Sparse Indexing: Large Scale, Inline Deduplication Using Sampling and Locality

Mark Lillibridge[†], Kave Eshghi[†], Deepavali Bhagwat[‡], Vinay Deolalikar[†], Greg Trezise[♯], and Peter Camble[♯]
[†]*HP Labs* [‡]*UC Santa Cruz* [♯]*HP Storage Works Division*
first.last@hp.com

Abstract

We present *sparse indexing*, a technique that uses sampling and exploits the inherent locality within backup streams to solve for large-scale backup (e.g., hundreds of terabytes) the chunk-lookup disk bottleneck problem that inline, chunk-based deduplication schemes face. The problem is that these schemes traditionally require a full chunk index, which indexes every chunk, in order to determine which chunks have already been stored; unfortunately, at scale it is impractical to keep such an index in RAM and a disk-based index with one seek per incoming chunk is far too slow.

We perform stream deduplication by breaking up an incoming stream into relatively large segments and deduplicating each segment against only a few of the most similar previous segments. To identify similar segments, we use sampling and a sparse index. We choose a small portion of the chunks in the stream as samples; our sparse index maps these samples to the existing segments in which they occur. Thus, we avoid the need for a full chunk index. Since only the sampled chunks' hashes are kept in RAM and the sampling rate is low, we dramatically reduce the RAM to disk ratio for effective deduplication. At the same time, only a few seeks are required per segment so the chunk-lookup disk bottleneck is avoided. Sparse indexing has recently been incorporated into number of Hewlett-Packard backup products.

1 Introduction

Traditionally, magnetic tape has been used for data back up. With the explosion in disk capacity, it is now affordable to use disk for data backup. Disk, unlike tape, is random access and can significantly speed up backup and restore operations. Accordingly, disk-to-disk backup (D2D) has become the preferred backup option for organizations [3].

Deduplication can increase the effective capability of a D2D device by one or two orders of magnitude [4]. Deduplication can accomplish this because backup sets have massive redundancy due to the facts that a large proportion of data does not change between backup sessions and that files are often shared between machines. Deduplication, which is practical only with random-access devices, removes this redundancy by storing duplicate data only once and has become an essential feature of disk-to-disk backup solutions.

We believe chunk-based deduplication is the deduplication method best suited to D2D: it deduplicates data both across backups and within backups and does not require any knowledge of the backup data format. With this method, data to be deduplicated is broken into variable-length chunks using content-based chunk boundaries [20], and incoming chunks are compared with the chunks in the store by hash comparison; only chunks that are not already there are stored. We are interested in *inline* deduplication, where data is deduplicated as it arrives rather than later in batch mode, because of its capacity, bandwidth, and simplicity advantages (see Section 2.2).

Unfortunately, inline, chunk-based deduplication when used at large scale faces what is known as the *chunk-lookup disk bottleneck problem*: Traditionally, this method requires a full chunk index, which maps each chunk's hash to where that chunk is stored on disk, in order to determine which chunks have already been stored. However, at useful D2D scales (e.g., 10-100 TB), it is impractical to keep such a large index in RAM and a disk-based index with one seek per incoming chunk is far too slow (see Section 2.3).

This problem has been addressed in the literature by Zhu *et al.* [28], who tackle it by using an in-memory Bloom Filter and caching index fragments, where each fragment indexes a set of chunks found together in the input. In this paper, we show a different way of solving this problem in the context of data stream deduplication (the D2D case). Our solution has the advantage that it uses significantly less RAM than Zhu *et al.*'s approach.

To solve the chunk-lookup disk bottleneck problem, we rely on *chunk locality*: the tendency for chunks in backup data streams to reoccur together. That is, if the last time we encountered chunk A, it was surrounded by chunks B, C, and D, then the next time we encounter A (even in a different backup) it is likely that we will also encounter B, C, or D nearby. This differs from traditional notions of locality because occurrences of A may be separated by very long intervals (e.g., terabytes). A derived property we take advantage of is that if two pieces of backup streams share any chunks, they are likely to share many chunks.

We perform stream deduplication by breaking up each input stream into segments, each of which contains thousands of chunks. For each segment, we choose a few of the most similar segments that have been stored previously. We deduplicate each segment against only its chosen few segments, thus avoiding the need for a full chunk index. Because of the high chunk locality of backup streams, this still provides highly effective deduplication.

To identify similar segments, we use sampling and a sparse index. We choose a small portion of the chunks as samples; our sparse index maps these samples' hashes to the already-stored segments in which they occur. By using an appropriate low sampling rate, we can ensure that the sparse index is small enough to fit easily into RAM while still obtaining excellent deduplication. At the same time, only a few seeks are required per segment to load its chosen segments' information avoiding any disk bottleneck and achieving good throughput.

Of course, since we deduplicate each segment against only a limited number of other segments, we occasionally store duplicate chunks. However, due to our lower RAM requirements, we can afford to use smaller chunks, which more than compensates for the loss of deduplication the occasional duplicate chunk causes. The approach described in this paper has recently been incorporated into a number of Hewlett-Packard backup products.

The rest of this paper is organized as follows: in the next section, we provide more background. In Section 3, we describe our approach to doing chunk-based deduplication. In Section 4, we report on various simulation experiments with real data, including a comparison with Zhu *et al.*, and on the ongoing productization of this work. Finally, we describe related work in Section 5 and our conclusions in Section 6.

2 Background

2.1 D2D usage

There are two modes in which D2D is performed today, using a network-attached-storage (NAS) protocol and using a Virtual Tape Library (VTL) protocol:

In the NAS approach, the backup device is treated as a network-attached storage device, and files are copied to it using protocols such as NFS and CIFS. To achieve high throughput, typically large directory trees are coalesced, using a utility such as tar, and the resulting tar file stored on the backup device. Note that tar can operate either in incremental or in full mode.

The VTL approach is for backward compatibility with existing backup agents. There is a large installed base of thousands of backup agents that send their data to tape libraries using a standard tape library protocol. To make the job of migrating to disk-based backup easier, vendors provide Virtual Tape Libraries: backup storage devices that emulate the tape library protocol for I/O, but use disk-based storage internally.

In both NAS and VTL-based D2D, the backup data is presented to the backup storage device as a stream. In the case of VTL, the stream is the virtual tape image, and in the case of NAS-based backup, the stream is the large tar file that is generated by the client. In both cases, the stream can be quite large: a single tape image can be 400 GB, for example.

2.2 Inline versus out-of-line deduplication

Inline deduplication refers to deduplication processes where the data is deduplicated as it arrives and before it hits disk, as opposed to out-of-line (also called post-process) deduplication where the data is first accumulated in an on-disk holding area and then deduplicated later in batch mode. With out-of-line deduplication, the chunk-lookup disk bottleneck can be avoided by using batch processing algorithms, such as hash join [24], to find chunks with identical hashes.

However, out-of-line deduplication has several disadvantages compared to inline deduplication: (a) the need for an on-disk holding area large enough to hold an entire backup window's worth of raw data can substantially diminish storage capacity,[1] (b) all the functionality that a D2D device provides (data restoration, data replication, compression, etc.) must be implemented and/or tested separately for the raw holding area as well as the deduplicated store, and (c) it is not possible to conserve network or disk bandwidth because every chunk must be written to the holding area on disk.

