MUCH: Multithreaded Content-Based File Chunking

Youjip Won, Kyeongyeol Lim and Jaehong Min

Abstract—In this work, we developed a novel multithreaded variable size chunking method, MUCH, 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. MUCH 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. We 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), we achieved up to ×4 increase in the chunking performance (MByte/sec). MUCH 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.

Index Terms—Content-based Chunking, Deduplication, Multithread

1 Introduction

It is known that in most of the modern information systems, only 3% of the data is newly added to the storage and 5% of the existing data is updated[1]. Reducing the data size plays a key role in filesystems, backup systems[2], web proxy[3], and even small storage devices[4], [5], [6]. Elaborate techniques, e.g., compressing the data[7], eliminating the redundant information within or between files (deduplication)[1], storing only updated parts of data (delta encoding)[8], 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[9], [10], arranging the fingerprints with respect to the access locality in deduplication[2], [11], [6], and minimizing cache miss penalty in in-memory database[12]. Variable size chunking is very CPU intensive and chunking speed is far behind the I/O bandwidth of the modern storage devices. For example, a single core CPU (2.80 GHz, core i5) can chunk the data stream at 97 MByte/sec[11]. Meanwhile, most of the modern storage devices offer faster than 500 MByte/sec I/O bandwidth[11], [13]. The speed of variable size chunking is merely 1/5 of that of the storage devices.

• *Y. Won, K. Lim, J, Min is with the Division of Computer Science and Engineering, Hanyang University, Seoul 133-791, Korea. E-mail: {yjwo|lkyeol}@hanyang.ac.kr, jh.leo.min@gmail.com*

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 solution. However, this will work only for dedicated deduplication systems, e.g., VTL[14], and will raise cost, standardization, and usability issues in general purpose computers (desktop PCs, servers).

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[15] 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 [16] and is in commercial deduplicated backup product [17].

2 Related Work

There are a number of areas where deduplication techniques play a key role: distributed and shared file systems[18][19], backup[2], peer-to-peer file systems[20], and web-proxy servers[3]. Recently, deduplication techniques have been employed in the SSD controllers to reduce the amount of writing to the NAND page[4], [5], [6]. In backup systems, a number of techniques are traditionally used to save network bandwidth and storage spaces: incremental backup, compression, and delta encoding. The

incremental backup[21] technique is used to avoid redundant backup of unchanged files. Compression is widely used to reduce the size of the data[22]. Delta encoding is effective when changes are small. It is used in many applications including source control[23] and backup[7]. Douglis et al. 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[8], [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. They showed that variable size chunking yields the best deduplication ratio[22]. Tang et al. introduced the chunk size control algorithm, TTTD (two thresholds, two divisors)[24]. Meister et al. compared the influence of different chunking approaches[25].

In variable size chunking, Rabin's algorithm[26] is widely used to establish the chunk boundaries[14], [19], [2]. Despite the computational simplicity of Rabin's algorithm, variable size file chunking is still a very CPU intensive operation[11], [1] since modulo arithmetic needs to be performed in every byte position of a file. Min et al. developed INC-K (Incremental Modulo-K) algorithm[11] 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[19], [18]. Henson [27], however, stated that this 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[28]. The data should be preserved for much longer than that[1].

A number of in-memory data structures, such as Bloom filter[31], skip list[32] and etc, are employed to maintain fingerprints in a more compact manner. Bloom filter is widely used in distributed file systems[33], deduplication systems[2], and peer-to-peer file systems[20].

To expedite the fingerprint lookup, a number of new search structures have been proposed. Hamilton et al. proposed "Commonality Factoring", fingerprinting the fingerprints [34]. SISL (Stream-Informed Segment Layout) stores the fingerprints in the same order in which they are generated to preserve locality of fingerprint lookup[2]. Lillibriged et al. proposed a sampling technique that reduces the disk I/O with a tolerable level of penalty on the deduplication ratio[14]. 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[35]. Min et al.[16] 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[10]. A number of works proposed to protect the important chunks via ECC and redundant group[36], [37].

Yang Zhang et al. proposed to use distributed systems for deduplication[38]. 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[39], [40].

None of the above mentioned works actually addressed the issue of improving the chunking performance. Multithreaded variable size file chunking proposed in this study compliments the prior works in deduplication. Multi-core chunking algorithm proposed in this work makes the variable size chunking faster (up to 400% 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.

The rest of the paper is organized as follows. Section 3 provides the background information on MUCH. Section 4 explains the concept of multithreaded chunking. Section 5 describes the performance model for MUCH. Section 6 provides an in-depth analysis of the performance model of MUCH. In section 7, we present the results of the physical experiments. Section 8 concludes the paper.

3 Problem Assessment: Overhead of Variable Size File Chunking

3.1 Synopsis: Deduplication

Deduplication consists of four key technical ingredients: (i) file chunking, (ii) duplication detection, (iii) enhancing reliability for deduplicated data, and (iv) information life cycle management. There are two types of chunking: variable size chunking[19], [14] and fixed size chunking[19], [14]. Duplication detection is a process of determining whether a given chunk is redundant or not. In this phase, the deduplication process generates a summary information (fingerprint) for a chunk and compares the fingerprint against a set of the existing fingerprints. In deduplicated data, certain data blocks are shared by multiple files, where loss cannot be tolerated. A number of

[1]The financial and medical records should be preserved for seven[29] to 30 years[30].

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2014.2322600, IEEE Transactions on Computers

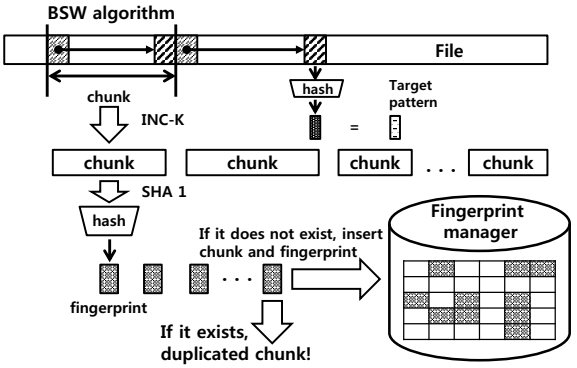


Fig. 1. Deduplication process

works proposed to adjust the size of parity/checksum and the degree of replication based on the number of references to the respective chunk[11]. Information Lifecycle Management (ILM) is the process of migrating the files to the next lower level in the storage hierarchy. If the data is old enough and has not been accessed for a certain amount of period, it is removed from the storage and *archived*. Deduplication adds another dimension of complexity in Information Lifecycle Management[41]. While the deduplication effort is about exploiting the redundancies along the spatial axis (within a file or among the files) or along the temporal axis (across the different versions of the same file), ILM in deduplication is about reconstructing the eliminated redundancies. The essence of the issue here is how to efficiently scan the reference counters of the data blocks in the deduplicated storage[11].

3.2 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[42] and for large-scale file systems[18]. 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.

3.3 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 BSW (Basic Sliding Window) algorithm[19] is widely used in variable size chunking. Fig. 2 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

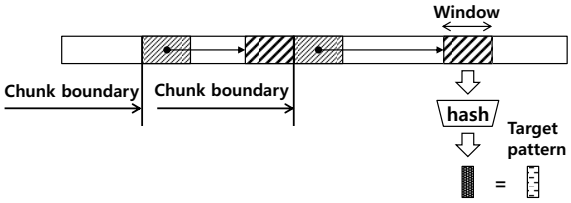


Fig. 2. The basic sliding window algorithm

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, we can use incremental hash functions, e.g., Rabin's algorithm[26] and INC-K[11], which are much cheaper than cryptographic hash functions such as SHA-1 and MD5.

Let us briefly explain the Rabins algorithm. For a given byte string, $b_1, \cdots, b_n$, we need to obtain a signature for each substring $\{b_1, \cdots, b_\beta\}, \{b_2, \cdots, b_{\beta+1}\}, \{b_3, \cdots, b_{\beta+2}\}, \cdots$. In Rabins algorithm, the signature is represented $Rf(b_1, b_2, \cdots, b_\beta) = (b_1 p^{\beta-1} + b_1 p^{\beta-2} + \cdots + b_\beta) \ mod \ M$, where β and M are irreducible polynomial. 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. Eq. 1 illustrates Rabins signature computation formula, which uses the previous signature values. It requires only one shift operation, one addition operation, and one modulo operation.

