RPE: The Art of Data Deduplication

***D*ilip Simha**
Advisor: Professor Tzi-cker Chiueh
Committee advisors: Professor Erez Zadok & Professor Donald Porter
Department of Computer Science, StonyBrook University

Sep 9 2011

Contents

Abstract

These days data storage is often talked about in units of tera and peta bytes. The demand for storing more and more data calls for technological innovations in multiple fronts. Some applications demand storing and processing huge amounts of data, while others need to do it faster and more securely. Hardware innovations have come a long way to handle these humungous data processing requirements. On the storage side, tapes have advanced to hard disks, solid state disks and flash drives. Processing speeds and physical memory sizes have increased. But having efficient software that make better utilization of the present and upcoming hardware will help overcome many barriers and scale to future demands, such as those spurred by the advent of cloud computing. In a cloud data center, applications and data from multiple tenants are all stored and managed in one place. If commonality among these data can be identified, unnecessary duplicates could be eliminated and storage hardware resources could be optimized. The process of identifying duplicate data and removing them is called deduplication. Both duplicate identification and removal are crucial in building large-scale very large-scale storage systems. To identify duplicates, every incoming data object is matched with already stored objects. To scale to very large data sets, efficient indexing techniques are required to speed up this matching process. With deduplication, a physical data object may be referenced by multiple data logical objects. How to efficiently reclaim a physical data object when all its associated logical data objects cease to exist is a challenge, because the metadata size is too large to fit into memory. In this report we will describe these design issues in deduplication, examine how proposed techniques tackled them, and finally present our own design – the deduplication engine used in ITRI CloudOS.

Chapter 1

Classification of Deduplication

1.1 Granularity of Deduplication

Deduplication can be done at different levels of granularity, File Level, Block level or Byte level. There is no clear rule that decides which approach is better than the other. It's a tradeoff which depends on the context where deduplication is applied and the operational costs.

File level deduplication uses entire file as the basic unit of deduplication. Files are compared against each other typically using cryptographic hashes and when they match, they are marked as duplicates and shared. Duplicate file is removed and that disk space is made free. Advantage with this approach is the simplicity in design but since it operates at the file level, it needs to manipulate the guest operating systems in their file system layer which makes it difficult to be ported to multiple operating systems. This also has an obvious disadvantage in systems where files are often modified rather than being shared without any modification. Even if there is a single byte change between any 2 files, they look completely different and separate copies have to be stored for them.

In extreme-bin [3], each file has a whole-file fingerprint, which means that a match of the whole-file fingerprint indicates that the whole file is a duplicate. Entries in the fingerprint index are distributed to K nodes one by one based on modular operation, or other distributed hash table functions. The whole container corresponding to a fingerprint index entry is distributed to the same node as the fingerprint index entry. If two fingerprint index entries happen to have the same container but distributed to two different nodes, the same containers are duplicated on two nodes. When a file is backed up, only the representative fingerprint is used to route the file to a node, all other fingerprints of the file are not used for routing purpose.

Block level deduplication uses blocks as the basic unit of deduplication. Block matching is similar to the approach used in file level deduplication except that it's done at block level. There are variations which consider single block or a group of blocks as the basic unit to be considered for lookup operation. Since this operates at block level, it only needs to modify the host operating system at block layer interface. Hence it can support any guest operating system without any modification.

This approach works very well in cases where file modifications are very common. When 2 files differ in only a few bytes and share most of the other data, only the blocks that have changed and are not seen before are stored as a new copy, rest of the blocks in the files are shared. Since the granularity of object to be compared is lower in block level deduplication, it has a higher deduplication rate than file level deduplication. But since there are more comparisons and more data has to be stored, the stress on systems resources(CPU and memory) are higher in block level deduplication.

To improve the lookup performance, group of blocks are aggregated and they form the basic unit of deduplication. So an interesting option is whether to consider fixed group or a variable size group as the basic unit of deduplication.

In fixed size group, forming a group is straightforward. But when a file is modified, having the same fixed size grouping strategy might not yield a good deduplication rate. If a few bytes are added at the beginning of the file, then a few more blocks are added, shifting the entire grouping pattern. Since the grouping strategy at the time of deduplication will have no information on this, we miss identifying all the duplicates. We will have a detailed look into this in later sections.

Byte level deduplication uses much lower granularity than block level, ensuring higher accuracy of finding duplicates. Byte level deduplication would fetch the best possible deduplication rate but the time and resources needed for deduplication are too high which often makes them not scalable.

1.2 Positioning of Deduplication

Positioning of deduplication can be classified as either at real time(on-line) or at the time of backup(off-line) or as Source Deduplication or Target Deduplication.

1.2.1 On-Line vs Off-Line Deduplication

In On-Line deduplication, duplicates are eliminated before data is saved on disk. Since this involves duplicate identification at run time, it needs to be extremely fast. Usual practice is to make cryptographic hashes of the objects and store them in an index. This index size is quite huge and doesn't fit in main memory. So there are several ideas to handle it, a few being: ChunkStash [4] suggests to use an SSD to store those elements that cannot fit in main memory index. Data Domain [29] stores the objects on disk but stores the sampled index(SFI) in memory and uses bloom filter to increase the lookup speed.

The advantage with on-line deduplication is that there is lesser stress on disk controllers as the data storage needs directly correlate with the observed deduplication efficiency. But since this happens at real time, it shares CPU and memory resources with user applications. It needs to be constantly in aware of system load to decide how aggressive the deduplication operations should be.

Off-Line deduplication works at the time of backup. So this works on removing already stored duplicate blocks on disk. Even though this doesn't have run time requirements like On-Line deduplication, it still needs to complete the backup process within a short span of time. Typically incremental backups are scheduled once a day and to backup huge amount of data running into several tera bytes, backup operation should process duplicates at some minimum threshold rate. This rate is calculated based on how much data needs to be backed up on an average and then how much time is available until next backup operation starts. Based on this rate CPU, disk and memory resources are suitably allocated.

Disadvantage is that data has to be first stored on disk even if it's a duplicate. Only later sometime, duplicates are freed. Even though the disk size is bounded eventually after deduplication, the disk storage capacity is not fixed to actual unique data on the system.

Although different in many aspects, out-of-line and online deduplication fit well in their corresponding arenas. For example, because the primary concern of a live storage system [9, 17, 6] is to provide low-latency access to live data and temporary space utilization is not the top concern, deferring the data strikes a reasonable balance between performance overhead and space utilization. On the the other hand, backup systems focus more on backup throughput than the latency of individual backup requests, and storage utilization is the first priority.

1.2.2 Source vs Target Deduplication

To guarantee data safety on system crash, typically data is backed up on a remote location. Data deduplication can be applied at either individual nodes, which is called source deduplication or at the remote node called target deduplication.

In source deduplication, duplicates are eliminated within each node and only the unique data is sent over the network to be stored at remote storage node. This saves the precious network bandwidth but utilizes more resources at source node often competing for resources with other client applications.

In target deduplication, entire data is sent to remote node where duplicates are identified and removed. This consumes more network bandwidth because redundant duplicate data is transferred over the network. The chances of finding duplicates are higher in target deduplication because multiple nodes backup their data over the remote node and hence there's a larger range of data available to match with.

A combination of both these approaches can be used, provided there is enough time and resource to allocate for deduplication operation. For example, NetApp's Snapvault uses such a combined approach.

1.3 Primary Memory vs Secondary Memory Deduplication

While secondary memory deduplication deals with data on disk, primary memory deduplication deals with data on RAM making operating system(OS) pages as the basic unit of deduplication. In virtual environments, multiple guest OS instances run on a host OS. If guest OS's are identical, then most of the OS related data stay unmodified for long time. Since host OS allots physical memory for guest OS's, common data pages on guest OS's can be shared, making redundant memory available in the free memory pool.

RedHat reported that using linux kernel same page merging or kernel shared memory (KSM), KVM can run as many as 52 Windows XP VM's with 1 GB of RAM each on a server with just 16GB of RAM. KVM scans parts of memory and calculates a hash of entire OS page. It then tries to match identical pages using these hash values. Since most of the OS libraries are unchanged, most of the OS memory can be shared. The duplicate pages are marked as free and shared page is write protected with a COW flag, so that any modification to that page, will result in a new page to the host which modified it.

Primary memory deduplication competes for CPU with user applications and hence it's usage should be moderated based on user application load. Hence not all pages are scanned for duplicates in OS kernel. For example on a linux kernel, only those marked with madvise system call are considered for scanning.

1.4 Network Deduplication

Network deduplication is about saving network bandwidth by transferring only unique data from source node to target node. The target node typically looks up it's index of stored objects to check if it already has a copy of the incoming object. Only unique objects can be transferred to the target node saving the previous network bandwidth. This can be visualized as an optimization over target deduplication. Instead of sending the entire data over the network, only their fingerprints can be sent. Target node can lookup it's index to find any matching duplicate and only unique data can then be transferred over the network.

Chapter 2

Critical Operations in Data Deduplication

2.1 Fingerprint Index Lookup

Input stream to deduplication process consists of billions of changed blocks every day. For each block, deduplication should tell whether it's a duplicate or not.

A naive approach of finding duplicates, considers every incoming object and examines the entire collection of already stored objects which could easily run into billions. It has to then answer the question "is the object seen before and if yes where is it stored". Since all of them cannot be kept in memory, most of them have to be stored on disk. Usual practice is to make cryptographic hashes of the objects and store them in an index. But even this index cannot fit into memory. Lets look at a typical example to see why. If we are dealing with 100 tera bytes of data and each block is 4KB. We have 25 Giga blocks and if hash size of each block is 20 bytes, we have 500 Giga bytes. If deduplication efficiency is 50%, we have 250 GB of data. And of late requirements have scaled up to peta bytes of storage and that only makes our scenario even worse. So clearly this index structure cannot fit in main memory. If it's stored on disk, then on an average even if we have to make 1 disk I/O for each lookup, the deduplication rate is bottlenecked by I/O speed. Since we calculate cryptographic hashes on each object, there is hardly any locality among neighboring objects. So OS level block caching or read ahead would not help much.

2.2 Negative Filtering

Bloom filter is a space efficient probabilistic data structure which tells if an element is NOT a member of a set with probability 1. Because of it's bit representation, it occupies very less space and hence can handle huge amount of data in a very short time. It's used extensively in index lookup process in deduplication to avoid costly disk lookups. However bloom filter has a couple of serious limitations. Firstly, when an item is deleted, it has to be removed from bloom filter too or else it will produce false positives. Secondly, when the number of items inserted to bloom filter grows large, most of the bits in bloom filter are set and hence most of the membership queries result in false positives.

2.2.1 Scalability of Filter

The main advantage of bloom filter is it's small size because of which it can fit completely in main memory and hence can provide fast lookup results. But when the amount of data that needs to be indexed grows too high in the scale of peta bytes, the size of bloom filter should grow accordingly. If the bloom filter size is not adequate, then most of the lookups will result in false positives making the filter redundant. But increasing the size of bloom filter will result in having some parts of the filter in memory and most part on disk. Since disk I/O is very slow, it can cause bottleneck to the lookup operation.

