The Logic of Physical Garbage Collection in Deduplicating Storage

Fred Douglis[*], Abhinav Duggal[*], Philip Shilane[*], Tony Wong[*], Shiqin Yan[*†], Fabiano Botelho[+]
[*]*Dell EMC* [†]*University of Chicago* [+]*Rubrik, Inc.*

Abstract

Most storage systems that write in a log-structured manner need a mechanism for *garbage collection* (GC), reclaiming and consolidating space by identifying unused areas on disk. In a deduplicating storage system, GC is complicated by the possibility of numerous references to the same underlying data. We describe two variants of garbage collection in a commercial deduplicating storage system, a *logical* GC that operates on the files containing deduplicated data and a *physical* GC that performs sequential I/O on the underlying data. The need for the second approach arises from a shift in the underlying workloads, in which exceptionally high duplication ratios or the existence of millions of individual small files result in unacceptably slow GC using the file-level approach. Under such workloads, determining the liveness of chunks becomes a slow phase of logical GC. We find that physical GC decreases the execution time of this phase by up to two orders of magnitude in the case of extreme workloads and improves it by approximately 10–60% in the common case, but only after additional optimizations to compensate for its higher initialization overheads.

1 Introduction

Since the advent of log-structured file systems (LFS) [20], there has been work to optimize the cost of "cleaning" the file system to consolidate live data and create large contiguous areas of free space [15]. Most past efforts in this area have been to optimize I/O costs, as any effort to read and rewrite data reduces the throughput available for new data. With deduplicating storage systems [17, 28] there is an additional complication, that of identifying what data is live in the first place. As new data are written to a system, duplicate *chunks* are replaced with references to previously stored data, so it is essential to track each reference.

Further, as workloads evolve, some systems experience very different usage than traditional deduplicating backup storage systems were intended to support [1]. The Data Domain File System (DDFS) [29] was designed to handle a relatively low number (thousands) of relatively large files (GBs), namely the full and incremental backups that have been the mainstay of computer backups for decades [2, 24]. Data in these backups would be deduplicated, with the remaining content packed into "compression regions" that would be further reduced via standard compression such as Lempel-Ziv (LZ). *Total compression* (TC) is the product of these two factors; DDFS was optimized for TC in the $20-40\times$ range, because of the number of "full backups" a system would typically store. This has been changing dramatically in some environments, with technology trends increasing the deduplication ratio as well as the numbers of files represented in storage systems (§2).

DDFS uses a *mark-and-sweep* [27] algorithm that determines the set of live chunks reachable from the live files and then frees up unreferenced space. There are alternatives such as reference counting [8], but as we discuss in §6, complexity and scalability issues have led to our current approach.

We initially performed garbage collection at the *logical* level, meaning the system analyzed each live file to determine the set of live chunks in the storage system. The shift to using individual file-level backups, rather than tar-like aggregates, meant that the number of files in some systems increased dramatically. This results in high GC overhead during the *mark* phase, especially due to the amount of random I/O required. At the same time, the high deduplication ratios in some systems cause the same live chunks to be repeatedly identified, again greatly increasing GC overhead. The time to complete a single cycle of GC in such systems could be on the order of several days. Since backing up data concurrently with GC results in contention for disk I/O and processing, there is a significant performance implication to such long GC cycles; in addition, a full system might run out of capacity while awaiting space to be reclaimed.

Therefore, we redesigned GC to work at the *physical* level: instead of GC *enumerating* all live files and their referenced chunks, entailing random access, GC performs a series of sequential passes through the *storage containers* containing numerous chunks [17]. Be-

cause the I/O pattern is sequential and because it scales with the physical capacity rather than the deduplication ratio or the number of files, the overhead is relatively constant and proportional to the size of the system [11].

This paper describes the two versions of our garbage collection subsystem, logical and physical GC (respectively LGC and PGC). A detailed description of LGC is useful both for understanding its shortcomings and because this was the version used within the commercial product for many years. We recognize that since LGC was first implemented, there have been several publications describing other GC systems in detail [8, 9, 21, 22], and we view the technical contributions of this paper to be the insights leading to the new and greatly improved PGC subsystem. PGC has undergone an evolution, starting with a change to the order of container access and then being further optimized to lower memory usage and avoid the need for multiple passes over the data. We compare LGC to the earlier implementation of PGC, which has been deployed at customer sites for an extended time, and to the newer "phase-optimized physical GC" (PGC$^+$), incorporating additional optimizations.

In summary, the contributions of this paper include:

- A detailed description of two approaches to garbage collection in a deduplicating storage system.
- An analysis of the changing workloads that have caused the previous approach to be replaced by a new GC algorithm whose enumeration time scales with the physical capacity of the system rather than the logical (pre-deduplication) capacity or the number of files.
- A comparison of GC performance on deployed systems that upgraded from LGC to PGC, demonstrating up to a 20× improvement in enumeration times.
- A detailed comparison of the performance of the various GC algorithms in a controlled environment, demonstrating up to a 99× improvement in enumeration times.

In the remainder of the paper, we provide additional motivation into the problem of scalable GC (§2) and then describe the two GC algorithms, logical and physical GC (§3). §4 describes our evaluation methodology, and §5 analyzes customer systems to compare the techniques in the field and lab testbeds to compare them in controlled environments. §6 discusses related work, and we conclude with final remarks (§7).

2 Background and Motivation

Freeing unreferenced space is a basic storage system operation. While there are multiple ways space management is implemented in traditional storage, when a file is deleted, blocks referenced from its inodes can be freed immediately by marking a free bitmap or updating a free list. For deduplicated storage, determining which chunks are referenced has added complexity as a chunk may have numerous references both within a single file and across many files written at various times. While some deduplication research assumed a FIFO deletion pattern [8], file deletion can generally be in any order.

There are a number of properties to consider when designing and evaluating a garbage collection algorithm:

- All referenced chunks must be preserved so user data can be retrieved.
- Most unreferenced chunks must be removed to free space.
- The system must support client reads and writes during garbage collection.
- System overheads should be minimized.

We specifically state that *most* unreferenced chunks must be removed instead of *all*, since it is often more efficient to focus cleaning on containers that are mostly unreferenced, rather than relocating numerous live chunks to reclaim a small amount of space. This is particularly relevant in log-structured storage systems commonly used by deduplicated storage, which tend to exhibit a bimodal distribution of container liveness [19], where containers tend to be mostly dead or mostly live. We see similar distributions in our workloads.

2.1 Deletion Algorithms

2.1.1 Reference Counts

An intuitive technique to determine when chunks are referenced is to maintain a count for each chunk. When a duplicate chunk is discovered during the write-path, a counter is incremented, and when a file is deleted, counters are decremented; at zero, the chunk can be freed. While this could take place in-line as files are written and deleted [25], it is more common to implement at least partially-offline reference counts [8, 22]. Strzelczak et al. [22] implemented an epoch-based deletion system with counters to support concurrent writes and deletes. We discuss the drawbacks of reference counts, such as snapshot overheads, in related work (§6).

2.1.2 Mark-and-Sweep

Another approach is to run an asynchronous algorithm to determine which chunks are referenced by live files. Mark-and-sweep proceeds in two phases. The first phase walks all of the live files and *marks* them in a data structure. The second phase scans the containers and *sweeps* unreferenced chunks. Guo et al. [9] implemented a Grouped Mark and Sweep to mark referenced containers. While the *sweep* phase is often the slowest phase [5]

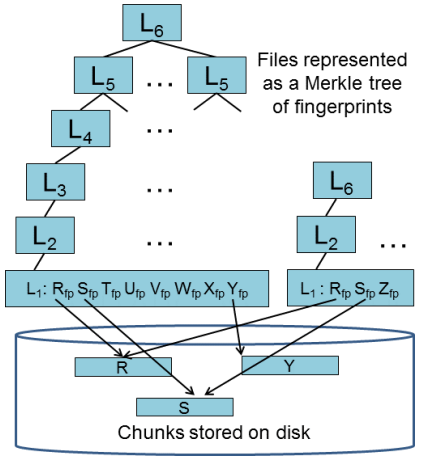


Figure 1: Files in deduplicated storage are often represented with a tree of fingerprints.

because live chunks are read from disk and written to new containers, we have found that the *mark* phase can be the bottleneck in newer systems because enumerating live fingerprints involves random disk I/O. There are also complexities regarding data being deduplicated against chunks in the process of being garbage collected (§3.5).