2.3 The chunk-lookup disk bottleneck

The traditional way to implement inline, chunk-based deduplication is to use a *full chunk index*: a key-value index of all the stored chunks, where the key is a chunk's hash, and the value holds metadata about that chunk, including where it is stored on disk [22, 14]. When an incoming chunk is to be stored, its hash is looked up in the full index, and the chunk is stored only if no entry is found for its hash. We refer to this approach as the full index approach.

Using a small chunk size is crucial for high-quality chunk-based deduplication because most duplicate data regions are not particularly large. For example, for our data set Workgroup (see Section 4.2), switching from 4 to 8 KB average-size chunks reduces the deduplication factor (original size/deduplicated size) from 13 to 11; switching to 16 KB chunks further reduces it to 9.

This need for a small chunk size means that the full chunk index consumes a great deal of space for large stores. Consider, for example, a store that contains 10 TB of unique data and uses 4 KB chunks. Then there are 2.7×10^9 unique chunks. Assuming that every hash entry in the index consumes 40 bytes, we need 100 GB of storage for the full index.

It is not cost effective to keep all of this index in RAM. However, if we keep the index on disk, due to the lack of short-term locality in the stream of incoming chunk hashes, we will need one disk seek per chunk hash lookup. If a seek on average takes 4 ms, this means we can look up only 250 chunks per second for a processing rate of 1 MB/s, which is not acceptable. This is the chunk-lookup disk bottleneck that needs to be avoided.

3 Our Approach

Under the sparse indexing approach, *segments* are the unit of storage and retrieval. A segment is a sequence of chunks. Data streams are broken into segments in a two step process: first, the data stream is broken into a sequence of variable-length chunks using a chunking algorithm, and, second, the resulting chunk sequence is broken into a sequence of segments using a segmenting algorithm. Segments are usually on the order of a few megabytes. We say that two segments are similar if they share a number of chunks.

Segments are represented in the store using their *manifests*: a manifest or segment recipe [25] is a data structure that allows reconstructing a segment from its chunks, which are stored separately in one or more chunk containers to allow for sharing of chunks between segments. A segment's manifest records its sequence of chunks, giving for each chunk its hash, where it is stored on disk, and possibly its length. Every stored segment has a manifest that is stored on disk.

Incoming segments are deduplicated against similar, existing segments in the store. Deduplication proceeds in two steps: first, we identify among all the segments in the store some that are most similar to the incoming segment, which we call *champions*, and, second, we deduplicate against those segments by finding the chunks they share with the incoming segment, which do not need to be stored again.

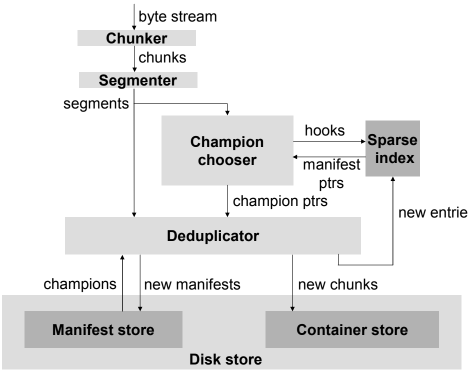

Figure 1: **Block diagram of the deduplication process**

To identify similar segments, we sample the chunk hashes of the incoming segment, and use an in-RAM index to determine which already-stored segments contain how many of those hashes. A simple and fast way to sample is to choose as a sample every hash whose first n bits are zero; this results in an average *sampling rate* of $1/2^n$; that is, on average one in 2^n hashes is chosen as a sample. We call the chosen hashes *hooks*.

The in-memory index, called the *sparse index*, maps hooks to the manifests in which they occur. The manifests themselves are kept on disk; the sparse index holds only pointers to them. Once we have chosen champions, we can load their manifests into RAM and use them to deduplicate the incoming segment. Note that although we choose champions because they share hooks with the incoming segment (and thus, the chunks with those hashes), as a consequence of chunk locality they are likely to share many other chunks with the incoming segment as well.

We will now describe the deduplication process in more detail. A block diagram of the process can be found in Figure 1.

3.1 Chunking and segmenting

Content-based chunking has been studied at length in the literature [1, 16, 20]. We use our Two-Threshold Two-Divisor (TTTD) chunking algorithm [13] to subdivide the incoming data stream into chunks. TTTD produces variable-sized chunks with smaller size variation than other chunking algorithms, leading to superior deduplication.

We consider two different segmentation algorithms in this paper, each of which takes a target segment size as a parameter. The first algorithm, *fixed-size segmentation*, chops the stream of incoming chunks just before the first chunk whose inclusion would make the current segment longer than the goal segment length. "Fixed-sized" segments thus actually have a small amount of size variation because we round down to the nearest chunk boundary. We believe that it is important to make segment boundaries coincide with chunk boundaries to avoid split chunks, which have no chance of being deduplicated.

Because we perform deduplication by finding segments similar to an incoming segment and deduplicating against them, it is important that the similarity between an incoming segment and the most similar existing segments in the store be as high as possible. Fixed-size segmentation does not perform as well here as we would like because of the *boundary-shifting problem* [13]: Consider, for example, two data streams that are identical except that the first stream has an extra half-a-segment size worth of data at the front. With fixed-size segmentation, segments in the second stream will only have 50% overlap with the segments in the first stream, even though the two streams are identical except for some data at the start of the first stream.

To avoid the segment boundary-shifting problem, our second segmentation algorithm, *variable-size segmentation*, uses the same trick used at the chunking level to avoid the boundary-shifting problem: we base the boundaries on landmarks in the content, not distance. Variable-size segmentation operates at the level of chunks (really chunk hashes) rather than bytes and places segment boundaries only at existing chunk boundaries. The start of a chunk is considered to represent a landmark if that chunk's hash modulo a predetermined divisor is equal to -1. The frequency of landmarks—and hence average segment size—can be controlled by varying the size of the divisor.

To reduce segment-size variation, variable-size segmentation uses TTTD applied to chunks instead of data bytes. The algorithm is the same, except that we move one chunk at a time instead of one byte at a time, and that we use the above notion of what a landmark is. Note that this ignores the lengths of the chunks, treating long and short chunks the same. We obtain the needed TTTD parameters (minimum size, maximum size, primary divisor, and secondary divisor) in the usual way from the desired average size. Thus, for example, with variable-size segmentation, mean size 10 MB segments using 4 KB chunks have from 1,160 to 7,062 chunks with an average of 2,560 chunks, each chunk of which, on average, contains 4 KB of data.

3.2 Choosing champions

Looking up the hooks of an incoming segment S in the sparse index results in a possible set of manifests against which that segment can be deduplicated. However, we do not necessarily want to use all of those manifests to deduplicate against, since loading manifests from disk is costly. In fact, as we show in Section 4.3, only a few well chosen manifests suffice. So, from among all the manifests produced by querying the sparse index, we choose a few to deduplicate against. We call the chosen manifests champions.