Cheng et al [7] proposes to use a counting bloom filter and use a timestamp in counter field. The timestamps decay over time and are replaced by new incoming elements into bloom filter. This approach makes the bloom filter scalable, but comes at a cost of increasing false positives. Having a decaying timestamp is like an LRU policy. On one hand it makes room for new incoming elements and avoids false positives but on the other hand it evicts stale copies and hence increases false positives. Jiansheng et al [26] proposes to use an hash bucket matrix structure with each row of the matrix having it's own bloom filter. All the bloom filters are further isomorphic meaning they share the same hash functions which enables parallel access to all the filters. It further uses a dual cache mechanism to make sure false positives from bloom filters do not cause any serious performance bottleneck with slow disk I/O's. Combination of these parallel accesses to bloom filters and dual cache mechanism facilitates a very fast deduplication operation. But the evaluation results show that for a data set of 10TB size, bloom filter requires around 10GB RAM. This clearly makes the system non scalable.

Jiansheng et al [27] further optimizes their previous (MAD2 [26]) approach by using a detached counting bloom filter scheme which uses a very small percentage of the filter in memory and still covers majority of the lookup operation without incurring any disk I/O. [21, 10] proposes to use a combination of RAM and flash storage to provide faster bloom filter accesses. The survey [25] gives an excellent overview of various bloom filter architectures.

Bender et al [2] propose an alternative architecture which provides very fast insertion, deletion and lookup capabilities. They use a combination of Cascade filter on Flash disk and a Quotient filter on RAM. Quotient filter is an approximate membership query data structure that's functionally similar to a bloom filter. It's built using hash tables and the hashing function is based on quotienting, a concept introduced by Knuth [18]. Its expected to use 20% more space than a traditional bloom filter which is much better than 4x space requirements of a counting bloom filter. Cascade filters comprises of a collection of quotient filters organized into a cache oblivious data structure that's specifically optimized for flash disks. The combined filter technique is reported to give a 40X speedup in insertion throughput with a downside of 3 fold slowdown in lookup throughput over traditional bloom filter.

2.2.2 Deletion in Filter

When blocks get removed from deduplication index, they have to be removed from bloom filter too. But typical bit structured bloom filter uses a single bit to record membership of multiple blocks. So when a block is removed from deduplication index, the corresponding bits in the associated bloom filter cannot be simply reversed to 0, as they can be shared by other elements and reset operations can introduce false negatives.

Li et al [12] proposes using counting bloom filter which uses a counter instead of just a bit representation. Counter value is incremented and decremented as and when blocks are added and deleted respectively. But having a counter instead of a single bit, significantly increases the size of bloom filter and hence makes it infeasible to fit in main memory. Jiansheng et al [26] proposes to rebuild the bloom filter upon deletion operation. To amortize the cost of rebuilding, an array of bloom filters are used and size of each bloom filter is kept sufficiently low.

2.3 Garbage collection

In a data backup system with data deduplication capability, 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 identify physical blocks in a data backup system that are no longer needed. The first approach is global mark and sweep, which freezes a data backup system, scans all of its active backup snapshot representations, marks those physical blocks that are referenced, 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 updates a physical blocks metadata whenever it is referenced by a new backup snapshot or dereferenced by an expired backup snapshot. The first approach requires an extended pause time and the second approach requires long scanning time both of which are typically proportional to a data backup systems size, and thus is not appropriate for petabyte-scale data backup systems. Symantec [16] highlights this problem and confirms that field report from Symantec corporation observed that reference management is indeed the major bottleneck to scalability.

2.4 Update Intensive Data Structure

When a logical block is modified, it is going to be backed up in the next backup run, and the information required to back up this logical block includes its logical block number(LBN), the physical block number (PBN) of the physical block previously mapped to this LBN (BPBN), and the PBN of the physical block currently mapped to this LBN (CPBN). The data backup system uses the logical block's payload to determine whether it is a duplicate of some existing physical block (whose PBN is DPBN) in the data backup system. Therefore, backing up a logical block triggers the updates of the GC-related metadata of up to three physical blocks. Assume a data backup system manages a 4-PB physical disk space with a 4-KB block size, then there are one billion physical blocks, and we need an array of one billion entries to record their GC-related metadata. Obviously this GC metadata array (GMA) cannot fit into the memory, and there is no reason to expect much locality in the accesses to this array. Hence the entire deduplication application bottlenecks in handling these huge number of random metadata updates if they are not handled intelligently.

Chapter 3

Deduplication Techniques

There are several critical components of a deduplication process and each of them can severely impact the performance of deduplication operation. Lets look at different techniques used to handle these components.

3.1 Timing of Backup

Snapshots are typically taken once per day, to backup critical user data. As storage requirements grow rapidly, the amount of data that needs to be processed each day grows very high. So along with maintaining a high deduplication ratio, getting a high deduplication throughput is also once of the most important tasks. But backing up an entire volume every day takes huge amount of time and brings down the throughput.

In case of offline secondary storage deduplication, input to deduplication process is a critical factor. In full backup, entire data is sent for backup application to be applied for deduplication. For the very first time, entire data is backed up. On subsequent backups, even though entire data is backed up, typically only a small percentage of data is changed. So it's natural for full backups to have a very high deduplication efficiency with a higher overall processing time.

To solve the low throughput problem, only changed data can be passed to deduplication and this approach is called incremental backup. Typically the changed data is recorded using timestamps on files or by using copy on write semantics at block level. Since the number of objects to be passed for deduplication is quite lesser than full backup, deduplication can finish it's task faster. But since the input to deduplication consists of no obvious redundant data, deduplication efficiency could be much lower than that in full backup.

Restoring the backup can be quite challenging in incremental backups. All the incremental backups have to be applied one after the other and these backups will result in random accesses to the individual blocks. However in a full backup, blocks can be updated sequentially resulting a faster recovery process. This is called Recovery time objective (RTO).

In order to achieve high backup throughput and also faster RTO, typically a combination of full-backup and incremental backups are applied. A weekly full backup followed by daily incremental backups can result in an overall better performance.

When data that's backed up is not too huge, having daily full backup can be beneficial. EMC Avamar white paper talks about advantages of having only full backups and incorporates it too.

3.2 Faster Index Lookup Strategies

Index lookup is a key operation which is very critical to the performance of deduplication process. This can be broadly classified into prefetching and sampling. Most of the techniques know today try to yield better lookup performance by either concentrating on prefetching or sampling and some use a combination of these techniques. Prefetching is a technique used to load a group of fingerprints before hand in anticipation that they would be used very soon in future. Different techniques vary in granularity of objects that are prefetched, decisions on what to prefetch and how much to prefetch. Sampling is a technique used to filter out redundant copies of data and store only most useful data that can either act as duplicates themselves or act as anchors to locations which contain possible duplicates.

Venti [23] pioneers the content-addressable storage (CAS) by computing the fingerprint (i.e., SHA1 hash value) of a data block and using the computed fingerprint instead of a logical block number to address the data block. Data blocks are deduplicated because data blocks with the same content have the same fingerprint. However, Venti does not focus on deduplicating data blocks, and more efforts are spent ensuring write-once-read-many property. In concrete, Venti does not address two performance problems associated with fingerprint-based deduplication. Firstly, the access locality is lost because adjacent data blocks have very different fingerprint values. Secondly, for a large-scale storage system, the fingerprint index can not fit into main memory [29, 20] and the lookup of the fingerprint index during data deduplication can incur extra disk I/Os, which can incur a significant performance overhead. Venti uses a combination of block cache, index cache and disk striping to improve the write throughput. Even after using 8 high end Cheetah SCSI disks for striping, the final write speed is 6.5 MB/Sec. The speed is so low because of the low cache hit ratio. This meant using caching to improve random disk I/O would only fetch limited throughput.

Zhu et al [29] proposed an online deduplication technique for a disk-to-disk (D2D) [1] data backup system. Because the fingerprint index can not fit into memory and resides on the disk, the paper proposes two techniques to minimize the overhead due to I/O access of the fingerprint index. Namely, the two techniques are (1) a bloom filter [5] based summary vector to avoid unnecessary fingerprint lookup, and (2) a locality-preserving data placement scheme to leverage the spatial locality of the input fingerprints. As the first optimization technique, the auxiliary bloom filter covers all fingerprint values of data blocks and accounts for a non-negligible amount of memory usage. On one side, a miss in the bloom filter indicates there is no such fingerprint value in the index structure. On the other hand, a hit in the bloom filter is not decisive in determining if the fingerprint value of interest exists in the fingerprint index, and a search to the index structure is inevitable. The second optimization technique preserves the locality by loading/evicting the fingerprints based on a container rather than a continuous range of fingerprint values because neighboring fingerprints do not reflect the data locality. The container corresponds to a continuous range of logical blocks from the input backup stream, and contains all metadata related to the continuous range, including the fingerprints and physical locations of these blocks. A query miss of one fingerprint in the container triggers the loading of all fingerprints in the container, predicting other fingerprints in the container will be queried in subsequent fingerprint queries. In most cases, the prediction is correct due to data locality in the input backup stream. This design helped achieve 100 MB/sec throughput and more importantly demonstrated that disk I/O is no longer the bottleneck in index lookup operation.

The sparse indexing scheme [20] uses a sampling fingerprint index instead of a whole fingerprint index to further reduce the memory usage of deduplication in an online deduplication system for 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. Among all matched ranges, a champion is chosen to deduplicate against. The sampling ratio can be used to trade-off the deduplication quality and the memory usage. In one extreme, if all fingerprint values in the range are sampled, the deduplication efficiency is at it's best. In the other extreme, if only one fingerprint value in the range is sampled, the deduplication algorithm can err in choosing the champion range and therefore the deduplication efficiency drops. Although the segment match based on sampled fingerprint works well for examined workload, it is not clear how effective it is for other backup workloads, including changed blocks within a file or an email, which our deduplication techniques focus on.

ChunkStash [4] observed that even though Data domain SY[29] idea made sure most of the lookup operations happen in memory, there is a small portion of them which cannot avoid random disk I/O. Since these random disk I/O's are so slow, improving upon them could significantly boost the overall throughput. So they propose using flash storage and flash storage aware algorithms and data structures to help achieve faster index lookup operation. Flash storage can perform faster than SCSI disks but slower than RAM. But random updates to flash are typically slow, so this paper provides a solution to overcome it by converting random updates into log structured append operations. The evaluations show that using flash disks they could get more than 200 MB/sec throughput. The high cost and low lifetime of a flash disk are the main factors which makes adopting this solution tricky.

Foundation [24] leverages commodity USB external hard drives to archive digital files in a similar fashion to Venti. Different from Venti but similar to the Data Domain scheme, a 16 MB segment is stored continuously to preserve locality for sequential read and fingerprint caching. For each fresh write, up to 3 disk accesses are encountered: (1) the lookup of fingerprint, (2) appending the data payload to the end of the data log, and (3) fingerprint updates to the on-disk fingerprint index. Bloom filter is employed to filter out unnecessary lookup in (1). For (3), in updating the fingerprint store, similar to BOSC, a buffer is employed to accommodate fingerprint updates and a single sequential scan is used to commit updates when the buffer is full, which shows the effectiveness of BOSC in deduplication. However, the fingerprint updates are not logged to the disk and their durability is not ensured, which can cause problem when the fingerprint update buffer is not filled for a long time.

The Extreme-Bin [3] approach exploits file similarity, which has been used in a deduplication approach targeting at non-traditional backup workloads that are composed of individual files with little or no locality. For each file, Extreme Bin chooses the smallest chunk hash as the representative fingerprint. Files sharing the same representative fingerprint are grouped into a bin, which is the basic scope of chunk level deduplication. Representative fingerprints of all the bins are organized into a primary index that is sufficiently small to remain in RAM, so that only one disk access to its corresponding bin is needed for each incoming file. Since Extreme Binning samples only one fingerprint for each file, the probability of similar files being grouped into the same bin is highly dependent on their similarity degree. According to Broder's Theorem, the probability that different files share the same representative fingerprint will decrease as the number of files grow.