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

Min et al. algebraically optimized the Rabins algorithm with respect to the register size[16] 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 chunks smaller than the minimum chunk size. If the average chunk size is 10 KByte, establishing the minimum chunk size at 2 KByte can eliminate approximately 20% 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[16].

We physically examined the I/O bandwidth of the storage devices and the speed of variable size chunk-

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2014.2322600, IEEE Transactions on Computers

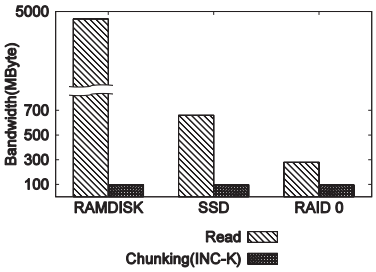


Fig. 3. Read and chunking bandwidth

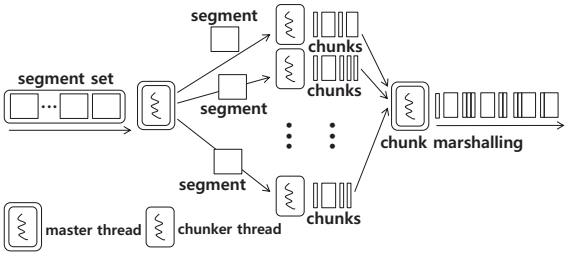


Fig. 4. System organization of multithreaded chunking, MUCH

ing. We used three storage devices with different bandwidths: RAMDISK (DDR3 DRAM, 1333 MHz), SSD (OCZ RevoDrive 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 KByte and 8 KByte, respectively. Fig. 3 illustrates the results. Read bandwidths of RAMDISK, SSD, and RAID 0 (three HDDs) yielded 4.9 GByte/sec, 660 MByte/sec, and 280 MByte/sec, respectively. On the other hand, the speed of variable size chunking was 97 MByte/sec for all storage devices. In variable size chunking, we used INC-K algorithm[11], 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.

4 MUCH: Multithreaded Variable Size Chunking

4.1 Organization

The basic idea of multithreaded chunking (MUCH) 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

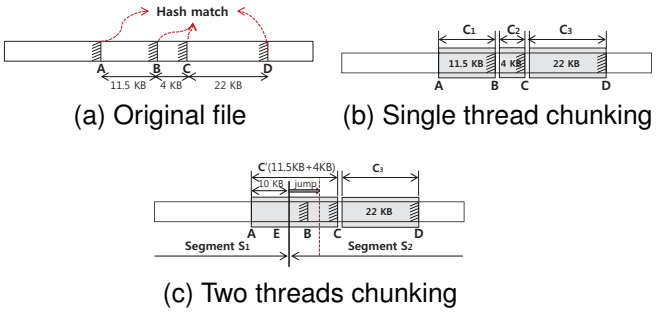


Fig. 5. Multithreaded Chunking Anomaly

works in an iterative manner. Fig. 4 illustrates the organization of MUCH, multithreaded chunking system.

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[19], 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.

4.2 Multithread Chunking Anomaly

Most existing chunking algorithms establish the lower and upper bounds on chunk size to expedite the chunking process[19]. 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. Fig. 5 illustrates an example. In the original file, there are four prospective chunk boundaries, A, B, C, and D (Fig. 5a). 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 (Fig. 5b). We labeled the three chunks as C_1, C_2, and C_3. Now, we chunked the file with two chunker threads. Fig. 5c illustrates the situation. In Fig. 5c, the file is partitioned into S_1 and S_2, with E being the segment boundary. Each segment is to be processed by the respective thread. In S_1, chunk boundary is established at A. Chunk boundary is also set at E because the chunker thread reaches the end of the segment. In S_2, chunker thread skips

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2014.2322600, IEEE Transactions on Computers

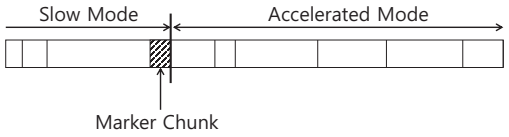


Fig. 6. Dual Mode Chunking

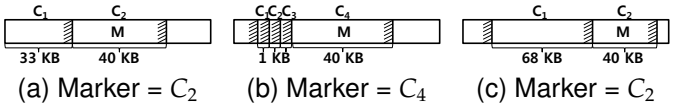


Fig. 7. Examples of marker chunks

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 S_2, and therefore, is undetected. In S_2, the first chunk boundary is set at C. Depending on whether the master thread coalesces the last chunk of S_1 or the first chunk of S_2, the final result corresponds to either $\{C_{AE}, C_{EB}, C_{BC}, C_3\}$ or $\{C_{AE}, C_{EC}, C_3\}$. Neither of these is identical to the result of single threaded chunking, $\{C_1, C_2, C_3\}$. 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[28]. Multithreading degree and the segment size need to be updated to properly reflect the characteristics of the underlying hardware, e.g., the number of CPU cores, storage bandwidth, etc, and to exploit the hardware capabilities. 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 A given multithreaded chunking algorithm satisfies *chunking invariability* if it generates the identical chunks independent of multithreading degree and the segment size.

4.3 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 Invariability: *Dual Mode Chunking*.

In Dual Mode Chunking, the chunker thread starts chunking at *slow mode* and then switches to *accelerated mode* 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 chunk size 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 4.3.

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

i) It is generated in the slow mode.
ii) Chunk size is larger than C_{min} and smaller than or equal to $C_{max} - C_{min}$, where C_{min} and C_{max} are the minimum and the maximum chunk sizes, respectively.
iii) It is not the first chunk in a segment.

Fig. 7 illustrates examples of marker chunks. Let us assume that the minimum and the maximum chunk sizes are 2 KByte and 64 KByte, respectively. In Fig. 7a, the first chunk, C_1, is larger than 2 KByte and smaller than 64 KByte. However, C_1 cannot be a marker chunk because it is the first chunk in the segment (violation of condition(iii)). In Fig. 7b, C_4 is a marker chunk because C_1, C_2, and C_3 are smaller than C_{min}(violation of condition (ii)). In Fig. 7c, C_2 is a marker chunk because C_1 is larger than the maximum chunk size, 64 KByte (violation of condition (ii)).

4.4 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 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 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, c_i, is coalesced with the next one, c_{i+1}, and c_{i+1} is a marker chunk, the newly created chunk cannot be larger than the maximum size since $c_i < c_{min}$ and $c_{i+1} \leq c_{min} + c_{max}$, where c_{min} and c_{max} denote the minimum and the maximum chunk sizes.

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

Proof: Refer to Appendix B.

4.5 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.

5 Performance Modeling

5.1 Signature Generation Overhead in Single Threaded Chunking

Let p be the probability that the chunk boundary condition is met at a given byte position. For example, if the chunk boundary condition is that the signature matches predefined bit pattern of 10 bit, p corresponds to $\frac{1}{2^{10}}$. Let S be the chunk size. The average chunk size in single threaded chunking can be formulated as in Eq. 2.

$$E[S] = \sum_{i=c_{min}}^{c_{max}} i(1-p)^{i-1}p$$
$$= c_{min}(1-p)^{c_{min}-1} + \frac{1}{p}(1-p)^{c_{min}} - (1-p)^{c_{max}}(\frac{1}{p} + c_{max}) \quad (2)$$

TABLE 1
Modeling parameters: p=2^{-13}, c_{min}=2 KB, c_{max}=64 KB, λ=14 nsec, δ=4 msec, ζ=8 usec, w=48 byte

Symbol	Meaning
l_{acc}	no. of signatures in accelerated mode
l_{slow}	no. of signatures in slow mode
n_{acc}	average no. of chunks in accelerated mode
n_{slow}	average no. of chunks in slow mode
s_a	average chunk size in accelerated mode
s_b	average chunk size in slow mode
p	signature match probability
c_{min}	minimum chunk size
c_{avg}	average chunk size
c_{max}	maximum chunk size
c_{mark}	upper bound of marker chunk
λ	time taken to generating single signature
δ	disk seek time
ζ	time taken to merging two chunks
w	window size
B_{raw}	disk bandwidth
M	segment size
N	number of threads

Let M and c_{min} be the file size and the minimum chunk size. In single threaded chunking, the expected number of signatures generated, l_s, can be formulated as in Eq. 3.

$$l_{slow} = M - \frac{M}{E[S]} \cdot c_{min} \quad (3)$$

For each chunk, the chunker thread skips the first c_{min} byte and then computes signatures for all byte positions until the chunk boundary is set. The number of chunks in a file is $\frac{M}{E[S]}$. $\frac{M}{E[S]}c_{min}$ corresponds to the number of byte positions where signatures are *not* computed.

