Efficient Deduplication Techniques for Modern Backup Operation

Jaehong Min, Daeyoung Yoon, and Youjip Won

Abstract—In this work, we focus on optimizing the deduplication system by adjusting the pertinent factors in fingerprint lookup and chunking, the factors which we identify as the key ingredients of efficient deduplication. For efficient fingerprint lookup, we propose fingerprint management scheme called LRU-based Index Partitioning. For efficient chunking, we propose Incremental Modulo-K(INC-K) algorithm which is optimized Rabin's algorithm where we significantly reduce the number of arithmetic operations exploiting the algebraic nature of modulo arithmetic. LRU-based Index Partitioning uses the notion of *tablet* and enforces access locality of the fingerprint lookup in storing fingerprints. We maintain tablets with LRU manner to exploit temporal locality of the fingerprint lookup. To preserve access correlation across the tablets, we apply prefetching in maintaining tablet list. We propose Context-aware chunking to maximize chunking speed and deduplication ratio. We develop prototype backup system and performed comprehensive analysis on various factors and their relationship: average chunk size, chunking speed, deduplication ratio, tablet management algorithms, and overall backup speed. By increasing the average chunk size from 4 KB to 10 KB, chunking time increases by 34.3 percent, deduplication ratio decreases by 0.66 percent and the overall backup speed increases by 50 percent (from 51.4 MB/sec to 77.8 MB/sec).

Index Terms—Deduplication, chunking, backup, index partitioning, fingerprint lookup.

1 Introduction

1.1 Motivation

The recent introduction of digital TV, digital camcorders, and other communication technologies has rapidly accelerated the amount of data being maintained in digital form. In 2007, for the first time ever, the total volume of digital contents exceeded the global storage capacity, and it is estimated that by 2011 only half of the digital information will be stored [1]. Further, the volume of automatically generated information exceeds the volume of human generated digital information [1]. Compounding the problem of storage space, digitized information has a more fundamental problem: it is more vulnerable to error compared to the information in legacy media, e.g., paper, book, and film. When data is stored in a computer storage system, a single storage error or power failure can put a large amount of information in danger. To protect against such problems, a number of technologies to strengthen the availability and reliability of digital data have been used, including mirroring, replication, and adding parity information. In the application layer, the administrator replicates the data onto additional copies called "backups" so that that the original information can be restored in case of data loss.

Due to the exponential growth in the volume of digital data, the backup operation is no longer routine. Further, exploiting commonalities in a file or among a set of files when storing and transmitting contents is no longer an option. By properly removing information redundancy in a file system, the amount of information to manage is effectively reduced, significantly reducing the time and space requirement of managing information, e.g., backups. Fig. 1 schematically illustrates the effect of data deduplication.

The deduplication module partitions a file into chunks, generates the respective summary information, which we call a fingerprint, and looks up Fingerprint Table to determine if the respective chunk already exists. If it does not exist, the fingerprint value is inserted into Fingerprint Table. Chunking and fingerprint management is the key technical constituents which governs the overall deduplication performance. There are a number of ways for chunking, e.g., variable size chunking, fixed size chunking, or mixture of both. There are a number of ways to managing fingerprints. Legacy index structure, e.g., B+tree, and hashing does not fit for deduplication workload. A sequence of fingerprints generated from a single file(or from a set of files) does not yield any spatial locality in Fingerprint Table. On the same token, a sequence of fingerprint lookup operations can result in a random read on Fingerprint Table, and therefore each fingerprint lookup can result in disk access. Given that most of the deduplication operation needs to be performed online, it is critical that fingerprint lookup and insert is performed with minimal disk access.

In this work, we identify two key technical ingredients in the deduplication backup system, chunking and fingerprint lookup and develop novel mechanisms to address each of these issues. The contribution of our work is three folds. First, we develop a novel chunking method called context-aware chunking. In context-aware chunking, we exploit the algebraic nature of the *modulo* arithmetic and develop the *Incremental Modulo-K* algorithm, which significantly reduces the computational overhead of signature generation. We adaptively apply a fixed size or variable size chunking subject to the file type.

Second, we develop an efficient fingerprint management scheme called LRU-based index partitioning. We partition a fingerprint table into smaller sized tables called *tablets* which

• *The authors are with the Department of Computer Science, Hanyang University, Seoul 133-791, Korea.*
E-mail: jhmin@macroimpact.com, {dayyoung, yjwon}@ece.hanyang.ac.kr.

Manuscript received 16 Nov. 2009; revised 12 Aug. 2010; accepted 5 Oct. 2010; published online 7 Dec. 2010.
Recommended for acceptance by R. Gupta.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-2009-11-0573.
Digital Object Identifier no. 10.1109/TC.2010.263.

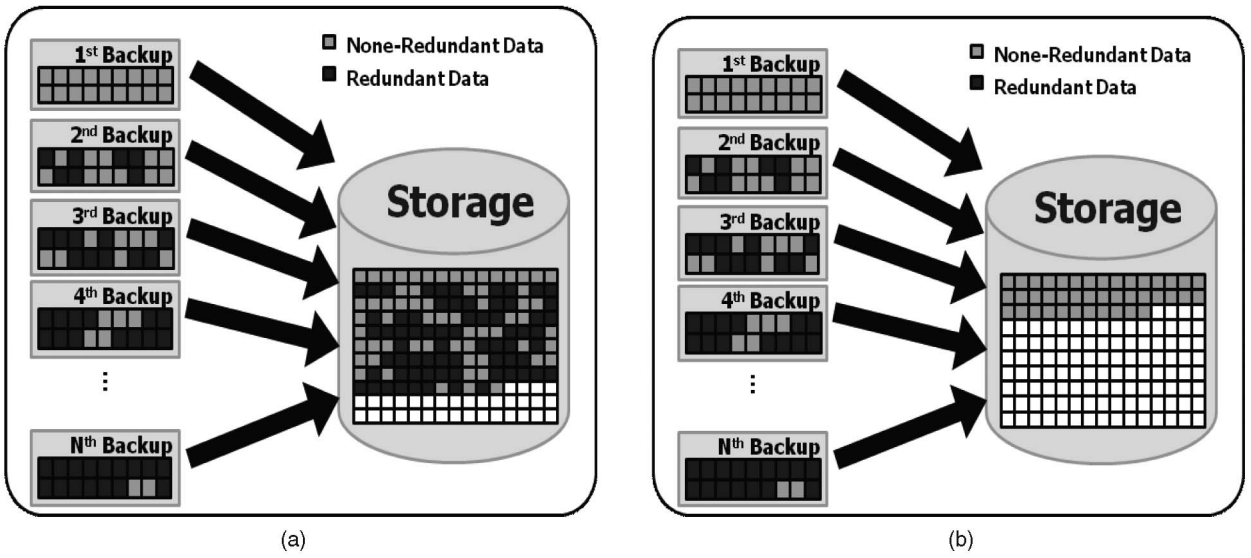

Fig. 1. Data deduplication. (a) Normal backup. (b) Deduplication backup.

is a few MB. When inserting a sequence of fingerprints to a tablet than to a large size table, we place a sequence of fingerprint values onto the disk in a clustered manner. With the index partitioning, we can enforce the spatial locality of the fingerprint lookup sequence via confining the search range into a relatively smaller file. The fingerprint lookup overhead, however, can linearly increase with the number of tablets. To minimize the overhead of the scanning tablets, we maintain the access history of the tablets as the LRU list and we prefetch the next tablet.

Third, by physical experiment, we study the performance relationship between the chunking and fingerprint lookup overheads. While numerous preceding works have focused on developing novel chunking algorithms and fingerprint lookup techniques, few works have paid attention to the relationship between the two. Chunking a file in larger granularity makes the chunking process more time consuming but makes the fingerprint lookup faster. It is very important to identify the appropriate set of tuning knobs and to properly orchestrate the chunking process and the fingerprint lookup in order to improve the overall deduplication performance. We put particular effort in performing physical experiment in comprehensive manner and in its analysis. We develop prototype backup system and perform extensive experiment to analyze the performance behavior of the given technique. We use six different average chunk sizes, eight chunking algorithms, four fingerprint management algorithms, and three different data sets. We examine the relationships among the average chunk size, chunking speed, deduplication ratio, backup speed under different combinations of above mentioned factors.

There are a number of important lessons in this study. LRU-based index partitioning exhibits very good scalability and can fully exploit the disk bandwidth. Fixed-size chunking works well with regard to both the backup bandwidth and the degree of redundancy detection. This is because large files, which constitute the dominant fraction of the data volume, are multimedia files. These files are either identical or entirely different. On the other hand, small text files, e.g., source codes, have many commonalities across different versions of the operating system source code tree. However, these files are usually small and detecting commonalities in them may not justify the efforts of variable-size chunking. Chunking the file(s) in finer granularity enabled us to more quickly generate variable-size chunks and to find more commonalities among them. However, we also found that chunking significantly increases the fingerprint lookup overhead. By increasing the target pattern size from 11 bits to 13 bits, the deduplication detection rate decreased by two percent and the chunking performance decreased from approximately 150 MB/sec to 100 MB/sec with files being in memory. However, the overall deduplication speed increased from 51 MB/sec to 77 MB/sec.

1.2 Related Works

There largely exist three approaches for reducing the size of information: delta encoding, duplication elimination, and compression. Each of these techniques is used independently or in a combined manner to improve the space efficiency and network bandwidth utilization. Delta encoding stores only the differences between sequential data. It is a common and efficient method to reduce data redundancy when changes are small. It is used in many applications including source control [2] and backup [3]. Kalkarni et al. [4] proposed redundancy elimination at block level (REBL), which is a combination of block suppression, delta encoding, and compression.

Backup applications also exploit information redundancy to reduce the amount of information to be backed up [5], [6], [7]. In a distributed file system, the deduplication technique has been used to reduce the network traffic involved in synchronizing file system contents between a client and a server [8]. In a SAN file system, when different files share an identical piece of information, each file harbors the pointer to the shared data chunk instead of maintaining redundant information [9]. Detecting information redundancy is widely used when locating the document source for a multisource download application [10].

The web environment is an important area for duplication detection and elimination. These applications compute the fingerprints of the contents in the proxy server and eliminate the retrieval of the same data [11], [12].

Won et al. found that chunking is one of the major overheads for the deduplication process [13], [6]. There are a two basic approaches in partitioning a file: variable-size chunking and fixed-size chunking. A number of preceding works have adopted fixed-size chunking for backup applications [14] and large-scale file systems [9]. The variable-size chunking algorithm is widely used in various application domains of duplication elimination such as backups, file systems, and data transfers [12], [15], [10], [8], [16], [7]. Policroniades et al. examined the effectiveness of variable-size chunking and fixed-size chunking using website data, different data profiles in academic data, source codes, compressed data, and packed files [17]. A few works proposed to apply variable size chunking and fixed size chunking based upon the characteristics of the file. Liu et al. proposed ADMAD scheme [18] which applies different file chunking methods based upon the metadata of individual files. Context-Aware chunking proposed in our work shares the basic idea with Liu's work. However, we only use file extension rather than all file metadata to reduce the overhead of accessing file metadata. Meister et al. exploited the file characteristics, e.g., compressed file, email archive, etc., in chunking a file [19] and analyzed the deduplication efficiency under various chunking schemes. While they show that delta encoding yield the best deduplication ratio in desktop applications, e.g., MS Office, Zip, compared to variable size and fixed size chunking, they did not address the issue of excessive fingerprint generation in delta encoding. Mandagere et al. examine the effect of different chunking methods (fixed size chunking, variable size chunking) and different chunk size settings over deduplication performance metrics: fold factor, reconstruction overhead, and CPU utilization [20]. They did not examine effect of chunk size over entire backup speed, on which we perform comprehensive study. Similar with Meister's work [19], the reduction on redundancy detection rate is marginal, from 34.9 to 33.2 percent when chunk size increase from 8 to 16 KB. Instead of deduplicating files at chunk granularity, Bolosky et al. proposed a method to detect duplicate data using file granularity [21]. Deep Store is designed to adaptively accommodate different types of chunking mechanisms [22].