2.2 Deduplicated File Representation

Figure 1 shows our use of Merkle trees [16] for deduplicated storage. We refer to chunks directly written by users as L_0, meaning the lowest level of the tree. Consecutive L_0 chunks are referenced with an array of fingerprints by an L_1 chunk, which itself is identified by a fingerprint. An array of L_1 fingerprints is referenced by an L_2 chunk, continuing to the root of the tree; the root is always labeled L_6 for convenience, even if the file is small enough not to need intermediate nodes such as the example on the right side of the figure. We refer to the L_1-L_6 chunks as L_p chunks, where p is a parameter that ranges from 1 to 6 and indicates metadata representing the file. Deduplication takes place because a chunk can be referenced multiple times. The file system is a forest of Merkle trees, but these trees are not disjoint, particularly at the lowest level.

Though not shown in the figure, L_p chunks are themselves stored on disk in containers, which include a relatively small (hundreds of KB) metadata section with a list of fingerprints for the chunks within the container. Thus they may be read more quickly than the full container.

As an example, consider a system with 100TB of physical capacity, 20× TC, 8KB L_0 chunks, and 20-byte fingerprints. The logical capacity, *i.e.*, the storage that can be written by applications, is 2PB ($20 * 100$TB). Since each 8KB logically written by a client requires a 20-byte fingerprint stored in an L_1, the L_1 chunks are 5TB. The upper levels of the tree (L_2–L_6) are also stored in containers but are smaller. This example highlights that the *mark* phase cannot be fully performed in memory, as the L_1 references must be read from disk.

Data Domain supports a *fastcopy* [7] system command to efficiently copy an existing file using the same underlying Merkle tree. It creates the new file with a new name, and therefore a new L_6 root of the tree, but that tree then references the identical L_P chunks. As this operation involves only the root of the tree, it is trivially fast and does not increase physical space in use beyond the one chunk containing the L_6.

2.3 Performance Issues with Enumeration

The *mark* phase of mark-and-sweep involves enumerating all of the live fingerprints referenced from live files and marking a data structure for each fingerprint. A number of issues affect performance:

- **Deduplication and compression**. The enumeration cost grows with logical space; some datasets have extremely high TC, making the logical space very large and unreasonably slow to enumerate. Note that deduplication can vary considerably, while LZ compression is usually within a small range (2–4).
- **Number of files** in the system and file size distribution. Every file has metadata overhead to be processed when enumerating a file, and for small files, the overhead may be as large as the enumeration time itself. Also, if the system issues parallel enumeration threads at the file level (processing several files at once), then one extremely large file that cannot be further parallelized might slow overall enumeration, though such skew would be rare.
- **Spatial locality** of the L_p tree. Traversing the L_p tree stops at the L_1 level since all L_0 references are recorded in L_1 chunks. Fragmentation of L_p chunks on disk increases the amount of random I/O needed for enumeration [12, 13]. While sequentially written files tend to have high locality of L_p chunks, creating a new file from incremental changes harms locality as enumeration has to jump among branches of the L_p tree.

2.4 Technology Trends and Enumeration

Two main technology trends increase enumeration time [1]. First, clients are creating many frequent backups by writing the incremental changes such as the changed blocks or files since the previous backup. Since only incremental changes are transferred, backups are faster. *Synthetic* full backups are created by copying a previous full backup and applying the user's changes

since it was created, which is an efficient operation with support within deduplicated storage. A new file is generated by referencing unmodified portions of the previous backup, causing deduplication ratios to increase. This allows a user to view and restore a full backup without the need to restore an earlier full backup and apply intervening incremental changes: the traditional backup strategy of daily incrementals and weekly full backups. For a given period of time, this means backup storage systems may now contain numerous full backups (with the corresponding high deduplication ratio) instead of numerous incremental backups (with a lower deduplication ratio). (Unlike *fastcopy*, synthetic full backups create a full new L_P tree; thus the overhead for storing the metadata is about 1% of the logical size of the data.)

As the speed of backup completion has increased, and applications such as databases desire more frequent point-in-time backups, these backups have become more frequent (multiple times per day). Similarly, applications may take frequent snapshots on deduplicated storage that effectively make a virtual copy of the data [2]. As a result of these technology trends, some systems see deduplication ratios as much as $100-300\times$ or higher (with correspondingly high TC), whereas $10\times$ deduplication was typical for traditional backup environments [24].

A second technology trend is the increase in the file count. This arises from shifts in usage as described above and from the use of deduplication to support backups of individual files rather than aggregates (similar to *tar* files) that represent an entire backup as a single file [24]. It also can arise from users accessing a deduplicating appliance via another interface like NFS, treating it more like primary storage than backup.

As TC and number of small files increase, enumeration time becomes a large component of overall garbage collection. In order to support these new workloads and continue to scale performance, we developed a new *mark* algorithm that replaces logical enumeration in depth-first order with physical enumeration that proceeds in breadth-first order with sequential disk scans.

3 Architecture

This section describes the original LGC algorithm and the changes to enable PGC and then PGC$^+$. One issue common to both forms of GC is how to deal with online updates to the state of the file system, which we discuss after the various algorithms (§3.5).

3.1 Logical GC

Data Domain's original garbage collection technique is logical, using a mark-and-sweep algorithm. We briefly give an overview of the algorithm before describing each

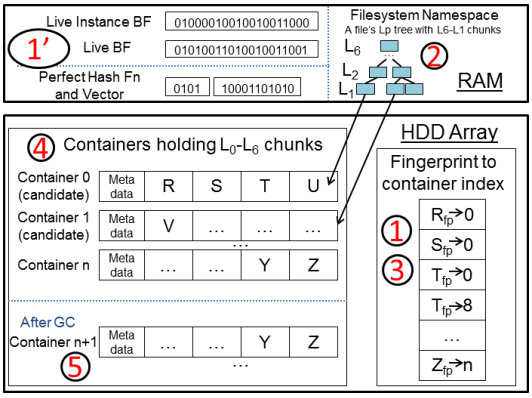


Figure 2: Components of LGC and PGC. The red numbers correspond to GC phases (*1'* applies only to PGC).

phase in more detail. The first step involves walking through all of the file L_p trees and recording the chunk fingerprints in a Bloom filter [18] as chunks that are live. We use a Bloom filter since it reduces memory requirements compared to an index of fingerprints, though even the Bloom filter is quite large (§3.1.2). GC then "sweeps" away any chunks that are unreferenced by reading in container metadata regions and determining whether the chunks are recorded in the Bloom filter. Container metadata regions are shown in Figure 2 and contain a list of fingerprints for chunks within the multi-megabyte container. Live chunks are copied forward into new containers, and old containers are deleted.

A fingerprint match in the Bloom filter could be a false positive, which results in a dead chunk being misidentified as live and retained, but such examples would be rare; these can be bounded by sizing the Bloom filter to limit its false positive rate. Also, the hash functions used by the Bloom filter are changed between runs so that false positives are likely to be removed the next time. Note that live chunks will never be misidentified as dead.

3.1.1 LGC Steps

LGC proceeds in five phases shown in Figure 2: Merge, Enumeration, Filter, Select, and Copy. The first four collectively implement *mark* while the Copy phase is *sweep*. A second set of these phases may be necessary (§3.1.2).

1. Merge: This synchronizes recent index updates from memory to disk so that the next phase can iterate over the on-disk index. In this phase, we also take a snapshot of the file system so we can clean from a consistent viewpoint, which consists of copying the root directory of the file system name space.