The algorithm by which we choose champions is as follows: we choose champions one at a time until the maximum allowable number of champions are found, or we run out of candidate manifests. Each time we choose, we choose the manifest with the highest non-zero score, where a manifest gets one point for each hook it has in common with S that is not already present in any previously chosen champion. If there is a tie, we choose the manifest most recently stored. The choice of which manifests to choose as champions is done based solely on the hooks in the sparse index; that is, it does not involve any disk accesses.

We don't give points for hooks belonging to already chosen manifests because those chunks (and the chunks around them by chunk locality) are most likely already covered by the previous champions. Consider the following highly-simplified example showing the hooks of S and three candidate manifests (m_1–m_3):

S	**b**	**c**	**d**	**e**	**m**	**n**
m_1	a	**b**	**c**	**d**	**e**	f
m_2	z	a	**b**	**c**	**d**	f
m_3	**m**	**n**	o	p	q	r

The manifests are shown in descending order of how many hooks they have in common with S (common hooks shown in bold). Our algorithm chooses m_1 then m_3, which together cover all the hooks of S, unlike m_1 and m_2.

3.3 Deduplicating against the champions

Once we have determined the champions for the incoming segment, we load their manifests from disk. A small cache of recently loaded manifests can speed this process up somewhat because adjacent segments sometimes have champions in common.

The hashes of the chunks in the incoming segment are then compared with the hashes in the champions' manifests in order to find duplicate chunks. We use the SHA1 hash algorithm [15] to make false positives here extremely unlikely. Those chunks that are found not to be present in any of the champions are stored on disk in chunk containers, and a new manifest is created for the incoming segment. The new manifest contains the location on disk where each incoming chunk is stored. In the case of chunks that are duplicates of a chunk in one or more of the champions, the location is the location of the existing chunk, which is obtained from the relevant manifest. In the case of new chunks, the on-disk location is where that chunk has just been stored. Once the new manifest is created, it is stored on disk in the manifest store.

Finally, we add entries for this manifest to the sparse index with the manifest's hooks as keys. Some of the hooks may already exist in the sparse index, in which case we add the manifest to the list of manifests that are pointed to by that hook. To conserve space, it may be desirable to set a maximum limit for the number of manifests that can be pointed to by any one hook. If the maximum is reached, the oldest manifest is removed from the list before the newest one is added.

3.4 Avoiding the chunk-lookup disk bottleneck

Notice that there is no full chunk index in our approach, either in RAM or on disk. The only index we maintain in RAM, the sparse index, is much smaller than a full chunk index: for example, if we only sample one out of every 128 hashes, then the sparse index can be 128 times smaller than a full chunk index.

We do have to make a handful of random disk accesses per segment in order to load in champion manifests, but the cost of those seeks is amortized over the thousands of chunks in each segment, leading to acceptable throughput. Thus, we avoid the chunk-lookup disk bottleneck.

3.5 Storing chunks

We do not have room in this paper, alas, to describe how best to store chunks in chunk containers. The scheme described in Zhu *et al.* [28], however, is a pretty good starting point and can be used with our approach. They maintain an open chunk container for each incoming stream, appending each new (unique) chunk to the open container corresponding to the stream it is part of. When a chunk container fills up (they use a fixed size for efficient packing), a new one is opened up.

This process uses chunk locality to group together chunks likely to be retrieved together so that restoration performance is reasonable. Supporting deletion of segments requires additional machinery for merging mostly empty containers, garbage collection (when is it safe to stop storing a shared chunk?), and possibly defragmentation.

3.6 Using less bandwidth

We have described a system where all the raw backup data is fed across the network to the backup system and only then deduplicated, which may consume a lot of network bandwidth. It is possible to use substantially less bandwidth at the cost of some client-side processing if the legacy backup clients could be modified or replaced. One way of doing this is to have the backup client perform the chunking, hashing, and segmentation locally. The client initially sends only a segment's chunks' hashes to the back-end, which performs champion choosing, loads the champion manifests, and then determines which of those chunks need to be stored. The back-end notifies the client of this and the client sends only the chunks that need to be stored, possibly compressed.

4 Experimental Results

In order to test our approach, we built a simulator that allows us to experiment with a number of important parameters, including some parameter values that are infeasible in practice (e.g., using a full index). We apply our simulator to two realistic data sets and report below on locality, overall deduplication, RAM usage, and throughput. We also report briefly on some optimizations and an ongoing productization that validates our approach.

4.1 Simulator

Our simulator takes as input a series of (chunk hash, length) pairs, divides it into segments, determines the champions for each segment, and then calculates the amount of deduplication obtained. Available knobs include type of segmentation (fixed or variable size), mean segment size, sampling rate, maximum number of champions loadable per segment, how many manifest IDs to keep per hook in the sparse index, and whether or not to use a simple form of manifest caching (see Section 4.7).

We (or others when privacy is a concern) run a small tool we have written called *chunklite* in order to produce chunk information for the simulator. Chunklite reads from either a mounted tape or a list of files, chunking using the TTTD chunking algorithm [13]. Except where we say otherwise, all experiments use chunklite's default 4 KB mean chunk size,[2] which we find a good trade-off between maximizing deduplication and minimizing per-chunk overhead.

The simulator produces various statistics, including the sum of lengths of every input chunk (original size) and the sum of lengths of every non-removed chunk (deduplicated size). The estimated deduplication factor is then original size/deduplicated size.

4.2 Data sets

We report results for two data sets. The first data set, which we call *Workgroup*, is composed of a semi-regular series of backups of the desktop PCs of a group of 20 engineers taken over a period of three months. Although the original collection included only an initial full and later weekday incrementals for each machine, we have generated synthetic fulls at the end of each week for which incrementals are available by applying that week's incrementals to the last full. The synthetic fulls replace the last incremental for their week; in this way, we simulate a more typical daily incremental and weekly full backup schedule. We are unable to simulate file deletions because this information is missing from the collection.

Altogether, there are 154 fulls and 392 incrementals in this 3.8 TB data set, which consists of each of these backup snapshots tar'ed up without compression in the order they were taken. We believe this data set is representative of a small corporate workgroup being backed up via tar directly to a NAS interface. Note that because these machines are only powered up during workdays and because the synthetic fulls replace the last day of the week's back up, the ratio of incrementals to fulls (2.5) is lower than would be the case for a server (6 or 7).

The second data set, which we call *SMB*, is intended, by contrast, to be representative of a small or medium business server backed up to virtual tape. It contains two weeks (3 fulls, 12 incrementals) of Oracle & Exchange data backed up via Symantec's NetBackup to virtual tape. The Exchange data was synthetic data generated by the Microsoft Exchange Server 2003 Load Simulator (LoadSim) tool [19], while the Oracle data was created by inserting rows from a real 1+ TB Oracle database belonging to a compliance test group combined with a small number of random deletes and updates. This data set occupies 0.6 TB and has less duplication than one might expect because Exchange already uses single instance storage (each message is stored only once no matter how many users receive it) and because NetBackup does true Exchange incrementals, saving only new/changed messages.

