Avoiding the Disk Bottleneck in the Data Domain Deduplication File System

Benjamin Zhu
Data Domain, Inc.

Kai Li
Data Domain, Inc. and Princeton University

Hugo Patterson
Data Domain, Inc.

Abstract

Disk-based deduplication storage has emerged as the new-generation storage system for enterprise data protection to replace tape libraries. Deduplication removes redundant data segments to compress data into a highly compact form and makes it economical to store backups on disk instead of tape. A crucial requirement for enterprise data protection is high throughput, typically over 100 MB/sec, which enables backups to complete quickly. A significant challenge is to identify and eliminate duplicate data segments at this rate on a low-cost system that cannot afford enough RAM to store an index of the stored segments and may be forced to access an on-disk index for every input segment.

This paper describes three techniques employed in the production Data Domain deduplication file system to relieve the disk bottleneck. These techniques include: (1) the Summary Vector, a compact in-memory data structure for identifying new segments; (2) Stream-Informed Segment Layout, a data layout method to improve on-disk locality for sequentially accessed segments; and (3) Locality Preserved Caching, which maintains the locality of the fingerprints of duplicate segments to achieve high cache hit ratios. Together, they can remove 99% of the disk accesses for deduplication of real world workloads. These techniques enable a modern two-socket dual-core system to run at 90% CPU utilization with only one shelf of 15 disks and achieve 100 MB/sec for single-stream throughput and 210 MB/sec for multi-stream throughput.

1 Introduction

The massive storage requirements for data protection have presented a serious problem for data centers. Typically, data centers perform a weekly full backup of all the data on their primary storage systems to secondary storage devices where they keep these backups for weeks to months. In addition, they may perform daily incremental backups that copy only the data which has changed since the last backup. The frequency, type and retention of backups vary for different kinds of data, but it is common for the secondary storage to hold 10 to 20 times more data than the primary storage. For disaster recovery, additional offsite copies may double the secondary storage capacity needed. If the data is transferred offsite over a wide area network, the network bandwidth requirement can be enormous.

Given the data protection use case, there are two main requirements for a secondary storage system storing backup data. The first is low cost so that storing backups and moving copies offsite does not end up costing significantly more than storing the primary data. The second is high performance so that backups can complete in a timely fashion. In many cases, backups must complete overnight so the load of performing backups does not interfere with normal daytime usage.

The traditional solution has been to use tape libraries as secondary storage devices and to transfer physical tapes for disaster recovery. Tape cartridges cost a small fraction of disk storage systems and they have good sequential transfer rates in the neighborhood of 100 MB/sec. But, managing cartridges is a manual process that is expensive and error prone. It is quite common for restores to fail because a tape cartridge cannot be located or has been damaged during handling. Further, random access performance, needed for data restores, is extremely poor. Disk-based storage systems and network replication would be much preferred if they were affordable.

During the past few years, disk-based, "deduplication" storage systems have been introduced for data protection [QD02, MCM01, KDLT04, Dat05, JDT05]. Such systems compress data by removing duplicate data across files and often across all the data in a storage system. Some implementations achieve a 20:1 compression ratio (total data size divided by physical space used) for 3 months of backup data using the daily-incremental and weekly-full backup policy. By substantially reducing the footprint of versioned data, deduplication can make the costs of storage on disk and tape comparable and make replicating data over a WAN to a remote site for disaster recovery practical.

The specific deduplication approach varies among system vendors. Certainly the different approaches vary in how effectively they reduce data. But, the goal of this paper is not to investigate how to get the greatest data reduction, but rather how to do deduplication at high speed in order to meet the performance requirement for secondary storage used for data protection.

The most widely used deduplication method for secondary storage, which we call Identical Segment Deduplication, breaks a data file or stream into contiguous segments and eliminates duplicate copies of identical segments. Several emerging commercial systems have used this approach.

The focus of this paper is to show how to implement a high-throughput Identical Segment Deduplication storage system at low system cost. The key performance challenge is finding duplicate segments. Given a segment size of 8 KB and a performance target of 100 MB/sec, a deduplication system must process approximately 12,000 segments per second.

An in-memory index of all segment fingerprints could easily achieve this performance, but the size of the index would limit system size and increase system cost. Consider a segment size of 8 KB and a segment fingerprint size of 20 bytes. Supporting 8 TB worth of unique segments, would require 20 GB just to store the fingerprints.

An alternative approach is to maintain an on-disk index of segment fingerprints and use a cache to accelerate segment index accesses. Unfortunately, a traditional cache would not be effective for this workload. Since fingerprint values are random, there is no spatial locality in the segment index accesses. Moreover, because the backup workload streams large data sets through the system, there is very little temporal locality. Most segments are referenced just once every week during the full backup of one particular system. Reference-based caching algorithms such as LRU do not work well for such workloads. The Venti system, for example, implemented such a cache [QD02]. Its combination of index and block caches only improves its write throughput by about 16% (from 5.6MB/sec to 6.5MB/sec) even with 8 parallel disk index lookups. The primary reason is due to its low cache hit ratios.

With low cache hit ratios, most index lookups require disk operations. If each index lookup requires a disk access which may take 10 msec and 8 disks are used for index lookups in parallel, the write throughput will be about 6.4MB/sec, roughly corresponding to Venti's throughput of less than 6.5MB/sec with 8 drives. While Venti's performance may be adequate for the archival usage of a small workgroup, it's a far cry from the throughput goal of deduplicating at 100 MB/sec to compete with high-end tape libraries. Achieving 100 MB/sec, would require 125 disks doing index lookups in parallel! This would increase the system cost of deduplication storage to an unattainable level.

Our key idea is to use a combination of three methods to reduce the need for on-disk index lookups during the deduplication process. We present in detail each of the three techniques used in the production Data Domain deduplication file system. The first is to use a Bloom filter, which we call a *Summary Vector*, as the summary data structure to test if a data segment is new to the system. It avoids wasted lookups for segments that do not exist in the index. The second is to store data segments and their fingerprints in the same order that they occur in a data file or stream. Such *Stream-Informed Segment Layout* (SISL) creates spatial locality for segment and fingerprint accesses. The third, called *Locality Preserved Caching*, takes advantage of the segment layout to fetch and cache groups of segment fingerprints that are likely to be accessed together. A single disk access can result in many cache hits and thus avoid many on-disk index lookups.

