A Scalable Deduplication and Garbage Collection Engine for Incremental Backup

Dilip Nijagal Simha
Stony Brook University & ITRI
New York, USA
dnsimha@cs.stonybrook.edu

Maohua Lu
IBM Almaden Research Labs
California, USA
lum@us.ibm.com

Tzi-cker Chiueh
Stony Brook University & ITRI
Hsinchu, Taiwan
tcc@itri.org.tw

ABSTRACT

Very large block-level data backup systems need scalable data deduplication and garbage collection techniques to make efficient use of the storage space and to minimize the performance overhead of doing so. Although the deduplication and garbage collection logic is conceptually straightforward, their implementations pose a significant technical challenge because only a small portion of their associated data structures could fit into memory. In this paper, we describe the design, implementation and evaluation of a data deduplication and garbage collection engine called *Sungem* that is designed to remove duplicate blocks in incremental data backup streams. *Sungem* features three novel techniques to maximize the deduplication throughput without compromising the deduplication ratio. First, *Sungem* puts *related* fingerprint sequences, rather than fingerprints from the same backup stream, into the same container in order to increase the fingerprint prefetching efficiency. Second, to make the most of the memory space reserved for storing fingerprints, *Sungem* varies the sampling rates for fingerprint sequences based on their stability. Third, *Sungem* combines reference count and expiration time in a unique way to arrive at the first known incremental garbage collection algorithm whose bookkeeping overhead is proportional to the size of a disk volume's incremental backup snapshot rather than its full backup snapshot. We evaluated the *Sungem* prototype using a real-world data backup trace, and showed that the average throughput of *Sungem* is more than 200,000 fingerprint lookups per second on a standard X86 server, including the garbage collection cost.

Categories and Subject Descriptors

D.4.2 [**Software**]: OPERATING SYSTEMS —*Storage Management*

Keywords

Deduplication, garbage collection, backup storage

Copyright © 2013 Association for Computing Machinery. ACM acknowledges that this contribution was authored or co-authored by an employee, contractor or affiliate of the national government of Taiwan. As such, the government of Taiwan retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only.

SYSTOR'13, June 30 – July 02 2013, Haifa, Israel

Copyright 2013 ACM 978-1-4503-2116-7/13/06 ...$15.00.

1. INTRODUCTION

As data redundancy grows due to increased data sharing [10], data deduplication has become a key functionality requirement of commercial storage system products, especially secondary storage or data backup systems [18, 9]. The two key metrics for assessing a data deduplication technology are *data deduplication ratio*, the ratio between the amount of data before deduplication and that after deduplication, and *data deduplication throughput*, the throughput at which a data deduplication engine determines whether a backup data block is a duplicate or not. Data deduplication can take place at multiple levels of abstraction. *File-level* deduplication uses individual files as basic units of deduplication, whereas *block-level* deduplication [2, 20] uses fixed-sized or variable-sized disk blocks as basic units of deduplication. Although *file-level* deduplication can leverage file-related information to optimize the deduplication efficiency [2], it requires file system support and thus is not easily portable. In contrast, *block-level* deduplication operates at the disk volume level and is transparent to high-level software using the block access interface, and thus is the focus of this work.

When applied to data backup systems, deduplication technologies can be further categorized according to whether they are designed to handle outputs from a full backup operation or from an incremental backup operation. When taking a full backup of a disk volume, a backup client takes a snapshot of the disk volume, calculates a fingerprint value for every block in the volume, transmits all calculated fingerprint values to the backup server, receives responses from the backup server, and transmits only those blocks that the backup server considers new. Because the amount of change to a disk volume between two consecutive backup operations is expected to be a small percentage of the volume's size, the snapshots of consecutive full backup operations are expected to overlap significantly. As a result, the data deduplication ratio of a full backup snapshot is typically very high, sometimes up to 100% [20]. However, such data deduplication efficiency is somewhat artificial, because there is a simpler way than data deduplication to remove the redundancy between consecutive full backup snapshots. In an incremental backup operation, a backup client employs *block change tracking* (BCT) [12] technology to keep track of all changes to a disk volume, and at backup time, transmits to the backup server only fingerprint values associated with modified disk blocks. The backup server consults with the data deduplication engine to find out if any of the modified blocks is a duplicate. If a modified block is a duplicate,

it is not transmitted. Compared with a full backup operation's fingerprint stream, the fingerprint stream submitted by an incremental backup operation to the backup server not only is much smaller and more fragmented, but also exhibits much less locality.

Block-level data deduplication works by computing a fingerprint value for every disk block stored in the repository, organizing the fingerprints into an index structure, and determining whether an incoming disk block is a duplicate by computing the block's fingerprint and consulting with the fingerprint index to see if there is a match. So fundamentally the data deduplication problem is one of index look-up, where the index is too large to fit into memory. For example, for a one-petabyte data backup system whose disk block size is 4KB, the space required to hold all the fingerprints is 4TB, assuming each fingerprint costs 16 bytes, and the associated fingerprint index most likely cannot fit into a standard server's main memory. Worse yet, there is no spatial locality among neighboring fingerprint values because fingerprints are generated by hash functions. As a result, in the worst case every look-up into the fingerprint index could result in at least one disk I/O, which drastically slows down the data backup process.

Several solutions to the performance problem associated with fingerprint index look-up have been proposed, and they all are based on the following assumptions:

- Data being shared among users, e.g., a PowerPoint file or a photo image file, is typically larger than a single disk block.
- A data sharing unit typically spans a consecutive sequence of logical blocks in a disk volume.

Taken together, they suggest that when multiple instances of a data sharing unit are backed up, multiple identical sequences of logical blocks are likely to appear in the backup streams. There are two ways to exploit the fact that identical sequences of logical disk blocks are repeated multiple times. First, the fingerprints of the blocks in a recurring sequence are stored in a container, and the entire container is fetched into memory whenever any fingerprint of an incoming block in a backup stream matches a fingerprint in the recurring sequence. This technique, used in Data Domain [20], essentially corresponds to *prefetching*, because fingerprints associated with a basic data sharing unit are likely to be accessed together, and thus should be brought into memory via one disk I/O operation to amortize the disk access overhead. Second, instead of storing every fingerprint in the fingerprint index, it is sufficient to put into the fingerprint index only one of the fingerprint values associated with a data sharing unit, because it takes only one fingerprint match to bring in the remaining fingerprints in the same data sharing unit. This technique, known as *sparse indexing* [14], corresponds to *sampling* of the fingerprint index, and is able to significantly reduce the fingerprint index size and thus increase the probability of looking up the fingerprint index without incurring disk I/O.

This paper describes the design, implementation and evaluation of a data deduplication engine called *Sungem*, which is designed as a component of a data backup system that supports incremental backup operations from primary storage servers, and provides two functions: (a) determining if a new backup block is a duplicate and thus should not be deposited into the backup system through a fingerprint check, and (b) updating the garbage collection data structures associated with deposited blocks in the backup system. *Sungem* features the following innovations:

1. Putting *related* fingerprint sequences, rather than fingerprints from the same backup stream, into the same container in order to increase the fingerprint prefetching efficiency,
2. Using different sampling rates, rather than a fixed sampling rate, for fingerprint sequences with different maturity to further cut down the fingerprint index size without degrading the data deduplication ratio, and
3. Combining reference count and expiration time to arrive at a novel garbage collection (GC) algorithm whose bookkeeping overhead is both *distributed* over each backup operation and proportional only to the size of the change to a disk volume between consecutive backup operations and is independent of the volume's size itself.