We have chosen data sets with daily incrementals and weekly fulls rather than just daily fulls because such data sets are harder to deduplicate well, and thus provide a better test of any deduplication system. Incrementals are harder to deduplicate because they contain less duplicate material and because they have less locality: any given incremental segment likely contains files from many segments of the previous full whereas a full segment may only contain files from one or two segments of the previous full. Series of all fulls do generate higher deduplication factors, beloved of marketing departments everywhere, however.

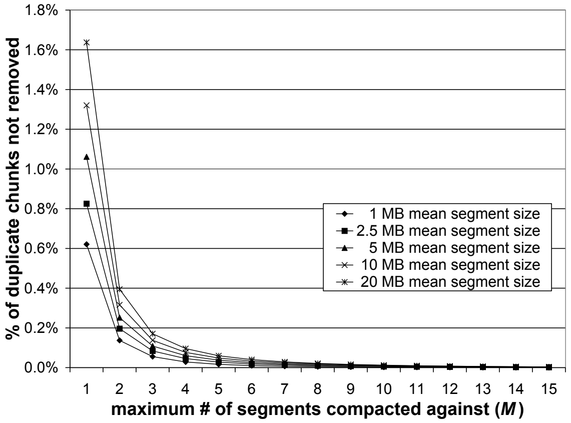

Figure 2: **Conservative estimate (GREEDY) of deduplication effectiveness obtainable by deduplicating each segment against up to M prior segments** for data set **Workgroup**. Shown are results for 5 different segment sizes, with all segments chosen via variable-size segmentation.

4.3 Locality

In order for our approach to work, there must be sufficient locality present in real backup streams. In particular, we need locality at the scale of our segment size so that most of the deduplication possible for a given segment can be obtained by deduplicating it against a small number of prior segments. The existence of such locality is a necessary, but not sufficient condition: the existence of such segments does not automatically imply that sparse indexing or any other method can efficiently find them.

Whether or not such locality exists is an empirical question, which we find to be overwhelmingly answered in the affirmative. Figures 2 and 3 show a conservative estimate of this locality for our data sets for a variety of segment sizes. Here, we show how well segment-based deduplication could work given near-perfect segment choice when each segment of the given data set can only be deduplicated against a small number M of prior segments. We measure deduplication effectiveness by the percentage of duplicate chunks that deduplication fails to remove; the smaller this number, the better the deduplication.

Because computing the optimal segments to deduplicate against is infeasible in practice, we instead estimate the deduplication effectiveness possible by using a simple greedy algorithm (GREEDY) that chooses the segments to deduplicate a given segment S against one at a time, each time choosing the segment that will produce the maximum additional deduplication. While GREEDY

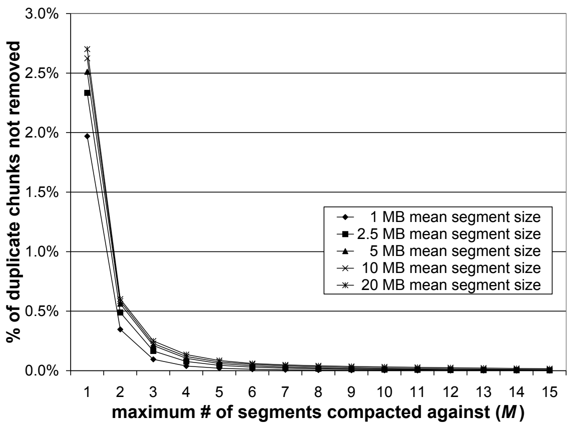

Figure 3: **Conservative estimate (GREEDY) of deduplication effectiveness obtainable by deduplicating each segment against up to M prior segments** for data set **SMB**. Shown are results for 5 different segment sizes, with all segments chosen via variable-size segmentation.

does an excellent job of choosing segments, it consumes too much RAM to ever be practical.

As you can see, there is a great deal of locality at these scales: deduplicating each input segment against only 2 prior segments can suffice to remove all but 1% of the duplicate chunks (0.1% requires only 3 more). Not shown is the zero segment case ($M = 0$) where 93-98% of duplicate chunks remain due to duplication within segments (segments are automatically deduplicated against themselves). Larger segment sizes yield slightly less locality, presumably because larger pieces of incrementals include data from more faraway places.

Likely sources of locality in backup streams include writing out entire large items even when only a small part has changed (e.g., Microsoft Outlook's mostly append-only PST files, which are often hundreds of megabytes long), locality in the order items are scanned (e.g., always scanning files in alphabetical order), and the tendency for changes to be clustered in small areas.

4.4 Overall deduplication

How much of this locality are we able to exploit using sampling? Figure 4 addresses this point by showing for data set Workgroup and 10 MB variable size segments how much of the possible deduplication efficiency we obtain. Even with a sampling rate as low as 1/128, we remove all but 1.4% of the duplicate data given a maximum of 10 or more champions per segment (0.7% for 1/64).

Figures 5 and 6 show the overall deduplication produced by applying our approach to the two data sets.

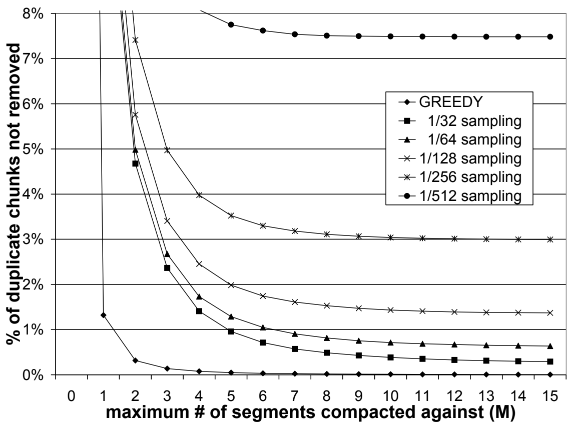

Figure 4: **Deduplication efficiency obtained by using sparse indexing with 10 MB average-sized segments for various maximum numbers of champions (M) and sampling rates** for data set **Workgroup**. Shown for comparison is GREEDY's results given the same data. Variable-size segmentation was used.

As can be seen, the degree of deduplication achieved falls off as the sampling rate decreases and as the segment size decreases. The amount of deduplication remains roughly constant as sampling rate is traded off against segment size: halving the sampling rate and doubling the mean segment size leaves deduplication roughly the same. This can be seen most easily in Figure 7, which plots overall deduplication versus the average number of hooks per segment (equal to segment size/chunk size×sampling rate). We believe this relationship reflects t7 he fact that deduplication quality using sparse indexing depends foremost on the number of hooks per segment.

Note that these figures show simulated deduplication, not real deduplication. In particular, they take into account only the space required to keep the data of the non-deduplicated chunks. Including container padding, the space required to store manifests, and other overhead would reduce these numbers somewhat. Similar overhead exists in all backup systems that use chunk-based deduplication. On the other hand, these numbers do not include any form of local compression of chunk data. In practice, chunks would be compressed (e.g., by Ziv-Lempel [29]), either individually or in groups, before storing to disk. Such compression usually adds an additional factor of 1.5–2.5.