Our evaluation shows that these techniques are effective in removing the disk bottleneck in an Identical Segment Deduplication storage system. For a system running on a server with two dual-core CPUs with one shelf of 15 drives, these techniques can eliminate about 99% of index lookups for variable-length segments with an average size of about 8 KB. We show that the system indeed delivers high throughput: achieving over 100 MB/sec for single-stream write and read performance, and over 210 MB/sec for multi-stream write performance. This is an order-of-magnitude throughput improvement over the parallel indexing techniques presented in the Venti system.

The rest of the paper is organized as follows. Section 2 presents challenges and observations in designing a deduplication storage system for data protection. Section 3 describes the software architecture of the production Data Domain deduplication file system. Section 4 presents our methods for avoiding the disk bottleneck. Section 5 shows our experimental results. Section 6 gives an overview of the related work, and Section 7 draws conclusions.

2 Challenges and Observations

2.1 Variable vs. Fixed Length Segments

An Identical Segment Deduplication system could choose to use either fixed length segments or variable length segments created in a content dependent manner. Fixed length segments are the same as the fixed-size blocks of many non-deduplication file systems. For the purposes of this discussion, extents that are multiples of

some underlying fixed size unit such as a disk sector are the same as fixed-size blocks.

Variable-length segments can be any number of bytes in length within some range. They are the result of partitioning a file or data stream in a content dependent manner [Man93, BDH94].

The main advantage of a fixed segment size is simplicity. A conventional file system can create fixed-size blocks in the usual way and a deduplication process can then be applied to deduplicate those fixed-size blocks or segments. The approach is effective at deduplicating whole files that are identical because every block of identical files will of course be identical.

In backup applications, single files are backup images that are made up of large numbers of component files. These files are rarely entirely identical even when they are successive backups of the same file system. A single addition, deletion, or change of any component file can easily shift the remaining image content. Even if no other file has changed, the shift would cause each fixed sized segment to be different than it was last time, containing some bytes from one neighbor and giving up some bytes to its other neighbor. The approach of partitioning the data into variable length segments based on content allows a segment to grow or shrink as needed so the remaining segments can be identical to previously stored segments.

Even for storing individual files, variable length segments have an advantage. Many files are very similar to, but not identical to other versions of the same file. Variable length segments can accommodate these differences and maximize the number of identical segments.

Because variable length segments are essential for deduplication of the shifted content of backup images, we have chosen them over fixed-length segments.

2.2 Segment Size

Whether fixed or variable sized, the choice of average segment size is difficult because of its impact on compression and performance. The smaller the segments, the more duplicate segments there will be. Put another way, if there is a small modification to a file, the smaller the segment, the smaller the new data that must be stored and the more of the file's bytes will be in duplicate segments. Within limits, smaller segments will result in a better compression ratio.

On the other hand, with smaller segments, there are more segments to process which reduces performance. At a minimum, more segments mean more times through the deduplication loop, but it is also likely to mean more on-disk index lookups.

With smaller segments, there are more segments to manage. Since each segment requires the same metadata size, smaller segments will require more storage footprint for their metadata, and the segment fingerprints for fewer total user bytes can be cached in a given amount of memory. The segment index is larger. There are more updates to the index. To the extent that any data structures scale with the number of segments, they will limit the overall capacity of the system. Since commodity servers typically have a hard limit on the amount of physical memory in a system, the decision on the segment size can greatly affect the cost of the system.

A well-designed duplication storage system should have the smallest segment size possible given the throughput and capacity requirements for the product. After several iterative design processes, we have chosen to use 8 KB as the average segment size for the variable sized data segments in our deduplication storage system.

2.3 Performance-Capacity Balance

A secondary storage system used for data protection must support a reasonable balance between capacity and performance. Since backups must complete within a fixed backup window time, a system with a given performance can only backup so much data within the backup window. Further, given a fixed retention period for the data being backed up, the storage system needs only so much capacity to retain the backups that can complete within the backup window. Conversely, given a particular storage capacity, backup policy, and deduplication efficiency, it is possible to compute the throughput that the system must sustain to justify the capacity. This balance between performance and capacity motivates the need to achieve good system performance with only a small number of disk drives.

Assuming a backup policy of weekly fulls and daily incrementals with a retention period of 15 weeks and a system that achieves a 20x compression ratio storing backups for such a policy, as a rough rule of thumb, it requires approximately as much capacity as the primary data to store all the backup images. That is, for 1 TB of primary data, the deduplication secondary storage would consume approximately 1 TB of physical capacity to store the 15 weeks of backups.

Weekly full backups are commonly done over the weekend with a backup window of 16 hours. The balance of the weekend is reserved for restarting failed backups or making additional copies. Using the rule of thumb above, 1 TB of capacity can protect approximately 1 TB of primary data. All of that must be backed up within the 16-hour backup window which implies a throughput of about 18 MB/sec per terabyte of capacity.

Following this logic, a system with a shelf of 15 SATA drives each with a capacity of 500 GB and a total usable capacity after RAID, spares, and other overhead of 6 TB could protect 6 TB of primary storage and must therefore be able to sustain over 100 MB/sec of deduplication throughput.

2.4 Fingerprint vs. Byte Comparisons

An Identical Segment Deduplication storage system needs a method to determine that two segments are identical. This could be done with a byte by byte comparison of the newly written segment with the previously stored segment. However, such a comparison is only possible by first reading the previously stored segment from disk. This would be much more onerous than looking up a segment in an index and would make it extremely difficult if not impossible to maintain the needed throughput.

To avoid this overhead, we rely on comparisons of segment fingerprints to determine the identity of a segment. The fingerprint is a collision-resistant hash value computed over the content of each segment. SHA-1 is such a collision-resistant function [NIST95]. At a 160-bit output value, the probability of fingerprint collision by a pair of different segments is extremely small, many orders of magnitude smaller than hardware error rates [QD02]. When data corruption occurs, it will almost certainly be the result of undetected errors in RAM, IO busses, network transfers, disk storage devices, other hardware components or software errors and not from a collision.

3 Deduplication Storage System Architecture

To provide the context for presenting our methods for avoiding the disk bottleneck, this section describes the architecture of the production Data Domain File System, DDFS, for which Identical Segment Deduplication is an integral feature. Note that the methods presented in the next section are general and can apply to other Identical Segment Deduplication storage systems.

At the highest level, DDFS breaks a file into variable-length segments in a content dependent manner [Man93, BDH94] and computes a fingerprint for each segment. DDFS uses the fingerprints both to identify duplicate segments and as part of a segment descriptor used to reference a segment. It represents files as sequences of segment fingerprints. During writes, DDFS identifies duplicate segments and does its best to store only one copy of any particular segment. Before storing a new segment, DDFS uses a variation of the Ziv-Lempel algorithm to compress the segment [ZL77].

