Fast, Inexpensive Content-Addressed Storage in Foundation

Sean Rhea[*]
Meraki, Inc.

Russ Cox, Alex Pesterev[*]
MIT CSAIL

Abstract

Foundation is a preservation system for users' personal, digital artifacts. Foundation preserves all of a user's data and its dependencies—fonts, programs, plugins, kernel, and configuration state—by archiving nightly snapshots of the user's entire hard disk. Users can browse through these images to view old data or recover accidentally deleted files. To access data that a user's current environment can no longer interpret, Foundation boots the disk image in which that data resides under an emulator, allowing the user to view and modify the data with the same programs with which the user originally accessed it.

This paper describes Foundation's archival storage layer, which uses content-addressed storage (CAS) to retain nightly snapshots of users' disks indefinitely. Current state-of-the-art CAS systems, such as Venti [34], require multiple high-speed disks or other expensive hardware to achieve high performance. Foundation's archival storage layer, in contrast, matches the storage efficiency of Venti using only a single USB hard drive. Foundation archives disk snapshots at an average throughput of 21 MB/s and restores them at an average of 14 MB/s, more than an order of magnitude improvement over Venti running on the same hardware. Unlike Venti, Foundation does not rely on the assumption that SHA-1 is collision-free.

1 Introduction

We are "living in the midst of digital Dark Ages" [23]. As computer users increasingly store their most personal data—photographs, diaries, letters—only in digital form, they practically ensure that it will be unavailable to future generations [28].

Considering only the cost of storage, this state of affairs seems inexcusable. A half-terabyte USB hard drive now costs just over $100, while reliable remote storage has become an inexpensive commodity: Amazon's S3 service [1], for example, charges only $0.15/GB/month.

Alas, mere access to the bits of old files does not imply the ability to interpret those bits. Some file formats may be eternal—JPEG, perhaps—but most are ephemeral. Furthermore, the interpretation of a particular file may require a non-trivial set of support files. Consider, for example, the files needed to view a web page in its original form: the HTML itself, the fonts it uses, the right web browser and plugins. The browser and plugins themselves depend on a particular operating system, itself depending on a particular hardware configuration. In the worst case, a user in the distant future might need to replicate an entire hardware-software stack to view an old file as it once existed.

Foundation is a system that preserves users' personal digital artifacts regardless of the applications with which they create those artifacts and without requiring any preservation-specific effort on the users' part. To do so, it permanently archives nightly snapshots of a user's entire hard disk. These snapshots contain the complete software stack needed to view a file in bootable form: given an emulator for the hardware on which that stack once ran, a future user can view a file exactly as it was. To limit the hardware that future emulators must support, Foundation confines users' environments to a virtual machine. Today's virtual machine monitor thus serves as the template for tomorrow's emulator.

Using emulation for preservation is not a new idea (see, e.g. [15, 35, 38]), but by archiving a complete image of a user's disk, Foundation captures *all* of the user's data, applications, and configuration state as a single, *consistent* unit. By archiving a new snapshot every night, Foundation prevents the installation of new applications from interfering with a user's ability to view older data—e.g., by overwriting the shared libraries on which old applications depend with new and incompatible versions [8]. Users view each artifact using the most recent snapshot that correctly interprets that artifact. There is no need for them to manually create an emulation environment particular to each artifact, or even to choose in advance which artifacts will be preserved.

Of course, such comprehensive archiving is not without risk: the cost of storing nightly snapshots of users' disks indefinitely may turn out to be prohibitive. On the other hand, the Plan 9 system archived nightly snapshots of its file system on a WORM jukebox for years [32, 33], and the subsequent Venti system [34] drastically reduced the storage required for those archives by using content-addressed storage (CAS) [18, 44] to automatically identify and coalesce duplicate blocks between snapshots.

The Plan 9 experience, and our own experience using a 15-disk Venti system to back up the main file server of a research group at MIT, convinced us that content-addressed storage was a promising technique for reducing Foundation's storage costs. Venti, however, requires multiple, high-performance disks to achieve acceptable archival throughput, an unacceptable cost in the consumer setting in which we intend to deploy Foundation. A new design seemed necessary.

[*] Work done while at Intel Research, Berkeley

The core contribution of this paper is the design, implementation, and evaluation of Foundation's content-addressed storage system. This system is inspired by Venti [34], but we have modified the Venti design for consumer use, replacing Venti's expensive RAID array and high speed disks with a single, inexpensive USB hard drive. Foundation achieves high archival throughput on modest hardware by using a Bloom filter to quickly detect new data and by making assumptions about the structure of duplicate data—assumptions we have verified using over a year of Venti traces. Our evaluation of the resulting system shows that Foundation achieves read and write speeds an order of magnitude higher than Venti on the same hardware.

While we built Foundation for digital preservation, content-addressed storage is useful in other contexts, and we believe Foundation will enable other applications of CAS that were previously confined to the enterprise to enter the consumer space. As an anecdotal example, we note that within our own households, most computers share a large percentage of their files—digital photos, music files, mail messages, etc. A designer of a networked household backup server could easily reduce its storage needs by adopting Foundation as its storage system.

In this paper, however, we focus on the CAS layer itself. To ground the discussion, Section 2 provides background on the Foundation system as a whole. Sections 3–5 then present the main contributions of the paper—the design, implementation, and evaluation of Foundation's content-addressed storage layer. Section 6 surveys related work, Section 7 describes future work, and Section 8 concludes.

2 Background: Foundation

Figure 1 shows the major components of a Foundation system. The host operating system runs on the raw hardware, providing a local file system and running Foundation. Users work inside the active VM, which runs a conventional OS like Windows XP or Linux atop Foundation's *virtual machine monitor* (VMM). The VMM stores virtual machine state (disk contents and other metadata) in the local file system. Every night, Foundation's *virtual machine archiver* takes a real-time snapshot of the active VM's state, storing the snapshot in the *CAS layer*.

In addition to taking nightly snapshots of the VM's state, the VM archiver also provides read-only access to previously-archived disk images. The VMM uses this functionality to boot past images; the figure shows an archived VM snapshot running in a separate VM. As a convenience, Foundation provides a *file system snapshot server* that interprets archived disk images, presenting each day's file system snapshot in a synthetic file tree (like Plan 9's dump file system [32] or NetApp's `.snapshot` directories [17]) that VMs can access over SMB.[1] A user finds files from May 1, 1999, for example, in `/snapshot/1999/05/01/`. This gives the active VM access to old data, but it cannot guarantee that today's system will be able to understand the data. The fallback of being able to boot the VM image provides that guarantee.

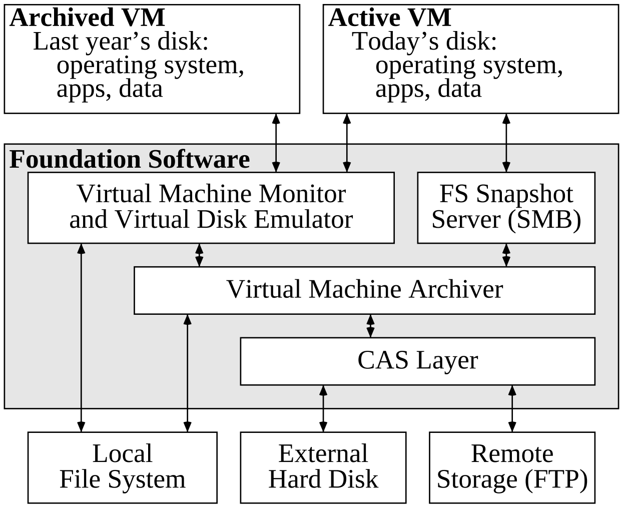

Figure 1: Foundation system components. A Foundation user works inside the active VM, which is archived daily to an external hard disk and (optionally) a remote location. Foundation presents archival file system data using SMB and enables users to interpret obsolete file formats by booting VM snapshots from days or years past.