5.2 Signature Generation in Multithreaded Chunking

We computed the number of signature generations in multithreaded chunking. Multithread chunking has a notion of a *segment*. Let M be the size of a segment. Let l_{MUCH} be the number of signature generations for a segment. Let l_{slow} and l_{acc} be the number of signature generations in slow mode and in accelerated mode, respectively. In slow mode, signatures are computed in all byte positions. The number of signature generations in the chunker thread of MUCH can be formulated as in Eq. 4.

$$l_{MUCH} = l_{slow} + l_{acc} = M - \frac{M - l_{slow}}{E[S]} \cdot c_{min} \quad (4)$$

The number of signature generations in accelerated mode can be obtained in the same manner as in Eq. 3. However, the size of the accelerated mode region is the size of a segment subtracted by the size of the slow mode region. The number of signature computations

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2014.2322600, IEEE Transactions on Computers

in accelerated mode, l_a, can be formulated as in Eq. 5.

$$l_{acc} = (M - l_{slow}) - \frac{M - l_{slow}}{E[S]} \cdot c_{min} \quad (5)$$

Now, we delve into finding the expected size of the slow mode region. In slow mode, if a marker chunk is found, chunking mode changes from the slow to the accelerated mode. Let p_m be the probability that a chunk satisfies the chunk size requirement of marker chunk in Definition 4.3, i.e., the chunk size is larger than or equal to the minimum chunk size and smaller than or equal to (max size - min size). Then, p_m can be formulated as in Eq. 6.

$$p_m = \sum_{i=c_{min}}^{c_{max}-c_{min}} (1-p)^{i-1}p$$
$$=(1-p)^{c_{min}-1} - (1-p)^{c_{mark}}, \text{ where } c_{mark} = c_{max} - c_{min} \quad (6)$$

We computed the expected number of chunks in slow mode. Let $P(M = i)$ be the probability that i^{th} chunk is the marker. From the definition of the marker chunk, the first chunk cannot be the marker. Then, $P(M = 1) = 0$, $P(M = 1) = p_m, \ldots$. The probability that the i^{th} chunk is the marker is formulated as $P(M = i) = (1-p_m)^{i-2}p_m, i \geq 2$. The expected number of chunks in the slow mode region can be formulated as $E[M] = \sum_{i\geq 2} iP(M = i)$ and can be summarized as in Eq. 7.

$$E[M] = \frac{1}{p_m} + 1 \quad (7)$$

There is another interpretation of Eq. 7. The probability that there are i non marker chunks at slow mode can be formulated as $(1-p_m)^{i-1}p_m$. Then, the expected number of non marker chunks in the slow mode region can be formulated as $\sum_{i=1}^{\infty} i \cdot (1-p_m)^{i-1}p_m = \frac{1}{p_m}$. In the slow mode region, there is one marker chunk and $\frac{1}{p_m}$ number of non marker chunks on average.

We computed the expected chunk size in slow mode. Let S_b and S_m be the size of a non-marker chunk and the size of a marker chunk in the slow mode region, respectively. The probability that the size of a chunk is i, given that it is a marker, is a conditional probability. Therefore $P(S_m = i)$ can be formulated as $P(S_m = i) = \frac{P(S=i)}{p_m} = \frac{(1-p)^{i-1}p}{(1-p)^{c_{min}-1}-(1-p)^{c_{mark}}}$. The expected size of a marker chunk, $E[S_m]$, can be formulated as in Eq. 8. Details of derivations are in Appendix A.

$$E[S_m] = \frac{1}{(1-p)^{c_{min}-1} - (1-p)^{c_{mark}}} \cdot \Big(c_{min}(1-p)^{c_{min}-1} + \frac{1}{p}(1-p)^{c_{min}} - (c_{max} - c_{min} + \frac{1}{p})(1-p)^{c_{max}+c_{min}}\Big) \quad (8)$$

In a similar manner, we computed the expected size of non-marker chunks in slow mode. The probability that the size of a non-marker chunk is i corresponds to $P(S_b = i) = \frac{P(S=i)}{((1-p)^{w-1}-(1-p)^{c_{min}-1})+((1-p)^{c_{mark}}-(1-p)^{M+2})} = \frac{(1-p)^{i-1}p}{((1-p)^{w-1}-(1-p)^{c_{min}-1})+((1-p)^{c_{mark}}-(1-p)^{M+2})}$. A signature is computed for every w byte window. In our implementation, the window size is set to 48 Byte. The expected size of non-marker chunks, $E[S_b]$, in slow mode can be formulated as in Eq. 9. Details of derivations are in Appendix A.

$$E[S_b] = \Big(w(1-p)^{w-1} + \frac{1}{p}(1-p)^w - (1-p)^{c_{min}-1}(\frac{1}{p} + c_{min} - 1) + (c_{mark}+1)(1-p)^{c_{mark}} + \frac{1}{p}((1-p)^{c_{mark}+1} - (1-p)^{M+1}) - M(1-p)^{M+1}\Big) \cdot \frac{1}{((1-p)^{w-1} - (1-p)^{c_{min}-1}) + ((1-p)^{c_{mark}} - (1-p)^{M+2})} \quad (9)$$

In slow mode, the chunker thread computes signatures for every byte position and, therefore, the number of signature generations equals to the size of the slow mode region. The number of chunks in the slow mode region corresponds to $\frac{1}{p_m}$ and the last chunk in the slow mode region is the marker chunk. $\frac{1}{p} + w - 1$ is the average size of the first chunk of a segment in slow mode chunking. Finally, the number of signature generations in the slow mode region, l_{slow}, can be formulated as in Eq. 10.

$$l_{slow} = (\frac{1}{p_m} - 1)E[S_b] + E[S_m] + (\frac{1}{p} + w - 1) \quad (10)$$

Comparing Eq. 3 and Eq. 4, MUCH introduces additional signature generations, $\frac{l_{slow}}{E[S]} \cdot c_{min}$. This additional overhead is caused by Slow Mode chunking, which does not impose the minimum nor the maximum size requirements on the chunk sizes. With large segments, this overhead becomes less significant since relatively smaller portion of the segments is processed in Slow Mode chunking. The normalized multithreaded chunking overhead can be denoted by $\gamma = \frac{l_{MUCH} - l_{single}}{l_{single}} = \frac{\frac{l_{slow}}{E[S]} \cdot C_{min}}{l_{single}}$. We visualized the normalized multithreaded chunking overhead in Fig. 8. For segments larger than 200 KByte, the overhead is less than 4%. This additional overhead is offset by the performance improvement obtained from chunking multiple segments in a parallel manner.