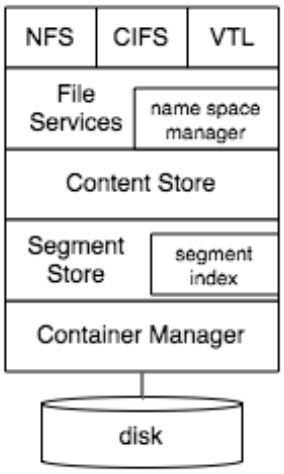

Figure 1: Data Domain File System architecture.

Figure 1 is a block diagram of DDFS, which is made up of a stack of software components. At the top of the stack, DDFS supports multiple access protocols which are layered on a common File Services interface. Supported protocols include NFS, CIFS, and a virtual tape library interface (VTL).

When a data stream enters the system, it goes through one of the standard interfaces to the generic File Services layer, which manages the name space and file metadata. The File Services layer forwards write requests to Content Store which manages the data content within a file. Content Store breaks a data stream into segments, uses Segment Store to perform deduplication, and keeps track of the references for a file. Segment Store does the actual work of deduplication. It packs deduplicated (unique) segments into relatively large units, compresses such units using a variation of Ziv-Lempel algorithm to further compress the data, and then writes the compressed results into containers supported by Container Manager.

To read a data stream from the system, a client drives the read operation through one of the standard interfaces and the File Services Layer. Content Store uses the references to deduplicated segments to deliver the desired data stream to the client. Segment Store prefetches, decompresses, reads and caches data segments from Container Manager.

The following describes the Content Store, Segment Store and the Container Manager in detail and discusses our design decisions.

3.1 Content Store

Content Store implements byte-range writes and reads for deduplicated data objects, where an object is a linear

sequence of client data bytes and has intrinsic and client-settable attributes or metadata. An object may be a conventional file, a backup image of an entire volume or a tape cartridge.

To write a range of bytes into an object, Content Store performs several operations.

- *Anchoring* partitions the byte range into variable-length segments in a content dependent manner [Man93, BDH94].
- *Segment fingerprinting* computes the SHA-1 hash and generates the segment descriptor based on it. Each segment descriptor contains per segment information of at least fingerprint and size
- *Segment mapping* builds the tree of segments that records the mapping between object byte ranges and segment descriptors. The goal is to represent a data object using references to deduplicated segments.

To read a range of bytes in an object, Content Store traverses the tree of segments created by the segment mapping operation above to obtain the segment descriptors for the relevant segments. It fetches the segments from Segment Store and returns the requested byte range to the client.

3.2 Segment Store

Segment Store is essentially a database of segments keyed by their segment descriptors. To support writes, it accepts segments with their segment descriptors and stores them. To support reads, it fetches segments designated by their segment descriptors.

To write a data segment, Segment Store performs several operations.

- *Segment filtering* determines if a segment is a duplicate. This is the key operation to deduplicate segments and may trigger disk I/Os, thus its overhead can significantly impact throughput performance.
- *Container packing* adds segments to be stored to a container which is the unit of storage in the system. The packing operation also compresses segment data using a variation of the Ziv-Lempel algorithm. A container, when fully packed, is appended to the Container Manager.
- *Segment Indexing* updates the segment index that maps segment descriptors to the container holding the segment, after the container has been appended to the Container Manager.

To read a data segment, Segment Store performs the following operations.

Figure 2: Containers are self-describing, immutable, units of storage several megabytes in size. All segments are stored in containers.

- *Segment lookup* finds the container storing the requested segment. This operation may trigger disk I/Os to look in the on-disk index, thus it is throughput sensitive.
- *Container retrieval* reads the relevant portion of the indicated container by invoking the Container Manager.
- *Container unpacking* decompresses the retrieved portion of the container and returns the requested data segment.

3.3 Container Manager

The Container Manager provides a storage container log abstraction, not a block abstraction, to Segment Store. Containers, shown in Figure 2, are self-describing in that a metadata section includes the segment descriptors for the stored segments. They are immutable in that new containers can be appended and old containers deleted, but containers cannot be modified once written. When Segment Store appends a container, the Container Manager returns a container ID which is unique over the life of the system.

The Container Manager is responsible for allocating, deallocating, reading, writing and reliably storing containers. It supports reads of the metadata section or a portion of the data section, but it only supports appends of whole containers. If a container is not full but needs to be written to disk, it is padded out to its full size.

Container Manager is built on top of standard block storage. Advanced techniques such as Software RAID-6, continuous data scrubbing, container verification, and end to end data checks are applied to ensure a high level of data integrity and reliability.

The container abstraction offers several benefits.

- The fixed container size makes container allocation and deallocation easy.
- The large granularity of a container write achieves high disk throughput utilization.
- A properly sized container size allows efficient full-stripe RAID writes, which enables an efficient software RAID implementation at the storage layer.

4 Acceleration Methods

This section presents three methods to accelerate the deduplication process in our deduplication storage system: summary vector, stream-informed data layout, and locality preserved caching. The combination of these methods allows our system to avoid about 99% of the disk I/Os required by a system relying on index lookups alone. The following describes each of the three techniques in detail.

4.1 Summary Vector

The purpose of the Summary Vector is to reduce the number of times that the system goes to disk to look for a duplicate segment only to find that none exists. One can think of the Summary Vector as an in-memory, conservative summary of the segment index. If the Summary Vector indicates a segment is not in the index, then there is no point in looking further for the segment; the segment is new and should be stored. On the other hand, being only an approximation of the index, if the Summary Vector indicates the segment is in the index, there is a high probability that the segment is actually in the segment index, but there is no guarantee.

The Summary Vector implements the following operations:

- `Init`()
- `Insert`(fingerprint)
- `Lookup`(fingerprint)

We use a Bloom filter to implement the Summary Vector in our current design [Blo70]. A Bloom filter uses a vector of m bits to summarize the existence information about n fingerprints in the segment index. In `Init()`, all bits are set to 0. `Insert(a)` uses k independent hashing functions, h_1, …, h_k, each mapping a fingerprint a to [0, m -1] and sets the bits at position $h_1(a)$, …, $h_k(a)$ to 1. For any fingerprint x, `Lookup(x)` will check all bits at position $h_1(x)$, …, $h_k(x)$ to see if they are all set to 1. If any of the bits is 0, then we know x is definitely not in the segment index. Otherwise, with high probability, x will be in the segment index, assuming reasonable choices of m, n, and k. Figure 3 illustrates the operations of Summary Vector.