Foundation's CAS layer provides efficient storage of nightly snapshots taken by the VM archiver. The CAS layer stores archived data on an inexpensive, external hard disk. Users can also configure the CAS layer to replicate its archives onto a remote FTP server for fault tolerance. To protect users' privacy, the CAS layer encrypts data before writing to the external hard drive or replicating it. It also signs the data and audits the local disk and replica to detect corruption or tampering.

As a simple optimization, Foundation interprets the partition table and file systems on the guest OS's disk to identify any swap files or partitions. It treats such swap space as being filled with zeros during archival.

The remainder of this section discusses the components of Foundation in detail, starting with the VMM and continuing through the VM archiver and CAS layer.

2.1 Virtual Machine Monitor

Foundation uses VMware Workstation as its virtual machine monitor. Foundation configures VMware to store the contents of each emulated disk as a single, contiguous file, which we call the disk image. VMware's snapshot

[1] Providing the snapshot tree requires that Foundation interpret the partition table and file systems on the guest OS's disk. Foundation interprets ext2/3 and NTFS using third-party libraries. Support for other file systems is easy to add, and if no such library exists, a user can always boot the VM image to access a file.

facility stores the complete state of a VM at a particular instant in time. Foundation uses this facility to acquire consistent images of the VM's disk image.

To take a snapshot, VMware reopens the disk image read-only and diverts all subsequent disk writes to a new partial disk image. To take a second snapshot, VMware reopens the first partial disk image read-only and diverts all subsequent disk writes to a second partial disk image. A sequence of snapshots thus results in a stack of partial disk images, with the original disk image at the bottom. To read a sector from the virtual disk, VMware works down the stack (from the most recent to the oldest partial disk image, ending with the original disk) until it finds a value for that sector [2].

To discard a snapshot, VMware removes the snapshot's partial disk image from the stack and applies the writes contained in that image to the image below it on the stack. Notice that this procedure works for discarding any snapshot, not just the most recent one.

The usual use of snapshots in VMware is to record a working state of the system before performing a dangerous operation. Before installing a new application, for example, a user can snapshot the VM, rolling back to the snapshotted state if the installation fails.

2.2 Virtual Machine Archiver

Foundation uses VMware's snapshot facility both to obtain consistent images of the disk and to track daily changes between such images.

Foundation archives consistent images of the disk as follows. First, the VM archiver directs VMware to take a snapshot of the active VM, causing future disk writes to be diverted into a new partial disk image. The archiver then reads the now-quiescent original disk image, storing it in the CAS layer along with the VM configuration state and metadata about when the snapshot was taken. Finally, the virtual machine archiver directs VMware to discard the snapshot. Using a snapshots in this way allows Foundation to archive a consistent disk image without suspending the VM or interrupting the user.

Note that the above algorithm requires Foundation to scan the entire disk image during the nightly archival process. For a large disk image, this process can take considerable time. For this reason, Foundation makes further use of the VMM's snapshotting facility to track daily changes in the disk image as illustrated in Figure 2.

Between snapshots, the VM archiver keeps VMware in a state where the bottom disk image on the stack corresponds to the last archived snapshot (say, snapshot *k*), with VMware recording writes since that snapshot in a partial disk image. To take and archive snapshot *k*+1, the VM archiver takes another VMware snapshot, causing VMware to push a new partial disk image onto the stack. The VM archiver then archives only those blocks written to the now read-only partial disk image for snapshot *k*. Once those blocks have been saved, the VM archiver directs VMware to discard snapshot *k*, merging those writes into the base disk image.

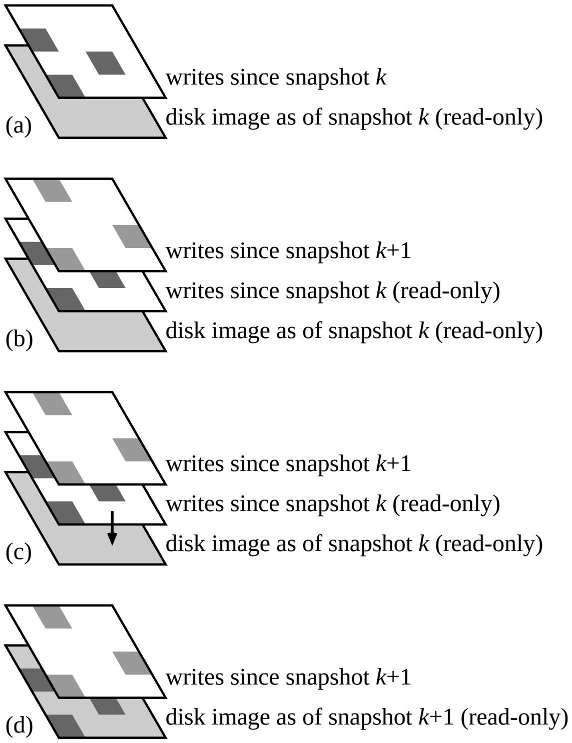
writes since snapshot k
disk image as of snapshot k (read-only)
(a)
writes since snapshot k+1
writes since snapshot k (read-only)
disk image as of snapshot k (read-only)
(b)
writes since snapshot k+1
writes since snapshot k (read-only)
disk image as of snapshot k (read-only)
(c)
writes since snapshot k+1
disk image as of snapshot k+1 (read-only)
(d)

Figure 2: The VMware disk layers when the VM archiver archives disk image snapshot *k*+1. (a) Before the snapshot. The base VMware disk corresponds to snapshot *k*, already archived; since then VMware has been saving disk writes in a partial disk image layered on top of the base image. (b) During the snapshot archival process. The VM archiver directed VMware to create a new snapshot, *k*+1, adding a second partial disk image to the disk stack. The earlier partial disk image contains only the disk sectors that were written between snapshots *k* and *k*+1. The VM archiver saves these using the CAS layer. (c) After the snapshot has been archived. The VM archiver directs VMware to discard snapshot *k*. VMware applies the writes from the corresponding partial disk image to the base disk image and (d) discards the partial disk image.

Using VM snapshots in this way allows Foundation to archive a consistent image of the disk without blocking the user during the archival process. However, because Foundation does not yet use VMware's "SYNC driver" to force the file system into a consistent state before taking a snapshot, the guest OS may need to run a repair process such as *fsck* when the user later boots the image. An alternate approach would archive the machine state and memory as well as the disk, and "resume", rather than boot, old snapshots. We have not yet explored the additional storage costs of this approach.

2.3 CAS Layer

Foundation's CAS layer provides the archival storage service that the VM archiver uses to save VM snapshots. This service provides a simple *read*/*write* interface: passing a disk block to *write* returns a short handle, and *read*, when passed the handle, returns the original block. Internally, the CAS layer coalesces duplicate writes, so that writing the same block multiple times returns the same handle and only stores one copy of the block. Coalescing duplicate writes makes storing many snapshots feasible; the additional storage cost for a new snapshot is proportional only to its new data. The rest of this paper describes the CAS layer in detail.

3 CAS Layer Design

Foundation's CAS layer is modeled on the Venti [34] content-addressed storage server, but we have adapted the Venti algorithms for use in a single-disk system and also optionally eliminated the assumption that SHA-1 is free of collisions, producing two operating modes for Foundation: *compare-by-hash* and *compare-by-value*.

In this section, we first review Venti and then introduce Foundation's two modes. We also discuss the expected disk operations used by each algorithm, since those concerns drove the design.

3.1 Venti Review

The Venti content-addressed storage server provides SHA-1-addressed block storage. When a client writes a disk block, Venti replies with the SHA-1 hash of the block's contents, called a *score*, that can be used to identify the block in future read requests. The storage server provides read/write access to disk blocks, typically ranging in size from 512 bytes up to 32 kilobytes. Venti clients conventionally store larger data streams in hash trees (also known as Merkle trees [29]).

As illustrated in Figure 3, Venti stores blocks in an append-only data log and maintains an index that maps blocks' scores to their offsets in the log. Venti implements this index as a on-disk hash table, where each bucket contains (score, log offset) pairs for a subsection of the 160-bit score space. Venti also maintains two write-through caches in memory: the *block cache* maps blocks' scores to the blocks' values, and the *index cache* maps blocks' scores to the blocks' log offsets.