There are a number of aspects to expedite the fingerprint lookup. The first issue is to introduce main memory filter. To avoid disk I/O when looking up the index, a main memory filter called the Bloom filter [23] has been introduced in various applications including backups [7], distributed file systems [24], and web proxies [11]. The important question yet to be answered is the relationship between the false positive rate and the overall lookup performance. Mitzenmacher [11] made the interesting observation that minimizing the false positive rate of the Bloom filter did not necessarily yield the optimal performance of the web proxy lookup. Rather, sacrificing the false positive rate and making the bit vector of the Bloom filter more compressive actually improves the fingerprint lookup overhead. The second aspect is to reduce the number of fingerprints used in comparison. Lillibriged et al. proposed to use sampling to reduce the number of fingerprints. It can reduce the memory requirement and the degradation in deduplication ratio is reasonable [25]. They constructed a sparse index that contained sample chunks and mapped the chunks to their reference segments. Bobbarjung et al. proposed hierarchical way of maintaining chunks and the respective redundancy checking algorithm, fingerdiff [26]. They proposed to make chunk size smaller, e.g., 1 KB, and maintain a set of chunks as a single unit. They aimed at improving the deduplication ratio by using small chunk and reducing the number of comparisons by maintaining a group of chunks as a single unit. This approach does not work when there is a significant change in chunk size, i.e., removal or insertion of data. To reduce the number of comparison, Aronovich et al. proposed to maintain summary information in larger unit, e.g., 10 MB, and deduplicate the data based upon its similarity with the existing data [27]. To reduce the overhead of fingerprint lookup, Hamilton et al. maintain fingerprints in hierarchical manner. They maintain tree of fingerprints where a parent node's fingerprint is the hash value of the fingerprints of the child nodes [28]. Bhagawat et al. exploit the file similarity instead of locality in deduplication [29]. Their work manifests itself when backing up a set of small files arriving from different hosts. Third approach is to reduce the disk overhead in fingerprint lookup. The key ingredient is to enforce access locality in storing fingerprints at the storage. Zhu et al. [7] proposed a technique called SISL, where they simply append the incoming fingerprints at the end of existing table. Spyglass, a file metadata search system, proposed hierarchical partitioning of name space organization for performance and scalability [30]. Spyglass exploits namespace locality to improve performance since the files that satisfy a query are often clustered in only a portion of the namespace.

Since a number of files share same piece of information, loss of a chunk may result in loss of multiple files. A number of works addressed the reliability issue in deduplication. To enhance reliability of deduplicated data, Liu et al. proposed to form a set of variable size chunks into fixed size objects and to append ECC [31]. Bhagawat et al. proposed to apply different levels of replication for each chunk [32]. They proposed to determine replication level based upon the amount of information loss when the respective chunk becomes unavailable.

Recently, Efstathopoulos et al. dealt with the issue of chunk garbage collection. They proposed a mechanism called grouped mark-and-Sweep [33].

The rest of this paper is organized as follows: Section 2 describes the organization of PRUNE. Section 3 describes the fingerprint lookup mechanism of PRUNE including index partitioning and the relationship between the false positive rate and the overall deduplication performance. Section 4 describes context-aware chunking. Section 5 presents the result of our experimental study, and Section 6 concludes the paper.

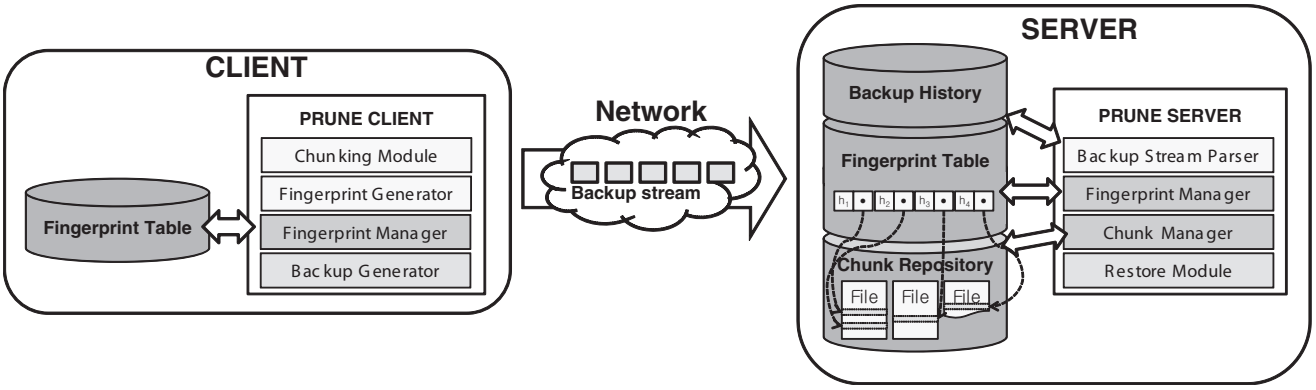

Fig. 2. System organization.

2 System Overview

2.1 System Organization

PRUNE(Prompt Redundancy Elimination) is designed for distributed environment where backups are located at a remote site.[1] We use the terms "client" and "server" for the location of the original data to be backed up and the location of the backup files, respectively. Deduplication consists of three components: chunking, fingerprint generation, and detection of redundancy.

There are four modules on the client side: Chunking Module, Fingerprint Generator, Fingerprint Manager, and Backup Generator. Chunking Module partitions a file into a number of fragments called chunks. For each chunk, PRUNE generates summary information to expedite the process of chunk comparison. Fingerprint Generator generates fingerprint for each chunk. Cryptographic hash function, e.g., SHA-1 [34], MD5 [35], or SHA-256 [34] can be used to generate the fingerprint. PRUNE currently uses SHA-1 as in [13]. It generates 160 bit fingerprint for each chunk. Fingerprint Manager is responsible for the "insert," "delete," and "search" of the fingerprints in Fingerprint Table. The efficient management of Fingerprint Table is one of the key factors of success in the deduplication system. In this work, we develop LRU-based index partitioning scheme in order to manage Fingerprint Table. We will delve into details of this in Section 3. Backup Generator is responsible to assemble a sequence of chunk information along with the appropriate metadata and to create backup object called Backup Stream. Backup Generator transfers backup data in header-data streaming fashion described in Fig. 4a.

There are four modules on the server side: Backup Stream Parser, Fingerprint Manager, Chunk Manager, and Restore Module. In the server, the Backup Stream Parser receives incoming byte stream and assembles them into the object called Backup History. Fingerprint Manager at the server is responsible for inserting a fingerprint to fingerprint database or finding the location of the chunk with a given fingerprint. The Chunk Manager at the backup server receives Chunk Data from the Backup Stream Parser and stores it in Chunk Repository. Chunk Manager returns the location of the chunk to the Fingerprint Manager. Fingerprint Manager maintains the chunk location for each fingerprint. The Restore Module in the server is responsible for constructing the Restore Stream(Fig. 4c) based upon the Backup History. Fig. 2 illustrates the overall system organization of PRUNE.

1. PRUNE at its very inception stage has been published at Won et al. [13]. PRUNE has gone through complete overhauling since then.

2.2 Chunking Module

Chunking is the operation of scanning a file and partitioning it into pieces. Each file piece is called a chunk and is a unit of redundancy detection. There are two types of chunking: fixed-size chunking and variable-size chunking. For fixed-size chunking, a file is partitioned into fixed size units, e.g., 8 KB blocks. Fixed-size chunking is conceptually simple and fast. However, this method has an important drawback: when a small amount of data is inserted into a file or deleted from a file, an entirely different set of chunks is generated from the updated file. To effectively address this problem, variable-size chunking, which is also known as content-based chunking, has been proposed [8].

For variable-size chunking, the chunking boundary is determined based on the content of Chunk Data, not on the offset from the beginning of the file. The Basic Sliding Window algorithm slides the window from the beginning of a file, one byte at a time (Fig. 3). A window is a byte stream of a given length. We generate a signature for the window and determine if it matches the predefined pattern called the "target pattern." If the signature matches the target pattern, the end position of a window is set at the end of the chunk. Otherwise, the window is shifted by one byte and the signature is generated again. Partitioning the file based on its content is a very CPU-demanding process [13]. Efficient chunking is one of the key ingredients that governs the overall deduplication performance.

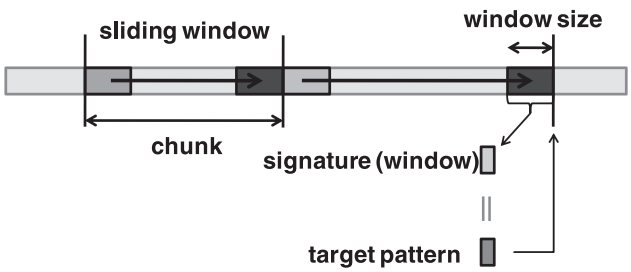

Fig. 3. Sliding Window algorithm.

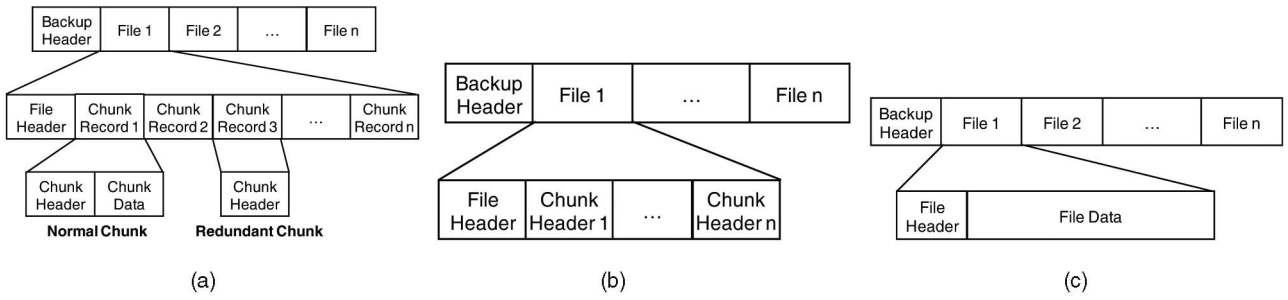

Fig. 4. Data structure of PRUNE. (a) Backup Stream, (b) Backup History, (c) Restore Stream.

Let us provide an example. We assume that the average chunk size was 8 KB. For a 10 MB file, there will then be 1,280 chunks. When the minimum chunk size is set to 2 KB, creating an 8 KB chunk implies that there exist approximately six thousand signature computations and comparisons. With an 8 KB average chunk size, the ratio between the number of chunks and the total number of signature operations (computations and comparisons) is 1:6,000. Henceforth, the number of signature computations is at least three orders of magnitudes larger than generating summary information of a chunk.

2.3 Fingerprint Generator

Fingerprint Generator generates the summary information for each chunk. We call is a fingerprint. To expedite the comparison, we examine the fingerprints of the chunks, instead of performing a byte-level comparison between two chunks. Some existing works [9], [8] state that the probability of a SHA-1 collision is much smaller than the hardware failure rate, and therefore we do not need to consider deduplication failure. Henson [36], however, stated that this probability should be used with discretion primarily because backup file lasts orders of magnitude longer than the hardware. Fingerprint Generator creates fingerprint for a chunk and passes the fingerprint to Fingerprint Manager.

2.4 Fingerprint Manager

Fingerprint Manager at the client and Fingerprint Manager at the server are responsible for detecting redundancy and for locating the respective chunk in Chunk Repository, respectively. Fingerprint Manager at the client is responsible for lookup and insert of the fingerprint to the existing set of fingerprints. In PRUNE, we maintain fingerprints as the collection of small fingerprint table called tablets. Fingerprint Manager maintains an array of pointers whose entry points to the individual tablets.

