WAN Optimized Replication of Backup Datasets Using Stream-Informed Delta Compression

Philip Shilane, Mark Huang, Grant Wallace, and Windsor Hsu
Backup Recovery Systems Division
EMC Corporation

Abstract

Replicating data off-site is critical for disaster recovery reasons, but the current approach of transferring tapes is cumbersome and error-prone. Replicating across a wide area network (WAN) is a promising alternative, but fast network connections are expensive or impractical in many remote locations, so improved compression is needed to make WAN replication truly practical. We present a new technique for replicating backup datasets across a WAN that not only eliminates duplicate regions of files (deduplication) but also compresses *similar* regions of files with delta compression, which is available as a feature of EMC Data Domain systems.

Our main contribution is an architecture that adds stream-informed delta compression to already existing deduplication systems and eliminates the need for new, persistent indexes. Unlike techniques based on knowing a file's version or that use a memory cache, our approach achieves delta compression across all data replicated to a server at any time in the past. From a detailed analysis of datasets and hundreds of customers using our product, we achieve an additional 2X compression from delta compression beyond deduplication and local compression, which enables customers to replicate data that would otherwise fail to complete within their backup window.

1 Introduction

Creating regular backups is a common practice to protect against hardware failures and user error. To protect against site disasters though, replicating backups to a remote repository is necessary. Shipping tapes has been a common practice but has the disadvantages of being cumbersome, open to security breaches, and difficult to verify success. Replicating across the WAN is a promising alternative, but high-speed network connectivity is expensive and has been reserved mainly for Tier 1, primary data, which has not been available for backup replication.

Moreover, WAN bandwidth has not increased with data growth rates. While we tend to think of important data residing in corporate centers or data warehouses, computation has become pervasive and valuable data is increasingly generated in remote locations such as ships, oil platforms, mining sites, or small branch offices. Network connectivity may either be expensive or only available at low bandwidths.

Since network bandwidth across the WAN is often a limiting factor, compressing data before transfer improves effective throughput. More data can be protected within a backup window, or, for the same reasons, data is protected against disasters more quickly. Numerous systems have explored data reduction techniques during network transfer including deduplication [14, 25, 35, 37], which is effective at replacing identical data regions with references. A promising technique to achieve additional compression is delta compression, which compresses relative to similar regions by calculating the differences [17, 19, 36].

For both deduplication and delta compression, the goal is to find previous data that is either a duplicate or similar to data being transferred. We would like the pool of eligible data to include previous versions, maximizing our potential compression gains. A standard approach is to use a full index across the entire dataset, which requires space on disk, disk I/O, and ongoing updates [1, 19]. An alternative is to use a partial index holding data that has recently been transferred, which removes the persistent structures but shrinks the pool of eligible data [35]. Depending on the backup cycle, a week's worth of data or more may have to reside in an index to achieve much compression. We present a novel technique called Stream-Informed Delta Compression that achieves identity and delta compression across petabyte backup datasets with no prior knowledge of file versions while also reducing the index overheads of supporting both compression techniques.

Repeated patterns in backup datasets have been leveraged to design effective caching strategies to minimize disk accesses for deduplication [2, 16, 20, 23, 39, 41]. Their key observation is that for backup workloads, current data streams tend to have patterns that correspond to an earlier stream, which can be leveraged for effective caching. Our investigations show that the same data patterns exist for identifying similar data as well as duplicates, without additional index structures.

Our technique assumes that backup data is stored in a deduplicated format on both the backup server and remote backup repository. As streams of data are written to the backup server, they are divided into content-defined chunks, a secure fingerprint is calculated over each chunk, and only non-duplicate chunks are stored in containers devoted to that particular stream.

We augment this standard technique by calculating a *sketch* of each non-duplicate chunk. Sketches, sometimes referred to as resemblance hashes, are weak hashes of the chunk data with the property that if two chunks have the same sketch they are likely near-duplicates. These can be used during replication to identify similar (non-identical) chunks. Instead of using a full index mapping sketches to chunks, we rely on the deduplication system to load a cache with sketches from a previous stream, which we demonstrate in Section 6 leads to compression close to using a full sketch index. During replication, chunks are deduplicated, and non-duplicate chunks are delta compressed relative to similar chunks that already reside at the remote repository. We then apply GZ [15] compression to the remaining bytes and transfer across the WAN to the repository where delta compressed data is first decoded and then stored.

There are several important properties of Stream-Informed Delta Compression. First, we are able to achieve delta compression against any data previously stored and are not limited to a single identified file or the size constraints of a partial index. Since delta compression relies upon a deduplication system to load a cache, there is a danger of missing potential compression, but our experiments demonstrate the loss is small and is a reasonable trade-off.

Second, our architecture only requires one index of fingerprints, while traditional similarity detection required one or more on-disk indexes for sketches [1, 19] or used a partial index with a decrease in compression. Another important consideration in minimizing the number of indexes is that updating the index during file deletion is a complicated step, and reducing complexity/error cases is important for production systems.

Our delta compression algorithm has been released commercially as a standard feature for WAN replication between Data Domain systems. Customers have the option of turning on delta compression when replicating between their deduplicated backup storage systems to achieve higher compression and correspondingly higher effective throughput. Analyzing statistics from hundreds of customers in the field shows that delta compression adds an additional 2X compression and enables the replication of more data across the WAN than could otherwise be protected.

2 Similarity Index Options

To achieve the highest possible compression during WAN replication, we would like to find similarity matches across the largest possible pool of chunks. While previous projects have delta encoded data for replication, the issue of indexing sketches efficiently has not been explored. In this section, we discuss tradeoffs for three indexing options.

2.1 Full Sketch Index

The conceptually simplest solution is to use a full index mapping from sketch to chunk. Unfortunately, for terabytes or petabytes of storage, the index is too large for memory and must be kept on disk, though several previous projects have used a full index for storing sketches [1, 18, 19, 40]. As an example, for a production deduplicated storage system with 256 TB of capacity, 8 KB average chunk size, and 16 bytes per record, the sketch index would be a half-TB. Sketches are random values so there is little locality in an index system, and every query will cause a disk access.

Also, a common technique is for sketches to actually consist of subunits called *super-features* that are indexed independently [4, 19]. Using multiple super-features increases the probability of finding a similar chunk (see Section 4.1), but it also requires a disk access for each super-feature's on-disk index, followed by a disk access for the base chunk itself. Unless the number of disk spindles increases, lookups will be slowed by disk accesses. Another detail that is often neglected is that each index has to be updated as chunks are written and deleted from the system, which can be complicated in a live system. Moving the index to flash memory decreases lookup time [10] but increases hardware cost.

2.2 Partial Sketch Index

An alternative to a full index is to use a partial index holding recently transmitted sketches, which would probably reside in memory, but could also exist on disk. The advantage of a partial index is that it can be created as data is replicated without the need for persistent data structures, and several projects [33, 35] and products [32] use a cache structure. Sizing and updating a partial index are important considerations. The most common implementations are FIFO or LRU policies [33], which have the advantage of finding similar chunks nearby in the replication stream, but will miss

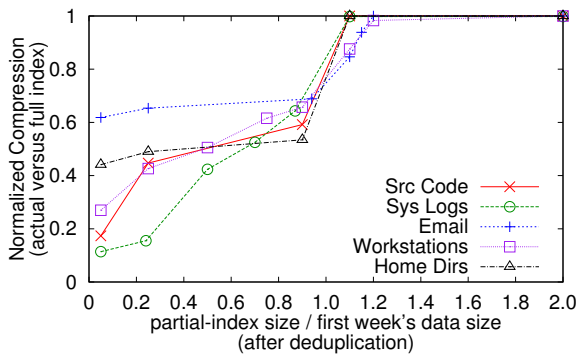

Figure 1: Optimal compression in a backup configuration (e.g. weekly full backup) requires an index to include at least a full backup cycle (1.0 on the x-axis).

distant matches. For backup workloads, repeated data may not appear until next week's full backup takes place, and enterprise organizations typically have hundreds to thousands of primary storage machines to be backed up within that time. Therefore, a partial index would have to be large enough to hold all of an organization's primary data. Riverbed [32] uses an array of disks to index recently transferred data.

