Primary Data Deduplication – Large Scale Study and System Design

Ahmed El-Shimi Ran Kalach Ankit Kumar Adi Oltean Jin Li Sudipta Sengupta
Microsoft Corporation, Redmond, WA, USA

Abstract

We present a large scale study of primary data deduplication and use the findings to drive the design of a new primary data deduplication system implemented in the Windows Server 2012 operating system. File data was analyzed from 15 globally distributed file servers hosting data for over 2000 users in a large multinational corporation.

The findings are used to arrive at a chunking and compression approach which maximizes deduplication savings while minimizing the generated metadata and producing a uniform chunk size distribution. Scaling of deduplication processing with data size is achieved using a RAM frugal chunk hash index and data partitioning – so that memory, CPU, and disk seek resources remain available to fulfill the primary workload of serving IO.

We present the architecture of a new primary data deduplication system and evaluate the deduplication performance and chunking aspects of the system.

1 Introduction

Rapid growth in data and associated costs has motivated the need to optimize storage and transfer of data. Deduplication has proven a highly effective technology in eliminating redundancy in backup data. Deduplication's next challenge is its application to primary data - data which is created, accessed, and changed by end-users, like user documents, file shares, or collaboration data. Deduplication is challenging in that it requires computationally costly processing and segmentation of data into small multi-kilobyte chunks. The result is large metadata which needs to be indexed for efficient lookups adding memory and throughput constraints.

Primary Data Deduplication Challenges. When applied to primary data, deduplication has additional challenges. First, expectation of high or even duplication ratios no longer hold as data composition and growth is not driven by a regular backup cycle. Second, access to data is driven by a constant primary workload, thus heightening the impact of any degradation in data access performance due to metadata processing or on-disk fragmentation resulting from deduplication. Third, deduplication must limit its use of system resources such that it does not impact the performance or scalability of the primary workload running on the system. This is paramount when applying deduplication on a broad platform where a variety of workloads and services compete for system resources and no assumptions of dedicated hardware can be made.

Our Contributions. We present a large and diverse study of primary data duplication in file-based server data. Our findings are as follows: (i) Sub-file deduplication is significantly more effective than whole-file deduplication, (ii) High deduplication savings previously obtained using small ∼4KB variable length chunking are achievable with 16-20x larger chunks, after chunk compression is included, (iii) Chunk compressibility distribution is skewed with the majority of the benefit of compression coming from a minority of the data chunks, and (iv) Primary datasets are amenable to partitioned deduplication with comparable space savings and the partitions can be easily derived from native file metadata within the dataset. Conversely, cross-server deduplication across multiple datasets surprisingly yields minor additional gains.

Finally, we present and evaluate salient aspects of a new primary data deduplication system implemented in the Windows Server 2012 operating system. We focus on addressing the challenge of scaling deduplication processing resource usage with data size such that memory, CPU, and disk seek resources remain available to fulfill the primary workload of serving IO. The design aspects of our system related to primary data serving (beyond a brief overview) have been left out to meet the paper length requirements. The detailed presentation and evaluation of those aspects is planned as a future paper.

Paper Organization. The rest of this paper is structured as follows. Section 2 provides background on deduplication and related work. Section 3 details data collection, analysis and findings. Section 4 provides an overview of the system architecture and covers deduplication processing aspects of our system involving data chunking, chunk indexing, and data partitioning. Section 5 highlights key performance evaluations and results involving these three aspects. We summarize in Section 6.

2 Background and Related Work

Duplication of data hosted on servers occurs for various reasons. For example, files are copied and potentially modified by one or more users resulting in multiple fully or partially identical files. Different documents may

share embedded content (e.g., an image) or multiple virtual disk files may exist each hosting identical OS and application files. Deduplication works by identifying such redundancy and transparently eliminating it.

Related Work. *Primary* data deduplication has received recent interest in storage research and industry. We review related work in the area of data deduplication, most of which was done in the context of *backup* data deduplication.

Data chunking: Deduplication systems differ in the granularity at which they detect duplicate data. Microsoft Storage Server [5] and EMC's Centera [12] use file level duplication, LBFS [18] uses variable-sized data chunks obtained using Rabin fingerprinting [22], and Venti [21] uses individual fixed size disk blocks. Among content-dependent data chunking methods, Two-Threshold Two-Divisor (TTTD) [13] and bimodal chunking algorithm [14] produce variable-sized chunks. *Winnowing* [23] has been used as a document fingerprinting technique for identifying copying within large sets of documents.

Backup data deduplication: Zhu et al. [26] describe an inline backup data deduplication system. They use two techniques to reduce lookups on the disk-based chunk index. First, a bloom filter [8] is used to track existing chunks in the system so that disk lookups are avoided on non-existing chunks. Second, portions of the disk-based chunk index are prefetched into RAM to exploit sequential predictability of chunk lookups across successive backup streams. Lillibridge et al. [16] use *sparse indexing* to reduce in-memory index size at the cost of sacrificing deduplication quality. HYDRAstor [11] is a distributed backup storage system which is content-addressable and implements global data deduplication, and serves as a back-end for NEC's primary data storage solutions.

Primary data deduplication: DEDE [9] is a decentralized host-driven block-level deduplication system designed for SAN clustered file systems for VMware ESX Server. Sun's ZFS [6] and Linux SDFS [1] provide inline block-level deduplication. NetApp's solution [2] is also at the block level, with both inline and post-processing options. Ocarina [3] and Permabit [4] solutions use variable size data chunking and provide inline and post-processing deduplication options. iDedup [24] is a recently published primary data deduplication system that uses inline deduplication and trades off capacity savings (by as much as 30%) for performance.

3 Data Collection, Analysis, and Findings

We begin with a description of datasets collected and analysis methodology and then move on to key findings.

Workload	Srvrs	Users	Total Data	Locations
Home Folders (HF)	8	1867	2.4TB	US, Dublin, Amsterdam, Japan
Group File Shares (GFS)	3	*	3TB	US, Japan
Sharepoint	1	500	288GB	US
Software Deployment Shares (SDS)	1	†	399GB	US
Virtualization Libraries (VL)	2	†	791GB	US
Total	15		6.8TB	

Table 1: Datasets used for deduplication analysis. *Number of authors (users) assumed in 100s but not quantifiable due to delegated write access. †Number of (authors) users limited to < 10 server administrators.

