disk. The following systems trade CPU and memory for bandwidth savings.

2.4.1 Case Study: LBFS

The low bandwidth file system (LBFS) [18] guarantees close-to-open consistency. LBFS also minimizes network transmission at all costs, within the consistency model. LBFS prioritizes deduplication ratio and relies heavily on client-side caching in order to achieve these goals.

LBFS introduces variable-sized chunking to maximize its deduplication ratio. The **sliding window method** (SW), also shown in Algorithm 1, has two parameters: window size w, and target pattern size t. SW computes a fingerprint over all overlapping w-width byte ranges until observing that the t low-order bits of the fingerprint are equal to 0. SW defines a *break point* at the last byte in w. SW then shifts the window right by w bytes, and repeats the process until reaching the end of the file.

Algorithm 1 - Sliding window method

```
1:  param NUMERIC w,t
2:  param FILE f
3:  INT i ← 0
4:  while (i+w) ≤ |f| do
5:      t ← FINGERPRINT(f[i,...,i+w])
6:      if t = 0 then
7:          define chunk boundary at f[i+w]
8:          i ← i+w
9:      else
10:         i ← i+1
11:     end if
12: end while
```

A negative binomial distribution allowing r failures has a mean of $(pr)/(1-p)$, so the sliding window method yields an expected chunk size of $2^t - 1 + w$. LBFS selects $t = 13, w = 48$, for an expected chunk size of $\approx$ 8K. Thus, the calculation of a single chunk boundary requires $\approx$ 8K individual fingerprint calculations.

Fortunately, Rabin fingerprints [21] are a reasonably efficient choice for chunking calculations because their calculation is incremental. Large parts of a Rabin fingerprint computation for one window can be reused in the computation for the next overlapping window. Note, however, that Rabin fingerprinting is used to identify chunk boundaries, not to uniquely identify block contents. CAS requires that each chunk be separately fingerprinted with a cryptographically strong hash, such as `SHA-1`, in order to prevent collisions.

Variable sized chunking presents an elegant solution to the problem of data insertion. Recall that fixed-size chunks are defined by byte offsets within a file. A file f', produced by adding a single byte at the front of the file f, will have no fixed-size blocks in common with f. This is called the boundary-shifting problem. The entire file must be retransmitted over the network, despite the fact that it exists in its entirety at the server. Variable sized chunking requires that as little as one block be sent.

After an insertion, the sliding window method can terminate as soon as a previously chunked block is identified. Consider the three cases shown in Figure 1 where (1) data is inserted into the middle of a chunk, (2) data is inserted that includes or produces a new chunk boundary, and (3) data is inserted into a window that contains a chunk boundary. In (1), only the single block containing the new data must be re-chunked. In both (2) and (3), the containing block up to any new boundary is fingerprinted, followed by the successor, until an existing chunk boundary is encountered. LBFS only transmits the new chunks across the network.

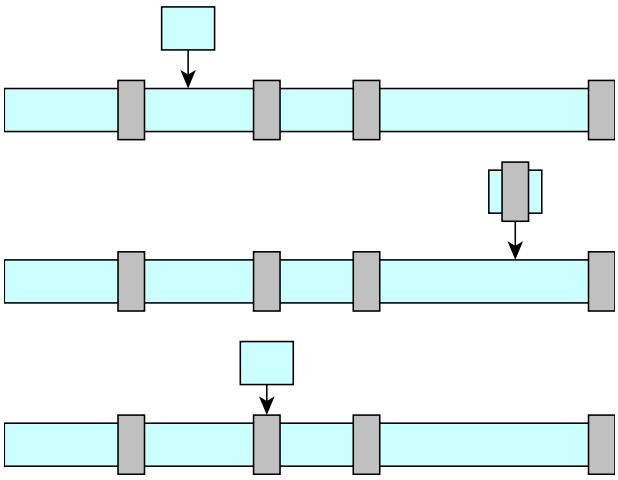

Figure 1: Three cases of a small data insertion into an existing file with variable sized chunks. Windows that represent chunk boundaries are shown in grey. In the first case, the insertion does not introduce any new chunks. A new fingerprint must be taken for the containing chunk, but no boundaries shift. In the second case, the insertion causes an existing chunk to be divided into two. The final example eliminates an existing chunk boundary, causing two previously existing chunks to be merged. LBFS only transmits new chunks across the network, reducing bandwidth.

Two drawbacks to variable sized chunking are an increased computational burden and a vulnerability to certain inputs. To one extreme, some inputs contain a chunk boundary every w bytes. It would be less expensive to transmit raw data than to transmit messages that iden-

tify duplicate blocks. On the other extreme, some data streams produce no internal chunk boundaries. Notably, a sequence of 0's has this property. LBFS defines a minimum chunk size of 2K and a maximum of 64K to handle these degenerate cases. The sliding window algorithm starts at an offset of 2K instead of 0, and after 64K, an artificial boundary is inserted.

Variable sized chunking refines the granularity of duplicate detection, and client-side caching and RPC pipelining further reduce the latency and bandwidth consumption of LBFS. In what follows, we describe the LBFS protocol and consistency model, highlighting the use of duplicate detection to minimize data transfer.

The LBFS protocol is a modified NFSv3, with extensions to leverage commonalities amongst files and reduce network traffic. Specifically, LBFS adds *read leases* and the following RPC calls:

GETHASH(`nfs_handle, offset, len`) :
returns the hash of each file chunk in the range as a list of $\{\langle \texttt{SHA-1}_1, \texttt{size}_1\rangle, \langle \texttt{SHA-1}_2, \texttt{size}_2\rangle, \ldots\}$ pairs.

MKTMPFILE(`nfs_handle, fd`) :
creates a temporary server file to be used for atomic updates. The serer maintains a mapping from the ⟨`user, fd, errlist`⟩ triple to the `nfs_handle`. `errlist` tracks all errors during `user`'s operations on the temporary file, `fd`.

COMMITTMP(`nfs_handle, fd`) :
copies the contents of the temporary file associated with `fd` to the file identified by `nfs_handle` if no errors are associated with `fd`.

CONDWRITE(`fd, offset, len, SHA-1`) :
if the server has data associated with `SHA-1` in its index, the server writes that data to the temporary file identified by `fd`. Otherwise, it replies with HASHNOTFOUND.

TMPWRITE(`fd, offset, len, data`) :
the server writes `data` to the temporary file identified by `fd`.

LBFS assumes that clients persistently cache their entire working set. File operations on valid cache entries can be satisfied locally, and cache validation is only necessary in the case of an expired lease.

LBFS maintains a local chunk database to index all chunks in the local cache. The chunk index is keyed by just the first 64 bits of a chunk's `SHA-1`, mapping to ⟨`file, offset, count`⟩ triples.

LBFS does *not* rely on its index for correctness, which has several implications. Collisions are possible with a reduced 64-bit key-space, so all `SHA-1` fingerprints must be recomputed before reconstructing a file. Since the index tracks only local files, the index must be updated upon file writes. However, the index does not need to be updated synchronously, since errors in the index are detected by rehashing. An error in the index just results in wasted work.

A file read is initiated by the client with a GETHASH call. The server chunks the requested file, and responds with a list of the chunks' `SHA-1` hashes. The client only issues READ calls for data chunks that are not present in its local cache.

File data is only written to the server when a process that modified the file has closed the file. The client begins a write with a MKTMPFILE, to which the client provides a unique `fd`. The client chunks the file and pipelines a CONDWRITE for each chunk. If the server already has the chunk, it acknowledges, otherwise it replies with a HASHNOTFOUND message. The client sends a TMPWRITE for each chunk of new data, and finishes the write with a COMMITTMP. Since COMMITTMP completes a write atomically, it can be pipelined with outstanding TMPWRITE calls. The atomic commit also ensures close-to-open consistency; if multiple clients write in parallel, the last to successfully complete a COMMITTMP will succeed. An example of the network traffic exchanged during read and write operations is shown in Figure 2.

LBFS does not technically provide content addressable storage. Files are stored by opaque NFS handles, and LBFS runs over any legacy file system with an NFS server implementation. Despite this, LBFS occupies a relatively unique position in the overall CAS design space and is very influential for future CAS systems.

Very few CAS systems are designed for interactive use: archival systems are typically limited to secondary storage solutions, and Tangler [25] to document publishing. The reason is simple: interactive systems require responsiveness. Latency can be intolerable when every write triggers data chunking, chunk hashing, and index insertions, or when every small read triggers a multiple-disk-seek index lookup operation. LBFS identifies an application — running a network file system over an unreliable, low-bandwidth, wide area network — where latency can improve using CAS techniques. In this context, network delay >> disk delay >> CPU delay. LBFS uses CAS techniques to lower the burden placed on an expensive resource at the expense of placing an increased burden on less expensive resources.

LBFS defers work until a time when the cost of that work may go unnoticed. The overheads of chunking, hashing, and indexing are pushed to file open and file close, where network delays dominate the total overhead.

2.4.2 Case Study: Redundant Network Traffic

Proxy caches reduce the unnecessary transmission of static data on the Web. However, redundant information also exists in dynamic content like media streaming. Spring et al. present a protocol-independent method to eliminate redundant traffic between two endpoints, trad-