2. RELATED WORK

2.1 Content-Addressable Storage

Content-addressable storage (CAS) [17, 9, 7, 18] deduplicates data in nature because it employs the fingerprint (i.e., SHA1 hash value) of a data block instead of a logical block number to access the data block. If two data blocks are identical, their fingerprints are the same and only one data copy is stored. Venti [17] pioneers CAS and it manages a standalone storage node. HYDRAstor [9] is a distributed backup CAS system with fault tolerance in the design. The Foundation [18] leverages commodity USB external hard drives to archive digital files in a similar fashion to Venti.

CAS systems employ various locality-preserving techniques to improve the performance. HYDRAstor [9] chunks backup images into segments based on the content, and each segment is stored continuously on commodity storage devices to preserve data locality. In Foundation [18], a 16 MB data segment is stored continuously to preserve data locality for sequential read and fingerprint caching. However, these CAS systems do not focus on the scalable deduplication when the deduplication metadata (e.g., FI) cannot fit into RAM.

2.2 Deduplication Techniques for Backup Storage Systems

In DataDomain [20], Li et al. proposed a set of scalable deduplication techniques for a disk-to-disk (D2D) [1] data backup system. Because the fingerprint index cannot fit into memory, DataDomain employs two techniques to minimize the overhead due to I/O access of the fingerprint index. Namely, the two techniques are (1) a bloom filter-based [3] summary vector to avoid unnecessary fingerprint lookup, and (2) a locality-preserving data placement scheme to leverage the spatial locality of the input fingerprints. These two deduplication techniques are optimized for full backups. However, it is not clear how effective they are for incremental backups because data locality is not easy to identify for incremental backups.

The sparse indexing scheme [14] contributes a third deduplication technique by employing a sampling fingerprint index instead of a whole fingerprint index to further reduce the memory usage of deduplication in a D2D data backup system. The insight of the proposed scheme is that duplicated

data blocks tend to be in a consecutive range with a non-trivial length. A match of sampling fingerprint values in the range indicates the matching of the whole range with a high probability. However, the sparse indexing scheme enforces a fixed sampling rate for all consecutive ranges without differentiating among consecutive ranges of different degrees of stability. In contrast, our proposed scheme varies the sampling rate based on the stability of the consecutive range.

Extreme-Bin [2] deduplicates data based on files in a distributed storage system. Fingerprint Index (FI) is distributed to multiple nodes. Each input file has a sampled fingerprint to route it to a distributed node, and a duplicate block in the input file is detected by consulting the block's fingerprint with the distributed FI. Besides a sampled fingerprint, each file has a whole-file fingerprint. A match of the whole-file fingerprint indicates that the whole file is a duplicate without comparing individual fingerprints in the file. In our proposed deduplication technique, each segment has a whole-segment fingerprint, which is more flexible. However, to save RAM space, it is desirable to have the file-level information, including the file identifier and file offsets, to represent the segment instead of individual physical block addresses.

2.3 Garbage Collection Techniques for Deduplication Storage Systems

HYDRAstor [9] employs the mark-and-sweep garbage collection (GC) technique [6] to reclaim physical blocks. As the per-peer machine storage capacity grows, the mark-and-sweep garbage collection is not scalable because the *m*ark phase takes longer for more logical volumes and the *m*ark phase can be prohibitively long if the counter of all physical blocks cannot fit into RAM. In *Sungem*, the bookkeeping overhead at backup time for GC is proportional to the change to a disk volume between consecutive backups rather than the volume itself.

3. LOCALITY-DRIVEN DATA DEDUPLICATION

3.1 System Architecture

Figure 1 shows the high-level data flow of an incremental data backup system that uses *Sungem* as the deduplication engine. A backup client, which could be a network file server or a DBMS server, constantly keeps track of block-level modifications, and at backup time submits to the backup server a stream of fingerprint values associated with the blocks that are modified since the last backup, ordered by their logical block addresses. If a backup client needs to backup multiple disk volumes, it will produce one fingerprint stream for each of them. Upon receiving a fingerprint stream, the backup server partitions it into *input segments*, each of which corresponds to a sequence of blocks with *contiguous* logical block numbers. For each input segment, *Sungem* first looks it up in the *sampled fingerprint index* (SFI), which contains sampled fingerprints from previously seen fingerprint segments and the IDs of the on-disk containers to which these sampled fingerprints belong. For every fingerprint in an input segment that hits in the SFI, *Sungem* brings its corresponding container into memory to determine if the input segment indeed contains any duplicate instances of any previously seen fingerprint sequence. To speed up container fetching, *Sungem* uses a memory-resident *container cache* to

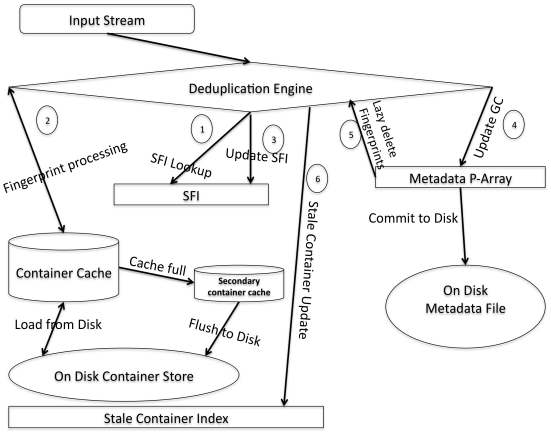

Figure 1: *Abstract view of the proposed data deduplication engine*

hold containers recently fetched from the on-disk container store, which are organized in an LRU order. For every input fingerprint, *Sungem* either declares it as a duplicate and returns the physical block number of the physical block in the repository that already has the same fingerprint, or declares it is not a duplicate. The backup server uses the outputs from *Sungem* to create the underlying representation for a backup snapshot.

3.2 Fingerprint Segmentation and Placement

Intuitively, a *Sungem* segment is meant to capture the notion of a data sharing unit. From an input fingerprint stream, *Sungem* first partitions it into multiple *input segments*, each of which corresponds to a sequence of fingerprints whose associated blocks have consecutive logical block numbers. To bound the length of a stored segment, if an input segment is longer than a threshold, $Usegment$, *Sungem* chops it into multiple input segments so that each of them is shorter than $U_{segment}$. Each input segment formed this way could contain zero, one, or multiple *stored segments*, which are segments that were previously seen and already stored in the repository. To identify the stored segments contained in an input segment, *Sungem* queries the SFI with every fingerprint in the input segment. If a SFI query results in a hit, the SFI returns one or multiple $\langle containerID, segmentID \rangle$ pairs, each of which corresponds to a potential stored segment that contains the query fingerprint.

After going through every fingerprint in an input segment, *Sungem* brings into memory all the containers indicated in the responses of the SFI hits. Each container contains a *per-container fingerprint index* that maps a fingerprint into an offset inside the container that holds information about the fingerprint's associated disk block, e.g., its physical block number. In addition, each container contains a *segment index*, which maps a fingerprint's offset inside a container to all the stored segments in which the fingerprint participates. *Sungem* keeps containers in an on-disk container store, and uses an in-memory container cache to keep containers that were recently fetched from the disk so as to reduce the disk access cost associated with fetching containers.