3.1 Methodology

We selected 15 globally distributed servers in a large multinational corporation. Servers were selected to reflect the following variety of file-based workload data seen in the enterprise:

- Home Folder servers host the file contents of user home folders (Documents, Photos, Music, etc.) of multiple individual users. Each file in this workload is typically created, modified, and accessed by a single user.
- Group File Shares host a variety of shared files used within workgroups of users. Each file in this workload is typically created and modified by a single user but accessed by many users.
- Sharepoint Servers host collaboration office document content within workgroups. Each file in this workload is typically modified and accessed by many users.
- Software Deployment shares host OS and application deployment binaries or packed container files or installer files containing such binaries. Each file in this workload is typically created once by an administrator and accessed by many users.
- Virtualization Libraries are file shares containing virtualization image files used for provisioning of virtual machines to hypervisor hosts. Each file in this workload is typically created and updated by an administrator and accessed by many users.

Table 1 outlines the number of servers, users, location, and total data size for each studied workload. The count

Dataset	Dedup Space Savings		
	File Level	Chunk Level	Gap
VL	0.0%	92.0%	∞
GFS-Japan-1	2.6%	41.1%	15.8x
GFS-Japan-2	13.7%	39.1%	2.9x
HF-Amsterdam	1.9%	15.2%	8x
HF-Dublin	6.7%	16.8%	2.5x
HF-Japan	4.0%	19.6%	4.9x
GFS-US	15.9%	36.7%	2.3x
Sharepoint	3.1%	43.8%	14.1x

Table 2: Improvement in space savings with chunk-level dedup vs. file-level dedup as a fraction of the original dataset size. Effect of chunk compression is *not* included. (The average chunk size is 64KB.)

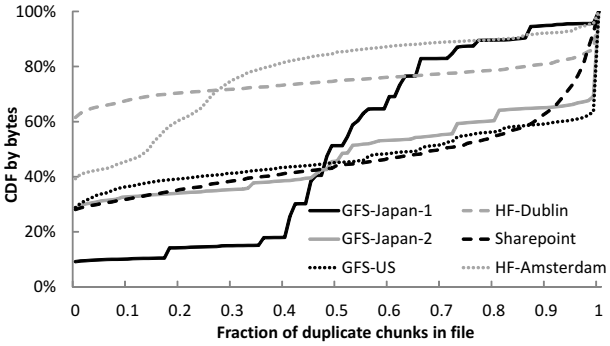

Figure 1: CDF of original (un-deduplicated) files by bytes, as a function of the fraction of chunks in a file that are duplicate, for various datasets.

of users reflects the number of content authors, rather than consumers, on the server.

Files on each server were chunked using a Rabin fingerprint [22] based variable sized chunker at different chunk sizes and each chunk was hashed using a SHA-1 [19] hash function. Each chunk was then compressed using gzip compression [25]. The resulting chunk size (before and after compression), chunk offset, SHA-1 hash, and file information for each chunk were logged and imported into a database for analysis.

To allow for detailed analysis of the effects of different chunking and compression techniques on specific file types, file group specific datasets where extracted from the largest dataset (GFS-US) to represent Audio-Video, Images, Office-2003, Office-2007, PDF and VHD (Virtualization) file types.

3.2 Key Findings

Whole-file vs. sub-file dedup: In Table 2, we compare whole-file and sub-file chunk-level deduplication. Whole-file deduplication has been considered earlier for primary data in the form of *single instance storage* [7]. The study in Meyer et al. [17], involving full desktop

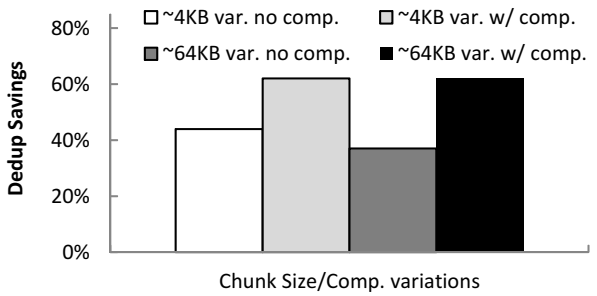

Figure 2: Dedup space savings (%) for chunking with ∼4KB/∼64KB variable chunk sizes, with impact of compression, for GFS-US dataset.

file systems in an enterprise setting, found that file-level deduplication achieves 75% of the savings of chunk-level deduplication. Table 2 shows a significant difference in space savings for sub-file dedup over whole-file dedup, ranging from 2.3x to 15.8x to ∞ (the latter in cases where whole file dedup gives no space savings). The difference in our results from Meyer's [17] is likely due to the inclusion of OS and application binary files which exhibit high whole file duplication in the earlier study [17] while this study focuses on user authored files which are more likely to exhibit sub-file duplication.

Figure 1 provides further insight into the big gap in space savings between chunk-level dedup and file-level dedup for several datasets. On the x-axis, we sort the files in increasing order of fraction of chunks that are duplicate (i.e., also occur in some other file). On the y-axis, we plot the CDF of files by bytes. We calculate a CDF of deduplication savings contributed by files sorted as in this graph, and find that the bulk of the duplicate bytes lie between files having 40% to 95% of duplicate chunks. Hence, sub-file dedup is necessary to identify duplicate data that occurs at a granularity smaller than that of whole files.

We also found reduced space savings for chunking with 64KB fixed size blocks when comparing it with ∼64KB variable size chunking (chunk compression included in both cases). This is because chunking with fixed size blocks does not support the identification of duplicate data when its duplicate occurrence is not aligned along block boundaries. Our findings agree with and confirm prior understanding in the literature [18].

Average Chunk Size: Deduplication systems for backup data, such as in [26], use typical average chunk sizes of 4 or 8KB. The use of smaller chunk sizes can give higher space savings, arising from duplicate detection at finer granularities. On the flip side, this is associated with larger index sizes and increased chunk metadata. Moreover, when the chunk is compressed, usually by a

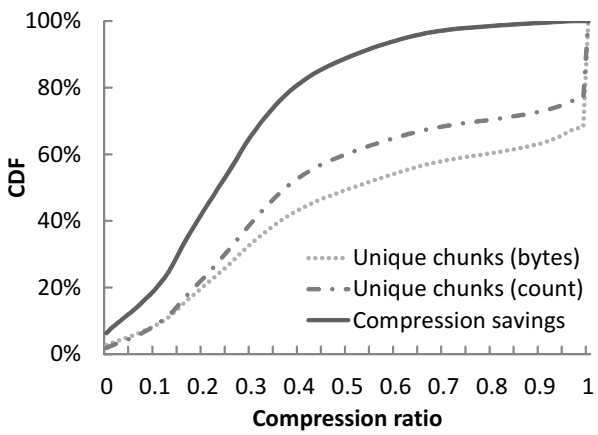

Figure 3: Chunk compressibility analysis for GFS-US dataset.