Another form of a partial-index is to use version information. As an example, rsync [37] uses file pathnames as the mechanism to find previous versions to perform compression before network transfer.

We analyze this experimentally in Figure 1, which shows how much compression is achieved as index coverage increases (more details are in Section 6). The datasets consist of two weeks worth of backup data, and the combination of deduplication and delta compression across both weeks is presented, normalized relative to compression achievable with a full index (right-most data points). This result shows a sharp increase in compression aligned with the one week boundary when sufficient data are covered by an index for both deduplication and delta compression. Effectively, a partial index would have to be nearly as large as a full index to achieve high compression.

2.3 Stream-Informed Sketch Cache

Numerous papers have explored properties of backup datasets and found that there are repeated patterns related to backup policies. These patterns have been leveraged in deduplication systems to prefetch fingerprints written sequentially by a previous data stream [2, 16, 20, 39, 41]. We discovered that similarity detection has the same stream properties as deduplication, because small edits to a file will probably be a similarity match to the previous backup of the same file, and edits may be surrounded by duplicate regions that can load a cache effectively. This exploration of similarity locality is one of the major contributions of our work.

Following on previous work, we could build a cache and indexing system similar to deduplicating systems (i.e. Bloom filters and indexes), but a disadvantage of this approach is that the number of indexing structures increases with the number of super-features and adds complexity to our system.

Instead, we leverage the same cache-loading technique used by our storage system for deduplication [41]. While loading a previous stream's fingerprints into a cache, we also load sketches from the same stream. This has the significant advantage of removing the need for extra on-disk indexes that must be queried and maintained, but it also has the potential disadvantage of less similarity detection than indexing sketches directly.

To explore these alternatives, we built a full sketch index, a partial index, and a stream-informed cache that piggy-backs on deduplication infrastructure. In Section 6 we explore trade-offs between these three techniques.

3 Delta Replication Architecture

While our research has focused on improving the compression and throughput of replication, it builds upon deduplication features of Data Domain backup storage systems. We first present an overview of our efficient caching technique before augmenting that architecture to support delta compression in replication.

3.1 Stream-Informed Cache for Deduplication

A typical deduplication storage system receives a stream consisting of numerous smaller files concatenated together in a tar-like structure. The file is divided into content-defined chunks [22, 25], and a secure hash value such as SHA-1 is calculated over each chunk to represent it as a fingerprint. The fingerprint is then compared against an index of fingerprints for previously stored chunks. If the fingerprint is new, then the chunk is stored and the index updated, but if the fingerprint already exists, only a reference to the previous chunk is maintained in a file's meta data. Depending on backup patterns and retention period, customers may experience 10X or higher deduplication (logical file size divided by post-deduplication size).

Early deduplication storage systems ran into a fingerprint index bottleneck, because the index was too large to fit in memory, and index lookups limited overall throughput [30]. Several systems addressed this problem by introducing caching techniques. The key insight of the Data Domain system [41] is that when a fingerprint is a duplicate, the following fingerprints will likely match data written consecutively in an earlier stream. We present our basic deduplication architecture along with highlighted modifications in Figure 2. Fingerprints

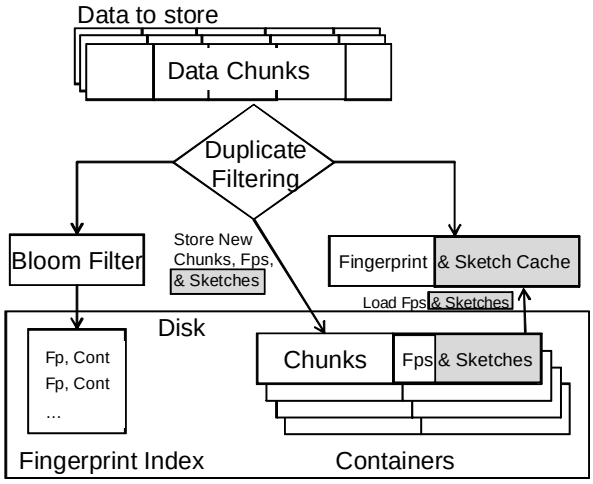

Figure 2: Data Domain deduplication architecture with cache, Bloom filter, fingerprint index, and containers. Highlighted modifications show sketches stored in containers and loaded in a stream-informed cache when fingerprints are loaded.

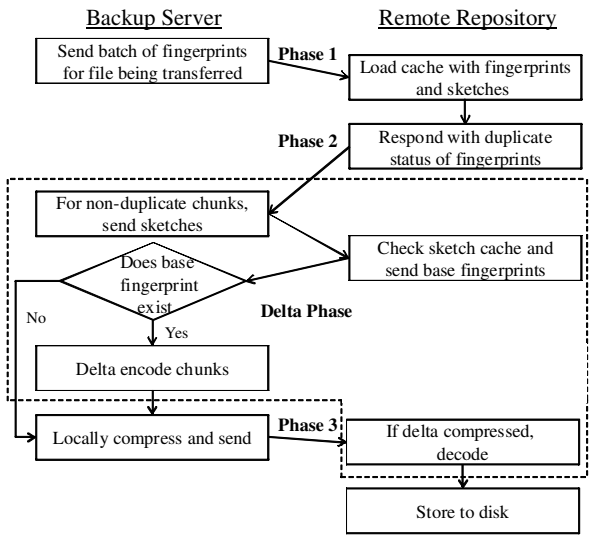

Figure 3: Replication protocol modified to include delta compression.

and chunks are laid out in containers and can be loaded into a fingerprint cache. When a chunk is presented for storage, its fingerprint is compared against the cache, and on a miss, a Bloom filter is checked to determine whether the fingerprint is likely to exist in an on-disk index. If so, the index is checked, and the corresponding container's list of fingerprints is loaded into the cache. When eviction occurs, based on an LRU policy, all fingerprints from a container are evicted as a group. Other techniques for maintaining fingerprint locality have been presented [2, 16, 20, 23, 39], which indexed either deduplicated chunks or the logical stream of file data.

3.2 Replication with Deduplication

For disaster recovery purposes, it is important to replicate backups from a backup server to a remote repository. Replication is a common feature in storage systems [28], and techniques exist to synchronize versions of a repository while minimizing network transfer [18, 37]. In most cases, these approaches result in completely reconstructing files at the destination.

For deduplication storage systems, it is natural to only transfer the unique chunks and the meta data needed to reconstruct logical files. Although not described in detail, products such as Data Domain BOOST [13] already support deduplicated replication by querying the remote repository with fingerprints and only transferring unique chunks, which can be compressed with GZ or other local compressors. Earlier work by Eshghi et al. [14] presented a similar approach that minimized network transfer by querying the remote repository with a hierarchical file consisting of hashes of chunks. These approaches removes duplicates in network-constrained environments.

