HydraFS: a High-Throughput File System for the HYDRAstor Content-Addressable Storage System

Cristian Ungureanu, Benjamin Atkin, Akshat Aranya, Salil Gokhale,
Stephen Rago, Grzegorz Całkowski, Cezary Dubnicki, and Aniruddha Bohra

NEC Laboratories America

cristian@nec-labs.com, atkin@google.com, aranya@nec-labs.com, salil@nec-labs.com,
sar@nec-labs.com, grzes@vmware.com, dubnicki@9livesdata.com, abohra@akamai.com

Abstract

A content-addressable storage (CAS) system is a valuable tool for building storage solutions, providing efficiency by automatically detecting and eliminating duplicate blocks; it can also be capable of high throughput, at least for streaming access. However, the absence of a standardized API is a barrier to the use of CAS for existing applications. Additionally, applications would have to deal with the unique characteristics of CAS, such as immutability of blocks and high latency of operations. An attractive alternative is to build a file system on top of CAS, since applications can use its interface without modification.

Mapping a file system onto a CAS system efficiently, so as to obtain high duplicate elimination and high throughput, requires a very different design than for a traditional disk subsystem. In this paper, we present the design, implementation, and evaluation of HydraFS, a file system built on top of HYDRAstor, a scalable, distributed, content-addressable block storage system. HydraFS provides high-performance reads and writes for streaming access, achieving 82–100% of the HYDRAstor throughput, while maintaining high duplicate elimination.

1 Introduction

Repositories that store large volumes of data are increasingly common today. This leads to high capital expenditure for hardware and high operating costs for power, administration, and management. A technique that offers one solution for increasing storage efficiency is data deduplication, in which redundant data blocks are identified, allowing the system to store only one copy and use pointers to the original block instead of creating redundant blocks. Deduplicating storage is ideally suited to backup applications, since they store similar data repeatedly, and with growing maturity is expected to become common in the data center for general application use.

Data deduplication can be achieved *in-line* or *off-line*. In both cases, data is eventually stored in an object store where objects are referenced through addresses derived from their contents. Objects can be entire files, blocks of data of fixed size, or blocks of data of variable size.

In a CAS system with in-line deduplication, the data blocks are written directly to the object store. Thus, they are not written to disk if they are deemed duplicates; instead, the address of the previously written block with the same contents is used. A CAS system with off-line deduplication first saves data to a traditional storage system, and deduplication processing is done later. This incurs extra I/O costs, as data has to be read and re-written, and requires additional storage space for keeping data in non-deduplicated form until the processing is complete.

While a CAS system with in-line deduplication does not have these costs, using it directly has two disadvantages: the applications have to be modified to use the CAS-specific API, and use it in such a way that the best performance can be obtained from the CAS system. To avoid the inconvenience of rewriting many applications, we can layer a file system on top of the object store. This has the advantage that it presents a standard interface to applications, permitting effective use of the CAS system to many applications without requiring changes. Additionally, the file system can mediate between the access patterns of the application and the ones best supported by the CAS system.

Designing a file system for a distributed CAS system is challenging, mainly because blocks are immutable, and the I/O operations have high latency and jitter. Since blocks are immutable, all data structures that hold references to a block must be updated to refer to the new

address of the block whenever it is modified, leading to multiple I/O operations, which is inefficient. Distributed CAS systems also impose high latencies in the I/O path, because many operations must be done in the critical path.

While file systems have previously been built for CAS systems, most have scant public information about their design. One notable exception is LBFS [18], which focuses on nodes connected by low bandwidth, wide area networks. Because of the low bandwidth, it is not targeted at high-throughput applications, and poses different challenges for the file system designers.

This paper describes the design, implementation, and evaluation of HydraFS, a file system layered on top of a distributed, content-addressable back end, HYDRAstor [5] (also called Hydra, or simply the "block store"). Hydra is a multi-node, content-addressable storage system that stores blocks at configurable redundancy levels and supports high-throughput reads and writes for streams of large blocks. Hydra is designed to provide a content-addressable block device interface that hides the details of data distribution and organization, addition and removal of nodes, and handling of disk and node failures.

HydraFS was designed for high-bandwidth streaming workloads, because its first commercial application is as part of a backup appliance. The combination of CAS block immutability, high latency of I/O operations, and high bandwidth requirements brings forth novel challenges for the architecture, design, and implementation of the file system. To the best of our knowledge, HydraFS is the first file system built on top of a distributed CAS system that supports high sequential read and write throughput while maintaining high duplicate elimination.

We faced three main challenges in achieving high throughput with HydraFS. First, updates are more expensive in a CAS system, as all metadata blocks that refer to a modified block must be updated. This metadata comprises mappings between an inode number and the inode data structure, the inode itself, which contains file attributes, and file index blocks (for finding data in large files). Second, cache misses for metadata blocks have a significant impact on performance. Third, the combination of high latency and high throughput requires a large write buffer and read cache. At the same time, if these data structures are allowed to grow without bound, the system will thrash.

We overcome these challenges through three design strategies. First, we decouple data and metadata processing through the use of a log [10]. This split allows the metadata modifications to be batched and applied efficiently. We describe a metadata update technique that maintains consistency without expensive locking. Second, we use fixed-size caches and use admission control to limit the number of concurrent file system operations such that their processing needs do not exceed the available resources. Third, we introduce a second-order cache to reduce the number of misses for metadata blocks. This cache also helps reduce the number of operations that are performed in the context of a read request, thus reducing the response time.

Our experimental evaluation confirms that HydraFS enables high-throughput sequential reads and writes of large files. In particular, HydraFS is able to support sequential writes to a single file at 82–100% of the underlying Hydra storage system's throughput. Although HydraFS is optimized for high-throughput streaming file access, its performance is good enough for directory operations and random file accesses, making it feasible for bulk data transfer applications to use HydraFS as a general-purpose file system for workloads that are not metadata-intensive.

This paper makes the following contributions. First, we present a description of the challenges in building a file system on top of a distributed CAS system. Second, we present the design of a file system, HydraFS, that overcomes these challenges, focusing on several key techniques. Third, we present an evaluation of the system that demonstrates the effectiveness of these techniques.

2 Hydra Characteristics

HydraFS acts as a front end for the Hydra distributed, content-addressable block store (Figure 1). In this section, we present the characteristics of Hydra and describe the key challenges faced when using it for applications, such as HydraFS, that require high throughput.

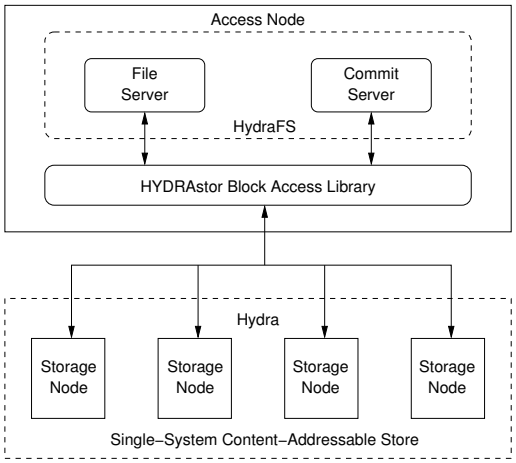

Figure 1: HYDRAstor Architecture.