Figure 4(a) gives pseudocode for the Venti read and write operations. To satisfy a read of a block with a given score, Venti first looks in the block cache. If the block is not found in the block cache, Venti looks up the block's offset in the log, first checking the index cache and then the index itself. If Venti finds a log offset for the block, it reads the block from the log and returns the block. Otherwise, it returns an error (not shown). Writes are handled similarly. Venti first checks to see if it has an existing offset for the block using the two in-memory caches and then the index, returning immediately if so. Otherwise, it appends the block to the log and updates its index and caches before returning.

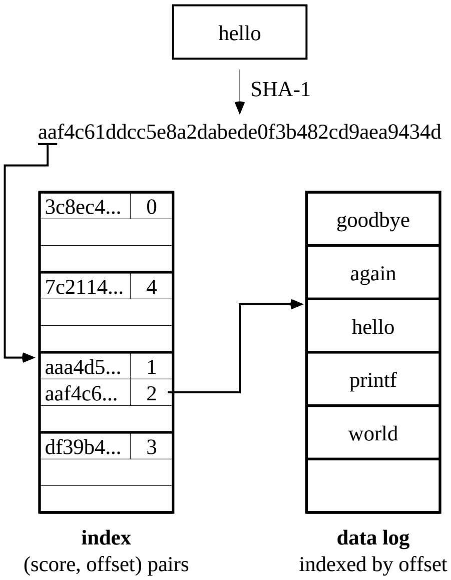

Figure 3: Venti's on-disk data structures. The SHA-1 hash of a data block produces a *score*, the top bits of which are used as a bucket number in the index. The index bucket contains an index entry—a (score, offset) pair—indicating the offset of the corresponding block in the append-only data log.

Note that Venti must read at least one block of its index to satisfy a read or write that misses in both the block and index caches. Because blocks' scores are essentially random, each such operation necessitates at least one seek to read the index. In a single-disk system, these seeks limit throughput to *block size*/*seek time*. The Venti prototype striped its index across eight dedicated, high-speed disks so that it could run eight times as many seeks at once.

3.2 Foundation: Compare-by-Hash Mode

While Venti was designed to provide archival service to many computers, Foundation is aimed at individual consumers and cannot afford multiple disks to mask seek latency. Instead, Foundation stores both its archive and index on a a single, inexpensive USB hard drive and uses additional caches to improve archival throughput.[2]

In compare-by-hash mode, Foundation optimizes for two request types: sequential reads (reading blocks in the order in which they were originally written) and fresh writes (writing new blocks).

Foundation stores its log as a collection of 16 MB *arenas* and stores for each arena a separate *summary* file that lists all of the (score, offset) pairs the arena contains.[3] To

[2] An alternative approach—storing the index in Flash memory—would eliminate seek cost for reads but greatly increase it for writes. Current Flash memories require around 40 ms for random writes.

[3] This design was inspired by Venti's log arenas. We do not know

(a) Venti

```
// Return block named by score.
read(score):
    if(data = blockcache.get(score))
        return data;
    offset = lookupscore(score);
    data = log.read(offset);
    blockcache.put(score, data);
    return data;

// Write data, returning score.
write(data):
    score = SHA1(data);
    if(lookupscore(score))
        return score;
    offset = log.write(data);
    index.write(score, offset);
    indexcache.put(score, offset);
    blockcache.put(score, data);
    return score;

// Return log offset for score.
lookupscore(score):
    if(offset = indexcache.get(score))
        return offset;
    if(offset = index.read(score))
        indexcache.put(score, offset);
        return offset;
    return nil;
```

(b) Foundation: Compare by Hash

```
// Return block named by score.
read(score):
    if(data = blockcache.get(score))
        return data;
    offset = lookupscore(score);
    data = log.read(offset);
    blockcache.put(score, data);
    return data;

// Write data, returning score.
write(data):
    score = SHA1(data);
    if(lookupscore(score))
        return score;
    offset = log.write(data);
    indexbuffer.write(score, offset);
    indexcache.put(score, offset);
    blockcache.put(score, data);
    bloomfilter.put(score);
    return score;

// Return log offset for score.
lookupscore(score):
    if(!bloomfilter.get(score))
        return nil;
    if(offset = indexcache.get(score))
        return offset;
    if(offset = index.read(score))
        sum = log.summary(offset);
        indexcache.put(sum);
        return offset;
    return nil;
```

(c) Foundation: Compare by Value

```
// Read block named by offset.
read(offset):
    if(data = blockcache.get(offset))
        return data;
    // No lookupscore!
    data = log.read(offset);
    blockcache.put(offset, data);
    return data;

// Write data, returning offset.
write(data):
    score = hash(data);
    if(offset = lookupdata(data, score))
        return offset;
    offset = log.write(data);
    indexbuffer.write(score, offset);
    indexcache.put(score, offset);
    blockcache.put(score, data);
    bloomfilter.put(score);
    return offset;

// Return log offset for data.
lookupdata(data, score):
    if(!bloomfilter.get(score))
        return nil;
    for(offset in indexcache.get(score))
        if(read(offset) == data)
            return offset;
    for(offset in index.read(score))
        if(offset in indexcache.get(score))
            continue;
        if(read(offset) == data)
            sum = log.summary(offset);
            indexcache.put(sum);
            return offset;
    return nil;
```

Figure 4: Algorithms for reading and writing blocks in (a) Venti and Foundation's (b) compare-by-hash and (c) compare-by-value modes. Italics in (b) mark differences from (a): the addition of a Bloom filter, the use of a buffer to batch index updates in *write*, and the loading of entire arena summaries into the index cache after a miss in *lookupscore*. Italics in (c) mark differences from (b): the use of log offsets to identify blocks, the use of an insecure hash function to identify potential duplicate writes, the possibility of multiple index entries for a given score, and the need to check existing blocks' contents against new data in *lookupdata*.

take advantage of the spatial locality inherent in sequential reads, each time Foundation reads its on-disk index to find the log offset of some block, it loads and caches the entire summary for the arena that spans the discovered offset. Reading this summary costs an additional seek. This cost pays off in subsequent reads to the same arena, as Foundation finds the log offsets of the affected blocks in the cached summary, avoiding seeks in the on-disk index.

Figure 5 summarizes the costs in disk operations of each path through the pseudocode in Figure 4. In addition to sequential reads and fresh writes, the figure shows costs for out-of-order reads (reading blocks in a different order than that in which they were written), sequential duplicate writes (writing already-written blocks in the same order in which they were originally written), and out-of-order duplicate writes (writing already-written blocks in a different order).

Note that for out-of-order disk reads and for the first disk read in each arena, compare-by-hash mode is slower than Venti, as it performs an additional seek to read the arena summary. In return, Foundation performs subsequent reads at the full throughput of the disk. Section 5 shows that this tradeoff improves overall throughput in real workloads.

whether Venti's design was inspired by the log segments of LFS [36].