Fanglu et al [16] suggests Progressive sampled indexing. Typically blocks are grouped together into segments and the fingerprint lookup index is sampled to make sure only important fingerprints are represented to cover maximum stored segments and also ensure that entire index fits in main memory. They suggest a sampling scheme based on amount of free memory available at the time of sampling rather than using the total main memory capacity. Such a progressive sampling scheme can dynamically change the sampling rate and yield better deduplication rate.

3.3 Medium of Backup Storage

Backup storage solutions use different mediums of storage. They can be either tape library, hard disk or SSD.

Data Domain [29] argues on why disk based solution is better than tape based ones. It's argued that sequential data transfer rates on tapes are better than that on disks. But the disadvantage of tapes is with managing them. It's manual and hence expensive and error prone. Critical restore operations can fail because of mishandling of tape cartridges. Moreover, random access operations are predominant in restore operations. So tape is not suitable as it's poor random access performance would seriously bottleneck the entire application.

Cost wise, even though disk costs are decreasing everyday, tape costs are decreasing too. But tape is preferred in cases where data needs to be backed up for a long time. Disks are believed to have a shorter life and hence it needs to be rotated which means higher maintenance cost. So even if disks are used primarily, data finally gets stored on tape in such cases. This process is often called Disk2Disk2Tape. This is typically seen in disaster recovery planning where it's ideal to store backed up data at remote locations in the form of removable media. So in most of those cases, data is copied from disk to tape and then archived in safe locations.

But research from Gartner[14] shows that tapes have a higher failure rate. Typically only 70% of the time tapes are accessed from archive and of them 10-50% of tape restores fail. But even though disks have a higher failure rate, they can be made more resilient using better technologies like RAID.

There is another interesting component called virtual tape library(VTL) which virtualizes disk storage as tape libraries allowing easier integration of VTL with existing backup software and recovery processes and policies. The benefits of such virtualization techniques include storage consolidation and faster data restore processes. TLFS [13] demonstrates the use of VTL in a file system and compares it with disk and tape storage solutions.

3.4 Variable Segment vs Fixed Segment Size

Input stream of blocks to deduplication can be grouped into chunks of fixed or variable size in order to maintain lesser metadata information in index lookups. Fixed size grouping is easier to implement but might miss some opportunities to deduplication. Even if there is a slight offset of blocks in the incoming stream to deduplication, fixed size segment will yield a totally new segment which might fail to find a duplicate. Variable size segments will have different anchor points to efficiently find the exact duplicate window in the input stream. But the cost of maintaining multiple anchor points can slow down the overall performance.

Most of the deduplication solutions[24, 29] use fixed size approach because of its simplicity. [28, 11] use variable-length chunks because they are effective for deduplication, but they cause internal fragmentation on the data-base. Bimodal Chunking [19] further optimizes variable length chunking. It uses smaller chunks in limited regions of transition from duplicate to non-duplicate data, and elsewhere it uses larger chunks.

Chapter 4

Garbage Collection Techniques

There are two general approaches to identify physical blocks in a data backup system that are no longer needed. Mark and Sweep approach and the other being local metadata bookkeeping which uses reference count or expiry time.

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* that maintains certain metadata for every physical block in the 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 reference system has 1PB worth of physical blocks with 8KB block size and supports four 32TB disk volumes, each of which is fully utilized. Assume one backup is taken for each disk volume every day and each backup snapshot is kept for 32 days and then discarded. Also assume that every block gets accessed 64 times before it's evicted. Even though this is an arbitrary estimate, we believe it's reasonable. Therefore, at any point in time, there are totally 128 backup snapshots in this system and each entry in a LogicalToPhysical (L2P) map costs 8 bytes to hold the physical block number. The percentage of change between consecutive backup snapshots of a disk volume is 5% of the volume's size.

GC Schemes	***Lookup cost***
Mark and Sweep	512 Billion
Ref Count	16 Billion
Expiry Time	8 Billion
Hybrid RC/ET	0.4 Billion

Table 4.1: *Comparison of the Lookup cost overheads for four garbage collection algorithms using a reference data backup system whose detailed configuration is described in the text.*

4.1 Mark and Sweep

The naive approach is called mark-and-sweep, which involves scanning through all per-snapshot mapping indices, marking those used physical blocks, and then scanning all physical blocks to reclaim those unmarked blocks. Since the system image is expected to not change at the time of mark phase, it's generally frozen and hence inaccessible to user for the duration of mark phase. The scan phase The expensive marking during the scanning makes the mark-and-sweep approach infeasible for the backup server.

For example, in the reference backup system, one needs to lookup $128 * \frac{32TB}{8KB} = 512$ Billion L2P map entries and that involves slow disk I/O, making snapshot management a very slow process. The physical block array (PBA) needs $\frac{1PB}{8KB} = 16T$ entries and if each entry needs a 1-bit flag to mark the presence of a block, the PBA occupies $\frac{16T}{8} = 2TBytes$. Clearly this doesn't fit in main memory and the mapping between logical blocks to physical blocks is largely random. Hence updating PBA also involves slow disk I/O which makes such a system not scalable. Another drawback with mark and sweep approach is the requirement to freeze the system during mark phase.

4.2 Read Only Mark and Sweep

Hydrastor [11] suggests a variation of Mark and Sweep which solves the problem of system freezing at the time of marking.Instead of freezing the system at mark phase, the system is made read only. Deletion is implemented with a per block reference counter which indicates how many blocks in the system point to that block. Incrementing the reference counter is delayed to avoid random update issue. They are instead held in per-block memory and later updated in read-only phase. That would convert random updates to sequential updates.

4.3 Grouped Mark and Sweep

Fanglu et al [16] propose group mark-and-sweep (GMS) mechanism, which is reliable, scalable, and fast. The key idea is to avoid touching every file in the mark phase and every container in the sweep phase. GMS achieves scalability because its workload becomes proportional to the changes instead of the capacity of the system.

By using a File Manager which organizes backups into multi-hierarchical levels, garbage collector can selectively mark only those group of files which have changed in the current snapshot. The sweep phase needs to scan only those containers which are used by a marked group. If the grouping is done efficiently, which they do based on backups which have file deletions only, then mark phase avoids scanning all the blocks on system and hence makes GC scalable. Those groups which are unchanged across backups, need not be considered for marking as the previous mark results can be reused. That further lessens the mark phase time.

They also suggest maintaining a reference list as a better solution. They claim it is immune to repeated updates and it can identify the les that use a particular segment. However, some kind of logging is still necessary to ensure correctness in the case of lost operations. More importantly, variable length reference lists need to be stored on disk for each segment. Every time a reference list is updated, the whole list (and possibly its adjacent reference listsdue to the lists variable length) must be rewritten. This greatly hurts the speed of reference management.

4.4 Reference Count based

The reference count (RC) based approach employs a per-physical block counter to avoid the time-consuming scanning in the mark-and-sweep approach. Each time a per-snapshot mapping entry is created, the counter of the corresponding physical block is incremented. At the time the snapshot is determined to be retired, for each physical block pointed by the per-snapshot mapping index, the associated counter is decremented. The decremented blocks are added to a D-change list of physical blocks. At the garbage collection time, the D-change list is scanned to determine which blocks to garbage collect. Because the D-changed list consists of physical blocks pertaining to reclaimed backup images, it is much smaller than the whole physical block space, and takes much less to scan.

Assuming each reference count array entry keeps a 2-byte reference count, then the number of lookups in the reference count array is $\frac{32TB}{8KB} * \frac{1}{64} * 128 * 2 = 16$ Billion, where a factor of 2 is multiplied because reference count of every block is updated both at creation and deletion times of a snapshot and we account for all 128 snapshots because we are comparing with mark and sweep approach which can be scheduled to run after aggregating multiple snapshot creation and deletion events. Although the number of lookups are much lesser than mark and sweep approach, updating 16 billion entries with random locality access will cause the system to bottleneck.

4.5 Expiry Time based

An expiry time (ET) based approach can further reduce the number of updates at the backup time by replacing the counter with an expiration time. But the incremental change list methodology cannot be used for expiration-time-based scheme because the expiration time of overwritten blocks with respect to the previous backup image need to be updated. In concrete, at the creation time of a backup image, the expiration time of each physical block in the entire volume is updated and nothing needs to be done when a snapshot is expired. At the garbage collection time, all physical block entries are scanned to find out those blocks whose expiration times have passed by. One key advantage of the expiration time-based scheme over the reference count-based scheme is that no action needs to be taken at the time when a backup snapshot is retired. Therefore, for the reference backup system, the total number of lookups in expiry time array required to create and retire backup snapshots at the end of each day is $\frac{32TB}{8KB} * \frac{1}{64} * 128 = 8$ Billion. A limitation of this scheme is that the retention period of a backup snapshot cannot be modified after the snapshot is taken.

Chapter 5

Our idea: *Sungem*

We now describe the design, implementation and evaluation of our data deduplication engine called *Sungem*.

5.1 Background

Before we describe the internals of *Sungem*, it's better we get a clear picture on how it's deployed on a typical client system. A typical data center consists of a dedicated set of systems called Compute Nodes (CN) where each node has several instances of OS images running on them. A set of dedicated systems are necessary to service these compute nodes and a few important of them being Distributed Main Storage Node (DMS), Distributed Secondary Storage Node (DSS) and Data Nodes (DN). Data Nodes are responsible for storing all the data. DMS is mainly responsible for maintaining metadata for each stored data and helps identify which data node holds a particular data. DSS is mainly responsible for backing up data and managing snapshot and deduplication. A DMS client which is a kernel module operates from Dom 0 on compute node. Figure 5.1 gives a clear picture of the entire system.

DSS triggers data backup operation with a snapshot create request which is typically scheduled with a daily incremental snapshot and a weekly full snapshot. DMS interacts with DMS client and has complete information of block level mappings of user data. So DMS maintains a delta list of changed blocks with respect to last snapshot operation. The delta list consists of Before Image PBN, Current Image PBN and LBN as discussed previously. DSS requests DMS for the delta list and then fetches the fingerprints for each block mapped in delta list. The delta list together with fingerprint values are then streamed to *Sungem* to find if duplicates exist or not. *Sungem* looks for possible duplicates and updates the reference count and expiry time in garbage collector which maintains metadata for each PBN on system. After deduplication, *Sungem* returns to DSS a subset of input delta list for which duplicates are found. DSS updates its data structures to maintain the snapshot

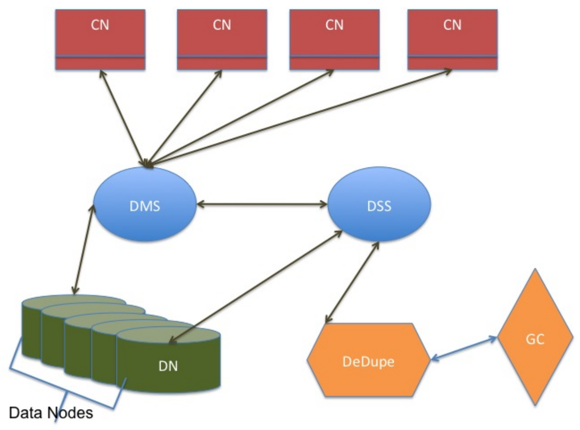

Figure 5.1: Overall view of data backup server

information for future snapshot restore and remove operations.