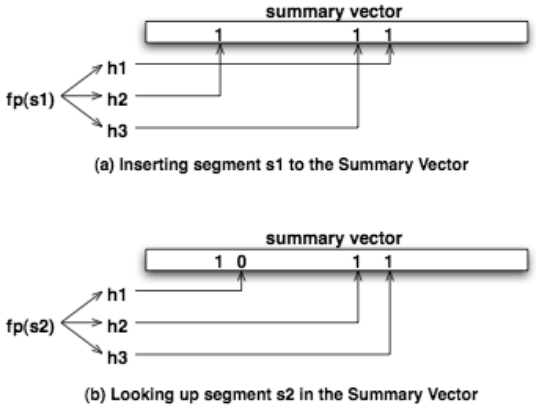

Figure 3: Summary Vector operations. The Summary Vector can identify most new segments without looking up the segment index. Initially all bits in the array are 0. On insertion, shown in (a), bits specified by several hashes, h1, h2, and h3 of the fingerprint of the segment are set to 1. On lookup, shown in (b), the bits specified by the same hashes are checked. If any are 0, as shown in this case, the segment cannot be in the system.

As indicated in [FCAB98], the probability of false positive for an element not in the set, or the *false positive rate*, can be calculated in a straightforward fashion, given our assumption that hash functions are perfectly random. After all n elements hashed and inserted into the Bloom filter, the probability that a specific bit is still 0 is

$$\left(1-\frac{1}{m}\right)^{kn} = e^{-kn/m}.$$

The probability of false positive is then:

$$\left(1-\left(1-\frac{1}{m}\right)^{kn}\right)^{k} \approx \left(1-e^{-kn/m}\right)^{k}.$$

Using this formula, one can derive a particular parameter to achieve a given false positive rate. For example, to achieve 2% false positive rate, the smallest size of the Summary Vector is 8 × n bits (m/n = 8) and the number of hash functions can be 4 (k = 4).

To have a fairly small probability of false positive such as a fraction of a percent, we choose m such that m/n is about 8 for a target goal of n and k around 4 or 5. For example, supporting one billion base segments requires about 1 GB of memory for the Summary Vector.

At system shutdown the system writes the Summary Vector to disk. At startup, it reads in the saved copy. To handle power failures and other kinds of unclean shutdowns, the system periodically checkpoints the

Summary Vector to disk. To recover, the system loads the most recent checkpoint of the Summary Vector and then processes the containers appended to the container log since the checkpoint, adding the contained segments to the Summary Vector.

Although several variations of Bloom filters have been proposed during the past few years [BM05], we have chosen the basic Bloom Filter for simplicity and efficient implementation.

4.2 Stream-Informed Segment Layout

We use Stream-Informed Segment Layout (SISL) to create spatial locality for both segment data and segment descriptors and to enable Locality Preserved Caching as described in the next section. A stream here is just the sequence of bytes that make up a backup image stored in a Content Store object.

Our main observation is that in backup applications, segments tend to reappear in the same of very similar sequences with other segments. Consider a 1 MB file with a hundred or more segments. Every time that file is backed up, the same sequence of a hundred segments will appear. If the file is modified slightly, there will be some new segments, but the rest will appear in the same order. When new data contains a duplicate segment x, there is a high probability that other segments in its locale are duplicates of the neighbors of x. We call this property segment duplicate locality. SISL is designed to preserve this locality.

Content Store and Segment Store support a *stream* abstraction that segregates the segments created for different objects, preserves the logical ordering of segments within the Content Store object, and dedicates containers to hold segments for a single stream in their logical order. The metadata sections of these containers store segment descriptors in their logical order. Multiple streams can be written to Segment Store in parallel, but the stream abstraction prevents the segments for the different streams from being jumbled together in a container.

The design decision to make the deduplication storage system stream aware is a significant distinction from other systems such as Venti.

When an object is opened for writing, Content Store opens a corresponding stream with Segment Store which in turn assigns a container to the stream. Content Store writes ordered batches of segments for the object to the stream. Segment Store packs the new segments into the data section of the dedicated container, performs a variation of Ziv-Lempel compression on the data section, and writes segment descriptors into the metadata section of the container. When the container fills up, it appends it with Container Manager and starts a new container for the stream. Because multiple streams can write to Segment Store in parallel, there may be multiple open containers, one for each active stream.

The end result is Stream-Informed Segment Layout or SISL, because for a data stream, new data segments are stored together in the data sections, and their segment descriptors are stored together in the metadata section.

SISL offers many benefits.

- When multiple segments of the same data stream are written to a container together, many fewer disk I/Os are needed to reconstruct the stream which helps the system achieve high read throughput.
- Descriptors and compressed data of adjacent new segments in the same stream are packed linearly in the metadata and data sections respectively in the same container. This packing captures duplicate locality for future streams resembling this stream, and enables Locality Preserved Caching to work effectively.
- The metadata section is stored separately from the data section, and is generally much smaller than the data section. For example, a container size of 4 MB, an average segment size of 8 KB, and a Ziv-Lempel compression ratio of 2, yield about 1K segments in a container, and require a metadata section size of just about 64 KB, at a segment descriptor size of 64 bytes. The small granularity on container metadata section reads allows Locality Preserved Caching in a highly efficient manner: 1K segments can be cached using a single small disk I/O. This contrasts to the old way of one on-disk index lookup per segment.

These advantages make SISL an effective mechanism for deduplicating multiple-stream fine-grained data segments. Packing containers in a stream aware fashion distinguishes our system from Venti and many other systems.

4.3 Locality Preserved Caching

We use *Locality Preserved Caching* (LPC) to accelerate the process of identifying duplicate segments.

A traditional cache does not work well for caching fingerprints, hashes, or descriptors for duplicate detection because fingerprints are essentially random. Since it is difficult to predict the index location for next segment without going through the actual index access again, the miss ratio of a traditional cache will be extremely high.

We apply LPC to take advantage of segment duplicate locality so that if a segment is a duplicate, the base segment is highly likely cached already. LPC is achieved

by combining the container abstraction with a segment cache as discussed next.

For segments that cannot be resolved by the Summary Vector and LPC, we resort to looking up the segment in the segment index. We have two goals on this retrieval:

- Making this retrieval a relatively rare occurrence.
- Whenever the retrieval is made, it benefits segment filtering of future segments in the locale.