2. Enumeration: To identify the live chunks, we enumerate all of the files referenced from the root. Enumeration proceeds in a depth-first order from the top level (L_6) and progresses down to the bottom interior node (L_1).

Enumeration stops at L_1, because it consists of fingerprints referencing data chunks (L_0) that are leaves of the tree. We record the fingerprints for live files in the Live Bloom Filter, indicating that the corresponding chunks should be preserved.

3. Filter: Because the storage system may contain a small[1] number of duplicate chunks stored in containers (*i.e.*, not perfectly deduplicated such as T_{fp} in the fingerprint index of Figure 2), we next determine which instance of the live fingerprints to preserve. The fingerprint-to-container index has the complete list of fingerprints in the system; its entries are organized into buckets of size 512 bytes to 384KB. We then iterate through the index. While any instance of a duplicate fingerprint could be preserved, our default policy is to preserve the instance in the most recently written container, which will prioritize read performance for the most recently written data over older backups [12, 13]. We use the *Live Bloom Filter* to track the existence of a chunk in the system and the Live *Instance* Bloom Filter to track the most recent instance of each chunk in the presence of duplicates. Thus, if a fingerprint exists in the Live Bloom filter, we record the combination of <fingerprint, container_ID> (specifically, the XOR of the two values) in the Live Instance Bloom Filter.

4. Select: We next estimate the liveness of containers to focus cleaning on those where the most space can be reclaimed with the least effort. Since containers tend to either be mostly live or mostly dead, we can reclaim the most space by focusing on mostly dead containers [20]. We iterate through the containers reading the metadata region holding chunk fingerprints. We calculate the <fingerprint, container_ID> value and check the Live Instance Bloom Filter. The liveness of each container is calculated as the number of live fingerprints divided by the total number of fingerprints in the container. We sort the liveness records in memory and select containers below a liveness threshold set dynamically based on our target goal for space to reclaim.

5. Copy: New containers are formed from live chunks copied out of previous containers. When a new container is full, it is written to disk, and dead containers are freed.

6. Summary (not shown): In systems where a system-wide Bloom filter is used to avoid wasted index lookups on disk [29], the Bloom filter is rebuilt to reflect the current live chunks. Its time is approximately the same as the Select phase and is typically dominated by the copy phase. As the Summary phase is being completely eliminated in future versions of DDFS [1], we omit it from further discussion.

While we have presented this algorithm as linear, there are opportunities for parallelism within phases such as reading multiple files during enumeration, multiple index buckets during filtering, and multiple containers during the Select and Copy phases.

3.1.2 Memory Optimization

As described, this algorithm assumes that there is sufficient memory for the Bloom filters to have a low false positive rate. Depending on the system configuration, memory may be limited; a Bloom filter tracking 200B (billion) entries would be hundreds of GBs in size, depending on the desired false positive rate. For such systems, we developed a technique to focus the cleaning algorithm on containers with the most dead chunks.

Before running the four phases described previously, we set a sampling rate based on the memory available for the Bloom filter and the number of chunks currently stored in the system. We then run the *mark* phases (Enumeration, Filter, and Select) but only consider fingerprints that match the sampling criteria before inserting into the Bloom filters. During the Select phase, we choose candidate containers that are mostly dead, but we also limit the total number of containers based on our memory limitation. We then create a Candidate Bloom Filter covering those containers by reading the container metadata regions of the candidate containers and inserting the fingerprints into the Candidate Bloom Filter. The above steps then are repeated, with the exception of Select, though limited by the Candidate Bloom Filter.[2] As an example, the Enumeration phase only adds live fingerprints to the Live Bloom Filter if the fingerprint is in the Candidate Bloom Filter. While this introduces a third Bloom filter, we only need two simultaneously because the Candidate Bloom Filter can be freed after the Enumeration phase. The result is that we apply cleaning in a focused manner to the candidate containers.

However, although the *sweep* phase of actually copying and cleaning containers usually dominates overhead, the need for two sets of *mark* phases adds considerable processing time. Typically, the more data stored on an appliance, the more likely the system will need two sets of *mark* phases. A system that is largely empty will still have enough DRAM to support GC on a loaded system, so the Bloom filters for the *mark* phase may fit in memory. Empirically, we find that about half of GC runs require two phases: typically, systems start less loaded, and over time they store more data and exceed the threshold. In a study of EMC Data Domain systems in 2011, Chamness found that half the systems would be expected to reach full capacity within about six months of the point studied [6].

[1] In practice, we find that about 10% of the data removed during GC is due to duplicates rather than unreferenced chunks.

[2] We refer to the first set of phases as *pre* phases, such as *pre-enumeration*.

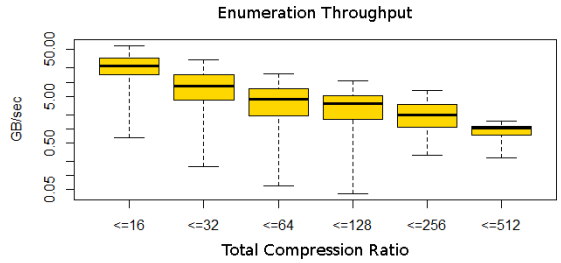


Figure 3: LGC enumeration throughput decreases as TC, bucketed into ranges, increases. Shown for the DD990 on a log-log scale.

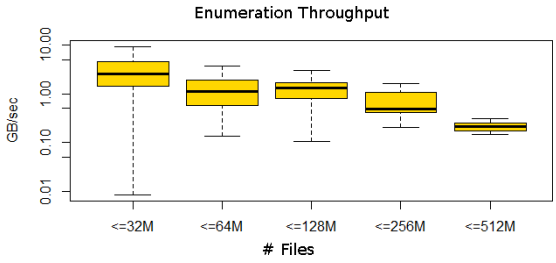


Figure 4: LGC enumeration throughput decreases with high file counts as a function of the number of files, bucketed into ranges. Shown for the DD2500 on a log-log scale.

3.1.3 The Need for Faster Enumeration

While logical enumeration has been successfully used in production systems for over a decade, technology trends discussed in §2 are increasing the total compression ratio and the number of small files. This creates a larger name space and increases enumeration time. Figure 3 shows the LGC enumeration throughput as a function of TC for thousands of deployed systems, using one of our largest available models, the DD990[3] (analysis of a smaller system shows similar results). We define enumeration throughput as GB of physical storage used in the system, divided by GC enumeration time; this normalizes GC performance based on the capacity in use. We show the enumeration range for a bucket with box-and-whisker plots: the whiskers show the 90th and 10th percentile, while the box shows the 75th and 25th, with the median result drawn as a horizontal line within the box. Throughput decreases steadily with TC, demonstrating that LGC enumeration time is related to the logical size of the dataset rather than its physical size. We show this more explicitly in lab experiments in §5.

In Figure 4, we show enumeration throughput versus file counts. We select a smaller system, the DD2500, which is the model[4] with the most files; we see that throughput decreases steadily with higher file counts. We therefore developed PGC, which replaces random I/O for logical enumeration with sequential I/O for physical enumeration. More details about the dataset used to create these figures are presented in §4.1.

3.2 Physical GC

PGC differs from LGC in several ways. First, we use *perfect hashing* [3, 14] (PH) as an auxiliary data structure to drive the breadth-first traversal of the forest of live files. Second, a new method of enumeration identifies live chunks from the containers rather than by iterating through each file. The version of GC that is currently generally in use runs PGC, and as we will describe below, it improves enumeration overhead for the new usage patterns, high deduplication and large numbers of individual files. Then, to eliminate some moderate performance degradation for the traditional use cases, we add additional optimizations (see §3.4).