For wide area data backup(WADB), backup data has to be placed in a remote location to recover data losses due to catastrophic failures in entire data center. By having WADB frequently, high availability can also be guaranteed. But to achieve high availability, data copy between remote sites have to be extremely fast. Deduplication on target remote site can help avoid redundant data transfer as it can help reuse data that's already available on remote site. So WADB module in DSS can communicate with remote site to first apply deduplication and then transfer only unique data across the sites to create a backup of the required volume.

5.2 Overview and Assumptions

Sungem is designed specifically to work efficiently with incremental backup workloads and to minimize garbage collection overhead. *Sungem* features four novel techniques to improve the deduplication throughput without adversely affecting the deduplication ratio. Firstly, *Sungem* puts related fingerprint sequences into a container in order to better exploit the spatial locality in fingerprint lookups. Secondly, *Sungem* varies the sampling rates for fingerprint sequences with different stability in order to reduce the total memory requirements of fingerprint indexes. Thirdly, *Sungem* combines reference count and expiration time to arrive at the first known garbage collection algorithm whose bookkeeping overhead for each incremental backup snapshot is proportional to the backup snapshots size rather than the volumes size. Finally, *Sungem* uses BOSC methodology to handle random metadata updates, which eliminates disk I/O bottleneck in garbage collection. These techniques together could lead to up to 70% improvement in deduplication rate and up to 35% improvement in deduplication efficiency.

Sungem is targeted at systems which satisfy the following basic assumptions:

Assumption on File Sharing: Files are shared very often either between users or between duplicate copies. If not for entire file, parts of a file are shared. Example: a subset of slides in a powerpoint presentation, part of the source code from a repository, email attachments cced to multiple people, critical data backed up at multiple places. So either the files are shared as a whole or some physical blocks corresponding to the file are shared. This means these set of blocks tend to reappear again and again in the deduplication input stream.

Assumption on Physical Block Locality: Blocks from the same file tend to be contiguous on secondary storage in most cases because of intelligent OS decisions and defragmentations. So blocks occurring contiguous in physical address space can be assumed to belong to same file or contiguous files in a directory. Probability of blocks from neighboring files repeating in same pattern in future is very low because over time files get modified and their blocks are all moved together to different locations(defragmentation). Even this low probability event is handled appropriately in our design.

Assumption on GC-Related Metadata Updates: Recall on the metadata update issue in a typical data backup system where backing up a logical block triggers the updates of the GC-related metadata of up to three physical blocks: CPBN, BPBN and DPBN. Interestingly, these metadata updates are not totally random.

CPBN's are allocated from global free physical block pool, which at any given day spans at most dozens of Gigabytes. The majority of BPBN's should also come from the recent global free physical block pool, because of the "blocks tend to die young" heuristic. DPBN's in theory can come from anywhere in the past. However, according to our study, when a block has multiple duplicate occurrences, the temporal distance between the first occurrence and the last occurrence is in most cases less than a week. That means even the working set for DPBN's is bounded. Moreover, the input stream to deduplication is expected to consist of blocks belonging to same or neighboring files. So there are more chances that metadata updates to these blocks are from physical locations close to each other. Hence the disk I/O needed to operate on these metadata updates are not totally random. We further consolidate the GC design using BOSC like design which will be explained in detail in the GC section.

5.3 Bloom Filter Chain

As seen in section 2.2, several bloom filter techniques attempt to solve the scalability and deletion issue in different ways but none of them handle both of them efficiently. We propose a technique to chain the bloom filters(BFC) in such a way that it's easy to replace aging filters with an empty new filter. Our research indicates that most of the blocks that are identified as duplicates are created within last few days and hence aging blocks are least likely to be targets of future deduplication and hence it makes sense to dispose off corresponding aging fingerprints and its associated filters. Thus having a BFC and replacing aging filters with new empty filters solves the deletion problem. To solve the scalability issue, number of filters in the chain can be increased and the size of this BFC is limited by the number of new fingerprints observed over last few days. To handle faster processing of BFC, similar to [27] all the filters in BFC share the same set of hash functions and does parallel processing on them.

The false positive probability for a finite bloom filter with m bits, n elements, and k hash functions is at most $(1 - e^{\frac{-kn}{m}})^k$. Therefore, if we have Z bits, we can use it to implement either a single Bloom filter or X Bloom filters, and the false positive probability is the same for both implementations with same set of K hash functions and when same n elements are inserted.

5.4 Segment-based Deduplication

A fingerprint segment in *Sungem* is meant to capture the notion of a data sharing unit, e.g. a popular MPEG movie, PowerPoint presentation, WORD document or archived file. 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 each of which is shorter than Usegment. 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 possible stored segments contained in an input segment, *Sungem* checks every fingerprint in the input segment against the sampled fingerprint index (SFI). If a SFI lookup results in a hit, the SFI returns one or multiple container ID's, each of which contains a potential stored segment that contains the fingerprint associated with the SFI lookup. *Sungem* will bring in all the containers indicated in the containerID's returned by the SFI. For each container fetched, *Sungem* first consults with its per-container fingerprint index to obtain the offset location of every fingerprint that hits in the SFI and returns this containers ID, and then uses each such offset to find all stored segments in this container that include the offsets corresponding fingerprint. Through this process, *Sungem* identifies all stored segments that contain some fingerprints in the input segment. By comparing every such stored segment with the input segment around the hit fingerprint they have in common, *Sungem* identifies all fingerprints in the input segment that match at least one fingerprint in some fetched container. Those fingerprints in the input segment that do not match any fingerprint in the fetched containers form one or multiple new stored segments. If none of the fingerprints in an input segment hit in the SFI, the entire input segment forms a new stored segment. To avoid proliferation of stored segments, each fingerprint is allowed to participate in up to K stored segments. That is, for each stored fingerprint, *Sungem* only records the K most recent appearing stored segments that contain the fingerprint. Figure 5.2

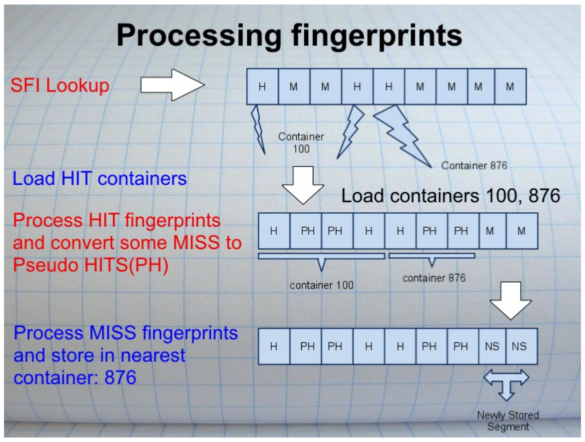

Figure 5.2: Fingerprint processing in *Sungem*

summarizes the fingerprint processing mechanism.

5.4.1 Sampling and Prefetching

To improve the effectiveness of fingerprint index (FI), entire FI has to be kept in memory but size of FI is typically huge and cannot fit in main memory completely. So FI has to be sampled which involves selecting important fingerprints and storing them in SFI. To further optimize SFI, when one of the fingerprints in SFI is referenced, it's neighboring fingerprints should also be brought into memory. This is based on the assumption that basic data sharing unit could span across multiple stored segments. Thus we store related fingerprints in the same container and entire container is fetched into memory when even a single fingerprint from the container is referenced.

5.4.2 Variable-Frequency Fingerprint Segment Sampling

When a new stored segment is formed, *Sungem* uses a fixed sampling frequency to pick representative fingerprints from the stored segment and stores only these sampled fingerprints in the SFI. To detect an exact duplicate of a stored segment in a future backup stream, keeping only one fingerprint in the SFI is sufficient. However, because a future backup stream may contain a fingerprint sequence that is actually a partial duplicate of a stored segment, keeping multiple sample fingerprints of every stored segment in the SFI increases the probability of successfully detecting such partial duplicates. The higher the sampling rate, the better the odds of detection, but also the more memory consumed. *Sungem* incorporates the following improvements to make even more effectively use of the memory space allocated to the SFI.

A stable stored segment is one that has been exactly matched by portions of multiple input segments after it was formed. Such a stored segment suggests that future matches to it will also be exact, and thus only one sample fingerprint is needed. Intuitively, a stable stored segment corresponds to a popular basic data sharing unit (BSU) that has appeared perhaps multiple times in multiple disk volumes. Algorithmically, a stored segment becomes stable after it has been exactly matched by at least TBSU input segments. Once a stored segment becomes stable, *Sungem* reduces the number of its sample fingerprint in the SFI to one, because one sample fingerprint is sufficient to capture all future duplicates of a stable stored segment. This reduction in the number of sample fingerprints for stable stored segments allows the SFI to hold more stored segments and thus boost the probability of deduplication. This idea is further generalized to reduce the number of representative fingerprints from a stored segment proportional to the number of times it's referenced. The SFI contains sample fingerprints from a subset of stored segments, which serve as anchors to bring in related containers, which contain more stored segments none of whose fingerprints are in the SFI. Therefore, it is still possible for a stored segment to match some input segment even though none of its fingerprints are kept in the SFI. When the SFI is full and new sample fingerprints are to be added, some existing fingerprints need to be evicted. *Sungem* maintains an LRU list to estimate the probability that each stored segment currently held in the SFI could match some future input segments. When the SFI is full, *Sungem* first reduces the sampling rate of those stored segments in the tail of the LRU list in an exponential fashion and eventually evict those whose number of sample fingerprints has already reached the minimum, one. Stable stored segments, by construction, tend to appear in the head of the LRU list and thus are unlikely to be evicted.

5.4.3 Clustering Related Segments into Containers

In *Sungem*, a container is meant to hold related segments. Two segments are related in *Sungem* either because they appear close to each other in an input fingerprint stream, or because they share common fingerprints. In contrast, previous data deduplication systems [21, 16] group segments into containers only based on their proximity in the input fingerprint streams. When *Sungem* starts processing an input fingerprint stream, it allocates an empty container as the streams default container, which is meant to hold input segments that do not match any stored fingerprints. Once the default container is half full, *Sungem* allocates another empty container and uses it this streams new default container. *Sungem* allocates a new default container when the old default container is half full so that the old default container has room to gather segments that are related to those segments already stored in it. Initially, input segments are put into the same (default) container simply because they appear in the input fingerprint stream close together. However, as stored segments accumulate over time, portions of new input segments start to match some of the stored segments partially or exactly. A fingerprint sequence in an input segment that partially matches an existing stored segment forms a new stored segment and is placed in the same container as the existing stored segment, because they share common fingerprints. A fingerprint sequence in an input segment that does not match any stored fingerprints also form a new stored segment and is placed in

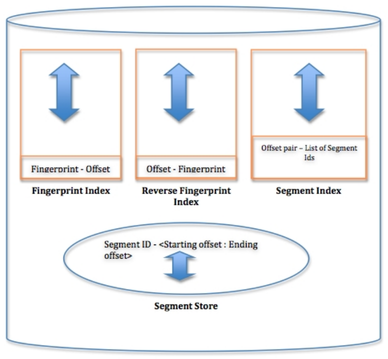
Fingerprint - Offset
Fingerprint Index
Offset - Fingerprint
Reverse Fingerprint Index
Offset pair – List of Segment Ids
Segment Index
Segment ID - <Starting offset : Ending offset>
Segment Store

Figure 5.3: Structure of a Container in *Sungem*

the same container as the stored segment that precedes it in the input segment. If this newly formed stored segment is the first in the input segment, it is placed in the default container. As an example, suppose an input segment has 250 fingerprints, the fingerprint subsequences [1..34] and [206..250] exactly match the stored segments B and F, respectively. the fingerprint subsequences [79..100] and [101..123] and [147..205] partially match the stored segments A, E and D, respectively, and the fingerprint sequences [35..78] and [124..146] do not match any stored segment. In this case, the two fingerprint sequences [79..100] and [101..123] together form a new stored segment that is stored in the same container as the stored segment that its leading sequence, [79..100], partially matches, i.e. A; the fingerprint sequence [147..205] forms a new stored segment that is stored in the same container as the stored segment that it itself partially matches, i.e., D; the fingerprint sequences [35..78] forms a new stored segment that is stored in the same container as the stored segment that its preceding fingerprint sequence, [1..34], matches, i.e. B; and the fingerprint sequence [124..146] forms a new stored segment that is stored in the same container as its preceding fingerprint sequence, [79..123], i.e., A Compared with other data deduplication systems [21, 16], *Sungem* place segments that share common fingerprints or are adjacent in temporal proximity into the same container, whereas others base their placement decision solely on the temporal proximity criterion. An intuitive benefit of putting related segments in the same container is that one disk I/O could bring in all possible stored segments that a sample fingerprint participates in. Moreover, *Sungem*s segment placement algorithm has the desirable side effect of gradually replacing non-stable stored segments with stable stored segments, which correspond to high-level data sharing units. In other words, because *Sungem* limits the number of stored segments (K below) a fingerprint can participate in, stored segments that appear earlier but do not see any exact matches subsequently may quickly be replaced with shorter stored segments that do see exact matches subsequently.

