iJRASET
International Journal For Research in Applied Science and Engineering Technology



INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY

Volume: 4 **Issue:** IV **Month of publication:** April 2016

DOI:

www.ijraset.com

Call: 08813907089 | E-mail ID: ijraset@gmail.com

Volume 4 Issue IV, April 2016
ISSN: 2321-9653
International Journal for Research in Applied Science & Engineering Technology (IJRASET)

Multithreaded Variable Chunking in Source Based Deduplication in Cloud Backup Services using Support Vector Machine

K.S. Demel Abisha[1], S. Revathy Rajesh[2]
Computer Science and Engineering, Anna University

Abstract*— Cloud computing is typically defined as a type of computing that relies on sharing computing resources rather than having local server or personal devices to handle applications. we developed a novel multithreaded variable size chunking method,which exploits the multicore architecture of the modern microprocessors. The legacy single threaded variable size chunking method leaves much to be desired in terms of effectively exploiting the bandwidth of the state of the art storage devices. Multithreaded variable size chunking guarantees chunking invariability: The result of chunking does not change regardless of the degree of multithreading or the segment size. This is achieved by inter and intra-segment coalescing at the master thread and Dual Mode Chunking at the client thread. Then developed an elaborate performance model to determine the optimal multithreading degree and the segment size. MUCH is implemented in the prototype deduplication system. By fully exploiting the available CPU cores (quad-core), This achieved up to 4 increase in the chunking performance (MByte/sec).Multithreaded variable size chunking successfully addresses the performance issues of file chunking which is one of the performance bottlenecks in modern deduplication systems by parallelizing the file chunking operation while guaranteeing Chunking Invariability by using Support Vector Machine.

Keywords*— Cloud Computing, Content-based chunking, deduplication, multithread, Support vector machine Introduction.

I. INTRODUCTION

Cloud is a large pool of easily usable and accessible virtualized resources; these resources can be dynamically reconfigured to adjust to a variable load, allowing also for optimum resource utilization. This pool or resources is typically exploited by a pay-per-use model in which guarantees are offered by the Infrastructure Provider by means of customized SLAs. Cloud computing is typically defined as a type of computing that relies on sharing computing resources rather than having local server or personal devices to handle applications. The goal of cloud computing is to apply traditional supercomputing or high-performance computing power normally used by military research facilities portfolios, to deliver personalized information, to provide data storage. Cloud computing uses networks of large groups of servers typically running low-cost consumer technology with specialized connections to spread data-processing chores across them. This shared IT infrastructure contains large pools of systems that are linked together. Often, virtualization techniques are used to maximize the power of cloud computing. Cloud backup service is the core technology of cloud storage. Cloud backup stores data located at the client side into the cloud storage service provider through network so as to recover data in time. Cloud backup service has become cost-effective solution that is adopted by many organizations as their alternate data protection strategy.It is known that in most of the modern information systems, only 3 percent of the data is newly added to the storage and 5 percent of the existing data is updated . Reducing the data size plays a key role in filesystems, backup systems , web proxy , and even small storage devices. Elaborate techniques, e.g., compressing the data, eliminating the redundant information within or between files (deduplication) , storing only updated parts of data (delta encoding),have been developed to effectively address the original objective of reducing the data size. File chunking and duplication detection are two essential components in the deduplication In this work, we dedicate our efforts in eliminating the performance overhead in variable size chunking. Numerous techniques have been proposed for faster duplication detection: creating efficient key value store, arranging the fingerprints with respect to the access locality in deduplication, and minimizing cache miss penalty in in-memory database. Variable size chunking is very CPU intensive and chunking speed is far behind the I/O bandwidth of the modern storage devices. The speed of variable size chunking is merely 1/5 of that of the storage devices. We carefully argue that relatively little attention has been paid on improving the speed of variable size chunking despite its significant performance implications. The efficiency of variable size chunking becomes more important as the storage becomes faster. File chunking mechanisms need to be devised to effectively exploit the bandwidth of the underlying I/O interface. Augmenting hardware logic (ASIC or FPGA) for chunking may be a possible

©IJRASET: All Rights are Reserved

solution. However, this will work only for dedicated deduplication systems. In this work, we exploited the multiple cores in modern CPUs for file chunking. Most of the commercially available CPUs have two or more cores. Recently, even the smartphones are loaded with quad-core CPUs. The objective of this work is to develop a file chunking algorithm that effectively exploits the multiple cores of modern CPUs. MUCH is implemented in our prototype deduplication system, PRUNE and is in commercial deduplicated backup product.