Using variable instead of fixed-size segmentation improves deduplication using sparse indexing as can be seen from Figure 8. This improvement is due to increased locality: with fixed-size segmentation, there are more segments that produce substantial deduplication

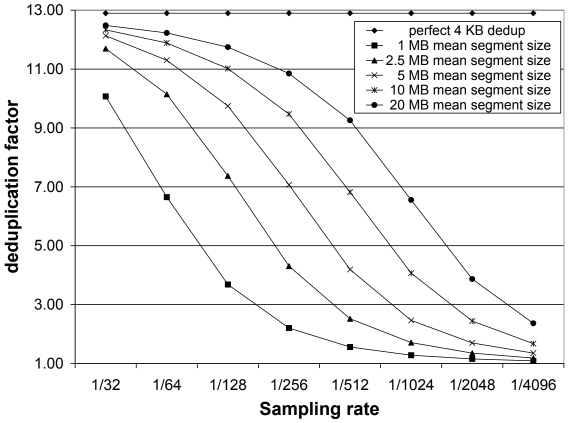

Figure 5: **Deduplication produced using sparse indexing with up to 10 champions (M=10) for various sampling rates and segment sizes** for data set **Workgroup**. For each point, the deduplication factor (deduplicated size/original size) is shown. Shown for comparison is perfect 4 KB deduplication, wherein all duplicate chunks are removed. Variable-size segmentation was used.

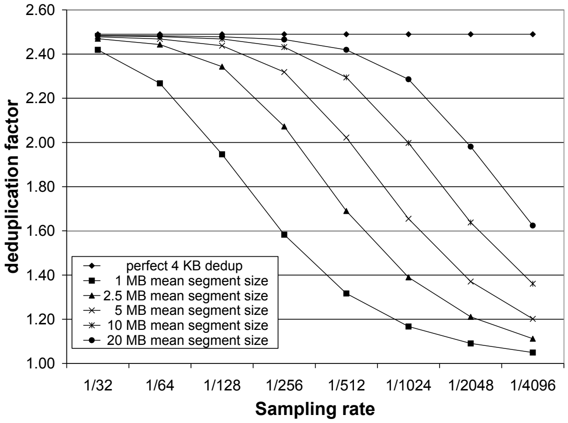

Figure 6: **Deduplication produced using sparse indexing with up to 10 champions (M=10) for various sampling rates and segment sizes** for data set **SMB**. For each point, the deduplication factor (deduplicated size/original size) is shown. Shown for comparison is perfect 4 KB deduplication, wherein all duplicate chunks are removed. Variable-size segmentation was used.

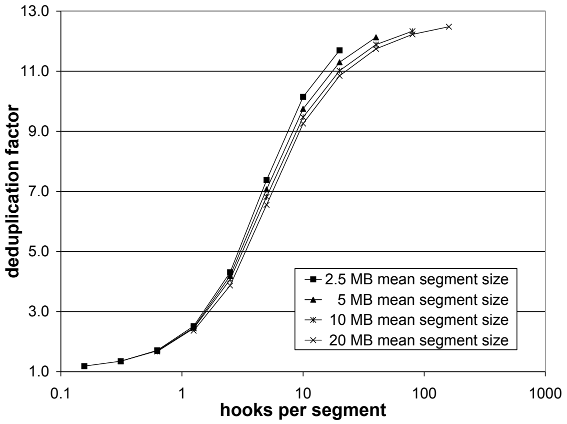

Figure 7: **Deduplication produced using sparse indexing with up to 10 champions (M=10) versus the average number of hooks per segment** for various sampling rates and segment sizes for data set **Workgroup**. For each segment size, sampling rates (from right to left) of 1/32, 1/64, 1/128, 1/256, 1/512, 1/1024, 1/2048, and 1/4096 are shown. Variable-size segmentation was used.

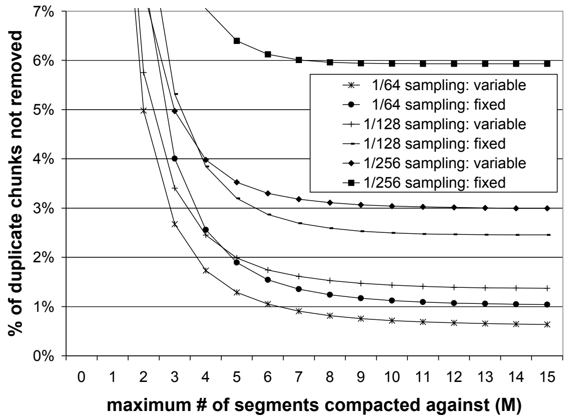

Figure 8: **Fixed versus variable-size segmentation with 10 MB average size segments for selected sampling rates** for data set **Workgroup**.

that must be found in order to achieve high levels of deduplication quality. Because this introduces more opportunities for serious mistakes (e.g., missing such a segment due to poor sampling), sparse indexing does substantially worse with fixed-size segmentation.

4.5 RAM usage and comparison with Zhu *et al.*

Since one of the main objectives of this paper is to argue that our approach significantly reduces RAM usage for comparable deduplication and throughput to existing approaches, we briefly describe here the approach used by Zhu *et al.* [28], which we call the Bloom Filter with Paged Full Index (BFPFI) approach.

BFPFI uses a full disk-based index of every chunk hash. To avoid having to access the disk for every hash lookup, it employs a Bloom Filter and a cache of chunk container indexes. The Bloom Filter uses one byte of RAM per hash and contains the hash of every chunk in the store. If the Bloom filter does not indicate that an incoming chunk is already in the store, then there is no need to consult the chunk index. Otherwise, the cache is searched and only if it fails to contain the given chunk's hash, is the on-disk full chunk index consulted. Each time the on-disk index must be consulted, the index of the chunk container that contains the given chunk (if any) is paged into memory.

The hit rate of the BFPFI cache (and hence the overall throughput) depends on the degree of chunk locality of the input data: because chunk containers contain chunks that occurred together before, high chunk locality implies a high hit rate. The only parameter that impacts the deduplication factor in BFPFI is the average chunk size, since it finds all the duplicate chunks. Smaller chunk sizes increase the deduplication factor at the cost of requiring more RAM for the Bloom filter.

Both approaches degrade under conditions of poor chunk locality: with BFPFI, throughput degrades, whereas with sparse indexing, deduplication quality degrades. Unlike with BFPFI, with sparse indexing it is possible to guarantee a minimum throughput by imposing a maximum number of champions, which can be important given today's restricted backup window times. It is, of course, impossible to guarantee a minimum deduplication factor because the maximum deduplication possible is limited by characteristics of the input data that are beyond the control of any store.

The amount of RAM required by one of our sparse indexes or the Bloom filter of the BFPFI approach is linearly proportional to the maximum possible number of unique chunks in that store. Accordingly, we plot RAM usage as the ratio of RAM required per amount of physical disk storage. Figure 9 shows the estimated RAM usage of sparse indexes with 5, 10, and 20 MB variable-sized segments as well as the Bloom filter used by BFPFI. We assume here a local compression factor of 2, which allows 100 TB of disk to store twice as many chunks as would otherwise be possible. Because it is easy to achieve good RAM usage if deduplication quality can be neglected, we also show the deduplication factor for data set Workgroup for each case.