(a) Venti	(b) Foundation: by Hash	(c) Foundation: by Value
Out-of-order read		
seek+read index bucket seek+read log block	seek+read index bucket seek+read arena summary seek+read log block	seek+read log block
Cost		
2 seeks + I+L reads	3 seeks + I+L+S reads	1 seek + L reads
Sequential read		
same as out-of-order	if(first block in arena) seek+read index bucket seek+read arena summary seek to log block read log block	if(first block in arena) seek to log block read log block
Cost		
2 seeks + I+L reads	$(1/A) \times$ (3 seeks + I+S reads) + L reads	$(1/A) \times$ 1 seek + L reads
Out-of-order duplicate write		
seek+read index bucket	seek+read index bucket seek+read arena summary	seek+read index bucket $(C+1) \times$ seek+read log block seek+read arena summary
Cost		
1 seek + I reads	(2 seeks + I+S reads)	(C+3 seeks + I+(C+1)L+S reads)
Out-of-order duplicate write — index entry cached		
no disk operations	no disk operations	seek + read log block
Cost		
none	none	1 seek + L reads
Sequential duplicate write		
same as out-of-order	if(first block in arena) same as out-of-order	if(first block in arena) same as out-of-order else read log block
Cost		
1 seek + I reads	$(1/A) \times$ (2 seeks + I+S reads)	$(1/A) \times$ (C+3 seeks + I+(C+1)L+S reads) + (1-1/A) $\times$ L reads
Fresh write		
seek+read index bucket seek+write log block seek+write index bucket	if(Bloom filter false positive) seek+read index bucket seek to end of log write log block if(index buffer full) flush index buffer	if(Bloom filter false positive) seek+read index bucket $C \times$ seek+read log block seek to end of log write log block if(index buffer full) flush index buffer
Cost		
3 seeks + I reads + L+I writes	$B \times$ (2 seeks + I reads) + L writes + $(1/W) \times$ 1 index buffer flush	$B \times$ (C+2 seeks + I+CL reads) + L writes + $(1/W) \times$ 1 index buffer flush

Figure 5: Disk operations required to handle the five different read/write cases. A is the number of blocks per arena, B is the probability of a Bloom filter false positive, C is the probability of a hash collision, I is the size of an index bucket, L is the size of a log data block, S is the size of an arena summary, and W is the size of the write buffer in index entries.

On fresh writes, Venti performs three seeks: one to read the index and determine the write is fresh, one to append the new block to the log, and one to update the index with the block's log offset (see Figure 5).

Foundation eliminates the first of these three seeks by maintaining an in-memory Bloom filter [6] summarizing the all of the scores in the index. A Bloom filter is a randomized data structure for testing set membership. Using far less memory than the index itself, the Bloom filter can check whether a given score is in the index, answering either "probably yes" or "definitely no". A "probably yes" answer for a score that is *not* in the index is called a *false positive*. Using enough memory, the probability of a false positive can be driven arbitrarily low. (Section 4 discuses sizing of the Bloom filter.) By first checking the in-memory Bloom filter, Foundation determines that a write is fresh without reading the on-disk index in all but a small fraction of these writes.

By buffering index updates, Foundation also eliminates the seek Venti performs to update the index during a fresh write. When this buffer fills, Foundation applies the updates in a single, sequential pass over the index. Fresh writes thus proceed in two phases: one phase writes new data to the log and fills the index update buffer; a second phase flushes the buffer. During the first phase, Foundation performs no seeks within the index; all disk writes sequentially append to the end of the log. In return, it must occasionally pause to flush the index update buffer; Section 5 shows that this tradeoff improves overall write throughput in real workloads.

3.3 Foundation: Compare-by-Value Mode

In compare-by-value mode, Foundation does not assume that SHA-1 is collision-free. Instead, it names blocks by their log offsets, and it uses the on-disk index only to identify *potentially* duplicate blocks, comparing each pair of potential duplicates byte-by-byte.

While we originally investigated this mode due to (in our opinion, unfounded) concerns about cryptographic hash collisions (see [5, 16] for a lively debate), we were surprised to find that its overall write performance was close to that of compare-by-hash mode, despite the added comparisons. Moreover, compare-by-value is *always* faster for reads, as naming blocks by their log offsets completely eliminates index lookups during reads.

The additional cost of compare-by-value mode can be seen in the *lookupdata* function in Figure 4(c). For each potential match Foundation finds in the index cache or the index itself, it must read the corresponding block from the log and perform a byte-by-byte comparison.

For sequential duplicate writes, Foundation reads the blocks for these comparisons sequentially from the log. Although these reads consume disk bandwidth, they require a seek only at the start of each new arena. For out-of-order duplicate writes, however, the relative cost of compare-by-value is quite high. As shown in Figure 5, Venti and compare-by-hash mode complete out-of-order duplicate writes without any disk activity at all, whereas compare-by-value mode requires a seek per write.

On the other hand, hash collisions in compare-by-value mode are only a performance problem (as they cause additional reads and byte-by-byte comparisons), not a correctness one. As such, compare-by-value mode can use smaller, faster (and less secure) hash functions than Venti and compare-by-hash. Our prototype, for example, uses the top four bytes of an MD4 hash to select an index block, and stores the next four bytes in the block itself. Using four bytes is enough to make collisions within an index block rare (see Section 4). It also increases the number of entries that fit in the index write buffer, making flushes less frequent, and decreases the index size, making flushes faster when they do occur. Both changes improve the performance of fresh writes.

Section 5 presents a detailed performance comparison between Venti and Foundation's two modes.

3.4 Compare-by-Hash vs. Compare-by-Value

It is worth asking what other disadvantages, other than decreased write throughput, compare-by-value incurs in naming blocks by their log offsets.

The Venti paper lists five benefits of naming blocks by their SHA-1 hashes: (1) blocks are immutable: a block cannot change its value without also changing its name; (2) writes are idempotent: duplicate writes are coalesced; (3) the hash function defines a universal name space for block identifiers; (4) clients can check the integrity of data returned by the server by recomputing the hash; and (5) the immutability of blocks eliminates cache coherence problems in a replicated or distributed storage system.

Benefits (1), (2), and (3) apply also to naming blocks by their log offsets, as long as the log is append-only. Log writes are applied at the client in Foundation—the remote storage service is merely a secondary replica—so (5) is not an issue. Foundation's compare-by-value mode partially addresses benefit (4) by cryptographically signing the log, but naming blocks by their hashes, as in compare-by-hash mode, still provides a more end-to-end guarantee.

Our own experience with Venti also provides one obscure, but interesting case in which naming blocks by their SHA-1 hashes provides a small but tangible benefit. A simultaneous failure of both the backup disk and the remote replica may result in the loss of some portion of the log, after which reads for the lost blocks will fail. In archiving the user's current virtual machine, however, Foundation may encounter many of the lost blocks. When it does so, it will append them to the log as though they were new, but because it names them by their SHA-1 hashes, they will have the same names they had before the failure. As such, subsequent reads for the blocks will begin succeeding again. In essence, archiving current data can sometimes "heal" an injured older archive. We have used this technique successfully in the past to recover from corrupted Venti archives.

4 Implementation

The Foundation prototype consists of just over 14,000 lines of C++ code. It uses VMware's VIX library [3] to take and delete VM snapshots. It uses GNU parted, libext2, and libntfs to read interpret disk images for export in the `/snapshot` tree.

The CAS layer stores its arenas, arena summaries, and index on an external USB hard disk. To protect against loss of or damage to this disk, the CAS layer can be configured to replicate the log arenas over FTP using libcurl.

Providers such as `dot5hosting.com` currently lease remote storage for as little as $5/month for 300 GB of space. While this storage may not be as reliable as that offered by more expensive providers, we suspect that fault-tolerance obtained through the combination of one local and one remote replica is sufficient for most users' needs. The CAS layer does not replicate the arena summaries or index, as it can recreate these by scanning the log.

While users may trust such inexpensive storage providers as a secondary replica for their data, they are less likely to be comfortable entrusting such providers with the contents of their most private data. Moreover, the external hard drive on which Foundation stores its data might be stolen. The CAS layer thus encrypts its log arenas to protect users' privacy, and it cryptographically signs the arenas to detect tampering.

For good random-access performance, our implementation uses a hierarchical HMAC signature and AES encryption in counter mode [11] to sign and encrypt arenas. The combination allows Foundation to read, decrypt, and verify each block individually (i.e., without reading, decrypting, and verifying the entire arena in which a block resides). Foundation implements its hierarchical HMAC and counter-mode AES cipher using the OpenSSL project's implementations of AES and HMAC. (It also uses OpenSSL's SHA-1 and MD4 implementations to compute block hashes.)

Foundation uses the file system in user-space (FUSE) library to export its `/snapshot` tree interface to the host OS. The guest OS then mounts the host's tree using the SMB protocol. To provide the archived disk images for booting under VMware, Foundation uses a loopback NFS server to create the appearance of a complete VMware virtual machine directory, including a `.vmx` file, the read-only disk image, and a `.vmdk` file that points to the read-only image as the base disk while redirecting new writes to an initially empty snapshot file.