For each hit fingerprint, *Sungem* fetches the associated container from disk, consults with the container's fingerprint index to obtain the offset location of the hit fingerprint, uses

this offset information to identify all stored segments in this container that include the hit fingerprint, and finally performs fingerprint-by-fingerprint comparison between each such stored segment and the input segment, anchored at the hit fingerprint. After this process, each fingerprint in the input segment either matches some fingerprint in some stored fingerprint, or does not match any previously seen fingerprint. Each maximal-length sequence of fingerprints that matches some stored fingerprint form a *matched subsegment*, which could be identical to, a subset of or a superset of an existing stored segment. Every matched subsegment that is not identical to any existing stored segment forms a new stored segment, and *Sungem* places it in the same container as the *first* fingerprint of this subsegment. Note that a new stored segment derived from a matched subsegment could match partially multiple existent stored segments, but *Sungem* places it in the same container as the first matched stored segment. When a new stored segment is a superset of an existing stored segment, *Sungem* is putting related stored segments in the same container. The fingerprints in the input segment that do not match any stored fingerprint form a new stored segment, and *Sungem* places it in the default container, which holds newly discovered stored segments from an input backup stream that are not related to any existing stored segments. If none of the fingerprints in an input segment hits in the SFI, the entire input segment forms a new stored segment and is stored in the default container. As an input backup stream's default container grows in size and becomes X% full, where X is the *fill-up threshold*, *Sungem* allocates and switches to a brand new default container, so that the old default container can hold future stored segments that share common fingerprints with its stored segments.

In previous designs [20, 14], stored segments are assigned to the same container because they appear temporally close to each other. In *Sungem*, stored segments are assigned to the same container either because they appear temporally close to each other or because they share common fingerprints. The rationale of putting fingerprint-sharing stored segments in the same container is to bring in as many potentially matching stored segments in one container access as possible. For example, a popular image may be used in three different PowerPoint slides, each of which in turn may be included in many PowerPoint files. *Sungem*'s segment-to-container assignment scheme enables the fingerprint sequences associated with the three PowerPoint slides to form stored segments that are put into the same container. Whenever a fingerprint corresponding to a block in the popular image appears in an input backup stream, *Sungem* could bring in the stored segments corresponding to the three PowerPoint slides by fetching a single container.

To avoid proliferation of stored segments, each stored fingerprint is allowed to participate in up to K stored segments. More precisely, for each fingerprint, *Sungem* only records the most recently appearing stored segments in which the fingerprint is in, because these segments are more likely to match future input segments.

3.3 Variable Fingerprint Sampling

When a new stored segment is formed, *Sungem* uses a fixed sampling rate to pick representative fingerprints from the stored segment and inserts them into the SFI. Ideally, the sampling rate should be high enough to capture most data sharing units, but low enough to keep SFI's memory space efficiently utilized. *Sungem* employs a variable fingerprint sampling scheme to achieve the best of both worlds.

Fingerprints sampled from the same stored segment and inserted into the SFI are treated as a single entity and called a *fingerprint group*. The last access time of a fingerprint group is the last time any of its fingerprints matches a fingerprint in a new input segment. The fingerprint groups in the SFI are organized into a LRU list based on their last access time. When the SFI needs to evict fingerprints, it down-samples the fingerprint groups in the tail of the LRU list to 1 per group, and keeps on doing this until it exhausts all fingerprint groups above a certain age; after that it starts removing entire fingerprint groups to free up space.

When a fingerprint group's fingerprints have matched fingerprints in more than a certain number of new input segments since its formation, it becomes a *stable* fingerprint group, and the associated stored segment is considered a well-defined deduplication target. *Sungem* reduces the number of representative samples in a stable fingerprint group to one per group, because one fingerprint sample is sufficient to capture future instances of the associated stored segment.

The above two optimizations allows *Sungem* to apply a high sampling rate to new stored segments while effectively reclaiming memory space from older stored segments. If a stored segment later proves itself to be a useful deduplication target, *Sungem* reduces its sampling rate to one per segment. If a stored segment turns out to be a not-so-useful deduplication target, *Sungem* also reduces its sampling rate to one per segment.

3.4 Parallel Data Deduplication

The sequential version of *Sungem* processes each input segment using the following steps:

1. Looking up every fingerprint in the SFI,
2. Fetching into memory all containers referenced by all the hit responses from the SFI,
3. Performing fingerprint-by-fingerprint comparison between the input segment and all relevant stored segments in the fetched containers, and
4. Identifying new stored segments, putting them into proper containers, and inserting their sampled fingerprints into the SFI.

A simple strategy for K-way parallel data deduplication is to partition the fingerprint space into K portions, for example using a modulo K function, and assign each fingerprint in an input segment to one of the K nodes using the same partitioning function, where each node runs the sequential version of *Sungem*. This strategy is fully parallelized and relatively simple to implement, but it has a major flaw: the fingerprints associated with each data sharing unit are likely to form K stored segments. This means the total number of stored segments is increased by a factor of K, and more seriously the number of container-related disk I/Os required by processing of an input segment is also increased by a factor of K. Because the most likely bottleneck of a data deduplication engine is disk accesses associated with container fetches, a parallelization strategy that significantly increases the number of required disk I/Os is unacceptable.

To overcome the limitation of the simple parallelization strategy above, *Sungem* uses the following parallelization

strategy, which requires a master node and K slave nodes. Given an input segment, the master node broadcasts it to all K slave nodes, each of which looks up every fingerprint *in its share* (e.g. using a modulo K partitioning function) in its local SFI, and returns to the master node those fingerprints that hit in its SFI and the associated hit responses. The master node again broadcasts the accumulated hit responses to all K slave nodes, each of which then fetches containers that are referenced in the hit responses and stored in its local disks, performs fingerprint-by-fingerprint comparison, and returns to the master node the hit/miss status of the input fingerprints it touches. The master completes stored segment processing, and broadcasts new stored segments and their associated containers.

In *Sungem*'s parallelization strategy, fingerprints are processed in a partitioned fashion at Steps 1 and 2. But processing at Steps 3, 4 and 5 are data-driven, i.e., whichever nodes hold the needed stored segment perform the associated computation. This strategy forms a single stored segment for each data sharing unit and stores it in one of slave nodes. Although this parallelization strategy uses the same number of container-related disk I/Os for each input segment as the sequential version, it incurs additional inter-node communications cost, and may result in potential load imbalance.

4. SCALABLE GARBAGE COLLECTION

4.1 Comparative Analysis

In a data backup system that supports data deduplication, a physical block may be referenced by multiple backup snapshots. Because a backup snapshot typically has a finite retention period, the number of references to a physical block varies over time. When a physical block is no longer referenced by any backup snapshot, it should be reclaimed and reused. There are two general approaches to identifying physical blocks in a data backup system that are no longer needed. The first approach is *global mark and sweep*, which logically or physically freezes all active backup snapshot representations, scans each of them, marks those physical blocks that are referenced by these snapshots, and finally singles out those physical blocks that are not marked as garbage blocks. The second approach is *local metadata bookkeeping*, which maintains certain metadata with each physical block and locally updates a physical block's metadata whenever it is referenced by a new backup snapshot or de-referenced by an expired backup snapshot. The first approach does not incur any run-time performance overhead but may require an extended pause time, which is proportional to the storage system size, and thus is not appropriate for petabyte-scale data backup systems. But, most commercial products adopt modified mark and sweep approaches that minimize the pause time to a great extent and seems to be reasonably effective. However it is still batch-oriented rather than incremental as in the case of our algorithm. Consequently, *Sungem* takes the second approach, which incurs run-time performance overhead due to metadata bookkeeping. How to minimize this metadata bookkeeping overhead is an important design consideration of *Sungem*'s garbage collection algorithm.