In the client, when a fingerprint is passed to Fingerprint Manager, it searches the list of tablets to see if a given fingerprint is redundant.

We use commodity DBMS(Berkeley DB [37]) for lookup and insert for each tablet. One table in the list is designated as *current*. If the fingerprint is new, it is inserted at the *current* tablet. If current table reaches predefined maximum tablet size, new tablet is created and the newly created tablet is set as "current."

In our system, Fingerprint Tables reside in both the client and the server. Fingerprint Table in the client contains only fingerprint values and is primarily used for duplicate detection. The server side Fingerprint Table is primarily used to locate the respective chunk data. The server side Fingerprint Table carries the fingerprint, the size of the chunk data, the reference count of the chunk that denotes the number of shared files, and the location of the chunk data that is represented by *<file id, offset>* of Chunk Repository, which is harbored by the server and consists of a number of chunk archives.

In the server, Fingerprint Manager examines fingerprint table and identifies the location of a given chunk based upon fingerprint.

2.5 Data structures

There are three important backup objects in PRUNE: Backup Stream, Backup History, and Restore Stream. Figures in Fig. 4 illustrate the organization of these data structures. Backup Stream is a sequence of bytes generated at the client. Backup Stream is generated by Backup Generator at the client and sent to server. Backup Stream consists of the Backup Header and a set of file information (Fig. 4a). Backup Header consists of Prune version and Backup requested time. The file information is composed of File Header and the array of chunk information. Each chunk information consists of Chunk Header, and Chunk Data. If a certain chunk is redundant, Chunk Data field is empty. Fingerprint of a chunk is stored at Chunk Header field. Chunk Type field of Chunk Data can be in one of three states: *original*, *compressed*, or *redundant*. When Chunking Module is informed that the given chunk is nonredundant, Chunking Module passes the chunk to Backup Generator. When a user starts backup session, the user specifies whether chunks are to be compressed or not. If compression option is specified, Chunking Module compresses each chunk data before passing it to Backup Generator. When Chunking Module is informed that a given chunk is redundant, Chunking Module passes only chunk metadata to Backup Generator. Backup Generator sets the value of Chunk type field when it creates Backup Stream. When a chunk is compressed, the original size and the size of the compressed chunk are different. Chunk Header carries fields for both of these sizes. Table 1 illustrates the details of Backup Header, File Header, and Chunk Header.

Backup server receives incoming packets, reassembles the packets into the object called Backup History. Fig. 4b illustrates the organization of Backup History. The structure of the Backup History is precisely identical to the Backup Stream except that Backup History does not have Chunk Data. When Chunk Data arrives at the server, Chunk Manager stores them at Chunk Repository. When an incoming Chunk Header is marked as redundant, server side Fingerprint Manager examines the fingerprint database

TABLE 1
Header Organization

Object	Field	Type
Backup Header	Prune version	string
	Backup requested time	int
File Header	Header size	int
	File mode	int
	Owner name	string
	Group name	string
	File size	int
	Creation time	int
	Access time	int
	Modified time	int
	Path	string
Chunk Header	Current chunk size	int
	Original chunk size	int
	Chunk type	int
	Fingerprint	char
	Chunk repository file name	string
	Chunk repository file offset	int

and returns the location of the respective chunk. Chunk Header of the Backup History is set to the location of the respective chunk.

Restore Stream is reconstructed based upon the Backup History. Restore Stream is created via replacing all Chunk Header in Backup History object with actual chunk. If chunk is stored at Chunk Repository in compressed form, it is first uncompressed and then put into Restore Stream. Restore Stream is built after referencing Backup History and consolidating one or more Chunk Data into integrated files. Usually Restore Stream is passed to the client and is used to restore files and directories. Fig. 4c illustrates the structure of the Restore Stream. Restore Stream has similar data structure with TAR.

3 LRU-Based Index Partitioning

3.1 Filter-Based Fingerprint Lookup

Detecting redundancy consists mainly of two tasks: chunking and fingerprint lookup. Chunking is a very CPU-demanding process. Fingerprint lookup is a key technical ingredient for efficient deduplication. Let us provide an example. Assume that the data size is 30 TB and that the average chunk size is 8 KB. There are then approximately 3.75 billion chunks. If we use a 160 bit SHA-1 fingerprint (20 B), the total number of fingerprints corresponds to 75 GB excluding any managerial overhead (such as a pointer, the fill factor of the database table, and an additional index table). Updating and searching the 75 GB table is a nontrivial operation especially when it needs to be performed online.

We use a Bloom filter to determine whether or not a given fingerprint exists in Fingerprint Table (Fig. 5). To estimate whether an element is in a set, many preceding works [38], [7], [24], [39] have used the memory-based filter developed by Bloom [23]. Consulting the Bloom filter does not entail disk traffic. It is a very effective method to reduce disk traffic for index lookup. It is particularly effective when most of the lookups result in positive results. A Bloom filter consists of k hashing functions, $h_1, \ldots, h_k$, and the bit array M of size m bits. With input x, the Bloom filter sets $M[h_i[x]] = 1$ for $i = 1, \ldots, k$. For a given input x, if $M[h_i[x]] == 1$ for $i = 1, \ldots, k$, x is already determined as

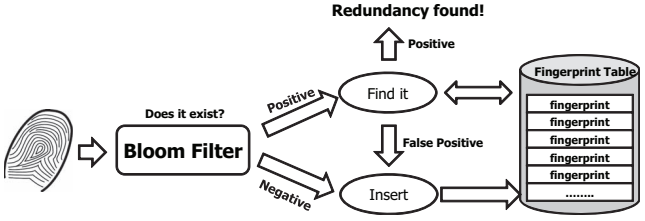

Fig. 5. Bloom filter for searching the fingerprint table.

positive, i.e., already exists. The Bloom filter resides in the main memory and consulting it does not cause any disk traffic. One concern of using the Bloom filter is the false positive ratio. It is possible that even though the Bloom filter identifies a fingerprint as existent, the fingerprint may not actually exist in Fingerprint Table. Reducing the false positive rate is an important issue in using the Bloom filter because a false positive triggers the search function of Fingerprint Table, which accesses the disk.

The false positive rate can be formulated as follows: let m, k, and n be the bit array size, the number of hash functions, and the number of fingerprints, respectively. Assume that the hash value is uniformly distributed over $[0 \ldots m-1]$ and hash functions $h_1, \ldots, h_k$ are independent. The probability that bit i is still 0 after inserting one fingerprint corresponds to $(1 - \frac{1}{M})^k$, and the probability that bit i is still 0 after inserting n fingerprints corresponds to $(1 - \frac{1}{M})^{kn}$. Therefore, when a new fingerprint arrives, the probability that its hash values refer to bits that are a priori set to 1 corresponds to $(1 - (1 - \frac{1}{M})^{kn})^k$, where n is the number of inserted elements. Given the number of fingerprints n, the false positive rate is subject to the number of hash functions k and the size of the Bloom filter m. The false positive rate is formulated as

$$\left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k \approx (1 - e^{-kn/m})^k. \quad (1)$$

As the Bloom filter size increases, the false positive ratio decreases. If the focus is placed on minimizing the false positive rate, the number of hash functions, k, which minimizes the false positive rate via differentiating it with k is obtained using a simple calculation, $k = (\ln 2)m/n \approx 0.7\, m/n$, with n and m being the number of inserted elements and the bit vector size, respectively. Further interested readers are referred to Horowitz et al. [40].

3.2 Tablet-Based Index Partitioning

Legacy search structures, e.g., the hash-based index and B^+ tree-based index, place the entry based on the "key value," not on the order in which the key values are inserted. This is because the value of adjacent fingerprints in the fingerprint lookup sequence will be uniformly distributed. With legacy search structure, e.g., B+ tree, or hash table, it is not likely that adjacent fingerprints in the lookup sequence are stored in clustered manner. Legacy index structures leaves much to be desired to properly exploit the workload characteristic of fingerprint lookup in deduplicate backup. The most brute-force way of preserving the access locality is to use append-only file structure in storing fingerprint. However, append-only file structure has $O(n)$ search time complexity and it cannot be used in practice.

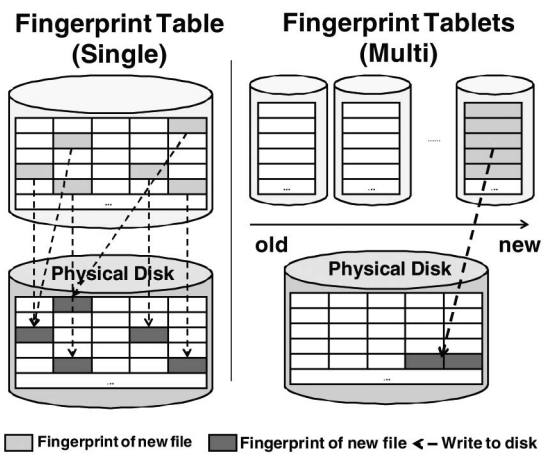

Fig. 6. Single index and index partitioning.

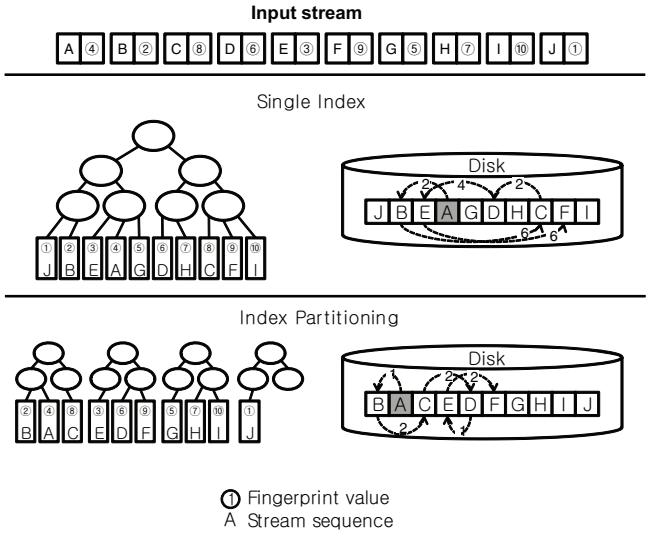

Fig. 7. Table versus tablets.

To effectively exploit the workload characteristics of fingerprint lookup, we develop an *LRU-based index partitioning* scheme. The basic idea is to form a fingerprint table with a number of smaller size "tablets." Fig. 6 illustrates two types of the fingerprint table structures: single large table and collection of small tablets. The physical distance between the fingerprints in a table(or in a tablet) is governed by the size of a table(or a tablet). For a single large table, fingerprints in the table may be far apart in the storage. With tablet-based approach, the physical distance between the fingerprints in a tablet is shorter. Via storing a sequence of fingerprints using tablets instead of single large table, we can store adjacent fingerprints in the incoming fingerprint lookup sequence within the relatively smaller region. Using the concept of tablet, we impose spatial locality in storing the fingerprints. The size of a tablet is small enough so that it can be loaded into the main memory. Determining the size of the tablet is of important concern. As we use smaller size tablets, the fingerprint lookups within a tablet will lie within a smaller region of the storage. However, there will be more number of tablets to examine.

Fig. 7 illustrates Fingerprint Table with a single large index tree and Fingerprint Table with multiple tablets, respectively. In Fig. 7, ten chunks are generated and are labeled from A to J in the order in which they are created. The number in each chunk denotes the fingerprint value of the respective chunk. When Fingerprint Table consists of a single large table (Fig. 7), the total seek distance to lookup the fingerprints of chunks A through F corresponds to 20. Next, we use the tablets with the maximum number of entries of three. There are four such tablets, each of which has a B+tree-based structure. A new tablet is created as fingerprints are inserted. When a set of fingerprints is maintained with four tablets, the total seek distance to determine the fingerprints for chunks A through F corresponds to eight blocks. The total seek distance is reduced when a set of fingerprints is managed with tablets. This is because the fingerprints can be clustered based on their lookup sequence.