3.3 Delta Replication

We expand upon standard replication for deduplication systems by introducing delta compression to achieve higher total compression than deduplication and local compression can achieve. We modified the basic architecture in Figure 2, adding sketches to the container meta data section. Sketches are designed so that similar chunks often have identical sketches. As data is written to a deduplicating storage node, non-duplicate chunks are further processed to create a sketch, which is stored in the container along with the fingerprint. During duplicate filtering at the repository, both fingerprints and sketches are loaded into a cache. In later sections, we explore trade-offs of this architecture decision.

3.4 Network Protocol Considerations for Delta Compression

The main issue to address is that both source and destination must agree on and have the same base chunk, the source using it to encode and the destination to decode. Figure 3 shows the protocol we chose for combining deduplication and delta compression. The backup server sends a batch of fingerprints to the remote repository, which loads its cache, performs filtering, and responds indicating which corresponding chunks are already stored. For delta compression, the backup server then sends the sketches of unique chunks to the repository, and the repository checks the cache for matching sketches. The repository responds with the fingerprint corresponding to the similar chunk, called the base fin-

gerprint, or indicates that there is no similarity match. If the backup server has the base fingerprint, it delta compresses a chunk relative to the base before local compression and transfer. At the repository, delta encoded and compressed chunks are uncompressed and decoded in preparation for storage.

We considered sending sketches with fingerprints in Phase 1, but sending sketches after filtering (Phase 2) reduces wasted meta data overhead, compared to sending the sketches for all chunks. Fingerprint filtering occurs on the destination, and its cache is properly set up to find similar chunks. So in practice, it is best if the destination performs similarity lookup.

4 Implementation Details

In this section, we discuss: creating sketches, selecting a similar base chunk, and delta compression relative to a base.

4.1 Similarity Detection with Sketches

In order to delta compress chunks, we must first find a similar chunk already replicated. Numerous previous projects have used sketches to find similar matches, and our technique is most similar to the work of Broder et al. [4, 5, 6].

Intuitively, similarity sketches work by identifying "features" of a chunk that would not likely change even as small variations are introduced in the data. One approach is to use a rolling hash function over all overlapping small regions of data (e.g. 32 byte windows) and choose as the feature the maximal hash value seen. This can be done with multiple different hash functions generating multiple features. Chunks that have one or more features (maximal values) in common are likely to be very similar, but small changes to the data are unlikely to perturb the maximal values [4].

Figure 4 shows an example with data chunks 1 and 2 that are similar to each other and have four sketch features (maximal values) in common. They have the same maximal values because the 32-byte windows that generated the maximal values were not modified by the added regions (in red). If different regions had changed it could affect one or more of the maximal values, so different maximal features would be selected to represent chunk 2. This would cause a feature match to fail. In general, as long as some set of the maximal values are unchanged, a similarity match will be possible.

For our sketches we group multiple features together to form "super-features" (also called super-fingerprints in [19]). The super-feature value is a strong hash of the underlying feature values. If two chunks have an identical super-feature then all the underlying features match. Using super-features helps reduce false positives and requires chunks to be more similar for a match to be found.

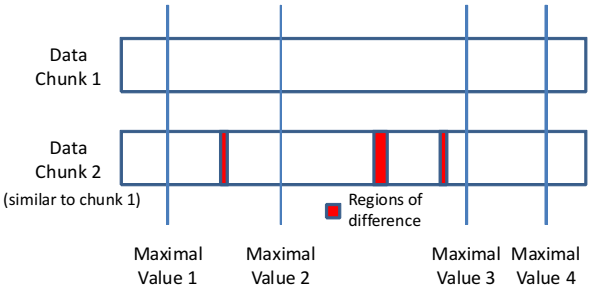

Figure 4: Similar chunks tend to have the same maximal values, which can be used to create features for a sketch.

To generate multiple, independent features, we first generate a Rabin fingerprint *Rabin_fp* over rolling windows w of chunk C and compare the fingerprint against a mask for sampling purposes. We then permute the Rabin fingerprint to generate multiple values with function π_i with randomly generated coprime multiplier and adder values m and a.

$$fp = Rabin\_fp(w)$$

$$\pi_i(fp) = (m_i * fp + a_i) \; mod \; 2^{32}$$

If the result of $\pi_i(fp)$ is maximal for all w, then we retain the Rabin fingerprint as $feature_i$. After calculating all features, a super-feature sf_j is formed by taking a Rabin fingerprint over k consecutive features. We represent consecutive features as $feature_{b...e}$ for beginning and ending positions b and e, respectively.

$$sf_j = Rabin\_fp(feature_{j*k...j*k+k-1})$$

As an example, to produce three super-features with $k = 4$ features each, we generate twelve features, and calculate super-features over the features 0...3, 4...7, and 8...11.

We performed a large number of experiments varying the number of features per super-feature and number of super-features per sketch. Increasing the number of features per super-feature increases the quality of matches, but also decreases the number of matches found. Increasing the number of super-features increases the number of matches but with increased indexing requirements. We typically found good similarity matches with four features per super-feature and a small number of super-features per sketch. These early experiments were completed with datasets that consisted of multiple weeks of backups and had sizes varying from hundreds of gigabytes to several terabytes. We explore the delta compression benefits of using more than one super-feature in Section 6.4.

To perform a similarity lookup, we use each super-feature as a query to an index representing the corresponding super-features of previously processed chunks.

Chunks that match on more super-features are considered better matches than those that match on fewer super-features, and experiments show a correlation between number of super-feature matches and delta compression. Other properties can be used when selecting among candidates including age, status in a cache, locality on disk, or other criteria.

4.2 Delta Compression

Once a candidate chunk has been selected, it is referred to as the *base* used for delta compression, and the *target* chunk currently being processed will be represented as a 1-level delta of the base. To perform delta encoding, we use a technique based upon Xdelta [21] which is optimized for compressing highly similar data regions.

We initialize the encoding by iterating through the base chunk, calculating a hash value at subsampled positions, and storing the hash and offset in a temporary index. We then begin processing the target chunk by calculating a hash value at rolling window positions. We look up the hash value in the index to find a match against the base chunk. If there is a match, we compare bytes in the base and target chunks forward and backward from the starting position to create the longest match possible, which is encoded as a `copy` instruction. If the bytes fail to match, we issue an `insert` instruction to insert the target's bytes into the output buffer, and we also add this region to the hash index. During the backward scans, we may intersect a region previously encoded. We handle this by determining whether keeping the previous instruction or updating it will lead to greater compression. Since we are performing delta compression at the chunk level, as compared to the file level, we are able to maintain this temporary index and output buffer in memory.

5 Experimental Details

We perform actual replication experiments on working hardware with multi-month datasets whenever practical, but we also use simulators to compare alternative techniques. In this section, we first present the datasets tested, then details of our experimental setup, and finally compression metrics.

5.1 Datasets

In this paper we use backup datasets collected over several months as shown in Table 1, which lists the type of data, total size in TB, months collected, deduplication, delta, GZ, and total compression. Total compression is measured as data bytes divided by replicated bytes (after all types of compression) and is equivalent to the multiplication of deduplication, delta, and GZ. For the compression values, we used results from our default configuration. These datasets were previously studied for deduplication [11, 27] but not delta compression. Note that our deduplication results vary slightly (within 5%) from Dong et al. [11] due to implementation differences.

We also highlight steady-state delta compression after a seeding period has completed. For all of the datasets except `Email`, seeding was one week, and the period after seeding is the remaining months of data. Customers often handle initial seeding by keeping pairs of replicating machines on a LAN (when new hardware is installed) until seeding completes and then move the destination machine to the long-term location. Alternatively, seeding can be handled using backups available at the destination. While there is some delta compression within the seeding period, delta compression increases once a set of base chunks become available, and the period after seeding is indicative of what customers will experience for the lifetime of their storage.