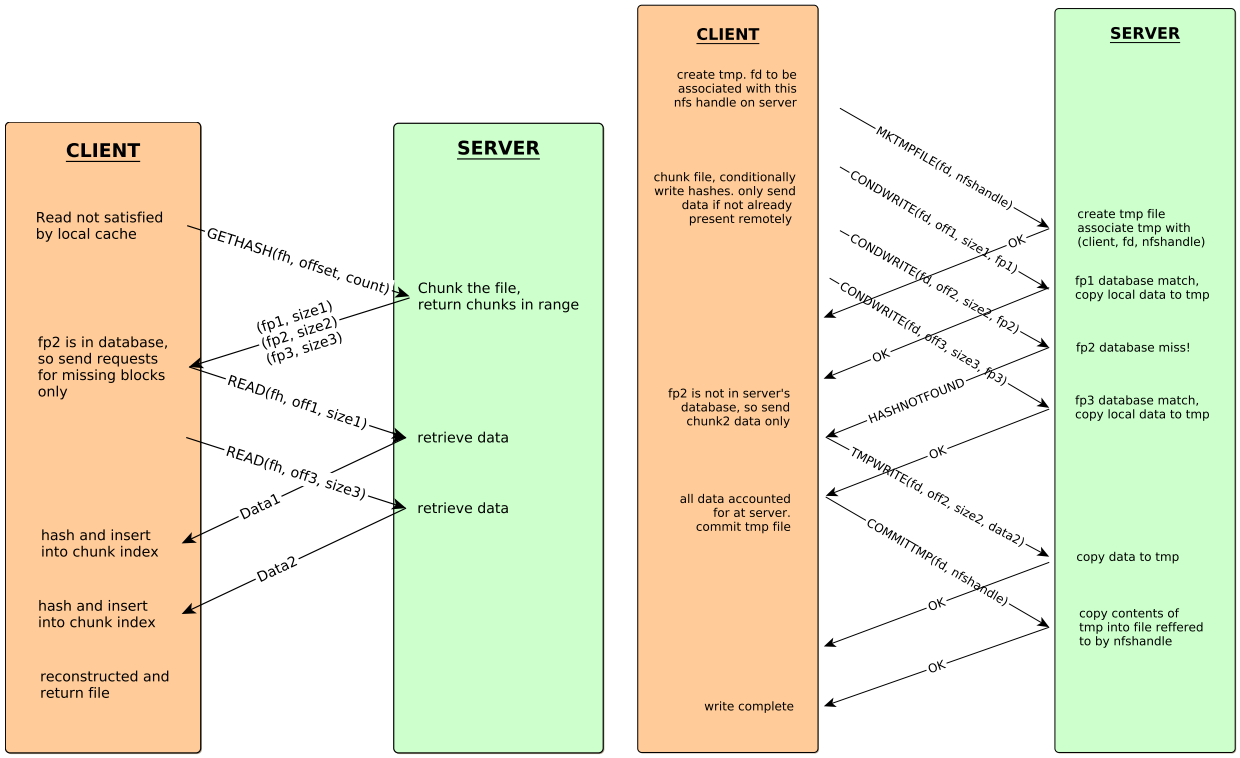

Figure 2: LBFS Client-server communication during example read (left) and write (right) operations. Note that LBFS heavily pipelines its messages. Specifically, the write (left) example shows a COMMITTMP message sent with an outstanding TMPWRITE. LBFS relies on the reliable and in-order delivery of its RPC calls.

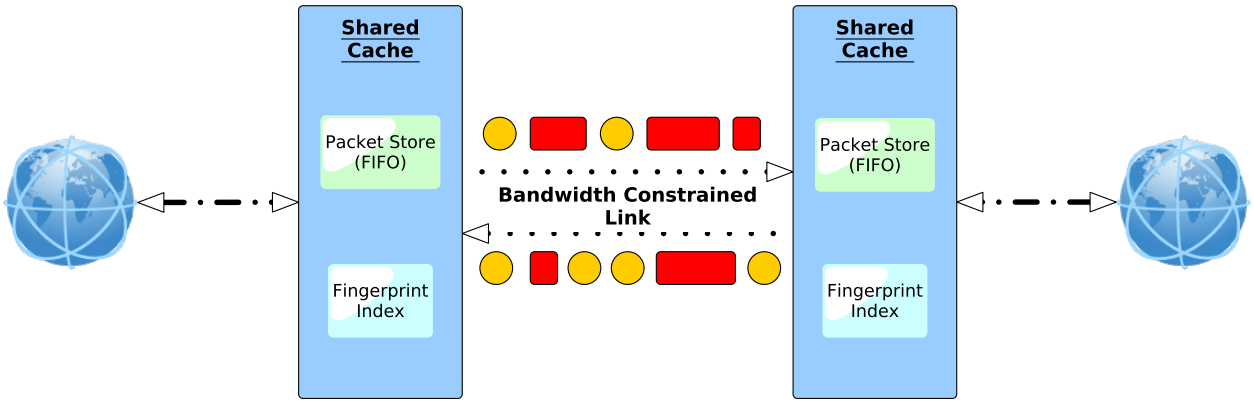

Figure 3: A proposed architecture to remove redundant network traffic. Two nodes communicating over a bandwidth-constrained link maintain shared caches. Each incoming packet is chunked and fingerprinted. The packet's representative fingerprints are checked against the fingerprint index. If a match is found, the match window is expanded byte by byte until a maximal redundant region is identified. The packet is encoded and sent as a smaller message — one that includes pointers instead of duplicate data. Unmodified data packets are shown as red rectangles. Circles depict encoded packets.

ing CPU and memory for reduced network bandwidth [23].

For each incoming packet, the sender constructs a set of **representative fingerprints** to use as anchors within the packet. The fingerprint of the first window, and any fingerprint whose last n bytes are zero, are considered representative. Each representative fingerprint is checked against the fingerprint index. For each match found, the incom-

ing packet and corresponding cached packet are compared byte by byte in each direction until the maximal region of overlap is identified. Repeated strings are encoded in a now smaller packet, and passed to the channel endpoint. The receiving channel endpoint expands the packet to its original form and forwards it to its destination. Both the sender and receiver add the packet to their packet cache, and the representative fingerprints to their index. Oldest members are evicted to keep caches manageable. Figure 3 shows the proposed architecture.

Like LBFS, this system is an example where CAS techniques are applied when the "storage" is the network rather than a persistent physical medium. The authors detect that 30% of incoming traffic and 60% of outgoing traffic is redundant with this system, so it appears to be applicable to common workloads. When the cost of bandwidth is more valuable than memory or CPU, this tradeoff makes sense. However, performance is highly reliant upon patterns in the underlying data and cache size.

2.5 Virtual Memory

The primary goal of VMware ESX server is the overcommitment of physical memory amongst many virtual machines [26]. ESX server proposes ballooning, content based sharing, and an idle memory tax—three independent techniques to manage memory in unmodified guest OSes. Content based sharing uses CAS methods, specifically hash-and-compare, to detect and minimize redundancy in memory pages.

Content based sharing is functionally similar to the deduplicate detection of fixed-size disk blocks. ESX server calculates the fingerprint of a candidate physical page's contents. If an identical fingerprint has been previously observed, the duplicate page is rehashed to ensure that its contents have not changed. On success, duplicate guest physical pages are mapped to a single host physical page and marked copy on write.

The rate of churn amongst memory pages is very high, making the fingerprinting and comparison of each page impractical. Thus, ESX server samples to minimize overheads. In practice, the most common instance of redundant data is the "zero page".

ESX server represents an interesting point in the CAS design space. ESX server performs duplicate detection over a fixed pool of rapidly changing candidate pages. Data is not immutable, so collisions must be verified as matches for correctness. The fingerprint index is small enough to fit in memory and must support rapid deletions.

3 Improving Dedup Systems

The two primary determinants of deduplication efficiency are chunking method and index management. Although related — average chunk size dictates the index size — the index and chunking algorithm can be optimized separately. Subsection 3.1 introduces the disk-bottleneck problem and discusses various methods to manage it. Subsection 3.2 discusses optimizations to the sliding window algorithm and Rabin fingerprinting. Subsection 3.3 follows with a discussion on how to manage the evolution of the data store.

3.1 The Disk Bottleneck: Efficient Indexing

Each new data segment added to the chunk store requires a corresponding ⟨`hash, location`⟩ entry be inserted to the index. The index quickly grows beyond the size of main memory for high-volume systems. Consider a chunk store which has grown to 20TB of unique data. If we store only a chunk's `SHA-1` hash (20B) in the index, an average chunk size of 4KB would result in a 100GB index!

Caching is often used to maintain high performance when data structures exceed the bounds of memory. Unfortunately, `SHA-1` fingerprints are independently and uniformly distributed, displaying no locality of reference. Index caching is not possible, so other means must be developed.

3.1.1 Case Study: Data Domain

The Data Domain deduplication system [29] introduces a three-tiered system to efficiently manage index queries. The system has two goals: (1) minimize the number of on-disk index lookups, and (2) use any work done in consultation of the index to aid in satisfying future queries.

The first level is the **summary vector**, a simple Bloom filter [3] that stores chunk fingerprints. A summary vector query is an imperfect, in-memory check for index membership. False negatives are impossible, and the false positive rate is tunable. A false positive just means advancing to level two, and does not add any additional disk seeks.

The **locality preserving cache** (LPC) comprises the second tier. The LPC is an in-memory hash table of index entries containing only recently accessed and candidate entries. An LPC cache miss directs the control flow to the final tier, the actual on-disk index.

The **stream informed segment layout** (SISL) groups chunks from a single data stream into the same **container**. Container layout is depicted in Figure 4. Metadata for each chunk is kept at the head of the container. When a miss occurs in the LPC, go to the disk to find the fingerprint's container. Then prefetch the segment descriptors for all members of that container into the LPC. In

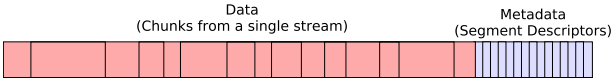

Figure 4: Container abstraction. Data from a single stream is chunked and appended to a until the container is filled. Chunk descriptors are stored contiguously at the end of a container, in order to be efficiently read into to the locality preserving cache. Containers are self-describing.

this way, the disk seek done to fetch one hash is used to preload candidates and prevent future lookups. Figure 5 shows the possible paths of a fingerprint lookup.

The summary vector, LPC, and SISL work together to reduce aggregate disk seeks for backup workloads. They are effective as a unit because each component fills a different role, the relative importance of which changes over time. The summary vector filters queries for new data blocks, which are common during an initial backup; however, fewer new blocks are seen thereafter. LPC prefetching loads the previously encountered block fingerprints from a single container into the cache. The LPC is most effective for data that has been previously stored and rarely changes. The LPC and SISL rely on chunk locality. Chunk locality is a reasonable assumption for backup workloads where files change incrementally. However, for frequently changing files, chunk locality LPC prefecting populates the cache with dead block fingerprints.

Together, the three techniques removed up to 99% of disk accesses for index lookups in two real deduplication workloads.

3.1.2 Case Study: PRUNE

Prompt RedUNdancy Elimination (PRUNE) is an inline deduplication archival system [17] that maintains a partitioned index. Specifically, the PRUNE index is broken into a series of smaller tables, called *tablets*, and each tablet is a self-contained fingerprint database. Any given fingerprint is stored in exactly one tablet, and the set of all tablets comprises the complete index. A tablet is small enough to fit into main memory.

At any moment, there is one tablet that is denoted *current*. All insertions are made to the current tablet. When the current tablet is full, a new, empty tablet is created and marked as current. This design ensures that fingerprints inserted in close temporal proximity will be stored in close spatial proximity as well.

PRUNE keeps the set of all tablet descriptors in a linked list. A fingerprint lookup first queries the table at the head of the list, and proceeds through the list in order until the fingerprint is found. On a "hit", the containing tablet descriptor is moved to the front of the list. Thus, tablets are sorted in LRU order. PRUNE also experiments with *tablet prefetching*. The hypothesis is that fingerprints in adjacent tablets are accessed together. On a tablet hit, both the containing tablet and its successor are moved to the front of the list. Tablet prefetching has no noticeable effect when compared to plain LRU.

PRUNE's partitioned index is somewhat similar to Data Domain's LPC, but it does not rely on the SISL to be effective. It is a index optimization that can complement, rather than replace, other measures. The partitioned index introduces spatial locality to fingerprints that exhibit temporal locality, despite the uniform distribution of `SHA-1` hashes. The one parameter to optimize is tablet size. If the tablet is too large, it will not fit in memory; smaller tablet sizes will introduce additional tablet searches. Tablets are ideal for systems with small, non-overlapping working sets; keeping the entire working set in a single memory-managed tablet would eliminate the disk bottleneck entirely.