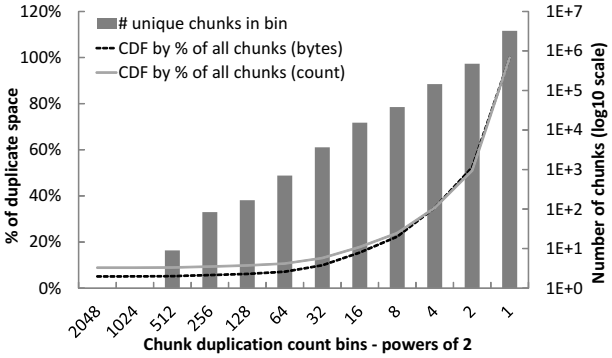

Figure 4: Chunk duplication analysis for GFS-Japan-1 dataset.

LZW-style [25] dictionary based lossless compressor, the smaller chunk size leads to reduced compression performance, as the adaptive dictionary in the lossless compressor has to be reset for each chunk and is not allow to grow to significant size. The adverse impact of these effects is more pronounced for a primary data deduplication system that is also serving live data.

We resolve the conflict by looking for reasonable tradeoffs between larger average chunk sizes and higher deduplication space savings. In Figure 2, we plot the deduplication space savings with and without chunk compression on the GFS-US dataset when the average chunk size is varied from 4KB to 64KB. We see that high deduplication savings achieved with 4KB chunk size are attainable with larger 64KB chunks with chunk compression, as the loss in deduplication opportunities arising from use of larger chunk sizes is canceled out by increased compressibility of the larger chunks.

Chunk compressibility: In Figure 3, we examine the distribution of chunks by compression ratio for the GFS-US dataset. We define the compression ratio c as (size of compressed chunk)/(size of original chunk). Therefore, a chunk with $c = 0.7$ saves 30% of its space when compressed. On the x-axis, we sort and bin unique chunks in the dataset by their compression ratio. On the y-axis, we plot the cumulative distribution of those unique chunks by both count and bytes. We also plot the cumulative distribution of the compression savings (bytes) contributed by those unique chunks across the same compression ratio bins.

We find significant skew in where the compression savings come from – 50% of the unique chunks are responsible for 86% of the compression savings and those chunks have a compression ratio of 0.5 or lower. On the other hand, we find that roughly 31% of the chunks (42% of the bytes) do not compress at all (i.e., fall at compression ratio = 1.0). This result is consistent across all datasets studied (including subsets of datasets created by grouping files of the same type) and it implies that it is feasible for a primary data deduplication system to be selective when compressing chunks. By checking compression ratios and storing compressed only those chunks whose compression ratios meet a certain threshold, the amount of decompression involved during data access can be reduced (assuming each chunk is equally likely to be accessed which is at least true for full file read IO requests). This allows for capturing most of the savings of chunk compression while eliminating the cost of decompression on the majority of chunks.

Chunk duplication analysis: In Figure 4, we examine the distribution of chunk duplication in primary data. On the x-axis, we bin the chunks by number of times they are duplicated in the dataset in power of 2 intervals. The rightmost bin contains unique chunks. Thereafter, moving towards the left, each bin contains chunks duplicated in the interval $[2^i, 2^{i+1})$ for $i \geq 1$. On the y-axis, the bars represent the total number of unique chunks in each bin and the CDFs show the distribution of unique chunks by both count and bytes across the bins. First, we see that about 50% of the chunks are unique, hence a primary data deduplication system should strive to reduce the overhead for serving unique data. Second, the majority of duplicate bytes reside in the middle portion of the distribution (between duplication bins of count 2 and 32), hence it is not sufficient to just deduplicate the top few bins of duplicated chunks. This points to the design decision of *deduplicating all chunks that appear more than once*. This trend was consistent across all datasets.

Data partitioning: We examine the impact of partitioning on each dataset by measuring the deduplication savings (without compression) within partitions and across partitions. We partition each dataset using two methods, namely (i) partitioning by file type (extension), and

Dataset	Dedup Space Savings		
	Global	Clustered by	
		File type	File path
GFS-US	36.7%	35.6%	24.3%
GFS-Japan-1	41.1%	38.9%	32.3%
GFS-Japan-2	39.1%	36.7%	24.8%
HF-Amsterdam	15.2%	14.7%	13.6%
HF-Dublin	16.8%	16.2%	14.6%
HF-Japan	19.6%	19.0%	12.9%

Table 3: Space savings for global dedup vs. dedup within partitioned file clusters, by file type or file path similarity. (See text on why some datasets were excluded.)

Dataset	Total Size	Per-Server Dedup Savings	Cross-Server Dedup Savings	Cross-Server Dedup Benefit
All Home-Folder Srvrs	2438GB	386GB	424GB	1.56%
All File-Share Srvrs	2897GB	1075GB	1136GB	2.11%
All Japan Srvrs	1436GB	502GB	527GB	1.74%
All US Srvrs	3844GB	1292GB	1354GB	1.61%

Table 4: Dedup savings benefit of cross-server deduplication, as fraction of original dataset size.

(ii) partitioning by directory hierarchy where partitions correspond to directory subtrees with total bytes at most 10% of the overall namespace. We excluded two datasets (Virtualization and Sharepoint) because all or most files were of one type or had a flat directory structure, so no meaningful partitioning could be done for them. The remaining datasets produced partitions whose size varied from one-third to less than one-tenth of the dataset.

As seen in Table 3, we find the loss in deduplication savings when partitioning by file type is negligible for all datasets. On the other hand, we find partitioning by directory hierarchy to be less effective in terms of dedup space savings.

Since the original datasets were naturally partitioned by server, we then inspect the effect of merging datasets to find out the impact of deduplication savings across servers. In Table 4, we combine datasets both by workload type and location. We find that the additional savings of cross-server deduplication to be no more than 1-2% above per-server savings.

This implies that it is feasible for a deduplication system to reduce resource consumption by performing partitioned deduplication while maintaining comparable space savings.

4 System Design

We first enumerate key requirements for a primary data deduplication system and discuss some design implications arising out of these and the dataset analysis in Section 3. We then provide an overview of our system. Specific aspects of the system, involving data chunking, chunk indexing, and data partitioning are discussed next in more detail.

4.1 Requirements and Design Implications

Data deduplication solutions have been widely used in backup and archive systems for years. Primary data deduplication systems, however, differ in some key workload constraints, which must be taken into account when designing such systems.

1. **Primary data:** As seen in Section 3, primary data has less duplication than backup data and more than 50% of the chunks could be unique.

2. **Primary workload:** The deduplication solution must be able to deduplicate data as a background workload since it cannot assume dedicated resources (CPU, memory, disk I/O). Furthermore, data access must have low latency - ideally, users and applications would access their data without noticing a performance impact.