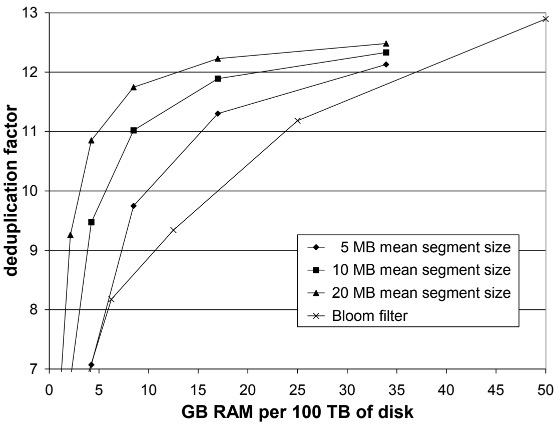

Figure 9: **RAM space required per 100 TB of disk for sparse indexing** with up to 10 champions (M=10) **and for a Bloom filter**. For each point, the deduplication factor for data set Workgroup is shown. Each sparse indexing series shows points for sampling rates of (right to left) 1/32, 1/64, 1/128, 1/256, and 1/512 while the Bloom filter series shows points for chunk sizes of 4, 8, 16, and 32 KB. Variable-size segmentation was used.

You will note that our approach uses substantially less RAM than BFPFI for the same quality of deduplication. For example, for a store with 100 TB of disk, a sparse index with 10 MB segments and 1/64 sampling requires 17 GB whereas we estimate a Bloom filter would require 36 GB for an equivalent level of deduplication. Alternatively, starting with a Bloom filter using 8 KB chunks (the value used by Zhu *et al.* [28]), which requires 25 GB for 100 TB, we estimate we can get the same deduplication quality (using 4 KB chunks) but use only 10 GB (10 MB segments) or 6 GB (20 MB segments) of RAM. For comparison, the Jumbo Store [14], which keeps a full chunk index in RAM, would need 1,500 GB for the second case.

A sparse index has one key for each unique hook encountered; when using a sampling rate of $1/s$, on average $1/s$ of unique chunks will have a hash which qualifies as a hook. Because of the random nature of hashes and the law of large numbers (we are dealing with billions of unique chunks), we can treat this average as a maximum for estimation purposes. To conserve RAM needed by our simulated sparse indexes, we generally limit the number of manifest IDs per hook in our sparse indexing experiments to 1; that is, for each hook, we simulate keeping the ID of only the last manifest containing that hook. This slightly decreases deduplication quality (see Section 4.7), but saves a lot of RAM.

Such a sparse index needs to be big enough to hold u/s keys, each of which has exactly 1 entry, where u is the maximum number of unique chunks possible. The sampling rate is thus the primary factor controlling RAM usage for our experiments. The actual space is ku/s where k is a constant depending on the exact data structure implementation used. For this figure, we have used $k = 21.7$ bytes based on using a chained hash table with a maximum 70% load factor, 4-byte internal pointers, 8-byte manifest IDs, and 4 key check bytes per entry. Using only a few bytes of the key saves substantial RAM but means the index can—very rarely—make mistakes; this may occasionally result in the wrong champion being selected, but is unlikely to substantially alter the overall deduplication quality. We calculate the size of the BFPFI Bloom filter per Zhu *et al.* as 1 byte per unique chunk [28].

Additional RAM is needed for both approaches for per stream buffers. In our case, the per stream space is proportional to the segment size.

4.6 Throughput

Because we do not move around or even simulate moving around chunk data, we cannot estimate overall read or write throughput. However, because we do collect statistics on how many champions are loaded per segment, we can estimate the I/O burden that loading champions places on a system using our approach. Aside from this I/O and writing out new manifests, the only other I/O our system needs to do when ingesting data is that required by any other deduplicating store: reading in the input data and writing out the non-deduplicated chunks.

Similarly, the majority of the computation required by our approach—chunking, hashing, and compression—also must be done by any chunk-based deduplication engine. For an alternative way of getting a handle on the throughput our approach can support, see Section 4.8 where we briefly describe some early throughput measurements of a product embodying our approach.

Figure 10 shows the average number of champion manifests actually loaded per segment for the data set Workgroup with up to 10 champions per segment allowed. The equivalent chart for SMB (not shown) is similar, but scaled down by a factor of 2/3. You will notice that that the average number loaded is substantially less than the maximum allowed, 10. This is primarily because most segments in these data sets can be completely deduplicated using only a few champions (GREEDY does not load substantially more champions than 1/32-sampling). Lower sampling rates result in even fewer champions being loaded because sparser indexes result in fewer candidate champions being identified.

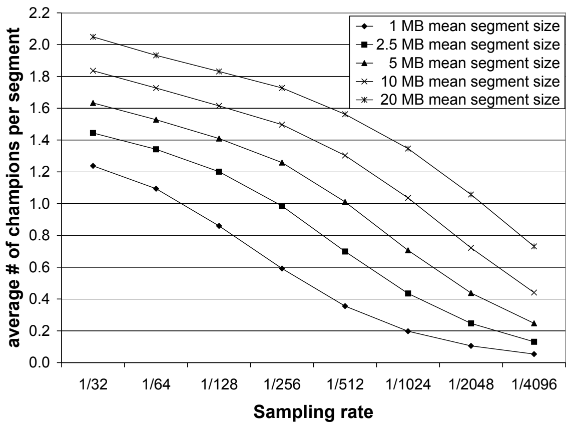

Figure 10: **Average number of champions actually loaded per segment using sparse indexing with up to 10 champions (M=10) for various sampling rates and segment sizes** for data set **Workgroup**. Variable-size segmentation was used.

Loading a champion manifest requires a random seek followed by a quite small amount of sequential reading (manifests are a hundredth of the size of segments and measured in KBs). Accordingly, the I/O burden due to loading champions is best measured in terms of the average number of seeks (equivalently champions loaded) per unit of input data. Figure 11 shows this information for the Workgroup data set. Note that the ordering of segment sizes has reversed: although bigger segments load more champions each, they load their champions so much less frequently that their overall rate of loading champions per megabyte of input data, and hence, their I/O burden is less.

If we conservatively assume that loading a champion manifest takes 20 ms and that we load 0.2 champions per megabyte on average, then a single drive doing nothing else could support a rate of $1/(0.2 \cdot 20 \text{ ms/MB}) = 250$ MB/s. Of course, as we mentioned above, there is other I/O that needs to be done as well. However, in practice deduplication systems are usually deployed with 10 or more drives so the real number before other I/O needs is more like 2.5 GB/s. This is a sufficiently light burden that we expect that some other component of the system will be the throughput bottleneck.

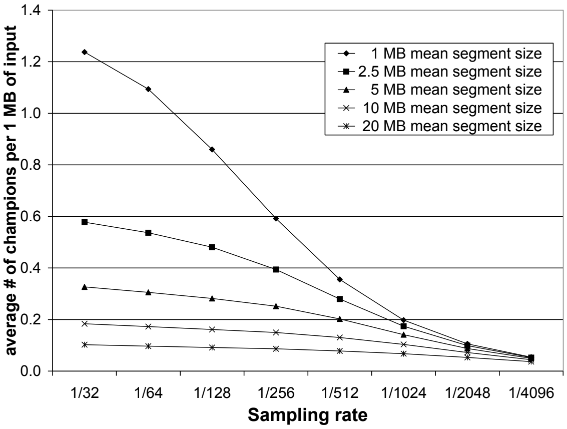

Figure 11: **Average number of champions actually loaded per 1 MB of input data using sparse indexing with up to 10 champions (M=10) for various sampling rates and segment sizes** for data set **Workgroup**. Variable-size segmentation was used.