To simplify the following discussion, let's assume every backup snapshot is represented by a logical-to-physical (L2P) map, which maps logical block numbers in a backup snapshot to their corresponding physical block numbers. In addition, the garbage collector maintains a *physical block array* (P-array) that maintains metadata for each physical block in the backup system. To facilitate the comparison among different garbage collection algorithms in terms of their performance overheads, let's use a reference data backup system with the following configuration and assumptions. The system has 1PB worth of physical blocks, and consists of 16 machines that collectively support an aggregate disk throughput of 1GB/sec for sequential accesses, and four 32TB disk volumes, each of which is full. Assume one backup is taken for each disk volume every day and each backup snapshot is kept for 32 days and then discarded. Therefore, at any point in time, there are in total 128 backup snapshots in the system. The block size is 8KB . Each entry in a L2P map costs 32 bytes. The percentage of change between consecutive backup snapshots of a disk volume is 5% of the volume's size. Accessing a 8KB block from disk takes 5 msec on average, and every 8KB block fetched on average services 64 accesses to that block before being evicted. The last two assumptions are somewhat arbitrary, but we made them only to quantitatively compare the *relative* performance of the garbage collection algorithms we consider below.

The most expensive step of *global mark-and-sweep* is the mark step, scanning the L2P maps of all active backup snapshots and marking those physical blocks that are referenced. For the reference backup system, one needs to scan $128 * \frac{32TB}{8KB} * 32 = 16$TB and accesses the P-array 512 billion times in a largely random fashion. Assume each P-array entry keeps a 1-bit flag, the P-array would be 16GB in size. If the 16GB P-array fits into memory, then the first step of *mark-and-sweep* takes around $\frac{16TB}{1GB/sec} = 16000$ seconds or 4.4 hours. If the 16GB P-array does not fit into memory, then the first step will take hundreds of hours.

The simplest example of the local metadata bookkeeping approach is *reference counting*, which maintains a reference count for each physical block to record the number of backup snapshots that point to it. When a backup snapshot of a disk volume is taken, the reference count of every physical block the snapshot references is incremented. When a backup snapshot of a disk volume is retired, the reference count of every physical block the snapshot references is decremented. When a physical block's reference count reaches 0, it is collected and put in the free pool. Assuming each P-array entry keeps a 2-byte reference count, the P-array needs 256GB. Because the mapping between logical blocks and physical blocks is largely random, accesses to the P-array do not have much spatial locality. In the reference system, 4 new backup snapshots are created and 4 old backup snapshots are retired every day. Therefore the total amount of time required to create and retire these snapshots at the end of each day is $4 * 2 * \frac{32TB}{8KB} * \frac{1}{64} * 0.005s * \frac{1}{16} = 44$ hours, where 4 refers to 4 disk volumes, 2 refers to the newly created and expired snapshots of each volume, $\frac{1}{64}$ refers to the assumed degree of reuse for every block fetched, and $\frac{1}{16}$ refers to the fact that 16 machines are used concurrently to support this operation. Obviously 44 hours is prohibitively long, and is even longer than the inter-backup interval, which is one day.

Because the retention period of a disk volume is known beforehand, it is possible to determine the last moment at which a backup snapshot continues to reference a physical block at the time when the backup snapshot is created. Suppose a backup snapshot is created at time T and its reten-

GC Algorithms	Backup Time Operations	Garbage Collection Time Operation	Elapsed Time
Mark-and-Sweep	None	Scanning per-snapshot L2P maps	Backup: 0; GC: 4.4 hours
Counter-Based	RC update of every logical block in new and expired L2P maps	None	Backup: 44 hours; GC: 0
Expiration Time-Based	ET update of every logical block in new L2P maps	Scanning physical block array	Backup: 22 hours; GC: 256 seconds
Hybrid	RC and ET update of every modified logical blocks in new L2P maps	Scanning recycle list	Backup: 1.1 hour; GC: 19.2 seconds

Table 1: *Comparison of the four garbage collection algorithms for a reference backup system whose detailed configuration is described in the text.*

tion period is R, then this snapshot will not reference any of the physical blocks it references after $T + R$. Assume we maintain an expiration time for every physical block, which indicates the time after which the block can be freed. When a backup snapshot of a disk volume is taken, the expiration time of every physical block the backup snapshot references is set to the larger of the current expiration time and the current time plus the snapshot's retention period. With this arrangement, no additional actions need to be taken when a backup snapshot of a disk volume is retired. To reclaim garbage blocks, one scans the P-array, each entry of which in this case maintains a 2-byte expiration time, and those physical blocks whose expiration time is less than the current time are garbage blocks.

One key advantage of the expiration time-based scheme over the reference count-based scheme is that no actions need to be taken at the time when a backup snapshot is retired. Therefore, for the reference backup system, the total amount of time required to create and retire backup snapshots at the end of each day is $4 * \frac{32TB}{8KB} * \frac{1}{64} * 0.005s * \frac{1}{16} = 22$ hours. Sequentially scanning the P-array, which costs 256GB, to reclaim garbage blocks in this case takes about 256 seconds.

The main weakness with the *reference count*-based and *expiration time*-based garbage collection scheme is that their performance overhead at backup time is proportional to the full size of the disk volume being backed up, rather than the size of the backup snapshot, which corresponds to changes to the disk volume and is much smaller if incremental backups are taken. We propose a *hybrid* garbage collection algorithm specifically for incremental backup systems. It maintains both a reference count and an expiration time for each physical block, and its performance overhead at backup time is proportional to the size of an incremental backup snapshot rather than the snapshot's underlying disk volume.

An incremental backup snapshot consists of a set of entries each of which corresponds to a logical block that has been modified since the last backup. Each incremental backup snapshot entry thus consists of a logical block number (LBN), a before image physical block number (BPBN) that points to the physical block to which the logical block LBN was mapped in the last backup, and a current image physical block number (CPBN) that points to the physical block to which the logical block LBN is currently mapped. At backup time, given an entry ⟨LBN, BPBN, CPBN⟩, the reference count of BPBN is decremented, the reference count of CPBN is incremented, and the expiration time of BPBN is set to the maximum of its current value and the current time plus the retention period of the disk volume being backed up. With this design, when a If the reference count of BPBN reaches zero, the physical block BPBN together with its expected expiration time is put into a recycle list. At garbage collection time, the physical blocks in the recycle list are scanned and those whose expiration time is less than the current time are garbage blocks.

In general, a disk volume has a current image and multiple backup snapshots. The reference count of a physical block in this algorithm keeps track of the number of current images, but not their associated backup snapshots, that currently point to it. The expiration time of a physical block records the time after which no backup snapshot will reference it. If there is at least one current image pointing to a physical block, this physical block cannot be a garbage block and its expiration time could be ignored. Whenever a logical block in a current image is modified, the current image no longer points to the physical block associated with the logical block before the modification, and the expiration time of this before-image physical block is updated to incorporate the retention time requirement of the current image's associated disk volume.

The main advantage of the proposed hybrid garbage collection algorithm is the number of P-array entries that it needs to modify is proportional to the number of modified blocks in a input snapshot. Therefore, for the reference backup system, the total amount of time required to create and retire backup snapshots at the end of each day is $0.05 * 4 * \frac{32TB}{8KB} * \frac{1}{64} * 0.005s * \frac{1}{16} = 1.1$ hour, where 0.05 is the average percentage of blocks in a disk volume that are modified in each day. Each P-array entry costs 3 bytes, 2 bytes for expiration time and 1 byte for reference count; so the P-array costs 384GB in total. The recycle list is at most 5% of the total number of physical blocks. So sequentially scanning the recycle list to reclaim garbage blocks takes about $\frac{384GB*0.05}{1GB/sec} = 19.2$ seconds.