3.3 LRU-Based Tablet Management

To reduce the number of tablets examined, we maintain the list of tablets in an LRU (least recently used) manner. When the fingerprint is found at a certain tablet, PRUNE moves the respective tablet to the head of linked list. We call it LRU-based tablet management because least recently used tablet will be located at the tail of the list. In this manner, we maintain temporal locality on fingerprint search. If the fingerprint of the next lookup resides at the same tablet as its preceding one, PRUNE can examine only one tablet to determine whether the fingerprint is redundant or not.

Fig. 8 illustrates three different fingerprint management schemes. Fig. 8a shows that a set of tablets are maintained simply as a linked list. It is termed as "Linear." The search always starts from the head of the linked list. Fig. 8b illustrates LRU-based tablet management. In Fig. 8b, the recently *hit* tablet is moved to the head of the list. When tablet 3 is hit, it is moved to the head of the list. To preserve this access correlation across the tablets, we enhance LRU-based management with prefetching (Fig. 8c). It is likely that the fingerprints in the adjacent tablets are accessed in consecutive manner. When a tablet is moved to the head of the list, we also move the successor tablet in the linked list together. We call this *prefetching*. When tablet 3 is hit, tablet 3 and tablet 4(successor of tablet 3) are moved to the head of the list. When prefetched table is "hit," same rule applies. It is moved to the head of the list along with its successor.

4 Context-Aware Chunking

4.1 Chunk Size Model

When designing the deduplication backup system, the most fundamental task of the initial phase is to establish the size of the target pattern of the variable-size chunking. The target pattern size governs the average chunk size and the amount of fingerprints to manage. The average chunk size governs the size of Fingerprint Table. With smaller chunks, it is

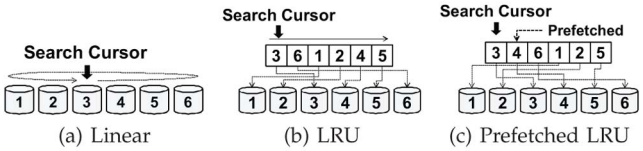

Fig. 8. LRU-based index partitioning.

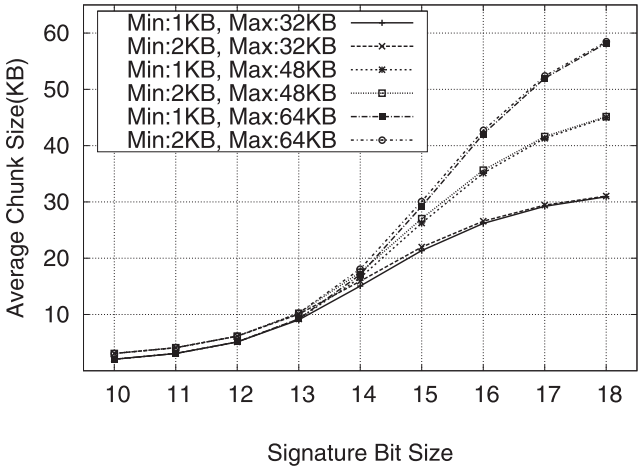

Fig. 9. Average chunk size under varying lower and upper bound of the chunk size.

easier to identify commonalities among files and to eliminate redundancy. However, as chunk size decreases, Fingerprint Table size becomes larger and the fingerprint lookup overhead increases.

We develop an analytic model that allows for accurate computation of the expected chunk size for a given target pattern size.

Most existing works assume that the average chunk size is determined based on the geometric distribution, i.e., with an n bit target pattern, and the average chunk size is assumed to be 2^n. In the sliding window protocol [8], most implementations establish the lower and upper bounds of the chunk size to avoid excessive chunking and indefinite chunk growth, respectively. With the minimum and maximum chunk size being 2 KB and 64 KB, respectively, the geometric distribution tends to underestimate the average chunk size. Let S_{min} and S_{max} denote the lower and upper bounds of the chunk size, respectively. Let p be the probability that the signature matches the target pattern. If the target pattern consists of n bits, then, $p = (\frac{1}{2})^n$. The average chunk size $E[S]$ can be computed as in (2). Let N be $S_{max} - S_{min}$.

$$\begin{aligned} E[S] &= S_{min} + \sum_{i=1}^{N} i\left(\frac{1}{2^n}\right)\left(1-\frac{1}{2^n}\right)^{i-1} \\ &\quad + N\left(1 - \sum_{i=1}^{N}\left(\frac{1}{2^n}\right)\left(1-\frac{1}{2^n}\right)^{i-1}\right) \qquad (2) \\ &= S_{min} + 2^n\left(1-\left(1-\frac{1}{2^n}\right)^N\right). \end{aligned}$$

For a 13-bit target pattern, the average chunk sizes obtained from (2) and from the geometric distribution corresponds to 10 KB and 8 KB, respectively. Fig. 9 plots the average chunk size based on (2). We use two minimum chunk sizes, 1 and 2 KB and the maximum chunk sizes, 32, 48, and 64 KB. The X and Y axes of Fig. 9 denote the target pattern size and average chunk size, respectively. As can be seen, the average chunk size was not sensitive to the minimum chunk sizes of 1 KB or 2 KB.

We chunk a set of files with varying target pattern size and examine the accuracy of the chunk size model developed in this work. The lower and upper bounds of the chunk size are set to 2 KB and 64 KB, respectively. Table 2 illustrates the quantile statistics of the chunk size distribution, the mean chunk size obtained from our analytical model, $E(S)$ and mean chunk size obtained from geometric model, $E(geo)$. The sample means were approximately 4 KB, 6 KB, and 10 KB for the 11 bit, 12 bit, and 13 bit target pattern size, respectively. Our model precisely estimates the average chunk size within the one percent error.

TABLE 2
Chunk Size: Quantile Statistics of Sample Distribution, Mean from Chunk Size Model and Mean from Geometric Model

	Chunk size (KByte)					
Bit	25%	50%	Mean	75%	E[S]	E[geo]
10	2.3	2.7	3.0	3.4	3.0	1.0
11	2.6	3.4	4.0	4.8	4.0	2.0
12	3.1	4.8	6.1	7.6	6.0	4.0
13	4.4	7.6	10.1	13.1	10.0	8.0
14	6.9	13.5	17.8	24.3	17.7	16.0
15	11.9	24.7	29.6	46.2	29.4	32.0

4.2 Incremental Modulo-K

In variable-size chunking, we generate the signature for the byte sequence in a given window and determine if the signature matches a given target pattern. Most existing works [39], [41], [42], [43], [8] have used Rabin's algorithm for generating a signature. Rabin's signature generation algorithm for the byte sequence $b_1, \ldots, b_\beta$ is defined as

$$Rf(b_1, b_2, \ldots, b_\beta) = (b_1 p^{\beta-1} + b_2 p^{\beta-2} + \cdots + b_\beta) \bmod M,$$

where β and M are irreducible polynomials. For a given byte string $b_1, \ldots, b_N$, we obtain the signature for the substrings $\{b_1, \ldots, b_\beta\}, \{b_2, \ldots, b_{\beta+1}\}, \{b_3, \ldots, b_{\beta+2}\}, \ldots$. The main advantage of Rabin's algorithm over other hashing functions, e.g., SHA-1, SHA-2, and MD-5, is its ability to compute the signature in an incremental fashion. For each substring, the signature value can be incrementally obtained from the previous value, using

$$\begin{aligned} &Rf(b_i, \ldots, b_{i+\beta-1}) = \\ &\quad \left((Rf(b_{i-1}, \ldots, b_{i+\beta-2}) - b_{i-1}\, p^{\beta-1})p + b_{i+\beta-1}\right) \bmod M. \end{aligned} \qquad (3)$$

Rabin's algorithm is very efficient. However, since there exists an excessive number of signature computations, it is mandatory to make the signature computations more efficient.

We exploit the algebraic nature of the modulo arithmetic and improve the chunking overhead. The following are three basic properties of the modulo arithmetic:

1. $(a + b) \bmod M = ((a \bmod M) + (b \bmod M)) \bmod M$,
2. $(a \times b) \bmod M = ((a \bmod M) \times (b \bmod M)) \bmod M$, and
3. $a \bmod M = a$ if $a < m$.

In (3), modulo M is computationally expensive and important to effectively optimize. Usually, M is chosen as a prime number and M is set at $2^{31} - 1$ in our system. Also, p and β in (3) correspond to the base and window

sizes of Fig. 3, respectively. We set the window size β to 48 as in [8] and p corresponds to 2^8. Rabin's algorithm in (3) deals with a 48-byte string. In our system, a 48-byte string is represented as an array of six integers of `unsigned long long` (8 bytes). There are two tasks we want to achieve: 1) reduce the number of modulo operations and 2) replace the modulo operation with cheaper operations, e.g., `SHIFT` and `XOR`. For the $N\ mod\ M$ operation with N and M being 64 bits (`unsigned long long`) and 32 bits (`unsigned long`), respectively, there are thirty-two `XOR` operations. We reduce the overhead of the modulo operation by exploiting its algebraic nature. Let us represent N with the bit string $N = p_{63} \dots p_0$. We partition this 64-bit pattern $(p_{63} \dots p_0)$ into three parts: $P_2 = p_{63}p_{62}$, $P_1 = p_{61} \dots p_{31}$, and $P_0 = p_{30} \dots p_0$. Then, the original bit string N can be represented as $N = P_2 \times 2^{62} + p_1 \times 2^{31} + P_0$. Given $2^{31}\ mod\ (2^{31} - 1) = 1$, $N\ mod\ M$ now can be computed as in

$$N\ mod\ M = (P_2 + P_1 + P_0)\ mod\ M, \text{ where } M = 2^{31} - 1. \tag{4}$$

Equation (4) uses two additions and one or two XOR's. For generating signature for 48 Byte string, Rabin's algorithm uses 32 XOR's whereas INC-K algorithm uses 4 ADD' and two XOR's. We significantly decrease the computational overhead of the generating signature.

4.3 Context-Aware Chunking

Variable-size chunking is less effective when dealing with multimedia content, compressed files, or encrypted content. In these type of files, minor modification on the original file entirely changes the final form. For these files, it is likely that two files are either identical or entirely different. Therefore, applying computationally expensive variable-size chunking to compressed and encrypted file may not be worth its effort. For text files, documents, source codes, etc., only a small portion of the data gets updated. Exploiting this property, context-aware chunking uses two knobs to tune a chunking file. The first knob deals with the chunking method. Context-aware chunking applies either to variable-size or fixed-size chunking based on certain criteria. The notion of context-aware chunking proposed in this work shares the basic idea with the preceding works [18], [19], [20].

We categorized the files into two sets: $\mathcal{IM}$ (immutable) and $\mathcal{M}$ (mutable). We apply simple static rule in categorizing the files. We examine the extension of a file and determine the category. Multimedia contents, e.g., $\{^*.\text{avi}, ^*.\text{wmv}, ^*.\text{mkv}, ^*.\text{mp3}, ^*.\text{mp4}, ^*.\text{jpg}, ^*.\text{png}\}$, and compressed files, e.g., $\{^*.\text{tar.gz}, ^*.\text{tgz}, ^*.\text{zip}, ^*.\text{rar}, ^*.\text{lzh}, ^*.\text{alz}, ^*.\text{bzip}\}$, are categorized as immutable. Files with other extensions are categorized as mutable. There can be an exception to this rule. It is possible that video file may be updated due to video editing operation. Also, Linux source code may be immutable since people rarely edit the Operating System source code. Further elaborate data and user behavior study is required to determine whether a given file is immutable or not. We plan to address this issue in separate context.