3. **Broadly used platform:** The solution cannot assume a specific environment – deployment configurations may range from an entry-level server in a small business up to a multi-server cluster in an enterprise. In all cases, the server may have other softwares installed, including software solutions which change data format or location.

Based on these requirements and the dataset analysis in Section 3, we made some key design decisions for the system which we outline here.

Deduplication Granularity: Our earlier data analysis has shown that for primary datasets, whole file and sub-file fixed-size chunk deduplication were significantly inferior to sub-file variable size chunk deduplication. Furthermore, we learned that chunk compression yields significant additional savings on top of deduplication with greater compression savings on larger chunks, hence closing the deduplication savings gap with smaller chunk sizes. Thus, an average chunk size of about 80KB (and size range 32-128KB) with compression yields savings comparable to 4KB average chunk size with compression. Maximizing savings and minimizing metadata are both highly desirable for primary data

deduplication where tighter system resource constrains exist and there's less inherent duplication in the data. Our system uses variable size chunking with optional compression, and an average chunk size of about 80KB.

Inline vs. Post-processing Deduplication: An important design consideration is when to deduplicate the data. Inline deduplication processes the data synchronously on the write path, before the data is written to disk; hence, it introduces additional write latencies and reduces write throughput. On a primary data server, however, low write latency is usually a requirement as writes are common, with typical 1:3 write/read ratios [15].

Post-processing deduplication processes the data asynchronously, after it has been written to disk in its original format. This approach has the benefit of applying time-based policies to exploit known file access patterns. On file servers, most files are not accessed after 24 hours from arrival [15]. Therefore, a solution which deduplicates only older files may avoid additional latency for most accessed files on the server. Furthermore, post processing deduplication has the flexibility to choose when (e.g., idle time) to deduplicate data.

Our system is based on a post-processing approach, where the agility in deduplication and policies is a better fit for a primary data workload than inline approach.

Resource Usage Scaling with Data Size: One major design challenge a deduplication solution must face is scale and performance - how to scale to terabytes of data attached to a single machine, where (i) CPU memory and disk IOPS are scarce and used by the primary workload, (ii) the deduplication throughput must keep up with the data churn, (iii) dedicated hardware cannot be assumed and (iv) scale out to other machines is optional at best but cannot be assumed. Most, if not all, deduplication systems use a chunk hash index for identifying chunks that are already stored in the system based on their hash. The index size is proportional to the hash size and the number of unique chunks in the system.

Reducing the memory footprint for post-processing deduplication activity is a necessity in primary data servers so that enough memory is available for serving primary workloads. Common index designs in deduplication systems minimize the memory footprint of the chunk hash index by trading some disk seeks with using less memory. The index size is still relative to number of chunks, therefore, beyond a certain data scale, the index will just not fit within the memory threshold assigned to the deduplication system.

In our system, we address that level of scale by using two techniques that can work in concert or separately to reduce the memory footprint for the deduplication process. In the first technique, we use a low RAM footprint chunk hash index that uses about 6 bytes of RAM for every unique chunk stored in the system. This is discussed in more detail in Section 4.4.

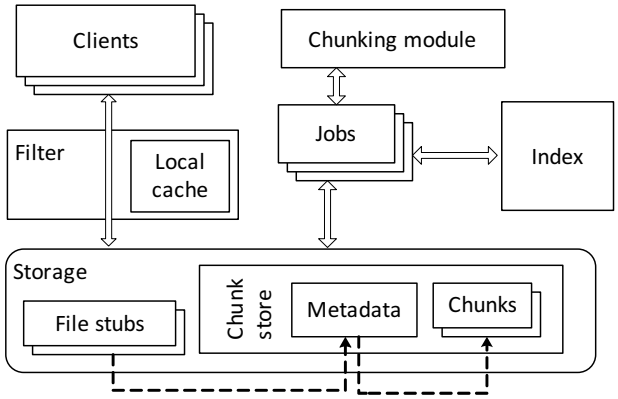

Figure 5: Deduplication engine architecture.

In the second technique, we partition the data into smaller buckets or partitions and deduplicate within each partition. Partitioning may be either permanent, in which case duplicate data may exist across partitions, or temporary, in which case data will be deduplicated across partitions in a subsequent phase called *reconciliation*. This implies that the deduplication process needs to scale only to the maximum partition size. This is discussed in more detail in Section 4.5.

4.2 System Architecture

In Sections 4.3, 4.4, and 4.5, we expand on the post-processing deduplication aspects of our system, including data chunking, chunk indexing, and data partitioning and reconciliation. To provide context for this discussion, we provide here an overview of our overall primary data deduplication system, as illustrated in Figure 5. The design aspects of our system related to primary data serving (beyond a brief overview in this section) have been left out to meet the paper length requirements. The detailed presentation and evaluation of those aspects is planned as a future paper.

Deduplication engine. The deduplication engine consists of a file system filter driver and a set of background jobs (post-processing deduplication, garbage collection, compaction, scrubbing) necessary to maintain continuous deduplication process and to maintain the integrity of the underlying store. The deduplication filter redirects the file system interactions (read, write, etc.) to transparently enable the same file semantics as for a traditional server. The background jobs are designed to run in parallel with the primary server IO workload. The engine actively monitors the primary work load (file serving) and the load of the background jobs. It automatically allocates resources and backs off background jobs to ensure

that these jobs do not interfere with the performance of the primary workload. Under the cover, the deduplication engine maintains the chunks and their metadata in large container files (chunk store) which enable fast sequential IO access on management operations.

Background jobs. The functionalities of background jobs in the deduplication engine are as follows. The post-processing deduplication job progressively scans the underlying volume and identifies candidate deduplication files, which are all files on the server that meet a certain deduplication policy criteria (such as file age). It scans the candidate file, chunks it into variable size chunks (Section 4.3), detects duplicate chunks using a chunk hash index (Section 4.4), and inserts unique chunks in the underlying chunk store, after an optional compression phase. Subsequently, the file itself is *transactionally* replaced with a virtual file stub, which contains a list of references to the associated chunks in the chunk store. Optionally, the stub may also contain extra references to sub-streams representing ranges of non-deduplicated data within the file. This is particularly useful when a deduplicated file is modified, as the modified portion can be represented as sub-streams of non-deduplicated data to support high speed and low latency primary writing operation. The candidate files are incrementally deduplicated – progress state is periodically saved to minimize expensive restarts on a large file in case of job interruptions such as job back-off (triggered by yielding resources to the primary workload) or cluster failover.