Table 1 shows a detailed comparison among the four garbage collection algorithms discussed in this section. Batched garbage collection algorithms such as *mark and sweep* run periodically, require system pause, touch a fixed amount of metadata in each activation that is independent of their period, and incur largely sequential disk accesses if the P-array is memory resident. Incremental garbage collection algorithms such as *reference count* and *expiration time*, run incrementally, do not require system pause, touch an amount of metadata within a time interval that grows with the interval's length, and incur largely random disk accesses. However, because the total amount of metadata that the proposed hybrid garbage collection algorithm needs to touch is proportional to the amount of block-level change, it can be shown that its total metadata update overhead is no worse than any known *mark and sweep* variants. The proposed hybrid garbage collection algorithm is thus the first known garbage collection algorithm that is both incremental, in terms of

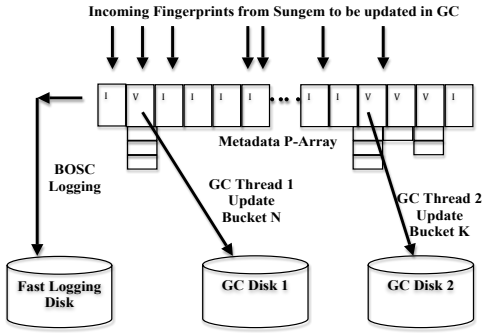

Figure 2: *Figure indicating metadata updates in garbage collection at backup time.*

not requiring system pause, and minimal, in terms of total metadata update overhead.

4.2 Batched Updates of P-Array

Even though the theoretical overhead of the hybrid garbage collection algorithm is the smallest, it still carries two major implementation challenges. First, its accesses to the P-array, which is too large to fit into memory, are largely random and therefore could incur significant disk I/O overhead. Second, after a physical block is chosen to be recycled, the physical block's fingerprint needs to be removed from the rest of the deduplication engine, including the SFI, and the container holding the fingerprint. This fingerprint-removing overhead is a significant part of the garbage collection process regardless of the actual algorithm used to determine which physical blocks are recyclable.

To address the first problem, *Sungem* adopts the BOSC (Batched Operation and Sequential Commit) mechanism [19] to modify the on-disk P-array. More specifically, *Sungem* partitions the P-array into a set of chunks, and allocates a per-chunk queue for each such chunk. Each update to the P-array is put into the per-chunk queue associated with the P-array entry to be updated. Multiple background threads are used to sequentially scan the on-disk P-array, by fetching to memory each chunk whose per-chunk queue is non-empty, committing all updates in the chunk's per-chunk queue to the chunk, and writing the chunk back to disk. Using BOSC, *Sungem* requires mostly sequential disk accesses to update the P-array. Figure 2 gives an overview of the garbage collection setup.

To address the second problem, *Sungem* uses a lazy update approach to removing a recycled block's fingerprint from the deduplication engine. If the container holding the recycled block's fingerprint is memory-resident, *Sungem* deletes it immediately; otherwise *Sungem* queues the fingerprint to be deleted in the container's per-container queue and actually deletes it only when the container is fetched into memory. *Sungem* deletes the recycled block's fingerprint from the SFI immediately, because the SFI is always memory-resident. When a chunk of the P-array is brought in, the garbage collector scans the entries in the chunk to identify those whose reference count is zero and whose expiration time has expired, and puts them in the free list. This mechanism piggy-backs garbage collection with P-array accesses and thus reduces the garbage collection overhead to the minimum.

File Type	Conjectured Append Behavior	Contribution Percentage (%)
VM-related Files	Append	35.1
Multimedia Files	Overwrite	21.6
System Files	Overwrite	21.9
User Documents	Overwrite	11.5
Installation Media	Overwrite	5.1
Log Files	Append	1.6
Mails	Overwrite	1.6
Database Files	Overwrite	1.6

Table 2: *The set of file types appearing in the collected trace, their conjectured append behaviors, and their contribution percentages in terms of size.*

5. PERFORMANCE EVALUATION

5.1 Evaluation Methodology

We have completed a Java implementation of the first *Sungem* prototype, which is incorporated in a commercial product called ITRI Cloud OS. To evaluate the effectiveness of the design decisions and the implementation efficiency of this prototype, we collected a real-world backup trace from a production environment, and used it in a trace-driven evaluation study.

Two evaluation metrics were used. The first metric is the data deduplication ratio, which is expressed as the percentage of duplicate fingerprints over the input fingerprints before deduplication. The second metric is the data deduplication throughput, which is measured in terms of the number of fingerprints that can be processed per second, including the overheads of both deduplication and garbage collection.

5.1.1 Trace Collection and Analysis

To derive a real-world trace of incremental data backup streams, we wrote a user-level tool that tracks and records changed files in a file system on a Windows machine within a period of time, and deployed this tool on 23 desktop machines of a research laboratory to collect the set of changed files every day on each machine for 10 weeks. Each of these 23 machines was used predominantly by a single user. We will refer to the resulting changed file trace as ***Workgroup*** in the following discussion. At the beginning of the trace collection period, the user-level tool traversed a file system, and for each file recorded into a database its last modify time and a 64-byte MD5 fingerprint for every 4KB block in it. At the end of every day, this tool traversed the file system, and compared the current modification time of each traversed file with its previously recorded modification time if it existed. If the previous modification time of a traversed file did not exist, the file was newly created. If the current and previous modification times of a traversed file were different, the file had been modified. In either case, the tool further computed a 64-byte MD5 fingerprint for every 4KB block in that file, and recorded into database its last modification time and all fingerprints computed this way.

Our tool tracks the file-level changes between consecutive versions of a file by comparing their constituent fingerprint sequences. It assumes that when a file is modified, the entire file is over-written. However, in some cases, modifications to a file could be in the form of appends. Because we had no way of knowing how applications actually modified files, we

relied on the file type information to infer whether a file was overwritten or appended at the block level when its file-level change indicated an append-like pattern.

Table 2 shows the list of file types appearing in the raw collected trace, their contribution percentage in terms of size, and our conjectures of whether they were overwritten or appended at the block level. VM-related files include files that support virtual machines, e.g., vmdk, vmem and vdi files. Multimedia files include all audio and video files. System files include files in the system directory, including Windows and "Program Files" directories. User documents include Microsoft Office files, pictures, and development files. Installation media refer to those files that are meant to install programs, e.g., iso and msi files. Log files include system log files and application log files. Mails include files that are updated by Microsoft outlook, including files with the suffix pst and ost. Database files cover all database files used by either applications or the operating system. Eventually we decided to remove VM image-related files because we believe these files are relatively rare in a typical office environment. After all necessary pre-processing, we produced a trace of daily fingerprint streams, which was initially 10.8GB in size, and eventually grew to 43.7GB at the end of the trace collection period. From the collected trace, the size of the average daily increment change is about 1.5% of all the files on these 23 machines taken together. However, in this trace when a file is modified, the delta consists of not only the changed data but the entire file data. Hence the deduplication ratio is much higher than a typical block level incremental snapshot. Lets call this trace as $userTrace$.

To measure some precise characteristics of *Sungem* we use another trace which is created based on the idea from the paper [13] as follows: Create a vm image called $vmImageBase$ using vmplayer and use windows 7 as the guest OS. To $vmImageBase$ add language packs and some huge windows applications to generate another image $vmImageModified$. The measured deduplication ratio in these traces are 20% in each trace. But when they are passed as input one after the other, deduplication ratio is around 34%. Meaning, around 14% of duplicates were found in modified vm image with respect to original vm image. But with this trace, the amount of data available is too short to test various features of a deduplication engine. Hence we use this trace sparingly only when the amount of input data doesn't influence the analysis. Lets call this trace as $VmTrace$.

The hardware testbed used in the evaluation of the *Sungem* prototype consists of two quad-core 3.4GHz Intel Core i-7 processor, 14GB of RAM, five 7200-RPM WD Caviar blue hard disks of 1TB each, one for the system disk, two for storing the GC metadata P-array on a striped software RAID with a 64-Kbyte stripe unit size, and the remaining two for storing the deduplication fingerprints on a striped software RAID.