The second knob in context-aware chunking deals with compression. One of the prime objectives of deduplication is to reduce the data volume to save either storage space or network bandwidth. Compressing a chunk is a viable option to reduce the data volume. For multimedia data types, e.g., video clips, photographs, and mp3 music, compressing a chunk may not bring significant reduction to the data volume. PRUNE allows the system administrator to choose these options in a flexible manner. Fig. 10 schematically illustrates the context-aware chunking mechanism.

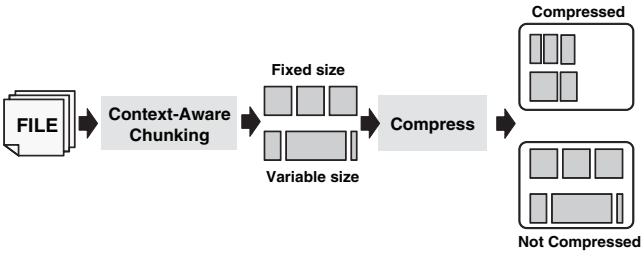

Fig. 10. Context-aware chunking.

5 Experiment

5.1 Experiment Setup

We develop a prototype backup system and examine the performances of the techniques developed in this work. We perform comprehensive analysis on various aspects of deduplication backup:

1. Performance of Incremental Modulo-K algorithm,
2. chunk Size versus Deduplication Ratio,
3. effect of Tablet Size over Backup Speed,
4. performance of Different Chunking Algorithms,
5. effect of Locality in Backup Speed, and
6. effect of Context Aware Chunking.

We use 27 different tablet sizes, four different tablet management schemes, eight different chunking methods, and three different target bit pattern sizes. Data set used in each experiment is chosen to effectively achieve the objective of the respective experiment.

Our experiment is performed on AMD Phenom (Quad-Core 2.2 GHz) with 2 GB of RAM. We maintain fingerprints and chunks in separate storage devices. Fingerprints are stored on one hard disk drive (WD Raptor, 10,000 RPM, 74 GB). Chunks and data files are stored on a disk array with two disks (Seagate Barracuda, 7,200 RPM, 1 TB each). The maximum bandwidth of this disk array was 140 MB/sec.

5.2 Performance of Incremental Modulo-K

This experiment focuses on evaluating the algebraic efficiency of two algorithms: Rabin's algorithm and INC-K. We use two data sets and measure the speed of chunking under six target pattern sizes. To manifest the algorithmic efficiencies of two algorithms, we load the files in main memory. Table 3 summarizes the data sets used in the experiment. First data set consists of a single 700 MB multimedia file. Second data set consists of 23,810 files. Total size is 253 MB. Mean and median of file size is 11.2 KB and 4.2 KB, respectively.

Our chunking mechanism skips the minimum size of chunk before it starts signature computation. For the multimedia file, the minimum size chunks are rarely generated, and the number of signature computations was proportional

TABLE 3
Data set for Chunking Speed Experiment

Dataset	Type	Total(MB)	# of files	Avg	Median	Min	Max
Multimedia	AVI file	700.46	1				
Linux kernel source(2.6.25)	Source code	253.19	23,810	11,150	4,198	6	875,265

to the size of the files. For the Linux kernel source, most of the source code files are less than a few kilobytes, and the number of files governs the number of computations. This is the main reason why the single large multimedia data set tends to perform more signature computations.

Fig. 11 illustrates the result. X-axis and Y-axis denote the target pattern size and chunking speed. With 10 bit target pattern, chunking speed for Linux Source code corresponds to 250 MB/sec and 227 MB/sec for INC-K and Rabin's algorithm, respectively. With 14 bit target pattern, chunking speed for Linux Source code corresponds to 102 MB/sec and 97 MB/sec for INC-K and Rabin's algorithm, respectively. This is because the number of signature computation is linearly proportional to the chunk size and chunking speed is inversely proportional to the number of signature computation.

Three important results should be noted. First, the chunking speed with Linux source data set is higher than the chunking speed with large multimedia data set. For 10 bits, we obtained 250 MB/sec for the performance of Linux source data set whereas 238 MB/sec is observed in single large file chunking. Second, the Incremental Modulo-K algorithm is faster than Rabin's fingerprint algorithm by 10-15 percent for both data sets. Third, the signature generation speed became significantly slower as we increased the size of the target pattern. For 10 bits, we were able to obtain 250 MB/sec and 238 MB/sec for the Linux kernel source and the multimedia data set, respectively. For 12 bits, the signature generation speed becomes 122 MB/sec for both data sets. Be reminded that the entire file is in the memory. The larger target pattern results in larger chunks, and therefore will reduce the fingerprint lookup overhead. However, as can be seen in Fig. 11, it negatively affects the chunking speed.

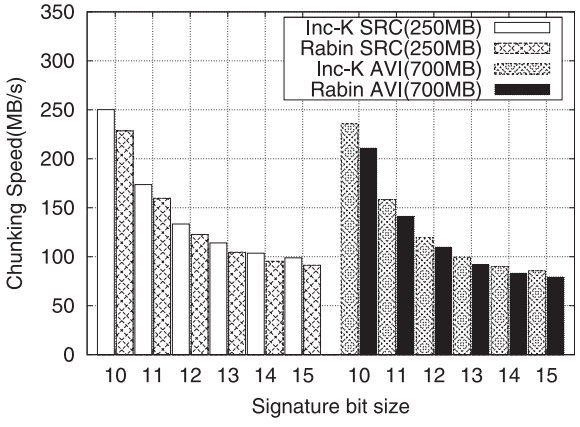

Fig. 11. Signature generation.

5.3 Average Chunk Size versus Redundancy Detection Rate

We examine the degree of duplication detection for 11-bit, 12-bit, and 13-bit target patterns. With 11 bit, 12 bit and 13 bit target pattern, average chunk size corresponds to 4.0 KB, 6.0 KB and 10.1 KB, respectively. We use the data sets shown in Table 4. We run ten rounds of backup and examine the deduplication ratio and the number of chunks for each round. At the first round, PRUNE performs deduplicated backup for 10 GB of data. Then, new 10 GB of data are added and PRUNE is run again. We repeat this process until the total data volume reaches 100 GB.

Fig. 12 illustrates the numbers of chunks generated and the fractions of duplicated chunks subject to the varying target pattern size. Fig. 12a illustrates the number of newly created chunks for each round of backup. The number of chunks generated by variable size chunking increases by a factor of 2.5 when we use the 11-bit target pattern instead of the 13-bit pattern. For each X value, there are three bars each of which denotes the number of chunks for the respective target pattern size. We can observe that the number of chunks decreases inversely proportional to the chunk size. Let us examine the deduplication ratio under varying target pattern sizes. Fig. 12b illustrates the results. Y-axis denotes deduplication ratio. As can be seen, the difference in deduplication ratios, with respect to target pattern sizes, are not significant and practically negligible. From 11 to 13 bits, the duplication detection rate decreases by only 0.66 percent. Using 11 bit pattern instead of 13 bit pattern, the number of fingerprints increases by a factor of 2.5 and deduplication ratio improves by 0.66 percent. We find that with 11 bit pattern size, chunking speed decreases by 33.9 percent (Fig. 11). However, overall deduplication speed becomes faster as we use larger target pattern size (Fig. 14). The 0.66 percent improvement in deduplication ratio does not offset the 2.5x increase in the number of fingerprints and subsequent increase management overhead.

5.4 Tablet Size versus Backup Speed

We examine the backup speed under varying tablet sizes through physical experiment. Figures in Fig. 13 illustrate the backup speed and fingerprint lookup latency under varying tablet sizes. We use the 100 GB data set in Table 4. In this experiment, backup speed reaches the peak when tablet size is 1.5 MB. Backup speed becomes slower when tablet size is bigger than 1.5 MB. The trend becomes clearer in the second backup round (Fig. 13a).

In our experiment, the second backup is slower than the first one under most tablet sizes. This is because in the first backup, fingerprint database(table or collection of tablets) is initially empty and fingerprint lookup overhead is not significant. In the second backup, PRUNE examines the tablets which harbor the fingerprints inserted during

TABLE 4
Backup Data Set

	Type	Size(KB)	%	# of files	Avg(KB)	Median(KB)	Min	Max(KB)	# of files
1st	iso	10,443,954	98.64	4	622,806	13,606	31	7,174,976	17
	exe	85,125	0.80	4					
	zip	58,540	0.55	2					
	etc	86	0.00	7					
2nd	iso	5,385,864	49.15	3	6,661	205	29	4,015,684	1,645
	zip	2,975,948	27.16	6					
	avi	1,576,199	14.39	53					
	jpg	363,149	3.31	1,388					
	etc	655,936	5.99	195					
3rd	iso	10,828,307	99.86	12	328,581	415	29	3,785,554	33
	jpg	7,567	0.07	18					
	zip	7,298	0.07	1					
	etc	6	0.00	2					
4th	zip	7,733,254	74.01	294	1,969	10	23	1,859,254	5,308
	iso	1,287,538	12.32	7					
	exe	917,158	8.78	51					
	etc	511,194	4.89	4,956					
5th	avi	4,935,219	47.31	15	750	2	1	787,047	13,904
	zip	2,409,503	23.10	99					
	jpg	1,526,885	14.64	2,834					
	mp3	1,084,786	10.40	57					
	etc	475,314	4.56	10,899					
6th	mp3	5,502,683	51.18	1,048	8,220	4,432	63	442,848	1,308
	zip	5,198,443	48.35	80					
	etc	50,478	0.47	180					
7th	pdf	4,235,124	38.93	346	10,491	2,516	9	333,614	1,037
	zip	2,799,575	25.73	150					
	chm	2,263,578	20.81	311					
	iso	1,527,420	14.04	7					
	etc	53,961	0.50	223					
8th	avi	10,527,222	100.00	36	214,850	239,166	30,637	627,052	49
	etc	435	0.00	13					
9th	avi	11,129,970	85.14	49	17,477	2,491	6,144	568,530	748
	jpg	1,686,879	12.90	695					
	zip	221,773	1.70	1					
	etc	34,344	0.26	3					
10th	avi	10,005,304	100	29	345,010	357,794	238,242K	717,276	29
Total		108,506,020	100	24,078	4,506	8,324	1	7,174,976	24,078

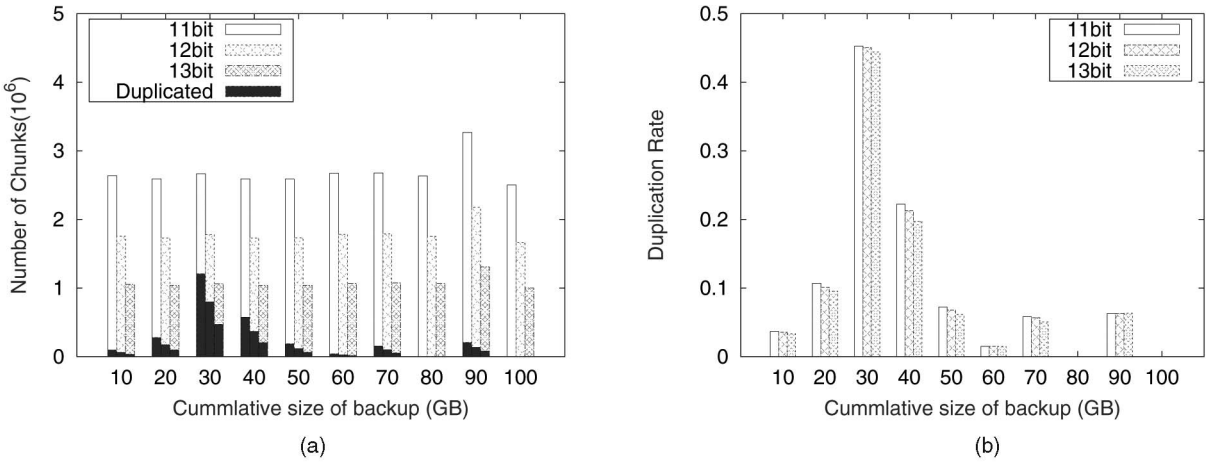

Fig. 12. Effect of target pattern size over Deduplication Ratio. (a) Number of chunks. (b) Fraction of redundancy.

the first backup. Fingerprint lookup overhead becomes more significant and as a result, backup becomes slower in the second round.