3.2.1 Perfect Hashing

Determining whether a fingerprint is referenced by a live file can be represented as a membership query, and PH is a data structure to minimize memory requirements. Briefly, PH consists of a hash function that uniquely maps from a key to a position in a hash vector, called a Perfect Hash Vector (PHV). The hash function is generated by analyzing the static set of fingerprints under consideration, and the PHV is used to record membership by setting specific bit(s) assigned to each fingerprint. Previous work has focused on generating a PH function while reducing memory/analysis overheads [3,4,14]. Our technique builds directly from Botelho et al., which used PH in Data Domain to trade off analysis time and memory for secure deletion [5].

As shown in Figure 2, we allocate memory for a PH function and PHV. There is one PH function for a set of fingerprints; using the function, we get the position of the bit for a fingerprint in the PHV. We use PHs to represent the L_6 through L_1 layers of the file tree during enumeration for multiple reasons. First, the false positive rate associated with a Bloom filter would cause the entire branch of chunks under an incorrectly-labeled L_p chunk to remain in the system. Second, our PH is more compact than a Bloom filter, hence it uses less memory for representing these levels. Third, PH assists in level-by-level checksum matching to ensure that there is no corruption (§3.2.2).

[3] Specifications for Data Domain systems appear in §4.

[4] We do not have a good explanation for why customers would use this model to store extremely large numbers of files, more so than other platforms, as no Data Domain system is optimized for that sort of workload. However, if systems with many small files do not require large physical capacities, it would be natural to use lower-end systems.

3.2.2 PGC Steps

PGC has a new Analysis phase (labeled phase 1' in Figure 2) but otherwise has the same phases as LGC, with changes to the Enumeration phase. Again, *mark* is everything but Copy.

1. Merge: This is the same as LGC.

1'. Analysis: We create the PH function by analyzing the fingerprints in the on-disk index. The result is the unique mapping from fingerprint to offset in a PHV. The PH function and vector, in total, use less than 4 bits per L_p fingerprint.

2. Enumeration: Unlike LGC, instead of walking the file tree structure, we perform a series of sequential container scans. We first record the live L_6 fingerprints in the PHV based on files referenced in the namespace by setting a walk bit. Our system has an in-memory structure that records which L_p types (L_6, L_5, ... L_0) exist in each container, so we can specifically scan containers with L_6 chunks. If the L_6 is marked as walk in the PHV, we mark the confirm bit, and we then parse the L_5 fingerprints stored in the L_6 and set the walk bits for those entries in the PHV. We then repeat these steps scanning for L_5 chunks, L_4 chunks, etc. until L_1 chunks are read. When setting the walk bit for any chunk (L_6-L_1), we also record the fingerprint in the Live Bloom Filter, as in LGC.

While it is possible for a container to be read multiple times because it holds chunks of multiple types (e.g. both L_6s and L_5s), the dominant cost comes from L_1 containers, which are read just once. Thus we can decrease the random I/O and overall run time compared to logical enumeration of files, especially with high file counts and deduplication ratios.

3. Filter, **4. Select**, **5. Copy:** These follow LGC.

The correctness of a garbage collection algorithm is critical to prevent data loss. First, GC has to ensure that no chunks are missed during enumeration. Second, if there is already a corruption, GC should stop and alert an administrator to attempt data recovery. The problem is more severe due to deduplication, as a corrupted L_p chunk can result in a large number of files being corrupted due to sharing of the L_p trees. To meet these requirements, physical enumeration calculates checksums per L_P level by making use of the PHV. Due to space considerations we cannot include details, but we briefly describe the approach.

We use two checksums for each level, one for parent chunks (P) and another for child chunks (C). When processing a chunk at level L_k, we XOR its array of referenced hashes for level $L_{k\text{-}1}$ and add it to child checksum C of level $L_{k\text{-}1}$. When processing the chunk at level $L_{k\text{-}1}$, we add the chunk's hash to the P checksum for $L_{k\text{-}1}$. If there is no corruption, at the end of the $L_{k\text{-}1}$ scan, P and C for $L_{k\text{-}1}$ should match. The checksums are calculated and matched for all L_1-L_6 levels, and using the PH function prevents including a hash value repeatedly. Before calculating an XOR, we check whether a chunk is already included in the checksum by checking the appropriate bit in the PHV. We use a PH function that maps from a chunk hash value to two bits, representing 1) finding a chunk as a member of its parent's array of hashes and 2) finding the chunk itself at the next lower level.

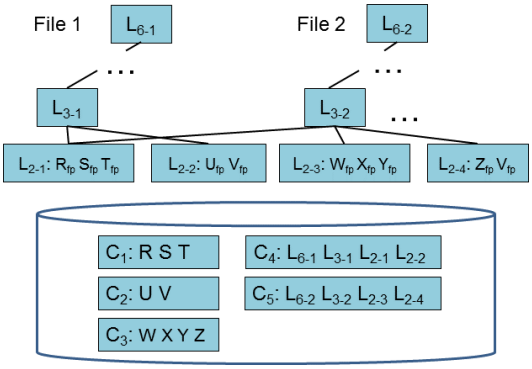


Figure 5: Enumeration example showing the L_p tree for two files and chunks stored in containers on disk.

3.3 Enumeration Example

We next present an example of logical and physical enumeration shown in Figure 5. The L_p trees for two files are shown as well as the location of chunks within containers (C_1-C_5) on disk. File 2 was created by copying File 1 and inserting new content after chunk T. The L_p trees are simplified for this example. Starting with logical enumeration, we begin with File 1. We first read container C_4 holding the root chunk for the file, $L_{6\text{-}1}$, which also reads $L_{3\text{-}1}$, $L_{2\text{-}1}$, and $L_{2\text{-}2}$ into memory. Accessing the L_1 chunks referenced from $L_{2\text{-}1}$ and $L_{2\text{-}2}$ involves reading containers C_1 and C_2. Logically enumerating File 2 follows similar steps; because $L_{3\text{-}2}$ refers to $L_{2\text{-}1}$, container C_1 is read a second time.

Physical enumeration reads all of the L_6 chunks before the L_5 chunks continuing down to the L_1 chunks. In this example, containers C_4 and C_5 are read to access the L_6 chunks. These containers are read again to access the L_3 and L_2 chunks (depending on whether the containers are still in the memory cache). Next the L_1 chunks are read, so containers C_1, C_2, and C_3 are read once to access their L_1 chunks.

3.4 Phase-optimized Physical GC (PGC$^+$)

Even though PGC reduces the time taken by LGC for enumeration in cases of high deduplication or file counts, it adds up to 2 extra analysis phases. PGC$^+$ improves upon PGC in several ways, especially by reducing the

GC phases. As discussed in §3.1.2, LGC and PGC often must run focused cleaning because there is not enough memory for a Bloom filter containing all the fingerprints in the system. On our systems, GC only has sufficient memory to store 25–40% of the total fingerprints in the Bloom filter while keeping a low enough rate of false positives. PGC$^+$ improves on PGC/LGC by making more efficient use of memory, virtually eliminating the need for focused cleaning, using these key ideas:

1. Replace the Live Vector of L_0 chunks with a PHV. This reduces the memory requirement from 6 bits per fingerprint to 2.8 bits per fingerprint (a 2.1× reduction).

2. Remove the Live Instance Vector and eliminate duplicates using a dynamic duplicate removal algorithm in the copy phase (see §3.4.2). This reduces memory requirements by 2×.

Between these changes, we get a 4.2× reduction in memory consumption, enough to use PGC$^+$ by default. However, the system detects whether there is enough memory to store all the fingerprints; if not, it falls back automatically to PGC with its multiple passes. We expect this to happen only in cases with exceptionally high (Lempel-Ziv) compression, resulting in many more chunks being stored. An analysis of customer systems found that only 0.02% of systems exceed the threshold for unique fingerprints, a value that varies across platforms from 10B at the low end to over 200B for the largest systems.[5]

3.4.1 PGC$^+$ Phases

1. Merge: Same as LGC and PGC.

1'. Analysis: Walk the index and create a PH L_P vector and a PH Live Vector. Build a per-container duplicate counter at the same time. This is a hash table indexed by container IDs which contains a count of duplicates across all the other container IDs. This counter is used for the duplicate count estimation algorithm in the select phase.