LPC implements a segment cache to cache likely base segment descriptors for future duplicate segments. The segment cache maps a segment fingerprint to its corresponding container ID. Our main idea is to maintain the segment cache by groups of fingerprints. On a miss, LPC will fetch the entire metadata section in a container, insert all fingerprints in the metadata section into the cache, and remove all fingerprints of an old metadata section from the cache together. This method will preserve the locality of fingerprints of a container in the cache.

The operations for the segment cache are:

- `Init()`: Initialize the segment cache.
- `Insert(container)`: Iterate through all segment descriptors in container metadata section, and insert each descriptor and container ID into the segment cache.
- `Remove(container)`: Iterate through all segment descriptors in container metadata section, and remove each descriptor and container ID from the segment cache.
- `Lookup(fingerprint)`: Find the corresponding container ID for the fingerprint specified.

Descriptors of all segments in a container are added or removed from the segment cache at once. Segment caching is typically triggered by a duplicate segment that misses in the segment cache, and requires a lookup in the segment index. As a side effect of finding the corresponding container ID in the segment index, we prefetch all segment descriptors in this container to the segment cache. We call this Locality Preserved Caching. The intuition is that base segments in this container are likely to be checked against for future duplicate segments, based on segment duplicate locality. Our results on real world data have validated this intuition overwhelmingly.

We have implemented the segment cache using a hash table. When the segment cache is full, containers that are ineffective in accelerating segment filtering are leading candidates for replacement from the segment cache. A reasonable cache replacement policy is Least-Recently-Used (LRU) on cached containers.

4.4 Accelerated Segment Filtering

We have combined all three techniques above in the segment filtering phase of our implementation.

For an incoming segment for write, the algorithm does the following:

- Checks to see if it is in the segment cache. If it is in the cache, the incoming segment is a duplicate.
- If it is not in the segment cache, check the Summary Vector. If it is not in the Summary Vector, the segment is new. Write the new segment into the current container.
- If it is in the Summary Vector, lookup the segment index for its container Id. If it is in the index, the incoming segment is a duplicate; insert the metadata section of the container into the segment cache. If the segment cache is full, remove the metadata section of the least recently used container first.
- If it is not in the segment index, the segment is new. Write the new segment into the current container.

We aim to keep the segment index lookup to a minimum in segment filtering.

5 Experimental Results

We would like to answer the following questions:

- How well does the deduplication storage system work with real world datasets?
- How effective are the three techniques in terms of reducing disk I/O operations?
- What throughput can a deduplication storage system using these techniques achieve?

For the first question, we will report our results with real world data from two customer data centers. For the next two questions, we conducted experiments with several internal datasets. Our experiments use a Data Domain DD580 deduplication storage system as an NFS v3 server [PJSS*94]. This deduplication system features two-socket duel-core CPU's running at 3 Ghz, a total of 8 GB system memory, 2 gigabit NIC cards, and a 15-drive disk subsystem running software RAID6 with one spare drive. We use 1 and 4 backup client computers running NFS v3 client for sending data.

5.1 Results with Real World Data

The system described in this paper has been used at over 1,000 data centers. The following paragraphs report the deduplication results from two data centers, generated from the auto-support mechanism of the system.

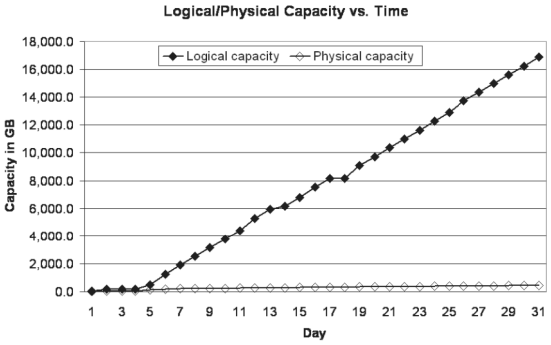

Figure 4: Logical/Physical Capacities at Data Center A

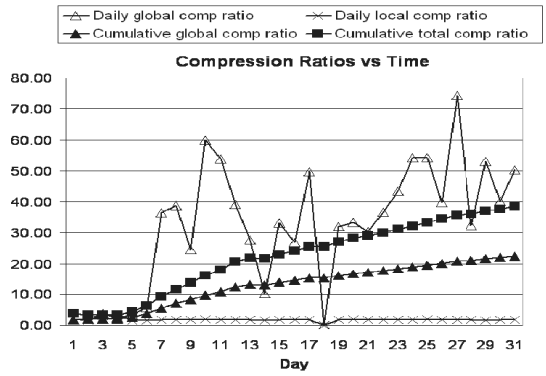

Figure 5: Compression Ratios at Data Center A

	Min	Max	Average	Standard deviation
Daily global compression	10.05	74.31	40.63	13.73
Daily local compression	1.58	1.97	1.78	0.09

Table 1: Statistics on Daily Global and Daily Local Compression Ratios at Data Center A

Data center A backs up structured database data over the course of 31 days during the initial deployment of a deduplication system. The backup policy is to do daily full backups, where each full backup produces over 600 GB at steady state. There are two exceptions:

- During the initial seeding phase (until 6th day in this example), different data or different types of data are rolled into the backup set, as backup administrators figure out how they want to use the deduplication system. A low rate of duplicate segment identification and elimination is typically associated with the seeding phase.
- There are certain days (18th day in this example) when no backup is generated.

Figure 4 shows the logical capacity (the amount of data from user or backup application perspective) and the physical capacity (the amount of data stored in disk media) of the system over time at data center A.

At the end of 31st day, the data center has backed up about 16.9 TB, and the corresponding physical capacity is less than 440 GB, reaching a total compression ratio of 38.54 to 1.

Figure 5 shows daily global compression ratio (the daily rate of data reduction due to duplicate segment elimination), daily local compression ratio (the daily rate of data reduction due to Ziv-Lempel style compression on new segments), cumulative global compression ratio (the cumulative ratio of data reduction due to duplicate segment elimination), and cumulative total compression ratio (the cumulative ratio of data reduction due to duplicate segment elimination and Ziv-Lempel style compression on new segments) over time.

At the end of 31st day, cumulative global compression ratio reaches 22.53 to 1, and cumulative total compression ratio reaches 38.54 to 1.

The daily global compression ratios change quite a bit over time, whereas the daily local compression ratios are quite stable. Table 1 summarizes the minimum, maximum, average, and standard deviation of both daily global and daily local compression ratios, excluding seeding (the first 6) days and no backup (18th) day.