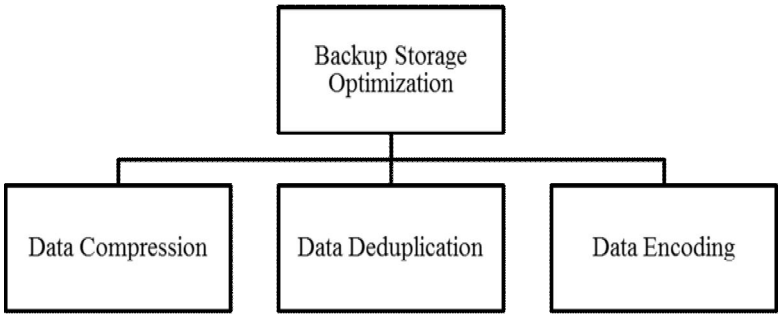


Fig. 1.1 Backup Storage Optimization Techniques

There are three different optimization technique that are used to overcome the storage optimization problem described above. These three optimization techniques are shown in the fig. 1.1. Among these three optimization techniques, the data deduplication is said to be most efficient one. There are number of techniques are traditionally used to save network bandwidth and storage spaces: incremental backup, compression, and delta encoding. The incremental backup technique is used to avoid redundant backup of unchanged files. Compression is widely used to reduce the size of the data. Delta encoding is effective when changes are small. It is used in many applications including source control and backup . Douglis and Iyengar proposed redundancy elimination at block level (REBL), which is a combination of block suppression, delta encoding, and compression. It needs to examine every pair of files to detect redundancy across the multiple files. Support Vector Machines are based on the concept of decision planes that define decision boundaries. A decision plane is one that separates between a set of objects having different class memberships. A schematic example is shown in the illustration below. In this example, the objects belong either to class GREEN or RED. The separating line defines a boundary on the right side of which all objects are GREEN and to the left of which all objects are RED. Any new object (white circle) falling to the right is labeled, i.e., classified, as GREEN (or classified as RED should it fall to the left of the separating line).

Deduplication can be categorized into two basic approaches:

In the target-based approach, deduplication is performed in the destination storage system. The client is not having knowledge about the deduplication strategies. This method have the advantage of increasing storage utilization, but does not save bandwidth. In Source based deduplication, elimination of duplicate data is performed close to where data is created, rather than where data is stored as in the case of target deduplication. The Source deduplication approach works on the client machine before it is transmitted specifically, the client software communicates with the backup server (by sending hash signatures) to check for the existence of files or blocks. Duplicates are replaced by pointers and the actual duplicate data is never sent over the network.

II. RELATED WORKS

Policroniades and Pratt 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. They showed that variable size chunking yields the best deduplication ratio. Eshghi and Tang introduced the chunk size control algorithm, two thresholds, two divisors (TTTD). Meister and Brinkmann compared the influence of different chunking approaches. In variable size chunking, Rabin's algorithm is widely used to establish the chunk boundaries. Despite the computational simplicity of Rabin's algorithm, variable size file chunking is still a very CPU intensive operation[1],since modulo arithmetic needs to be performed in every byte position of a file. Min et al. developed Incremental Modulo-K (INC-K) algorithm to improve the computational complexity of Rabin's algorithm. INC-K is an algebraically optimized version of Rabin's algorithm. Most works, if not all, on deduplication are grounded on the assumption that two different data blocks are very unlikely to yield the same fingerprints. Henson, however, stated that this

©IJRASET: All Rights are Reserved

probability may not be valid since the backed up files lasts much longer then hardware and therefore the probability of fingerprint conflicts can be much longer than we can expect. The computer hardware is upgraded every three to five years. The data should be preserved for much longer than that.. A number of in-memory data structures, such as Bloom filter, skip list and, etc., are employed to maintain fingerprints in a more compact manner. Bloom filter is widely used in distributed file systems, deduplication systems, and peer-to-peer file systems. To expedite the fingerprint lookup, a number of new search structures have been proposed. Hamilton and Olsen proposed "Commonality Factoring", fingerprinting the fingerprints. Stream-Informed Segment Layout (SISL) stores the fingerprints in the same order in which they are generated to preserve locality of fingerprint lookup. Lillibriged et al. proposed a sampling technique that reduces the disk I/O with a tolerable level of penalty on the deduplication ratio. In distributed backup, it is unlikely that a sequence of fingerprints bears any locality since the fingerprints generated from individual backup nodes are interleaved. Bhagwat et al. proposed a technique called Extreme Binning to handle distributed backup. Min et al. proposed index partitioning to limit the size of a single fingerprint lookup structure and maintain a set of fingerprints with multiples of such. Debnath et al. proposed to use an SSD as a key-value store for fingerprint lookup and developed a new key-value store structure optimized for SSDs, chunkstash. A number of works proposed to protect the important chunks via ECC and redundant group. Zhang et al. proposed to use distributed systems for deduplication.They partitioned the incoming data stream using fixed size chunking and sent the partitions to storage nodes. Each node adopts multithreading to generate MD-5 based fingerprint. A few works proposed to exploit GPU to expedite the deduplication. Multi-core chunking algorithm proposed in this work makes the variable size chunking faster (up to 400 percent in the current state of the art quad-core CPU) by exploiting the existing CPU cores. MUCH enables the deduplication system to well exploit the performance of the underlying storage devices.

III. PROBLEMASSESSMENT: OVERHEAD OF VARIABLE SIZE FILE CHUNKING

A. Synopsis: Deduplication

Data deduplication strategies are basically classified into two types based on data.