By default, the prototype uses a 192 MB index cache—with 128 MB reserved for buffering index writes and the remaining 64 MB managed in LRU order—and a 1 MB block cache. It also caches 10 arena summaries in LRU order, using approximately 10 MB more memory. The prototype stores index entries with 6 bytes for the log offset, 20 bytes for the score in compare-by-hash mode, and 4 bytes for the score in compare-by-value mode. It sizes the index to average 90% full for a user-configurable expected maximum log size. In compare-by-hash mode, a 100 GB log yields a 5.6 GB index. The same log yields a 2.2 GB index in compare-by-value mode. The prototype relocates index block overflow entries using linear probing. It sizes its Bloom filter such that half its bits will be set when the log is full and lookups see a 0.1% false positive rate. For a 100 GB log, the Bloom filter consumes 361 MB of memory. To save memory, the prototype loads the Bloom filter only during the nightly archival process; it is not used during read-only operations such as booting an image or mounting the `/snapshot` tree.

Currently, the Foundation prototype uses 512-byte log blocks to maximize alignment between data stored from different file systems. Using a 512-byte block size also aligns blocks with file systems within a disk, as the master boot record (MBR), for example, is only 512-bytes long, and the first file system usually follows the MBR directly. That said, per-block overheads are an significant factor in Foundation's performance, so we are considering increasing the default block size to 4 kB (now the default for most file systems) and handling the MBR as a special case.

5 Evaluation

To evaluate Foundation, we focus on the performance of saving and restoring VM snapshots, which corresponds directly to the performance of the CAS layer.

The most important performance metric for Foundation is how long it takes to save the VM disk image each night. Many users suspend or power down their machines at night; a nightly archival process that makes them wait excessively long before doing so is a barrier to adoption. (We envision that snapshots are taken automatically as part of the shutdown/sleep sequence.) We are also concerned with how long it takes to boot old system images and recover old file versions from the `/snapshot` tree, though we expect such operations to be less frequent than nightly backups, so their performance is less critical.

We evaluate Foundation's VM archiver in two experiments. First, we analyze the performance of the CAS layer on microbenchmarks in three ways: using the disk operation counts from Figure 5, using a simulator we wrote, and using Foundation itself. These results give insight into Foundation's performance and validate the simulator's predictions. Second, we measure Foundation's archival throughput under simulation on sixteen months of nightly snapshots using traces derived from our research group's own backups.

In both experiments, we compare Foundation in compare-by-hash and compare-by-value mode with a third mode that implements the algorithms described in the Venti paper. Making the comparison this way rather than using the original Venti software allows us to compare the algorithms directly, without worrying about other variables, such as file system caches, that would be different between Foundation and the actual Venti. (Although we do not present the results here, we have also implemented Foundation's compare-by-hash improvements in Venti itself and obtained similar speedups.)

5.1 Experimental setup

We ran our experiments on a Lenovo Thinkpad T60 laptop with a 2 GHz Intel Core 2 Duo Processor and 2 GB of

	Expected Throughput (kB/s)			Actual Throughput (kB/s)		
	Venti	Foundation		Venti	Foundation	
		By-Hash	By-Value		By-Hash	By-Value
out-of-order read	18	7.4	37	15	4.8	19
sequential read	18	29,000	33,000	76	13,000	16,000
out-of-order duplicate write	36	9.2	7.4	79	6.0	5.2
index entry cached	39,000	39,000	37	22,000	22,000	19
sequential duplicate write	36	39,000	29,000	78	23,000	16,000
fresh write	12	4,000	8,100	37	3,800	7,100
without write buffer flush	n/a	11,000	11,000		7,900	8,400

Figure 6: Predicted and actual sustained performance, in MB/s, of the three systems on the cases listed in Figure 5 using the hardware described in Section 5.1. The actual performance of our Venti implementation is faster than predicted, because operating system readahead eliminates some seeks. The actual performance of Foundation is slightly slower than predicted because of unmodeled per-block overheads: using a 4096-byte block size (instead of 512 bytes) matches predictions more closely.

RAM. The laptop runs Ubuntu 7.04 with a Linux 2.6.20 SMP kernel. The internal hard disk is a Hitachi Travelstar 5K160 with an advertised 11 ms seek time and 64 MB/s sustained read/write throughput, while the external disk is a 320 GB Maxtor OneTouch III with an advertised 9 ms seek time and 33 MB/s sustained read/write throughput.

Since Foundation uses both disks through the host OS's file system, we measured their read and write throughput through that interface using the Unix `dd` command. For read throughput, we copied a 2.2 GB file to `/dev/null`; for write throughput, we copied 2.2 GB of `/dev/zero` into a half-full partition. The Hitachi sustained 38.5 MB/s read and 32.2 MB/s write throughput; the Maxtor sustained 32.2 MB/s read and 26.5 MB/s write throughput.

To measure average seek time plus rotational latency through the file system interface, we wrote a small C program that seeks to a random location within the block device using the `lseek` system call and reads a single byte using the `read` system call. In 1,000 such "seeks" per drive, we measured an average latency of 15.0 ms on the Hitachi and 13.6 ms on the Maxtor. The system was otherwise idle during both our throughput and seek tests.

The simulator uses the disk speeds we measured and the same parameters (cache sizes, etc.) as our implementation. Rather than store the Bloom filter directly, it assumes a 0.1% probability of a false positive.

5.2 Microbenchmarks

To understand Foundation's performance, we consider the disk operations required for each of the six read or write cases shown in Figure 5. For each case, we count the number of seeks and the amount of data read from and written to the disk. From these and the disk parameters measured and reported above, we compute the speed of each algorithm in each case. Figure 6 shows the predicted performance and the performance of the prototype. (The simulated performance matches the predictions made using the equations in Figure 5 exactly.)

In both prediction and in reality, compare-by-hash is significantly faster than Venti for sequential accesses, at the cost of slowing out-of-order accesses, which load arena summaries that end up not being useful. Compare-by-value reads faster than compare-by-hash, since it avoids the index completely, but it handles duplicate writes slower, since it must compare each potential duplicate to previously-written data from the log.

The most dramatic difference between compare-by-hash and compare-by-value is the case of an out-of-order duplicate write for which the index entry cache has a corresponding record, but the block cache does not. In this case, Venti and compare-by-hash can declare the write a duplicate without any disk accesses, while compare-by-value must load the data from disk, resulting in dramatically lower throughput. (The throughput for Venti and compare-by-hash is limited only by the bandwidth of the local disk in this case.)

Sequential duplicate writes are fast in both Foundation modes. In compare-by-hash mode, Foundation is limited by the throughput of the local disk containing the snapshot. The arena summaries needed from the external disk are only 5% the size of the snapshot itself. In compare-by-value mode, Foundation must read the snapshot from the local disk and compare it again previously-written data from the log disk. Having two disks arms here is the key to good performance: on a single-disk system the performance would be hurt by seeks between the two streams.

Fresh writes proceed at the same speed in both Foundation modes except for the index buffer flushes. Because index entries are smaller in compare-by-value mode, the 128 MB buffer holds more entries and needs to be flushed less frequently: after every 4 GB of fresh writes rather than every 2.3 GB. At that rate, index flushes are still an important component of the run time. Using a larger buffer size or a larger data block size would reduce the flush frequency, making the two modes perform more similarly.