Fig. 13b illustrates the relationship between lookup latency and tablet size. "Lookup latency" denotes the average latency of looking up one fingerprint. In the first backup, lookup latency gets longer with the tablet size. Most of the fingerprints generated at the first backup are *nonredundant* and cause `insert` operation. In the second backup, lookup latency is more sensitive to tablet size and quickly increases as tablet becomes larger. Since the sequence of incoming fingerprints, i.e., SHA-1 hash value, does not exhibit any spatial locality when it is stored at the table, consecutive fingerprint lookups can yield completely random read on a table(or a tablet). As tablet becomes larger, random read operation on a tablet entails more significant disk head

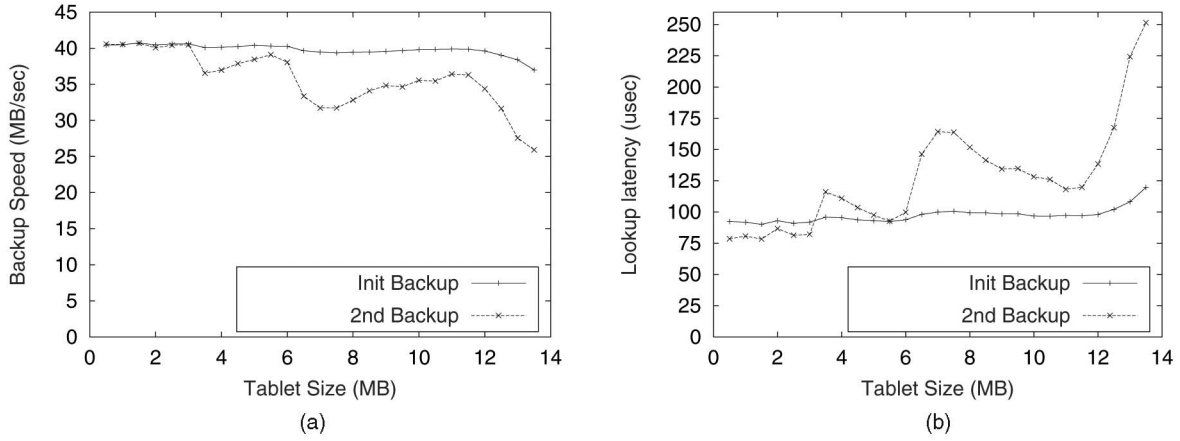

Fig. 13. Tablet Size versus Backup Performance. (a) Tablet size versus Backup Speed. (b) Tablet size versus Lookup latency.

movement. With 1.5 MB size tablet, fingerprint lookup latency is 90 usec and 78 usec in the first backup and the second backup, respectively. With 13.5 MB size tablet, lookup latency corresponds to 119 usec and 251 usec, respectively. These experiments(Figs. 13a and 13b) illustrate the importance of properly clustering the fingerprints.

We observe sinusoidal pattern on backup speed in Figs. 13a and 13b. It takes approximately 40 hours to run single set of experiments and we repeat same set of experiments multiple times. This sinusoidal trend persists across different sets of experiments. Detailed analysis of the graph requires in depth examination on internal behavior, e.g., buffer cache replacement, and swap page management, of Operating System and DBMS. We do not delve into details since it is beyond the scope of this paper. Based on the result of this experiment, rest of the experiments uses tablet size of 1.5 MB.

We finally measure the backup speed under varying target pattern size.

5.5 Fingerprint Management Methods

We compare the performance of the four different fingerprint management schemes: single index, index partitioning with simple linked list (Linear), LRU-based index partitioning, and LRU-based index partitioning with prefetching. In "Single" method, all fingerprints are managed as one database table. In "Linear," fingerprints are managed as the linked list of tablets. "LRU" and "Prefetch" maintain the tablet list with LRU and LRU with prefetching, respectively. We perform backup on the same data set (100 GB Data set in Table 4) twice. To effectively examine the lookup efficiencies of the algorithms, we need to avoid the *insert* operation in the second backup. For this reason, we use the identical data set in the first and the second backup. We use 27 different tablet sizes and measure the fingerprint lookup latencies at the first backup and the second backup.

Fig. 15 illustrates the result. "Single" yields the worst performance. Backup speed is 2.8 MB/sec and 3.3 MB/sec for initial backup and second backup, respectively. In the initial backup, Fingerprint Table is initially empty and all fingerprint lookup results in "insert" operation. Backup speed is slower in the initial backup than in the second backup when we use "single." This is because the single index scheme has a B+tree structure which has significant tree reorganization overhead.

In "Linear," the speed of the initial and the second backup correspond to 76.9 MB/sec and 42.9 MB/sec, respectively. In "Linear," the speed of backup decreases significantly in the second backup. Each of the tablets are relatively small, and therefore the overhead of inserting an entry to a tablet is much smaller than the overhead of inserting an entry to large size table. With tablet based approach, we were able to

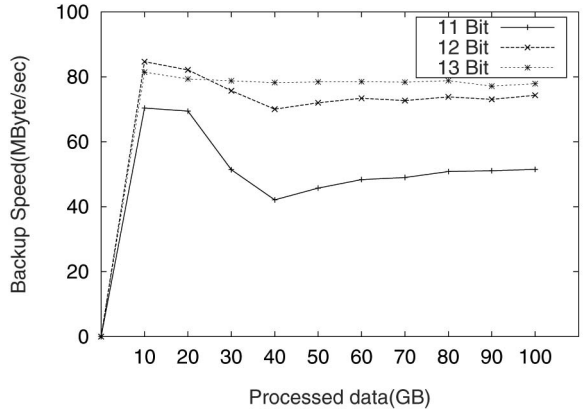

Fig. 14. Target pattern size versus backup speed.

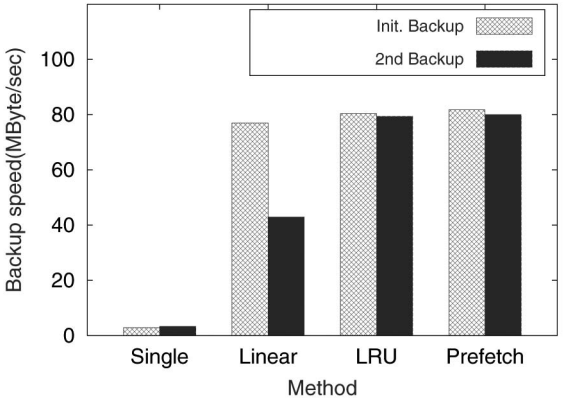

Fig. 15. Performance of tablet management methods.

achieve fast backup speed, 76.9 MB/sec in the initial backup. The second backup yields slightly different result. In the second round backup, each fingerprint lookup examines a number of fingerprint tablets (all tablets in worst case) and backup speed becomes much slower. In our experiment, the speed of second backup (42.9 MB/sec) is 60 percent of the speed of the initial backup (76.9 MB/sec).

Enhancing the index partitioning with LRU effectively resolved the speed decrease problem in the second backup. With LRU, the speed of the initial backup is the same as the speed of the initial backup with "Linear." However, LRU manifests itself in the second backup. In LRU, speed of the second backup yields 79.3 MB/sec which is on par with the speed of the first round backup, 80.3 MB/sec. We confirmed that LRU-based index partitioning effectively incorporates the access locality of the fingerprint lookup in managing tablets. We go one step further and add prefetching mechanism in LRU-based index partitioning. This is to reorganize the tablet list properly incorporating the access correlation across the tablets. According to our experiment, the effect of "prefetching" is not significant.

Let us compare the backup speeds of four fingerprint management schemes. The speed of the initial backups correspond to 2.8 MB/sec, 76.9 MB/sec, 80.3 MB/sec, and 81.7 MB/sec for "Single," "Linear," "LRU," and "LRU with prefetching," respectively. "Single" yields the worst performance while the rest of the three yield similar performance. The speed of the second backup correspond to 3.3 MB/sec, 42.9 MB/sec, 79.3 MB/sec, and 80 MB/sec, respectively. In managing fingerprints with tablets, we increase the backup speed by 14 times from 3.3 to 42.9 MB/sec. With *tablet*, we successfully clustered the fingerprints incorporating the order they are looked up. Via introducing LRU based tablet management scheme, we further increase the backup speed by 84.8 percent from 42.9 MB/sec with "Linear" to 79.3 MB/sec in "LRU." With pretching, backup speed increases by 0.8 percent from "LRU" method.

5.6 Effectiveness of Enforcing Locality

We measure how many tablets are accessed in each fingerprint lookup and analyze the relationship between the backup speed and the average number of tablet accesses per fingerprint lookup. We perform deduplication on the data sets in Table 4. There are ten backup operations. At each round, we add approximately 10 GB of new data. The dashed and dotted lines in Fig. 16 illustrate the deduplication bandwidth of the single index and LRU-based index partitioning schemes, respectively. When there are a small number of fingerprints in Fingerprint Table, the use of the single index or the index partitioning scheme does not make much difference. However, as the total number of fingerprints increases, the performance difference between the two becomes significant. The single index scheme does not scale well. With index partitioning, the backup performance does not decrease significantly as Fingerprint Table size increases. In the first backup, both the single index-based approach and the index partitioning approach yield a 80 MB/sec backup speed. In the tenth backup section, the total file size for the backup reaches 100 GB. With LRU-based index partitioning, the backup speed reaches 79 MB/sec. However, with the single index approach, the backup speed decreases to 4 MB/sec due to the excessive fingerprint lookup overhead.

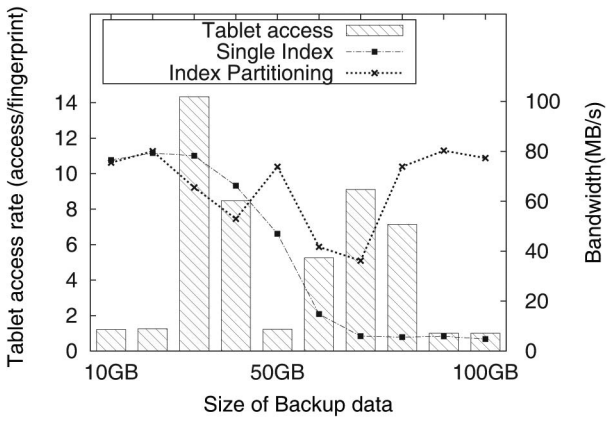

Fig. 16. Single Index versus Index Partitioning.

We observe in Fig. 16 that for LRU-based index partitioning, the deduplication performance varies widely for each round. The total file size of the backups in the second round and in the fourth round are 20 GB and 40 GB, respectively. The backup speed decreases from 80 MB/sec at the second round to 53 MB/sec at the fourth round (40 GB). At the fifth round, the backup speed again increases to 79 MB/sec. In the sixth and seventh round, the backup speed decreases to 41 MB/sec and 39 MB/sec, respectively.