These datasets consist of large "tar" type files representing many user files or objects concatenated together by backup software. Except for `Email` (explained below), these datasets consist of a repeated pattern of a weekly full backup followed by six, smaller incremental backups.

Source Code Repository: Backups from a version control repository containing source code.

Workstations: Backups from 16 desktops used by software engineers.

Email: Backups from a Microsoft Exchange server. Unlike the other datasets, `Email` consists of daily full backups, and the seeding phase consists of a single backup instead of a week's worth of data.

System Logs: Backups from a server's /var directory, mostly consisting of emails stored by a list server.

Home Directories: Backups from software engineers' home directories containing source code, office documents, etc.

5.2 Delta Replication Experiments

Many of our experiments were performed on production hardware replicating between pairs of systems in our lab. We actually used a variety of machines that varied in storage capacity (350 GB - 5 TB), RAM (4 GB - 16 GB), and computational resources (2 - 8 cores). We have controlled internal parameters and confirmed that disparate machines produce consistent results. Unless specifically stated, we ran all experiments with 3 super-features per sketch, 12 MB sketch cache, 8 KB average chunk size, and 4.5 MB containers holding meta data and locally compressed chunks. When applying local compression, we create compression regions of approximately 128 KB of chunks.

5.3 Simulator Experiments

We compare our technique of replication with a fingerprint index and sketch cache against two alternative ar-

Name	Properties		Entire Dataset				Seeding	After Seeding			
	TB	Months	Dedupe	Delta	GZ	Total	GB	Dedupe	Delta	GZ	Total
Source Code	4.6	6	20.25	2.97	3.24	194.86	140	24.91	3.75	3.99	372.72
Workstations	4.9	6	5.62	4.44	1.93	48.16	166	5.70	4.62	1.91	50.30
Email	5.2	7	6.79	1.95	2.95	39.06	16	6.90	1.97	2.96	40.24
System Logs	5.4	4	37.19	2.39	2.38	211.54	254	57.94	3.55	2.86	588.26
Home Dirs	12.9	3	19.20	1.90	1.48	53.99	491	31.66	2.89	1.91	174.76

Table 1: Summary of datasets. Deduplication, delta, and GZ compression factors are shown across the entire dataset as well as for the period after seeding, which was typically one week.

chitectures: 1) full fingerprint and sketch indexes and 2) a partial-index of fingerprints and sketches implementing an LRU eviction policy.

Before building the production system, we actually started with a simplified simulator that maintained a full index of fingerprints and sketches in memory. To decrease memory overheads, we use 12 bytes per fingerprint as compared to larger fingerprints necessary for a product such as a 20 byte SHA-1. In a separate analysis, we found that 12 byte fingerprints only cause a small number of collisions out of the hundreds of millions of chunks processed. To maximize throughput and simplify the code, we try to keep the entire index in RAM. Also, instead of implementing a full replication protocol, we record statistics as the client deduplicates and delta compresses chunks without network transfer. Our simulator did not apply local compression with the same technique as our replication system, so comparisons to the simulator do not include local compression.

Our second simulator explores the issues of data locality and index requirements with an LRU partial-index of fingerprints and sketches. This partial-index is a modification of the previous simulator with the addition of parameters to control the index size. The partial-index only holds meta data, fingerprints and sketches, which each reference chunks stored on disk. The fingerprint and sketches for a chunk maintain the same age in the partial-index, so they are added and evicted as a unit. If a fingerprint is referenced as a duplicate of incoming data or a sketch is selected as the best similarity match for compression, the age is updated.

5.4 Compression Metrics

Our focus is on improving replication across the WAN, specifically for customers with low network connectivity. For that reason, we mostly focus on compression metrics, though we also present throughput results from experiments and hundreds of customer systems.

We tend to use the term *compression* generically to refer to any type of data reduction during replication such as deduplication, delta compression, or local compression with an algorithm such as GZ. Compression is calculated as *original_bytes/post_compression_bytes*. However, we generally use the term *total compression* to mean data reduction achieved by deduplication, delta, and GZ in combination. As an example, if the deduplication factor is $10X$, delta is $2X$, and GZ is $1.5X$ then total compression is $30X$ since these techniques have a multiplicative effect. A compression factor of $1X$ indicates no data reduction. In order to show different datasets on the same graph, we often plot *normalized compression*, which is total compression of a particular experiment divided by the maximum total compression. As explained in Section 6, maximum compression is measured using a full index or the appropriate baseline for each experiment and dataset. Normalized compression is in the range $(0...1]$.

6 Results

In this section, we begin by exploring parameters of our system (cache size, number of super-features, and multi-level delta) and then compare Stream-Informed Delta Compression to alternative techniques such as using a full sketch index or maintaining a partial-index of recently used sketches. We then investigate the interaction of delta and GZ compression.

6.1 Sketch Cache Size

When designing our cache-based delta system, sizing the cache is an important consideration. If datasets have similarity locality that matches up perfectly to deduplication locality, then a cache holding a single container could theoretically achieve all of the possible compression. With a larger cache, similarity matches may be found to chunks loaded in the recent past, with compression growing with cache size. We found that the hit rate is maximized with a cache sized consistently across datasets even though Home Directories is over twice as large as the other datasets.

We evaluated the sketch cache hit rate in Figure 5, by increasing the sketch cache size (x-axis) and measuring the number of similarity matches found in the cache relative to using a full index. The sketch cache size refers to the amount of memory required to hold sketches, which is approximately 12 bytes per super-feature. Therefore a cache of 12 MB corresponds to 1 million super-features

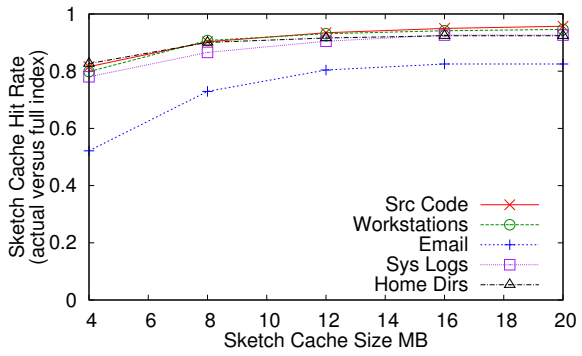

Figure 5: Locality-informed sketch cache hit rate reaches its maximum with a cache of 12-16 MB.

Name	% Post-Dedupe Bytes	Encoding Factor	Delta Factor
Source Code	82	8.91	3.75
Workstations	81	30.05	4.62
Email	55	10.05	1.97
System Logs	77	15.65	3.55
Home Dirs	68	30.11	2.89
Median	77	15.65	3.55

Table 2: Datasets, percent of post-deduplication bytes delta encoded, delta encoding factor, and resulting delta factor for each dataset, which corresponds to Table 1 after seeding.

and $1/3$ million chunks, since we have 3 super-features per sketch by default.

With a cache of 4 MB, the hit rate is between 50% and 90% of the maximum, and the hit rate grows until around 12 or 16 MB, when it is quite close to the final value we show at 20 MB. `Email` showed the worst hit rate, maxing at around 80%, which is still a reasonably high result. `Email` has worse deduplication locality than the other datasets and this impacts delta compression in a data-dependent manner. Regardless of the dataset size (5 TB up to 13 TB) and deduplication (5-37X), all of the datasets reached their maximum hit rates with a similarly sized cache. Our implementation has a minimum cache size related to the large batches of chunks transferred during replication as well as the multiple stages of pipelined replication that either add data to the cache or need to check for matches in the cache.