2. Filter: This phase is omitted for PGC$^+$ (see §3.4.2).

3. Enumeration: Same as PGC, but for every live L_0 chunk, add its fingerprint to the PHV.

4. Select: Iterate through the containers and use the live PHV and the duplicate counter to estimate the liveness of the container.

5. Copy: Copy live chunks forward by iterating containers backwards and dynamically remove duplicates. Backward iteration is needed to preserve the *latest* copy of a fingerprint.

[5]The maximum encountered on any system is 170B, which fits on that system without the need to revert to PGC.

3.4.2 PGC$^+$ Optimizations

Replacing the remaining Bloom filter with a PHV requires optimizations to compensate for extra overheads. Some, *e.g.* parallelization, could be applied to LGC as well.

Parallelization. One problem was an implementation requirement that entries in the on-disk index be returned in order. Originally processing the index was single-threaded, but by dividing the index into distinct partitions, these could be processed by threads in parallel. This parallelism compensates for the extra work of calculating PH functions for all the fingerprints in the system.

Memory lookups. PH functions are organized in an array, so the first step is to find the position in the array. The header information at this location refers to the actual location of the hash function in memory (second access). Accessing the hash function (third access) produces the offset of the bit to set/check. The fourth access is to the actual bit. In addition, initially these accesses were not NUMA-aware. Through a combination of restructuring the memory locations, adding prefetching, and adding NUMA-awareness, we improved PH latency to be comparable to the original Bloom filters.

Dynamic Duplicate Estimation. Just like with LGC and PGC, we estimate the liveness of each container in the PGC$^+$ Select phase to clean the mostly dead containers. LGC and PGC have a Pre-Filter phase before the Select phase, so when the <fingerprint `XOR` container_ID> lookup is done in the Live Instance Vector, we know the count of unique live chunks in that container. With PGC$^+$, there is no Pre-Filter before the Select phase. Thus the lookup in the Live Vector indicates whether chunks are live or dead but for live chunks does not distinguish among duplicates.

To estimate the unique liveness (which excludes live duplicates) of a container in PGC$^+$, we first build a duplicate counter per container (`ANALYSIS_DUP_CNTR`) in the Analysis phase. This counter tracks the number of chunks in a container with duplicates in other containers. Since the Analysis phase is before the Enumeration phase, this counter includes both live and dead duplicate chunks. Then in the Select phase, a Bloom filter is created to track the dead chunks in the container set. During the Select phase, we walk the container set in reverse, from the end to the beginning, to find the latest copy of any duplicate chunk. For every container, we read the container metadata to get the fingerprints and look up these fingerprints in the Live Vector. If the Live Vector indicates the chunk is dead, we insert the dead chunk into the Bloom filter. If the Dead Vector already includes the chunk, we increment a dead duplicate counter (`SELECT_DEAD_DUP_CNTR`) for that container.

Finally, the live duplicate count per container is:

$$LIVE\_DUP\_CNTR = ANALYSIS\_DUP\_CNTR - SELECT\_DEAD\_DUP\_CNTR$$

Hence the container liveness is given by

$$CONTAINER\_LIVENESS = LIVE\_CHUNK\_COUNT - LIVE\_DUP\_CNTR$$

In general, the number of dead chunks is much lower than live chunks (a $\frac{1}{10}$ ratio). Thus the memory needed for the Dead Vector is small. By keeping a Dead Vector to count the dead duplicates, we are able to estimate the correct liveness of containers in a manner similar to LGC and PGC.

Dynamic Duplicate Removal. In LGC and PGC, copy forward works from the lowest to highest container ID and copies live chunks into new containers. In PGC$^+$, to delete older duplicates and preserve the latest copy of the chunk, the copy forward happens in reverse order. Each time a live chunk is copied forward, subsequent chunks with the same fingerprint are treated as duplicates. To mark these duplicates as dead, we clear the corresponding bit in the PHV. The clearing of the liveness bit works because the PHV maps each fingerprint uniquely to the vector. While iterating backward, the next time we see the chunk whose bit is cleared in the PHV, we treat it as dead. Hence, unlike PGC where the Live Instance Vector is needed for removing duplicates, duplicate detection in PGC$^+$ happens in the copy phase. As discussed above, this uses half the memory and can represent 2× more fingerprints.

The copy phase only focuses on the set of containers selected in the select phase, which frees the maximum space in the allotted GC time. But in order to remove duplicates, the PGC$^+$ copy phase must flip the PHV bit for chunks belonging to containers that are not selected for copy forward. That way, older duplicates corresponding to those chunks are treated as dead and can be cleaned. For this, PGC$^+$ must read metadata sections of unselected containers too, and update the PHV. This involves extra container metadata reads, thus it slightly increases the overhead of the copy phase compared to PGC (see §5.2).

3.5 Online GC

It is unreasonable to take the system offline during GC, so while GC is running, our customers are also doing backups. Because our deduplication process checks against any chunks in the system, if a client writes a chunk that is a duplicate of an otherwise unreferenced (dead) chunk, it switches the unreferenced chunk to a referenced (live) state, known as a chunk resurrection. Another point is that open files may not yet be reachable from the root directory, so we specifically track open files and enumerate their L_p trees.

We added a process to inform GC of incoming chunks while GC runs, so chunks can be added to the Live (Instance) Vector. When resurrections take place during the Copy phase, there is a danger that a resurrected chunk may be in the process of being deleted. We see if the resurrection is in the container range currently being processed; if so, we write a (potentially) duplicate chunk for safety, but duplicates written during the Copy phase can be removed during the next GC run. If the resurrection takes place outside of the container range, then we add the <fingerprint, container_ID> to the Live Instance Vector, or the fingerprint to the Live Vector for PGC$^+$.

GC needs not only to support online ingest (adding new data to the system while GC is running), it must also support other tasks such as restores and replication. Thus we have added controls to limit the resource usage of GC, which are configurable on a per-system basis. We call this control *throttling*, but the manner in which GC is throttled has changed from LGC to PGC. Since PGC limits GC performance in ways LGC does not, direct comparisons between the two are problematic.

We take two tacts to address the differences in performance. The first is to consider head-to-head comparisons of LGC and PGC only in the extreme cases that PGC was intended to address. If PGC gives better performance than LGC even when PGC is potentially throttling itself more, we can see that PGC is strictly better than LGC (§5.1). The second is to run in-lab experiments with throttling manually disabled for all GC algorithms (§5.2-5.3).

4 Methodology

This section describes our experimental methodology: we used a combination of telemetry from deployed systems and controlled experiments.

4.1 Deployed Systems

Like many vendors [2, 10], our products can send optional reports on system behavior [24]. We use these reports to extract the following information from each report: serial number, model, system version, date, logical and physical capacity in use, file counts, and GC timings.

We exclude any reports showing <1TB physical capacity in use, <1 hour or >2 weeks of GC execution, or other outliers.[6] This leaves us with one or more reports per system, which may reflect LGC or PGC taking place.

[6]The outliers with low usage or unusually fast GC are uninteresting (for example, a system that is not in active use and has no garbage to collect). Any rare cases of unusually high times are attributed to bugs that were subsequently fixed.

Most of our evaluation focuses on a large set of deployed systems running LGC. We use a second set of deployed systems running PGC (PGC$^+$ not yet being available) for head-to-head comparisons of the extreme cases (high deduplication rates or file counts). We looked at systems that were upgraded from LGC to PGC and did not change capacity, file counts, or deduplication ratio by more than 5%. We found that for the more "typical" systems, there was moderate (and occasionally significant) degradation when running PGC instead of LGC, but it is not possible to attribute that effect to the PGC algorithm versus throttling effects or one-time overheads occurring in the process of the upgrade. We exclude throttling in the in-lab experiments reported later. The timings were collected from Jan-2014 to Sep-2016; both sets (LGC and PGC) contain thousands of systems.