The garbage collection job periodically identifies orphaned chunks, i.e., chunks not referenced by any files. After garbage collection, the compaction job is periodically invoked to compact the chunk store container files, to reclaim space from unreferenced chunks, and to minimize internal fragmentation. Finally, the scrubbing job periodically scans the file stubs and the chunk store container files to identify and fix storage-level corruptions.

File reads. File reads are served differently depending on how much file data is deduplicated. For a regular, non-deduplicated file, reads are served from the underlying file system. For fully-deduplicated files, a read request is fulfilled in several steps. In the first step, a file-level read request is translated into a sequence of read requests for each of the underlying chunks through the file-level redirection table. This table essentially contains a list of chunk-IDs and their associated offsets within that particular file. In a second step, the chunk-ID is parsed to extract the index of its container and the virtual offset of the chunk body within the container. In some cases, if the container was previously compacted, the virtual chunk offset is not identical with the physical offset – in that case, a secondary address translation is done through another per-container redirection table. Finally, the read request data is re-assembled from the contents of the corresponding chunks. A similar sequence of operations applies to partially-deduplicated files, with one important difference – the system maintains an extra per-file bitmap to keep track of the non-deduplicated regions within the file, which are kept in the original file contents as a "sparse" stream.

File writes. File writes do not change the content of the chunk store. A file write simply overwrites a range of bytes in the associated sparse stream for that file. After this operation, some chunks may hold data that is old (obsolete) with respect to this file. No explicit deallocation of these chunks is done during the write, as these chunks will be garbage collected later (provided they are not referenced by any other file in the system). The deduplication filter has the metadata and logic to rebuild a file from ranged allocated in the sparse stream and ranges backed up by chunks in the chunk store.

4.3 Data Chunking

The chunking module splits a file into a sequence of chunks in a content dependent manner. A common practice is to use Rabin fingerprinting based sliding window hash [22] on the data stream to identify chunk boundaries, which are declared when the lower order bits of the Rabin hash match a certain pattern P. The length of the pattern P can be adjusted to vary the average chunk size. For example, we use a $|P| = L = 16$ bit pattern, giving an average chunk size of $S_{avg} = 64KB$.

Minimum and maximum chunk sizes. It is desirable for the chunking module in a primary dedup system to generate a chunk size distribution where both very small and very large chunks are undesirable. The very small sized chunks will lead to a larger number of chunks to be indexed, which increases the load of the indexing module. Moreover, for small chunks, the ratio between the chunk metadata size to chunk size is high, leading to performance degradation and smaller dedup savings. The very large sized chunks may exceed the allowed unit cache/memory size, which leads to implementation difficulties in other parts of the dedup systems.

A standard way to avoid very small and very large chunk sizes is to use S_{min} and S_{max} thresholds for minimum and maximum chunk sizes respectively. To enforce the former, a chunk boundary within S_{min} bytes of the last chunk boundary is simply suppressed. The latter is enforced by declaring a chunk boundary at S_{max} bytes when none has been found earlier by the hash matching rule.

One consequence of this size dependent boundary enforcement is the accumulation of peaks around S_{min} and S_{max} in the chunk size distribution and the possible reduction in dedup space savings due to declaration of chunk boundaries that are not content dependent. For example, with $S_{max} = 2 \times S_{avg}$, it can be shown that for ran-

dom data, 14% of chunks will not find a chunk boundary within S_{max}, which is a significant fraction. We design a *regression chunking algorithm* that aims to reduce the peaks in the chunk size distribution around S_{max}, while preserving or even improving the dedup space savings.

Reducing forced boundary declaration at maximum chunk size. The basic idea involves relaxing the strict pattern matching rule – we allow matches on suffixes of the bit pattern P so as to avoid forced boundary declaration at maximum chunk size when possible. Let k denote the maximum number of prefix bits in the pattern P whose matching can be relaxed. Then, we attempt to match the last $L-i$ bits of the pattern P with the lower order bits of the Rabin hash, with *decreasing priority* for $i = 0, 1, \ldots, k$. A boundary is declared at maximum chunk size *only* when no relaxed suffixes match after S_{max} bytes are scanned. For ties within the same suffix match, the later match position in the sequence is used.

In summary, this technique enables the maximum chunk size bound to be satisfied in a content dependent manner *more often* through gradual regression to larger match sets with smaller number of matching bits. With $k = 1$ level of regression, the probability of forcing a chunk boundary at S_{max} for random data reduces to 1.8%. At $k = 4$ levels of regression, the probability is extremely low at 10^{-14}. We use this value of k in our system.

Chunking throughput performance. To maintain fast chunking throughput performance, it is crucial not to break out of the core matching loop often, or increase the complexity of the core loop through additional comparisons. By using regression chunking with nested matching bits, we need to match against only the smallest $(L-k)$-bit suffix of the pattern in the core loop. Only when the smallest suffix match is satisfied do we break out of the core loop and further evaluate the largest suffix match. Regression chunking can be implemented by forward processing the data only once (without ever needing to track back); the only additional state needed is the last match position for each relaxed suffix match.

For the case $k = 2$, regression chunking has some similarities with the TTTD chunking algorithm [13] which uses a second smaller size pattern to declare chunk boundaries when a maximum chunk size is reached. Regression chunking uses additional match patterns to reduce the probability of forcing a chunk boundary at S_{max} to a very small number. Moreover, by making the match patterns progressive suffixes of a base pattern, it maintains fast chunking performance as explained above.

4.4 Chunk Indexing

In a typical deduplication system, the hash index is a design challenge for scale and resource consumption since the index size is relative to the data size. On a primary data deduplication system, conserving memory and disk IOPS resources is crucial. This paper discusses two methods for addressing the index scale. In this section, we present a hash index designed to have a low memory footprint and use reduced read/write disk IOPS so that it can scale to larger datasets and deliver high insert/lookup performance. It incorporates ideas from ChunkStash [10], but the use of flash memory for index storage is not mandatory. In the next section, we present a design for partitioning the index. The two methods can be used independently or in conjunction with each other. The index design is summarized in Figure 6.

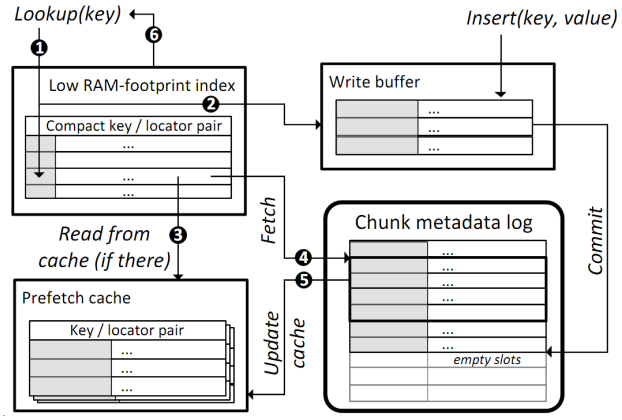

Figure 6: Chunk hash index.