2.1 Model

HydraFS runs on an *access node* and communicates with the block store using a library that hides the distributed

nature of the block store. Even though the block store is implemented across multiple *storage nodes*, the API gives the impression of a single system.

The HYDRAstor block access library presents a simple interface:

Block write The caller provides a block to be written and receives in return a receipt, called the *content address*, for the block. If the system can determine that a block with identical content is already stored, it can return its content address instead of generating a new one, thus eliminating duplicated data. Multiple resiliency levels are available to control the amount of redundant information stored, thereby allowing control over the number of component failures that a block can withstand.

The block access library used on the access node has the option of querying the storage nodes for the existence of a block with the same hash key to avoid sending the block over the network if it already exists. This is a block store feature, imposing only a slight increase in the latency of the write operations, that is already tolerated by the file system design.

Block read The caller provides the content address and receives the data for the block in return.

Searchable block write Given a block and a *label*, the block is stored and associated with the label. If a block with the same label but different content is already stored, the operation fails. Labels can be any binary data chosen by the client, and need not be derived from the contents of the block.

Two types of searchable blocks are supported: *retention roots* that cause the retention of all blocks reachable from them, and *deletion roots* that mark for deletion the retention roots with the same labels. Periodically, a garbage collection process reclaims all blocks that are unreachable from retention roots not marked for deletion.

Searchable block read Given a label, the contents of the associated retention root are returned.

The searchable block mechanism provides a way for the storage system to be self-contained. In the absence of a mechanism to retrieve blocks other than through their content address, an application would have to store at least one content address *outside* the system, which is undesirable.

2.2 Content Addresses

In HYDRAstor, content addresses are opaque to clients (in this case, the filesystem). The block store is responsible for calculating a block's content address based on a secure, one-way hash of the block's contents and other information that can be used to retrieve the block quickly.

For the same data contents, the block store can return the same content address, although it is not obliged to do so. For example, a block written at a higher resiliency level would result in a different content address even if an identical block were previously written at a lower resiliency level. The design also allows for a byte-by-byte comparison of newly-written data blocks whose hashes match existing blocks. Collisions (different block contents hashing to the same value) would be handled by generating a different content address for each block. For performance reasons, and given that the hash function is strong enough to make collisions statistically unlikely, the default is to not perform the check.

Because the content address contains information that the file system does not have, it is impossible for the file system to determine the content address of a block in advance of submitting it to the block store. At first blush, since the latency of the writes is high, this might seem like a problem for performance, because it reduces the potential parallelism of writing blocks that contain pointers to other blocks. However, this is not a problem for two reasons. First, even if we were to write all blocks in parallel, we still would have to wait for all child blocks to be persistent before writing the searchable retention root. The interface is asynchronous: write requests can complete in a different order than that in which they were submitted. If we were to write the searchable block without waiting for the children and the system were to crash, the file system would be inconsistent if the retention root made it to disk before all of its children.

Second, the foreground processing, which has the greatest effect on write performance, writes only shallow trees of blocks; the trees of higher depth are written in the background, so the reduction in concurrency is not significant enough to hurt the performance of streaming writes. Thus, although the high latency of operations is a challenge for attaining good performance, the inability of the file system to calculate content addresses on its own does not present an additional problem.

2.3 Challenges

Hydra presents several challenges to implementing a file system that are not encountered with conventional disk subsystems. Some of the most notable are: *(i)* blocks are immutable, *(ii)* the latency of the block operations is very high, and *(iii)* a *chunking* algorithm must be used to determine the block boundaries that maximize deduplication, and this results in blocks of unpredictable and varied sizes.

2.3.1 Immutable Blocks

When a file system for a conventional disk subsystem needs to update a block, the file system can simply rewrite it, since the block's address is fixed. The new contents become visible without requiring further writes, regardless of how many metadata blocks need to be traversed to reach it.

A CAS system, however, has to store a new block, which may result in the new block's address differing from the old block's address. Because we are no longer interested in the contents of the old block, we will informally call this an "update." (Blocks that are no longer needed are garbage collected by Hydra.) But to reach the new block, we need to update its parent, and so on recursively up to the root of the file system. This leads to two fundamental constraints on data structures stored in a CAS system.

First, because the address of a block is derived from a secure, one-way hash of the block's contents, it is impossible for the file system to store references to blocks not yet written. Since blocks can only contain pointers to blocks written in the past, and more than one block can contain the same block address, the blocks form directed acyclic graphs (DAGs).

Second, the height of the DAG should be minimized to reduce the overhead of modifying blocks. The cost to modify a block in a file system based on a conventional disk subsystem is limited to the cost to read and write the block. In a CAS-based file system, however, the cost to modify a block also includes the cost to modify the chain of blocks that point to the original block. While this problem also occurs in no-overwrite file systems, such as WAFL [11], it is exacerbated by higher Hydra latencies, as discussed in the next section.

2.3.2 High Latency

Another major challenge that Hydra poses is higher latencies than conventional disk subsystems. In a conventional disk subsystem, the primary task in reading or writing a disk block is transferring the data. In Hydra however, much more work must be done before an I/O operation can be completed. This includes scanning the entire contents of the block to compute its content address, compressing or uncompressing the block, determining the location where the block is (or will be) stored, fragmenting or reassembling the blocks that are made up of smaller fragments using error-correcting codes, and routing these fragments to or from the nodes where they reside. While conventional disk subsystems have latencies on the order of milliseconds to tens of milliseconds, Hydra has latencies on the order of hundreds of milliseconds to seconds.

An even higher contributor to the increased latency comes from the requirement to support high-throughput reads. With conventional disk subsystems, placing data in adjacent blocks typically ensures high-throughput reads. The file system can do that because there is a clear indication of adjacency: the block number. However, a CAS system places data based on the block content's hash, which is unpredictable. If Hydra simply places data contiguously based on temporal affinity, as the number of streams written concurrently increases, the blocks of any one stream are further and further apart, reducing the locality and thus causing low read performance.

To mitigate this problem, the block store API allows the caller to specify a *stream hint* for every block write. The block store will attempt to co-locate blocks with the same stream hint by delaying the writes until a sufficiently large number of blocks arrive with the same hint. The decision of what blocks should be co-located is up to the file system; in HydraFS all blocks belonging to the same file are written with the same hint.

The write delay necessary to achieve good read performance depends by the number of concurrent write streams. The default value of the delay is about one second, which is sufficient for supporting up to a hundred concurrent streams. Thus, the write latency is sacrificed for the sake of increased read performance. To cope with the large latencies but still deliver high throughput, the file system must be able to issue a large number of requests concurrently.