Figure 5.3 shows the content of each container. Stored segments in a container are maintained as an LRU list, and a stored segment is moved to the LRU lists head whenever it is partially or exactly matched by an input segment. The per-container fingerprint index (FI) organizes the fingerprints of a container into a hash table keyed by the fingerprint values, and allows one to find the fingerprint offset. Fingerprint offsets are further used to find list of all stored segments that hold the given fingerprint offsets using Segment Index(SI) tree. SI is a tree keyed by disjoint fingerprint offset pair and whose values are a list of stored segment id's.

For each stored segment, *Sungem* finds the longest matching length of fingerprints in the given Input Segment by comparing each fingerprint in stored segment to the ones in input segment. Since the stored segment needs access to fingerprint using it's offset from segment store, *Sungem* uses a reverse Fingerprint Index (RFI). RFI fetches fingerprint values given the fingerprint offset as key.

The maximum of the longest matching fingerprint sequence is chosen as the best match and used as duplicate cover for fingerprints in given input segment. If part of a stored segment matches as duplicate, then that stored segment needs to be split into 2 parts. Stored Segments can also be merged when neighboring stored segments have contiguous fingerprint offsets in FI. Degree of fingerprint sharing among multiple stored segments is controlled by K Factor that we discussed previously.

Sungem keeps containers in an on-disk container store, and uses an in-memory container store cache to keep containers that were recently fetched from the disk so as to reduce the disk access cost associated with fetching containers.

5.5 Hybrid Garbage Collection Algorithm

As seen in section4 there are two general garbage collection approaches. The first approach is *global mark and sweep*, which freezes a data backup system, scans all of its active backup snapshot representations, marks those physical blocks that are referenced, 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 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 requires an extended pause time that is typically proportional to a data backup system's size, and thus is not appropriate for petabyte-scale data backup systems. Consequently, *Sungem* takes the second approach, which incurs some run-time performance overhead due to metadata bookkeeping but allows it to be scalable. Minimizing this metadata bookkeeping overhead is a key design issue of *Sungem*.

5.5.1 Key Idea

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 *total* size of the disk volume being backed up, rather than the size of *change* to the disk volume in an incremental backup operation, which is much smaller. We propose a *hybrid* garbage collection algorithm specifically for incremental backup systems. It maintains both a reference count and an expiration time for every physical block, and its performance overhead at backup time is proportional to the size of an incremental backup snapshot rather than the disk volume for which the snapshot is taken.

An incremental backup snapshot of a disk volume consists of a set of entries each of which corresponds to a logical block in the disk volume that has been modified since the last backup. Every 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, a current image physical block number (CPBN) that points to the physical block to which the logical block LBN is currently mapped and a duplicate image physical block number (DPBN) that is identified as a duplicate copy of CPBN by deduplication engine. At backup time, given a snapshot entry ⟨LBN, BPBN, CPBN, DPBN⟩ of a disk volume V, the reference count of $BPBN$ is decremented, if $DPBN$ doesn't exist then the reference count of $CPBN$ is incremented or else $CPBN$ is freed and reference count of $DPBN$ is incremented. If $BPBN$'s reference count reaches 0, then it's expiration time is set to the maximum of its current value and the current time plus the retention period of V. At garbage collection time, the physical blocks whose reference count are 0 and whose expiration time is less than the current time are freed.

The reference count of a physical block in this algorithm keeps track of the number of disk volumes, not their backup snapshots, that *currently* points to the given physical block. The expiration time of a physical block records the time after which no backup snapshot will reference it. If there is at least one disk volume currently pointing to a physical block, that physical block cannot be garbage collected and its expiration time is immaterial and thus can be ignored. Whenever a logical block of a disk volume is modified, the disk volume no longer points to the logical block's before image physical block, and the expiration time of that physical block is modified to incorporate the retention time requirement of this disk volume's last backup snapshot. The main advantage of the proposed scheme is that the number of metadata updates is proportional to the amount of change to a disk volume rather than the volume's size itself.

5.5.2 Comparison between RC, ET and Hybrid Garbage Collection

Using the example used to compare different GC schemes, the total number of lookups required to manage snapshots are: $\frac{32TB}{8KB} * \frac{1}{64} * 128 * 0.05 = 0.4$ Billion. The major factor that brings down the lookup count is the operation over 5% delta list change instead of the complete list of blocks in a snapshot.

Compared with *global mark and sweep*, both the *reference count* and *expiration time* scheme shift the performance overhead from the garbage collection time to the backup time, and trade sequential disk accesses for random disk accesses. The ideal garbage collection algorithm is one whose overall efficiency is comparable to that of *global mark and sweep* and that distributes the garbage collection-related overhead over time. The proposed hybrid garbage collection algorithm strikes the best balance among the various GC algorithms discussed, whose overhead comparison is shown in Table 4.1.

However, the metadata bookkeeping necessary for this new hybrid approach might cause some issues because every block needs a centralized database which should be updated based on every lookup operation over the snapshot's delta list. Since the number of blocks on a system can range over several peta bytes and metadata required for maintaining each block is several bytes, BOSC technique is used to optimize the random updates over this metadata database. Lets have a brief overview over BOSC technique before we continue explaining about its adoption.

5.6 BOSC

Batching Operations with Sequential Commit is a generic disk I/O optimization methodology. It's basic idea is to aggregate the incoming random updates in memory coupled with a fast logging mechanism and then a separate background thread sequentially scans the in-memory structure holding the updates to commit them sequentially to the underlying disk. The input thread issuing random updates can be assured of data safety using a fast logging mechanism called TRAIL[8, 22] and hence random disk I/O update throughput can be held very high. To illustrate, assume a B+ tree is used to hold the data. BOSC mechanism can be applied to B+tree very easily by maintaining a per page queue array where each leaf node corresponds to a disk block and each index in per page queue array maps to it's corresponding disk block.

BOSC has three distinct performance advantages. First, because the size of a leaf index page's request queue is typically much smaller than the leaf page itself, typically no more than one tenth, BOSC is able to provide a larger *effective* coverage of the target index's leaf pages because it essentially employs finer-grained caching than page-based caching and thus makes more efficient use of the buffer memory. For example, suppose the available buffer memory is 512MB and each leaf page is 8KB, then at most 64K pages can be cached. However, using per-page request queuing, the same amount of buffer memory can cover up to 640K pages because the per-page request queue is only one tenth of the index page size. This efficiency improvement originates from the fact that the request queuing approach focuses only on those index entries that are to be inserted and does not need to waste precious physical memory space on the parts of an index page that are either empty or already inserted. In contrast, page-based caching cannot get away with such waste.

The second performance advantage of BOSC is that it can accumulate a set of index updates to the same page and commit them to disk in one batch. That is, whenever a leaf index page is brought in, it is likely that BOSC can insert multiple entries into the page, thus effectively amortizing the associated disk read and write I/O cost over multiple index update operations.

Finally, with the consistency guarantee provided by write-ahead logging, BOSC can exploit the flexibility of asynchronous index updates to maximize the disk I/O efficiency. In particular, BOSC can commit those *pending* leaf index pages, i.e., pages whose request queue is non-empty, in the order of the locations of their corresponding on-disk blocks. Consequently, BOSC can bring in the disk blocks associated with pending leaf index pages using (largely) sequential disk I/O, thus can significantly reduce the cost of bringing in each pending leaf index page. Sequentially committing pending leaf index pages means that index updates are not reflected to disk in the temporal order of their occurrence. If the logging module crashes in the midst of committing an index's pending leaf pages, the index's on-disk representation may be left in an inconsistent state. However, this does not compromise the index's consistency, because replaying the index update log at recovery time should restore it to a consistent state. To reduce the log replay time, BOSC constantly keeps track of the *youngest* index update request characterized by the property that every index update request older than it has already been committed, and logs it to disk every time it changes. This special index update request serves as the starting point of the log replay at recovery time.

BOSC is largely independent of the internal structure and search algorithms used in an index and therefore is applicable to most if not all known disk-aware indexing schemes.

5.6.1 BOSC-based Update of Metadata Database

As seen in Hybrid Garbage Collection algorithm 5.5, backing up a logical block triggers the updates of the GC-related metadata of up to three physical blocks(CPBN, BPBN, DPBN). Assume a data backup system manages a 1-PB physical disk space with a 8-KB block size, then there are approximately 128 billion physical blocks, and we need an array of 128 billion entries to record their GC-related metadata. Obviously this GC metadata array (GMA) cannot fit into the memory, and there is no reason to expect much locality in the accesses to this array and hence issuing random disk I/O updates would bring down the throughput of entire deduplication operation. Prior research work[22] on handling random updates in continuous data protection(CDP) systems proves that BOSC methodology is efficient in converting random updates into almost sequential I/O avoiding any disk I/O bottleneck issues. We therefore adopt the BOSC methodology to handle metadata updates in GMA.

Since updates to $CPBN$ are expected to have some locality, it would be efficient to handle updates to a group of blocks together. Hence GMA is partitioned into 128-Kbyte chunks, and each GMA update is queued in its corresponding per-chunk queue after logging it. One or multiple background commit threads are used to to commit the updates to those chunks that have enough pending updates to disk using largely sequential disk I/O. When multiple commit threads are in action, they work on disjoint but neighboring chunks simultaneously.