3.1.3 Case Study: AMQ Data Structures

Bloom filters are a standard mechanism to query index membership. Bloom filters provide a tunable false positive rate, and can therefore prevent unnecessary disk accesses for fingerprints absent from the index. Large index sizes, however, may cause a Bloom filter to exceed memory capacity, at which point Bloom filter queries themselves trigger disk seeks.

Bender et al. [2] introduce an approximate membership query (AMQ) data structure called the **cascade filter**, that is designed to efficiently spill out of memory and onto flash storage. A single cascade filter comprises l levels of increasingly large **quotient filters**. The first level contains one in-memory quotient filter of size M, with the remaining quotient filters maintained on flash. The i^{th} on-flash quotient filter is of size 2^iM, because quotient filters implement efficient doubling, halving, and merge operations.

A quotient filter breaks a fingerprint into two parts: the most significant q bits (quotient), and the least significant r bits (remainder). The quotient determines a fingerprint's index, and is never explicitly stored. The bucket indexed by an element's quotient is called its *canonical slot*, and the remainder is stored in a fingerprint's canonical slot in the absence of collisions.

In the case of a collision, remainders are stored in a *run*. A run is a sorted list of remainders with the same canonical slot. A maximal sequence of adjacent runs is called a *cluster*. Collision resolution can cause an entire cluster to shift — including remainders that do not share canonical slots. In fact, only the first colliding run in a cluster will start at its canonical slot.

Three additional bits are kept at each bucket to help resolve collisions. The `is_occupied` bit denotes that a

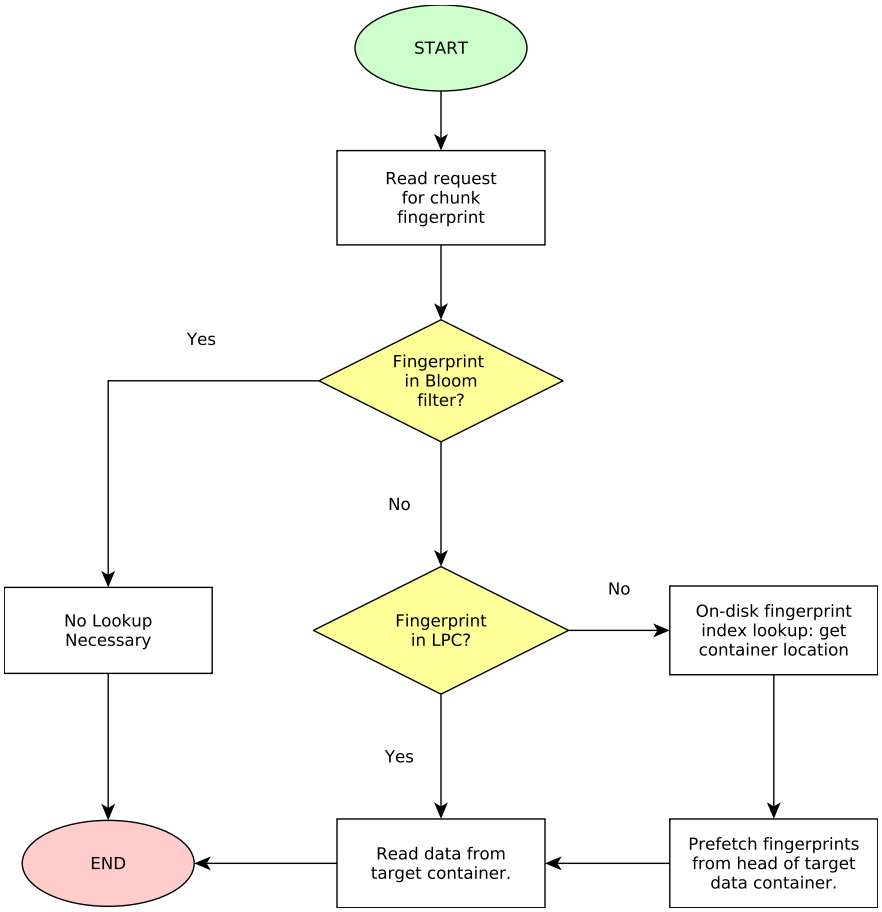

Figure 5: The Data Domain file system mitigates the disk bottleneck with its three-tiered architecture. The first level is a Bloom filter, which provides efficient in-memory approximate membership queries. A positive Bloom filter hit triggers a lookup in the in-memory locality preserving cache. A miss in the locality preserving cache finally results in an index lookup. Instead of reading just the requested fingerprint, all chunk descriptors from the containing segment are read into the locality preserving cache, evicting the oldest set of segment chunk descriptors.

bucket is the canonical slot for at least one fingerprint present in the quotient filter. The `is_continuation` bit signifies that a bucket stores an element of a run. The `is_shifted` bit signifies that the element in the bucket is not in its canonical slot. A quotient filter example can be seen in Figure 6 that details the usage of these bits.

Clearly, quotient filters can be efficiently merged (similar to merging sorted lists), halved in size (reinsert runs from left to right), and doubled in size (reinsert runs from right to left). These operations drive the cascade filter's l-level design.

Insertions are always made into the in-memory quotient filter at level 0. Whenever the level 0 quotient filter reaches maximum load, the first empty level i is identified. Quotient filters on levels $0,\ldots,i-1$ are merged into level i and cleared. Thus, insertions are always satisfied in-memory and require $O((\log(n/M))/B)$ amortized blocks writes/clears, where n is the number of fingerprints and B is the block size.

Lookups require one block read per level, until the fingerprint is found. This results in a worst-case lookup of $O(\log(n/M))$.

AMQ data structures are an essential optimization to avoid the disk bottleneck in archival CAS systems. Yet Bloom filters are often sufficient — the value of cascade filters only comes when a Bloom filter cannot fit into main memory. Cascade filters are write-optimized because archival systems provide cold storage. The primary concern is that backups complete in the prescribed window. And although index lookups are more frequent than insertions, the design of cascade filters creates locality of reference despite the uniform distribution of fingerprints. The most recently inserted fingerprints will always be at the uppermost levels.

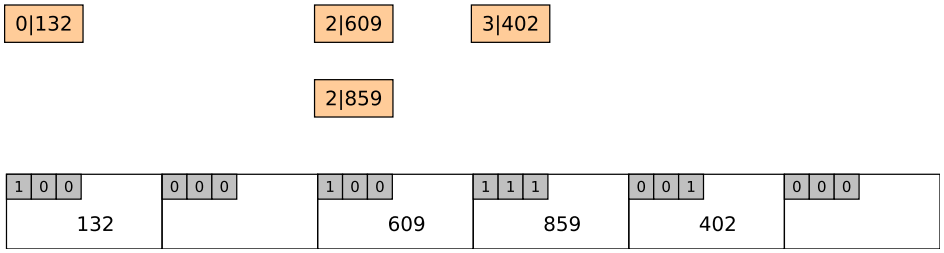

Figure 6: The four members inserted into the quotient filter are each shown above their canonical slot, with quotient and remainder separated by a vertical bar. The first bit signifies that an index is the canonical slot for some element in the quotient filter, the second bit that the element stored in the slot is part of a run, and the third bit that the slot is not the canonical slot for the element it contains.

3.1.4 Case Study: Sparse Indexing

Sparse indexing (SI) is another technique to avoid the chunk index disk bottleneck in an inline archival deduplication system [14]. The system borrows its disk layout and data container policies from Data Domain [29], but takes an otherwise different approach to the problem - SI fits its entire index in memory by storing only a fraction of its fingerprints in the index.

SI breaks each data stream into variable-sized chunks using the two thresholds two divisors (TTTD) chunking algorithm [9]. Chunks are then grouped into **segments**, also using TTTD: TTTD is applied to chunk fingerprints instead of chunk data. The sliding window is shifted one chunk hash at a time instead of one byte at a time, and a cut point is defined when the window's fingerprint modulo a pre-defined divisor is -1. The segment is SI's unit of storage and retrieval, so all chunk data referenced by hashes before the cut point are deduplicated and represented as a group. The hashes of each chunk in a segment are stored in a segment manifest as set of $\{\langle$`chunk hash, location, length`$\rangle,\ldots\}$.

For an incoming segment S_i, SI samples a portion of S_i's chunk hashes to be stored in the in-memory index. Sampled hashes are referred to as **hooks**. A hook is chosen when the first n bits of the chunk's hash are 0, yielding a sampling rate of $1/2^n$. Index entries map each hook to a list of manifests in which the hook appears, and are of the form $\langle$`hook, `$\{$`manifest`$_i,\ldots\}\rangle$.

For each hook in S_i, SI retrieves from the index the list of manifests that contain it. Then, SI iteratively selects the manifest that contains the most hooks not present in previously chosen manifests, until the maximum number of manifests are chosen (or none remain). The selected manifests are loaded into memory, and S_i is deduplicated against them.

Chunks not present in any manifest are written into a data container on disk. SI creates a new manifest for S_i and adds it to the manifest store. S_i's hooks are then added to the index. For hooks already present in the index, S_i's manifest is appended to the list of containing manifests.

The design above illustrates destination deduplication — the client sends all chunks to the server, even if the chunk is a duplicate. The authors propose a compromise between source and destination deduplication, in which the client chunks, segments, and hashes the data locally. The client then sends just the hashes to the server, which chooses manifests, identifies duplicates, and requests only the chunks it needs. Such an implementation would greatly reduce network traffic.

SI relies on the assumption of *chunk locality*: the idea that chunks in backup streams reoccur together. Chunk locality implies that when two pieces of backup streams share chunks, those backup streams share many chunks. SI's reliance on chunk locality limits the technique's applicability to archival backups, since random reads and writes do not exhibit this pattern.

The in-memory sparse index eliminates the disk bottleneck for fingerprint lookups, but SI requires several disk seeks to load manifests. However, segments are on the order of megabytes, so manifest reads are amortized over thousands of chunks; loading manifests incurs negligible overhead.

Duplicate chunks can be stored if the chunk's fingerprint is not unique, but is not present an any of the candidate manifests. This does not compromise correctness; it merely lowers the deduplication ratio. Chunk locality will minimize, but not eliminate, this occurrence.

3.2 Chunking

Chunking is the division of a data stream into one or more segments. The segment is the granularity at which data is stored and redundant data detected; chunking, more than any other system component, determines the system's deduplication ratio.

There are two dimensions in which to evaluate a chunking algorithm: run time performance and chunk quality. Section §3.2.1 describes the two thresholds, two divisors (TTTD) algorithm [9]. TTTD reduces chunk size vari-

ance and lowers the overhead introduced by small modifications.