The predictions match Foundation's actual performance to within a factor of 2.25, and the relative orderings are all the same. Foundation is slower than predicted because the model does not account for time spent encrypting, signing, verifying, and decrypting the log; time spent

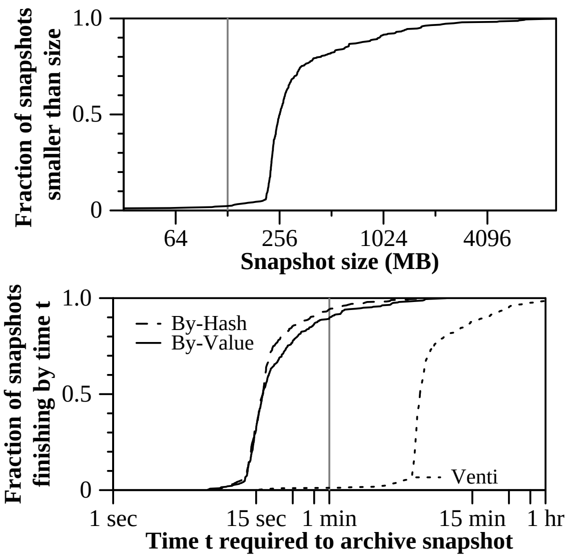

Figure 7: Distribution of sizes and write times for 400 nightly snapshots of one of our research group's home directory disks.

compressing and decompressing blocks; and constant (per 512-byte block) overheads in the run-time system. Using 4096-byte blocks and disabling encryption, compression, and authentication yields performance that matches the predictions more accurately.

5.3 Trace-driven Simulation

We do not yet have long-term data from using Foundation, but as mentioned earlier, our research group takes nightly physical backups of its central file server using a 15-disk Venti server. The backup program archives entire file system images, using the file system block size as the archival block size. We extracted over a year of block traces from each of the file server's 10 disks. These traces contain, for each night, a list of the disk blocks changed from the previous night, along with the blocks' SHA-1 hashes. We annotated each trace with the data log offsets each block would have been stored at if data from the disk were the only data in a Venti or Foundation server. We then ran the traces in our simulator to compare the two Foundation operating modes and the Venti mode.

To conserve space, we discuss the results from only one of the traces here. The relative performance of the three algorithms, however, is consistent across traces. The disk for the trace we chose hosts the home directories of four users. The trace covers 400 days. When the trace starts, the disk has 41.7 GB of data on it; when the trace ends, the disk has 69.9 GB of data on it. The parameters for the simulation are the same as described in Section 5.1, except that blocks are 32 kB, to match the traces, rather than 512 bytes as in Foundation.

The most important metric is the duration of the nightly snapshot process. Figure 7 plots the distributions of snapshot sizes and completion times. Even though 95% of snapshots are larger than 128 MB, the vast majority of snapshots—90% for compare-by-value and 94% for compare-by-hash—finish in a minute or less.

Figure 8 breaks down the average performance of a snapshot backup. The differences in snapshot speed—849 kB/s for Venti, 20,581 kB/s for compare-by-hash, and 15,723 kB/s for compare-by-value—are accounted for almost entirely by the time spent seeking in the external disk. Foundation's use of the Bloom filter and arena summaries reduces the number of seeks required in compare-by-hash mode by a factor of 240 versus Venti. Compare-by-value mode reintroduces some seeks by reading log blocks to decide that writes are duplicates during lookup.

To access archived data, Foundation users will either use the file system snapshot server or boot an archived VM. In both cases, the relevant parts of disk can be read as needed by the file system browser or the VMM.

In many cases, Foundation can satisfy such reads quickly. Comparing the measured performance of Foundation in Figure 6 with the performance of our test system's internal hard drive, we note that Foundation's compare-by-value mode is only 1.8 times slower for out-of-order reads and 2.2 times slower for sequential reads. However, in eliminating duplicate data during writes, Foundation may introduce additional seeks into future reads, since the blocks of a disk image being read may originally have been stored as part other disk images earlier in the log. We call this problem *fragmentation*.

Unfortunately, we do not have traces of the reads requests serviced by our research group's Venti server, so it is difficult to simulate to what degree users will be affected by such fragmentation in practice. As an admittedly incomplete benchmark, however, we simulate reading entire disk images for each snapshot. We return to the fragmentation problem in Section 7.2.

Figure 9 summarizes the performance of reading full disk images. Again the differences in performance are almost entirely due to disk seeks: 739 minutes seeking for Venti, 41 minutes seeking for compare-by-hash, and 35 minutes seeking for compare-by-value. Since compare-by-value eliminates index lookups during read, its seeks are all within the data log. Such seeks are due to fragmentation, and for compare-by-value mode they account for the entire difference between the predicted performance of sequential reads in Figure 6 with the simulated performance in Figure 9.

In compare-by-value mode, since the block identifiers are log offsets, the reads could be reordered to reduce the amount of seeking. As a hypothetical, the column labeled "Sorted" shows the performance if the block requests were first sorted in increasing log offset. This would cut the total seek time from 35 minutes to 8 minutes, also improving the number of block cache hits by a factor of 24. Although making a list of every block may not be realis-

	Venti	Foundation	
		By-Hash	By-Value
average snapshot write speed	849 kB/s	20,581 kB/s	15,723 kB/s
average snapshot time	648.4 s	26.7 s	35.0 s
... reading external disk	4.1 s	2.2 s	4.5 s
... writing external disk	21.6 s	20.0 s	19.8 s
... seeking in external disk	622.3 s	2.6 s	10.7 s
... waiting for local disk; external disk idle	0.4 s	2.0 s	0.0 s
average # of external disk seeks	46,099	192	792
... to index data	31,415	133	133
... to log data	14,684	58	658
average # of lookup calls; these do ...	17,196	2,526	2,526
... # of index seeks (also # of index reads)	16,731	73	73
... # of log seeks	0	0	442
... # of log reads	0	0	2,482

Figure 8: Statistics gathered while writing 400 nightly snapshots, in simulation. The average snapshot size is 537 MB. Because the VMM identifies which blocks have changed since the previous snapshot, on average only 78.5 MB of blocks are duplicates. There are, however, occasional large spikes of duplicates. 9 of the 400 nights contain over 1 GB of duplicate blocks; 2 contain over 5 GB.

	Venti	Foundation			
		By-Hash	By-Value		
			Unsorted	Sort-1024	Sorted
average disk image restore speed	1,271 kB/s	13,894 kB/s	15,309 kB/s	20,842 kB/s	28,940 kB/s
average disk image restore time	775 min	71 min	64 min	47 min	34 min
... reading external disk	36 min	30 min	30 min	29 min	26 min
... seeking in external disk	739 min	41 min	35 min	18 min	8 min
average # of block cache hits	9,660	9,660	9,660	31,203	232,597
average # of index entry cache hits	222,936	1,824,023	0	0	0
average # of external disk seeks	3,283,167	182,337	153,853	79,495	35,218
... to index data	1,613,802	25,431	0	0	0
... to log data	1,669,365	156,906	153,853	79,495	35,218
average # of external disk reads	3,450,540	1,849,454	1,836,738	1,815,196	1,613,802
... of index data	1,613,802	12,716	0	0	0
... of log data	1,836,738	1,836,738	1,836,738	1,815,196	1,613,802

Figure 9: Statistics gathered while reading disk images of 400 nightly snapshots, in simulation. The average disk image is 56 GB.

tic, a simple heuristic can realize much of the benefit. The column labeled “Sort-1024” shows the performance when 1024 reads at a time are batched and sorted before being read from the log. This simple optimization cuts the seek time to 18 minutes, while still improving the number of block cache hits by a factor of 3.2.

6 Related Work

Related Work in Preservation Most preservation work falls into one of two groups. (The following description is simplified somewhat; see Lee et al. [24] for a detailed discussion.) The first group (e.g. [12, 14, 37, 41]) proposes archiving a limited set of popular file formats such as JPEG, PDF, or PowerPoint. This restriction limits the digital artifacts that can be preserved to those than can be encoded in a supported format. In contrast, Foundation preserves both the applications and configuration state needed to view both popular and obscure file formats.

In the case that a supported format becomes obsolete, this first group advocates automated “format migration”, in which files in older formats are automatically converted to more current ones. Producing such conversion routines can be difficult: witness PowerPoint’s inability to maintain formatting between its Windows and Mac OS versions. Furthermore, perfect conversion is sometimes impossible, as between image formats that use lossy compression. Rather than migrate formats forward in time, Foundation enables travel back in time to the environments in which old formats can be interpreted.