Log-structured organization. The chunk metadata records, comprising of SHA-256 hash and location information, are organized in a log-structured manner on secondary storage. Newly inserted chunk records are held in a write buffer in RAM and *appended* to this log in a *batched* manner so that write IOPS are amortized across multiple index insert operations.

Low RAM footprint index. A specialized in-memory hash table is used to index chunk metadata records on secondary storage, with hash collisions resolved by a variant of cuckoo hashing [20]. The in-memory hash table stores 2-byte compact key signatures instead of full 32-byte SHA-256 chunk hashes so as to strike tradeoffs between RAM usage and false secondary storage reads. The compressed key signature also serves to eliminate disk accesses for lookups on non-existent chunks (with very low false positives). The index footprint is extremely low at about 6 bytes of RAM per chunk.

Prefetch cache. This technique aims to reduce disk IOPS during lookups and has been used for backup data dedup in [26]. When a chunk record R is read from the log, a total of $p = 1000$ sequential records are read in *one single IO*, starting from the location of record R, and inserted into a *prefetch cache* maintained in RAM. While sequential predictability of chunk hash lookups for primary data dedup is expected to be much less than that

for backup data dedup, we did not use smaller values of p since disk seek times dominate for prefetching. The prefetch cache is sized to contain $c = 100,000$ entries, which consumes (an acceptable) 5MB of RAM. It uses an LRU replacement policy.

Disk accesses for index lookups. An index lookup hits the disk *only when* the associated chunk is a duplicate *and* is not present in the prefetch cache (modulo a low false positive rate). The fraction of index lookups hitting disk during deduplication is in the 1% ballpark for all evaluated datasets.

4.5 Data Partitioning and Reconciliation

In Section 4.1, we motivated the need to reduce memory footprint for scale and presented partitioning as one of the techniques used. Partitioning scopes the deduplication task to a smaller namespace, where the namespace size is controlled by the deduplication engine. Therefore, the resource consumption (RAM, IOPS) can be very precise, possibly provided as an external threshold to the system. In this section, we discuss a design for deduplicating within partitions and then reconciling the partitions. The space savings are comparable to deduplicating the entire namespace without partitioning.

Data Partitioning Methods. In Section 3, we demonstrated that partitioning by server or file type is effective - most of the space savings are achieved by deduplicating within the partition, additional savings by deduplicating across partitions are minimal. Other methods may partition by namespace (file path), file time, or file similarity fingerprint. Partitions need to be constructed such that the maximum partition size can be indexed within a defined memory threshold. As the system scales, the number of partitions grow, but the memory consumption remains constant. A solution may consider a hybrid hierarchical partitioning technique, where permanent partitioning is applied at one level and then temporary partitioning is applied at the next level when indexing a permanent partition is too large to fit in memory. The temporary partitions may then be reconciled while the permanent partitions are not. In our design, we use permanent partitioning based on servers, then apply temporary partitioning based on file type or file time bound by a memory threshold. We then reconcile the temporary partitions with an efficient reconciliation algorithm, which is memory bound and utilizes large sequential reads to minimize IOPS and disk seeks.

Two-phase Deduplication. We divide the deduplication task into two phases:

1. *Deduplication within a partition:* The deduplication process is similar to deduplicating an entire dataset except that the hash index loads only the hashes belonging to chunks or files within the partition. We divide the hashes into partitions based on either file type or file age and then, based on memory threshold, we load hashes to the index such that the index size is within the threshold. In this deduplication phase, new chunks ingested into the system are deduplicated only if they repeat a chunk within the partition. However, if a new chunk repeats a chunk in another partition, the duplication will not be detected and the chunk will be stored in the system as a new unique chunk.

2. *Reconciliation of partitions:* Reconciliation is the process of deduplication across partitions to detect and remove duplicate chunks resulting from the previous phase. The design for the reconciliation process has the same goals of minimizing RAM footprint and disk seeks. Given two partitions, the reconciliation process will load the hashes of one partition (say, partition 1) into a hash index, and then scan the chunks belonging to the second partition (say, partition 2), using sequential I/O. We assume that the chunk store is designed to allow sequential scan of chunks and hashes. For each chunk of partition 2, the reconciliation process looks up the hash in the index loaded with all the hashes of partition 1 to detect duplication. One efficient way of persisting the detected duplicates is to utilize a merge log. The merge log is a simple log where each entry consists of the two chunk ids that are a duplicate of each other. The engine appends to the merge log with sequential writes and may batch the writing of merge log entries for reducing IOPS further.

   Once the chunk duplication is detected, reconciliation uses a chunk-merge process whose exact implementation depends on the chunk store. For example, in a chunk store that stores chunks within container files, chunk-merge process may read the merge log and then create new revision of the containers (using sequential read and write), omitting the duplicate chunks. At the end of the second phase, there are no duplicate chunk instances and the many partitions turn into a single reconciled partition.

Reconciliation Strategies. While reconciliation is efficient, it still imposes some I/O overhead on the system. The system may self-tune based on availability of resources and idle time whether to reconcile all partitions or selectively. In the case that reconciliation across all partitions is the selected strategy, the following method is used. The basic idea is to consider some number of unreconciled partitions at a time and grow the set of reconciled partitions by comparing the current group of unreconciled partitions to each of those in the already reconciled set. This is done by indexing in memory the

current group of unreconciled partitions and then scanning sequentially the entire reconciled set of partitions and building merge logs for all. This process repeats itself until all unreconciled partitions join the reconciled set. The number of unreconciled partitions considered in each iteration depends on the amount of available memory – this allows a tradeoff between the speed of the reconciliation process and the amount of memory used.

Our reconciliation strategy above uses the *hash join* method. An alternative method is *sort-merge* join, which is not only more CPU intensive (since sorting is more expensive than building a hash table) but is also disk IO expensive (as it requires reading and writing buckets of sorted partitions to disk).

Selective reconciliation is a strategy in which only some subsets of the partitions are reconciled. Selecting which partitions to reconcile may depend on several criteria, such as file types, signature computed from the data within the partition, or some other criteria. Selective reconciliation is a tradeoff between reducing IOPS and achieving maximal deduplication savings.