The comparison of the runtime performances of two chunking methods is difficult. The speed of identifying boundaries can be directly compared, but a system that identifies more shared data will save disk I/O of duplicate writes. TTTD provides more uniformly sized chunks, which lowers the overall overhead of data insertions, but does so at the cost of additional computations.

3.2.1 Case Study: Two Thresholds Two Divisors

The two thresholds, two divisors algorithm (TTTD) is a generalization of the standard sliding window algorithm (SW) described in Section §2.4.1. SW has three parameters: window size w, divisor D, and target t. LBFS observed that for some inputs, SW produces very small or very large chunks. LBFS enhanced the algorithm with minimum and maximum chunk size threshold ($T_{\texttt{min}}$ and $T_{\texttt{max}}$) to bound chunk size. TTTD again extends the algorithm and adds a second divisor D'.

For window W and target t, we say a *fingerprint match* against divisor D occurs if

$$\text{FINGERPRINT}(W) \mod D \equiv t \tag{2}$$

The TTTD algorithm slides a fixed-length window W across the data stream byte by byte, starting at $T_{\texttt{min}}$. TTTD checks for a fingerprint match against D and D' at each position. A fingerprint match against D immediately halts execution and defines a chunk boundary. A fingerprint match against D' is saved for future use. TTTD continues until reaching $T_{\texttt{max}}$. The most recent fingerprint match against D', if any, defines the chunk boundary. Otherwise, a boundary is inserted at $T_{\texttt{max}}$.

A second divisor reduces chunk size variance — fewer boundaries are defined by $T_{\texttt{max}}$. Chunks defined by $T_{\texttt{max}}$ are fixed-size chunks, and therefore suffer the same boundary shifting problem. TTTD reduces the overhead of data that is replicated not because it is redundant, but because a chunk modification disassembled its previous chunk. An appropriate choice of D' reduces the overhead of the second check.

3.3 Evolving the chunk store

An optimal deduplication system is insufficient if it cannot scale to the desired workload. Resource reclamation, returning allocated resources that are no longer in active use, is one way to increase a system's scalability. Grouped mark and sweep (GMS) [8] reclaims unused data blocks and scales proportional to the working set size rather than the total storage capacity.

The traditional mark and sweep algorithm is broken into two phases. The first phase is to iterate through all *files*, and mark the segments that are in use. The second phase is to iterate through all *segments* and release any segments that are not marked. Mark and sweep is resilient to errors because it can be repeated without side effects if stopped prematurely. However, mark and sweep is not scalable because the sweep phase must touch each segment in the file system.

Archival backups are incremental; between consecutive backups, only a small working set of files change. GMS reduces the number of files to touch in both the mark and sweep phases by tracking changes at the granularity of groups of files. GMS defines a *backup* as a set of files, and a *group* as a set of backups. Between successive runs, GMS maintains a list of all groups with modified files. GMS only marks files in changed groups. Marks are stored between runs and reused for any unchanged groups. Between runs, GMS also keeps a list of all modified containers. Only containers with blocks contained by groups with deleted files are used in the sweep phase.

The GMS goal is to scale with the size of the working set—the mark phase marks only modified groups and the sweep phase sweeps only modified containers. In practice, the actual scalability is determined by average backup, group, and container size, and the locality of chunk reference. A working set that contains blocks spanning many containers will suffer the same scalability issues as traditional mark and sweep, so this technique is primarily applicable to archival backups.

4 Data Verification

Increasing proportions of the world's information are generated solely as electronic records. Unfortunately, the stakeholders of most digital documents are not the ones in control of the data's physical storage. Documents are often *produced* by one party in order to be *consumed* by a second party while being *managed* by a third party; and each party holds unique and often conflicting incentives and allegiances.

Content addressable storage can verify data integrity. A user simply recalculates and compares content addresses to confirm that the data received exactly matches the data requested. CAS also creates a unique global namespace, preventing document overwrites, as well attacks such as "name squatting", where a namespace is polluted with empty or irrelevant documents to distract from or crowd out more meaningful documents. Unfortunately, users must maintain mappings from human-readable handles to metadata structures in order to reconstruct complete files from content addresses. This leads to the common pattern where untrusted servers export a content addressed block store, over which mutually distrusting clients layer high level abstractions to implement trusted storage. Table 2

summarizes the three systems discussed in this section.

4.1 Resisting censorship

In general, a system to resist censorship would have the following properties:

Documents should be publishable anonymously, and dissociable from any one publisher. Protecting the identities of content producers and providers shields them from third parties abusing influence. Additionally, it is unlikely that the document be censored in its entirety when multiple independent publishers are in control of the document.

Data should be mobile and replicated. Mobility and replication defend against denial of service and acts of nature. Replication also disperses responsibility, so no one host can be singled out by an influential third party.

The system should resist both malicious servers and malicious publishers.

Actors should have incentives to cooperate and possess limited capability to harm the system.

All parties should be aware whenever censorship has taken place. Users should know if and when the system has failed, even if recovery is not possible.

4.1.1 Case Study: Tangler

Tangler [25] is an example of a censorship-resistant system. The basis of Tangler's design is to tie the integrity of each document to the integrity of other, previously existing documents through block **entanglement**. Entanglement does two things: (1) directly incorporates replication into the publishing process, and (2) makes reconstruction of source blocks from entangled blocks impossible if blocks are modified. Entanglement is essential to Tangler's function, and will be discussed below. However, this section only describes entanglement at a high level; the focus of this document is on applications of CAS.

Tangler acknowledges that adversarial nodes will exist regardless of its best efforts. The system is built to be self-policing: as long as the majority of nodes are good, the Tangler network can detect *and* eject adversarial nodes. Every Tangler server can validate a response to any Tangler request. If a server fails to respond, a user can forward its request to a second server, enlisting the second server as a witness of bad behavior. Malicious servers can also be detected when contradictory responses are signed by the same key. Additionally, Tangler takes steps to limit the effect an adversary may have in two ways. First, publishers may only consume storage resources up to some fraction of the resources that they themselves are willing to provide. This limits the ability of adversaries to deny service by flooding the network to capacity. Capacity is managed through a credit system. Second, a node must perform work that benefits Tangler, before it receives full privilege in the Tangler network; a new server cannot consume storage during the server's first month in the network. This ensures that any adversary that does harm the system has already contributed towards an equal amount of beneficial work.

The architecture of Tangler is divided into three components: (1) publishing, (2) reconstruction, and (3) network management. The following will detail Tangler components in that order.

Documents are anonymously published in **collections**, or groups of files published under the same public key. The owner of the collection is the only one capable of updating previously published documents, since he maintains sole possession of the private key. Public keys name collections, and private keys sign them. The integrity of a collection is verified using a hash tree [16]. Hash trees allow users to verify large amounts of data but store only a single `SHA-1` hash.

The publication process takes as input a public/private key pair and a directory, and outputs a set of entangled blocks. Entangled blocks are named by their `SHA-1` hash. A hash tree is created for the collection and signed by the collection private key. The resulting data is distributed to nodes across the network.

The process of publishing a single data block replicates two previously existing serer blocks. Further, any change made to those dependent blocks will be evident when reconstructing the original data. In order to publish a file, Tangler requires the file be split into fixed-size data blocks. Tangler uses 16K blocks, with the last block padded if necessary. Each 16K data block is entangled with two random server blocks retrieved from the pool of previously published server blocks, producing two new server blocks. The two old server blocks are replicated and the two new server blocks are added to the pool. Figure 7 shows the process of data block entanglement. No plaintext data blocks are ever stored.

Next, the metadata required to reconstruct the file—the `SHA-1` of each entangled block—is recorded and the metadata is entangled with two randomly selected server blocks. The output of this entanglement is a list of four server block names that can be used to reconstruct the file.

Each file in the directory goes through this process of data segmentation and data block entanglement, then metadata production and metadata entanglement.

The list of metadata block names and the metadata's dependent blocks are recorded in the collection's root, which

System	Description	Keywords
Tangler	Censorship-resistant document publishing	Fixed-size block entanglement, replication, self-policing, overlay network
SUNDR	Tamper evident storage on untrusted servers	Fork consistency, global block store, hash trees
SFS-RO	Read-only distributed file system on untrusted storage over untrusted channels	Self-certifying path names

Table 2: Data verification file systems.

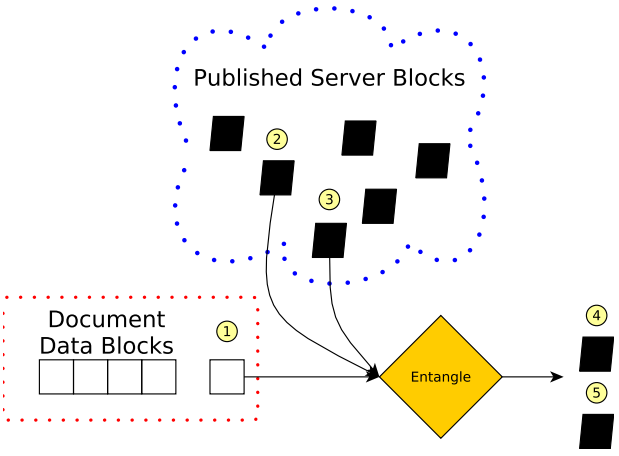

Figure 7: Tangler files are first broken into 16KB data blocks. Each data block is entangled with 2 random blocks from the server block pool. Here, data block 1 is entangled with server blocks 2 and 3. The outputs are two new server blocks, blocks 4 and 5. The data block is discarded, but server blocks 2 and 3 are replicated when blocks 2, 3, 4 and 5 are all published to the block pool.

is digitally signed and indexed by `SHA-1` hash. Collection roots store version numbers, so as to make update as simple as republishing with an incremented version.

Eventually Tangler grows a web of dependencies, large enough that any unauthorized document changes do not go unnoticed.

Tangler's block entanglement is based on Shamir's secret sharing algorithm [22]. A secret s is divided into n shares, such that $k \leq n$ of the shares are required to reconstruct n. Tangler uses $n = 4, k = 3$ — entanglement produces 4 server blocks, and 3 of them are sufficient to reconstruct the original data. We omit a complete description of the entanglement algorithm for brevity, as it is outside the focus of this work.

The Tangler network's design is self policing, and it incentivizes all actors to contribute toward the network's greater good. However, the CAS paradigm is the source of Tangler security. Specifically, Tangler leverages the properties of `SHA-1` fingerprints to verify data, distribute blocks, and prevent censorship:

- `SHA-1` is a one-way function, so an adversary cannot search for individual blocks by human-readable means; he must already have complete knowledge of the block in order to locate it.
- An adversary who successfully makes a change to a block only creates a new one; the old block will always exist at its original content address.
- An adversary in control of a server and able to falsify block responses will be detected by users hashing and comparing blocks upon receipt.
- The random distribution of addresses ensures that blocks within a collection's block set will likely not be co-located, dissociating the responsibility of a collection from any one server.