We verified the accuracy of our model Eq. 4 through physical experiments. We examined the length of slow mode region and the number of chunks in slow mode under varying target pattern sizes. We set the minimum and the maximum chunk sizes at 2 KByte and 64 KByte, respectively. The degree of multithreading is 4 and the file size is 1 GByte. Fig. 9 illustrates the results. Fig. 9a and Fig. 9b illustrate the length of the slow mode region and the number of chunks in the slow mode region from the analytical models (Eq. 10) and from physical experiments. As shown in Fig. 9,

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2014.2322600, IEEE Transactions on Computers

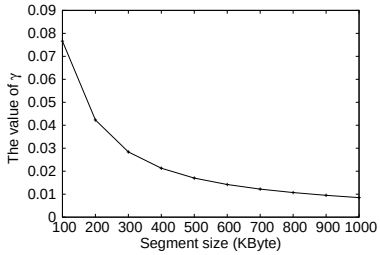
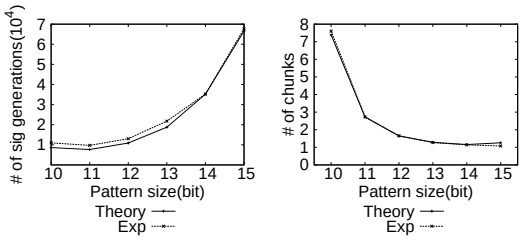


Fig. 8. Normalized multithreaded chunking overhead



(a) No. of signature generations (b) No. of Chunks

Fig. 9. Slow mode model validation: File size = 250 MByte, c_{min}=2 KByte, c_{max}=64 KByte, segment size = 100 KByte

the performance model in Eq. 10 precisely models the real world behavior.

Let N be the number of threads. The number of signature generations in each thread corresponds to l_{MUCH}/N. Then, the normalized performance improvement from multithreaded chunking can be formulated as $l_{MUCH}/N \cdot l_{single}$.

5.3 Chunking Performance in MUCH

Three components constitute the overhead of multicore chunking: (i) chunking overhead, T_{chunk}, (ii) time to fetch a segment set from the disk overhead, T_{disk}, and (iii) chunk marshalling overhead, $T_{marshall}$. Chunking is performed by the chunker thread while segment set fetching and chunk marshalling are performed by the master thread. We developed a performance model for multicore chunking. Without loss of generality, we computed the overhead for a single segment set.

Let λ be the time to compute one signature. Then, the time to chunk a segment can be modeled as in Eq. 11. where M is the size of a segment.

$$\begin{aligned} T_{chunking} &= \lambda \cdot l_{MUCH} \\ &= \lambda \cdot (l_{slow} + l_{acc}) \\ &= \lambda \cdot (M - (M - l_{slow}) \cdot \frac{c_{min}}{E[S]}) \end{aligned} \tag{11}$$

With M byte segment and N chunker threads, the segment set fetching overhead, T_{disk}, corresponds to Eq. 12, where δ and B_{raw} are the average disk overhead (seek and rotational latency) and the maximum disk bandwidth, respectively. We assumed that a single segment is not fragmented, which is not an unreasonable assumption given the aggressive block pre-allocation policy of modern filesystems[43] and the modeling scheme of existing disk scheduling efforts[44].

$$T_{disk} = \delta + \frac{M \cdot N}{B_{raw}}, B_{raw} = \text{sequential read bandwidth} \tag{12}$$

In chunk marshalling, the number of coalescing operations is bounded by the total number of chunks generated in the slow mode. $T_{clsc} = \zeta \cdot \frac{1}{p_m} \cdot N$, ζ = time to coalesce two segments. Finally, the overhead of marshalling, $T_{marshall}$, corresponds to $T_{mshl} = T_{clsc} + T_{split}$, where the time for split operation, T_{split}, is negligible. The chunking bandwidth of MUCH can be formulated as in Eq. 13.

$$B_{MUCH} = \frac{M \cdot N}{\max\left\{T_{chunking}, T_{disk}\right\} + T_{marshall}} \tag{13}$$

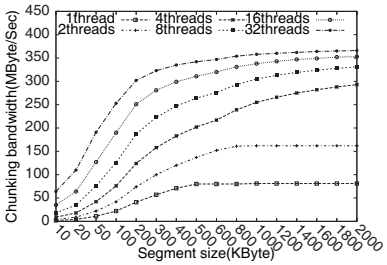


Fig. 10. Chunking bandwidths under varying segment sizes: $B_{max} = 280MByte/sec$, $\delta = 4msec$, $c_{min} = 2KByte$, $c_{max} = 64KByte$, target pattern size: 13 bit, $\lambda = 14nsec$

6 Storage speed and The number of CPU cores

It is not practical to increase the segment size beyond a certain threshold value since the additional performance gain becomes marginal. To address this issue, we modeled the theoretical upper bound on chunking bandwidth with infinite size segment and then obtained the segment size which achieves ρ fraction of the ideal chunking speed.

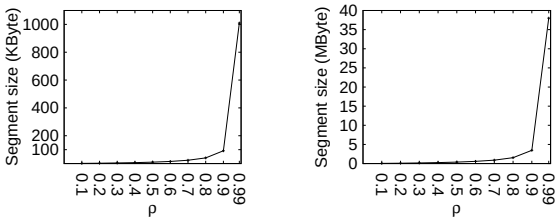
Let B^*_{chunk} and B^*_{IO} be the ideal bandwidth when the overall throughput is governed by the CPU and I/O, respectively. They can be formulated as in Eq. 14 and Eq. 15, respectively.

$$B^*_{chunking} = \frac{E[S]}{\zeta + \lambda(E[S] - C_{min})} \cdot N \tag{14}$$

$$B^*_{IO} = B_{raw} \tag{15}$$

Let ρ be the ratio between the actual bandwidth and the ideal bandwidth, i.e., $\rho = \frac{B_{chunking}}{B^*_{chunking}}$ or $\rho = \frac{B_{IO}}{B^*_{IO}}$. Let $M_{chunking}(\rho)$ and $M_{IO}(\rho)$ be the segment sizes to achieve

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2014.2322600, IEEE Transactions on Computers



(a) CPU bound (b) Disk bound

Fig. 11. The optimal segment size

utilization ρ for CPU bound systems and IO bound systems, respectively. Then, $M_{chunking}(\rho)$ and $M_{IO}(\rho)$ can be formulated as in Eq. 16 and Eq. 17, respectively.

$$M_{chunking}(\rho) = \frac{\rho(E[S](\zeta + \frac{\zeta}{P_m}) - l_{slow}(\zeta - \lambda C_{min}))}{(1-\rho)(\zeta + \lambda(E[S] - C_{min}))} \quad (16)$$

$$M_{IO}(\rho) = \frac{\rho B_{raw}(\delta + N\zeta\frac{1}{P_M})}{N(1-\rho)} \quad (17)$$

Fig. 11 illustrates the segment sizes under varying number of threads. When a system is CPU bound (Fig. 11a), the segment size is not significantly governed by the number of threads. When the system is I/O bound, it is not necessary to increase the segment set size beyond a certain threshold point. The size of a single segment decreases as the number of threads in multicore chunking increases.

Based on Eq. 13, we can compute the effective number of threads to saturate a given I/O interface. We assumed that the time to generate a single signature corresponds to 14 nsec. It is obtained from Intel Core i5 CPU (2.80 Ghz). We used INC-K algorithm[1]. Up to an IO bandwidth of 100 MByte/sec, single thread is sufficient to fully exploit the IO bandwidth. For SATA 2.0 and SAS 300 interfaces, we needed to allocate four chunker threads. For SATA 3.0 (600 MByte/sec) drive, we needed to allocate three threads. For FC over optical fiber (2 GByte/sec), we needed 32 cores; We needed to exploit a different approach, e.g., GPGPU, instead of relying on CPU cores, to fully utilize the underlying IO bandwidth.

7 Experiment

7.1 Experiment Setup

We implemented MUCH in a prototype deduplicated backup system, PRUNE[11]. 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 tree (totaling 341 MByte). Table 2 summarizes the characteristics of the data sets used in our experiment. We examined MUCH under three storage devices with different performance characteristics: RAMDISK, high performance SSD, and RAID 0 based storage device

TABLE 2
Datasets

Set	Size	Type	Size (MB)	Avg. (MB)
ISO img.	2GB	iso	2048	
Mixed Files	200GB	iso	66764.8	551.7
		rar	48742.4	18.1
		avi	61030.4	1245.5
		zip	13516.8	281.6
		jpg	4300.8	1.9
		exe	7372.8	37.8
		cab	3276.8	16.1
		all	204853.1	36.9
Linux	341.2MB	*c,*h	341.2	0.011

TABLE 3
Specifications of three storage devices

Type	Vendor	Model
RAMDISK	SAMSUNG	DDR3, 1333 MHz
SSD	OCZ	REVODRIVE3X2 (PCI-E, 480GByte)
RAID 0	Seagate	ST31000528AS x 3EA (SATA2, 1Tbyte HDD)

TABLE 4
Various statistical characteristics of filesystems

	Average	Median
web server	207.7 KByte	7.1 KByte
e-learning server	173.3 MByte	1.5 KByte
DB server	29.9 KByte	7.3 KByte
mail server	306.6 KByte	2.8 KByte

that consists of three hard disk drives (Table 3). We ran MUCH on Intel Core i5 (2.80 GHZ, quad-core) with 4 GByte of DRAM and Linux 2.6.32 with EXT4 filesystem.