Another useful strategy is delayed reconciliation – rather than reconciling immediately after deduplication phase, the deduplication engine may defer it to server's idle time. With this strategy, the tradeoff is with how long it takes to achieve maximal savings. As seen in Section 3, deduplication across (appropriately chosen) partitions yields incremental additional savings, therefore both selective or delayed reconciliation may be the right tradeoff for many systems.

5 Performance Evaluation

In Section 3, we analyzed our datasets around multiple aspects for dedup space savings and used those findings to design our primary data deduplication system. In this section, we evaluate some other aspects of our primary data deduplication system that are related to post-processing deduplication.

Post-processing deduplication throughput. Using the GFS-US dataset, we examined post-processing deduplication throughput, calculated as the amount of original data processed per second. An entry-level HP ProLiant SE326M1 system with one quad-core Intel Xeon 2.27 GHz L5520 and 12 GB of RAM was used, with a 3-way RAID-0 dynamic volume on top of three 1TB SATA 7200 RPM drives.

To perform an apples-to-apples comparison, we ran a post-processing deduplication session multiple times with different indexing options in a controlled environment. Each deduplication session uses a *single thread*. Moreover, we ensured that there were no CPU-intensive tasks running in parallel for increased measurement accuracy. The baseline case uses a *regular index* where the full hash (SHA-256, 32 bytes) and location information (16 bytes) for each unique chunk is stored in RAM. The *optimized index* uses the RAM space efficient design described in Section 4.4. The number of data partitions for this workload was chosen as three by the system according to a implementation heuristic; however, partitioning could be done with as many partitions as needed. The resource usage and throughput numbers are summarized in Table 5. The following important observations can be drawn from Table 5:

1. **RAM frugality:** Our optimized index reduces the RAM usage by about 8x. Data partitioning can reduce RAM usage by another 3x (using 3 partitions as chosen for this workload). *The overall RAM usage reduction with optimized index and data partitioning is as much as 24x.*

2. **Low CPU utilization:** The median *single core* utilization (measured over a single run) is in the 30-40% range for all four cases. Compared to the baseline case, it is slightly higher with optimized index and/or data partitioning because of indexing work and reconciliation respectively. Note that modern file servers use multi-core CPUs (typically, quad core or higher), hence this leaves enough room for serving primary workload.

3. **Low disk usage:** (not shown in Table 5) The median disk queue depth is zero in all cases. At the 75-th percentile, the queue depth increases by 2 or 3 as we move from baseline to optimized index and/or data partitioning. In the optimized index case, the increase is due to extra time spent in disk-based index lookups. With data partitioning, the increase is mainly due to reconciliation (which uses mostly sequential IOs).

4. **Sustained deduplication throughput:** Even as RAM usage goes down significantly with optimized index and data partitioning, *the overall throughput performance remains mostly sustained in the range of 26-30 MB/sec*, with only about a 10% decrease for the lowest RAM usage case. This throughput is sufficient to keep up with data ingestion rate on typical file servers, which is small when compared to total stored data. In a four month dynamic trace study on two file servers hosting 3TB and 19TB of data, Leung et al. [15] reported that 177.7GB and 364.4GB of data was written respectively. This computes to an average ingestion rate of 0.03 MB/sec and 0.04 MB/sec respectively, which is *three orders of magnitude lower* than the obtained single deduplication session throughput.

	Regular Index (Baseline)	Optimized index	Regular index w/ partitions	Optimized index w/ partitions
Throughput (MB/s)	30.6	28.2	27.6	26.5
Partitioning factor	1	1	3	3
Index entry size (bytes)	48	6	48	6
Index memory usage	931MB	116MB	310MB	39MB
Single core utilization	31.2%	35.2%	36.8%	40.8%

Table 5: Deduplication processing metrics for regular and optimized indexes, without and with data partitioning, for GFS-US dataset. Disk queue depth (median) is zero in all cases and is not shown.

The above two observations confirm that we meet our design goal of *sustained post-processing deduplication performance at low overhead so that server memory, CPU, and disk resources remains available for serving primary workload*. We also verified that *deduplication processing is parallelizable across datasets and CPU cores/disks* – when datasets have disk diversity and the CPU has at least as many cores as dataset deduplication sessions, the aggregate deduplication throughput scales as expected, assuming sufficient RAM is available.

It is also interesting to note that the extra lookup time spent in the optimized index configuration (for lookups going to disk) is comparable with the time spent in reconciliation in the regular index with partitioning case. To explore this further, we have compared the contribution of various sub-components in the deduplication session in Table 6. As can be seen, in the case of optimized index without partitioning, the main impact on throughput reduction comes from the index lookup time, when lookups miss in the prefetch cache and hit disk. In the case of data partitioning with regular index, the index lookup time is greatly reduced (at the cost of additional memory) but the main impact is deferred to the partition reconciliation phase. We also observe that the CPU time taken by the data chunking algorithm remains low compared with the rest of the post-processing phase.

Chunk size distribution. We compare the chunk size distribution of the regression chunking algorithm described in Section 4.3 with that of basic chunking on the GFS-US dataset. In Figure 7, we see that regression chunking achieves a more uniform chunk size distribution – it flattens the peak in the distribution around maximum chunk size (128KB) (by relaxing the chunk boundary declaration rule) and spreads out the distribution more uniformly between minimum and maximum chunk sizes. Regression chunking obtains an average chunk size of 80KB on this dataset, which is a 5% decrease over that obtained by basic chunking. This can also be attributed to the effect discussed above.

Deduplication Activity	Optimized index	Regular index w/ partitioning
Index lookup	10.7%	0.4%
Reconciliation	n/a	7.0%
Compression	15.1%	15.3%
SHA hashing	14.3%	14.6%
Chunking	9.7%	9.7%
Storing unique data	11.3%	11.5%
Reading existing data	12.6%	12.8%

Table 6: Percentage time contribution to the overall post-processing session for each component of deduplication activity for GFS-US dataset.

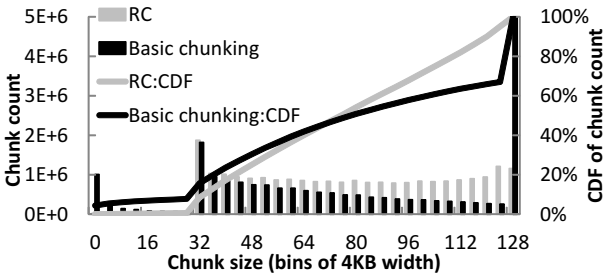

Figure 7: Distribution of chunk size for GFS-US dataset.