There are various ways to evaluate GC performance, and in particular, evaluate the change in the context of a particular system. Our usual approach to compare similar systems is to sort them by some metric, such as physical capacity, and divide them into buckets. Note that these buckets are marked as $\leq$ some value, but they reflect only those points that are greater than the next lower bucket (*i.e.*, they are not cumulative across all lower buckets). In addition we generally find that showing the buckets on a log scale emphasizes the outliers. Since PGC$^+$ is only recently available to customers, we cannot use telemetry to evaluate systems using PGC$^+$.

4.2 Controlled Experiments

To compare LGC, PGC, and PGC$^+$ in a controlled manner, we perform a number of GC experiments in a lab environment. To ensure that the data is the same across experiments, we obtain or generate the test dataset, then reset to the state of the test dataset anytime the system is modified. For most experiments, we use a load generator, the same one used to evaluate content sanitization in DDFS [5]; it is similar to the work of Tarasov, *et al.* [23]. The tool creates files of a specified size, using a combination of random and predictable content to get desired levels of deduplication and compression. Each new file contains significant content from the previous generation of the same "lineage," while some content is deleted as new content is added. There should not be deduplication across independent lineages, which can be written in parallel to achieve a desired throughput.

For some of the experiments, we went through some extra steps to create what we call our *canonical dataset*. As we wrote new data, we ran GC several times to age the system and degrade locality: for example, ingesting to 5% of capacity and using GC to reset to 3% of capacity before ingesting to 7% and running GC again. (We note that the number of unfiltered duplicates removed in this process is higher than the number of duplicates that typically appear in the field (10% or less.)) Ingestion was halted when we reached 10% of the physical capacity of the largest system, the DD990. The canonical dataset is 1.1 PiB of logical data deduplicated and compressed to 30.1 TiB of physical data (36.6× TC). We used this dataset to compare different appliances and also as the starting point for evaluating high deduplication rates.

The canonical dataset is about 25% of the capacity of the DD860, which we used for other experiments. To evaluate the impact of physical capacity in use, we wanted to start with a lower capacity, so we ingested 20% of physical capacity at a time. For this experiment we did not run destructive GC (physically reorganizing data and removing extra duplicates) during ingestion. The TC for that experiment varied from 17.9×–18.3×.

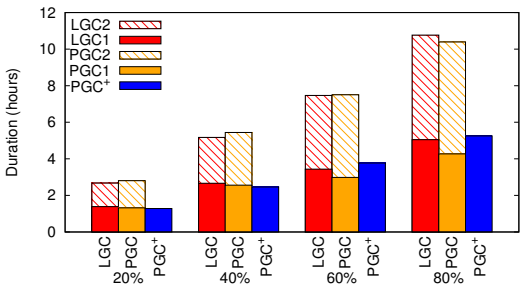
To start with the same predictable content, we use *collection replication* [7] to make a total copy of the data on another appliance. To focus on the *mark* part of GC, we can identify what data should be reclaimed but terminate GC before initiating *sweep* and modifying the file system. Multiple *mark* runs (of any GC type) can be measured without modifying the state of the system. If we include the copy phase, i.e. *sweep*, we replicate back from the saved copy to restore the system to the original state. Each 4MB container, as well as the index, will be the same, though containers may be stored in different physical locations on disk. We use replication to evaluate high file counts, copying from a quality assurance system that has been seeded with 900M files.

Since we find that many GC runs require two *mark* phases, it is important to evaluate runs with both one *mark* pass and two. We do this by collecting timings of both sets and indicating them separately in the results via stacked bar graphs. LGC-1 and PGC-1 are the lower (solid) bars; these may be compared directly with the single PGC$^+$ bar, or the higher shaded bars labeled LGC-2 and PGC-2 may be considered. In general, less full systems need one pass and more full systems need two.

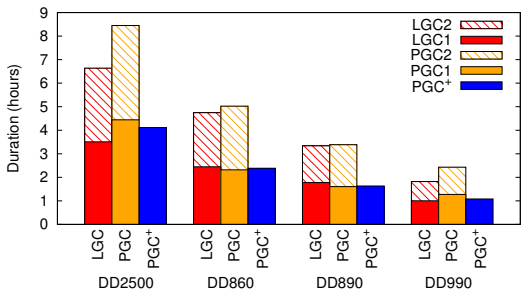
4.2.1 Test Environment

We use a variety of backup appliances for our tests. We report the time taken for the *mark* phase for our canonical dataset on four separate systems. We then use a specific system to evaluate some aspect of the workload, such as *copy* phase variability, capacity in use, high file counts, or variation in duplication ratios.

Table 1 shows specifications for the systems we used in our tests. It shows cores, memory, physical capacity available, and the amount of that capacity used by the canonical dataset, ranging from 10–25%. After comparing all four systems on one dataset, we use the DD860 for extensive comparisons on different workloads.



(a) GC duration by physical space



(b) GC duration by platform

Figure 6: (a) GC duration is a linear function of physical space, but PGC$^+$ is consistently faster than the other algorithms. PGC is consistently slightly slower than LGC. (b) GC duration varies across platforms. PGC$^+$ consistently outperforms LGC and PGC on all platforms. PGC is consistently slower than LGC when two phases are required, but the difference is variable for just one phase, with PGC slightly faster in two cases.

Systems	DD2500	DD860	DD890	DD990
CPU (cores×GHz)	8 × 2.20	16 × 2.53	24 × 2.80	40 × 2.40
Mem (GB)	64	70	94	256
Phy. Capacity(TiB)	122	126	167	319
Canonical Dataset util.(%)	25	25	19	10

Table 1: System Specifications.

5 Evaluation

In this section we provide a detailed comparison of LGC, PGC, and PGC$^+$. §5.1 provides a high-level comparison across a set of deployed systems with anomalous enumeration performance (thus these omit PGC$^+$). §5.2 presents lab-based experiments comparing all three GC variants on basic datasets (nominal deduplication and file counts), investigating the penalty from PH analysis when moving from LGC to PGC and the compensating benefit from eliminating multi-round *marking* before the sweep phase (PGC$^+$). §5.3 then focuses on specific challenges for LGC, high deduplication ratios and large file counts.

5.1 Deployed Systems

We start by evaluating the change from LGC to PGC in deployed systems. When comparing an upgraded system, we can consider the time it took to run LGC before the upgrade and the time it took to run PGC after the upgrade. (Recall that we only consider upgraded systems that are within 5% of each other in various metrics in both cases).

Across all systems, we find that 75% of systems suffer some form of performance degradation when moving from LGC to PGC. As stated before, this is due to various factors including the change to throttling. As customers are able to upgrade to PGC$^+$, we focus on emphasizing the benefits of PGC over LGC in the extreme scenarios, then discuss lab-based comparisons in the remainder of this section. We find that PGC improves enumeration performance over LGC by as much as 20× in the case of the greatest improvement for high deduplication, and by 7× for the greatest improvement for high file counts.

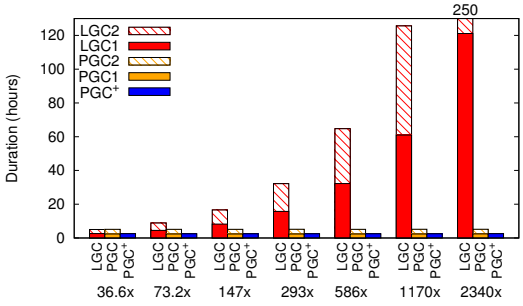
5.2 Standard Workloads

To start, we explain why we focus on *mark* performance rather than *sweep* (Copy). The copy times are proportional to physical space used, and they are not problematic (the maximum time for any system is always "reasonable" because resources such as cores and memory scale with disk capacity). Copy for PGC$^+$ is slightly slower than for PGC or LGC (which use an identical algorithm) due to the PHV overhead, but we measured this to be only a 2–3% increase. Running Copy on the same workload three times per algorithm, we found a tiny variation among multiple runs (a standard deviation less than 0.5%). Thus, for the remainder of the paper, we focus on *mark* performance, though we give some examples of the PGC$^+$ *sweep* times for comparison.