2.3.3 Variable Block Sizes

The file system affects the degree of deduplication by how it divides files into blocks, a process we call *chunking*. Studies have shown that variable-size chunking provides better deduplication than fixed-size chunking ([15], [20]). Although fixed-size chunking can be sufficient for some applications, backup streams often contain duplicate data, possibly shifted in the stream by additions, removals, or modifications of files.

Consider the case of inserting a few bytes into a file containing duplicate contents, thereby shifting the contents of the rest of the file. If fixed-size chunking is used, and the number of bytes is not equal to the chunk size, duplicate elimination would be defeated for the range of file contents from the point of insertion through the end of the file. Instead, we use a content-defined chunking algorithm, similar to the one in [18], that produces chunks of variable size between a given minimum and maximum.

This design choice affects the representation of files. With a variable block size, an offset into a file cannot be mapped to the corresponding block by a simple mathematical calculation. This, along with the desire to have DAGs of small height, led us to use balanced tree structures.

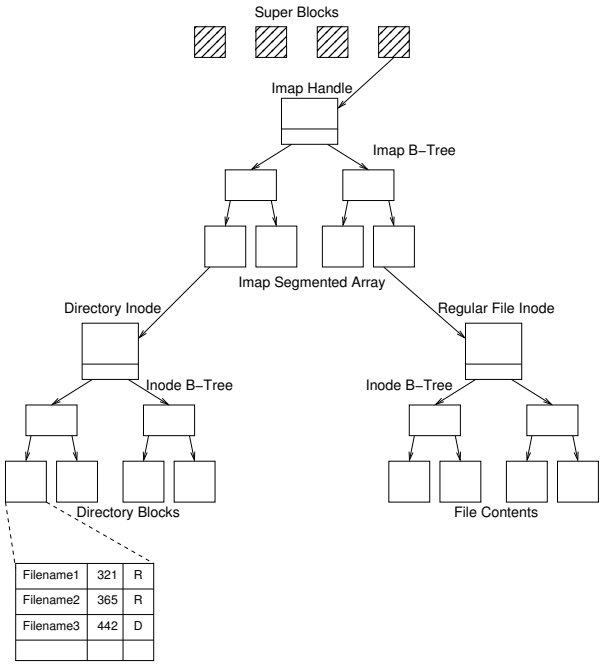

Figure 2: HydraFS persistent layout.

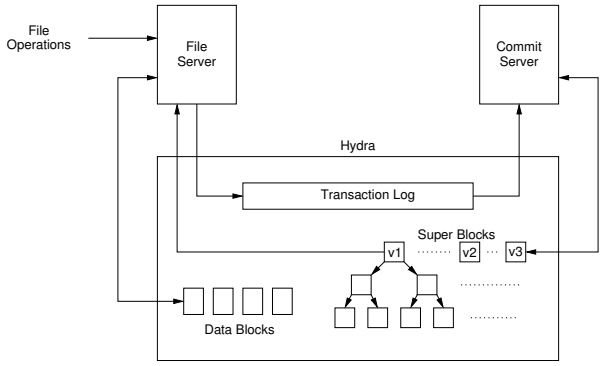

Figure 3: HydraFS Software Architecture.

3 File System Design

HydraFS design is governed by four key principles. First, the primary concern is the high throughput of sequential reads and writes. Other operations, such as metadata operations, file overwrites, and simultaneous reads and writes to the same file, are supported, but are not the target for optimization. Second, because of the high latencies of the block store, the number of dependent I/O operations must be minimized. At the same time, the system must be highly concurrent to obtain high throughput. Third, the data availability guarantees of HydraFS must be no worse than those of the standard Unix file systems. That is, while data acknowledged before an `fsync` may be lost in case of system crash, once an `fsync` is acknowledged to the application, the data must be persistent. Fourth, the file system must efficiently support both local file system access and remote access over NFS and CIFS.

3.1 File System Layout

Figure 2 shows a simplified view of the HydraFS file system block tree. The file system layout is structured as a DAG, with the root of the structure stored in a searchable block. The searchable block contains the file system super block, which holds the address of the *inode map* (called the "imap") together with the current file system version number and some statistics. The imap is conceptually similar to the inode map used in the Log-Structured File System [23]. In HydraFS, the imap is a variable-length array of content addresses and allocation status, stored as a B-tree. It is used to translate inode numbers into inodes, as well as to allocate and free inode numbers.

A regular file inode indexes data blocks with a B-tree so as to accommodate very large files [27] with variable-size blocks. Regular file data is split up into variable-size blocks using a chunking algorithm that is designed to increase the likelihood that the same data written to the block store will generate a match. Thus, if a file is written to the block store on one file system, and then written to another file system using the same block store, the only additional blocks that will be stored by the block store will be the metadata needed to represent the new inode, and its DAG ancestors: the imap and the superblock. The modifications of the last two are potentially amortized over many inode modifications.

Although the immutable nature of Hydra's blocks naturally allows for filesystem snapshots, this feature is not yet exposed to the applications that use HydraFS.

3.2 HydraFS Software Architecture

HydraFS is implemented as a pair of user-level processes that cooperate to provide file system functionality (see Figure 3). The FUSE file system module [8] provides the necessary glue to connect the servers to the Linux kernel file system framework. The *file server* is responsible for managing the file system interface for clients; it handles client requests, records file modifications in a persistent transaction log stored in the block store, and maintains an in-memory cache of recent file modifications. The *commit server* reads the transaction log from the block store, updates the file system metadata, and periodically generates a new file system version.

This separation of functionality has several advantages. First, it simplifies the locking of file system metadata (discussed further in Section 3.3). Second, it allows the

commit server to amortize the cost of updating the file system's metadata by batching updates to the DAG. Third, the split allows the servers to employ different caching strategies without conflicting with each other.

3.3 Write Processing

When an application writes data to a file, the file server accumulates the data in a buffer associated with the file's inode and applies a content-defined chunking algorithm to it. When chunking finds a block boundary, the data in the buffer up to that point is used to generate a new block. The remaining data is left in the buffer to form part of the next block. There is a global limit on the amount of data that is buffered on behalf of inodes, but not yet turned into blocks, to prevent the memory consumption of the inode buffers from growing without bound. When the limit is reached, some buffers are flushed, their content written to Hydra even though the chunk boundaries are no longer content-defined.

Each new block generated by chunking is marked dirty and immediately written to Hydra. It must be retained in memory until Hydra confirms the write. The file server must have it available in case the write is followed by a read of that data, or in case Hydra rejects the block write due to an overloaded condition (the operation is resubmitted after a short delay). When Hydra confirms the write, the block is freed, but its content address is added to the *uncommitted block table* with a timestamp and the byte range that corresponds to the block.

The uncommitted block table is a data structure used for keeping modified file system metadata in memory. Since there is no persistent metadata block pointing to the newly-written data block, this block is not yet reachable in a persistent copy of the file system.