The negative impacts of CAS are few. The Tangler usage model is not demanding of high performance, so low latency and CPU consumption are not a concern; network delay will likely be the bottleneck. Tangler design is markedly different from deduplication systems for an obvious reason: replication is a primary Tangler goal.

4.2 Self-certifying file systems

The level of security actually provided to data is often dictated by the available data management options and not the needs of the data's stakeholders. In what follows, we detail the implementations of two systems that address this problem in different ways. SUNDR [13] allows clients to store data on untrusted servers, and guarantees clients relative freshness. Further, the server cannot interpret the data or alter file contents or metadata. Read-only SFS [11] exports a global tamper-evident file system with self-certifying pathnames. SFS endpoints can engage in secure communication over unsecured channels without reliance on a third party.

4.2.1 Case Study: SUNDR

SUNDR provides clients a file system interface to remote storage. SUNDR [15] servers are managed by hosts that are unable to interpret or modify stored data. And though there is no guarantee that the server will behave, SUNDR

guarantees that any misbehavior will be detectable.

SUNDR is implemented in a two-level architecture. Its foundation is the block server, which stores chunks of data indexed by the cryptographic hash of their contents. And though servers store blocks, they neither interpret nor understand them, enabling the owners of the physical server to manage the storage without access to the file system itself. The upper tier, the file system, is implemented entirely on clients on top of the block store abstraction.

All client communication with the server occurs over an authenticated channel. Servers possess public/private key pairs, but can only access the server public key; private key knowledge is restricted to the superusers of each file system. A client and server negotiate a symmetric session key for message authentication codes (MAC). A MAC ensures that data arrives unmolested.

SUNDR's blocks are addressed by the hash of their contents. When storing a block, the the server also maintains a list of all users who reference the block, and a per-user reference count. These reference counts are used to implement garbage collection. Block store requests can be for an entire data block, or, to save bandwidth, a combination of new data and regions of existing blocks to be concatenated. This allows for network efficient updates of existing structures.

The file system contains several key data structures. The *virtual inode* contains a file's metadata, and is used to reconstruct the file and verify its contents. The virtual inode contains a list of the hashes of each block that makes up the file. If the block list is too large to fit in one inode, the last block hash refers to an indirect block, which contains a continuation of the block list. Indirect blocks are used to chain lists, and chains can extend to as many indirect blocks as necessary. A consequence of this design is that a seek to a long file requires the retrieval of multiple indirect blocks.

Virtual inodes contain a 64-bit per-user inode number, and the *i-table* maps these numbers to hashes of the virtual inode's data block. The i-table itself is also broken into blocks, converted into a hash tree, and stored on the server. The i-table's hash-tree's root is called the *i-handle*. The relationship between SUNDR data structures is represented in Figure 8.

SUNDR maintains its consistency through counters. Each user stores a *version structure*, shown in Figure 9. The version structure contains the user's i-handle, and a set of ⟨`user, version number`⟩ pairs that completely represent that user's view of the file system state. A violation of SUNDR's consistency protocol is illustrated in Figure 10.

Figure 9: The SUNDR version structure. A user's version structure stores the user's uid, ihandle, and version list. A version list contains, for each user in the system, the user's most recent version number. The entire structure is signed.

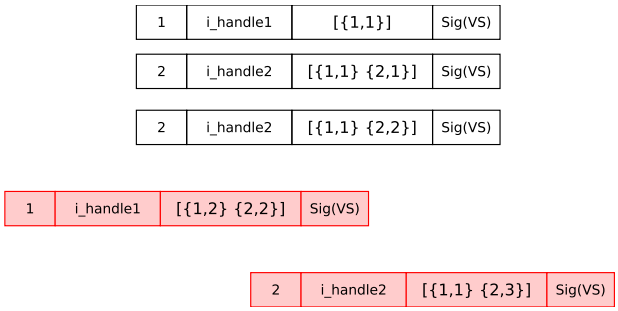

Figure 10: A sequence of writes by two users where fork consistency is violated. The version list presented to user 1 and the version structure presented to user 2 can never be resolved.

This protocol is sufficient to guarantee **relative freshness**.

Definition A file system provides **relative freshness** iff, whenever user u_1 sees the effects of an open or close operation O by u_2, then at least until u_2 performed O, u_1 had close-to-open consistency with respect to u_2.

CAS is essential to the design of SUNDR. CAS forms the basis for verifying blocks and detecting unauthorized modifications. CAS allows clients to address blocks rather than files, preventing the server from interpreting the data's logical structure.

SUNDR is in no way optimized for run-time or resource efficiency. However the overheads induced by the use of legacy index structures are secondary to the cryptological and bookkeeping overheads required to guarantee consistency. We also observe the common structure of a complex client overlaying its file system over the block store exported by simple servers.

4.2.2 Case Study: Read-only SFS

Read-only data often has performance, availability, and security requirements. For example, Mozilla's Firefox web browser is publicly available for download as a binary executable, and Mozilla uses replication and caching to lower download speeds and increase availability. This replication adds to the attack surface as there are more servers and therefore more opportunities for a binary to be

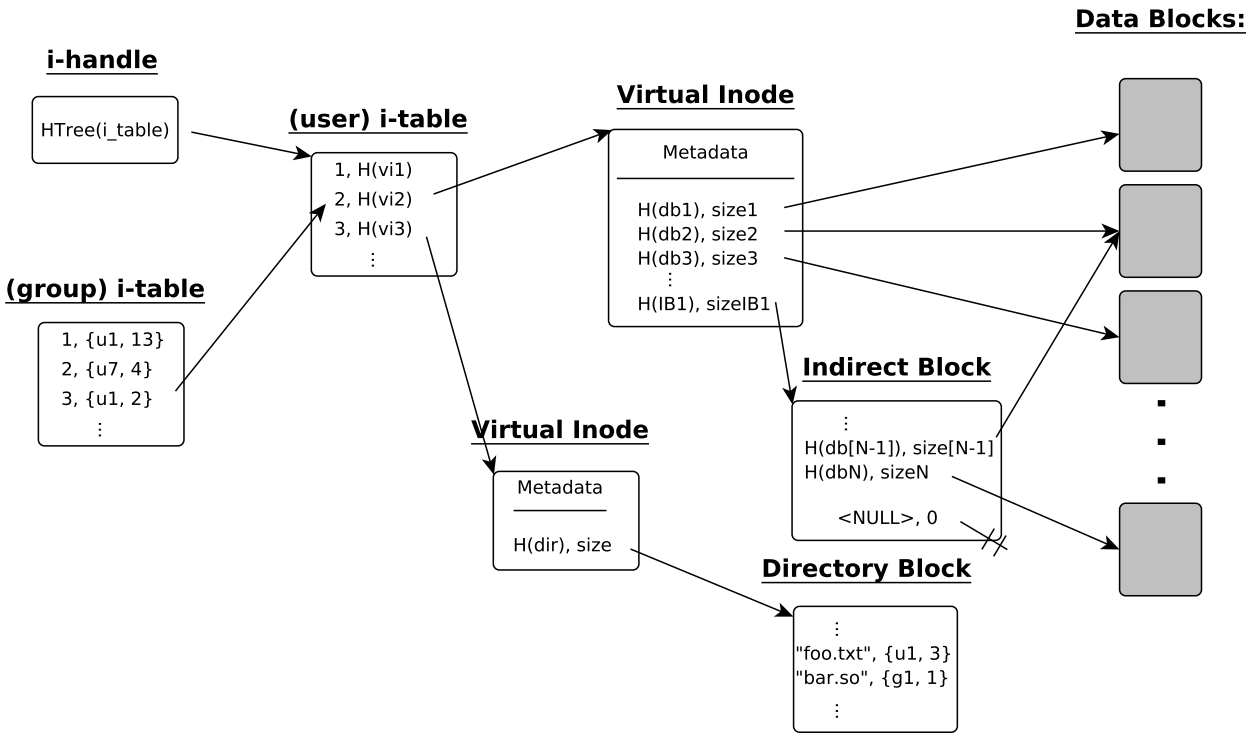

Figure 8: The interaction of data structures in SUNDR. Each user keeps an *i-handle* to retrieve and verify its file system tree. Per-user i-tables store $\{usr\_inumber \rightarrow ihandle\}$ mappings. A per-group i-table adds a layer of indirection, instead storing $\{grp\_inumber \rightarrow \{principal, ihandle\}\}$ mappings. Virtual inodes contain metadata, followed by a list of block `SHA-1` hashes. For files that are large, indirect blocks store a continuation of the virtual inode's list of data block hashes. Directories list $\{pathname \rightarrow \{principal, inumber\}$ mappings for each child.

compromised. The fast and secure read-only file system (SFS-RO) verifies the integrity of read-only data stored on untrusted servers, and the simple server design of SFS-RO scales to many simultaneous connections [11].

SFS-RO functionality is broken into two roles, client and server. The server is kept simple. A server's primary job is to satisfy requests for data blocks. The client is responsible for implementing the file system, as well as performing all cryptological operations.

Every file system has a public key, which is encoded in the name of each file. All cryptological operations are performed by the client, so a server will never require the private key. The private key is used to digitally sign file system objects.

An administrator runs the SFS-RO database generator (`sfsrodb`) to create a file system. The `sfsrodb` takes as inputs a directory of files and a private key. The `sfsrodb` outputs a signed database mapping `SHA-1` handles to metadata and data blocks. SFS-RO uses the same virtual inode metadata structure as SUNDR. Blocks are keyed by `SHA-1(iv, data)` handles to limit a client's exposure to `SHA-1` collisions. The file system owner then replicates the signed database and all data as a collection of content-addressed blocks onto untrusted servers.

A server can be any untrusted machine running the SFS read-only server daemon (`sfsrosd`). `sfsrosd` responds to two RPCs: one that looks up a `SHA-1` handle in the database and serves the corresponding data block (`getfsdata`), and one that returns a signed handle of the file system's root node as part of the file system summary structure (`getfsinfo`). The `FS_INFO` structure is shown in Figure 11. The `FS_INFO` structure is sufficient for a client to reconstruct and verify the entire file system.

SFS-RO uses time to loosely enforce consistency—a signed file system will expire after a configurable amount of time. The `start` field in the `FS_INFO` structure is a timestamp of the file system signature. The file system remains valid until (`start` + `duration`). Hence, the `duration` field limits a client's exposure to stale data. `duration` also represents a commitment by the file system owner to periodically republish the file system. A balance between publishing overhead and the window of vulnerability must found. To protect from a server "rolling back" the file system to an unexpired but succeeded version, clients cache the most recent timestamp.