File-level deduplication, where only one copy of the file is stored in file level deduplication. If the two files are producing same hash value, then they are identical.

In Block-level deduplication, each file is fragmented into blocks and one copy of each block is stored. Each block may be of fixed-sized (static) or variable-sized chunks. In Fixed size chunks, size of each block is same. In case of variable, size of each chunk is different.

B. Fixed Size Chunking

There are two approaches in partitioning a file into chunks: fixed size chunking and variable size chunking. In fixed size chunking, a file is partitioned into fixed size units, e.g., 8 KByte blocks. It is simple, fast, and computationally very cheap. A number of proceeding works have adopted fixed-size chunking for backup applications and for large-scale file systems. However, when a small amount of content is inserted to or deleted from the original file, the fixed size chunking may generate a set of chunks that are entirely different from the original ones even though most of the file contents remain intact.

C. Variable Size Chunking

Variable size chunking partitions a file based on the content of the file, not the offset. Variable size chunking is relatively robust against the insertion/deletion of the file. The Basic Sliding Window (BSW) algorithm is widely used in variable size chunking.The BSW algorithm establishes a window of byte stream starting from the beginning of a file. It computes a signature, which is a hash value of byte stream in the window region. If the signature matches the predefined bit pattern, the algorithm sets the chunk boundary at the end of the window. After each comparison, the window slides one byte position and compute hash function again. Since the window slides one byte position at a time, we can use incremental hash functions, e.g., Rabin's algorithm and INC-K which are much cheaper than cryptographic hash functions such as SHA-1 and MD5.When we scan the byte string and compute Rabins algorithm over the sliding window, Rabins signature values can be computed from the previous ones in incremental fashion. Min et al. algebraically optimized the Rabins algorithm with respect to the register size and proposed INC-K algorithm. Even though we use incremental hash functions for chunking, variable size chunking is still a computationally intensive task. To reduce the computational complexity, most variable size chunking approaches establish the lower and the upper bounds on the chunk size. When the minimum chunk size is set, e.g., at 2 KByte, chunking module skips computing a signature for the chunk smaller than the minimum chunk size. If the average chunk size is 10 KByte, establishing the minimum chunk size at 2 KByte can eliminate

©IJRASET: All Rights are Reserved

approximately 20 percent of the signature computation. The length of the target bit pattern governs the average chunk size. If the target bit pattern is 13 bits long, the probability that a given pattern matches the target corresponds to 2_13 and the average chunk size approximately corresponds to 8 KByte. Min et al. developed an analytical model that calculates the optimal minimum and maximum chunking sizes. We physically examined the I/O bandwidth of the storage devices and the speed of variable size chunking. We used three storage devices with different bandwidths: RAMDISK (DDR3 DRAM, 1,333 MHz), SSD (OCZ Revo-Drive 3 X2), and RAID 0 (three HDDs). In variable size chunking, the average chunk size was set at approximately 8 KByte (13 bit signature size) with the minimum and the maximum chunk sizes being 2 and 8 KByte, respectively. In variable size chunking, we used INC-K algorithm, which is algebraically optimized version of Rabins algorithm. The variable size chunking operation saturates the CPU and the speed of variable size chunking is far slower than the I/O bandwidths of the storage devices.

IV. MULTITHREADED VARIABLE SIZE CHUNKING

A. Organization

The basic idea of multithreaded chunking is simple and straight forward. MUCH partitions a file into small fractions called segments and distributes them to the chunker threads. The chunker thread partitions the allocated fraction of the file into chunks. There are two types of threads in MUCH: the master thread and the chunker thread. The master thread partitions a file into segments, distributes them to chunker threads, collects the chunking results, and performs post-processing, if necessary. The chunker thread is responsible for partitioning a given segment into chunks, applying variable size chunking. MUCH works in an iterative manner. Despite its conceptual simplicity, MUCH requires elaborate technical treatments to be effective. The first issue is to guarantee chunking invariability: The result of chunking should not be affected by the number of chunker threads. The legacy basic sliding window protocol, under multithreading, will produce different chunks, depending on the degree of multithreading. we call this situation Multithread Chunking Anomaly. To address this issue, we developed Dual Mode Chunking, which enables us to guarantee chunking invariability. We formally proved that Dual Mode Chunking guarantees Chunking Invariability. The second issue is handling of small files. When a file is small, multi-core chunking may not be able to exploit the computing capabilities of all CPU cores. This issue is particularly important when the files for deduplication are mostly less than a few KBytes, e.g., Linux source tree. We developed a technique called Dynamic Segment Set Prefetching to handle this issue.