An alternative is to update the persistent metadata immediately, but this has two big problems. The first is that each data block requires the modification of all metadata blocks up to the root. This includes inode index blocks, inode attribute block, and imap blocks. Updating all of them for every data block modification creates substantial I/O overhead. The second is that the modification to these data structures would have to be synchronized with other concurrent operations performed by the file server. Since the metadata tree can only be updated one level at a time (a parent can be written only after the writes of all children complete), propagation up to the root has a very high latency. Locking the imap for the duration of these writes would reduce concurrency considerably, resulting in extremely poor performance. Thus, we chose to keep dirty metadata structures in memory and delegate the writing of metadata to the commit server.

When the commit server finally creates a new file system super block, the file server can clean its dirty metadata structures (see Section 3.4). To provide persistence guarantees, the metadata operations are written to a log which is kept persistently in Hydra until they are executed by the commit server.

Sequentially appending data to files exhibits the best performance in HydraFS. Random writes in HydraFS incur more overhead than appends because of the chunking process that decides the boundaries of the blocks written to Hydra. The boundaries depend on the content of the current write operation, but also on the file data adjacent to the current write range (if any). Thus, a random write to the file system might generate block writes to the block store that include parts of blocks already written, as well as any data that was buffered but not yet written since it was not a complete chunk.

3.4 Metadata Cleaning

The file server must retain dirty metadata as a consequence of delegating metadata processing to the commit server to avoid locking. This data can only be discarded once it can be retrieved from the block store. For this to happen, the commit server must sequentially apply the operations it retrieves from the log written by the file server, create a new file system DAG, and commit it to Hydra.

To avoid unpredictable delays, the commit server generates a new file system version periodically, allowing the file server to clean its dirty metadata proactively. Instead, if the file server waits until its cache fills up before asking the commit server to generate a new root, then the file server would stall until the commit server completes writing all the modified metadata blocks. As mentioned before, this can take a long time, because of the sequential nature of writing content-addressable blocks along a DAG path.

Metadata objects form a tree that is structurally similar to the block tree introduced in Section 3.1. To simplify metadata cleaning, the file server does not directly modify the metadata objects as they are represented on Hydra. Instead, all metadata modifications are maintained in separate lookup structures, with each modification tagged with its creation time. With this approach, the metadata that was read from Hydra is always clean and can be dropped from the cache at any time, if required.

When the file server sees that a new super block has been created, it can clean the metadata objects in a top-down manner. Cleaning a metadata object involves replacing its cached clean state (on-Hydra state) with a new version, and dropping all metadata modification records that have been incorporated into the new version.

The top-down restriction is needed to ensure that a discarded object will not be re-fetched from Hydra using an out-of-date content address. For example, if the file server were to drop a modified inode before updating the imap

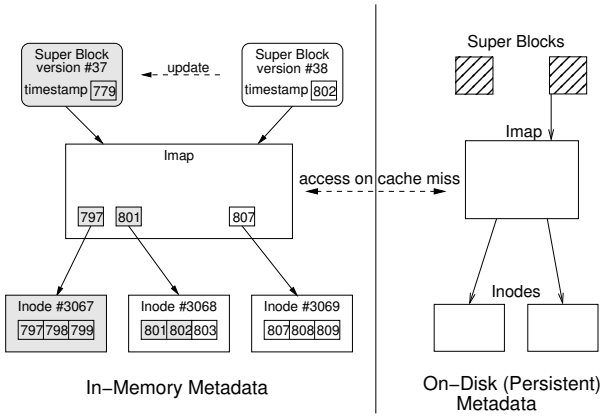

Figure 4: Cleaning of In-Memory Metadata.

first, the imap would still refer to the old content address and a stale inode would be fetched if the inode were accessed again.

Figure 4 shows an example of metadata cleaning. The file server keeps an in-memory list of inode creation (or deletion) records that modify the imap, as well as uncommitted block table records for the inodes, consisting of content addresses with creation timestamps (and offset range, not shown). The file server might also have cached blocks belonging to the old file system version (not shown). Inode 3067 can be discarded, because all of its modifications are included in the latest version of the super block. Inode 3068 cannot be removed, but it can be partially cleaned by dropping content addresses with timestamps 801 and 802. Similarly, creation records up to timestamp 802 can be dropped from the imap. Note that in-memory inodes take precedence over imap entries; the stale imap information for inode 3068 will not be used as long as the inode stays in memory.

3.5 Admission Control

Both servers accept new events for processing after first passing the events through admission control, a mechanism designed to limit the amount of memory consumed. Limits are determined by the amount of memory configured for particular objects, such as disk blocks and inodes. When an event arrives, the worst-case needed allocation size is reserved for each object that might be needed to process the event. If memory is available, the event is allowed into the server's set of active events for processing. Otherwise, the event blocks.

During its lifetime, an event can allocate and free memory as necessary, but the total allocation cannot exceed the reservation. When the event completes, it relinquishes the reservation, but it might not have freed all the memory it allocated. For example, a read event leaves blocks in the cache. Exhaustion of the memory pool triggers a reclamation function that frees cached objects that are clean.

Admission control solves two problems. First, it limits the amount of active events, which in turn limits the amount of heap memory used. This relieves us from having to deal with memory allocation failures, which can be difficult to handle, especially in an asynchronous system where events are in various stages of completion. Second, when the memory allocated for file system objects is tuned with the amount of physical memory in mind, it can prevent paging to the swap device, which would reduce performance.

3.6 Read Processing

The file system cannot respond to a read request until the data is available, making it harder to hide high CAS latencies. To avoid I/O in the critical path of a read request, HydraFS uses aggressive read-ahead for sequential reads into an in-memory, LRU cache, indexed by content address. The amount of additional data to be read is configurable with a default of 20MB.

To obtain the content addresses of the data blocks that cover the read-ahead range, the metadata blocks that store these addresses must also be read from the inode's B-tree. This may require multiple reads to fetch all blocks along the paths from the root of the tree to the leaf nodes of interest. To amortize the I/O cost, HydraFS caches both metadata blocks and the data blocks, uses large leaf nodes, and high fan-out for internal nodes.

Unfortunately, the access patterns for data and metadata blocks differ significantly. For sequential reads, accesses to data blocks are close together, making LRU efficient. In contrast, because of the large fan-out, consecutive metadata block accesses might be separated by many accesses to data blocks, making metadata eviction more likely. An alternative is to use a separate cache for data and metadata blocks, but this does not work well in cases when the ratio of data to metadata blocks differs from the ratio of the two caches. Instead, we use a single *weighted LRU* cache, where metadata blocks have a higher weight, making them harder to evict.

To further reduce the overhead of translating offset-length ranges to content addresses, we use a per-inode look-aside buffer, called the *fast range map* (FRM), that maintains a mapping from an offset range to the content address of the block covering it. The FRM has a fixed size, is populated when a range is first translated, and is cleared when the corresponding inode is updated.

Finally, we also introduce a read-ahead mechanism for metadata blocks to eliminate reads in the critical path of the first access to these blocks. The B-tree read-ahead augments the priming of the FRM for entries that are likely to be needed soon.