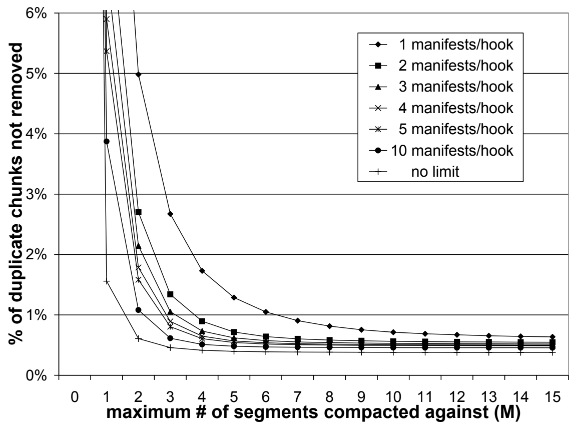

Figure 12: **Deduplication efficiency obtained as the maximum number of manifest IDs kept per hook varies** for 10 MB average size segments and a sampling rate of 1/64 for data set **Workgroup**. Variable-size segmentation was used.

4.7 Optimization

Figure 12 shows how capping the number of manifest IDs kept per hook in the sparse index affects deduplication quality. Keeping only one manifest ID per hook does reduce deduplication somewhat (99.29% versus 99.44% duplicate chunks removed for $M = 10$ here), but greatly reduces the amount of RAM required for the sparse index.

All the experiments we have reported on do not use any manifest caching at all. While the design of our simulator unfortunately makes it hard to implement manifest caching correctly (we compute the ith champion for each segment in parallel), we were able to conservatively approximate a cache just large enough to hold the champions from the previous segment.[3] We find that even with this suboptimal implementation, manifest caching reduces champions *loaded* per segment and slightly improves deduplication quality. For 10 MB variable size segments, $M = 10$, and 1 in 64 sampling for data set Workgroup, for example, our version of manifest caching lowers the average number of champions loaded per segment by 3.9% and improves the deduplication factor by 1.1%.

4.8 Productization

Our approach is being used to build a family of VTL products that use deduplication internally to increase the amount of data they can support. Already on the market are the HP D2D2500 and the D2D4000. Most of the work described in this paper, however, was done before even a prototype of these products was available.

A third-party testing firm, Binary Testing Ltd., was hired by HP to test the D2D4000's deduplication performance [5]. The D2D4000 configuration they tested has 6 750 GB disk drives running RAID 6, 8 GB RAM, 2 AMD Opteron 3 Ghz dual core processors, and a 4 Gb fiber channel link. We report a few representative excerpts from their report here: changing 0.4% of every file of a 4 GB file server data set every day and taking fulls for three months produced a deduplication factor of 69.2; the same change schedule applied to a 4 GB exchange server produced a factor of 24.9. Instead changing only 20% of the items every day but by 5% yielded factors of 25.5 (for the file server) and 40.3 (for the Exchange server). Note that these numbers include all overhead and local compression.

Preliminary throughput testing of a similar system with 12 750 GB disk drives shows write rates of 90 MB/s (1 stream) and 120 MB/s (4 streams) and read rates of 40-50 MB/s (1 stream) and 25-35 MB/s (4 streams). The restore path was still being optimized when these measurement were taken, so those numbers may improve substantially. We believe these product results validate our approach, and demonstrate that we have not overlooked any crucial points.

5 Related Work

Chunking has been used to weed out near duplicates in repositories [16], conserve network bandwidth [20], and reduce storage space requirements [1, 22, 23, 27]. It has also been used to synchronize large data sets reliably while conserving network bandwidth [14, 17].

Archival and backup storage systems detect duplicate data at granularities that range from an entire file, as in EMC's Centera [12], down to individual fixed-size disk blocks, as in Venti [22], and variable-size data chunks, as in the Low-Bandwidth Network File System [20]. Variable-sized chunking has also been used in the commercial sector, for example, by Data Domain and Riverbed Technology. Deep Store [27] is a large-scale archival storage system that uses both delta compression [2, 11] and chunking to reduce storage space requirements. How much deduplication is obtained depends on the inherent content overlaps in the data, the granularity of chunks, and the chunking method [21]. Deduplication using chunking can be quite effective for data that evolves slowly (mainly) through small changes, additions, and deletions [26].

Chunking is just one of the methods in the literature used to detect similarities or content overlap between documents. Shingling [8] was developed by Broder for near duplicate detection in web pages. Manber [18], Brin *et al.* [7], and Forman *et al.* [16] have also developed techniques for finding similarities between documents in large repositories

Various approaches have been used to reduce disk accesses when querying an index. Database buffer management strategies [10] that aim to efficiently maintain a 'working set' of rows of the index in a buffer cache have been well researched. However, these strategies do not work in the case of chunk-based deduplication because chunk IDs are random hashes for which it is not possible to identify or maintain a working set.

Bloom filters [6] have also been used to minimize index accesses. A Bloom filter, which can give false positives but not false negatives, can be used to determine the existence of a key in an index before actually querying the index. If the Bloom filter does not contain the key, then the index does not need to queried thereby eliminating both an index and possibly a disk access. Bloom filters have been used by large scale distributed storage systems such as Google's BigTable [9] and by Data Domain [28].

Besides using Bloom filters to improve the deduplication throughput, Data Domain exploits chunk locality for index caching as well as for laying out chunks on disk. By using these techniques Data Domain can avoid a large number of disk accesses related to index queries. Venti uses a disk-based hash table divided into buckets where a hash function is used to map chunk hashes to appropriate buckets. To improve the index lookup performance, Venti uses caching, striping, and write buffering. Foundation [23] is an archival storage system that preserves users' data and dependencies by capturing and storing regular snapshots of every users' virtual machine. Chunking is used to deduplicate the snapshots. Foundation also uses a combination of Bloom filters and locality-friendly on-disk layouts to improve the performance of index lookups.

6 Conclusions

D2D backup is increasingly becoming the backup solution of choice, and deduplication is an essential feature of D2D backup. Our experimental evaluation has shown that there exists a lot of locality within backup data at the small number of megabytes scale. Our approach exploits this locality to solve the chunk-lookup disk bottleneck problem. Through content-based segmentation, sampling, and sparse indexing, we divide incoming streams into segments, identify similar existing segments, and deduplicate against them, yielding excellent deduplication and throughput while requiring little RAM.

While our approach allows a few duplicate chunks to be stored, we more than make up for this loss of deduplication by using a smaller chunk size (possible because of the small RAM requirements), which produces greater deduplication. Compared with the BFPFI approach, we use less than half the RAM for an equivalent high level of deduplication. The practicality of our approach has been demonstrated by its being used as the basis of a HP product family.