5.2 Overall Performance

We fed into the *Sungem* prototype with a backup trace spanning 6 days that consist more than 2 billion fingerprints, measured the deduplication throughputs and ratios, and observed that *Sungem* was able to consistently deliver throughput above 200K fingerprints/second while maintaining a very high deduplication throughput of 90% as shown in Figure 3. The throughput of *Sungem* is usually around 250K fingerprints/second but the average comes down because of

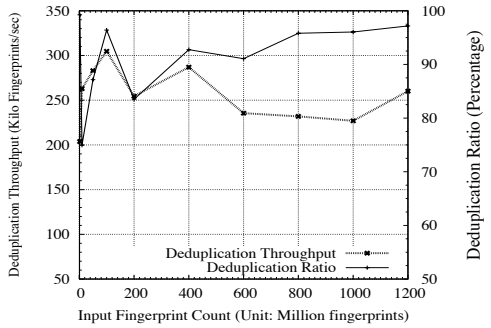

Figure 3: *The deduplication throughput and deduplication ratio of the* Sungem *prototype over an input trace of 2 billion fingerprints*

occasional long pauses caused by Java garbage collector. Java garbage collector maintains separate heap pools for short lived objects and long lived objects, which are called young generation and old generation, respectively. Applying the "Young Objects Die Young" assumption, Java garbage collector attempts to reclaim free memory in the young generation objects more frequently than that in the old generation ones. Efficient java programs tend to keep their objects short-lived. The *Sungem* prototype embraces this rule too, but at times when the container cache cannot capture the working set, more disk I/Os occur, making some objects long-lived and thereby pulling down the overall deduplication throughput. The dips in the deduplication throughput curve in Figure 3 arise precisely because *Sungem*'s working set at that instant exceeds the container cache. When a fingerprint segment hits an existing fingerprint segment, *Sungem* actually needs to do more work because it needs to bring in a container and performs fingerprint-by-fingerprint comparisons. When a fingerprint segment does not match any existing fingerprint segment, its fingerprints are filtered out by SFI and all subsequent steps in *Sungem* are skipped. Therefore, when the deduplication ratio of an input trace at an instant is higher, its deduplication throughput at that instant should be slower because of additional work. However, the correlation between deduplication ratio and deduplication throughput is not particularly strong in Figure 3 for two reasons. First, pauses caused by Java garbage collector have a non-trivial impact on the deduplication throughput and can happen any time. Second, *Sungem* effectively cuts down the disk access cost of containers when incoming fingerprints hit in the SFI and thus reduces the total overhead in the fingerprint hit case.

We also compared *Sungem*'s deduplication ratio with that from the baseline deduplication implementation, which uses a simple hash table to detect duplicates for fingerprints in an input backup stream. In all cases, the absolute difference in deduplication ratio is around 5%. This result shows that *Sungem* did not sacrifice deduplication ratio so as to deliver high deduplication throughput.

If an input backup trace contains only a small number of distinct fingerprints, the throughput of the deduplication engine is naturally high because of high access locality for fingerprints. To demonstrate this is not the case for the input backup trace used in this study, for each input block to the deduplication engine, we either increment its reference count or increment the reference count of its duplicate stored

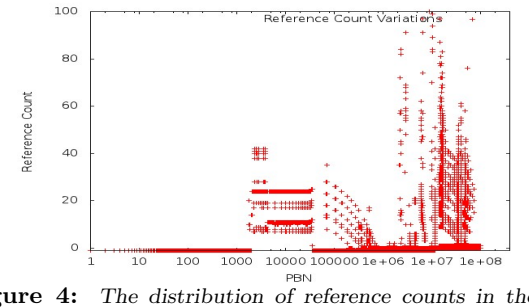

Figure 4: *The distribution of reference counts in the input trace. The X-axis is the physical block number (PBN) of each physical block in the backup system and The Y-axis shows how many times a physical block is referenced by different logical blocks.*

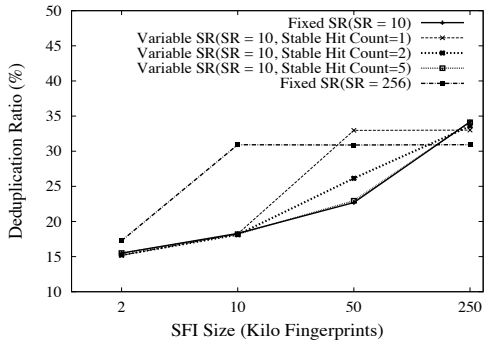

Figure 5: *Impact of the size of Sampled Fingerprint Index or SFI on the deduplication ratio*

copy. Figure 4 shows that the reference count of each physical block in the backup system after *Sungem* traverses the $userTrace$ trace is no more than 100 and that the number of unique fingerprints in the input trace is huge. Besides, we also measured the disk activity during the test run and found that the disk was heavily used at all times. The above two evidences prove that the high deduplication throughput of *Sungem* in Figure 3 is not because the working set of the input trace is small. As another measure, we tweaked the $userTrace$ to produce synthetic traces with deduplication ratios ranging from 0.08% to 95% and *Sungem* delivered a consistent high throughput above 200K fingerprints/second for all the cases.

5.3 Effectiveness of Sampled Fingerprint Index

With $vmTrace$, we varied the sampling rate (SR) and SFI size and measured the deduplication ratio for the following 5 configurations:

1. Fixed SR of 10, i.e., one out of every 10 fingerprints in each stored fingerprint segment is inserted into the SFI.
2. Fixed SR of 256 (each input segment has 256 fingerprints), meaning at all time only 1 fingerprint of each stored fingerprint segment is inserted into the SFI.
3. Variable SR configurations start with an SR of 10 and then switch to an SR of 256 when the number of hits reaches the stable hit count, which is set to 1, 2 or 5. Stable hit count corresponds to the number of hits a stored fingerprint segment needs to experience before it is considered stable and and every sample except one is removed from the SFI.

These five configurations correspond to five different ways of using the memory space allocated to the SFI. Figure 5 shows the deduplication ratios of the five configurations when run against the $vmTrace$ trace with varying SFI size. We have tried the same experimental set-up using the $userTrace$ trace, and the results are similar to those in $vmTrace$. When SFI size is 250K, fingerprints in the SFI are rarely replaced and the more space-consuming configurations produce better deduplication ratio results. For example, fixed SR of 10 is better than fixed SR of 256, and variable SR with a stable hit count of 10 is better than variable SR with a stable hit count of 1. On the other hand, when SFI size is 2K, fingerprints are replaced so frequently that the least space-consuming configuration wins out, i.e., fixed SR of 256.

When SFI size is 10K, fixed SR of 256 is better than all other configurations by a large margin, because the SFI space is too small to hold all the relevant fingerprints in other configurations. However, when SFI size is 50K, variable SR with a stable hit count of 1 actually produces higher deduplication ratio than fixed SR of 256, because the former makes the most efficient utilization of the SFI space.

Fixed SR of 256 works surprisingly well across all SFI sizes tested. This suggests that most stored fingerprint segments in *Sungem* that see repetitions could be successfully located when only one of their fingerprints is put into the SFI.