Yoon[45] 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 web-server, e-learning server, DB server and mail server correspond to 7.1 KByte, 1.5 KByte, 7.3 KByte, 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. Table 4 summarizes the statistics.

7.2 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

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2014.2322600, IEEE Transactions on Computers

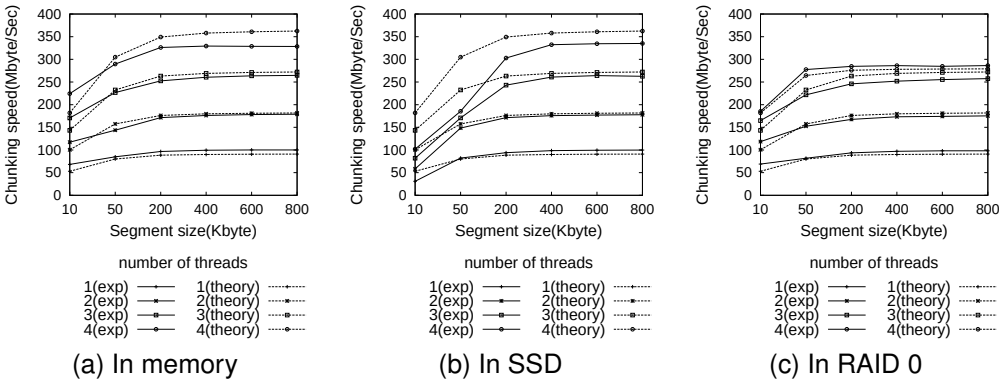


Fig. 12. Chunking bandwidths with ISO image

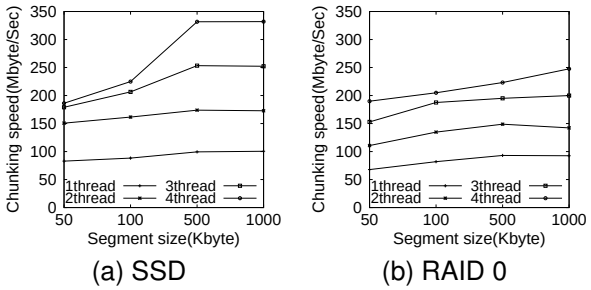


Fig. 13. Chunking bandwidths with mixed files

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 KByte to 800 KByte. Fig. 12a, Fig. 12b, and Fig. 12c illustrate the chunking bandwidths under varying segment size for RAMDISK, SSD, and RAID 0, respectively. It plots the results 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% 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[13] 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[4], [39], [40].

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 (Fig. 12c).

7.3 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 5546 files and the average file size was 36.9 MByte. Fig. 13a and Fig. 13b illustrate the results. For the SSD, with four chunker threads and 500 KByte segment size, the chunking speed reaches 350 MByte/sec. Chunking speed increased by 350% with four chunker threads compared to single thread. It is still 70% 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% of the sustained sequential read bandwidth.

7.4 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% of files are smaller than 20 KByte.

Fig. 14a, Fig. 14b, and Fig. 14c illustrate the results of our experiment. As we can see, 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 constitutes a significant fraction of the entire chunking overhead and, therefore, the benefit of employing parallelism becomes marginal.

7.5 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. Fig. 15 illustrates the results of the experiment. 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 (Fig 16c), employing multiple threads barely helps the chunking performance in an SSD and RAID 0 storage devices.

7.6 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. Fig. 17 illustrates the results. When a file is in the memory, DSSP significantly improves chunking bandwidth. With a 500 KByte file, chunking speed increased by 50% 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%. Given that most of the files we use are less than 1 MByte, DSSP is a critical mechanism to expedite the chunking process.

Fig. 18 and Fig. 19 illustrate 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%.

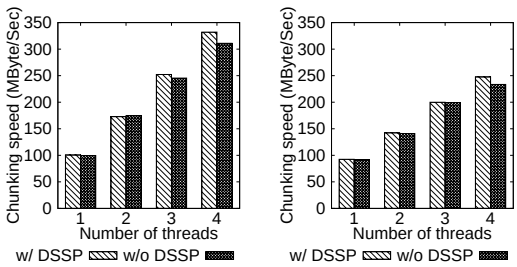


Fig. 18. Effect of multithreading degree in DSSP, mixed files

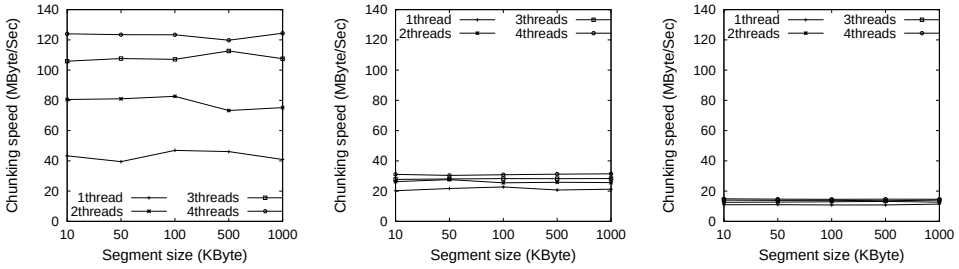
7.7 Multicore Chunking in Smartphones

Recently introduced high-end smartphones are equipped with four cores[46] or eight cores[47]. 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 MByte/sec 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. 20b), 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. 20 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 x4 and achieved up to 350 MByte/sec chunking performance. We increased the storage bandwidth utilization from 20% to 70%.

8 Conclusion

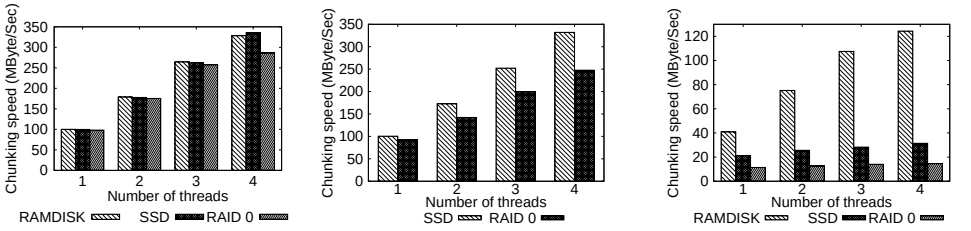
In this work, we proposed a novel multicore chunking algorithm, MUCH, which parallelizes the variable size chunking. To date, most of the existing works on deduplication focus on expediting the redundancy detection process, while less attention has been paid on how to make the file chunking faster. We found that variable size chunking is computationally very expensive and is a significant bottleneck in the overall

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2014.2322600, IEEE Transactions on Computers



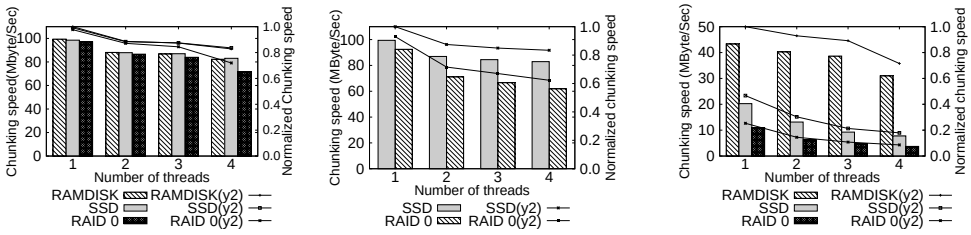
(a) In RAMDISK (b) In SSD (c) In RAID 0

Fig. 14. Chunking bandwidths with Linux source tree



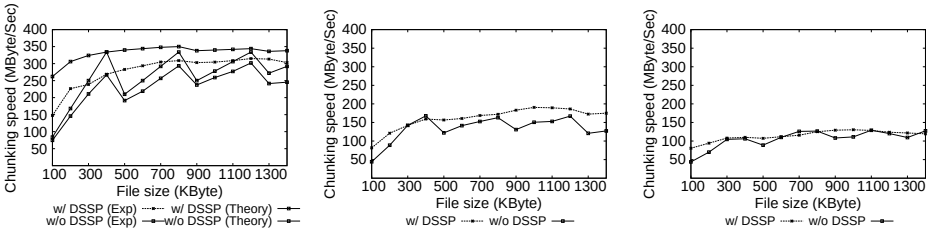
(a) ISO image, M = 800KByte (b) Mixed files, M = 1000KByte (c) Linux source tree, M = 50KByte