The bar graph in Fig. 16 explains this phenomenon. The figure indicates the number of tablets accessed for a single fingerprint lookup. In the first and second backups, the tablet access rate is one, i.e., PRUNE accesses one tablet for one fingerprint lookup. For sixth and seventh backup, tablet access rates are 5 and 9, respectively. For these rounds, the backup speeds are among the slowest, 41 MB/sec and 39 MB/sec, respectively. This is approximately 50 percent backup speed of the first and the second backup. In the backup rounds of 1,2,5,9, and 10, backup speeds are the highest with approximately 80 MB/sec. For these sessions, tablet access rate is the lowest, 1. On the other hand, in backup session 7, backup speed is the slowest (39 MB/sec) and tablet access rate is 9 which is the second highest value. Backup session 3 exhibits contradictory behavior. Tablet access rate is 14 for backup session 3, but the backup speed is still reasonable, 63 MB/sec. We carefully conjecture that this is because there is only 30 GB of data in the third backup session and most of the tablets can be fully loaded onto the main memory. The tablet access does not incur disk accesses, and therefore the backup speed does not decrease significantly even though tablet access rate is high. Correlation Coefficient[2] between the tablet size and the number of tablet accesses per lookup corresponds to -0.62. The overall backup speed is governed by the number of tablets accessed in a single fingerprint lookup. There is strong negative correlation between the number of tablet accesses per fingerprint lookup and the backup speed.

5.7 Effect of Context Aware Chunking

We examine the effect of different chunking algorithms in overall deduplication efficiency. different chunking algorithms. The efficiency of the deduplication is measured in two aspects: backup speed and the size of resulting deduplicated backup. There are four chunking algorithms:

2. $\rho_{X,Y} = \frac{E[(X-\mu_X)(Y-\mu_Y)]}{\sigma_{X,Y}}$.

TABLE 5
Data Set for Comparison of Different Chunking Methods

			Size (KByte)				
Dataset	# of files	Total(MB)	Avg(KB)	Median(KB)	Min	Max(KB)	Description
Linux kernel source	475,997	5,157.66	11	4	0	1,725	2.6.0 to 2.6.25
Multimedia	29	9,770.81	345,010	357,794	29	717,276	
Backup data	477,872	38,926.02	83	4	0	475,996	Multimedia, Photo, Source, etc

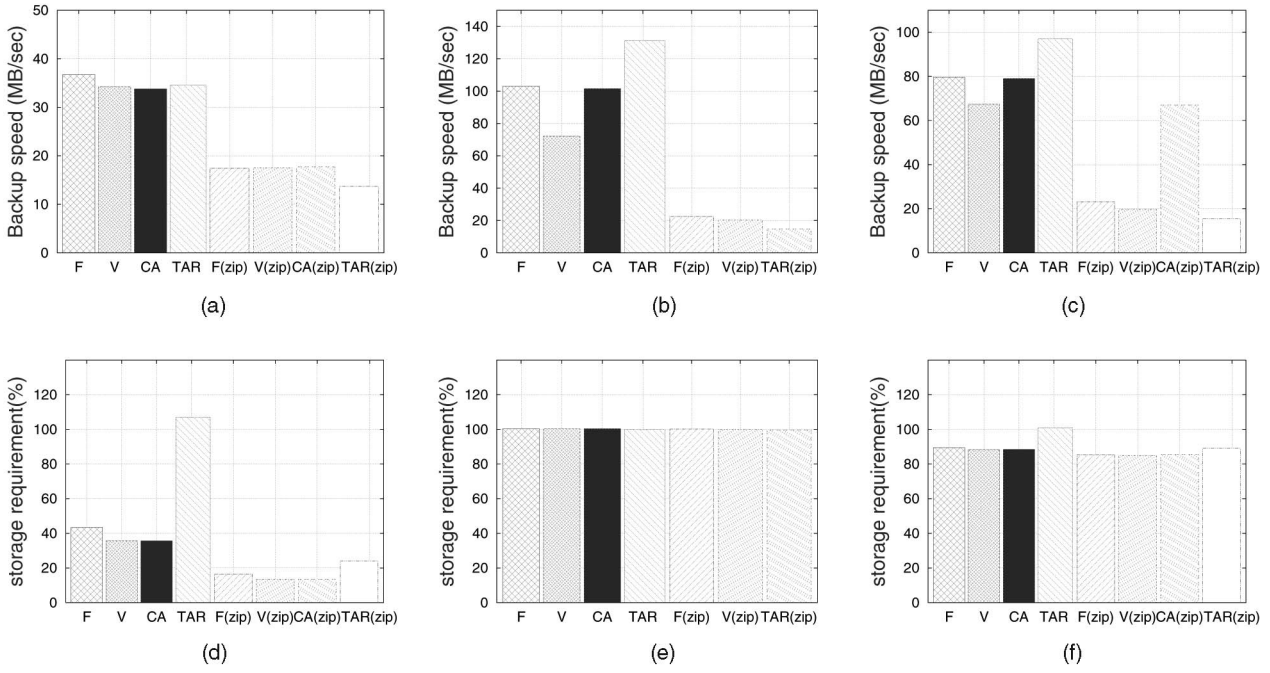

Fig. 17. Performance of different chunking methods: bandwidth versus data volume (F: fixed-size chunking, V: variable-size chunking, CA: context-aware chunking). (a) Linux source:backup bandwidth. (b) Multimedia: backup bandwidth. (c) Mixed data: backup bandwidth. (d) Linux source: backup capacity. (e) Multimedia: backup capacity. (f) Mixed data: backup capacity.

fixed size chunking, variable size chunking, context-aware chunking and no-chunking. Nochunking corresponds to legacy *tar* command. Each of these chunking algorithms has two bifurcations: plain and compressed. Combined altogether, we compare total eight chunking methods: Fixed Size Chunking(F), Variable size chunking(V), Content-Aware Chunking(CA), No Chunking(TAR), Fixed Size Chunking with compression(F-zip), Variable size chunking with compression(V-zip), Content-Aware Chunking with compression(CA-zip), and No Chunking with compression(TAR-zip).

For the comprehensive study, we use three data sets with different characteristics: 1) source code of Linux Operating System, 2) multimedia files and 3) mixture of multimedia files and source codes. We summarized the characteristics of the data sets in Table 5.

5.7.1 Linux Sources

The first data set is Linux kernel sources from 2.6.0 to 2.6.25. The total size is 5.1 GB. Files are usually small with average size of 11 KB. There are 475,997 files. Given that average chunk size is 10 KB, a single file is partitioned into two chunks on the average. Fig. 17 illustrates the results of experiment. As can be seen in Fig. 17a, fixed-size chunking yields the best backup speed, among eight chunking methods. However, it only yields 36.7 MB/sec. This is only 26 percent of the sequential IO bandwidth. All four chunking methods which do not have compression option yield similar backup speed from 34 to 36 MB/sec. Fixed-size chunking and variable-size chunking works at 36.7 MB/sec and 34.2 MB/sec, respectively. Fixed-size chunking is 7.6 percent faster than variable-size chunking. Context-aware chunking applies variable-size chunking to a text file which is classified as mutable, so the performance of context-aware chunking is close to the performance of variable-size chunking for the Linux source set. TAR option, i.e., no chunking, yields 34.5 MB/sec. When we add a compression phase to each chunking method, the chunking speed is reduced to 17.5 MB/sec for all three chunking schemes. By adding a compression phase, the backup bandwidth decreases by 50 percent.

TAR generates a 5.4 GB backup and the size of the final result increases by 5.8 percent (Fig. 17d). This is because TAR adds metadata at the beginning. Fixed-size chunking and variable-size chunking reduce the data from 5.1 GB to 2.2 GB and 1.8 GB, respectively. 57 percent and 65 percent of the storage requirement are eliminated for fixed-size chunking and variable-size chunking, respectively. Because

files in Linux source are classified as mutable, context-aware chunking scheme reduces 65 percent of the storage requirement which is almost identical to variable-size chunking scheme. Via compression, 84 percent, 88 percent, and 88 percent of the storage requirement are eliminated for fixed-size chunking, variable-size chunking, and context-aware chunking, respectively.

When we backup a set of small files, the overhead of accessing file metadata, i.e., *open(), close()*, is dominant in overall backup operation. Also, since a file consists of few number of chunks, the overhead of chunking, i.e., the overhead of generating signature, is insignificant. When we backup small files, backup speed does not vary significantly subject to the chunking schemes. Via deduplication, however, we can significantly reduce the storage requirement. Compression is not a viable option since backup speed decreases to half when we compress the chunks. Use of dedicated compression hardware can be a resolution to this. Although context-aware chunking does not yields neither the fastest backup speed nor the smallest storage requirement, the backup speed of context-aware chunking is almost identical to the variable chunking scheme which shows the best performance in backup speed and the efficiency of deduplication is also nearly same as variable chunking.

5.7.2 Multimedia Files

There are a total of 29 files and the average file size is 345 MB. The total file size sums up to 9.54 GB. Each file contains video data encoded by DivX codec. In multimedia files, the overall backup speed varies widely subject to the chunking method. TAR yields 131.1 MB/sec. Fixed-size chunking and variable-size chunking yield 103 and 72.2 MB/sec, respectively (Fig. 17b). Since we statically classify multimedia data as immutable, context-aware chunking applies fixed-size chunking. The backup speed difference between the fixed-size chunking and variable-size chunking comes from the overhead of generating signature for each byte position in the variable-size chunking. When the data set consists of small files (Linux sources), the overhead of open/close, i.e., accessing and manipulating file metadata, constitutes dominant portion of overall chunking performance, and the overhead of generating signature is insignificant. However, with multimedia files which tends to be large, the overhead of open/close is insignificant. Instead, the overhead of signature generation constitutes dominant portion of overall chunking performance. With compression, the overall backup speed decreases significantly. The backup speeds of fixed-size chunking scheme and variable-size chunking scheme decrease from 103 to 22.4 MB/sec and from 72.2 to 20.2 MB/sec, respectively.

Fig. 17e illustrates the result of the backup size in multimedia data set. The total backup sizes from the four chunking schemes are almost identical to the original data set size. There are no deduplication performance gain in multimedia data set. Compression does not reduce the backup size for neither fixed size chunking nor variable size chunking.

5.7.3 Mixed Set

We use a mixture of different type files: multimedia, photo, document, and source files. In this set, there were 477,872 files with an average file size of 83 KB. In this data set, TAR yields the best backup speed (96.9 MB/sec). However, TAR does not reduce the data volume (Fig. 17f). Among the chunking methods, fixed-size chunking and context-aware chunking yield good performances: 79.6 MB/sec and 79 MB/sec (Fig. 17c), respectively. With respect to capacity, variable-size chunking and context-aware chunking reduce the data volume by 11.6 percent and 11.5 percent where fixed-size chunking reduce 10 percent. Adding a compression phase, fixed-size chunking speed and variable-size chunking speed decrease from 79.6 to 23.1 MB/sec and 79 to 19.6 MB/sec, respectively. The chunking speeds decrease by 29 percent and 25 percent for fixed-size chunking and variable-size chunking, respectively. By using compression, the data volume decreases by five percent, four percent, and 3.4 percent for fixed-size chunking, variable-size chunking, and context-aware chunking, respectively. The backup speed of context-aware chunking yields as high as that of fixed size chunking. For the deduplication ratio, context-aware chunking reduces the storage requirement by 12 percent which is similar to variable-size chunking. Given the experiment results for three data sets, context-aware chunking yields the best efficiency on both backup speed and capacity.

6 Conclusions

In this work, we not only propose an LRU-based index partitioning and Context-Aware chunking mechanism to address the performance of deduplication backup, but we also put a great deal of effort into understanding the relationship between chunking overhead, fingerprint lookup overhead, and overall backup speed.

According to our experiment, fixed-size chunking works well with regard to both the backup bandwidth and the degree of redundancy detection. This is because large files, which constitute the dominant fraction of the data volume, are multimedia files. These files are either identical or entirely different. On the other hand, small text files, e.g., source codes, have many commonalities across different versions of the operating system source code tree. However, these files are usually small and detecting commonalities in them may not justify the efforts of variable-size chunking. Chunking the file(s) in finer granularity enabled us to more quickly generate variable-size chunks and to find more commonalities among them. However, we also found that finer grain chunking significantly increases the fingerprint management overhead. By increasing the target pattern size from 11 to 13 bits, the deduplication detection rate decreased by two percent and the chunking performance decreased approximately from 150 to 100 MB/sec with files being in memory. However, the overall backup speed increased from 51 to 77 MB/sec. This experimental result delivers a very important implication: for the deduplication speed, we put more emphasis on reducing the fingerprint lookup overhead than in finding more commonalities.