5.4 Content Proximity-Based Fingerprint Placement

Sungem strives to place fingerprint segments that share common fingerprints in the same container, and thus uses a *content proximity-based* (CP) approach to determine which container a stored fingerprint segment should be stored. In contrast, other deduplication systems [20, 14] use a temporal proximity-based (TP) approach in that they place stored fingerprint segments that are temporally close in their creation time into the same container. The TP approach to fingerprint segment placement is similar to a write-optimized file system, e.g. log-structured file system, because new stored fingerprint segments are simply appended to the default container until it becomes full. The CP approach to fingerprint segment placement is similar to a read-optimized file system, e.g., UFS, because it tries to place in the same container stored fingerprint segments that are likely to be referenced together when checking an input fingerprint segment against the fingerprint database. Therefore, we expect the TP approach to incur fewer disk write I/Os (for persisting containers) but more read disk I/Os (for determining if an input fingerprint hits the fingerprint database) than the CP approach. The *fill-up threshold* in the CP approach specifies the degree of fullness (in terms of percentage) of the default container before it is considered filled up. The residual capacity of a container in the CP approach is reserved for accommodating future stored fingerprint segments that share common fingerprints with those fingerprint segments already in the container. The TP approach actually corresponds to the CP approach with the fill-up threshold set to 100.

Table 4 shows a deduplication ratio and throughput comparison among variations of the CP scheme, each corresponding to a distinct fill-up threshold, when the input load is a 3-day trace consisting of 473 million fingerprints and the

Fill-up Threshold	Dedup Ratio	Dedup Throughput	Container Read Count	Container Write Count	Per-Segment Comparions
70%	93.11%	282.9K	1.238	0.0743	755
80%	93.17%	290.7K	1.248	0.0739	814
90%	93.14%	288.9K	1.259	0.0733	809
95%	93.16%	287.2K	1.267	0.0733	807
100%	93.26%	295.8K	1.264	0.0732	601

Table 4: *Deduplication ratio and throughput (fingerprint look-ups per sec) comparison among variations of the content proximity-based fingerprint segment placement approach, each corresponding to a distinct fill-up threshold, when the container cache size is 5000 containers. The TP approach corresponds to the CP approach with the fill-up threshold set to 100%.*

Commit Threads	*deduplication + vanilla GC*	*deduplication + BOSC-based GC*	*deduplication + without GC*
1	5879	54047	287204
2	6003	268218	287204
4	9858	277670	287204
10	8121	269272	287204

Table 3: *End-to-end throughputs(fingerprints processed/second) of a data deduplication engine with multiple garbage collector configurations.*

container cache size is 5000 containers. Surprisingly, the TP scheme beats all CP variants with different fill-up thresholds in terms of deduplication ratio and throughput. To understand why, we also measured the average number of distinct container reads and writes when processing a 256-fingerprint input fingerprint segment. As expected, the average number of container reads per input segment increases with the fill-up threshold, but the average number of container writes per input segment decreases with the fill-up threshold. However, the differences are too small to matter. Therefore, the deduplication throughputs across all fill-up thresholds are quite comparable.

Even though the CP variants with less than 100% fill-up thresholds offer the flexibility of clustering related stored segments, the TP scheme could provide the same clustering benefit if the consecutive temporal distance between instances of the same fingerprint sequence is smaller than the container size. However, in the input trace we used, the average temporal distance between consecutive instances of the same fingerprint sequence is actually much larger than the container size. Because the TP scheme tends to fill up a container before switching to a new container, it uses fewer containers and thus incurs a proportionally smaller number of container accesses. In addition, the average number of fingerprint comparisons per input segment required by the TP scheme is noticeably smaller than that required by the CP variants, as shown in the last column of Table 4, because the TP scheme is more capable of homing in to the matching stored segments without wasting unnecessary efforts exploring related candidate stored segments. Finally, the target scenarios that the CP scheme is optimized for, i.e., multiple existing stored segments that repeatedly appear together, simply is not very common. The above three reasons combined make the TP scheme the most performant under the input trace used in this study.

5.5 Garbage Collection Overhead

Effectiveness of our hybrid Garbage Collection scheme can be demonstrated by comparing it with a vanilla p-array update implementation, which buffers p-array update requests in a queue, and uses a background thread to commit them on a first come first serve basis. In Table 3, the throughput of the data deduplication engine without any p-array updates (the last column) sets an upper bound because it corresponds to a zero-cost p-array update scheme. When the BOSC-based p-array update scheme uses a single commit thread, the end-to-end throughput of the deduplication engine is decreased to 19% of the upper bound. By increasing the number of commit threads to 4 and therefore the disk I/O concurrency, it increases the end-to-end throughput to 97% of the upper bound. The number of commit threads represent a tradeoff between disk access locality and disk I/O concurrency. Empirically, the optimal number of commit threads for our experiment set-up seems to be 4. However, regardless of the number of commit threads used, the end-to-end throughput of the data deduplication engine using the vanilla p-array update scheme never exceeds 5% of the upper bound.

To directly assess the effectiveness of the proposed garbage collector, we measured the performance overhead of bookkeeping the reference count and expiration time data structures in the P-array for a series of daily incremental backup runs, each of which uses a delta list of fingerprints as input, and the results are shown in Table 5. The "No. of Records" column shows the number of records in the delta list of each day. The "No. of Entries" column shows the number of P-array entries that need to be modified. The "No. of Pages" column shows the number of P-array pages that are to be modified, where each page is 128KB. The "Bookkeeping Throughput" column is calculated by dividing the number of P-array pages to be modified by the bookkeeping time.

The fact that the number of P-array entries modified is indeed proportional to the number of records in the delta list shows that the proposed hybrid garbage collection algorithm is indeed scalable. The bookkeeping throughput decreases over time because there are more physical blocks in the data backup system as time goes by and the locality of the updates to the P-array become worse. However, the fact that the bookkeeping throughput remains relatively high demonstrates the effectiveness of the BOSC scheme in turning the random updates to the P-array into largely sequential disk reads and writes.

5.6 Effectiveness of Container Cache

Deduplication throughput is heavily influenced by the container cache size because large container cache could capture the working set of containers when processing input fingerprints and reduce the number of container-related disk I/Os. Figure 6 shows that for the given workload, the deduplica-

Day in Trace	No. of Records	No. of Entries	No. of Pages	Bookkeeping Time	Bookkeeping Throughput
1st	126 M	345 M	185282	71 s	334 MBps
2nd	345 M	917 M	799090	429 s	238 MBps
3rd	344 M	921 M	628042	1017 s	79 MBps
4th	317 M	852 M	597386	1089 s	71 MBps

Table 5: *The number of P-array entries modified as a result of the delta list of each day of the input backup trace and the associated bookkeeping time required to put these modifications to disk, when the number of bookkeeping threads is 10*

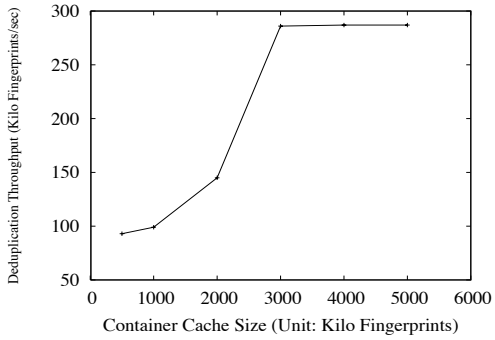

Figure 6: *Impact of the container cache size on* Sungem*'s deduplication throughput*

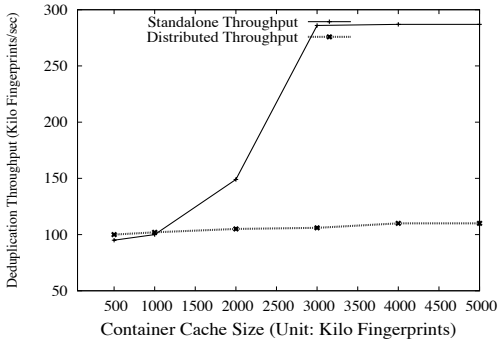

Figure 7: *Deduplication rate comparison between Parallel and Standalone Deduplication*

tion throughput is poor when the container cache size is below 3000, because the container cache cannot capture the active working set. However, as the container cache grows beyond 3000 containers, the deduplication throughput shoots up and stays flat even when the container cache grows to 5000. This suggests that for the given workload, a cache of 3000 containers is sufficient.