Fig. 15. Chunking bandwidths vs. multithreading degree (M: segment size)



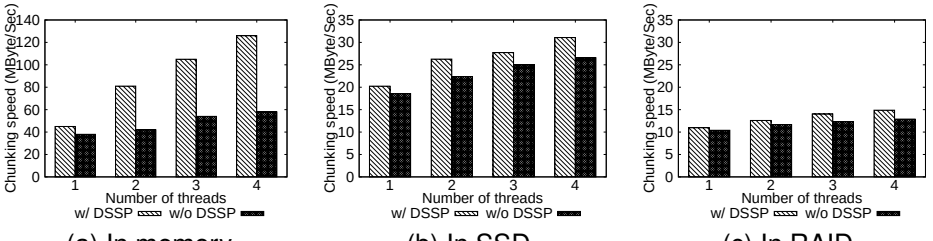
(a) ISO image (b) Mixed files (c) Linux source tree

Fig. 16. Scalability of multithreading



(a) In memory (b) In SSD (c) In RAID 0

Fig. 17. Performance of DSSP vs. file size



(a) In memory (b) In SSD (c) In RAID

Fig. 19. Effect of the number of threads in DSSP, Linux source tree

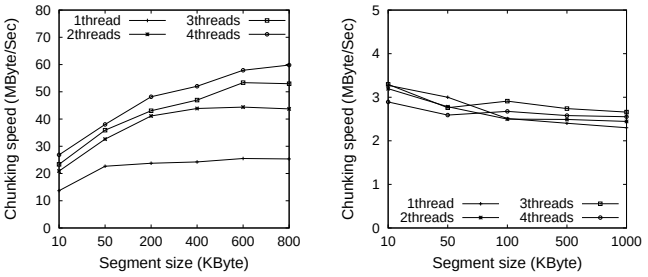


Fig. 20. MUCH in a smartphone (Galaxy S3, Exynos 4412 quad-core (1.4GHz), Android 4.1.2, Samsung KMVTU000LM eMMC (16GB))

deduplication process. The performance gap between the CPU chunking speed and the I/O bandwidth is expected to become wider with the recent introduction of faster I/O interconnections, and faster storage devices.

Incorporating this technology advancement, i.e., increase in the number of CPU cores and emergence of faster storage devices, we developed a parallel chunking algorithm, which aims at making the variable chunking speed on par with the storage I/O bandwidths. We found that the legacy variable size chunking algorithm yields a different set of chunks if the parallelism degree changes, a phenomenon referred to as *Multithreaded Chunking Anomaly*. We propose a multicore chunking algorithm, *MUCH*, which guarantees *Chunking Invariability*. We developed a performance model to compute the segment size that maximizes the chunking bandwidth while minimizing the memory requirement. Through extensive physical experiments, we showed that the performance of MUCH scales linearly with the number of cores. In quad-core CPUs, MUCH brings a 400% performance increase when the storage device is sufficiently fast. The benefits of MUCH are evident when it chunks large files, e.g., tar images of filesystem snapshot, at high performance storage. MUCH successfully increases the chunking performance with the factor being as high as the number of available CPU cores without any additional hardware assistance.

Acknowledgments

This work is supported in part by IT R&D program MKE/KEIT (No. 10041608, Embedded System Software for New-memory based Smart Device).

References

[1] Y. Won, R. Kim, J. Ban, J. Hur, S. Oh, and J. Lee, "Prun: Eliminating information redundancy for large scale data backup system," in *Proceedings of IEEE International Conference on Computational Sciences and Its Applications(ICCSA '08)*, Perugia, Italy, March 2008.

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

[3] M. Mitzenmacher, "Compressed bloom filters," *IEEE/ACM Trans. on Networking*, vol. 10, no. 5, pp. 604–612, October 2002.

[4] 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 *Proceedings of Mass Storage Systems and Technologies (MSST), 2012 IEEE 28th Symposium on*. Pacific Grove, CA, USA: IEEE, April 2012, pp. 1–12.

[5] F. Chen, T. Luo, and X. Zhang, "Caftl: A content-aware flash translation layer enhancing the lifespan of flash memory based solid state drives," in *Proceedings of the 9th USENIX conference on File and storage technologies*, San Jose, CA, USA, February 2011, pp. 77–90.

[6] A. Gupta, R. Pisolkar, B. Urgaonkar, and A. Sivasubramaniam, "Leveraging value locality in optimizing nand flash-based ssds," in *Proceedings of the 9th USENIX Conference on File and Storage Technologies*, San Jose, CA, USA, February 2011, pp. 91–104.

[7] M. Ajtai, R. Burns, R. Fagin, D. Long, and L. Stockmeyer, "Compactly encoding unstructured input with differential compression," *Journal of the ACM(JACM)*, vol. 49, no. 3, pp. 318–367, May 2002.

[8] F. Douglis and A. Iyengar, "Application-specific delta-encoding via resemblance detection," in *Proceedings of USENIX 2003*, San Antonio, TX, USA, June 2003.

[9] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky, "Silt: A memory-efficient, high-performance key-value store," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. ACM, 2011, pp. 1–13.

[10] B. Debnath, S. Sengupta, and J. Li, "Chunkstash: speeding up inline storage deduplication using flash memory," in *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, Boston, MA, USA, June 2010, pp. 215–229.

[11] Y. Won, J. Ban, J. Min, J. Hur, S. Oh, and J. Lee, "Efficient index lookup for de-duplication backup system," in *Proceedings of Modeling, Analysis and Simulation of Computers and Telecommunication Systems, 2008. MASCOTS 2008. IEEE International Symposium on*, Baltimore, MD, USA, September 2008, pp. 1–3(Poster Paper).

[12] A. Badam and V. S. Pai, "Ssdalloc: hybrid ssd/ram memory management made easy," in *Proceedings of the 8th USENIX conference on Networked systems design and implementation*. USENIX Association, 2011, pp. 16–16.

[13] Ocztechnology, "Ocz revo3 x2 product sheet," http://www.ocztechnology.com/.

[14] M. Lillibridge, K. Eshghi, D. Bhagwat, V. Deolalikar, G. Trezise, and P. Camble, "Sparse Indexing: Large Scale, Inline Deduplication Using Sampling and Locality," in *Proceedings of the 7th USENIX Conference on File and Storage Technologies (FAST'09)*, San Francisco, CA, USA, February 2009, pp. 111–124.

[15] C. Van Berkel, "Multi-core for mobile phones," in *Proceedings of the Conference on Design, Automation and Test in Europe*, Nice, France, March 2009, pp. 1260–1265.

[16] J. Min, D. Yoon, and Y. Won, "Efficient deduplication techniques for modern backup operation," *Computers, IEEE Transactions on*, vol. 60, no. 6, pp. 824–840, 2011.

[17] MacroImpact, "Sanique smart vault," http://www.macroimpact.com/subpage.php?p=m14.

[18] B. Hong, D. Plantenberg, D. Long, and M. Sivan-Zimet, "Duplicate data elimination in a SAN file system," in *Proceedings of the 12th NASA Goddard, 21st IEEE Conference on Mass Storage Systems and Technologies (MSST 2004)*, Greenbelt, MD, USA, April 2004, pp. 301–314.