B. Multithread Chunking Anomaly

Most existing chunking algorithms establish the lower and upper bounds on chunk size to expedite the chunking process. This is to avoid too frequent chunking and to avoid chunk sizes growing indefinitely. Establishing the lower and upper bounds on the chunk size complicates the problem when chunking becomes multithreaded. In the original file, there are four prospective chunk boundaries, A, B, C, and D . In chunking, the minimum chunk size is set at 2 KByte. With single threaded chunking, chunk boundaries are set at A, B, C,

and D. We labeled the three chunks as C1, C2, and C3. Now, we chunked the file with two chunker threads. the file is partitioned into S1 and S2, with E being the segment boundary. Each segment is to be processed by the respective thread. In S1, chunk boundary is established at A. Chunk boundary is also set at E because the chunker thread reaches the end of the segment. In S2, chunker thread skips the first 2 KByte (minimum chunk size), then starts chunking. The problem is that chunk boundary B is 1.5 KByte off from the beginning of S2, and therefore, is undetected. In S2, the first chunk boundary is set at C. Depending on whether the master thread coalesces the last chunk of S1 or the first chunk of S2, the final result corresponds to either fCAE; CEB; CBC; C3g or fCAE; CEC; C3g. Neither of these is identical to the result of single threaded chunking, fC1; C2; C3g. This problem occurs because of the lower and upper bounds on the chunk size. We call this phenomenon Multithreaded Chunking Anomaly. Normally, computer hardware is upgraded every three to five years. Multithreading degree and the segment size need to be updated to properly reflect the characteristics of the underlying hardware Multithreaded chunking algorithm should be guaranteed to generate the identical chunks on the same input file independent of the number of chunker threads and the segment size. We call this constraint “Chunking Invariability”.

Definition 1. A given multithreaded chunking algorithm satisfies chunking invariability if it generates the identical chunks independent of multithreading degree and the segment size.

C. Dual Mode Chunking

Though the upper and lower bounds on the chunk size cause Multithreaded Chunking Anomaly, removing these bounds is not a viable option because that will increase the computational overhead. We developed a novel method to guarantee Chunking

©IJRASET: All Rights are Reserved

Invariability: Dual Mode Chunking. In Dual Mode Chunking, the chunker thread starts chunking at slow mode and then switches to accelerated model. When certain conditions are met. In slow mode, chunker thread does not impose the minimum nor the maximum chunk size restrictions and, therefore, chunking proceeds slowly. In accelerated mode, chunking algorithm enforces the lower and upper bounds on the chunk size to expedite chunking. Chunking in the accelerated mode is the same as the legacy single threaded chunking. Chunker thread works in slow mode until it finds a chunk whose size is larger than the minimum chunksize and smaller than or equal to the maximum chunk size—minimum chunk size. If this chunk is the first chunk in the segment, the chunker thread does not switch modes and keeps chunking in the slow mode. This chunk is called the Marker Chunk. The marker chunk is defined in Definition 2.

Definition 2. A marker chunk is a chunk that satisfies the following three conditions:

1) It is generated in the slow mode.

2) Chunk size is larger than Cmin and smaller than or equal to Cmax _ Cmin, where Cmin and Cmax are the minimum and the maximum chunk sizes, respectively.

3) It is not the first chunk in a segment.

Let us assume that the minimum and the maximum chunk sizes are 2 and 64 KByte, respectively. The first chunk, C1, is larger than 2 KByte and smaller than 64 KByte. However, C1 cannot be a marker chunk because it is the first chunk in the segment (violation of condition)C4 is a marker chunk because C1, C2, and C3 are smaller than Cmin(violation of condition C2 is a marker chunk because C1 is larger than the maximum chunk size, 64 KByte (violation of condition).

D. Chunk Marshalling

Some chunks generated in the slow mode can be smaller than the minimum chunk size or larger than the maximum chunk size. The master thread performs post-processing so that the final results are identical to the results obtained from single threaded chunking. We call this post-processing activity chunk marshalling. In this the master thread performs two types of tasks. The first is chunk coalescing. The master thread coalesces a certain chunk, with the next one, if preceding one .

is smaller than the minimum size. Chunk coalescing repeats until the newly created chunk becomes larger than or equal to the minimum size. The second task is chunk splitting. If a chunk is larger than the maximum chunk size, it is partitioned into two chunks so that the size of the first chunk is the same as the maximum chunk size. The master thread splits the chunk recursively until the remaining chunk size is smaller than or equal to the maximum chunk size. As a result of chunk marshalling, the sizes of all chunks are guaranteed to lie between the minimum and the maximum chunk size. Remember that for all chunks created from the chunker threads, the last chunk in the slow mode is the marker chunk. The marker chunk plays a critical role in guaranteeing that, as a result of chunk marshalling, the sizes of all chunks lie between the minimum and maximum chunk sizes. A marker chunk is larger than or equal to the minimum chunk size and smaller than or equal to the(maximum—minimum) chunk size. When a certain chunk, ci, is coalesced with the next one, ciþ1, and ciþ1 is a marker chunk, the newly created chunk cannot be larger than the maximum size since ci < cmin and ciþ1 _ cmin þ cmax, where cmin and cmax denote the minimum and the maximum chunk sizes.

Theorem 1. Dual Mode Chunking and chunk marshalling guarantees chunk invariability.