3.7 Deletion

When a file is deleted in HydraFS, that file is removed from the current version of the file system namespace. Its storage space, however, remains allocated in the block store until no more references to its blocks exist and the back end runs its garbage collection cycle to reclaim unused blocks. The garbage collection is run as an administrative procedure that requires all modified cached data to be flushed by HydraFS to Hydra to make sure that there are no pointers to blocks that might be reclaimed.

Additional references to a file's blocks can come from two sources: other files that contain the same chunk of data, and older versions of the file system that contain references to the same file. References need not originate from the same file system, however. Since all file systems share the same block store, blocks can match duplicates from other file systems.

When a new version of a file system is created, the oldest version is marked for deletion by writing a deletion root corresponding to its retention root. The file system only specifies which super blocks are to be retained and which are to be deleted, and Hydra manages the reference counts to decide which blocks are to be retained and which are to be freed.

The number of file system versions retained is configurable. These versions are not currently exposed to users; they are retained only to provide insurance should a file system need to be recovered.

Active log blocks are written as shallow trees headed by searchable blocks. Log blocks are marked for deletion as soon as their changes are incorporated into an active version of the file system.

4 Evaluation

HydraFS has been designed to handle sequential workloads operating under unique constraints imposed by the distributed, content-addressable block store. In this section, we present evidence that HydraFS supports high throughput for these workloads while retaining the benefits of block-level duplicate elimination. We first characterize our block storage system, focusing on issues that make it difficult to design a file system on top of it. We then study HydraFS behavior under different workloads.

4.1 Experimental Setup

All experiments were run on a setup of five computers similar to Figure 1. We used a 4-server configuration of storage nodes, in which each server had two dual-core, 64-bit, 3.0 GHz Intel Xeon processors, 6GB of memory, and six 7200 RPM MAXTOR 7H500F0 SATA disks, of which five were used to store blocks, and one was used for logging by Hydra. Its redundancy is given by an erasure coding scheme [1] using 9 original and 3 redundant fragments. A similar hardware configuration was used for the file server, but with 8GB of memory and an ext3 file system on a logical volume split across two 15K RPM Fujitsu MAX3073RC SAS disks using hardware RAID: this file system was used for logging in HydraFS experiments, and for storing data in ext3 experiments. All servers run a 2.6.9 Linux kernel, because this was the version that was used in the initial product release. (It has since been upgraded to a more recent version; regardless, the only local disk I/O on the access node is for logging, so improvements in the disk I/O subsystem won't affect our performance appreciably.)

4.2 HydraFS Efficiency

The goal in this section is to characterize the efficiency of HydraFS and to demonstrate that it comes close to the performance supported by our block store. Unfortunately, since Hydra exports a non-standard API and HydraFS is designed for this API, it is not possible for us to use a common block store for both HydraFS and a disk-based file system, such as ext3. *It is important to note that we are not interested in the absolute performance of the two file systems, but how much the performance degrades when using a file system compared to a raw block device.*

To compare the efficiencies of HydraFS and ext3, we used identical hardware, configured as follows. We exported an ensemble of 6 disks on each storage node as an iSCSI target using a software RAID5 configuration with one parity disk. We used one access node as the iSCSI initiator and used software RAID0 to construct an ensemble that exposes one device node. We used a block size of 64KB for the block device and placed an ext3 file system on it. The file system was mounted with the `noatime` and `nodiratime` mount options. This configuration allows ext3 access to hardware resources similar to Hydra, although its resilience was lower than that of Hydra, as it does not protect against node failure or more than one disk failure per node.

Sequential Throughput: In this experiment, we use a synthetic benchmark to generate a workload for both the HydraFS and ext3 file systems. This benchmark generates a stream of reads or writes with a configurable I/O size using blocking system calls and issues a new request as soon as the previous request completes. Additionally, this benchmark generates data with a configurable fraction of duplicate data, which allows us to study the behavior of HydraFS with variable data characteristics. The throughput of the block store is measured with an application that uses the CAS API to issue in parallel as many block operations as accepted by Hydra, thus exhibiting the maximum level of concurrency possible.

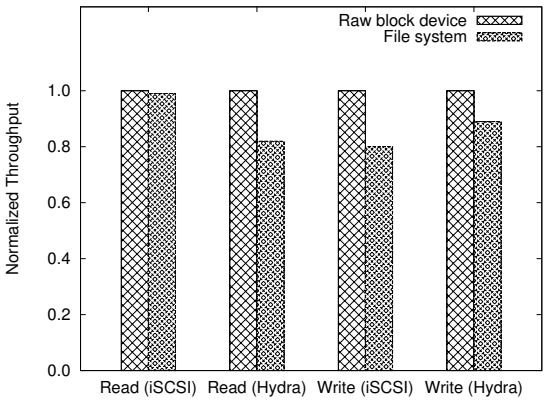

Figure 5: Comparison of raw device and file system throughput for iSCSI and Hydra

Figure 5 shows the read and write throughput achieved by ext3 and HydraFS against the raw block device throughput of the iSCSI ensemble and Hydra respectively. We observe that while the read throughput of ext3 is comparable to that of its raw device, HydraFS read throughput is around 82% of the Hydra throughput. For the write experiment, while ext3 throughput degrades to around 80% of the raw device, HydraFS achieves 88% of Hydra throughput, in spite of the block store's high latency.

Therefore, we conclude that the HydraFS implementation is efficient and the benefits of flexibility and generality of the file system interface do not lead to a significant loss of performance. The performance difference comes mostly from limitations on concurrency imposed by dependencies between blocks, as well as by memory management in HydraFS, which do not exist in raw Hydra access.

Metadata Intensive Workloads: To measure the performance of our system with a metadata-intensive workload, we used Postmark [12] configured with an initial set of 50,000 files in 10 directories, with file sizes between 512B and 16KB. We execute 30,000 transactions for each run of the benchmark. Postmark creates a set of files, followed by a series of transactions involving read or write followed by a create or delete. At the end of the run, the benchmark deletes the entire file set.

Table 1 shows the file creation and deletion rate with and without transactions, including the overall rate of transactions for the experiment. A higher number of transactions indicates better performance for metadata-intensive workloads.

We observe that the performance of HydraFS is much lower than that of ext3. Creating small files presents the worst case for Hydra, as the synchronous metadata operations are amortized over far fewer reads and writes than with large files. Moreover, creation and deletion are limited by the number of inodes HydraFS creates without going through the metadata update in the commit server. We keep this number deliberately low to ensure that the system does not accumulate a large number of uncommitted blocks that increase the turnaround times for the commit server processing, increasing unpredictably the latency of user operations. In contrast, ext3 has no such limitations and all metadata updates are written to the journal.