Data center B backs up a mixture of structured database and unstructured file system data over the course of 48 days during the initial deployment of a deduplication system using both full and incremental backups. Similar to that in data center A, seeding lasts until the 6th day, and there are a few days without backups (8th, 12-14th, 35th days). Outside these days, the maximum daily logical backup size is about 2.1 TB, and the smallest size is about 50 GB.

Figure 6 shows the logical capacity and the physical capacity of the system over time at data center B.

At the end of 48th day, the logical capacity reaches about 41.4 TB, and the corresponding physical capacity is about 3.0 TB. The total compression ratio is 13.71 to 1.

Figure 7 shows daily global compression ratio, daily local compression ratio, cumulative global compression ratio, and cumulative total compression ratio over time.

At the end of 48th day, cumulative global compression reaches 6.85, while cumulative total compression reaches 13.71.

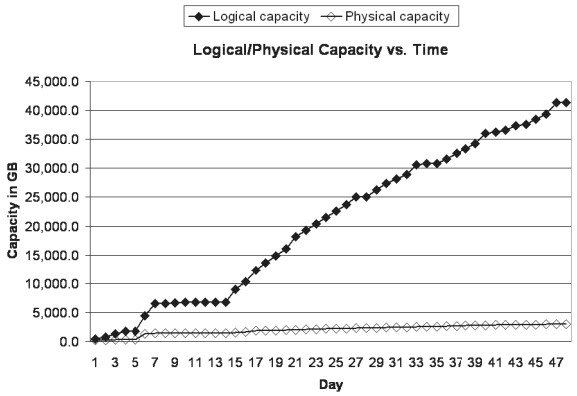

Figure 6: Logical/Physical Capacities at Data Center B.

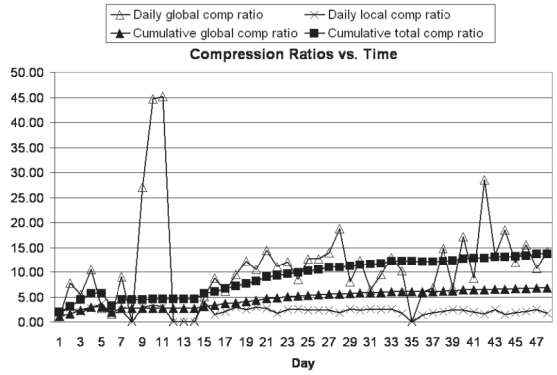

Figure 7: Compression Ratios at Data Center B.

	Min	Max	Average	Standard deviation
Daily global compression	5.09	45.16	13.92	9.08
Daily local compression	1.40	4.13	2.33	0.57

Table 2: Statistics on Daily Global and Daily Local Compression Ratios at Data Center B

	Exchange data	Engineering data
Logical capacity (TB)	2.76	2.54
Physical capacity after deduplicating segments (TB)	0.49	0.50
Global compression	5.69	5.04
Physical capacity after local compression (TB)	0.22	0.261
Local compression	2.17	1.93
Total compression	12.36	9.75

Table 3: Capacities and Compression Ratios on Exchange and Engineering Datasets

Table 2 summarizes the minimum, maximum, average, and standard deviation of both daily global and daily local compression ratios, excluding seeding and days without backup.

The two sets of results show that the deduplication storage system works well with the real world datasets. As expected, both cumulative global and cumulative total compression ratios increase as the system holds more backup data.

During seeding, duplicate segment elimination tends to be ineffective, because most segments are new. After seeding, despite the large variation in the actual number, duplicate segment elimination becomes extremely effective. Independent of seeding, Ziv-Lempel style compression is relatively stable, giving a reduction of about 2 over time. The real world observations on the applicability of duplicate segment elimination during seeding and after seeding are particularly relevant in evaluating our techniques to reduce disk accesses below.

5.2 I/O Savings with Summary Vector and Locality Preserved Caching

To determine the effectiveness of the Summary Vector and Locality Preserved Caching, we examine the savings for disk reads to find duplicate segments using a Summary Vector and Locality Preserved Caching.

We use two internal datasets for our experiment. One is a daily full backup of a company-wide Exchange information store over a 135-day period. The other is the weekly full and daily incremental backup of an Engineering department over a 100-day period. Table 3 summarizes key attributes of these two datasets.

These internal datasets are generated from production usage (albeit internal). We also observe that various compression ratios produced by the internal datasets are relatively similar to those of real world examples examined in section 5.1. We believe these internal datasets are reasonable proxies of real world deployments.

Each of the backup datasets is sent to the deduplicating storage system with a single backup stream. With respect to the deduplication storage system, we measure the number of disk reads for segment index lookups and locality prefetches needed to find duplicates during write for four cases:

(1) with neither Summary Vector nor Locality Preserved Caching;

(2) with Summary Vector only;

(3) with Locality Preserved Caching only; and

(4) with both Summary Vector and Locality Preserved Caching.

The results are shown in Table 4.

Clearly, the Summary Vector and Locality Preserved Caching combined have produced an astounding reduction in disk reads. Summary Vector alone reduces about 16.5% and 18.6% of the index lookup disk I/Os for exchange and engineering data respectively. The Locality Preserved Caching alone reduces about 82.4% and 81% of the index lookup disk I/Os for exchange and engineering data respectively. Together they are able to reduce the index lookup disk I/Os by 98.94% and 99.6% respectively.

In general, the Summary Vector is very effective for new data, and Locality Preserved Caching is highly effective for little or moderately changed data. For backup data, the first full backup (seeding equivalent) does not have as many duplicate data segments as subsequent full backups. As a result, the Summary Vector is effective to avoid disk I/Os for the index lookups during the first full backup, whereas Locality Preserved Caching is highly beneficial for subsequent full backups. This result also suggests that these two datasets exhibit good duplicate locality.

5.3 Throughput

To determine the throughput of the deduplication storage system, we used a synthetic dataset driven by client computers. The synthetic dataset was developed to model backup data from multiple backup cycles from multiple backup streams, where each backup stream can be generated on the same or a different client computer.

The dataset is made up of synthetic data generated on the fly from one or more backup streams. Each backup stream is made up of an ordered series of synthetic data versions where each successive version ("generation") is a somewhat modified copy of the preceding generation in the series. The generation-to-generation modifications include: data reordering, deletion of existing data, and addition of new data. Single-client backup over time is simulated when synthetic data generations from a backup stream are written to the deduplication storage system in generation order, where significant amounts of data are unchanged day-to-day or week-to-week, but where small changes continually accumulate. Multi-client backup over time is simulated when synthetic data generations from multiple streams are written to the deduplication system in parallel, each stream in the generation order.

