SS-CDC: A Two-stage Parallel Content-Defined Chunking for Deduplicating Backup Storage

Fan Ni*
University of Texas at Arlington
Arlington, TX
fan.ni@mavs.uta.edu

Xing Lin
NetApp
Sunnyvale, CA
Xing.Lin@netapp.com

Song Jiang
University of Texas at Arlington
Arlington, TX
song.jiang@uta.edu

ABSTRACT

Data deduplication has been widely used in storage systems to improve storage efficiency and I/O performance. In particular, content-defined variable-size chunking (CDC) is often used in data deduplication systems for its capability to detect and remove duplicate data in modified files. However, the CDC algorithm is very compute-intensive and inherently sequential. Efforts on accelerating it by segmenting a file and running the algorithm independently on each segment in parallel come at a cost of substantial degradation of deduplication ratio.

In this paper, we propose SS-CDC, a two-stage parallel CDC, that enables (almost) full parallelism on chunking of a file without compromising deduplication ratio. Further, SS-CDC exploits instruction-level SIMD parallelism available in today's processors. As a case study, by using Intel AVX-512 instructions, SS-CDC consistently obtains superlinear speedups on a multi-core server. Our experiments using real-world datasets show that, compared to existing parallel CDC methods which only achieve up to a 7.7× speedup on an 8-core processor with the deduplication ratio degraded by up to 40%, SS-CDC can achieve up to a 25.6× speedup with no loss of deduplication ratio.

CCS CONCEPTS

• **Information systems → Deduplication**;

KEYWORDS

Storage, Deduplication, Content-defined-Chunking (CDC)

ACM Reference Format:
Fan Ni, Xing Lin, and Song Jiang. 2019. SS-CDC: A Two-stage Parallel Content-Defined Chunking for Deduplicating Backup Storage. In *The 12th ACM International Systems and Storage Conference (SYSTOR '19), June 3–5, 2019, Haifa, Israel*. ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3319647.3325834

1 INTRODUCTION

Backup storage is a critical infrastructure in protecting users from data loss incidents, such as incautious data deletion. To minimize the performance impact on production services, backup jobs are usually scheduled after midnights or during weekends. To complete backing up of a large amount of data within a tight time window, the system has to provide sufficiently high backup performance [11, 33]. The single-stream[1] backup throughput measures how fast a system can process one backup stream. With a higher single-stream backup throughput, the backup system can complete a backup job more quickly within the backup window. While many backup systems support concurrent backups, they usually have a limit on the maximum number of concurrent backup streams [8, 18] to prevent resource contention, which could degrade the performance of single-stream backups.

Using Deduplication to Improve Space Efficiency. Along with the backup performance, space efficiency is also an important aspect of a backup storage system. Backup files usually contain a large amount of duplicate data due to small changes between two consecutive backups. Accordingly, data deduplication is often used to detect and remove redundant data among backups. A data deduplication scheme partitions input files into small chunks and only unique chunks are stored in the system. Deduplication ratio, which is defined as the ratio of the original data size and the size after deduplication, is used to measure its effectiveness in removing duplicate data. Prior research [24] has demonstrated significant space saving from deduplication, achieving deduplication ratios from 2~14× in production deployments.

*The majority of the work was completed when the author was an intern at NetApp, Sunnyvale, CA.

[1] A backup client often creates a tar-like backup file and transfers backup files as backup streams to the backup system.

SYSTOR'19, June 2019, Haifa, Israel

ACM ISBN 978-1-4503-6749-3/19/06…$15.00
https://doi.org/10.1145/3319647.3325834

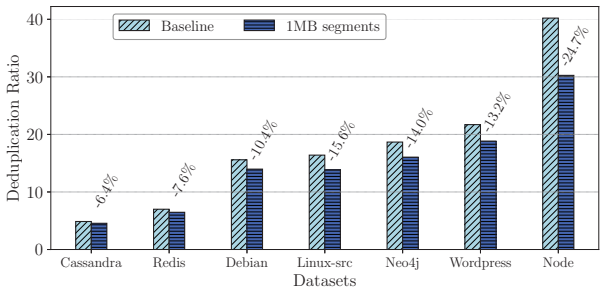


Figure 1: Deduplication ratio reduction caused by existing parallel CDC approaches.

However, deduplication also adds significant performance overhead to the system, especially with the variable-size chunking process that is commonly used in backup storage systems. A typical variable size chunking algorithm, such as content-defined chunking (CDC) [16], scans almost every byte in an input file using a fixed-size rolling window and calculates a hash value for each rolling window[2]. A chunk boundary is determined when two conditions are met. One is that the chunk size is within the range of pre-defined minimum and maximum chunk sizes. And the other is that the hash value of the rolling window matches a pre-defined value. As we need to calculate a hash value for the rolling window at almost every byte offset of a file, this process consumes significant CPU resource and has become a performance bottleneck in many backup storage systems [1, 2, 30].