Although it may be reasonable to use a larger cache in enterprise-sized servers, note that our experiments are for single datasets at a time. A storage server would normally handle numerous simultaneous streams, each needing a portion of the cache, so our single-stream results should be scaled accordingly. Since the locality of delta compression for backup datasets corresponds closely to identity locality, only a small cache is needed, and our memory requirements should scale well with the number of backup streams. Our intuition is that users/applications often make small modifications to files, so duplicate chunks indicate a region of the previous version of a file that is likely to provide delta compression.

6.2 Delta Encoding

Our similarity detection technique is able to find matches for most chunks during replication and achieves high encoding compression on those chunks. The second column of Table 2 shows the percentage of bytes after deduplication that are delta encoded after seeding. 55-82% of bytes undergo delta encoding with a median of 77%. Delta encoding factors vary from 8.91-30.11X with a median of 15.65X. As an example of how the delta factor is calculated for `System Logs`, 77% of bytes after deduplication are delta encoded to $\frac{1}{15.65}$ of their original size, and 23% of bytes are not encoded. Therefore, $\frac{1}{\frac{.77}{15.65}+.23} \approx 3.55$ (rounding in the tables affects accuracy), which is equivalent to dividing post-deduplication bytes by post-delta compression bytes.

While further improvements in encoding compression are likely possible, we are already shrinking delta encoded chunks to a small fraction of their original size. On the other hand, increasing the fraction of chunks that receive delta encoding could lead to larger savings.

6.3 Multi- vs 1-Level Delta

While we have described the delta compression algorithm as representing a chunk as a 1-level delta from a base, because we decode chunks at the remote repository, our delta replication is actually multi-level. Specifically, consider a delta encoded chunk B transferred across the network that is then decoded using base chunk C and stored. At a later time, another delta encoded chunk A is transferred across the network that uses B as a base. Although B exists in a decoded form, it was previously a 1-level delta encoded chunk, so A is effectively a 2-level delta because A referenced B, which referenced C. Our replication system, like many, does not bound the delta level, since chunks are decoded at the destination, and we effectively achieve multi-level delta across the network.

As compared to replicating delta compressed chunks, storing such chunks introduces extra complexity. Although n-level delta is possible for any value of n, decoding an n-level delta entails n reads of the appropriate base chunks, which can be inefficient in a storage system. For this reason, a delta storage system [1] may only support $1-$ or 2-level delta encodings to bound decode times.

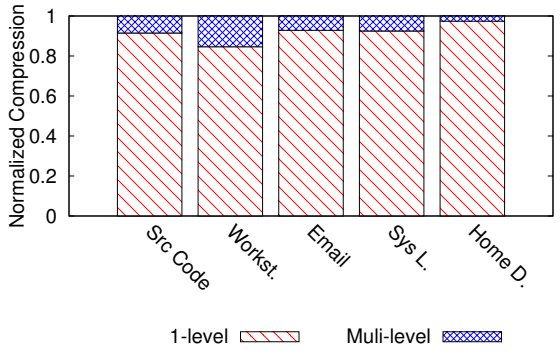

Figure 6: Multi-level delta compression improves 6-30% beyond 1-level delta.

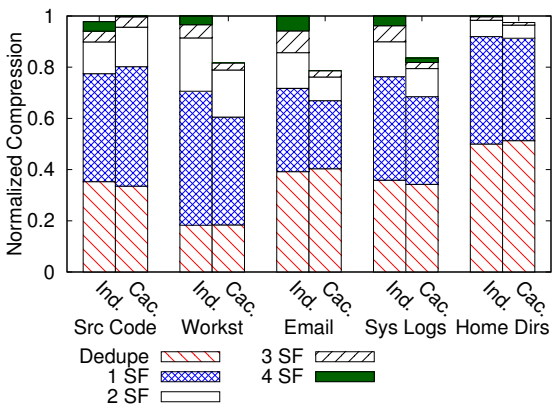

Figure 7: Using a stream-informed sketch cache results in nearly as much compression as using a full index, and using two super-features with a cache achieves more compression than a single super-feature index.

To compare the benefits of multi- and 1−level delta, we studied the compression differences. We modified our replication system so that after a chunk is delta encoded, its sketch is then invalidated. This ensures that delta encoded chunks will never be selected as the base for encoding other chunks, preventing 2-level or higher deltas.

In Figure 6, multi- and 1−level delta are compared, with multi-level delta adding 1.03 - 1.18X additional compression. As an example, `Source Code` increased from 178X to 194X total compression (deduplication, delta, and GZ), which is roughly similar to adding a second super-feature as discussed in Section 6.4. These results also highlight that 1-level delta is a reasonable approximation to multi-level, when multi-level is impractical. Unlike a storage system, we are able to get the compression benefits of multi-level without the slowdowns related to decoding n-level delta chunks.

6.4 Sketch Index vs Stream-Informed Sketch Cache

We next investigate how our stream-informed caching technique compares to the alternative of a full sketch index. We expect that using a full sketch index could find potential matches that a sketch cache will miss because of imperfect locality, but maintaining indexes for billions of stored chunks adds significant complexity. We explore the compression trade-offs by comparing delta replication with a cache against a simulator with complete indexes for each super-feature.

Figure 7 compares compression results for the index and cache options. The lowest region of each vertical bar is the amount of compression achieved by deduplication, and because of differences in implementation between our product and simulator, these numbers vary slightly. The next four sets of colored regions show how much extra compression is achieved by using 1-4 super-features. The cache experiments ran on production hardware, and the cache was fixed at 12 MB. Also, our simulator with index did not apply local compression, so only deduplication and delta compression are analyzed.

In all cases, using a single super-feature adds significant compression beyond deduplication alone, with decreasing benefit as the number of super-features increases. Although using a sketch cache generally has lower delta compression than an index, the results are reasonably close (`Workstations` with 1 super-feature and a cache is within 14% of the index with 1 super-feature). Importantly, we can use more than one super-feature in our cache with little additional overhead compared to multiple on-disk indexes for super-features. Using a cache with two or more super-features achieves greater compression than a single index, which is why we decided to pursue the caching technique.

An interesting anomaly is that `Source Code` achieved higher delta compression with a stream-informed sketch cache than a full index, even though we would expect a limited-size cache to be an approximation to a full index. We found that `Source Code` and `Home Directories` had extremely high numbers of potential similarity matches ($> 10,000$) all with the same number of super-feature matches, which was likely due to repeated headers in source files[1]. Selecting among the candidates leads to differences in delta compression, and the selection made by a stream-informed cache leads to higher compression for `Source Code` than our tie-breaking technique for the index (most recently written).

[1]This caused slowed throughput for `Home Directories`, and those experiments would not have completed without adjusting the sketch index. We modified the sketch index for all `Home Directories` results such that if a sketch has more than 128 similarity matches, the current sketch is not added to the index.

Home Directories had similar compression with either a cache or index.

Another unexpected result is that increasing the number of super-features used with our cache did not always increase total compression. Since we fix the size of our cache at 12 MB, when the number of super-features increases, fewer chunks are represented in the cache. The optimal cache size tends to increase with the number of super-features, but the index results indicate that adding super-features has diminishing benefit.

6.5 Partial-index of Fingerprints and Sketches

As a comparison to previous work, we implemented a partial-index of fingerprints and sketches that updates ages when either a chunk's fingerprint or sketch is referenced and evicts from the partial-index with an LRU policy. While it is somewhat unfair to compare a partial-index to our technique, it is useful for analyzing the scalability of such systems.