	Create		Delete		Overall
	Alone	Tx	Alone	Tx	
ext3	1,851	68	1,787	68	136
HydraFS	61	28	676	28	57

Table 1: Postmark comparing HydraFS with ext3 on similar hardware

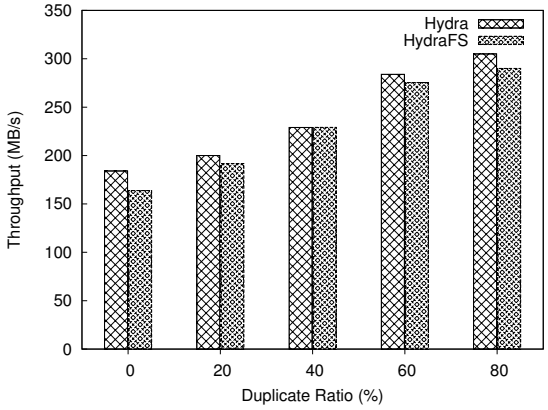

Figure 6: Hydra and HydraFS write throughput with varying duplicate ratio

4.3 Write Performance

In the experiment, we write a 32GB file sequentially to HydraFS using a synthetic benchmark. The benchmark uses the standard file system API for the HydraFS experiment and uses the custom CAS API for the Hydra experiment.

We vary the ratio of duplicate data in the write stream and report the throughput. For repeatability in the presence of variable block sizes and content-defined chunking, our benchmark is designed to generate a configurable average block size, which we set to 64KB in all our experiments.

Figure 6 shows the write throughput when varying the fraction of duplicates in the write stream from no duplicates (0%) to 80% in increments of 20%. We make two observations from our results. First, the throughput increases linearly as the duplicate ratio increases. This is

as expected for duplicate data as the number of I/Os to disk is correspondingly reduced. Second, for all cases, the HydraFS throughput is within 12% of the Hydra throughput. Therefore, we conclude that HydraFS meets the desired goal of maintaining high throughput.

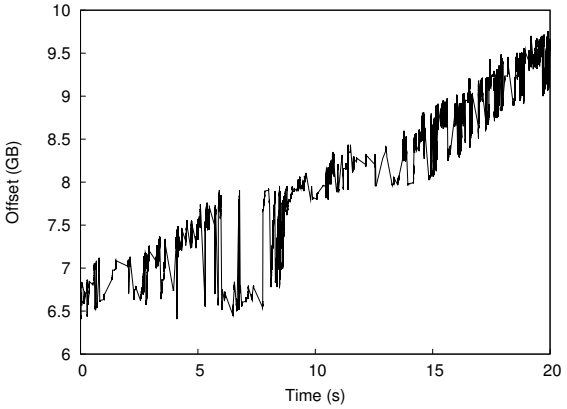

Figure 7: Write completion order

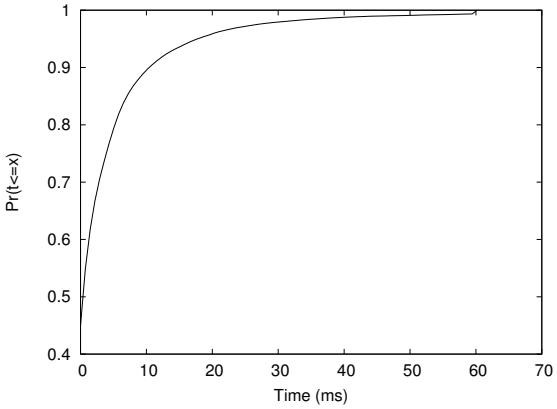

Figure 8: Write event lifetimes

To support high-throughput streaming writes, HydraFS uses a write-behind strategy and does not perform any I/O in the critical path. To manage its memory resources and to prevent thrashing, HydraFS uses a fixed size write buffer and admission control to block write operations before they consume any resources.

Write Behind: Figure 7 shows the order of I/O completions as they arrive from Hydra during a 20-second window of execution of the sequential write benchmark. In an ideal system, the order of completion would be the same as the order of submission and the curve shown in the figure would be a straight line. We observe that in the worst case the gap between two consecutive block completions in this experiment can be as large as 1.5GB, a testament to the high jitter exhibited by Hydra. Consequently, the latency of an internal compound operation requiring many block writes to the back end will experience a latency higher than the average even if all blocks are written in parallel.

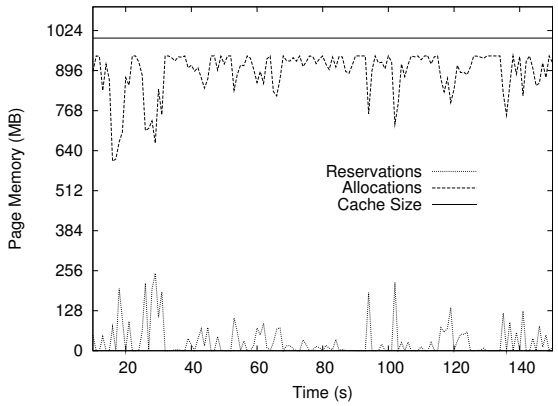

Figure 9: Resource reservation and allocations

To further understand the write behavior, we show the Cumulative Distribution Function (CDF) of the write event lifetimes in the system in Figure 8. The write event is created when the write request arrives at HydraFS and is destroyed when the response is sent to the client. Figure 8 shows that the 90^{th} percentile of write requests take less than 10 ms.

Admission control: In the experiments above, we show that HydraFS is highly concurrent even when the underlying block store is bursty and has high latency. To prevent the system from swapping under these conditions, we use admission control (see Section 3.5). In an ideal system, the allocations must be close to the size of the write buffer and the unused resources must be small to avoid wasting memory. Figure 9 shows the reservations and allocations in the system during a streaming write test. We observe that with admission control, HydraFS is able to maintain high memory utilization and only a fraction of the reserved resources are unused.

Commit Server Processing: Commit server processing overheads are much lower than file server overheads and we observe its CPU utilization to be less than 5% of the file server's utilization for all the experiments above. This allows the commit server to generate new versions well in advance of the file server filling up with dirty metadata, thus avoiding costly file server stalls.

4.4 Read Ahead Caching

In the following experiments, we generate a synthetic workload where a client issues sequential reads for 64KB blocks in a 32GB file. All experiments were performed with a cold cache and the file system was unmounted be-

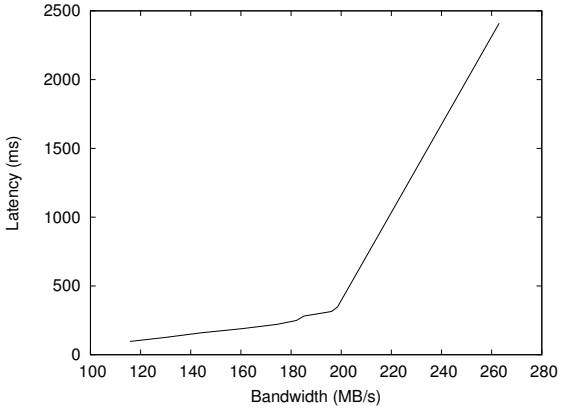

Figure 10: Read throughput vs. average latency

tween runs. Unless otherwise specified, the read-ahead window is fixed at 20MB.