E. Handling Small Files: Dynamic Segment Set Prefetching

When the master thread fetches a fraction of a file from disk to memory, the remaining portion of a file may be smaller than the size of a segment set. We call this phenomenon Segment Set Fragmentation. With the fragmented segment set, either only the subset of the available threads are allocated to the segments or the master thread partitions the fragmented segment set into smaller units than a segment to distribute them to all threads. When the segment size becomes smaller, the overhead of Dual Mode Chunking and chunk coalescing may overwhelm the performance gain obtained from parallelizing the chunking operation. We developed an effective technique, Dynamic Segment Set Prefetching, to address the Segment Set Fragmentation problem. The master thread checks if the next segment set is fragmented before it reads the segment set from the storage. If the next segment set is not fragmented, the master thread proceeds and fetches the segment set. Otherwise, the master thread fetches the current segment set as well as all remaining data. Then, the master thread allocates equal amounts of data to each chunker thread.

V. EXPERIMENT

A. Experiment Setup

We implemented MUCH in a prototype deduplicated backup system, PRUNE. We examined the effectiveness of multicore chunking in three different data sets: 2 GByte ISO image, a mixture of different sized files (totaling 200 GByte), and Linux source

©IJRASET: All Rights are Reserved

tree (totaling 341MByte). We examined MUCH under three storage devices with different performance characteristics: RAMDISK, high performance SSD, and RAID 0 based storage device that consists of three hard disk drives . We ran MUCH on Intel Core i5 (2.80 GHZ, quad-core) with 4 GByte of DRAMand Linux 2.6.32 with EXT4 filesystem. Young performed an extensive survey on various statistical characteristics of filesystems on different servers: web server, e-learning server, DB server, and mail server. The median and average file size of webserver, e-learning server, DB server and mail server correspond to 7.1, 1.5, 7.3, and 2.8 KByte, respectively. The average file size of web server, e-learning server, DB server, and mail server correspond to 207.7 KByte, 173.3 MByte, 29.9 KByte, and 306.6 KByte, respectively. It is worth noting that e-learning server and the mail server have heavy tailed file size distribution.

B. Chunking Performance: Large Sized Files

In chunking, the overhead of open() and close() system calls is non-trivial and becomes more significant as the underlying storage device gets faster. When the file is large, e.g., tar image of the filesystem snapshot, the overhead of opening and closing the file is not significant. When the file is small, e.g., Linux source files, the overhead of opening and closing a file constitutes a dominant fraction of the overall chunking time and the chunking performance degrades. We tested the chunking performance under three data sets with different file sizes. First, we chunked a large file (2 GByte) varying the segment sizes from 10 to 800 KByte. from theoretical model and physical experiments. We can see that our analytical model accurately estimates the chunking bandwidths. With one thread, chunking speed can reach approximately 96 MByte/ sec for all three storage devices. Sequential read bandwidths of RAMDISK (DRAM), SSD, and RAID 0 corresponded to 4.7 GByte/sec, 660 MByte/sec, and 280 MByte/sec, respectively. Storage bandwidths are significantly under utilized. For RAMDISK and SSD, chunking speed reached 340 MByte/sec, which is still only 7 and 51 percent of the sequential read speeds of RAMDISK and SSD, respectively. When the data was in RAID 0, we could fully exploit the storage bandwidth (280 MByte/sec). It is not possible to fully exploit the I/O bandwidths of the modern high performance storage devices, even when all the CPU cores are fully exploited. Recent emergence of PCIe attached high-end SSDs with multiple GByte/sec sequential I/O bandwidth will only widen the chasms between the storage speed and the chunking speed. A possible remedy for this problem is to utilize GPGPU to chunk files and/or to use dedicated hardware. In all three storage devices, chunking speed increased with segment size. When the file resides in a memory, increasing the segment size quickly saturates the CPU. When the file is on a disk, we need 400 KByte segment size to fully exploit the underlying storage

C. Chunking Performance: Medium Sized Files

We examined the chunking performance with medium sized files. The total file size was 200 GByte. These sets of files were created to mimic the multimedia data server that harbors relatively large files, e.g., images, music files, video files, executables, etc. There were a total of 5,546 files and the average file size was 36.9 MByte. For the SSD, with four chunker threads and 500 KByte segment size, the chunking speed reaches 350MByte/sec. Chunking speed increased by 350 percent with four chunker threads compared to single thread. It is still 70 percent of the SSD bandwidth. For RAID 0, by using four chunker threads with 1 MByte segment, the chunking speed reached 250 MByte/sec. It is 98 percent of the sustained sequential read bandwidth.

D. Chunking Performance: Small Sized Files

We tested the chunking bandwidth of MUCH using the Linux source tree. The objective of this experiment is to examine the effectiveness of MUCH when the file is small. The average file size is 11 KByte and 85 percent of files are smaller than 20 KByte. when the file is small, increasing the segment size does not improve the chunking performance. This is because when we partition a single file that is smaller than tens of KByte to multiple segments, each segment will yield only a few chunks, e.g., one or two chunks. The coalescing overhead constitutes7.7 Multicore Chunking in Smartphones Recently introduced high-end smartphones are equipped with four cores or eight cores. We examined the performance of the proposed multicore chunking algorithm in the mobile platform. We ported MUCH algorithm on a commercially available smartphone (Samsung Galaxy S3,Android 4.1.2, Exynos 4412 1.4 Ghz quad-core). The files were stored in the internal flash based-storage (/data partition, internal eMMC, Samsung KMVTU000LM). We measured the chunking performances while varying the number of cores. In chunking a large file (ISO Image, 2GByte), the chunking bandwidth increased from 20 to 60 MByte/sec when we increased the number of cores used in chunking from one to four in the Smartphone CPU (Exynos 4412). When the file is small (Linux source tree, Fig. 14b), the performance improvement from multicore chunking is not as significant as in the case of large files since the open() and close() overhead constitutes a significant fraction of the overall chunking latency.