To focus on the data patterns of typical backups, we limit this experiment to two full weeks of each dataset, which typically consists of a full backup followed by six incremental backups followed by another full and six incremental backups. For Email, we selected two full backups a week apart, since a full backup was created each day.

Figure 1 (presented in Section 2) shows the amount of compression achieved (deduplication and delta) as the partial-index size increases along the x-axis, which is measured as the fraction of the first week's data kept in a partial-index. When the partial-index is able to hold more than a week's worth of data (1.0 on x-axis), compression jumps dramatically as the second week's data compresses against the first week's data. To highlight this property, the horizontal axis is normalized based on the first week's deduplication rate, since the post-deduplication size affects how many fingerprints and sketches must be maintained.

These results highlight that techniques using a partial-index must hold a full backup cycle's worth of data (e.g. at least one full backup) to achieve significant compression, while our delta compression technique uses a combination of a deduplication index and stream-informed sketch cache to achieve high compression with small memory overheads. For storage systems with large backups or backups from numerous sources, our algorithm would tend to scale memory requirements better, since Figure 5 demonstrates that we only need a fixed-size cache regardless of the dataset size.

6.6 Interaction of Delta and Local Compression

Our replication system includes local compressors such as GZ that can be selected by the administrator. During replication, chunks are first deduplicated and many of the remaining chunks are delta compressed. All remaining data bytes (delta compressed or not) are then compressed with a local compressor. A subtle detail of delta compression is that it reduces redundancies within a chunk that appear in the previous base chunk and within itself, which overlaps with compression that local compressors might find.

Name	No Delta	With Delta		Delta
	GZ	Delta	GZ	Improve.
Source Code	7.20	3.75	3.99	2.08
Workstations	2.83	4.62	1.91	3.12
Email	3.12	1.97	2.96	1.87
System Logs	4.63	3.55	2.86	2.19
Home Dirs	3.12	2.89	1.91	1.77
Median	3.12	3.55	2.86	2.08

Table 3: Delta encoding overlaps with the effectiveness of GZ, but total compression including delta is still a 2X improvement beyond alternative approaches. Results are after initial seeding.

We evaluated the impact of delta compression on GZ and total compression by rerunning our replication experiments with GZ enabled and delta compression either enabled or disabled. Table 3 shows GZ compression achieved both with and without delta after seeding. Results with delta enabled are the same as Table 1. Deduplication factors are the same with or without delta enabled, and are removed from the table for space reasons. GZ and delta overlap by 5-50% (7.20X vs 3.99X for GZ on Source Code), but using delta in combination with GZ still provides improved total compression (2.08X for Source Code). The overlap of local compression and delta compression varies with dataset and type of local compressor selected (GZ, LZ, etc.), but we typically see significant advantages to using both techniques in combination with deduplication.

6.7 WAN Replication Improvement

We performed numerous replication experiments measuring network and effective throughput. Figure 8 shows a representative replication result for the Workstations dataset. Throughput was throttled at T3 speed (44 Mb/s) and measured every 10 minutes. We found effective throughput is 1-2 orders of magnitude faster than network throughput, which corresponds to total compression. Although throughput could be further improved with better pipelining and buffering, this result highlights that compression boosts effective throughput and reduces the time until transfer is complete.

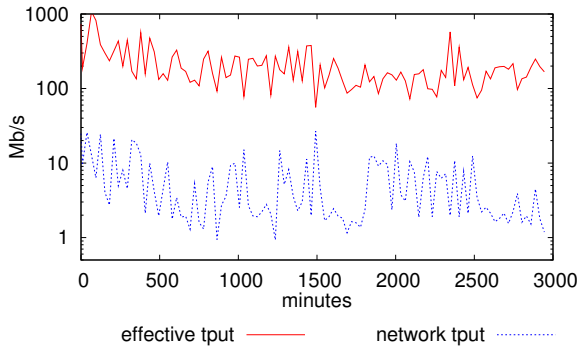

Figure 8: Effective throughput is higher than network throughput due to compression during replication.

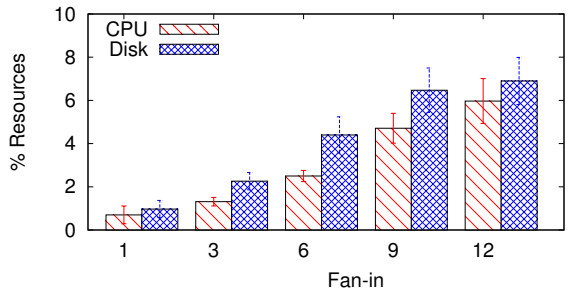

Figure 9: CPU and disk utilization grows fairly linearly on the remote repository as the number of replication streams increases. Error bars indicate a standard deviation.

7 Performance Characteristics

In this section, we discuss overheads of delta compression and limitations of stream-informed delta compression.

7.1 Delta Overheads

First, capacity overheads for storing sketches are relatively small. Each chunk stored in a container (after deduplication) also has a sketch added to the meta data section of the container, which is less than 20 bytes, but our stream-informed approach removes the need for a full on-disk index of sketches.

There are also two performance overheads added to the system: sketching on the write path and reading similar base chunks to perform delta compression. First, incoming data is sketched before being written to disk, which introduces a 20% slowdown in unoptimized tests. The sketching stage happens after deduplication, so after the first full backup, later backups experience less slowdown since a large fraction of the data is duplicate and does not need to be sketched. As CPU cores increase and pipelining is further optimized, this overhead may become negligible.

The second, and more sizable throughput overhead, is during replication when similar chunks are read from disk to serve as the base for delta compression, which limits our throughput by the read speed of our storage system. Our read performance varies with the number of disk spindles and data locality, which we are continuing to investigate. Remote sites also tend to have lower-end hardware with fewer disk spindles than data warehouses. For these reasons, we recommend turning on delta compression for low bandwidth connections (6.3 Mb/s or slower), where delta compression is not the bottleneck and extra delta compression multiplies the effective throughput. Also, it should be noted that read overheads only take place when delta compression occurs, so if no similarity matches are found, read overhead will be minimal.

Effectively, we are trading computation and I/O resources for higher network throughput, and we expect computation and I/O to improve at a faster rate than network speeds increase, especially in remote areas. We expect this tradeoff to become more important in the future as data sizes continue to grow. Improvements to our technique and hardware may also expand the applicability of delta replication to a larger range of customers.

Delta compression increases computational and I/O demands on both the backup server and remote repository. We set up an experiment replicating from twelve small backup servers (2 cores and 3-disk RAID) to a medium-sized remote repository (8 cores and 14-disk RAID) with a T1 connection (1.5 Mb/s). At the backup servers, the CPU and disk I/O overheads were modest (2% and 4% respectively). At the remote repository, CPU and disk overhead scaled linearly as the number of replication streams grew from 1 to 12 as shown in Figure 9. Measurements were made over every 30 second period after the seeding phase, and standard deviation error bars are shown. These results suggest that dozens of backup servers could be aggegated to one medium-sized remote repository. As future work, we would like to increase the scaling tests.

7.2 Stream-Informed Cache Limitations

Since we do not have a full sketch index, loss of cache locality translates to a loss in potential compression. While earlier experiments showed that stream-informed caching is effective, those experiments were on individual datasets. In a realistic environment, multiple datasets have to share a cache, and garbage collection further degrades locality on disk because live chunks from different containers and datasets can be merged into new containers.