The second group of preservationists (e.g. [15, 38]) advocates emulating old hardware and/or operating systems in order to run the original applications with which users viewed older file formats. Foundation uses emulation, but recognizes that simply preserving old applications and operating systems is not enough. Often, the rendering of a digital artifact is dependent on configuration state, optional shared libraries, or particular fonts. A default Firefox installation, for example, may not properly display a web page that contains embedded video, non-standard fonts, or Flash animations. Foundation captures all such state by archiving full disk images, but it limits the hardware that must be emulated to boot such images by confining users' daily environments within a VM.

An offshoot of the emulation camp proposes the construction of emulators specifically for archival purposes. Lorie proposed [27] storing with each digital artifact a program for interpreting the artifact; he further proposed that such programs be written in the language of a Universal Virtual Computer (UVC) that can be concisely specified and for which future emulators are easy to construct. Ford has proposed [13] a similar approach, but using an x86 virtual machine with limited OS support as the emulation platform. Foundation differs from these two systems in that it archives files with the same OS kernel and programs originally used to view them, rather than require the creation of new ones specific to archival purposes.

Internet Suspend/Resume (ISR) [22] and Machine Bank [43] use a VM to suspend a user's environment on one machine and resume it on another. SecondSite [10] and Remus [9] allow resumption of services at a site that suffers a power failure by migrating the failed site's VMs to a remote site. Like these systems, Foundation requires that a user's environment be completely contained within a VM, but for a different purpose: it allows the "resumption" of state from arbitrarily far in the past.

Related Work in Storage Hutchinson et al. [19] demonstrated that physical backup can sustain higher throughput than logical backup, but noted several problems with physical backup. First, since bits are not interpreted as they are backed up, the backup is not portable; Foundation provides portability by booting the entire image in an emulator. Second, it is hard to restore only a subset of a physical backup; Foundation interprets file system structures to provide the `/snapshot` tree, allowing users to recover individual files using standard file system tools. Third, obtaining a consistent image is difficult; Foundation implements copy-on-write within the VMM to do so, but other tools, such as the Linux's Logical Volume Manager (LVM) [25] could be used instead. Finally, incremental backups are hard; addressing blocks by their hashes as in Venti solves this problem.

The SUNDR secure network file system [26] also uses a Venti-like content-addressed storage server but uses a different solution than Founation to reduce index seeks. SUNDR saves all writes in a temporary disk buffer without deciding whether they are duplicate or fresh and then batches both the index searches to determine freshness and the index updates for the new data. Foundation avoids the temporary data buffer by using the Bloom filter to determine freshness quickly, buffering only the index entries for new writes, and never the content.

Microsoft Single-Instance Store (SIS) [7] identifies and collates files with identical contents within a file system, but rather than coallescing duplicates on creation, SIS instead finds them using a background "groveler" process.

A number of past file systems have provided support for sharing blocks between successive file versions using copy-on-write (COW) [17, 31, 39, 42]. These systems capture duplicate blocks between versions of the same file, but they fail to identify and coalesce duplicate blocks that enter the file system through different paths—as when a user downloads a file twice, for example. Moreover, they cannot coalesce duplicate data from multiple, distinct file systems; a shared archival storage server built on such systems would not be as space-efficient as one built on CAS.

Peabody [20] implements time travel at the disk level, making it possible to travel back in time to any instant and get a consistent image. Chronus [45] used Peabody to boot old VMs to find a past configuration error. Peabody uses a large in-memory content-addressed buffer cache [21] to coalesce duplicate writes. Because it only looks in the buffer cache, it cannot guarantee that all duplicate writes are coalesced. In contrast, Foundation is careful to find all duplicate writes.

LBFS [30] chooses block boundaries according to blocks' contents, rather than using a fixed block size, in order to better capture changes that shift the alignment of data within a file. Foundation is agnostic as to how block boundaries are chosen and could easily be adapted to do the same.

Time Machine [40] uses incremental logical backup to store multiple versions of a file system. It creates the first backup by logically mirroring the entire file system tree onto a remote drive. For each subsequent backup, Time Machine creates another complete tree on the remote drive, but it uses hard links to avoid re-copying unchanged files. Unlike Foundation, then, Time Machine cannot efficiently represent single-block differences. Even if a file changes in only one block, Time Machine creates a complete new copy on the remote drive. The storage cost of this difference is particularly acute for applications such as Microsoft Entourage, which stores a user's complete email database as a single file.

7 Future Work

The Foundation CAS layer is already a fully functioning system; it has been in use as one author's only backup strategy for six months now. In using it on a daily basis, however, we have discovered two interesting areas for future work: storage reclamation and fragmentation.

7.1 Storage Reclamation

Both the Plan 9 experience and our own experience with Venti seem to confirm our hypothesis that, in practice, content-addressed storage is sufficiently space-efficient that users can retain nightly disk snapshots indefinitely.

Nonetheless, it is not difficult to imagine usage patterns that would quickly exhaust the system's storage. Consider, for example, a user that rips a number of DVDs onto a laptop to watch during a long business trip, but shortly afterwards deletes them. Because the ripped DVDs were on the laptop for several nights, Foundation is likely to have archived them, and they will remain in the user's archive. After a number of such trips, the archive disk will fill.

One solution to this problem would allow users to selectively delete snapshots. This solution is somewhat risky, in that a careless user might delete the only snapshot that is able to interpret a valued artifact. We suspect that users would be even more frustrated, however, by having to add disks to a system that was unable to reclaim space they felt certain was being wasted.

Like Venti, Foundation encodes the metadata describing which blocks make up a snapshot as a Merkle tree and stores interior nodes of this tree in the CAS layer. To simplify finding a particular snapshot within the log, Foundation also implements a simple *system catalog* as follows. After writing a snapshot, Foundation writes the root of the snapshot's Merkle tree along with the time at which it took the snapshot to a file that it then archives in the CAS layer. It repeats this process after writing each subsequent snapshot, appending the new snapshot's root and time to the existing list and re-archiving the list. The last block in Foundation's log is thus always the root of the latest version of the system catalog.

Conceptually, deleting a snapshot resembles garbage collection in programming languages or log cleaning in LFS. First, the CAS layer writes a new version of the system catalog that no longer points to the snapshot. Then, the system reclaims the space used by blocks that are no longer reachable from any other catalog entry. A more recent snapshot, for example, may still point to some block in the deleted snapshot.[4]

Interestingly, the structure of Foundation's log makes identifying unreferenced blocks particularly efficient: as a natural result of the log being append-only, all pointers within the log point "backwards". Garbage collection can thus proceed in a single, sequential pass through the log using an algorithm developed by Armstrong and Virding for garbage collecting immutable data structures in the Erlang programming language [4].

The algorithm works as follows. Starting at the most recent log entry and scanning backwards, it maintains a list of "live" blocks initialized from the pointers in the system catalog. Each time it encounters a live block, it deletes that block from its list. If the block is a metadata block that contains pointers to other blocks, it adds these pointers to its list. If the algorithm encounters a block that is not in its list, then there are no live pointers to that block later in the log, and since all pointers point backwards, the algorithm can reclaim the block's space immediately. The system can also use this algorithm incrementally: starting from the end of the log, it can scan backward until "enough" space has been reclaimed, and then stop.

The expensive part of the Erlang algorithm is maintaining the list of live blocks. If references to many blocks occur much later in the log than the blocks themselves, this list could grow too large to fit in memory. We note, however, that a conservative version of the collector could use a Bloom filter to store the list of live blocks. Although false positives in the filter would prevent the algorithm from reclaiming some legitimate garbage, its memory usage would be fixed at the size of the Bloom filter.

Finally, to reclaim the space used by an unreferenced block, Foundation can simply rewrite the log arena in which the block occurs without the block, using an atomic rename to replace the old arena. Because this rewriting shifts the locations of other blocks in the arena, an extra pass is required in compare-by-value mode, where blocks' names are their locations in the log: the system must scan from the rewritten arena to the tail of the log, rewriting pointers to the affected arena as it goes. In compare-by-hash mode, however, blocks' names are independent of their locations in the log, so no extra pass is required.