There are two main advantages of using the synthetic dataset. The first is that various compression ratios can be built into the synthetic model, and usages approximating various real world deployments can be tested easily in house.

The second is that one can use relatively inexpensive client computers to generate an arbitrarily large amount of synthetic data in memory without disk I/Os and write in one stream to the deduplication system at more than 100 MB/s. Multiple cheap client computers can combine in multiple streams to saturate the intake of the deduplication system in a switched network environment. We find it both much more costly and technically challenging using traditional backup software, high-end client computers attached to primary storage arrays as backup clients, and high–end servers as media/backup servers to accomplish the same feat.

In our experiments, we choose an average generation (daily equivalent) global compression ratio of 30, and an average generation (daily equivalent) local compression ratio of 2 to 1 for each backup stream. These compression numbers seem possible given the real world examples in section 5.1. We measure throughput for one

	Exchange data		Engineering data	
	# disk I/Os	% of total	# disk I/Os	% of total
no Summary Vector and no Locality Preserved Caching	328,613,503	100.00%	318,236,712	100.00%
Summary Vector only	274,364,788	83.49%	259,135,171	81.43%
Locality Preserved Caching only	57,725,844	17.57%	60,358,875	18.97%
Summary Vector and Locality Preserved Caching	3,477,129	1.06%	1,257,316	0.40%

Table 4: Index and locality reads. This table shows the number disk reads to perform index lookups or fetches from the container metadata for the four combinations: with and without the Summary Vector and with and without Locality Preserved Caching. Without either the Summary Vector or Locality Preserved Caching, there is an index read for every segment. The Summary Vector avoids these reads for most new segments. Locality Preserved Caching avoids index lookups for duplicate segments at the cost an extra read to fetch a group of segment fingerprints from the container metadata for every cache miss for which the segment is found in the index.

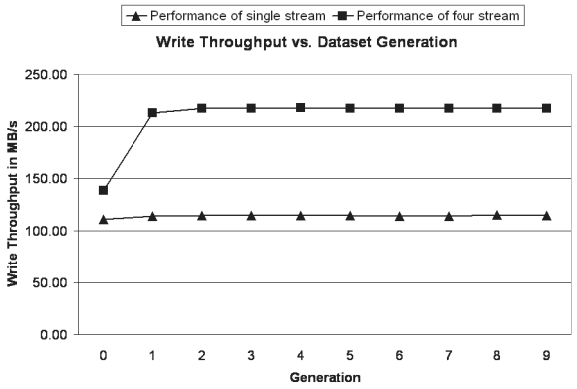

Figure 8: Write Throughput of Single Backup Client and 4 Backup Clients.

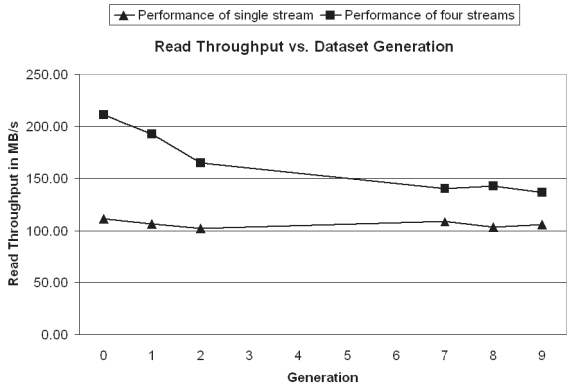

Figure 9: Read Throughput of Single Backup Client and 4 Backup Clients

backup stream using one client computer and 4 backup streams using two client computers for write and read for 10 generations of the backup datasets. The results are shown in Figures 8 and 9.

The deduplication system delivers high write throughput results for both cases. In the single stream case, the system achieves write throughput of 110 MB/sec for generation 0 and over 113 MB/sec for generation 1 through 9. In the 4 stream case, the system achieves write throughput of 139 MB/sec for generation 0 and a sustained 217 MB/sec for generation 1 through 9.

Write throughput for generation 0 is lower because all segments are new and require Ziv-Lempel style compression by the CPU of the deduplication system.

The system delivers high read throughput results for the single stream case. Throughout all generations, the system achieves over 100 MB/sec read throughput.

For the 4 stream case, the read throughput is 211 MB/sec for generation 0, 192 MB/sec for generation 1, 165 MB/sec for generation 2, and stay at around 140 MB/sec for future generations. The main reason for the decrease of read throughput in the later generations is that future generations have more duplicate data segments than the first few. However, the read throughput stays at about 140 MB/sec for later generations because of Stream-Informed Segment Layout and Locality Preserved Caching.

Note that write throughput has historically been valued more than read throughput for the backup use case since backup has to complete within a specified backup window time period and it is much more frequent event than restore. Read throughput is still very important, especially in the case of whole system restores.

5.4 Discussion

The techniques presented in this paper are general methods to improve throughput performance of deduplication storage systems. Although our system divides a data stream into content-based segments, these methods can also apply to system using fixed aligned segments such as Venti.

As a side note, we have compared the compression ratios of a system segmenting data streams by contents (about 8Kbytes on average) with another system using fixed aligned 8Kbytes segments on the engineering and exchange backup datasets. We found that the fixed alignment approach gets basically no global compression (global compression: 1.01) for the engineering data, whereas the system with content-based segmentation gets a lot of global compression (6.39:1). The main reason of the difference is that the backup software creates the backup dataset without realigning data at file boundaries. For the exchange backup dataset where the backup software aligns data at individual mailboxes, the global compression difference is less (6.61:1 vs. 10.28:1), though there is a significant gap.

Fragmentation will become more severe for long term retention, and can reduce the effectiveness of Locality Preserved Caching. We have investigated mechanisms to reduce fragmentation and sustain high write and read throughput. But, these mechanisms are beyond the scope of this paper.

6 Related Work

Much work on deduplication focused on basic methods and compression ratios, not on high throughput.

Early deduplication storage systems use file-level hashing to detect duplicate files and reclaim their storage space [ABCC*02, TKSK*03, KDLT04]. Since such

systems also use file hashes to address files. Some call such systems content addressed storage or CAS. Since their deduplication is at file level, such systems can achieve only limited global compression.