We ran an experiment with a midsize storage appliance with a 288 MB cache sized to handle approximately 20 replicating datasets. The experiment consisted of replicating a real dataset to this appliance while varying the number of synthetic datasets also replicated between 0, 24, and 49. This test was performed with three real datasets. The synthetic datasets were generated with an internal tool that had deduplication of 12X and delta compression of 1.7X, which exercises our caching infrastructure in a realistic manner. When the number of datasets was increased to 25 (1 real and 24 synthetic), delta compression decreased 0%, 6% and 12% among the three real datasets relative to a baseline of replicating each real dataset individually. Increasing to 49 synthetic datasets (beyond what is advised for this hardware) caused delta compression to decrease 0%, 12%, and 27% from the baseline for the three real datasets. Our intuition is that the variability in results is due to locality differences among these datasets. In general, these results suggest our caching technique degrades in a gradual manner as the number of replicating datasets increases relative to the cache size.

This experiment investigates how multiple datasets sharing a cache affect delta compression, and we validate these findings with results from the field presented in Section 8, where customers achieved 2X additional delta compression beyond deduplication even though their systems had multiple datasets sharing a storage appliance. While we do not know the upper bound on how much delta compression these customers could have achieved in a single-dataset scenario, these results suggest sizable network savings.

8 Results from Customers

Basic replication has been available with EMC Data Domain systems for many years using the deduplication protocol of Figure 3, and the extra delta compression stage became available in 2009. The version available to customers has a cache scaled to the number of supported replication streams.

We analyzed daily reports from several hundred storage systems used by our customers during the second week of August 2011, including a variety of hardware configurations. Reporting median values, a typical customer transferred 1 TB of data across a 3.1 Mb/s link during the week, though because of our compression techniques, much less data was physically transferred across the network. Median total compression was 32*X* including deduplication, delta, and local compression. Figure 10 shows the distribution of delta compression with 50% of customers achieving over 2*X* additional compression beyond what deduplication alone achieves, and outliers achieving 5*X* additional delta compression. Concurrent work [38] provides further analysis of replication and backup storage in general.

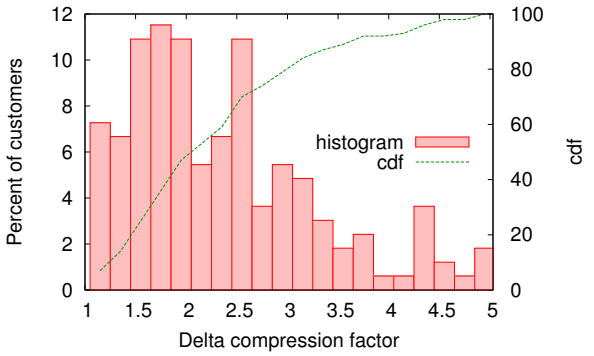

Figure 10: Distribution of delta compression. 50% of customers achieve over 2X additional delta compression.

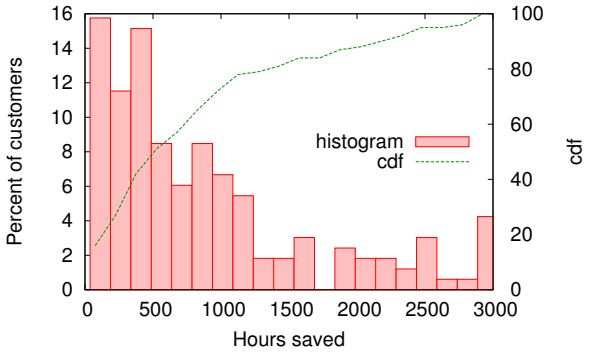

Figure 11: Distribution of hours saved by customers. We estimate that 50% of customers save over 588 hours of replication time per week because of our combination of compression techniques.

Finally, in Figure 11, we show how much time was saved by our customers versus sending data without any compression. Our reports indicate how much data was transferred, an estimate of network throughput (though periodic throttling is difficult to extract), and compression, so we can calculate how long replication would take without compression. The median customer would need 608 hours to fully replicate their data (more hours than are in a week), but with our combined compression, replication reduced to 20 hours (saving 588 hours of network transfer time). For such customers, it would be impossible for them to replicate their data each week without compression, so delta replication significantly increases the amount of data that can be protected.

9 Related Work

Our stream-informed delta replication project builds upon previous work in the areas of optimizing network

transfer, delta compression, similarity detection, deduplication, and caching techniques.

Minimizing network transfer has been an area of ongoing research. One of the earliest projects by Spring et al. [33] removed duplicate regions in packets with a synchronized cache by expanding from duplicate starting points. LBFS [25] divided a client's file into chunks and deduplicated chunks against any previously stored. Jumbo Store [14] used a hierarchical representation of files that allowed them to quickly check whether large subregions of files were unchanged. CZIP [26] applied a similar technique with user level caches to remove duplicate chunks while synchronizing remote repositories.

Most work in file synchronization has assumed that versions are well identified so that compression can be achieved relative to one (or a few) specified file(s). Rsync [37] is a widely used tool for synchronizing folders of files based on compressing against files with the same pathname. An improvement [35] recursively split files to find large duplicate regions using a memory cache.

Beyond finding duplicates during network transfer, delta compression is a well known technique for computing the difference between two files or data objects [17, 36]. Delta compression was applied to web pages [8, 24] and file transfer and storage [7, 9, 21, 34] using a URL and file name, respectively, to identify a previous version.

When versioning information is unavailable, a mechanism is needed to find a previous, similar file or data object to use as the base for delta compression. Broder [4, 5] performed some of the early work in the resemblance field by creating features (such as Rabin fingerprints [31]) to represent data such that similar data tend to have identical features. Features were further grouped into super-features to improve matching efficiency by reducing the number of indexes. Features and super-features were used to select an appropriate base file for deduplication and delta compression [12, 19], removing the earlier requirement for versioning information. TAPER [18] presented an alternative to super-features by representing files with a Bloom filter storing chunk fingerprints and measuring file similarity based on the number of matching bits between Bloom filters and then delta compressing similar files. Delta compression within the storage system has used super-feature techniques to identify similar files or regions of files [1, 40]. Aronovich et al. [1] used 16 MB chunks to decrease sketch indexing requirements and had hundreds of disk spindles for performance.

Storage systems have eliminated duplicate regions based on querying an index of fingerprints [3, 22, 29, 30]. Noting that the fingerprint index becomes much larger than will fit in memory and that disk accesses can become the bottleneck, Zhu et al. [41] presented a technique to take advantage of stream locality to reduce disk accesses by 99%. Several variants of this approach explored alternative indexing strategies to load a fingerprint cache such as moving the index to flash memory [10] and indexing a subset of fingerprints either based on logical or post-deduplication layout on disk [2, 16, 20, 23, 39]. Our similarity detection approach builds upon these caching ideas to load sketches as well as fingerprints into a stream-informed cache.

10 Conclusion and Future Work

In this paper, we present stream-informed delta compression for replication of backup datasets across a WAN. Our approach leverages deduplication locality to also find similarity matches used for delta compression. While locality properties of duplicate data have been previously studied, we present the first evidence that similar data has the same locality. We show that using a compact stream-informed cache to load sketches achieves almost as much delta compression as using a full index without extra data structures. Our technique has been incorporated into the Data Domain systems, and average customers achieve 2X additional compression beyond deduplication and save hundreds of hours of replication time each week.

In future work, we would like to expand the number of WAN environments that benefit from delta replication by improving the read throughput, which currently gates our system. Also, we would like to further explore delta compression techniques to improve compression and scalability.