7.2 Fragmentation

Most of our current work on the Foundation CAS layer has focused on reducing the number of seeks within the index. Having done so, however, we have noticed a potential secondary bottleneck: seeks within the data log itself. Consider the case of an archived snapshot made up of one block from each of all of the arenas in the log. Even if no seeks were required to determine the location of the snapshot's blocks, reading the snapshot would still incur one seek (into the appropriate arena) per block.

We have come to call this problem *fragmentation*. We have not yet studied the sources of fragmentation in detail. In our experience so far it is a visible problem, but not a serious one. We simply see some slowdown in reading later versions of disk images as they evolve over time.

Unfortunately, unlike the seeks within the system's index, seeks due to fragmentation cannot be eliminated; they are a fundamental consequence of coalescing duplicate writes (the source of Foundation's storage efficiency). We suspect that it also exists in file systems that perform copy-on-write snapshots, such as WAFL [17], although we have not found any reference to it in the literature.

We do note that fragmentation can be eliminated in any one snapshot, at the expense of others, by copying all of the blocks of that snapshot into a contiguous region of the log. If the system also removes the blocks from their original locations, this process resembles the "defragmentation" performed by a copying garbage collector. We are thus considering implementing within Foundation a version of the Erlang algorithm discussed above that reclaims

[4] Here Foundation differs from LFS, which collects all blocks not pointed to by the most recent version.

space by copying live data, rather than deleting dead data, in order to defragment more recently archived (and presumably, more frequently accessed) snapshots.

One other potential motivation for defragmenting more recent snapshots in this manner is that it will likely improve the write throughput of compare-by-value mode, since the blocks it compares against while writing are unlikely to change their ordering much between snapshots.

8 Conclusion

Foundation's approach to preservation—archiving consistent, nightly snapshots of a user's entire hard disk—is a straight-forward, application-independent approach to automatically capturing all of a user's digital artifacts and their associated software dependencies. Archiving these snapshots using content-addressed storage keeps the system's storage cost proportional to the amount of new data users create and eliminates duplicates that file-system-based techniques, such as copy-on-write, would miss. Using the techniques described in this paper, CAS achieves high throughput on remarkably modest hardware—a single USB hard disk—improving on the read and write throughput achieved by an existing, state-of-the-art CAS system on the same hardware by an order of magnitude.

9 Acknowledgments

This work benefits from useful discussions with Eric Brewer, Greg Ganger, David Gay, William Josephson, Michael Kaminsky, Jinyang Li, and Petros Maniatis.

References

[1] Amazon simple storage service (S3). `http://www.amazon.com/gp/browse.html?node=16427261`, 2007.

[2] VMware virtual machine disk format (VMDK) specification. `http://www.vmware.com/interfaces/vmdk.html`, 2007.

[3] VMware VIX API. `http://www.vmware.com/support/developer/vix-api/`, 2007.

[4] J. Armstrong and R. Virding. One pass real-time generational mark-sweep garbage collection. In *Intl. Workshop on Memory Management*, 1995.

[5] J. Black. Compare-by-hash: A reasoned analysis. In *USENIX Annual Tech. Conf.*, 2006.

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

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

[8] R. Chen. Getting out of DLL Hell. *Microsoft TechNet*, Jan. 2007.

[9] B. Cully et al. Remus: High availability via asynchronous virtual machine replication. In *NSDI*, 2008.

[10] B. Cully and A. Warfield. SecondSite: disaster protection for the common server. In *HotDep*, 2006.

[11] M. Dworkin. Recommendation for block cipher modes of operation: Methods and techniques. *NIST Special Publication 800-38A*, Dec. 2001.

[12] P. Festa. A life in bits and bytes (inteview with Gordon Bell). http://news.com.com/2008-1082-979144.html, Jan. 2003.

[13] B. Ford. VXA: A virtual architecture for durable compressed archives. In *FAST*, 2005.

[14] J. Gemmell, G. Bell, and R. Lueder. MyLifeBits: a personal database for everything. *CACM*, 49(1):88–95, Jan. 2006.

[15] S. Granger. Emulation as a digital preservation strategy. *D-Lib Magazine*, 6(10), Oct. 2000.

[16] V. Henson. An analysis of compare-by-hash. In *HotOS*, 2003.

[17] D. Hitz, J. Lau, and M. Malcolm. File system design for an NFS file server appliance. In *USENIX Winter Conf.*, 1994.

[18] J. Hollingsworth and E. Miller. Using content-derived names for configuration management. In *ACM SIGSOFT Symposium on Software Reusability*, 1997.

[19] N. Hutchinson et al. Logical vs. physical file system backup. In *OSDI*, 1999.

[20] C. B. M. III and D. Grunwald. Peabody: The time travelling disk. In *MSST*, 2003.

[21] C. B. M. III and D. Grunwald. Content based block caching. In *MSST*, 2006.

[22] M. Kozuch and M. Satyanarayanan. Internet Suspend/Resume. In *WMCSA*, 2002.

[23] T. Kuny. A digital dark ages? Challenges in the preservation of electronic information. In *63rd IFLA General Conference*, 1997.

[24] K.-H. Lee, O. Slattery, R. Lu, X. Tang, and V. McCrary. The state of the art and practice in digital preservation. *Journal of Research of the NIST*, 107(1):93–106, Jan–Feb 2002.

[25] A. Lewis. LVM HOWTO. `http://tldp.org/HOWTO/LVM-HOWTO/`, 2006.

[26] J. Li, M. Krohn, D. Mazieres, and D. Shasha. Secure untrusted data repository (SUNDR). In *OSDI*, 2004.

[27] R. Lorie. A methodology and system for preserving digital data. In *ACM/IEEE Joint Conf. on Digital Libraries*, 2002.

[28] C. Marshall, F. McCown, and M. Nelson. Evaluating personal archiving strategies for Internet-based information. In *IS&T Archiving*, 2006.

[29] R. Merkle. A digital signature based on a conventional encryption function. In *CRYPTO*, 1988.

[30] A. Muthitacharoen, B. Chen, and D. Mazières. A low-bandwidth network file system. In *SOSP*, Oct. 2001.

[31] OpenSolaris. What is ZFS? `http://opensolaris.org/os/community/zfs/whatis/`, 2007.

[32] R. Pike et al. Plan 9 from Bell Labs. *Computing Systems*, 8(3):221–254, Summer 1995.

[33] S. Quinlan. A cached WORM file system. *Software—Practice and Experience*, 21(12):1289–1299, 1991.

[34] S. Quinlan and S. Dorward. Venti: a new approach to archival storage. In *FAST*, 2002.

[35] T. Reichherzer and G. Brown. Quantifying software requirements for supporting archived office documents using emulation. In *ICDL*, 2006.

[36] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst.*, 10(1):26–52, 1992.

[37] D. Rosenthal, T. Lipkis, T. Robertson, and S. Morabito. Transparent format migration of preserved web content. *D-Lib Magazine*, 11(1), Jan. 2005.

[38] J. Rothenberg. Ensuring the longevity of digital documents. *Scientific American*, 272(1):42–47, Jan. 1995.

[39] D. Santry et al. Deciding when to forget in the Elephant file system. In *SOSP*, 1999.

[40] J. Siracusa. Mac OS X 10.5 Leopard: the Ars Technica review. `http://arstechnica.com/reviews/os/mac-os-x-10-5.ars/14`, 2007.

[41] M. Smith. Eternal bits. *IEEE Spectrum*, 42(7):22–27, July 2005.

[42] C. Soules, G. Goodson, J. Strunk, and G. Ganger. Metadata efficiency in versioning file systems. In *FAST*, 2003.

[43] S. Tang, Y. Chen, and Z. Zhang. Machine Bank: Own your virtual personal computer. In *IPDPS*, 2007.

[44] A. van Hoff, J. Giannandrea, M. Hapner, S. Carter, and M. Medin. The HTTP distribution and replication protocol. Technical Report NOTE-drp-19970825, W3C, 1997.

[45] A. Whitaker, R. S. Cox, and S. D. Gribble. Configuration debugging as search: finding the needle in the haystack. In *OSDI*, 2004.