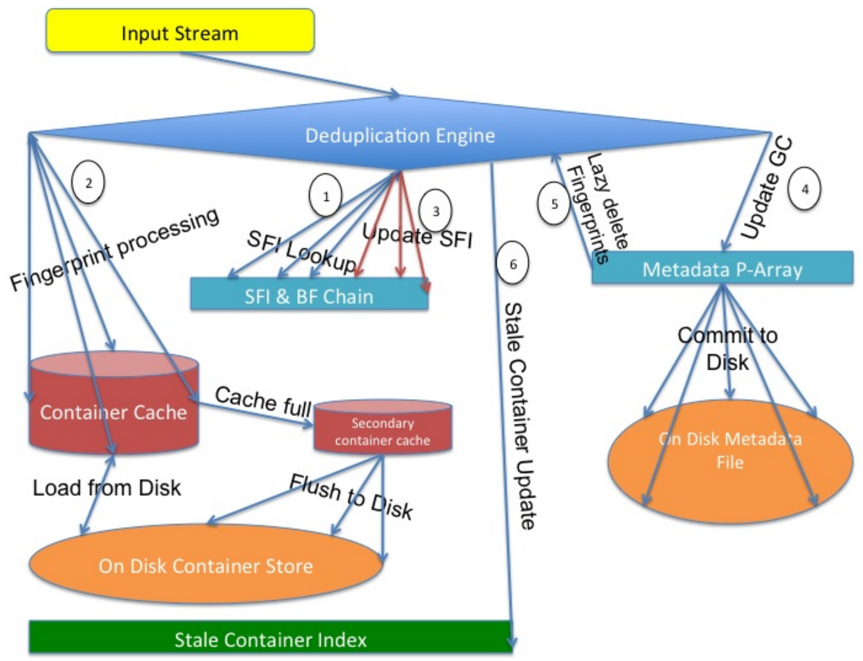

Figure 5.4: Abstract view of the proposed data deduplication engine

5.7 Lazy Deletion of Freed Blocks

Another interesting issue with GC is to make sure the freed blocks are not used as duplicates for incoming blocks in future. That would result in data loss as deduplication frees the current image block thinking it has found a duplicate and the duplicate block is already freed and possibly reallocated to some application. Since current image and duplicate image blocks do not have any locality, they are expected to go into different positions in the GMA index queue. Hence they get updated independently and coordinating between them is extremely complicated. It's not straightforward to delete the fingerprints corresponding to freed blocks from deduplication containers as that would involve increased locking contention of containers with deduplication process and also high disk I/O costs for bringing in the required containers to memory and then storing them back to disk. We propose to use lazy deletion of freed blocks. When a block is freed, we just mark the container holding the corresponding fingerprint as STALE in an in-memory map of stale containers. So in future when that particular STALE container is processed, all the unwanted fingerprints have to be deleted first. This approach helps save locking contention and unwanted disk I/O's.

5.8 Prototype Implementation

Figure 5.4 shows a high-level abstract view of *Sungem*.

- Container store is a very critical component because if it fails to capture the active working set, disk I/Os can bottleneck the entire operation. We further extended the container store to accommodate a secondary cache which acts like a buffer area from which containers are flushed to underlying disks periodically by a dedicated background thread. When primary cache is full, containers are moved from primary cache to secondary cache. Size of this secondary cache should be chosen corresponding to the data transfer rate of the underlying disks so that the secondary cache is never full.

- Size of GMA is another critical component which if configured wrongly, could bottleneck GC and hence the overall deduplication operation. This has to be tuned dynamically depending on the nature of users data. When there is very high locality, container store doesn't involve any disk I/Os and hence deduplication rate is very high. This means GMA updates occur at a very high speed exceeding the rate at which it can be flushed to disk by GC threads. In such cases, GC has to synchronize with deduplication engine to slow down which involves pausing the input operation to deduplication engine for a brief amount of time until GC clears the GMA to a considerable extent. But this is a rare scenario as most of the times deduplication engine involves disk I/Os from container store and that automatically brings down the deduplication rate and hence avoids high thrust on GMA.

- Containers require exclusive lock while it's being operated upon because some operations delete fingerprints from the container and some reference it. To efficiently synchronize all operations on a container, an exclusive lock should be taken before working on it but having an exclusive lock serializes multiple threads wanting to work on the same container simultaneously and hence results in drop in deduplication rate. To handle such scenarios when multiple threads try to work on the same container simultaneously, we measure the time taken to acquire exclusive lock on a container. When the time exceeds a threshold limit, that container is made read only allowing simultaneous access on the container to all other threads. Though this has the advantage of achieving a faster deduplication rate it could suffer from lower deduplication efficiency because new stored segments cannot be created in the same container but instead stored in a different default container which could possibly bring down the locality of reference in future. Hence careful tuning of read only container threshold time could help achieve the best combination of deduplication rate and efficiency.

Chapter 6

Performance Evaluation

6.1 Evaluation Methodology

We used a trace-driven approach to evaluate the effectiveness of *Sungem*, particularly its various key features that are designed specifically for incremental data backup systems.

6.1.1 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 production-mode desktop machines in 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 the *workgroup trace* 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 16-byte MD5 fingerprint for every 4KB block in it. This establishes the baseline image for the entire traced system. 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 16-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 could accurately track the file-level changes between consecutive versions of a file by comparing their constituent fingerprint sequences. However, to produce daily backup fingerprint streams, we need to derive block-level changes, and it is not always possible to faithfully reproduce block-level changes to a file system from its file-level changes. For example, suppose the $N + 1$-th

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 6.1: *The set of file types appearing in the collected trace, their conjectured append behaviors, and the byte count percentage of each type of files in the entire trace.*

version of a file appears to be a concatenation of the N-th version and an additional sequence of blocks. The file-level change our tool detected is the additional block sequence. However, this does not mean that the actual block-level change detected by a block-level change tracker consists only of those writes that create the additional blocks, because the $N + 1$-th version could be produced by overwriting rather than appending to the N-th version. 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. We recognize the above method of deriving block-level changes from file-level changes is not 100% accurate, e.g., it does not take into account de-allocated disk space or file system metadata, but we believe the resulting trace is still sufficiently useful for our study.

Table 6.1 shows the list of file types appearing in the collected trace, the byte count percentage of each file type, 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-related files because we believe these files are relatively rare in a typical office environment.

For file updates that are mainly appends, we only record as block-level changes those newly appended blocks in a new version. if the new version is longer than its previous version. However, if a new version of such a file is shorter than its previous version, this corresponds to a wrap-around scenario, in which case all blocks in the new version are treated as block-level changes. 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 10-week tracing period. From the collected trace, the average daily increment change is about 1.5% of the total size of all the files on these 23 machines.

We implemented the proposed data deduplication engine and garbage collector as a multi-threaded emulator running on a cluster of 6 PowerEdge SC1425 machines each with 4 GB memory and connected by a Gigabit Ethernet switch, and fed it with the collected trace, which consists of 23 daily fingerprint streams for a three-week period. To speed up the evaluation, we put all on-disk containers in memory. To efficiently use all the main memory available in this cluster, we employed MemCached [15] to expedite the trace-driven study. This prototype implementation sustained an average throughput of 12000 fingerprint lookups per second, assuming every container contains up to 2000 20-byte fingerprints. To measure the end-to-end throughput, we also implemented the proposed data deduplication engine and garbage collector that can run on a single machine.

There are 8 configurable parameters in the prototype implementation. $U_{segment}$ is the upper bound on the input segment length. K is the maximum number of segments in which a fingerprint can participate. T_{BSU} is the minimum number of times a segment needs to be referenced before it is considered stable and corresponds to a *basic sharing unit* (BSU). S_{SFI} is the amount of memory reserved to hold the SFI. S_{init} is the initial sampling rate for a store segment before it becomes stable. $S_{container}$ is the container store cache size. $T_{summary}$ is the minimum length of any segments that deserve a summary fingerprint. $U_{container}$ is the on-disk container size.

6.2 Deduplication Performance

In this section, we study the performance of *Sungem* as a whole, which is implemented as an end-to-end deduplication solution and its applied on ITRI CloudOS which is an enterprise level cloud operating system. The evaluation machine 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 GMA 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.

6.2.1 Deduplication Rate and Efficiency

We analyze performance of this end-to-end solution in terms of rate and efficiency. Rate of a deduplication engine is defined as the speed at which fingerprints can be processed by deduplication engine which involves identifying and removal of duplicates. Efficiency is defined as the percentage of duplicates identified in the incoming fingerprints.

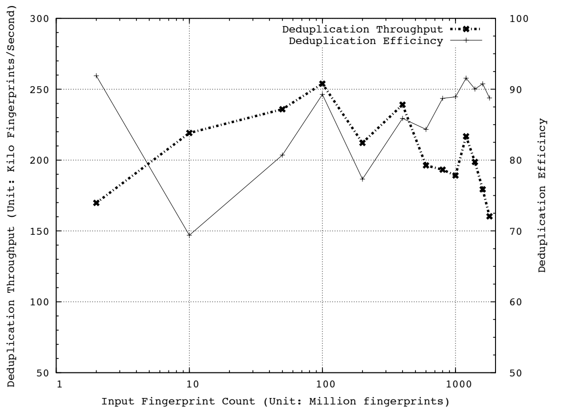

Figure 6.1: *Experiment to demonstrate consistency of Sungem over time.*

6.2.2 Performance Overhead Analysis

Using the trace data we supplied 2 billion fingerprints ranging across 6 consecutive days, as input and observed that *Sungem* was able to consistently deliver throughput above 150K fingerprints/second while maintaining a very high deduplication rate of 80% as shown in Figure 6.1. We observed that the throughput is most of the times around 220K but the average comes down because of occasional but long JVM pause times which we believe can be brought down by careful tuning of JVM together with optimization of deduplication code. JVM maintains separate heap pools for short lived objects and long lived objects called young and old generation respectively. JVM believes in "Most Objects Die Young" assumption and accordingly majority of young generation objects are garbage collected frequently and those objects that live longer are moved to old generation. Garbage collecting on old generation is expensive and hence slower because it's size is huge and naturally the time required to mark and sweep is higher. Therefore, typically efficient java programs make their best attempt to keep the objects short lived. *Sungem* too embraces this rule, but at times when container cache cannot capture the working set, disk I/O's are higher thereby making some objects long lived and thereby pulling down the overall deduplication rate. JVM can also be tuned to adjust the ratios of young and old generation heap sizes and also to adjust the threshold time which decides after how much time an object should be moved to old generation pool. We tune the JVM for these and other parameters which are beyond the scope of this analysis. The dips in deduplication rate in Figure 6.1 is precisely for the container cache working set issue. We further optimize *Sungem* by restarting it periodically to make sure old generation pool is never too big to handle. But this has to be done carefully and not too frequently as restarting *Sungem* involves restoring container caches and syncing other main data structures which is a time consuming process.

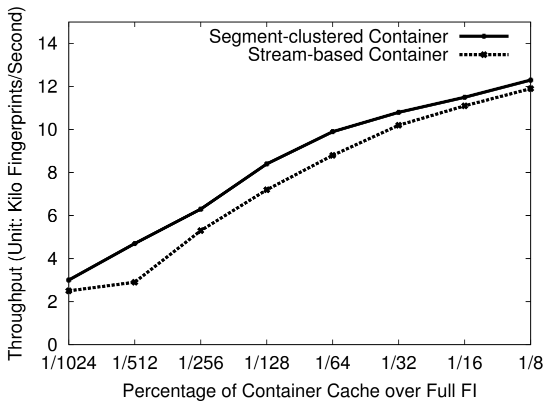

Figure 6.2: *Deduplication rate comparison between the proposed segment-based container management scheme and the standard stream-based container management scheme when the size of the container cache is varied from $\frac{1}{1024}$ to $\frac{1}{8}$ of the full fingerprint index's size. $U_{container}$ = 16384. $U_{segment}$ = 4096, and T_{BSU} = 2. The SFI is $\frac{1}{128}$ the size of the full fingerprint index.*

6.2.3 Effectiveness of Relatedness-based Segment Clustering

In this subsection, we report the deduplication rate of the *Sungem* prototype using the collected trace as the input workload. The measurements were taken on a Dell PowerEdge SC1425 machine, which is equipped with a 2.8 GHz Intel Xeon CPU, 4GB RAM and 2 80-GB 7200 RPM SATA hard drives. One SATA drive holds the input trace while the other drive holds the on-disk container store.