©IJRASET: All Rights are Reserved

E. Scalability

We examined the effects of parallelism. In three data sets, we increased the number of chunker threads to four and examined the chunking bandwidths. Chunking consists of three components: (i) reading a set of segments, (ii) chunking, and (iii) chunk coalescing. MUCH aims at improving the overall chunking performance by parallelizing the second component: chunking. On the other hand, if the storage device is slow, the time to read a set of segments will constitute a dominant fraction of chunking and, therefore, the benefit of employing multithreading becomes marginal. If the file is small (or if the segment size is small), the benefit of employing multithreading becomes marginal since relatively significant fraction of chunking overhead consists of chunk coalescing. Since I/O and chunk coalescing cannot be parallelized, as the fraction of these components gets larger, MUCH becomes less scalable.. The X and Y axis denote the number of chunker threads and the chunking bandwidths, respectively. As the average file size gets larger, MUCH becomes more scalable. When the file is small, employing multiple threads barely helps the chunking performance in an SSD and RAID 0 storage devices.

F. Effect of Dynamic Segment Set Prefetching

We examined the relationship between the file size and the effects of Dynamic Segment Set Prefetching. We varied the file sizes from 100 KByte to 4 MByte. In this experiment, MUCH used four threads with a segment size of 100 KByte. We used four threads. Segment size was 50 KByte. When a file is in the memory, DSSP significantly improves chunking bandwidth. With a 500KByte file, chunking speed increased by 50 percent by using DSSP. The effects of DSSP become less significant as the file size increases. However, even for a 3.7 MByte file, using DSSP increased the chunking bandwidth by more than 10 percent. Given that most of the files we use are less than 1Mbyte,DSSP is a critical mechanism to expedite the chunking process. The effects of DSSP for a set of mixed files and a set of Linux source files, respectively. DSSP is more effective when the files are relatively small and when each of the files consists of a small number of segments. For Linux sources, when the files were stored in the memory, e.g., RAMDISK, DSSP improved the chunking performance by twofold. For Linux sources, when the files were in RAID storage, the performance benefit of DSSP was approximately 10 percent.

G. Multicore Chunking in Smartphones

Recently introduced high-end smartphones are equipped with four cores or eight cores. We examined the performance of the proposed multicore chunking algorithm in the mobile platform. We ported MUCH algorithm on a commercially available smartphone (Samsung Galaxy S3, Android 4.1.2, Exynos 4412 1.4 Ghz quad-core). The files were stored in the internal flash based-storage (/data partition, internal eMMC, Samsung KMVTU000LM). We measured the chunking performances while varying the number of cores. In chunking a large file (ISO Image, 2GByte), the chunking bandwidth increased from 20 to 60 MByte/sec when we increased the number of cores used in chunking from one to four in the Smartphone CPU (Exynos 4412). When the file is small (Linux source tree, Fig. 14b), the performance improvement from multicore chunking is not as significant as in the case of large files since the open() and close() overhead constitutes a significant fraction of the overall chunking latency. Fig. 14 illustrates the results of our experiment. Followings are the results of the experiment. By utilizing all cores in quad-core CPU, we increased the chunking performance by _4 and achieved up to 350 MByte/sec chunking performance. We increased the storage bandwidth utilization from 20 to 70 percent.

VI. IMPLEMENTATION

A. Server Configuration

In a client/server environment, each computer still holds (or can still hold) its (or some) resources and files. Other computers can also access the resources stored in a computer, as in a peer-to-peer scenario. One of the particularities of a client/server network is that the files and resources are centralized. This means that a computer, the server, can hold them and other computers can access them. Since the server is always ON, the client machines can access the files and resources without caring whether a certain computer is ON. One of the consequences of a client/server network is that, if the server is turned OFF, its resources and sometimes most of the resources on the network are not available. In fact, one way to set up a client/server network is to have more than one server. In this case, each server can play a different role. Another big advantage of a client/server network is that security is created, managed, and can highly get enforced. To access the network, a person, called a user must provide some credentials, such as a username and a password. If the credentials are not valid, the user is prevented from accessing the network.