To characterize the read behavior, we study how the read latency varies at different throughput levels. Hydra responds immediately to read requests when data is available. In this experiment we vary the offered load to Hydra by limiting the number of outstanding read requests and measure the time between submitting the request and receiving the response. We limit the number of outstanding read requests by changing the read ahead window from 20MB to 140MB in increments of 15MB.

Figure 10 shows the variation of average latency of read requests when the Hydra throughput is varied. From the figure, we observe that the read latency at low throughput is around 115 ms and increases linearly until the throughput reaches 200MB/s. At higher throughput levels, the latency increases significantly. These results show that the read latencies with Hydra are much higher than other block store latencies. This implies that aggressive read-ahead is essential to maintain high read throughput.

Optimizations: As described in Section 3, to maintain high throughput, we introduced two improvements - Fast Range Map and B-tree Read Ahead (BTreeRA). For a sequential access pattern, once data blocks are read, they are not accessed again. However, the metadata blocks (B-tree blocks) are accessed multiple times, often with a large inter-access gap. Both our optimizations, FRM and BTreeRA, target the misses of metadata blocks.

Table 2 shows the evolution of the read performance with introduction of these mechanisms. The FRM optimization reduces multiple accesses to the metadata blocks leading to a 23% improvement in throughput. BTreeRA reduces cache misses for metadata blocks by issuing read ahead for successive spines of the B-tree concurrently with collecting index data from one spine. Without this prefetch, the nodes populating the spine of the B-tree must be fetched when initiating a read. Moreover, the address

	Thrpt (MB/s)	Accesses	Misses	
			Data	Metadata
Base	134.3	486,966	1,577	1,011
FRM	166.1	210,480	871	1,593
FRM+BTreeRA	183.2	211,632	438	945

Table 2: Effect of read path optimizations

of the block at the next level is available only after the current block is read from Hydra. For large files, with multiple levels in the tree, this introduces a significant latency, which would cause a read stall.

To confirm the hypothesis that the throughput improvements are from reduced metadata accesses and cache misses, Table 2 also shows the number of accesses and the number of misses in the cache for all three cases. We make the following observations: first, our assumption that improving the metadata miss rate has significant impact on read throughput is confirmed. Second, our optimizations add a small memory and CPU overhead but can improve the read throughput by up to 36%.

5 Related Work

Several existing systems use content-addressable storage to support enterprise applications. Venti [21] uses fixed-size blocks and provides archival snapshots of a file system, but since it never deletes blocks, snapshots are made at a low frequency to avoid overloading the storage system with short-lived files. In contrast, HydraFS uses variable-size blocks to improve duplicate elimination and creates file system snapshots more frequently, deleting the oldest version when a new snapshot is created; this is enabled by Hydra providing garbage collection of unreferenced blocks.

Centera [6] uses a cluster of storage nodes to provide expandable, self-managing archival storage for immutable data records. It provides a file system interface to the block store through the Centera Universal Access (CUA), which is similar to the way an access node exports HydraFS file systems in HYDRAstor. The main difference is that the entire HydraFS file system image is managed in-line by storing metadata in the block store as needed; the CUA keeps its metadata locally and makes periodic backups of it to the block store in the background.

Data Domain [4, 31] is an in-line deduplicated storage system for high-throughput backup. Like HydraFS, it uses variable-size chunking. An important difference is that their block store is provided by a single node with RAID-ed storage, whereas Hydra is composed of a set of nodes, and uses erasure coding for configurable resilience at the individual block level.

Deep Store [29] is an architecture for archiving immutable objects that can be indexed by searchable metadata tags. It uses variable-size, content-defined chunks combined with delta compression to improve duplicate elimination. A simple API allows objects to be stored and retrieved, but no attempt is made to make objects accessible through a conventional file system interface.

Many file system designs have addressed providing high performance, fault-tolerant storage for clients on a local area network. The Log-Structured File System (LFS) [23] and Write-Anywhere File Layout (WAFL) [11] make use of specialized file system layouts to allow a file server to buffer large volumes of updates and commit them to disk sequentially. WAFL also supports snapshots that allow previous file system versions to be accessed. LFS uses an imap structure to cope with the fact that block addresses change on every write. WAFL uses an "inode file" containing all the inodes, and updates the relevant block when an inode is modified; HydraFS inodes might contain data and a large number of pointers, so they are stored in separate blocks. Neither LFS nor WAFL support in-line duplicate elimination. Elephant [24] creates new versions of files on every modification and automatically selects "landmark versions," incorporating major changes, for long-term retention. The Low-Bandwidth File System [18] makes use of Rabin fingerprinting [16, 22] to identify common blocks that are stored by a file system client and server, to reduce the amount of data that must be transferred over a low-bandwidth link between the two when the client fetches or updates a file.

The technique of building data structures using hash trees [17] has been used in a number of file systems. SFSRO [7] uses hash trees in building a secure read-only file system. Venti [21] adds duplicate elimination to make a content-addressable block store for archival storage, which can be used to store periodic snapshots of a regular file system. Ivy [19] and OceanStore [14] build on top of wide-area content-addressable storage [26, 30]. While HydraFS is specialized for local-area network performance, Ivy focuses on file system integrity in a multi-user system with untrusted participants, and OceanStore aims to provide robust and secure wide-area file access. Pastiche [3] uses content hashes to build a peer-to-peer backup system that exploits unused disk capacity on desktop computers.

To remove the bottleneck of a single file server, it is possible to use a clustered file system in which several file servers cooperate to supply data to a single client. The Google File System [9] provides high availability and scales to hundreds of clients by providing an API that is tailored for append operations and permits direct communication between a client machine and multiple file servers. Lustre [2] uses a similar architecture in a general-purpose distributed file system. GPFS [25] is a parallel file system that makes use of multiple shared disks and distributed locking algorithms to provide high throughput and strong consistency between clients. In HYDRAstor, multiple access nodes share a common block store, but a file system currently can be modified by only a single access node.

The Frangipani distributed file system [28] has a relationship with its storage subsystem, Petal, that is similar to the relationship between HydraFS and Hydra. In both cases, the file system relies on the block store to be scalable, distributed, and highly-available. However, while HydraFS is written for a content-addressable block store, Frangipani is written for a block store that allows block modifications and does not offer duplicate elimination.

6 Future Work

While the back-end nodes in HYDRAstor operate as a cooperating group of peers, the access nodes act independently to provide file system services. If one access node fails, another access node can recover the file system and start providing access to it, but failover is neither automatic nor transparent. We are currently implementing enhancements to allow multiple access nodes to cooperate in the management of the same file system image, making failover and load-balancing an automatic feature of the front end.

Currently the file system uses a chunking algorithm similar to Rabin fingerprinting [22]. We are working on integrating other algorithms, such as bimodal chunking [13], that generate larger block sizes for comparable duplicate elimination, thereby increasing performance and reducing metadata storage overhead.

HydraFS does not yet expose snapshots to users. Although multiple versions of each file system are maintained, they are not accessible, except as part of a disaster recovery effort by system engineers. We are planning on adding a presentation interface, as well as a mechanism for allowing users to configure snapshot retention.