In Table 7, we plot the improvement in dedup space savings obtained by regression chunking over basic chunking on this dataset. Although the overall improvement is about 3%, we see significant improvements for some file types contained in that dataset – for example, the dedup savings increases by about 27% for pdf file types. Thus, depending on the mix of file types in the dataset, regression chunking can provide marginal to significant additional dedup space savings. This effect can be attributed to regression chunking declaring more

Dataset	Dedup Space Savings		
	Basic Chunking	Regression Chunking (RC)	RC Benefit
Audio-Video	2.98%	2.98%	0%
PDF	9.96%	12.70%	27.5%
Office-2007	35.82%	36.65%	2.3%
VHD	48.64%	51.39%	5.65%
GFS-US	36.14%	37.2%	2.9%

Table 7: Improvement in dedup space savings with regression chunking over basic chunking for GFS-US dataset. Effect of chunk compression is *not* included.

chunk boundaries in a content dependent manner instead of forcing them at the maximum chunk size.

6 Conclusion

We presented a large scale study of primary data deduplication and used the findings to inform the design of a new primary data deduplication system implemented in the Windows Server 2012 operating system. We found duplication within primary datasets to be far from homogenous, existing in about half of the chunk space and naturally partitioned within that subspace. We found chunk compressibility equally skewed with the majority of compression savings coming from a minority of the chunks.

We demonstrated how deduplication of primary file-based server data can be significantly optimized for both high deduplication savings and minimal resource consumption through the use of a new chunking algorithm, chunk compression, partitioning, and a low RAM footprint chunk index.

We presented the architecture of a primary data deduplication system designed to exploit our findings to achieve high deduplication savings at low computational overhead. In this paper, we focused on aspects of the system which address scaling deduplication processing resource usage with data size such that memory, CPU, and disk resources remain available to fulfill the primary workload of serving IO. Other aspects of our system related to primary data serving (beyond a brief overview), reliability, and resiliency are left for future work.

Acknowledgements. We thank our shepherd Emmett Witchel, the anonymous reviewers, and Mathew Dickson, Cristian Teodorescu, Molly Brown, Christian Konig and Mark Manasse for their feedback. We acknowledge Young Kwon from Microsoft IT for enabling real-life production data to be analyzed. We also thank members of the Windows Server Deduplication project team: Jim Benton, Abhishek Gupta, Ian Cheung, Kashif Hasan, Daniel Hefenbrock, Cosmin Rusu, Iuliu Rus, Huisheng Liu, Nilesh Shah, Giridhar Kasirala Ramachandraiah, Fenghua Yuan, Amit Karandikar, Sriprasad Bhat Kasargod, Sundar Srinivasan, Devashish Delal, Subhayan Sen, Girish Kalra, Harvijay Singh Saini, Sai Prakash Reddy Sandadi, Ram Garlapati, Navtez Singh, Scott Johnson, Suresh Tharamal, Faraz Qadri, and Kirk Olynyk. The support of Rutwick Bhatt, Gene Chellis, Jim Pinkerton, and Thomas Pfenning is much appreciated.

References

[1] Linux SDFS. `www.opendedup.org`.

[2] NetApp Deduplication and Compression. `www.netapp.com/us/products/platform-os/dedupe.html`.

[3] Ocarina Networks. `www.ocarinanetworks.com`.

[4] Permabit Data Optimization. `www.permabit.com`.

[5] Windows Storage Server. `technet.microsoft.com/en-us/library/gg232683(WS.10).aspx`.

[6] ZFS Deduplication. `blogs.oracle.com/bonwick/entry/zfs_dedup`.

[7] BOLOSKY, W. J., CORBIN, S., GOEBEL, D., AND DOUCEUR, J. R. Single instance storage in windows 2000. In *4th USENIX Windows Systems Symposium* (2000).

[8] BRODER, A., AND MITZENMACHER, M. Network Applications of Bloom Filters: A Survey. In *Internet Mathematics* (2002).

[9] CLEMENTS, A., AHMAD, I., VILAYANNUR, M., AND LI, J. Decentralized Deduplication in SAN Cluster File Systems. In *USENIX ATC* (2009).

[10] DEBNATH, B., SENGUPTA, S., AND LI, J. ChunkStash: Speeding Up Inline Storage Deduplication Using Flash Memory. In *USENIX ATC* (2010).

[11] DUBNICKI, C., GRYZ, L., HELDT, L., KACZMARCZYK, M., KILIAN, W., STRZELCZAK, P., SZCZEPKOWSKI, J., UNGUREANU, C., , AND WELNICKI, M. HYDRAstor: a Scalable Secondary Storage. In *FAST* (2009).

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

[13] ESHGHI, K. A framework for analyzing and improving content-based chunking algorithms. *HP Labs Technical Report HPL-2005-30 (R.1)* (2005).

[14] KRUUS, E., UNGUREANU, C., AND DUBNICKI, C. Bimodal Content Defined Chunking for Backup Streams. In *FAST* (2010).

[15] LEUNG, A. W., PASUPATHY, S., GOODSON, G., AND MILLER, E. L. Measurement and analysis of large-scale network file system workloads. In *USENIX ATC* (2008).

[16] LILLIBRIDGE, M., ESHGHI, K., BHAGWAT, D., DEOLALIKAR, V., TREZISE, G., AND CAMBLE, P. Sparse Indexing: Large Scale, Inline Deduplication Using Sampling and Locality. In *FAST* (2009).

[17] MEYER, D. T., AND BOLOSKY, W. J. A study of practical deduplication. In *FAST* (2011).

[18] MUTHITACHAROEN, A., CHEN, B., AND MAZIÈRES, D. A low-bandwidth network file system. In *SOSP* (2001).

[19] NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY, FIPS 180-1. Secure Hash Standard. U.S. Department of Commerce, 1995.

[20] PAGH, R., AND RODLER, F. F. Cuckoo hashing. *Journal of Algorithms 51*, 2 (May 2004), 122–144.

[21] QUINLAN, S., AND DORWARD, S. Venti: A New Approach to Archival Data Storage. In *FAST* (2002).

[22] RABIN, M. O. Fingerprinting by Random Polynomials. *Harvard University Technical Report TR-15-81* (1981).

[23] SCHLEIMER, S., WILKERSON, D. S., AND AIKEN, A. Winnowing: Local algorithms for document fingerprinting. In *ACM SIGMOD* (2003).

[24] SRINIVASAN, K., BISSON, T., GOODSON, G., AND VORUGANTI, K. iDedup: Latency-aware, Inline Data Deduplication for Primary Storage. In *USENIX FAST* (2012).

[25] WELCH, T. A technique for high-performance data compression. *IEEE Computer* (June 1984).

[26] ZHU, B., LI, K., AND PATTERSON, H. Avoiding the Disk Bottleneck in the Data Domain Deduplication File System. In *FAST* (2008).