K-factor	*Throughput*	*Ratio*
1	233	90.90
2	282	92.68
4	287	93.20
5	280	93.21
10	20	93.26

Table 6: *Impact of K-factor variations on* Sungem*'s deduplication throughput and ratio. Throughput is measured in Kilo fingerprints/second and Ratio as a percentage of Duplicate fingerprints/Input Fingerprints. Settings include cache size=5000, fillup-threshold=95%*

5.7 Impact of Controlling Stored Segment Formation

As explained in subsection 3.2, the K-factor controls the number of segments in which a fingerprint can participate and acts as a tradeoff between deduplication throughput and ratio. Higher K value allows *Sungem* to identify the best stored segment of maximal length and thereby reduces any additional work required to deduplicate the remaining fingerprints. But at the same time, it also requires extra effort in going over all possible matching stored segments to find the best match. From Table 6, K value of 4 gives the best possible throughput and hardly sacrifices on the deduplication ratio.

5.8 Parallel Deduplication tradeoffs

Section 3.4 explains two different parallel versions of *Sungem* which performs better than the other depending on the type of input workload. To induce disk boundedness or cpu boundedness into *Sungem*, we will vary the container cache size and then show the difference in deduplicate rates of the parallel model against a standalone model.

Figure 7 shows how the deduplication rate of Standalone deduplication drops when container cache size is brought down. With *userTrace* workload, with cache size less than 500, the system is disk bound because the rate at which containers get evicted from cache is higher than the rate at which they can be stored on disk. When the system is disk bound, standalone deduplication is bottlenecked in loading and storing containers, thereby bringing down the throughput to below 100K fingerprints/second. Hence parallelizing using naive parallel design will not improve the performance as it will only cause more disk I/O's and deteriorate the performance. But the performance of parallel design with 2 worker nodes and 1 master node doesn't deteriorate because when one node is bottlenecked by disk I/O, other node continues to utilize the CPU efficiently. Both nodes are bottlenecked by disk I/O when cache size is seriously low but that's not the point we are trying to prove. We see that when cache size is more than 3000, the working set of containers are cached appropriately and hence standalone deduplication drives a high deduplication rate of more than 250K fingerprints/second. The experiment clearly demonstrates the effectiveness of different parallelizing strategies under different conditions. However due to practical reasons, we couldn't test this with a large scale setup consisting of hundreds of computing nodes and this is an important part of our future work.

6. CONCLUSION

State-of-the-art scalable deduplication techniques are not optimized for incremental backup operations. This paper describes the design, implementation and evaluation of *Sungem*, which is designed specifically to work efficiently with incre-

mental backup operations. In particular, we make the following contributions:

- A scalable deduplication engine that delivers consistent high throughput across all ranges of dedupe ratios and improves the deduplication throughput by up to 40% without sacrificing the deduplication ratio, when compared with the state-of-the-art sparse-indexing scheme [14] running with the same amount of RAM, for incremental backup operations,
- The first known garbage collection algorithm whose bookkeeping operations are distributed over individual backup operations and which is scalable in the sense that its bookkeeping overhead for each backup operation is proportional to the change to a disk volume between consecutive backups rather than the volume itself, and
- A comprehensive evaluation of the deduplication and garbage collection engine with a real-world daily backup trace spanning 10 weeks.

7. REFERENCES

[1] B. Asaro and H. Biggar. Data de-duplication and disk-to-disk backup systems: Technical and business considerations. *The Enterprise Strategy Group*, 2007.

[2] D. Bhagwat, K. Eshghi, D. D. E. Long, and M. Lillibridge. Extreme binning: Scalable, parallel deduplication for chunk-based file backup. In *MASCOTS'09: Proceedings of the the 17th IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, september 2009.

[3] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

[4] W. J. Bolosky, S. Corbin, D. Goebel, and J. R. Douceur. Single instance storage in windows 2000. In *WSS'00: Proceedings of the 4th conference on USENIX Windows Systems Symposium*, pages 2–2, 2000.

[5] A. Clements, I. Ahmad, M. Vilayannur, and J. Li. Decentralized deduplication in san cluster file systems. In *ATC'09: 2009 USENIX Annual Technical Conference*, pages 101–114, 2009.

[6] D. Colnet, P. Coucaud, and O. Zendra. Compiler support to customize the mark and sweep algorithm. In *ISMM '98: Proceedings of the 1st international symposium on Memory management*, pages 154–165, 1998.

[7] E. Corp. EMC Centera: Content Addressed Storage System. http://www.emc.com/collateral/hardware/data-sheet/c931-emc-centera-cas-ds.pdf, 2008.

[8] J. R. Douceur, A. Adya, W. J. Bolosky, D. Simon, and M. Theimer. Reclaiming space from duplicate files in a serverless distributed file system. In *ICDCS '02: Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS'02)*, page 617, 2002.

[9] C. Dubnicki, L. Gryz, L. Heldt, M. Kaczmarczyk, W. Kilian, P. Strzelczak, J. Szczepkowski, C. Ungureanu, and M. Welnicki. Hydrastor: a scalable secondary storage. In *FAST '09: Proceedings of the 7th conference on File and storage technologies*, pages 197–210, 2009.

[10] L. Dubois and R. Amatruda. IDC Backup and Recovery/Data Deduplication report. Technical Report, IDC and EMC, Feb 2010.

[11] G. Forman, K. Eshghi, and J. Suermondt. Efficient detection of large-scale redundancy in enterprise file systems. *ACM SIGOPS Operating Systems Review*, 43(1):84–91, 2009.

[12] F. Guo and T. Chiueh. DAFT: Disk Geometry-Aware File System Traversal. In *MASCOTS'09: Proceedings of the the 17th IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 56–68, 2009.

[13] F. Guo and P. Efstathopoulos. Building a high-performance deduplication system. In *USENIXATC'11: Proceedings of the 2011 USENIX Conference on USENIX annual technical conference*, 2011.

[14] M. Lillibridge, K. Eshghi, D. Bhagwat, V. Deolalikar, G. Trezise, and P. Camble. Sparse indexing: large scale, inline deduplication using sampling and locality. In *FAST '09: Proceedings of the 7th conference on File and storage technologies*, pages 111–123, 2009.

[15] D. Meister and A. Brinkmann. Multi-level comparison of data deduplication in a backup scenario. In *SYSTOR '09: Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, pages 1–12, 2009.

[16] P. Nath, B. Urgaonkar, and A. Sivasubramaniam. Evaluating the usefulness of content addressable storage for high-performance data intensive applications. In *HPDC '08: Proceedings of the 17th international symposium on High performance distributed computing*, pages 35–44, 2008.

[17] S. Quinlan and S. Dorward. Venti: A new approach to archival data storage. In *FAST '02: Proceedings of the 1st USENIX Conference on File and Storage Technologies*, page 7, 2002.

[18] S. Rhea, R. Cox, and A. Pesterev. Fast, inexpensive content-addressed storage in foundation. In *ATC'08: USENIX 2008 Annual Technical Conference on Annual Technical Conference*, pages 143–156, 2008.

[19] D. N. Simha, M. Lu, and T. Chiueh. An update-aware storage system for low-locality update-intensive workloads. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '12, pages 375–386, New York, NY, USA, 2012. ACM.

[20] B. Zhu, K. Li, and H. Patterson. Avoiding the disk bottleneck in the data domain deduplication file system. In *FAST'08: Proceedings of the 6th USENIX Conference on File and Storage Technologies*, pages 1–14, 2008.