[19] A. Muthitacharoen, B. Chen, and D. Mazières, "A low-bandwidth network file system," *SIGOPS Oper. Syst. Rev.*, vol. 35, no. 5, pp. 174–187, 2001.

[20] P. Reynolds and A. Vahdat, "Efficient Peer-to-Peer Keyword Searching," *LECTURE NOTES IN COMPUTER SCIENCE*, vol. 2672, no. 2, pp. 21–40, 2003.

[21] A. Chervenak, V. Vellanki, and Z. Kurmas, "Protecting file systems: A survey of backup techniques," in *Proceedings of Joint NASA and IEEE Mass Storage Conference*, vol. 3, College Park, MA, USA, March 1998.

[22] C. Policroniades and I. Pratt, "Alternatives for detecting redundancy in storage systems data," in *Proceedings of the 2004 USENIX Annual Technical Conference*, Boston, MA, USA, June 2004, pp. 73–86.
[23] W. Tichy, "Rcs: a system for version control," *Software Practice & Experience*, vol. 15, no. 7, pp. 637–654, July 1985.
[24] K. Eshghi and H. Tang, "A framework for analyzing and improving content-based chunking algorithms," *Hewlett-Packard Labs Technical Report TR*, vol. 30, 2005.
[25] D. Meister and A. Brinkmann, "Multi-level comparison of data deduplication in a backup scenario," in *SYSTOR '09: Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*. Haifa, Israel: ACM, May 2009, pp. 1–12.
[26] M. Rabin, *Fingerprinting by random polynomials*. Center for Research in Computing Techn., Aiken Computation Laboratory, Univ., 1981.
[27] V. Henson, "An analysis of compare-by-hash," in *Proceedings of the 9th Workshop on Hot Topics in Operating Systems (HotOS IX)*, Lihue, Hawaii, USA, May 2003, pp. 13–18.
[28] S. Hanselman and M. Pegah, "The wild wild waste: e-waste," in *Proceedings of the 35th annual ACM SIGUCCS Fall Conference*, Lake Buena Vista, FL, USA, October 2007, pp. 157–162.
[29] "Final rule: Retention of records relevant to audits and reviews, securities and exchange commission, 17 cfr part 210, [release nos. 33-8180; 34-47241; ic-25911; fr-66; file no. s7-46-02], rin 3235-ai74, retention of records relevant to audits and reviews."
[30] W. H. Roach Jr, *Medical records and the law*. Jones & Bartlett Publishers, 2006.
[31] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, 1970.
[32] W. Pugh, "Skip lists: a probabilistic alternative to balanced trees," *Communications of the ACM*, vol. 33, no. 6, pp. 668–676, 1990.
[33] S. Ghemawat, H. Gobioff, and S. Leung, "The Google file system," *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5, p. 43, 2003.
[34] J. Hamilton and E. Olsen, "Design and implementation of a storage repository using commonality factoring," in *Mass Storage Systems and Technologies, 2003 (MSST 2003). Proceedings of 20th IEEE/11th NASA Goddard Conference on*. San Diego, CA, USA: IEEE, April 2003, pp. 178–182.
[35] D. Bhagwat, K. Eshghi, D. Long, and M. Lillibridge, "Extreme binning: Scalable, parallel deduplication chunk-based file backup," in *Proceedings of the 17th IEEE/ACM International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems(MASCOTS 2009)*, London, UK, September 2009.
[36] D. Bhagwat, K. Pollack, D. D. E. Long, T. Schwarz, E. L. Miller, and J.-F. Paris, "Providing high reliability in a minimum redundancy archival storage system," in *MASCOTS '06: Proceedings of the 14th IEEE International Symposium on Modeling, Analysis, and Simulation*, Monterey, CA, USA, September 2006, pp. 413–421.
[37] C. Liu, Y. Gu, L. Sun, B. Yan, and D. Wang, "R-admad: high reliability provision for large-scale de-duplication archival storage systems," in *ICS '09: Proceedings of the 23rd international conference on Supercomputing*, San Francisco, CA, USA, September 2009, pp. 370–379.
[38] Y. Zhang, Y. Wu, and G. Yang, "Droplet: A distributed solution of data deduplication," in *Grid Computing (GRID), 2012 ACM/IEEE 13th International Conference on*. IEEE, 2012, pp. 114–121.
[39] Z. Tang and Y. Won, "Multithread content based file chunking system in cpu-gpgpu heterogeneous architecture," in *Proceedings of Data Compression, Communications and Processing (CCP), 2011 First International Conference on*, Palinuro, Italy, Jun 2011, pp. 58–64.
[40] C. Kim, K.-W. Park, and K. H. Park, "Ghost: Gpgpu-offloaded high performance storage i/o deduplication for primary storage system," in *Proceedings of the 2012 International Workshop on Programming Models and Applications for Multicores and Manycores*. ACM, 2012, pp. 17–26.
[41] R. Sudarsan, S. Fenves, R. Sriram, and F. Wang, "A product information modeling framework for product lifecycle management," *Computer-Aided Design*, vol. 37, no. 13, pp. 1399–1411, 2005.
[42] S. Quinlan and S. Dorward, "Venti: A new approach to archival storage," in *Proceedings of FAST '02*, Monterey, CA, USA, January 2002, pp. 89–101.
[43] A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, and L. Vivier, "The new ext4 filesystem: current status and future plans," in *Proceedings of the Linux Symposium*, vol. 2. Citeseer, 2007, pp. 21–33.
[44] J. Bruno, J. Brustoloni, E. Gabber, B. Ozden, and A. Silberschatz, "Disk scheduling with quality of service guarantees," in *Multimedia Computing and Systems, 1999. IEEE International Conference on*, vol. 2. IEEE, 1999, pp. 400–405.
[45] Y. D. Young, "Empirical study for efficient deduplication methods." Hanyang Univ., 2011.
[46] SAMSUNG. Samsung galaxy s3 tech specs. http://www.samsung.com/africa_en/consumer/mobile-phone/mobile-phone/smart-phone/GT-I9305RWDXFA-spec.
[47] ——. Samsung galaxy s4 tech specs. http://www.samsung.com/africa_en/consumer/mobile-phone/mobile-phone/smart-phone/GT-I9500ZWEXFA-spec.

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 Division of Computer Science and Engineering, 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.

Kyeongyeol Lim received the BS degree in Information & Communication from Baekseok University in 2011. He is currently working toward the MS degree in the Division of Computer Science and Engineering, Hanyang University, Seoul, Korea. His research interests include deduplication systems and operating systems.

Jaehong Min 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 worked on developing deduplication based backup software as software engineer at MacroImpact Co. for three years. He is working on designing software architecture at Samsung Electronics.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2014.2322600, IEEE Transactions on Computers

APPENDIX A
EXPECTED CHUNK SIZE

$$
\begin{aligned}
E[S_m] &= \sum_{i=c_{min}}^{c_{max}-c_{min}} i \cdot P(S_m = i) \\
&= \sum_{i=c_{min}}^{c_{max}-c_{min}} i \cdot \frac{(1-p)^{i-1}p}{(1-p)^{c_{min}-1} - (1-p)^{c_{mark}}} \\
&= \frac{p}{(1-p)^{c_{min}-1} - (1-p)^{c_{mark}}} \sum_{i=c_{min}}^{c_{max}-c_{min}} i \cdot (1-p)^{i-1} \\
&= \frac{1}{(1-p)^{c_{min}-1} - (1-p)^{c_{mark}}} \cdot (c_{min}(1-p)^{c_{min}-1} \\
&\quad + \frac{1}{p}(1-p)^{c_{min}} - (c_{max} - c_{min} + \frac{1}{p})(1-p)^{c_{max}+c_{min}})
\end{aligned}
\tag{18}
$$

$$
\begin{aligned}
E[S_b] &= \sum_{i=w}^{c_{min}-1} iP(S_b = i) + \sum_{i=c_{mark}+1}^{M} iP(S_b = i) \\
&= \sum_{i=w}^{c_{min}-1} i \cdot \frac{(1-p)^{i-1}p}{((1-p)^{w-1} - (1-p)^{c_{min}-1}) + ((1-p)^{c_{mark}} - (1-p)^{M+2})} \\
&\quad + \sum_{i=c_{mark}+1}^{M} i \cdot \frac{(1-p)^{i-1}p}{((1-p)^{w-1} - (1-p)^{c_{min}-1}) + ((1-p)^{c_{mark}} - (1-p)^{M+2})} \\
&= \frac{1}{((1-p)^{w-1} - (1-p)^{c_{min}-1}) + ((1-p)^{c_{mark}} - (1-p)^{M+2})} \\
&\quad (w(1-p)^{w-1} + \frac{1}{p}(1-p)^{w} - (1-p)^{c_{min}-1}(\frac{1}{p} + c_{min} - 1) + (c_{mark} + 1) \\
&\quad (1-p)^{c_{mark}} + \frac{1}{p}((1-p)^{c_{mark}+1} - (1-p)^{M+1}) - M(1-p)^{M+1})
\end{aligned}
\tag{19}
$$

APPENDIX B
PROOF OF CHUNKING INVARIABILITY

We prove that Dual Mode Chunking and Inter-Segment Chunk Coalescing guarantee chunk invariability. Let $b(c)$ and $e(c)$ be the begin and end of a chunk and is represented by the file offset.