©IJRASET: All Rights are Reserved

B. Authentication

An authentication module is a plug-in that collects user information such as a user ID and password, and compares the information against entries in a database. If a user provides information that meets the authentication criteria, the user is validated and, assuming the appropriate policy configuration, granted access to the requested resource. If the user provides information that does not meet the authentication criteria, the user is not validated and denied access to the requested resource. OpenSSO Enterprise is deployed with a number of authentication modules. This module is created to centralize and encapsulate all data storage and retrieval duties on the system. This includes user profiles, success stories, banner ads, pictures, and messages. It also provides some services, such as authentication, network communication and search. all communications from the client come through the communications module. The provided interface of the high-level server component and the server communications module are thus identical. Two modules in the server handle administrator and user functions, respectively. Each of these modules will be described in detail in following sections.

C. Data Chunk

Variable size chunking partitions a file based on the content of the file, not the offset. Variable size chunking is relatively robust against the insertion/deletion of the file. The Basic Sliding Window (BSW) algorithm is widely used in variable size chunking explains the BSW algorithm. The BSW algorithm establishes a window of byte stream starting from the beginning of a file. It computes a signature, which is a hash value of byte stream in the window region. If the signature matches the predefined bit pattern, the algorithm sets the chunk boundary at the end of the window. After each comparison, the window slides one byte position and computes hash function again. Since the window slides one byte position at a time, Then use incremental hash functions.

D. Data Transfer

Some chunks generated in the slow mode can be smaller than the minimum chunk size or larger than the maximum chunk size. The master thread performs post-processing so that the final results are identical to the results obtained from single threaded chunking. We call this post-processing activity chunk marshalling. In this phase, the master thread performs two types of tasks. The first is chunk coalescing. The master thread coalesces a certain chunk, with the next one, if preceding one is smaller than the minimum size. Chunk coalescing repeats until the newly created chunk becomes larger than or equal to the minimum size. The second task is chunk splitting. If a chunk is larger than the maximum chunk size, it is partitioned into two chunks so that the size of the first chunk is the same as the maximum chunk size. The master thread splits the chunk recursively until the remaining chunk size is smaller than or equal to the maximum chunk size. As a result of chunk marshalling, the sizes of all chunks are guaranteed to lie between the minimum and the maximum chunk size. Remember that for all chunks created from the chunker threads, the last chunk in the slow mode is the marker chunk. The marker chunk plays a critical role in guaranteeing that, as a result of chunk marshalling, the sizes of all chunks lie between the minimum and maximum chunk size.

VII. CONCLUSION

we developed a novel multicore chunking algorithm, Multithreaded variable chunking, which parallelizes the variable size chunking. We found that variable size chunking is computationally very expensive and is a significant bottleneck in the overall deduplication process. we developed a parallel chunking algorithm, which aims at making the variable chunking speed on par with the storage I/O bandwidths. we presented and enhanced storage system utilizing file modification pattern to generate the active pairs of input for out optimized deduplication algorithm. The Support vector machine in yet to be implemented, such that the active pairs of input thus generated will be sent as population and the high optimized process of deduplication could done with large of results in a minimal amount of time. Thus, increasing the efficiency of the computing system, and reducing the time taken for each and every backup operation.

VIII. ACKNOWLEDGMENTS

I wish to extend my grateful acknowledgement and sincere thanks to my project Guide Mrs.S. REVATHY, M.E.,