Clients run the SFS read-only client daemon (`sfsrocd`)

```
struct FS_INFO {
  sfs_time start;
  unsigned duration;
  opaque   iv[16];
  sfs_hash root_fh;
  sfs_hash db_fh;
}
```

Figure 11: The FS_INFO structure is sufficient for a client to reconstruct and verify the entire file system. The start field records the timestamp of the signature, which is valid until (start+duration). The iv field is used to avoid SHA-1 collisions. The root_fh is the SHA-1 handle of the root directory, and the db_fh is the handle for a hash tree to verify all handles reachable from the root.

to interact with untrusted replicas. The sfsrocd receives and responds to all local OS file system requests. The sfsrocd must verify that all data read from untrusted servers is unmodified, authentic, and fresh. Freshness is handled by timestamps, as discussed above, and the digital signatures of FS_INFO structures provide authenticity. All blocks are content addressed, so verifying data integrity is trivial.

The db_fh is a handle for a hash tree verifying all handles reachable from the root. The db_fh can be inspected if an expected handle is reported missing by a misbehaving server. When the owner of a file system makes changes, some files handles may be removed. The validity of a handle_not_found response can be verified by the db_fh. Yet the db_fh design imposes some overheads. A single change in the file system requires much of the tree to be recomputed. The db_fh also has a negative effect on clients with open files. If the iv is changed, *all* open files become invalid. If a file is modified, any open instances of that file become invalid. Future versions of SFS-RO will abandoned the db_fh altogether. Instead, clients will construct the entire namespace locally, and then re-validate all open handles when an update occurs.

SFS-RO is most similar to SUNDR—both SUNDR and SFS-RO provide file system integrity on untrusted storage. But SFS-RO and SUNDR solve different problems. SUNDR users can read and write files with the guarantee of fork consistency. Yet these guarantees impose a heavy cost and do not necessary scale. SFS-RO needs to scale well and it does. SFS-RO keeps the server simple. The burdens of complexity are pushed to the client, and the read-only nature simplifies the consistency protocols. Thus, SFS-RO servers can support many simultaneous connections.

Part I: Beyond virtual disks

Section §5 discusses the requirements of virtual storage, presenting a case study of Ventana [19], a virtualization aware file system. Ventana takes encapsulation, isolation, and versioning — the most salient features of virtual disks — and layers them over a distributed object store. Section §6 revisits the content addressable storage paradigm, arguing in favor of CAS as the basis for VM storage. We present Conclave, a prototype implementation of a legacy Linux file system over the CAS abstraction. Section §7 discusses future work, experiments, and enhancements to Conclave design. Finally, Section §8 concludes.

5 Virtualization aware storage

Versioning, isolation, and migration are possible in virtual disks because virtual disks fully encapsulate storage. A virtual disk contains all files, captures all dependencies, and tracks all file system state of a running virtual machine. As a result, a virtual disk copy is sufficient to resume a VM on any compatible host. Full encapsulation also allows VM users to version file system state. A snapshot of VM storage can form the base of an *n*-ary tree—a golden client upon which customized virtual machines are based. Snapshots can also capture development milestones, or provide short term error recovery with a full-system roll-back.

Yet for virtual disks, the granularity of these operations is coarse. A user cannot migrate part of a virtual disk, version a subset of files, or selectively share just a part of storage. The granularity of a virtual disk operation is the entire virtual disk, all-or-nothing. These limitations are specific to virtual disks, and are not inherent to the operations themselves.

5.1 Case study: Ventana

Ventana provides the feature set of traditional distributed file systems, but adds versioning, isolation, and encapsulation to meet the needs of virtual storage.

Ventana files are tracked in *branches* to manage versioning. Related sets of files are grouped into file trees, and any tree can be tracked as a **private**, **shared**, **non-persistent**, or **volatile** branch. A branches always begins as an exact copy of some other branch, and form a linear chain of histories. Branch versions monotonically increase: there is always a *current* branch, and any two version numbers are directly comparable.

The two persistent branch types are distinguished by the number of expected simultaneous client actors. A private branch will only be modified by a single client VM — there is no guarantee that a change made to a file in a private branch will be propagated to a simultaneous external viewer. Conversely, changes made to a file in a shared branch are made immediately visible to others.

The remaining two branch types store ephemeral data. A non-persistent branch is deleted upon VM reboot. Non-persistent branches are designed for cases such as the storage of the Unix `/tmp` directory. Volatile branches are both non-persistent and local; volatile branches are deleted upon VM migration in addition to VM reboot.

There are some files, such as password files, where versioning exposes a security risk. Ventana allows **unversioned** files to accommodate. At any time, there exists one copy of an unversioned file: the current version. Unversioned files cannot be rolled back.

To initialize VM storage, the user defines a *view*. Views consist of one or more file trees mapped onto a namespace. View definitions specify a mount point, a branch type, and set of branch permissions for each tree. An example view might map the Debian root file system as a read-only public branch, a set of standard applications as a read- only public branch, and a home directory as a private branch. Such view structures let administrators push security patches to all users with a single branch patch, and still provide users isolated personal storage in their private branch.

Ventana protects file system resources with access control lists (ACLs) at three different granularities: file, branch, and version. Any access requires the permission of all ACLs protecting the resource.

File ACLs. The guest OS should have complete control over how it delegates privilege. *Guest file ACLs* are managed by the guest OS for this purpose. Guests may use any format, but the Ventana prototype implements guest file ACLs with standard Unix ACL semantics. Yet guest file ACLs are not sufficient to mediate privilege in a network file system. Sharing is central to Ventana's design; servers host the actual files, and servers may deny any principal access to any resource, even if the guest OS allows it. Servers exercise this control via *server file ACLs*. Both guest and server file ACLs are part of file metadata; modifications to file ACLs produce new file versions.

Version ACLs. Version ACLs are stored as a part of a file's version structure — not as part of the metadata object. Hence, a version ACL may be changed without producing a new version of the file. Ventana enforces version ACLs as follows: "r" gives a principal the right to read a file, and "c" the right to change a file's version ACL. Version ACLs control access to a particular version of a file, even if that version exists in multiple branches.

Branch ACLs. A branch ACL mediates access to all files in a branch and to all files in all that branch's ancestors.

Table 3 details the Ventana enforcement policies.

Ventana servers are layered over an object store. Each version of a file, each version of a file's metadata, and each branch is represented as a unique object. Files are identified by 128-bit random i-node number, and objects similarly have 128-bit integer ids. For example, a single file with i-node number 16 may have versions represented by objects 13, 37, and 42, spread across branches 19700101 and 12122012.

A single metadata server hosts all databases used to identify Ventana objects. The *version database* tracks which object number represents the latest version of a file in a particular branch. The *branch database* manages the file system's branch structures. Finally, there are two unnamed databases: the first maps branch names to object numbers and the other database contains VM configurations.

Client operation is implemented by the *host manager*. Ventana requires one host manager per platform. The host manager does not run in its own VM, but client VMs must communicate with the host manager to read and write data, and to take file system snapshots.

The host manager maintains both in-memory and on-disk caches for files and their metadata. These caches decrease latency and enable disconnected operation. Caches also add complexity to file system consistency — the remote version database must be polled for updates on every shared object access.

Host manager caches data at the granularity of whole objects. Objects are immutable and may therefore be cached indefinitely. Write buffering, in addition to reducing latency, limits excessive versioning: new versions are only created by snapshot operations. All writes commit to local disk and incur zero network traffic. Even explicit flushes are committed locally. Write operations are unimpeded by disconnection, and read requests are satisfied by the local cache when possible. However, local changes made to shared branches are not seen by external clients and vice versa. When connection is reestablished, updates are published to the server. Ventana does not address conflict resolution.

Ventana provides the features of a virtual disk without any of the unwanted baggage. The guest overlays a distributed object store with its own file system personality, divorcing VM storage from the physical host machine and host software. This separation of file system management from low-level storage abstractions is representative of the structures of many CAS systems mentioned above, and

an appropriate division of labor. However, we think that CAS provides a more expressive model than the object store, and consequently use CAS as the foundation of our virtualization aware file system prototype.

ACL	Enforcer	Versioned
File (guest)	Guest OS	Yes
File (server)	Ventana	Yes
Version	Ventana	No
Branch	Ventana	No

Table 3: A description of Ventana ACL types. File access requires permission according to all ACLs.

6 Conclave

This section introduces Conclave, a file system design that provides CONtent addressed loCaLly cAched ViEws of cloud storage. Figure 12 illustrates the relationship among the key components of Conclave.

Conclave is implemented in three tiers. Guest VMs, or *clients*, communicate locally with a single *CAS server*. Local CAS servers in turn communicate with an authoritative *cloud server*. Both local and cloud servers are static, but clients have no mobility restrictions. Clients may be checkpointed and migrated to any platform whose local CAS server is associated with the client's cloud.

Local CAS servers expose to clients two primary abstractions: the key-value store and a content addressed store of fixed-size blocks. The client layers its file system personality on top of these abstractions. This design divorces the guest OS management of high level file system abstractions from the low level storage optimizations of the CAS server. Further, the content addressed block store presents to clients the illusion of infinite storage; clients no longer struggle to incrementally add storage by manually resizing file system partitions, as they do with virtual disks.

6.1 Adapting CAS for primary storage

We have previously examined CAS in the context of verifiable storage (§4), censorship resistance (§4.1), and deduplication for large data backups (§2). The requirements for primary storage are vastly different. This subsection details the requirements of primary storage, and their influence on Conclave design.

Latency is critical to responsiveness and the user experience — low-latency must be a priority for any primary storage solution. Contrast this with other applications of CAS, where latency is either of no concern (backup, censorship) or a secondary consideration (verification).

Random reads and writes are the common access patterns of primary storage. Contrast this with backup workloads. Backups must complete in limited time windows, and consist of large streaming writes that exhibit chunk locality. Reads occur rarely, if at all.

Deletion of data is actually prohibited in Venti [20], and largely ignored by deduplication systems. In primary storage, files are constantly evolving, which would result in many unreachable blocks in a CAS block store. Blocks must be detected as unreachable and reclaimed in order for the system to scale. Minimizing the frequency of writes through heavy buffering would also help reduce the proliferation of unreachable blocks by reducing the storage of intermediate file state.

We deviate from common practices in order to adapt the CAS paradigm to primary storage. Specifically, Conclave stores all data in fixed-size, 4KB blocks. This drawbacks of fixed-size blocks are discussed in detail elsewhere (boundary-shifting problem §2.4.1), but storing some duplicate data does not conflict with our goals. Conclave uses fixed-size blocks for two primary reasons:

- **Efficient data transfer**. We created a Linux kernel module to perform fast copy-on-write (COW) transfers of physical pages between VMs. 4KB is the default Linux page size on the `x86-64` platform, and is also the common block size of "green disks". We consequently use 4KB as the unit of transfer in Conclave — both for bulk IPC and as the block size of data in the CAS store.
- **Fast chunking**. To reduce latency, we eliminate the sliding window fingerprint calculations used to identify chunk boundaries in other systems. Data is transferred in units of CAS blocks, so no chunking is required.