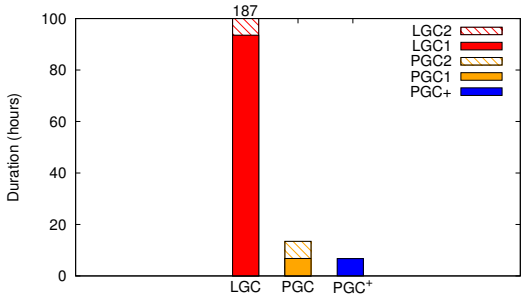
Within the *mark* phase, for the canonical dataset, the analysis portion of PGC$^+$ is 10–20% of total elapsed time. It is about 20% of elapsed time on the DD990 and about 10% on the other platforms.

We evaluated performance in a controlled environment using the load generator described in §4.2. We ingested data to different fractions[7] of the overall capacity of a specific system (DD860) and measured the time taken for the *mark* phases of LGC, PGC and PGC$^+$ to complete. As we can see in Figure 6(a), there is roughly a linear relationship between the GC duration and the physical capacity in use.

[7]The amount ingested is somewhat approximate, but the trend is clear.



(a) GC duration by total compression (log scale)



(b) GC duration with high file count

Figure 7: (a) As the deduplication ratio increases, LGC duration increases substantially while PGC and PGC$^+$ stay constant. (b) While LGC performance is poor when the number of files is unusually high, PGC and PGC$^+$ are substantially faster.

As mentioned in §3.1.2, the need for two phases arises from the number of chunks stored in the system; this is usually due to higher capacity in use, but higher compression can result in more chunks being stored in the same physical space. PGC is generally slightly slower than LGC at lower utilizations and slightly faster at higher ones; PGC$^+$ consistently completes at least 2× faster than either LGC or PGC when two phases are required. For low space utilization PGC$^+$ is slightly faster than LGC and PGC when a single phase is required; at the highest level tested, it would be about 20% slower than PGC if only one phase were required; however, two phases are needed for this dataset, so PGC$^+$ is uniformly faster.

Figure 6(b) shows the results of *mark* for different GC algorithms running with the canonical dataset on the four platforms. In general, higher-end platforms have better GC performance due to more resources (e.g., CPU, memory, storage bandwidth). There is some variation in the relative performance of the algorithms across platforms, but in general, PGC$^+$ is close to LGC and PGC for a single *mark* phase and much faster for two. DD2500, the lowest-end platform, shows the greatest degradation (17%) moving from LGC to PGC$^+$ when just one phase is needed. To give an idea of the relative performance of the *copy* phase, across these platforms the time for *copy* is $2.3-2.6\times$ that of the PGC$^+$ *mark* times.

5.3 Problematic Workloads

The move to PGC was spurred by two types of problematic workloads, high deduplication ratios and high file counts. These are depicted in Figure 7 using the DD860. In Figure 7(a), the *fastcopy* [7] system command is used to repeatedly double the TC from a moderate 36.6× up to an extreme 2340×. The enumeration time for LGC increases with deduplication ratio, while the times for PGC and PGC$^+$ are a function of the physical space, which is nearly constant. In the worst case, PGC$^+$ is nearly two orders of magnitude faster than LGC, and even if LGC needs only one *mark* phase, LGC is 47× slower. Note that the values for a 36.6× TC are the same as the DD860 in Figure 6(b); refer there for detail.

Figure 7(b) shows the GC *mark* times for a system with over 900M files, for different GC algorithms.[8] The system has a TC of only 2.3×, meaning there is almost no deduplication. PGC and PGC$^+$ take substantially less time than LGC because LGC has to traverse the L_p tree multiple times based on the number of logical files. LGC also induces substantial random I/Os. In contrast to LGC, PGC/PGC$^+$ can traverse the L_p tree in a limited sequence of passes and enumerate files with sequential I/Os. Compared with LGC, PGC$^+$ is 13.8×–27.8× faster for one or two *mark* phases, respectively. PGC$^+$ is virtually identical to a single phase of PGC and twice as fast than the two-phase run.

6 Related Work

While many deduplicated storage papers mention their technique to remove unreferenced chunks, only a few present a detailed implementation. We presented an overview of the differences between reference counts and mark-and-sweep algorithms in §2 and now provide more discussion of related work.

Fu et al. [8] maintained reference counts for containers as part of a technique to reduce fragmentation for faster restores. A limitation of their work is that their garbage collection technique only supports first-in-first-out deletion, while our system supports an arbitrary deletion pattern. Strzelczak et al. [22] describe

[8] This is data replicated to a DD860 from an internal Quality Assurance system that was seeded over the period of many months; it takes too long to write the individual files for us to use other high file counts, so we demonstrate the savings at the extreme.

a distributed reference count algorithm for HYDRAstor. They use an epoch-based approach to allow cleaning to take place while the system allows new writes. Their indexing mechanism is not described, but there is an indication that they have sufficient memory to track reference counts. Our system has smaller chunks (*e.g.* ~8KB) and uses compact data structures to reduce memory requirements. Simha et al. [21] describe a technique for efficiently handling incremental changes, though they do not generate full backups from incremental writes as our system supports, so they have a smaller name space to enumerate than our system. Their reference count technique leverages known expiration times for snapshots, while we support any deletion order. Without expiration times in advance, it would be necessary to walk an entire tree to update reference counts in the case of a snapshot or *fastcopy* [7], and deletions similarly require mass updating. Grouped Mark and Sweep [9] marks referenced containers. Their file representation has direct references to physical blocks, so it is unclear how they can copy forward live chunks to consolidate partially dead containers. In contrast, our file representation uses fingerprints and we use a fingerprint-to-container index to support container cleaning. One might use SSDs to store the reference counts, but then one must address write amplification from these updates. In future years other forms of memory, nonvolatile or otherwise, may be cost-effective for systems of this scale. Finally, reference counts are also difficult to maintain correctly in the presence of complex error conditions.

The most similar work to our own is the sanitization technique presented by Botelho et al. [5]. Like us, they used perfect hashes to compactly represent live references in deduplicating storage. A key difference is that they focused on sanitization rather than garbage collection. Sanitization is the process of securely deleting data to prevent the leakage of confidential information, so sanitization has the requirement of removing all unreferenced chunks. This means that they created a PH function and vector over *all* chunks (L_0-L_6) so that any unreferenced chunk could be removed; they still performed logical rather than physical enumeration of the file system. Our physical enumeration technique could possibly replace logical enumeration in their algorithm, and our other optimizations to PH are also applicable.

Techniques that defer cleaning and focus on the most efficient areas to clean tend to have some basis in early work on cleaning log structured storage [15, 20]. Other LFS optimizations such as hole-plugging [26] do not work well at the granularity of chunks that are packed and compressed in large units.

While our work focuses on garbage collection, the differences between the logical and physical view of a storage system were noted by Hutchinson, et al. [11]. In that work, the authors found that backing up a system by accessing its blocks at a physical level provided better performance than accessing blocks one file at a time. Thus the optimization of PGC to iterate through the physical locations of system metadata is similar to their optimization when copying data in the first place, but deduplication magnifies the distinction.

7 Conclusion

The shift in workloads has required a new approach to garbage collection in a deduplicating storage system. Rather than a depth-first mark-and-sweep GC algorithm, tracking live data from the perspective of individual files, we have moved to a breadth-first approach that takes advantage of the physical layout of the data.

PGC turns large numbers of random I/Os into a set of sequential scans of the entire storage system. It works well when deduplication is high (100–1000× rather than 10–20×), and it works well when the storage system is used for hundreds of millions of relatively small files rather than thousands of large tar-like backup files.