Compared with existing stream-based deduplication engines, which place stored segments into containers based solely on their temporal proximity, *Sungem* clusters stored segments into containers based on both their content, i.e., sharing common fingerprints, and their temporal proximity. Because *Sungem* strives to put *related* stored segments in the same container, it increases the percentage of segments in every container fetched into memory that are actually accessed. As a result, the deduplication rate of *Sungem* is higher than that of the stream-based deduplication engine across all container cache size, as shown in Figure 6.2. The relative performance improvement is up to 70% (e.g. when the container cache is $\frac{1}{512}$ the size of the full fingerprint index), and is more pronounced when the container cache is smaller. When the container cache is smaller, more disk I/Os are needed and higher efficiency of every disk I/O (in terms of bytes that are retrieved and subsequently accessed) leads to larger improvement in deduplication rate throughput.

As expected, the deduplication rate of both *Sungem* and the stream-based deduplication engine increases with the container cache size. However, the deduplication rate improvement is sub linear with respect to the increase in container cache size, because accesses to containers are largely random and do not exhibit a lot of locality.

Figure 6.3 shows that there exists an optimal container size that can achieve the highest deduplication rate. For the input trace, the optimal container size is 1MB. Larger container size enables

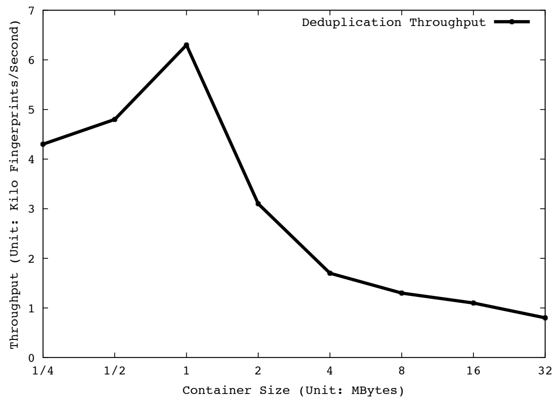

Figure 6.3: *The impact of the container size on Sungem's deduplication rate. The container cache is $\frac{1}{128}$ the size of the full fingerprint index. $U_{segment}$ = 4096, and T_{BSU} = 2. The SFI is $\frac{1}{128}$ the size of the full fingerprint index.*

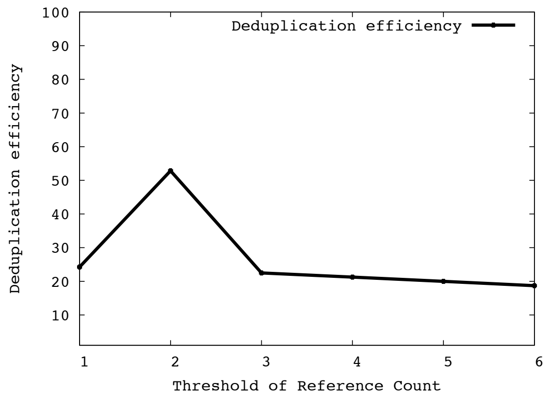

Figure 6.4: *The impact of T_{BSU} (reference count threshold) on the total deduplication efficiency. $U_{segment}$ = 4096. K=2, and the SFI is $\frac{1}{128}$ the size of the whole fingerprint index. The initial sampling rate is 1 out of 16.*

the cost of fetching a container to be amortized over more fingerprint accesses but may load a higher percentage of segments that are fetched but never accessed.

6.2.4 Effectiveness of Variable-Frequency Segment Sampling

Stable Stored Segment Criterion

A stored segment that has seen multiple exact matches after its formation is likely to correspond to a basic sharing unit (BSU). After *Sungem* considers a stored segment as a basic sharing unit, it reduces the number of samples for the segment in the SFI to 1. *Sungem* declares a stored segment as a BSU after the stored segment has been matched exactly by at least T_{BSU} input segment portions.

Figure 6.4 shows that the deduplication efficiency reaches its maximum when T_{BSU} is equal to

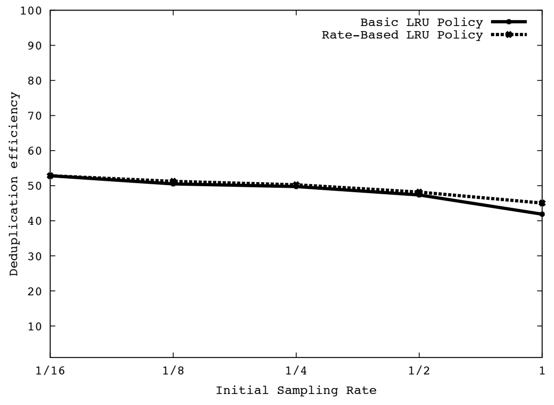

Figure 6.5: *Comparison of two SFI replacement policies,* Basic-LRU *and* Rate-based-LRU*, in terms of their associated deduplication efficiencies when the initial fingerprint sampling rate is varied from* $\frac{1}{16}$ *to 1. SFI is assumed to be* $\frac{1}{8}$ *the size of the full fingerprint index.* $U_{segment}$ *= 512,* K *= 2, and* T_{BSU} *= 2.*

2. When T_{BSU} is 1, the false positive (FP) rate is so high that many stored segments that are not a BSU remove their sample fingerprints from the SFI prematurely, and miss their opportunities of matching input segments and thus deduplicating them. When T_{BSU} is larger than 2, the percentage of stored segments that are considered as a BSU drops from 60% to less than 10%, and the memory space allocated to the SFI is wasted on holding sample fingerprints that could have been safely removed. This memory space waste causes some of the input segments that could have been matched and deduplicated to be missed, and the overall deduplication efficiency drops. As T_{BSU} continues to increase beyond 3, the deduplication efficiency decreases only slightly because the percentage of BSU segments also decreases slightly. Setting T_{BSU} to a large value essentially turns off the optimization of reducing the sampling rate for BSU segments. Figure 6.4 shows that this optimization could lead to more than 70% improvement in the deduplication efficiency (compare the deduplication efficiency when T_{BSU} = 2 and when T_{BSU} = 6).

Rate-based LRU Policy

Analysis of the trace shows the probability that temporal distance between two consecutive instances of a duplicate block is less than or equal to 9 days is more than 99%. The strong temporal locality suggests that the SFI should use a LRU-based replacement policy to manage the cached fingerprints stored in its limited memory space.

Figure 6.5 shows that *Sungem*'s *Rate-based LRU* scheme improves the deduplication efficiency only slightly over the *Basic LRU* scheme with *Sungem*'s *Rate-based LRU* scheme, and the improvement increases with the increase in the initial fingerprint sampling rate, S_{init} . The *Rate-based LRU* scheme down-samples the least recently used segments when the SFI runs out of memory, as opposed to eviction of the entire least recently used segments, as is the case in the *Basic LRU* scheme

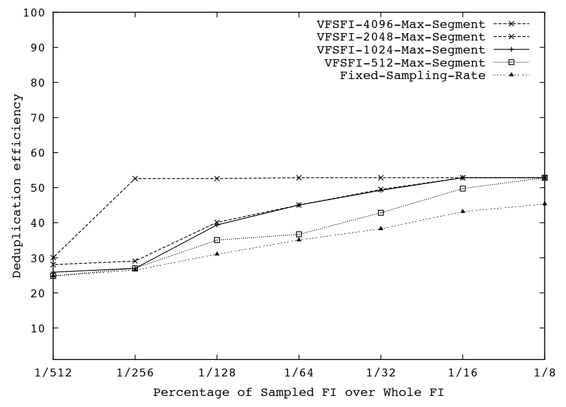

Figure 6.6: *The impacts of the SFI size (S_{SFI}) and the maximal input segment length (($U_{segment}$) on the deduplication efficincy of VFSFI and the fixed fingerprint sampling scheme. For VFSFI, $U_{segment}$ is varied from 512 to 4096. T_{BSU} = 2, K =2. The initial sampling rate is $\frac{1}{16}$.*

. The *Rate-based LRU* scheme performs better because it allows the SFI to cover a larger number of distinct segments. This performance advantage is more pronounced when the number of distinct segments that can fit into the SFI decreases, which occurs when the initial fingerprint sampling rate is higher.

Impact of SFI Size and Maximal Input Segment Length

Compared with fixed fingerprint sampling schemes, *Sungem*'s variable-frequency SFI (VFSFI) is expected to achieve better deduplication efficiencies because it uses the memory space of SFI more efficiently through the identification of BSU segments and the rate-based LRU replacement policy. Figure 6.6 shows the impacts of the SFI size (S_{SFI}) and the maximal input segment length ($U_{segment}$) on the deduplication efficiency of VFSFI and the fixed fingerprint sampling scheme. When the SFI is very small, improving SFI's memory space utilization does not help much. When the SFI is large, it is not necessary to use SFI's memory space very efficiently. That's why the relative deduplication efficiency improvement of VFSFI overs the fixed fingerprint sampling scheme is maximized (i.e., 35%) when the SFI is about $\frac{1}{32}$ the size of the full fingerprint index.

Figure 6.6 also shows that ($U_{segment}$) has a noticeable impact on VFSFI's deduplication efficiency. As $U_{segment}$ increases from 512 to 4096, BSU segments are larger, the SFI memory space saving due to reduced sampling is more, the SFI could cover more segments, and more input segments are deduplicated.

6.3 Garbage Collection Overhead

Effectiveness of our hybrid Garbage Collection scheme can be demonstrated by comparing it with a vanilla GMA update implementation, which buffers GMA update requests in a queue, and uses a

Commit Threads	*Deduplication + vanilla GC*	*Deduplication + BOSC-based GC*	*Deduplication without GC*
1	4441	40821	216938
2	4534	202582	216938
4	7583	209721	216938
10	6134	203363	216938

Table 6.2: *End-to-end throughputs(fingerprints processed/second) of a data deduplication engine with multiple garbage collector configurations.*

background thread to commit them on a first come first serve basis. In Table 6.2, the throughput of the data deduplication engine without any GMA updates (the last column) sets an upper bound because it corresponds to a zero-cost GMA update scheme. When the BOSC-based GMA 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 GMA update scheme never exceeds 5% of the upper bound.

Chapter 7

Distributed Deduplication and Garbage collection

7.1 Motivation

Over time, the amount of backup data keeps increasing but the time required to backup remains the same. That means backup operation has to operate faster and handle huge amount of data at same time. Having multiple parallel instances of deduplication engine on different nodes can solve the problem, provided all the engines cooperate efficiently. But SFI table size and GC metadata size pose some limitations to scalability.

SFI is an in-memory table and it's one of the most important data structures which guides the entire deduplication process as to where the duplicates can be found. As the deduplication engine gets more and more data, the representative sample for each stored segment in SFI goes down because SFI is a fixed memory data structure residing on RAM. We guarantee that even if one representative of a container is present in the SFI table, we can cover all the fingerprints in the corresponding input fingerprint. But when even 1 fingerprint from a container is not present in SFI table, there is no way the container can be reached. Such a scenario can be avoided by having a bigger in-memory table. But constraints on the in-memory resources limits the SFI table size and therefore distributing the SFI table to different nodes looks to be a feasible solution as it helps achieve an effectively bigger SFI table.