Additionally, Conclave relies heavily on client side caching for performance and scalability. Often, read and write requests can be satisfied in-memory. For writes, dirty data is not sent to the server until file close. This heavy write buffering prevents wasteful consumption of three resources: (1) CPU consumption for fingerprinting incremental updates that are overwritten, (2) disk consumption for writing intermediate blocks that are rendered unreachable, and (3) fingerprint index size by preventing dilution by intermediate block representations.

We describe the storage abstractions exported by CAS servers in subsection 6.2, and the communication protocol in subsection 6.3. Then we present the prototype Conclave design. Subsection 6.4 describes our implementa-

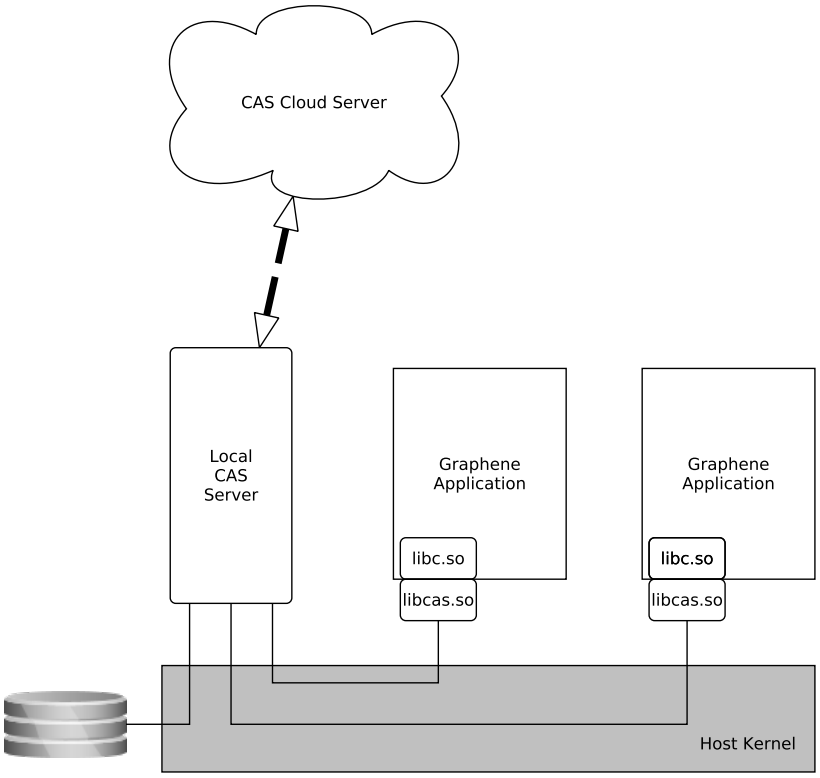

Figure 12: An overview of Conclave architecture. Each host contains one local CAS server, which has complete control over a region of physical media. Local clients communicate with the local CAS server. Local CAS servers send and retrieve blocks from an authoritative CAS cloud server. The cloud may coordinate with many local CAS servers, which makes client migration possible.

tion from the client's perspective, §6.5 from the server, and §6.6 from the cloud.

6.2 Storage abstractions

CAS servers provide three key storage abstractions over which clients layer file system personality. This subsection describes those abstractions. The relationships between these abstractions are illustrated in Figure 13.

Recipe: A file *recipe* stores file metadata, and is used to reconstruct a file from its constituent blocks. A recipe begins with a variable-length metadata section, into which clients may store data structures for internal management; the metadata section of our current legacy Linux file system implementation contains file size, ownership, permissions, and modification times. Clients may store metadata in arbitrary formats, and the CAS store makes no attempt to interpret this data. However, once allocated, the metadata section cannot be resized.

Following the metadata section, a recipe stores a list of a file's data block fingerprints. Like data blocks, recipes are chunked and committed to the CAS block store, so recipe blocks are limited to a fixed length. To accommodate large files, recipe blocks are chained. Each recipe block contains the `SHA-1` fingerprint of its successor, or `NULL` if it is the last block in a chain. We call the first recipe block in a chain the *recipe head*, and successors *indirect recipe blocks*. The fingerprint of a file's recipe head is its *CAS handle*, which is used for all read and write operations. Clients may interact with the CAS server directly, or they may map an arbitrary pathname to a CAS handle in one or more namespaces.

Namespace: A *namespace* is a key-value store that contains mappings from a pathname to the `SHA-1` hash of a recipe head. We use the term pathname loosely, as namespaces may be keyed by arbitrary binary data. A namespace is identified by its globally unique namespace identifier (NSID), which facilitates sharing and migration. A client that wishes to share a namespace with another client can communicate the NSID out of band. We expect the CAS handles of common system libraries to be stored in read-only namespaces with well-known NSIDs. When a client migrates to a new physical machine, the machine's local CAS server retrieves requested namespaces from the cloud, if they

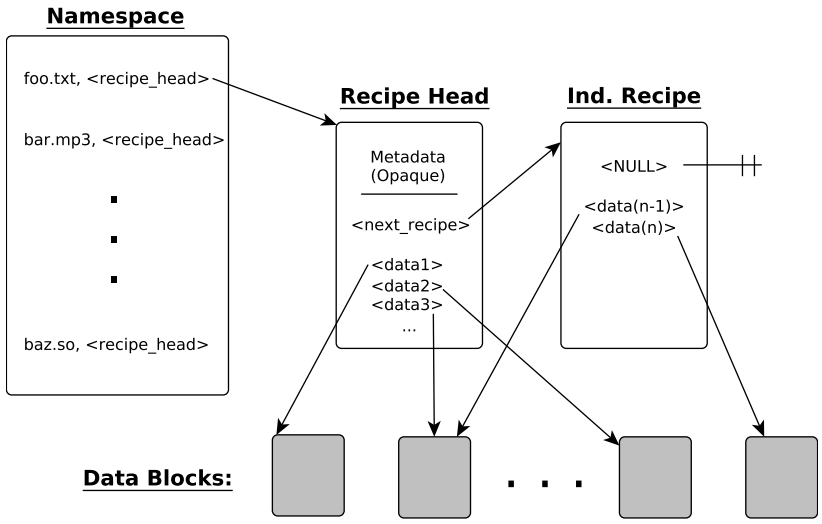

Figure 13: Relationships amongst the various Conclave storage abstractions. A namespace stores a set of related files in a key-value store — namespaces map arbitrary binary data to the `SHA-1` of a file's recipe head. A file recipe is used to reconstruct a file from the file's constituent data blocks. The recipe head starts with variable-sized metadata section, followed by a list of the file's data block `SHA-1` fingerprints. Conclave chains recipes for long files.

are not already present. A method to maintain consistency amongst shared namespaces at separate local CAS servers has been identified as future work.

Data block: A *data block* is a 4KB chunk of data that is fingerprinted, indexed, and committed to the CAS block store. A data block whose fingerprint is not contained in any active recipe chain is said to be an *unreachable block*. A recipe chain is active if the CAS handle of its recipe head is contained in one or more namespaces. Unreachable blocks may be reclaimed from CAS block stores (local server and cloud).

6.3 Client-server protocol

A client interacts with a local CAS server through a well defined API, as illustrated in Table 4. The server exposes 15 API calls to the client:

- Five calls deal exclusively with namespace management. Clients can add, query, remove, and list keys. `SHA-1` hashes are not iterable.
- Two calls read and two calls write data. A client may specify either the fingerprint of a recipe head, or a key from a valid namespace as the target of the operation. The advantage of using a namespace entry is that the server will update the ⟨`key,hash`⟩ mapping upon success.
- One call to assert a ⟨`key,hash`⟩ mapping.
- Two calls are used to start and end a *minitransaction* [1]. Minitransactions are *mini* because they do not allow read operations. A client may make assertions about the state of the namespace during a minitransaction. A batch of updates are executed atomically and durably if all assertions are true.

Function	Tx?	Notes
`hash = get(key, ns)`		
`put(key, hash, ns)`	Y	
`read(key, pages, offset ns)` `read(hash, pages, offset)`		Recipes may be referenced by key or hash.
`newhash = write(key, pages, offset, ns)` `newhash = write(hash, pages, offset)`	Y	
`m = getMeta(key, ns)`		
`newhash = putMeta(m, key, ns)`	Y	
`k1,k2... = listkeys(ns)` `k1,k2... = listkeys(prefix, delim, ns)`		
`newhash = dropblocks(key, newsize, ns)`		
`success = drop(key, ns)`		
`expect(key, hash, ns)`	Y	Assert key map to hash.
`xbegin()`	Y	Start minitransaction.
`success = xend()`	Y	End minitransaction.

Table 4: CAS Server API. Entries marked with a 'Y' in the Tx column may be issued in a minitransaction.

A VM is initialized with two channels. The first is a byte stream over which control messages are exchanged with the server. Control messages specify a message

type and the corresponding message parameters. The second channel is for bulk data transfers in units of physical pages. Each bulk transport channel endpoint maintains a FIFO queue of pending pages. Control messages specify the number and direction of pages to be passed, and the bulk IPC module adds the memory to the corresponding channel endpoint queue.

6.4 Client prototype

This section describes the prototype implementation of a legacy Linux file system in Graphene. Graphene clients layer file system personality over the exported server abstractions.

Linux files are represented by recipes. Metadata is persistently stored in the recipe head, including size, mode, modification time, and target path if a file is symbolic link. The client also maintains data structures analogous to those of the Linux VFS: `dentry`, `inode`, `file`, and `address_space`. The `dentry` stores both the pathname and the fingerprint of the recipe head for quick namespace assertions. The `address_space` buffers data and tracks dirty pages.

Once a file is cached, most file system operations can be satisfied with in-memory data structures. Caching serves the dual purpose of decreasing latency and preventing the proliferation of intermediate file state. Since CAS blocks are immutable, a premature write consumes disk and index space. Writes are deferred until all open references to buffered data are closed, at which point dirty pages are written in a batch. Aggressive caching until file close, together with atomic namespace updates, provides close-to-open consistency.

Graphene clients store rooted file system trees in isolated namespaces. Conclave represents namespaces as key-value stores, so the client is responsible for maintaining tree structure. Conclave was intentionally designed to keep the server simple—clients pay for their own complexity, but are at the same time granted much more freedom to deviate from traditional file system designs.

Namespaces may be shared amongst VMs. The totality of VM storage comprises the union of one or more namespaces. *Manifests* are used to initialize virtual machine storage. A Graphene manifest can specify a list of namespaces and their corresponding mount points, in addition to other recognized file system types.

6.5 Server prototype

Local Conclave servers are granted complete control over a fixed contiguous region of physical storage. In our prototype implementation, servers interact with storage through device drivers in the host OS.