Definition For chunk c and segment S, $c \subset S$ if $b(c) \geq b(S)$ and $e(c) \leq e(S)$.

Definition Two chunks c_i and c_j are *identical* and is denoted by $c_i = c_j$ if $b(c_i) = b(c_j)$ and $e(c_i) = e(c_j)$.

Theorem 2: Let S be a segment. Let $B = \{b_1, \ldots, bx_l\}$ and $C = \{c_1, \ldots, c_n\}$ be a set of chunks generated by slow mode chunking and single threaded chunking from S. If we apply chunk coalescing to B, then for a set of coalesced chunks, B^*, $B^* = C$.
We will skip the proof of lemma 2.

Theorem 3: For a given file, let $M = \{m_1, m_2, \ldots, m_l\}$ and $C = \{c_1, c_2, \ldots, c_n\}$ be a set of chunks created by multithreaded chunking algorithm, MUCH, and single threaded chunking, respectively. Then, $M = C$.

Proof:

If a file consists of one segment, the theorem holds from Lemma 2. We assume that a file consists of more than one segment. The proof is by induction on the number of chunks in a segment. $c_i \subset S_i$ means that chunk c_i belongs to segment S_i. When c_i lies across two segments, S_k and S_{k+1}, $c_i^L = c_i \cap S_k$ and $c_i^R = c_i \cap S_{k+1}$.

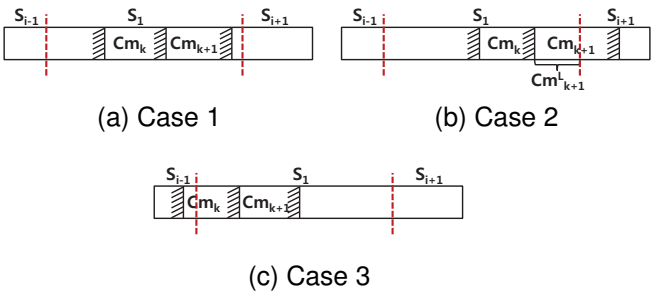


Fig. 21. Cases of theorem

Basis: $m_1 = c_1$.
In MUCH, the first segment in a file is processed in accelerated mode as in the case of single threaded chunking. Hence the claim.

Induction: Assume that $c_i = m_i, i = 1, \ldots, k$. We will show $c_{k+1} = m_{k+1}$. We categorize the situations into three cases based on the relative position of c_k and c_{k+1} in a segment (Fig. 21).

Case 1: $c_k \subset S_j$ and $c_{k+1} \subset S_j$ for $\exists S_j$.

Assume that m_{k+1} is created in accelerated mode. Since $b(m_{k+1}) = b(c_{k+1})$ and single and multithread chunking algorithms use the same algorithm, the chunking results should be identical. Assume that m_{k+1} consists of the chunks from the slow mode. Intra-segment chunk coalescing coalesces the chunks in the slow mode until the resulting chunk satisfies the minimum and the maximum chunk size bounds. If $e(m_{k+1}) < e(c_{k+1})$, $e(c_{k+1})$ is not the first byte offset which matches the bit pattern where the chunk size is larger than or equal to the minimum chunk size. Contradiction. If $e(m_{k+1}) > e(c_{k+1})$, Intra-Segment Chunks Coalescing should be set the end position of m_{k+1} at $e(c_{k+1})$. Contradiction. Hence, when m_{k+1} consists of the chunks from the slow mode, $m_{k+1} = c_{k+1}$. Hence the claim.

Case 2: $c_k \subset S_j$ and $c_{k+1} \cap S_{j+1} \neq \phi$ $\exists j$.

Assume $c_{k+1} \neq m_{k+1}$. Then, there are two cases: $e(m_{k+1}) < e(c_{k+1})$, and $e(m_{k+1}) > e(c_{k+1})$. Assume $e(m_{k+1}) < e(c_{k+1})$. Let $m_1^b, ..., m_l^b$ denote the chunks generated in slow mode for S_{j+1} with m_l^b being the marker chunk. Master thread performs chunk coalescing $m_1^b, m_2^b, ..., m_p^b$ so that the newly created chunk becomes larger than or equal to the minimum chunk size. Then, $m_{k+1} = m_{k+1}^L \cup m_1^b \cup \ldots \cup m_p^b, p \leq l$. Then, $size(m_{k+1}^L \cup m_1^b \cup ... \cup m_p^b) \geq c_{min}$ and $size(m_{k+1}^L \cup m_1^b \cup ... \cup m_{p-1}^b) < c_{min}$ should hold. From definition of

single threaded chunking, $e(c_{k+1})$ is the first position after $b(c_{k+1})$ so that $size(c_{k+1}) \geq c_{min}$. Contradiction. Therefore, $e(m_{k+1}) \geq e(c_{k+1})$ should hold.

Assume $e(m_{k+1}) > e(c_{k+1})$. ${m^L}_{k+1} \cup m_1^b \cup ... \cup m_p^b \equiv m_{k+1}$ and $e(m_{k+1}) = e(m_p^b)$. Then, $e(m_L^b) < e(c_{k+1})$. $e(c_{k+1})$ is the first position after $b(c_{k+1})$ which satisfies the chunk boundary condition and $e(c_{k+1}) - b(c_{k+1}) \geq c_{min}$.

From the definition of Chunk Coalescing, $size({m^L}_{k+1} \cup m_1^b \cup \ldots \cup m_p^b) \geq c_{min} size({m^L}_{k+1} \cup m_1^b \cup \ldots \cup m_{p-1}^b < c_{min})$ should hold. Since $e(m_{k+1}) > e(c_{k+1})$, $e(c_{k+1}) = e(m_m^b), \exists m \leq p$. Then, $c_{k+1} = m_{k+1}^L \cup m_1^b \cup \ldots m^b m, m < p$ and $size(c_{k+1}) \geq c_{min}$. This contradicts $size({m^L}_{k+1} \cup m_1^b \cup \ldots \cup m_{p-1}^b < c_{min}$. Hence, $e(m_{k+1}) \leq e(c_{k+1})$. From $e(m_{k+1}) \leq e(c_{k+1})$ and $e(m_{k+1}) \geq e(c_{k+1})$, $e(c_{k+1}) = e(c_{k+1})$ and hence the claim.

Case 3: $c_k \cap S_j \neq \phi$ and $c_k \cap S_{j+1} \neq \phi$ and $c_{k+1} \subset S_{j+1}$.

Let us establish a new segment S'_{j+1} such that $b(S'_{j+1}) = b(c_{k+1})$. Then, both c_{k+1} and m_{k+1} become the first chunks of S'_{j+1}. If m_{k+1} is in the accelerated mode region, $m_{k+1} = c_{k+1}$. If m_{k+1} is in the slow mode region, from Lemma 2, $c_{k+1} = m_{k+1}$.

From case 1, 2 and 3, $m_{k+1} = c_{k+1}$ should hold and hence the claim. ■

□