Acknowledgments

We would like to thank Graham Perry, John Czerkowicz, David Falkinder, and Kevin Collins for their help and support. We would also like to think the anonymous FAST'09 reviewers and Greg Ganger, our shepherd. This work was done while the third author was an intern at HP.

References

[1] ADYA, A., BOLOSKY, W. J., CASTRO, M., CERMAK, G., CHAIKEN, R., DOUCEUR, J. R., HOWELL, J., LORCH, J. R., THEIMER, M., AND WATTENHOFER, R. P. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *Proc. of the 5th Symposium on Operating Systems Design and Implementation (OSDI)* (Boston, MA, December 2002), pp. 1–14.

[2] AJTAI, M., BURNS, R., FAGIN, R., LONG, D. D. E., AND STOCKMEYER, L. Compactly encoding unstructured inputs with differential compression. *Journal of the Association for Computing Machinery 49*, 3 (May 2002), 318–367.

[3] ASARO, T., AND BIGGAR, H. Data De-duplication and Disk-to-Disk Backup Systems: Technical and Business Considerations. *The Enterprise Strategy Group* (July 2007).

[4] BIGGAR, H. Experiencing Data De-Duplication: Improving Efficiency and Reducing Capacity Requirements. *The Enterprise Strategy Group* (Feb. 2007).

[5] BINARY TESTING LTD. HP StorageWorks D2D4000 Backup System: a report and full performance test on Hewlett-Packard's SME data deduplication appliance. Available at http://h18006.www1.hp.com/products/storageworks/d2d_4000/relatedinfo.html, July 2008.

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

[7] BRIN, S., DAVIS, J., AND GARCÍA-MOLINA, H. Copy detection mechanisms for digital documents. In *SIGMOD '95: Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data* (San Jose, California, United States, 1995), ACM Press, pp. 398–409.

[8] BRODER, A. Z. On the resemblance and containment of documents. In *SEQUENCES '97: Proceedings of the Compression and Complexity of Sequences 1997* (Washington, DC, USA, 1997), IEEE Computer Society, pp. 21–29.

[9] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. E. Bigtable: a distributed storage system for structured data. In *OSDI'06: Proc. of the 7th USENIX Symposium on Operating Systems Design and Implementation* (Seattle, WA, 2006), pp. 205–218.

[10] CHOU, H.-T., AND DEWITT, D. J. An evaluation of buffer management strategies for relational database systems. In *Proc. of the 11th Conference on Very Large Databases (VLDB)* (Stockholm, Sweden, 1985), VLDB Endowment, pp. 127–141.

[11] DOUGLIS, F., AND IYENGAR, A. Application-specific delta-encoding via resemblance detection. In *Proceedings of the 2003 USENIX Annual Technical Conference* (San Antonio, Texas, June 2003), pp. 113–126.

[12] EMC CORPORATION. EMC Centera: Content Addressed Storage System, Data Sheet, April 2002.

[13] ESHGHI, K. A framework for analyzing and improving content-based chunking algorithms. Tech. Rep. HPL-2005-30(R.1), Hewlett Packard Laboratories, Palo Alto, 2005.

[14] ESHGHI, K., LILLIBRIDGE, M., WILCOCK, L., BELROSE, G., AND HAWKES, R. Jumbo Store: Providing efficient incremental upload and versioning for a utility rendering service. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST)* (San Jose, CA, February 2007), USENIX Association, pp. 123–138.

[15] Federal Information Processing Standard (FIPS) 180–3: Secure Hash Standard (SHS). Tech. Rep. 180–3, National Institute of Standards and Technology (NIST), Gaithersburg, MD, Oct 2008.

[16] FORMAN, G., ESHGHI, K., AND CHIOCCHETTI, S. Finding similar files in large document repositories. In *Proceeding of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)* (Chicago, IL, USA, August 2005), ACM Press, pp. 394–400.

[17] JAIN, N., DAHLIN, M., AND TEWARI, R. TAPER: Tiered approach for eliminating redundancy in replica synchronization. In *Proceedings of the 4th USENIX Conference on File and Storage Technologies (FAST)* (San Francisco, CA, USA, December 2005), pp. 281–294.

[18] MANBER, U. Finding similar files in a large file system. In *Proceedings of the Winter 1994 USENIX Technical Conference* (San Fransisco, CA, USA, January 1994), pp. 1–10.

[19] Microsoft exchange server 2003 load simulator. Download available at http://www.microsoft.com/downloads, February 2006.

[20] MUTHITACHAROEN, A., CHEN, B., AND MAZIÈRES, D. A low-bandwidth network file system. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)* (Banff, Alberta, Canada, October 2001), ACM Press, pp. 174–187.

[21] POLICRONIADES, C., AND PRATT, I. Alternatives for detecting redundancy in storage systems data. In *Proc. of the General Track, 2004 USENIX Annual Technical Conference* (Boston, MA, USA, June 2004), USENIX Association, pp. 73–86.

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

[23] RHEA, S., COX, R., AND PESTEREV, A. Fast, inexpensive content-addressed storage in Foundation. In *Proceedings of the 2008 USENIX Annual Technical Conference* (Boston, Massachusetts, June 2008), pp. 143–156.

[24] SHAPIRO, L. D. Join processing in database systems with large main memories. *ACM Transactions on Database Systems 11*, 3 (1986), 239–264.

[25] TOLIA, N., KOZUCH, M., SATYANARAYANAN, M., KARP, B., BRESSOUD, T., AND PERRIG, A. Opportunistic use of content addressable storage for distributed file systems. In *Proceedings of the General Track, 2003 USENIX Annual Technical Conference* (San Antonio, Texas, June 2003), USENIX Association, pp. 127–140.

[26] YOU, L. L., AND KARAMANOLIS, C. Evaluation of efficient archival storage techniques. In *Proceedings of the 21st IEEE / 12th NASA Goddard Conference on Mass Storage Systems and Technologies* (College Park, MD, April 2004).

[27] YOU, L. L., POLLACK, K. T., AND LONG, D. D. E. Deep Store: An archival storage system architecture. In *Proc. of the 21st International Conference on Data Engineering (ICDE '05)* (Tokyo, Japan, April 2005), IEEE, pp. 804–815.

[28] ZHU, B., LI, K., AND PATTERSON, H. Avoiding the disk bottleneck in the Data Domain deduplication file system. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST)* (San Jose, CA, USA, February 2008), USENIX Association, pp. 269–282.

[29] ZIV, J., AND LEMPEL, A. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory 23*, 3 (1977), 337–343.

Notes

[1] E.g., 30 fulls with 5% data change/day might when deduplicated occupy the space of $1 + 29 \cdot 5\% = 2.45$ fulls so the extra full worth of space needed by out-of-line amounts to requiring 29% more disk space.

[2] Standard TTTD parameters yield for this chunk size a minimum chunk size of 1,856 and a maximum chunk size of 11,299 using primary divisor 2,179 and secondary divisor 1,099.

[3] We get the effect of a size-M manifest cache by causing our simulator to do the following: each time it chooses the ith champion for a given segment, it immediately also deduplicates the given segment against the ith champion of the immediately preceding segment as well.