Although HydraFS is acceptable as a secondary storage platform for a backup appliance, the latency of file system operations makes it less suitable for primary storage. Future work will focus on adapting HydraFS for use as primary storage by using solid state disks to absorb the latency of metadata operations and improve the performance of small file access.

7 Conclusions

We presented HydraFS, a file system for a distributed content-addressable block store. The goals of HydraFS are to provide high throughput read and write access

while achieving high duplicate elimination. We presented the design and implementation of mechanisms that allow HydraFS to achieve these goals and handle the unique CAS characteristics of immutable blocks and high latency.

Through our evaluation, we demonstrated that HydraFS is efficient and supports up to 82% of the block device throughput for reads and up to 100% for writes. We also showed that HydraFS performance is acceptable for use as a backup appliance or a data repository.

A content-addressable storage system, such as HYDRAstor, provides an effective solution for supporting high-performance sequential data access and efficient storage utilization. Support for a standard file system API allows existing applications to take advantage of the efficiency, scalability, and performance of the underlying block store.

Acknowledgments

We would like to thank the anonymous reviewers and our shepherd, Ric Wheeler, for their comments, which helped improve the quality of this paper.

References

[1] J. Blomer, M. Kalfane, R. Karp, M. Karpinski, M. Luby, and D. Zuckerman. An XOR-based erasure-resilient coding scheme. Technical Report TR-95-048, International Computer Science Institute, Aug. 1995.

[2] Cluster File Systems, Inc. Lustre: A Scalable, High-Performance File System, Nov. 2002.

[3] L. P. Cox, C. D. Murray, and B. D. Noble. Pastiche: Making Backup Cheap and Easy. In *Proceedings of the Fifth USENIX Symposium on Operating Systems Design and Implementation (OSDI 2002)*, Boston, Massachusetts, Dec. 2002.

[4] Data Domain, inc. Data Domain DDX Array Series, 2006. http://www.datadomain.com/products/arrays.html.

[5] 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 *Proceedings of the Seventh USENIX Conference on File and Storage Technologies (FAST 2009)*, pages 197–210, San Francisco, California, Feb. 2009.

[6] EMC Corporation. EMC Centera Content Addressed Storage System, 2006. http://www.emc.com/centera.

[7] K. Fu, M. F. Kaashoek, and D. Mazières. Fast and Secure Distributed Read-only File System. In *Proceedings of the Fourth USENIX Symposium on Operating Systems Design and Implementation (OSDI 2000)*, San Diego, California, Oct. 2000.

[8] Filesystem in Userspace. http://fuse.sourceforge.net.

[9] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google File System. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, pages 29–43, Bolton Landing, New York, 2003.

[10] R. Hagmann. Reimplementing the Cedar File System Using Logging and Group Commit. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, pages 155–162, Austin, Texas, Nov. 1987.

[11] D. Hitz, J. Lau, and M. Malcolm. File System Design for an NFS Server Appliance. In *Proceedings of the Winter USENIX Technical Conference 1994*, pages 235–246, San Francisco, California, Jan. 1994.

[12] J. Katcher. Postmark: A New File System Benchmark. Technical Report 3022, Network Appliance, Inc., Oct. 1997.

[13] E. Kruus, C. Ungureanu, and C. Dubnicki. Bimodal Content Defined Chunking for Backup Streams. In *Proceedings of the Eighth USENIX Conference on File and Storage Technologies*, San Jose, California, Feb. 2010.

[14] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. OceanStore: An Architecture for Global-Scale Persistent Storage. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Systems (ASPLOS 2000)*, pages 190–202, Cambridge, Massachusetts, Nov. 2000.

[15] P. Kulkarni, F. Douglis, J. LaVoie, and J. M. Tracey. Redundancy Elimination Within Large Collections of Files. In *Proceedings of the 2004 USENIX Annual Technical Conference*, pages 59–72, Boston, Massachusetts, July 2004.

[16] U. Manber. Finding Similar Files in a Large File System. In *Proceedings of the USENIX Winter 1994 Technical Conference*, pages 1–10, San Fransisco, California, Jan. 1994.

[17] R. C. Merkle. Protocols for Public-Key Cryptosystems. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 122–133, Apr. 1980.

[18] A. Muthitacharoen, B. Chen, and D. Mazières. A Low-Bandwidth Network File System. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, pages 174–187, Lake Louise, Alberta, Oct. 2001.

[19] A. Muthitacharoen, R. T. Morris, T. M. Gil, and B. Chen. Ivy: A Read/Write Peer-to-peer File System. In *Proceedings of the Fifth USENIX Symposium on Operating Systems Design and Implementation (OSDI 2002)*, pages 31–44, Boston, Massachusetts, Dec. 2002.

[20] C. Policroniades and I. Pratt. Alternatives for Detecting Redundancy in Storage Systems Data. In *Proceedings of the 2004 USENIX Annual Technical Conference*, pages 73–86, Boston, Massachusetts, July 2004.

[21] S. Quinlan and S. Dorward. Venti: A New Approach to Archival Storage. In *Proceedings of the First USENIX Conference on File and Storage Technologies (FAST 2002)*, Monterey, California, Jan. 2002.

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

[23] M. Rosenblum and J. K. Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems*, 10(1):26–52, Feb. 1992.

[24] D. S. Santry, M. J. Feeley, N. C. Hutchinson, A. C. Veitch, R. W. Carlton, and J. Ofir. Deciding When to Forget in the Elephant File System. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles*, pages 110–123, Charleston, South Carolina, Dec. 1999.

[25] F. Schmuck and R. Haskin. GPFS: A Shared-Disk File System for Large Computing Clusters. In *Proceedings of the First USENIX Conference on File and Storage Technologies (FAST 2002)*, pages 231–244, Monterey, California, Jan. 2002.

[26] I. Stoica, R. Morris, D. Liben-Nowell, D. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. *IEEE/ACM Transactions on Networking*, 11(1):17–32, Feb. 2003.

[27] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck. Scalability in the XFS File System. In *Proceedings of the USENIX 1996 Technical Conference*, pages 1–14, San Diego, California, Jan. 1996.

[28] C. A. Thekkath, T. Mann, and E. K. Lee. Frangipani: A Scalable Distributed File System. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, pages 224–237, 1997.

[29] L. L. You, K. T. Pollack, and D. D. E. Long. Deep Store: An Archival Storage System Architecture. In *Proceedings of the Twenty-First International Conference on Data Engineering (ICDE 2005)*, Tokyo, Japan, Apr. 2005.

[30] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz. Tapestry: A Resilient Global-scale Overlay for Service Deployment. *IEEE Journal on Selected Areas in Communications*, 22(1):41–53, Jan. 2004.

[31] B. Zhu, K. Li, and H. Patterson. Avoiding the Disk Bottleneck in the Data Domain Deduplication File System. In *Proceedings of the Sixth USENIX Conference on File and Storage Technologies*, pages 269–282, San Jose, California, Feb. 2008.