Because of the other overheads of PGC, including the analysis necessary to use the space-efficient and accurate perfect hash functions [3, 14], the original PGC approach is not uniformly faster than the LGC approach it replaced. The improved PGC algorithm, referred to as PGC$^+$, uses PH in an additional way to reduce memory requirements enough to avoid two *mark* sequences during GC. We have demonstrated that it is comparable to the original LGC on traditional workloads whose fingerprints fit into memory (requiring a single *mark* phase), significantly faster when two passes are required, and orders of magnitude better on problematic workloads.

Acknowledgments

We thank the many Data Domain engineers who have contributed to its GC system over the years, especially Ed Lee, Guilherme Menezes, Srikant Varadan, and Ying Xie. We appreciate the feedback of our shepherd André Brinkmann, the anonymous reviewers, and several people within Dell EMC: Bhimsen Banjois, Orit Levin-Michael, Lucy Luo, Stephen Manley, Darren Sawyer, Stephen Smaldone, Grant Wallace, and Ian Wigmore. We very much appreciate the assistance of Mark Chamness and Rachel Traylor with data analysis; Lang Mach, Murali Mallina, and Mita Mehanti for the QA dataset access; and Duc Dang, David Lin, and Tuan Nguyen for debugging and performance tuning.

References

[1] ALLU, Y., DOUGLIS, F., KAMAT, M., SHILANE, P., PATTERSON, H., AND ZHU, B. Evolution of the Data Domain file system. *IEEE Computer*. To appear.

[2] AMVROSIADIS, G., AND BHADKAMKAR, M. Identifying trends in enterprise data protection systems. In *USENIX Annual Technical Conference (ATC'15)* (July 2015), pp. 151–164.

[3] BELAZZOUGUI, D., BOTELHO, F. C., AND DIETZFELBINGER, M. Hash, displace, and compress. In *Algorithms-ESA 2009*. Springer, 2009, pp. 682–693.

[4] BOTELHO, F. C., PAGH, R., AND ZIVIANI, N. Practical perfect hashing in nearly optimal space. *Information Systems* (June 2012). http://dx.doi.org/10.1016/j.is.2012.06.002.

[5] BOTELHO, F. C., SHILANE, P., GARG, N., AND HSU, W. Memory efficient sanitization of a deduplicated storage system. In *USENIX Conference on File and Storage Technologies (FAST'13)* (Feb 2013).

[6] CHAMNESS, M. Capacity forecasting in a backup storage environment. In *25th Large Installation System Administration Conference (LISA'11)* (2011).

[7] EMC CORP. *EMC® Data Domain® Operating System, Version 5.7, Administration Guide*, Mar. 2016. `https://support.emc.com/docu61787_Data-Domain-Operating-System-5.7.1-Administration-Guide.pdf?language=en_US`.

[8] FU, M., FENG, D., HUA, Y., HE, X., CHEN, Z., XIA, W., HUANG, F., AND LIU, Q. Accelerating restore and garbage collection in deduplication-based backup systems via exploiting historical information. In *USENIX Annual Technical Conference (ATC'14)* (2014).

[9] GUO, F., AND EFSTATHOPOULOS, P. Building a high-performance deduplication system. In *USENIX Annual Technical Conference (ATC'11)* (2011).

[10] HAO, M., SOUNDARARAJAN, G., KENCHAMMANA-HOSEKOTE, D., CHIEN, A. A., AND GUNAWI, H. S. The tail at store: a revelation from millions of hours of disk and ssd deployments. In *14th USENIX Conference on File and Storage Technologies (FAST'16)* (2016), pp. 263–276.

[11] HUTCHINSON, N. C., MANLEY, S., FEDERWISCH, M., HARRIS, G., HITZ, D., KLEIMAN, S., AND O'MALLEY, S. Logical vs. physical file system backup. In *Third Symposium on Operating Systems Design and Implementation (OSDI'99)* (1999).

[12] KACZMARCZYK, M., BARCZYNSKI, M., KILIAN, W., AND DUBNICKI, C. Reducing impact of data fragmentation caused by in-line deduplication. In *Proceedings of the 5th Annual International Systems and Storage Conference* (2012), ACM, p. 15.

[13] LILLIBRIDGE, M., ESHGHI, K., AND BHAGWAT, D. Improving restore speed for backup systems that use inline chunk-based deduplication. In *USENIX Conference on File and Storage Technologies (FAST'13)* (Feb 2013).

[14] MAJEWSKI, B. S., WORMALD, N. C., HAVAS, G., AND CZECH, Z. J. A family of perfect hashing methods. *The Computer Journal 39*, 6 (1996), 547–554.

[15] MATTHEWS, J. N., ROSELLI, D., COSTELLO, A. M., WANG, R. Y., AND ANDERSON, T. E. Improving the performance of log-structured file systems with adaptive methods. In *16th ACM Symposium on Operating Systems Principles* (1997), SOSP'97.

[16] MERKLE, R. C. A digital signature based on a conventional encryption function. In *Advances in Cryptology CRYPTO'87* (1988), Springer, pp. 369–378.

[17] PAULO, J., AND PEREIRA, J. A survey and classification of storage deduplication systems. *ACM Computing Surveys 47*, 1 (2014).

[18] PUTZE, F., SANDERS, P., AND SINGLER, J. Cache-, hash-and space-efficient bloom filters. In *Experimental Algorithms, 6th International Workshop, WEA 2007, Lecture Notes in Computer Science 4525*. Springer, 2007, pp. 108–121.

[19] ROSELLI, D. S., LORCH, J. R., ANDERSON, T. E., ET AL. A comparison of file system workloads. In *USENIX Annual Technical Conference, general track (ATC'00)* (2000), pp. 41–54.

[20] ROSENBLUM, M., AND OUSTERHOUT, J. K. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems (TOCS) 10*, 1 (1992), 26–52.

[21] SIMHA, D. N., LU, M., AND CHIUEH, T.-C. A scalable deduplication and garbage collection engine for incremental backup. In *6th International Systems and Storage Conference (SYSTOR'13)* (2013).

[22] STRZELCZAK, P., ADAMCZYK, E., HERMAN-IZYCKA, U., SAKOWICZ, J., SLUSARCZYK, L., WRONA, J., AND DUBNICKI, C. Concurrent deletion in a distributed content-addressable storage system with global deduplication. In *USENIX Conference on File and Storage Technologies (FAST'13)* (2013).

[23] TARASOV, V., MUDRANKIT, A., BUIK, W., SHILANE, P., KUENNING, G., AND ZADOK, E. Generating realistic datasets for deduplication analysis. In *USENIX Annual Technical Conference (ATC'12)* (Jun 2012).

[24] WALLACE, G., DOUGLIS, F., QIAN, H., SHILANE, P., SMALDONE, S., CHAMNESS, M., AND HSU, W. Characteristics of backup workloads in production systems. In *USENIX Conference on File and Storage Technologies (FAST'12)* (2012).

[25] WEI, J., JIANG, H., ZHOU, K., AND FENG, D. MAD2: A scalable high-throughput exact deduplication approach for network backup services. In *Massive Storage Systems and Technologies (MSST'10)* (2010), IEEE, pp. 1–14.

[26] WILKES, J., GOLDING, R., STAELIN, C., AND SULLIVAN, T. The HP AutoRAID hierarchical storage system. *ACM Transactions on Computer Systems (TOCS) 14*, 1 (1996), 108–136.

[27] WILSON, P. R. Uniprocessor garbage collection techniques. In *Memory Management*. Springer, 1992, pp. 1–42.

[28] XIA, W., JIANG, H., FENG, D., DOUGLIS, F., SHILANE, P., HUA, Y., FU, M., ZHANG, Y., AND ZHOU, Y. A comprehensive study of the past, present, and future of data deduplication. *Proceedings of the IEEE 104*, 9 (Sept. 2016), 1681–1710.

[29] ZHU, B., LI, K., AND PATTERSON, H. Avoiding the disk bottleneck in the Data Domain deduplication file system. In *USENIX Conference on File and Storage Technologies (FAST'08)* (Feb 2008).