For deduplication backup, a number of factors exist to optimize the performance: the false positive rate of the Bloom filter, the deduplication ratio, the chunking speed, the fingerprint lookup speed, etc. For optimizing deduplication performance, particular care needs to be taken to orchestrate the various factors involved in the entire deduplication process. Inappropriate optimization may result in an unacceptable penalty to the overall backup performance. This is

extremely important as maintaining a history of data is usually an irrevocable process once it has begun.

ACKNOWLEDGMENTS

This work is sponsored by MacroImpact Co, Ministry of Knowledge and Economics/KEIT, Korea (No. 10035202), National Research Lab Grant (ROA-2007-000-20114-0) of NRF, Korea. The authors would like to thank Dr. Sang Gue Oh and Dr. Jangsun Lee at Macroimpact for their insightfull comments on this work. Youjip Won is the Corresponding Author.

REFERENCES

[1] J. Gantz, C. Chute, A. Manfrediz, S. Minton, D. Reinsel, W. Schlichting, and A. Toncheva, *The Diverse and Exploding Digital Universe: An Updated Forecast of Worldwide Information Growth through 2011,* IDC, An IDC White Paper-Sponsored by EMC, Mar. 2008.

[2] W. Tichy, "Rcs: A System for Version Control," *Software Practice and Experience,* vol. 15, no. 7, pp. 637-654, July 1985.

[3] M. Ajtai, R. Burns, R. Fagin, D. Long, and L. Stockmeyer, "Compactly Encdoing Unstructured Input with Differential Compression," *J. ACM,* vol. 49, no. 3, pp. 318-367, May 2002.

[4] P. Kulkarni, F. Douglis, J. LaVoie, and J. Tracey, "Redundancy Elimination within Large Collections of Files," *Proc. USENIX Ann. Technical Conf., General Track,* pp. 59-72, 2004.

[5] F. Douglis and A. Iyengar, "Application-Specific Delta-Encoding via Resemblance Detection," *Proc. Conf. USENIX '03,* June 2003.

[6] Y. Won, J. Ban, J. Min, J. Hur, S. Oh, and J. Lee, "Efficient Index Lookup for De-Duplication Backup System," *Proc. IEEE Int'l Symp. Modeling, Analysis and Simulation of Computers and Telecomm. Systems (MASCOTS '08),* pp. 1-3, Sept. 2008.

[7] B. Zhu, K. Li, and H. Patterson, "Avoiding the Disk Bottleneck in the Data Domain Deduplication File System," *Proc. FAST '08: Sixth USENIX Conf. File and Storage Technologies,* pp. 1-14, 2008.

[8] A. Muthitacharoen, B. Chen, and D. Mazières, "A Low-bandwidth Network File System," *SIGOPS Operating Systems Rev.,* vol. 35, no. 5, pp. 174-187, 2001.

[9] B. Hong and D.D.E. Long, "Duplicate Data Elimination in a San File System," *Proc. 21st IEEE / 12th NASA Goddard Conf. Mass Storage Systems and Technologies (MSST),* pp. 301-314, Apr. 2004.

[10] H.P. nd David Andersen and M. Kaminsky, "Exploiting Similarity for Multi-Source Downloads Using File Handprints," *Proc. Symp. Networked Systems Design Implementation (NSDI '07),* Apr. 2007.

[11] M. Mitzenmacher, "Compressed Bloom Filters," *IEEE/ACM Trans. Networking,* vol. 10, no. 5, pp. 604-612, Oct. 2002.

[12] N.T. Spring and D. Wetherall, "A Protocol-Independent Technique for Eliminating Redundant Network Traffic," *Proc. SIGCOMM,* pp. 87-95, 2000.

[13] Y. Won, R. Kim, J. Ban, J. Hur, S. Oh, and J. Lee, "Prun: Eliminating Information Redundancy for Large Scale Data Backup System," *Proc. IEEE Int'l Conf. Computational Sciences and Its Applications(ICCSA '08),* 2008.

[14] S. Quinlan and S. Dorward, "Venti: A New Approach to Archival Storage," *Proc. Conf. File and Storage Technologies (FAST '02),* pp. 89-101, Jan. 2002.

[15] J.C. Mogul, Y.M. Chan, and T. Kelly, "Design, Implementation, and Evaluation of Duplicate Transfer Detection in http," *Proc. Symp. Networked Systems Design Implementation (NSDI '04),* p. 4, 2004.

[16] L.P. Cox, C.D. Murray, and B.D. Noble, "Pastiche: Making Backup Cheap and Easy," *SIGOPS Operating Systems Rev.,* vol. 36, no. SI, pp. 285-298, 2002.

[17] C. Policroniades and I. Pratt, "Alternatives for Detecting Redundancy in Storage Systems Data," *Proc. Conf. USEXNIX '04,* June 2004.

[18] C. Liu, Y. Lu, C. Shi, G. Lu, D. Du, and D. Wang, "ADMAD: Application-Driven Metadata Aware De-Duplication Archival Storage System," *Proc. Fifth IEEE Int'l Workshop Storage Network Architecture and Parallel I/Os (SNAPI '08),* pp. 29-35, 2008.

[19] D. Meister and A. Brinkmann, "Multi-Level Comparison of Data Deduplication in a Backup Scenario," *Proc. SYSTOR '09: The Israeli Experimental Systems Conf.,* pp. 1-12, May 2009.

[20] N. Mandagere, P. Zhou, M. Smith, and S. Uttamchandani, "Demystifying Data Deduplication," *Proc. ACM/IFIP/USENIX Middleware '08 Conf. Companion,* pp. 12-17, Dec. 2008.

[21] W.J. Bolosky, S. Corbin, D. Goebel, and J.R. Douceur, "Single Instance Storage in Windows 2000," *Proc. Fourth USENIX Windows Systems Symp.,* pp. 13-24, 2000.

[22] L.L. You, K.T. Pollack, and D.D.E. Long, "Deep Store: An Archival Storage System Architecture," *Proc. Int'l Conf. Data Engineering (ICDE '05),* pp. 804-8015, 2005.

[23] B.H. Bloom, "Space/Time Trade-Offs in Hash Coding with Allowable Errors," *Comm. ACM,* vol. 13, no. 7, pp. 422-426, 1970.

[24] F. Chang, J. Dean, S. Ghemawat, W.C. Hsieh, D.A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R.E. Gruber, "Bigtable: A Distributed Storage System for Structured Data," *Proc. Symp. Operating Systems Design and Implementation (OSDI '06),* pp. 205-218, 2006.

[25] M. Lillibridge, K. Eshghi, D. Bhagwat, V. Deolalikar, G. Trezise, and P. Camble, "Sparse Indexing: Large Scale, Inline Deduplication Using Sampling and Locality," *Proc. Seventh USENIX Conf. File and Storage Technologies (FAST '09),* 2009.

[26] D.R. Bobbarjung, S. Jagannathan, and C. Dubnicki, "Improving Duplicate Elimination in Storage Systems," *ACM Trans. Storage,* vol. 2, no. 4, pp. 424-448, 2006.

[27] L. Aronovich, R. Asher, E. Bachmat, H. Bitner, M. Hirsch, and S. Klein, "The Design of a Similarity Based Deduplication System," *Proc. SYSTOR '09: The Israeli Experimental Systems Conf.,* pp. 1-14, May 2009.

[28] J. Hamilton and E. Olsen, "Design and Implementation of a Storage Repository Using Commonality Factoring," *Proc. 20th IEEE/11th NASA Goddard Conf. Mass Storage Systems and Technologies(MSS '03),* Aug. 2003.

[29] D. Bhagwat, K. Eshghi, D. Long, and M. Lillibridge, "Extreme Binning: Scalable, Parallel Deduplication for Chunk-Based File Backup," *Proc. 17th IEEE Int'l Symp. Modeling, Analysis, and Simulation of Computer and Telecomm. Systems (MASCOTS '09),* Sept. 2009.

[30] A. Leung, M. Shao, T. Bisson, S. Pasupathy, and E. Miller, "Spyglass: Fast, Scalable Metadata Search for Large-Scale Storage Systems," *Proc. Six USENIX Conf. File and Storage Technologies (FAST '09),* 2009.

[31] C. Liu, Y. Gu, L. Sun, B. Yan, and D. Wang, "R-ADMAD: High Reliability Provision for Large-Scale De-Duplication Archival Storage Systems," *Proc. 23rd Int'l Conf. Supercomputing, (ICS '09),* pp. 370-379, 2009.

[32] D. Bhagwat, K. Pollack, D. Long, T. Schwarz, E. Miller, and J. Pâris, "Providing High Reliability in a Minimum Redundancy Archival Storage System," *Proc. 14th IEEE Int'l Symp. Modeling, Analysis, and Simulation of Computer and Telecomm. Systems (MASCOTS '06),* 2006.

[33] P. Efstathopoulos and F. Guo, "Rethinking Deduplication Scalability," *HotStorage '10, Second Workshop Hot Topics in Storage and File Systems,* June 2010.

[34] J. Burrows and D.O.C.W. DC, "Secure Hash Standard," *Federal Information Processing Standards Publication,* Apr. 1995.

[35] R. Rivest, "The MD5 Message Digest Algorithm, RFC 1321," *Internet Activities Board,* 1992.

[36] V. Henson, "An Analysis of Compare-by-Hash," *Proc. Conf. Hot Topics in Operating Systems (HOTOS '03),* 2003.

[37] "Berkeley db," http://www.oracle.com/technology/products/berkeley db/db/index.html, 2011.

[38] A. Broder and M. Mitzenmacher, "Network Applications of Bloom Filters: A Survey," *Internet Math.,* vol. 1, no. 4, pp. 485-509, 2004.

[39] N. Jain, M. Dahlin, and R. Tewari, "Taper: Tiered Approach for Eliminating Redundancy in Replica Synchronization," *Proc. FAST '05: Fourth Conf. USENIX File and Storage Technologies,* pp. 21-21, 2005.

[40] E. Horowitz, S. Sahni, and D. Mehta, *Fundamentals of Data Structures in C++.* Computer Science Press, 1995.

[41] A.Z. Broder and M. Mitzenmacher, "Network Applications of Bloom Filters: A Survey," *Internet Math.,* vol. 1, no. 4, pp. 485-509, 2003.

[42] L. Fan, P. Cao, J. Almeida, and A. Broder, "Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol," *IEEE/ACM Trans. Networking (TON),* vol. 8, no. 3, pp. 281-293, June 2000.

[43] P. Reynolds and A. Vahdat, "Efficient Peer-to-Peer Keyword Searching," *Lecture Notes in Computer Science,* pp. 21-40, Springer, 2003.

Jaehong Min received the BS and MS degrees in computer science from Hanyang University, Seoul, Korea, in 2008 and 2010, respectively. He is interested in file systems and operating systems. After graduation, he has been working as software engineer at MacroImpact Co., and is mainly working on developing deduplication based backup software.

Daeyoung Yoon received the BS degree in electric engineering from Korea University in 2005. He worked for LG Electronics as a software engineer. He is currently working toward the MS degree in the Department of Computer Science, Hanyang University, Seoul, Korea. His research interests include deduplication systems for managing massive data.

Youjip Won received the BS and MS degrees in computer science from the Seoul National University, Korea, in 1990 and 1992, respectively. He received the PhD degree in computer science from the University of Minnesota, Minneapolis, in 1997. After receiving the PhD degree, he joined Intel as a server performance analyst. Since 1999, he has been with the Department of Computer Science, Hanyang University, Seoul, Korea, as a professor. His research interests include operating systems, file and storage subsystems, multimedia networking, and network traffic modeling and analysis.