Venti removes duplicate fixed-size data blocks by comparing their secure hashes [QD02]. It uses a large on-disk index with a straightforward index cache to lookup fingerprints. Since fingerprints have no locality, their index cache is not effective. When using 8 disks to lookup fingerprints in parallel, its throughput is still limited to less than 7 MB/sec. Venti used a container abstraction to layout data on disks, but was stream agnostic, and did not apply Stream-Informed Segment Layout.

To tolerate shifted contents, modern deduplication systems remove redundancies at variable-size data blocks divided based on their contents. Manber described a method to determine anchor points of a large file when certain bits of rolling fingerprints are zeros [Man93] and showed that Rabin fingerprints [Rab81, Bro93] can be computed efficiently. Brin et al. [BDH94] described several ways to divide a file into content-based data segments and use such segments to detect duplicates in digital documents. Removing duplications at content-based data segment level has been applied to network protocols and applications [SW00, SCPC*02, RLB03, MCK04] and has reduced network traffic for distributed file systems [MCM01, JDT05]. Kulkarni et al. evaluated the compression efficiency between an identity-based (fingerprint comparison of variable-length segments) approach and a delta-compression approach [KDLT04]. These studies have not addressed deduplication throughput issues.

The idea of using Bloom filter [Blo70] to implement the Summary Vector is inspired by the summary data structure for the proxy cache in [FCAB98]. Their work also provided analysis for false positive rate. In addition, Broder and Mitzenmacher wrote an excellent survey on network applications of Bloom filters [AM02]. TAPER system used a Bloom filter to detect duplicates instead of detecting if a segment is new [JDT05]. It did not investigate throughput issues.

7 Conclusions

This paper presents a set of techniques to substantially reduce disk I/Os in high-throughput deduplication storage systems.

Our experiments show that the combination of these techniques can achieve over 210 MB/sec for 4 multiple write data streams and over 140 MB/sec for 4 read data streams on storage server with two dual-core processors and one shelf of 15 drives.

We have shown that Summary Vector can reduce disk index lookups by about 17% and Locality Preserved Caching can reduce disk index lookups by over 80%, but the combined caching techniques can reduce disk index lookups by about 99%.

Stream-Informed Segment Layout is an effective abstraction to preserve spatial locality and enable Locality Preserved Caching.

These techniques are general methods to improve throughput performance of deduplication storage systems. Our techniques for minimizing disk I/Os to achieve good deduplication performance match well against the industry trend of building many-core processors. With quad-core CPU's already available, and eight-core CPU's just around the corner, it will be a relatively short time before a large-scale deduplication storage system shows up with 400 ~ 800 MB/sec throughput with a modest amount of physical memory.

8 References

[ABCC*02] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *Proceedings of USENIX Operating Systems Design and Implementation (OSDI)*, December 2002.

[BM05] Andrie Z. Broder and Michael Mitzenmacher. Network Applications of Bloom Filters: A Survey. *Internet Mathematics*, 2005.

[BDH94] S. Brin, J. Davis, H. Carcia-Molina. Copy Detection Mechanisms for Digital Documents (weblink). 1994, also lso in *Proceedings of ACM SIGMOD*, 1995.

[Blo70] Burton H. Bloom. Space/time Trade-offs in Hash Coding with Allowable Errors. *Communications of the ACM*, *13* (7). 422-426.

[JDT05] N. Jain, M. Dahlin, and R. Tewari. TAPER: Tiered Approach for Eliminating Redundancy in Replica Synchronization. In Proceedings of *USENIX File And Storage Systems (FAST)*, 2005.

[Dat05] Data Domain, Data Domain Appliance Series: High-Speed Inline Deduplication Storage, 2005, http://www.datadomain.com/products/appliances.html

[FCAB98] Li Fan, Pei Cao, Jussara Almeida, and Andrie Z. Broder. Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol. in *Proceedings of ACM SIGCOMM'98*, (Vancouver, Canada, 1998).

[KDLT04] P. Kulkarni, F. Douglis, J. D. LaVoie, J. M. Tracey: Redundancy Elimination Within Large Collections of Files. In *Proceedings of USENIX Annual Technical Conference*, pages 59-72, 2004.

[Man93] Udi Manber. Finding Similar Files in A Large File System. Technical Report TR 93-33, Department of Computer Science, University of Arizona, October 1993, also in *Proceedings of the USENIX Winter 1994 Technical Conference*, pages 17–21. 1994.

[MCK04] J. C. Mogul, Y.-M. Chan, and T. Kelly. Design, implementation, and evaluation of duplicate transfer detection in HTTP. In *Proceedings of Network Systems Design and Implementation*, 2004.

[MCM01] Athicha Muthitacharoen, Benjie Chen, and David Mazières. A Low-bandwidth Network File System. In *Proceedings of the ACM 18th Symposium on Operating Systems Principles.* Banff, Canada. October, 2001.

[NIST95] National Institute of Standards and Technology, FIPS 180-1. Secure Hash Standard. US Department of Commerce, April 1995.

[PJSS*94] B. Pawlowski, C. Juszczak, P. Staubach, C. Smith, D. Lebel, and D. Hitz, NFS Version 3 Design and Implementation, In *Proceedings of the USENIX Summer 1994 Technical Conference*. 1994.

[QD02] S. Quinlan and S. Dorward, Venti: A New Approach to Archival Storage. In *Proceedings of the USENIX Conference on File And Storage Technologies (FAST)*, January 2002.

[RLB03] S. C. Rhea, K. Liang, and E. Brewer. Value-based web caching. In *WWW*, pages 619–628, 2003.

[SCPC*02] C. P. Sapuntzakis, R. Chandra, B. Pfaff, J. Chow, M. S. Lam, and M. Rosenblum. Optimizing the migration of virtual computers. In *Proceedings of USENIX Operating Systems Design and Implementation*, 2002.

[SW00] N. T. Spring and D. Wetherall. A protocol-independent technique for eliminating redundant network traffic. In *Proceedings of ACM SIGCOMM*, pages 87--95, Aug. 2000.

[TKSK*03] N. Tolia, M. Kozuch, M. Satyanarayanan, B. Karp, A. Perrig, and T. Bressoud. Opportunistic use of content addressable storage for distributed file systems. In *Proceedings of the 2003 USENIX Annual Technical Conference*, pages 127–140, San Antonio, TX, June 2003.

[YPL05] L. L. You, K. T. Pollack, and D. D. E. Long. Deep Store: An archival storage system architecture. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE '05)*, April 2005.

[ZL77] J. Ziv and A. Lempel. A universal algorithm for sequential data compression, *IEEE Trans. Inform. Theory*, vol. IT-23, pp. 337-343, May 1977.