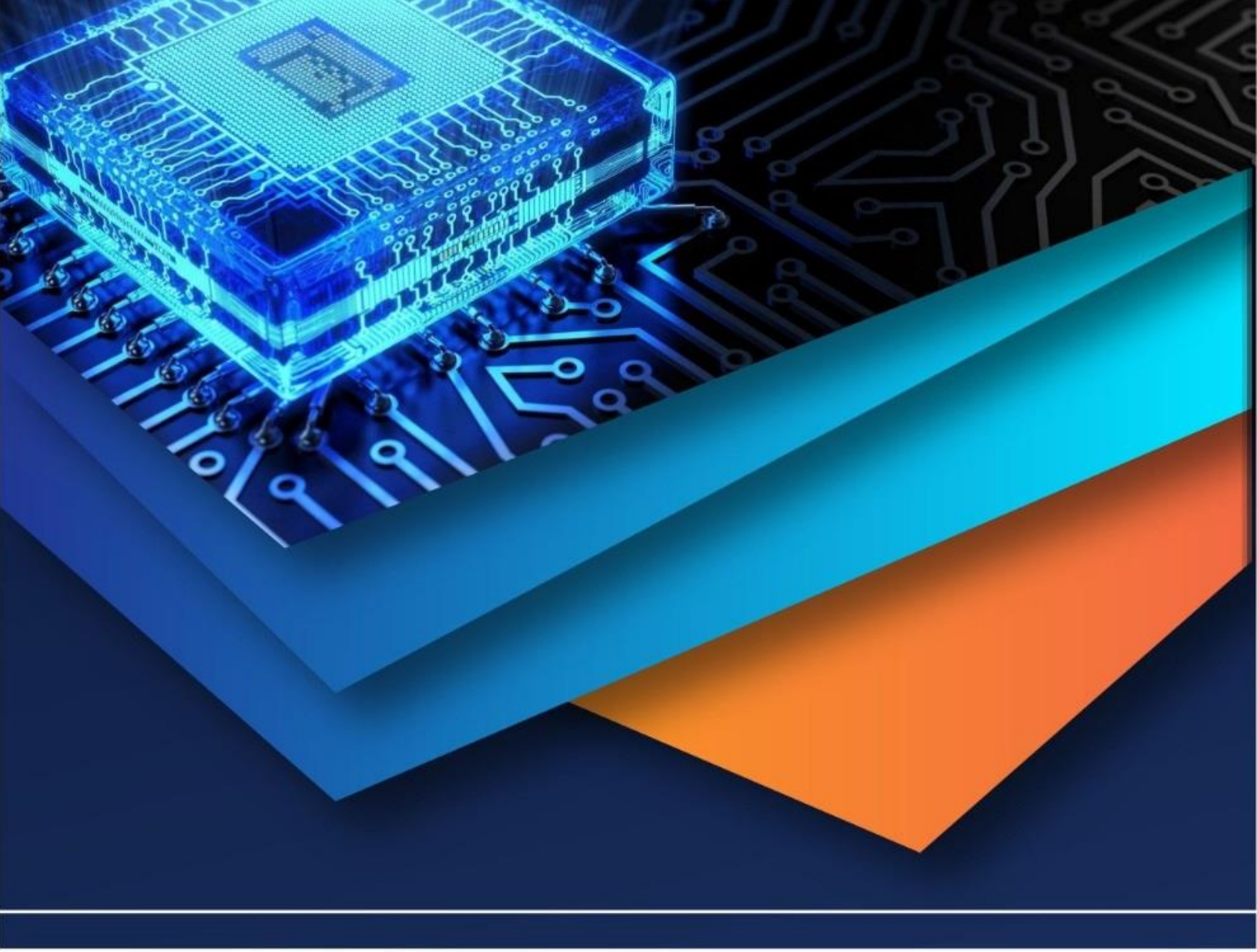
REFERENCES

[1] D. Bhagwat, K. Eshghi, D. Long, and M. Lillibridge, “Extreme binning:Scalable, parallel deduplication chunk-based file backup,”in Proc. 17th IEEE/ACM Int. Symp. Modelling, Anal. Simul. Comput.Telecommun. Syst., London, U.K., Sep. 2009, pp. 1–9.
[2] B. Debnath, S. Sengupta, and J. Li, “Chunkstash: Speeding up inlinestorage deduplication using flash memory,” in Proc. USENIX Conf.USENIX Annu. Tech. Conf., Boston,MA, USA, Jun. 2010, pp. 215–229.

©IJRASET: All Rights are Reserved

[3] K. Eshghi and H. Tang, "A framework for analyzing and improving content-based chunking algorithms," Hewlett-Packard Labs, Paolo Alto, CA, USA, Tech. Rep. TR, vol. 30, 2005.
[4] B. Hong, D. Plantenberg, D. Long, and M. Sivan-Zimet, "Duplicate data elimination in a SAN file system," in Proc. 21st IEEE/12th NASA Goddard Conf. Mass Storage Syst. Technol., Greenbelt, MD, USA, Apr. 2004, pp. 301–314.
[5] J. Kim, C. Lee, S. Lee, I. Son, J. Choi, S. Yoon, H. Lee, S. Kang, Y.Won, and J. Cha, "Deduplication in SSDS: Model and quantitative analysis," in Proc. IEEE 28th Symp. Mass Storage Syst. Technol.,Apr. 2012, pp. 1–12.
[6] M. Lillibridge, K. Eshghi, D. Bhagwat, V. Deolalikar, G. Trezise,and P. Camble, "Sparse indexing: Large scale, inline deduplication using sampling and locality," in Proc. 7th USENIX Conf. File Storage Technol., San Francisco, CA, USA, Feb. 2009, pp. 111–124.
[7] Y. Won, J. Ban, J. Min, J. Hur, S. Oh, and J. Lee, "Efficient index lookup for de-duplication backup system," in Proc. IEEE Int. Symp.Modeling, Anal. Simul. Comput.Telecommun.Syst. 2008, Baltimore,MD, USA, Sep. 2008, pp. 1–3.
[8] Y. Won, R. Kim, J. Ban, J. Hur, S. Oh, and J. Lee, "Prun: Eliminating information redundancy for large scale data backup system,"in Proc. IEEE Int. Conf. Comput. Sci. Appl., Perugia, Italy, Mar.2008, pp. 139–144.
[9] Y. Zhang, Y. Wu, and G. Yang, "Droplet: A distributed solution of data deduplication," in Proc. ACM/IEEE 13th Int. Conf. Grid Comput., 2012, pp. 114–121.
[10] B. Zhu, K. Li, and H. Patterson, "Avoiding the disk bottleneck inthe data domain deduplication file system," in Proc. 6th USENIXConf. File Storage Technol., Berkeley, CA, USA, 2008, pp. 1–14.

©IJRASET: All Rights are Reserved



10.22214/IJRASET

45.98

IMPACT FACTOR: 7.129

IMPACT FACTOR: 7.429

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY

Call : 08813907089 (24*7 Support on Whatsapp)