Server storage is divided by function into isolated, non-overlapping regions. The first two 4KB blocks are used as checkpoint summary blocks. Checkpoint summary blocks encapsulate the server state at a single point in time. Checkpoint summary blocks specify checkpoint completion time, the offset of a fingerprint index checkpoint region, and the start of the data block region. When a checkpoint is taken, the oldest valid summary block, if any, is overwritten. Conclave can restore itself in the event of system failure during a checkpoint operation by loading the checkpoint specified by the older of the two summary blocks.

The Conclave prototype uses a single in-memory radix tree as its fingerprint index, which is periodically serialized to one of two on-disk index checkpoint regions. The block store size is a server parameter, so the storage required to checkpoint a full index is known *a priori*. The two index regions are laid out consecutively.

The data section follows the index checkpoint regions. A block manager allocates space using an in-memory free-block bitmap. This bitmap is not check-pointed—it is generated from scratch during index deserialization. The data section is divided into 4KB blocks.

To commit a block, data structures must be reserved in order, to maintain consistency in the event of error. First, the block is fingerprinted an the index queried. If the block's fingerprint is found, no write is necessary. If the fingerprint is not found, a special dummy value is inserted to signify that the block write is pending. The free block bitmap reserves space on disk, and the block is written at the reserved offset. The index is finally updated to reflect write completion.

6.6 Cloud Prototype

We currently do not have an operational cloud prototype. The cloud server will ultimately be very similar in structure to a local CAS server. However, the cloud will be responsible for coordination amongst local CAS server peers, especially for client migration. The cloud will also be responsible for garbage collection, since the cloud is the authoritative block store.

We implemented a prototype for basic mark and sweep garbage collection on a local CAS server in anticipation of cloud development. The server iterates through each namespace, and adds the hash of each recipe-reachable block to a Bloom filter. The server than walks the fingerprint index and removes each block mapping that is not in the Bloom filter and has a timestamp older than the start of the garbage collection process. It finally returns the unmapped block to the free block map for future use.

7 Looking ahead

Conclave is a proof-of-concept prototype. This section discusses potential optimizations and proposes modifications to Conclave design. The majority of proposed design changes are to the CAS server, in anticipation of the cloud server implementation.

Index characteristics dramatically affect performance, and an in-memory index is not practical for large data stores. The three-tiered design of Conclave complicates the index requirements, and index structure selection is an open problem. We plan to investigate the efficacy of a partitioned index, similar to tablets in PRUNE [17]. The index will be broken into *slices*, each of which tracks an individual working set.

Conclave must manage the migration of data between local CAS servers and the cloud, both when clients migrate and when data exceeds a local CAS server's available space. Recall that a single cloud server is a system's authoritative CAS block store, and local CAS servers are caches of that cloud data. A partitioned index reduces the problem of data migration to working set management. If an index slice comprises the data blocks of a single client's working set, then that index can be sent to the cloud as a unit when the client migrates. Compare this to a unified index, where data is evicted at block granularity: each block eviction requires an index lookup, disk seek, network transfer of fingerprint data, and index delete. However, an index slice could be sent to the cloud as one data stream. The cloud server could walk through the slice and identify the blocks that are not already present. If an earlier version of the slice existed at the cloud, most lookups will be satisfied by queries to that single, old slice. The sender can then transmit the (mostly contiguous) blocks of the segment that the slice indexed as one network stream. The entire local index slice can be deleted, and blocks reclaimed, once the data is resident on the cloud server.

We plan to explore containers for Conclave storage management. The container abstraction is central to the disk layout policies of many archival CAS systems [8, 14, 17, 20, 29], and sparse indexing [14] uses containers as the unit of duplicate detection. Adapting containers will be a challenge. Containers are popular because backup workloads exhibit chunk locality, but we do not yet know if primary storage workloads will exhibit chunk locality. Containers would nonetheless be attractive for two reasons. First, containers are useful to implement grouped mark-and-sweep garbage collection: the namespace data structure can be enhanced to track all containers that store namespace-reachable data blocks. Second, containers could define the unit of indexing granularity for an LRU-partitioned tablet index. A container and its slice of the fingerprint index can be evicted from a local CAS server to the cloud as a unit, a simple server management policy.

Despite their benefits, containers would add complexity to CAS server design. One CAS server is shared amongst multiple clients. If clients interleave writes, a single container would include data from multiple working sets. Two clients, each with a working set size of one container, would instead span blocks across two containers. Another difficulty would be container cleaning. Some systems, like Venti, prohibit data deletion and therefore do not address this problem. For systems with resource reclamation, container cleaning is necessary to avoid fragmentation. A primary goal of Conclave design is to keep CAS servers simple. Expensive operations should be deferred and pushed to the cloud if possible. Segment cleaning, in addition to being an expensive operation, can *only* be carried out at the cloud—as the authoritative server, only the cloud knows which blocks can be safely reclaimed. If containers quickly accumulate unreachable blocks, resources would be wasted copying dead data to the cloud.

More performance evaluation of the Conclave prototype is necessary. We constructed a FUSE module that mimics client behavior in order to expedite the testing process. We can run any unmodified program on a mounted FUSE file system; development of the CAS server can thus proceed independently of Graphene.

The immutability of CAS blocks affects the aging of storage in ways that we do not yet understand. The following aspects must be studied in order to select an appropriate design:

- unreachable blocks
- working set size
- "hot" and "cold" data characteristics

A final point of exploration is the inclusion of scratch space. If the main sources of latency are fingerprinting and index lookups, it may be desirable to temporarily commit data to a circular log and provide intermediate pointers to address these blocks. Data can then be committed to the CAS store in the background, and pointers replaced with actual content addresses.

8 Conclusion

This paper introduces Conclave, a prototype file system designed in conjunction with Graphene, a Linux library OS. Conclave explores a unique point in the CAS design space: primary storage. Conclave layers guest OS personality atop a content addressed block store, and divorces the management of high-level file system abstractions from low-level storage optimizations. Future Conclave development will focus on low-level storage optimizations, such as block allocation, and attempt to address concerns of scalability that arise in a frequently evolving

block store.

References

[1] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. Sinfonia: a new paradigm for building scalable distributed systems. In *SOSP*, pages 159–174, 2007. 22

[2] M. A. Bender, M. Farach-Colton, R. Johnson, B. C. Kuszmaul, D. Medjedovic, P. Montes, P. Shetty, R. P. Spillane, and E. Zadok. Don't thrash: how to cache your hash on flash. In *HotStorage*, 2011. 10

[3] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *CACM*, 13(7):422–426, jul 1970. 9

[4] W. J. Bolosky, S. Corbin, D. Goebel, and J. R. Douceur. Single instance storage in windows 2000. In *WSS*, pages 13–24, 2000. 2, 4

[5] O. P. Cox, C. D. Murray, and B. D. Noble. Pastiche: making backup cheap and easy. In *OSDI*, pages 285–298, 2002. 2

[6] A. G. Dimakis, P. B. Godfrey, M. J. Wainwright, and K. Ramchandran. Network coding for distributed storage systems. In *INFOCOM*, 2007. 5

[7] C. Dubnicki, L. Gryz, L. Heldt, M. Kaczmarczyk, W. Kilian, P. Strzelczak, J. Szczepkowski, C. Ungureanu, and M. Welnicki. HYDRAstor: a scalable secondary storage. In *FAST*, pages 197–210, 2009. 4, 5

[8] P. Efstathopoulos and F. Guo. Rethinking deduplication scalability. In *HotStorage*, pages 7–7, 2010. 13, 24

[9] K. Eshghi and H. K. Tang. A framework for analyzing and improving content-based chunking algorithms. HP Laboratories Palo Alto HPL-2005-30R1, 2005. 2, 12

[10] L. Freeman. Looking beyond the hype: Evaluating data deduplication solutions. Netapp White Paper, September 2007. 2

[11] K. Fu, M. F. Kaashoek, and D. Mazières. Fast and secure distributed read-only file system. *TOCS*, 20(1):1–24, February 2002. 15, 17

[12] D. Geer. Reducing the storage burden via data deduplication. *Computer*, 41(12):15–17, Dec. 2008. 2

[13] J. Li, M. Krohn, D. Mazières, and D. Shasha. Secure untrusted data repository (SUNDR). In *OSDI*, pages 121–136, 2004. 15

[14] M. Lillibridge, K. Eshghi, D. Bhagwat, V. Deolalikar, G. Trezise, and P. Camble. Sparse indexing: large scale, inline deduplication using sampling and locality. In *FAST*, pages 111–123, 2009. 4, 12, 24

[15] D. Mazires and D. Shasha. Don't trust your file server. In *HotOS*, pages 113–118, 2001. 15

[16] R. C. Merkle. A digital signature based on a conventional encryption function. In *CRYPTO*, pages 369–378, 1988. 14

[17] J. Min, D. Yoon, and Y. Won. Efficient deduplication techniques for modern backup operation. *TC*, 60(6):824–840, june 2011. 2, 4, 10, 24

[18] A. Muthitacharoen, B. Chen, and D. Mazières. A low-bandwidth network file system. In *SOSP*, pages 174–187, 2001. 2, 4, 6

[19] B. Pfaff, T. Garfinkel, and M. Rosenblum. Virtualization aware file systems: Getting beyond the limitations of virtual disks. In *NSDI*, pages 353–366, 2006. 18

[20] S. Quinlan and S. Dorward. Venti: A new approach to archival storage. In *FAST*, pages 89–101, 2002. 2, 3, 4, 20, 24

[21] M. O. Rabin. Fingerprinting by random polynomials. Harvard Aiken Computational Laboratory TR-15-81, 1981. 2, 6

[22] A. Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, Nov. 1979. 15

[23] N. T. Spring and D. Wetherall. A protocol-independent technique for eliminating redundant network traffic. In *SIGCOMM*, pages 87–95, 2000. 8

[24] C. Ungureanu, B. Atkin, A. Aranya, S. Gokhale, S. Rago, G. Calkowski, C. Dubnicki, and A. Bohra. HydraFS: A high-throughput file system for the HYDRAstor content-addressable storage system. In *FAST*, pages 225–238, 2010. 4, 5

[25] M. Waldman and D. Mazières. Tangler: a censorship-resistant publishing system based on document entanglements. In *CCS*, pages 126–135, 2001. 7, 14

[26] C. A. Waldspurger. Memory resource management in VMware ESX server. *OSR*, 36(SI):181–194, 2002. 9

[27] Y. Won, J. Ban, J. Min, J. Hur, S. Oh, and J. Lee. Efficient index lookup for de-duplication backup system. In *MASCOTS*, pages 1–3, sept. 2008. 3

[28] L. You, K. Pollack, and D. D. E. Long. Deep Store: An archival storage system architecture. In *IDCE*, pages 804–815, 2005. 2, 3, 4

[29] B. Zhu, K. Li, and H. Patterson. Avoiding the disk bottleneck in the data domain deduplication file system. In *FAST*, pages 269–282, 2008. 3, 4, 9, 12, 24