Another interesting issue is with garbage collection which needs to maintain metadata for each block on the system. With the increase in the storage capacity of the system, GC needs to scale to handle more metadata. Since these metadata do not have any inter-dependency between each other, it's better if each node can handle different portions of metadata block updates individually.

7.2 Our Idea

The fundamental guideline behind the distributed version is to make it functionally equivalent to the standalone version with a slight overhead of network communication. We observe that the standalone version gives a very high throughput and makes the best possible attempt to amortize disk I/O's because of efficient parallelized architecture. But the entire operation continues to be disk bound in loading and storing container files. Hence the primary goal of distributed version is to not incur any additional disk I/Os. More importantly it tries to distribute the disk I/O's across multiple nodes.

The distributed version has a master node and a set of worker nodes. Master node is responsible for accepting incoming fingerprints and getting it deduplicated by worker nodes. Worker nodes run a a slightly modified version of standalone *Sungem* to accommodate network communication with master node.

7.2.1 Processing Hit Fingerprints

The input segment is formed on the master node in a way similar to the standalone version. This input segment is broadcasted to all worker nodes and each of them have a modified standalone deduplication version waiting for orders from Master Node. Each worker node looks up the SFI for all fingerprints in input segment and then processes the HIT fingerprints. Along with processing these HIT fingerprints, any neighboring fingerprints are also processed. So this step converts miss fingerprints to Pseudo HIT fingerprints using HIT fingerprints as the anchor. Each worker node then returns the updated input segment to master node.

7.2.2 Processing Remaining Fingerprints

The master node merges input segments from all worker nodes and updates them in it's version of the input segment. Using the HIT and Pseudo HIT fingerprints as the anchor, it processes MISS fingerprints. The intention is that these set of fingerprints will probably repeat itself in future and their locality information can be preserved by storing all these fingerprints in same container if possible. So in future if this input segment is repeated, it can be processed by loading as many less containers as possible from the disk/caches. Miss fingerprint are processed by sending the request to worker node that stores the nearest fingerprint in the given input segment. In case of no HITS in worker node, a worker node that's least loaded is selected to store the remaining MISS fingerprints in it's open container.

7.2.3 SFI Update

Master node broadcasts entire input segment to all worker nodes and each worker node aggregates all the fingerprints in the input segment that's stored locally on them. It then updates its SFI for each stored segment which is a subset of the input segment.

Since fingerprints get distributed evenly over all worker nodes over a period of time, SFI gets utilized evenly on all worker nodes.

7.2.4 Distributed Garbage Collection

Since metadata of each physical block are independent of each other, each worker node can manage independently a discrete subset of physical block numbers. After an input segment is deduplicated, master node broadcasts input segment to all worker nodes. Each worker node is responsible for garbage collecting fingerprints belonging to them. As an optimization, this step is merged with SFI update. Each worker node updates metadata similar to the standalone version of garbage collector. There is however a slight modification in the way fingerprints are deleted from the containers. Since containers might not be locally available on a worker node for each block to be freed, marking a container stale requires additional care. Containers which need to be marked stale are aggregated for the entire GC chunk(group of metadata block updates in a GC chunk) and are then forwarded to the master node requesting to mark them stale. The master node receives this stale request list and then segregates the list based on target worker node which holds the given fingerprint. Segregated fingerprint lists are then forwarded to their respective nodes. Each worker node maintains its own in-memory stale container map and they are updated similar to the standalone approach. But since the node at which target containers are stored and the node at which a block is garbage collected can be distinct, synchronizing the entire operation is complicated leading to tiny timing windows during which GC metadata could go inconsistent. To realize this scenario, consider the GC thread which tries to free a block. By the time it reaches the target worker node which stores the corresponding container, another deduplication thread could refer the to be deleted block as it's duplicate block. To make things worse, it could also free it's current image block. Then the GC thread could mark the container stale and free the corresponding block. So from the deduplication thread's point of view, by the time it tries to increase the reference count on duplicate block, the required block is already freed and thereby resulting in data loss. To avoid such a scenario, each stored fingerprint carries with it a timestamp to indicate the time at which it was last referenced as a duplicate. So GC thread can avoid freeing a block which has was accessed very recently and hence avoid any data loss issue.

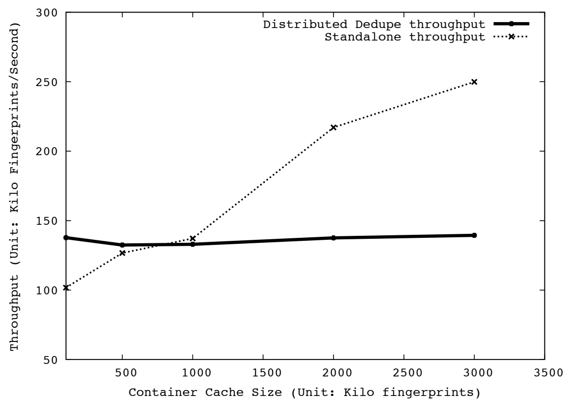

Figure 7.1: Throughput comparison between Distributed and Standalone Deduplication

7.2.5 Design Tradeoffs

Revisiting the distributed design, the primary assumption behind distributed approach is that the entire deduplication operation is disk bound on loading and storing container files. Based on that assumption, master node distributes the input segment to worker nodes and each worker node handles container loading and storing independently. But the assumption is based on input workload which need not be disk bound for all cases. Hence in situations where the system is not disk bound, its better to follow a naive distributed design where each worker node runs the unmodified standalone deduplication engine and let the master node split the input segment into multiple smaller input segments by hashing the fingerprints to each worker node. Each subset input segment is then forwarded to corresponding worker nodes. By splitting input segments into multiple smaller input segments, the number of disk I/O's on each worker node is expected to be same as that on standalone version. So collective count of disk I/O's on all worker nodes is almost increased by a factor of the number of worker nodes but since the workload is not disk I/O bound, this design would perform faster than the original distributed design.

7.3 Evaluations

The evaluation machine consists of two quad-core 3.4GHz Intel Core i-7 processor, 8GB of RAM, five 7200-RPM WD Caviar blue hard disks of 1TB each, one for the system disk and two for storing the GMA 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. The distributed prototype consists of three such similar machines with 1 master node and 2 worker nodes.

To induce disk boundedness into the system, we will vary the container cache size and then show the difference in throughput of distributed model against a standalone model.

Figure 7.1 shows how the throughput of Standalone deduplication drops when container cache size is brought down. With our trace 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 distributed design will not improve the performance as it will only cause more disk I/O's and deteriorate the performance. But the performance of distributed 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 2000, the working set of containers are cached appropriately and hence standalone deduplication drives a high throughput of more than 200K fingerprints/second. The experiment clearly demonstrates the effectiveness of different parallelizing strategies under different conditions.

Chapter 8

Future Work

As seen in section 5.1, very frequent WADB operations could guarantee High Availability(HA). But since WADB operation involves copying actual payload data between data centers which could be located very far geographically, data copying time could bottleneck the frequency of WADB updates and hence make HA ineffective sometimes. One possible solution to avoid this bottleneck is to provide network deduplication and avoid transferring unnecessary data. In brief, source node sends the fingerprints of all blocks to be copied to remote node and remote node checks from its local deduplication engine and asks for actual payload data from source node only if that block is not already present in it's data repository. Since this process involves transferring only fingerprints of each block and a few unique payload data across the network, data copy time can be brought down drastically. But since duplicates are found based on fingerprints and not on actual payload data there is some very low probability of a mismatch. Even though the probability of such a mismatch is guaranteed to be as low as a hardware error some applications cannot afford that too. Though byte-by-byte comparison can be applied later to confirm the duplicates, that essentially involves copying actual payload data across the network and hence defeats the purpose of network deduplication. So as future work, we are focussed on providing an efficient solution for HA.

Bibliography

[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] M. A. Bender, M. Farach-Colton, R. Johnson, B. C. Kuszmaul, D. Medjedovic, P. Montes, P. Shetty, R. P. Spillane, and E. Zadok. Dont thrash: How to cache your hash on flash. In *FAST '11: Proceedings of the 8th conference on File and storage technologies*, 2011.

[3] 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.

[4] J. L. Biplob Debnath, Sudipta Sengupta. Chunkstash: Speeding up inline storage deduplication using flash memory. In *USENIXATC'10: Proceedings of the 2010 USENIX Conference on USENIX annual technical conference*, 2010.

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

[6] 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.

[7] K. Cheng, L. Xiang, M. Iwaihara, H. Xu, and M. M. Mohania. Time- decaying bloom filters for data streams with skewed distributions. In *IEEE RIDE 05: Proceedings of the15th International Workshop on Research Issues in Data Engineering: Stream Data Mining and Applications*, 2005.

[8] T. Chiueh and L. Huang. Track-Based Disk Logging. In *DSN '02: Proceedings of the 2002 International Conference on Dependable Systems and Networks*, pages 429–438, Washington, DC, USA, 2002. IEEE Computer Society.

[9] 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.

[10] B. Debnath, S. Sengupta, J. Li, D. J. Lilja, and D. H. Du. Bloomflash: Bloom filter on flash-based storage. In *ICDCS: International Conference on Distributed Computing Systems*, 2011.

[11] 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.

[12] L. Fan, P. Cao, J. Almeida, and A. Z. Broder. Summary cache: A scalable wide-area web cache sharing protocol. In *ACM SIGCOMM '98: Conference on Applications, Technology, Architectures and Protocols for Computer Communications*, 1998.

[13] D. Feng, L. Zeng, F. Wang, and P. Xia. Tlfs: High performance tape library file system for data backup and archive.

[14] Gartner. http://www.mainframezone.com/storage/backup-recovery-business-continuity/tape-a-collapsing-star.

[15] M. Group. MemCacheD: a distributed memory object caching system. http://memcached.org/, 2010.

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

[17] N. A. Inc. Netapp Deduplication (ASIS). http://www.netapp.com/us/products/platform-os/dedupe.html, 2004.

[18] D. E. Knuth. *The The Art of Computer Programming : Sorting and Searching, volume 3*. Addison-Wesley, 1973.

[19] E. Kruus, C. Ungureanu, and C. Dubnicki. Bimodal content dened chunking for backup streams. In *FAST '10: Proceedings of the 8th conference on File and storage technologies*, 2010.

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

[21] G. Lu, B. Debnath, and D. H. Du. A forest-structured bloom filter with flash memory. In *MSST: Storage Conference*, 2011.

[22] M. Lu, D. Simha, and T. Chiueh. Scalable Index Update for Block-Level Continuous Data Protection. In *NAS '11: 6th IEEE International Conference on Networking, Architecture and Storage*, Dalian, China, 2011. IEEE Computer Society.

[23] 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.

[24] 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.

[25] S. Tarkoma, C. E. Rothenberg, and E. Lagerspetz. Theory and practice of bloom filters for distributed systems. *Communications Surveys and Tutorials, IEEE*, 99:1 – 25, 2011.

[26] J. Wei, H. Jiang, K. Zhou, and D. Feng. Mad2: A scalable high-throughput exact deduplication approach for network backup services. In *MSST: Storage conference*, 2010.

[27] J. Wei, H. Jiang, K. Zhou, D. Feng, and H. Wang. Detecting duplicates over sliding windows with ram-efficient detached counting bloom filter arrays. In *NAS: 6th IEEE International Conference on Networking, Architecture, and Storage*, 2011.

[28] L. L. You, K. T. Pollack, and D. D. E. Long. Deep store: An archival storage system architecture.

[29] 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.