Acknowledgments

We thank Fred Douglis, Kai Li, Stephen Manley, Hugo Patterson, Hyong Shim, Benjamin Zhu, Cezary Dubnicki (our shepherd), and our reviewers for their feedback. We would also like to acknowledge the many EMC engineers who continue to improve and support delta replication.

References

[1] L. Aronovich, R. Asher, E. Bachmat, H. Bitner, M. Hirsch, and S. T. Klein. The design of a similarity based deduplication system. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, SYSTOR '09, pages 6:1–6:14, New York, NY, USA, 2009.

[2] D. Bhagwat, K. Eshghi, D. D. Long, and M. Lillibridge. Extreme binning: scalable, parallel deduplication for chunk-based file backup. In *Proceedings of the 17th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, Sept. 2009.

[3] D. R. Bobbarjung, S. Jagannathan, and C. Dubnicki. Improving duplicate elimination in storage systems. *Trans. Storage*, 2:424–448, November 2006.

[4] A. Broder. On the resemblance and containment of documents. In *Proceedings of the Compression and Complexity of Sequences*, page 21, 1997.

[5] A. Broder. Identifying and filtering near-duplicate documents. In *Proceedings of the 11th Annual Symposium on Combinatorial Pattern Matching*, pages 1–10, 2000.

[6] A. Z. Broder, M. Charikar, A. M. Frieze, and M. Mitzenmacher. Min-wise independent permutations (extended abstract). In *Proceedings of the 30th annual ACM symposium on Theory of computing*, pages 327–336, New York, NY, USA, 1998.

[7] R. C. Burns and D. D. E. Long. Efficient distributed backup with delta compression. In *Proceedings of the 5th workshop on I/O in parallel and distributed systems*, pages 27–36, New York, NY, USA, 1997.

[8] M. C. Chan and T. Y. C. Woo. Cache-based compaction: a new technique for optimizing web transfer. In *INFOCOM'99 conference*, March 1999.

[9] Y. Chen, Z. Qu, Z. Zhang, and B.-L. Yeo. Data redundancy and compression methods for a disk-based network backup system. *International Conference on Information Technology: Coding and Computing*, 1:778, 2004.

[10] B. Debnath, S. Sengupta, and J. Li. Chunkstash: speeding up inline storage deduplication using flash memory. In *Proceedings of the USENIX Annual Technical Conference*, Berkeley, CA, USA, 2010.

[11] W. Dong, F. Douglis, K. Li, H. Patterson, S. Reddy, and P. Shilane. Tradeoffs in scalable data routing for deduplication clusters. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies*, 2011.

[12] F. Douglis and A. Iyengar. Application-specific delta-encoding via resemblance detection. In *Proceedings of the USENIX Annual Technical Conference*, pages 113–126, 2003.

[13] EMC Corporation. Data Domain Boost Software, 2010. `http://www.datadomain.com/products/dd-boost.html`.

[14] K. Eshghi, M. Lillibridge, L. Wilcock, G. Belrose, and R. Hawkes. 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*, 2007.

[15] J. L. Gailly and M. Adler. The GZIP compressor. `http://www.gzip.org`.

[16] F. Guo and P. Efstathopoulos. Building a high-performance deduplication system. In *Proceedings of the USENIX Annual Technical Conference*, 2011.

[17] J. J. Hunt, K.-P. Vo, and W. F. Tichy. Delta algorithms: an empirical analysis. *ACM Trans. Softw. Eng. Methodol.*, 7:192–214, April 1998.

[18] N. Jain, M. Dahlin, and R. Tewari. Taper: tiered approach for eliminating redundancy in replica synchronization. In *Proceedings of the 4th USENIX Conference on File and Storage Technologies*, 2005.

[19] P. Kulkarni, F. Douglis, J. LaVoie, and J. M. Tracey. Redundancy elimination within large collections of files. In *Proceedings of the USENIX Annual Technical Conference*, pages 59–72, 2004.

[20] M. Lillibridge, K. Eshghi, D. Bhagwat, V. Deolalikar, G. Trezise, and P. Camble. Sparse indexing: large scale, inline deduplication using sampling and locality. In *Proceedings of the 7th USENIX Conference on File and Storage Technologies*, pages 111–123, 2009.

[21] J. MacDonald. File system support for delta compression. Master's thesis, Department of Electrical Engineering and Computer Science, University of California at Berkeley, 2000.

[22] U. Manber. Finding similar files in a large file system. In *Proceedings of the USENIX Winter Technical Conference*, pages 1–10, 1994.

[23] J. Min, D. Yoon, and Y. Won. Efficient deduplication techniques for modern backup operation. *IEEE Transactions on Computers*, 99, 2010.

[24] J. C. Mogul, F. Douglis, A. Feldmann, and B. Krishnamurthy. Potential benefits of delta encoding and data compression for http. In *Proceedings of the ACM SIGCOMM 1997 Conference*, pages 181–194, 1997.

[25] A. Muthitacharoen, B. Chen, and D. Mazières. A low-bandwidth network file system. In *SOSP '01: Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 174–187, 2001.

[26] K. Park, S. Ihm, M. Bowman, and V. S. Pai. Supporting practical content-addressable caching with

CZIP compression. In *Proceedings of the USENIX Annual Technical Conference*, pages 14:1–14:14, Berkeley, CA, USA, 2007.

[27] N. Park and D. Lilja. Characterizing datasets for data deduplication in backup applications. In *IEEE International Symposium on Workload Characterization*, 2010.

[28] H. Patterson, S. Manley, M. Federwisch, D. Hitz, S. Kleiman, and S. Owara. Snapmirror: file system based asynchronous mirroring for disaster recovery. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, Berkeley, CA, USA, 2002.

[29] C. Policroniades and I. Pratt. Alternatives for detecting redundancy in storage systems data. In *Proceedings of the USENIX Annual Technical Conference*, pages 73–86, 2004.

[30] S. Quinlan and S. Dorward. Venti: a new approach to archival storage. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, 2002.

[31] M. O. Rabin. Fingerprinting by random polynomials. Technical report, Center for Research in Computing Technology, 1981.

[32] Riverbed Technology. Riverbed Steelhead Product Family, 2011. `http://www.riverbed.com/us/assets/media/documents/data_sheets/DataSheet%-Riverbed-FamilyProduct.pdf`.

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

[34] T. Suel and N. Memon. Algorithms for delta compression and remote file synchronization. In K. Sayood, editor, *Lossless Compression Handbook*. 2002.

[35] T. Suel, P. Noel, and D. Trendafilov. Improved file synchronization techniques for maintaining large replicated collections over slow networks. In *20th International Conference on Data Engineering*, 2004.

[36] D. Trendafilov, N. Memon, and T. Suel. zdelta: An efficient delta compression tool. Technical report, Department of Computer and Information Science at Polytechnic University, 2002.

[37] A. Tridgell. *Efficient algorithms for sorting and synchronization*. PhD thesis, Australian National University, April 2000.

[38] G. Wallace, F. Douglis, H. Qian, P. Shilane, S. Smaldone, M. Chamness, and W. Hsu. Characteristics of backup workloads in production systems. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, 2012.

[39] W. Xia, H. Jiang, D. Feng, and Y. Hua. Silo: A similarity-locality based near-exact deduplication scheme with low ram overhead and high throughput. In *Proceedings of the USENIX Annual Technical Conference*, 2011.

[40] L. You and C. Karamanolis. Evaluation of efficient archival storage techniques. In *Proceedings of the 21st Symposium on Mass Storage Systems*, Apr. 2004.

[41] B. Zhu, K. Li, and H. Patterson. Avoiding the disk bottleneck in the Data Domain deduplication file system. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, pages 269–282, February 2008.