Accelerating CDC-based Deduplication. To alleviate the bottleneck, many researchers have proposed to partition an input file into segments, and leverage parallel hardware, such as multi-core processors or GPGPU platforms, to perform chunking on the segments in parallel, termed as *parallel CDC* hereafter. While they receive performance benefit of parallelism to some extent, they have at least one of the two limitations. They do not provide either guarantee of *chunking invariability* [27] or compatibility to the SIMD platforms, such as Advanced Vector Extensions (AVX) [6] that is available in recent Intel and AMD processors or GPUs. We discuss each of the two limitations in the below.

Chunking invariability requires that a parallel chunking algorithm always generates the identical set of chunks independent of the parallelism degree and the segment size. However, many parallel CDC algorithms do not provide this guarantee. The chunks generated from a parallel chunking are usually different from those from a sequential chunking of the same file (sequential CDC) and they are also influenced by the segment size. The fundamental reason is that the boundary of the next chunk in a file is determined not only by contents in the chunk but also by the boundary of its previous chunk. Besides, to detect a new boundary we need to skip a certain number of bytes from the last boundary to maintain a minimum chunk size before starting the rolling-window-based hashing. Due to existence of this inherent dependency, chunk boundaries produced by independently performing CDC within individual segments are different from those produced by sequential CDC on the entire file. Since the segmentation enabling the parallelism is not based on the file content, the parallel CDC usually has a deduplication ratio lower than the sequential CDC. Figure 1 shows the deduplication ratios of sequential CDC and parallel CDC using 1MB segments and chunk size configuration from Dell EMC Data Domain (4KB, 8KB, and 12KB as the minimum, expected average, and maximum chunk sizes, respectively). The deduplication ratios from the parallel CDC are reduced by 6%∼25% compared to those of the sequential CDC.

The other limitation of many parallel CDC algorithms, in particular the multithreading chunking algorithms, is that they can only be accelerated with multiple cores, and cannot take advantage of instruction-level parallelism offered by the SIMD platforms, such as AVX or GPUs. The reason is that SIMD requires simultaneous execution of the same operation on different data. Any programs with frequent branches cannot be efficiently executed on such platforms. However, the chunking process does have frequent branches. At the same offset for different segments, some may detect valid chunk boundaries while others may not. As a result, applying existing CDC algorithms on the SIMD platforms cannot deliver the desired performance one may expect.

In the meantime, it becomes more and more important to leverage the SIMD platforms for compute-intensive tasks, such as chunking, for three reasons. One is that the cost per CPU core increases superlinearly as we move to processors with more cores. Using processors with a reasonable number of cores is a necessity for keeping the hardware cost within the budget. The second reason is that SIMD platforms provide better performance and power efficiency, as they can process multiple data elements in a single instruction. The third reason is these SIMD platforms are already or will soon be available in enterprise storage systems. On a backup system, compute-intensive chunking job is certainly a good candidate to utilize them. By offloading chunking to SIMD platforms, we can free up the CPU resources for other tasks, such as compression. These motivate us to re-examine parallel CDC to make it compatible with the SIMD platform.

[2]The hash function used here is different from the hash function used to generate the fingerprint to uniquely identify a chunk. To support efficient rolling hashing, we assume a hash function that can be incrementally calculated, such as CRC. For fingerprints, a cryptographic hash function, such as SHA1, is adopted to minimize hash collisions.

Our Contribution. In this work, we identify the root cause of deduplication ratio degradation of existing parallel CDC methods and provide quantitative analysis on it using real-world datasets. We propose SS-CDC, a two-stage parallel chunking algorithm, that can be parallelized by SIMD platforms and meanwhile provide chunking invariability. We implemented SS-CDC with Intel AVX instructions as a case study. To the best of our knowledge, this work is the first to use Intel AVX instructions for parallel chunking. Our experiments with real-world datasets show that compared to existing multithreading CDC method, SS-CDC can improve the deduplication ratio by about 47%, and achieve superlinear speedups (higher than the number of cores) [26] of up to additional 3.3× by exploiting parallelism from AVX.

2 BACKGROUND AND RELATED WORK

In this section we provide additional background on chunking techniques of data deduplication, especially the time-consuming content-defined chunking and efforts on its parallelization.

2.1 Fixed vs. Variable-size Chunking

A file can be partitioned into either fixed-size or variable-size chunks for deduplication. With fixed-size chunking, chunk boundaries are determined at offsets of multiple of a unit size. It is usually used in primary storage systems where the performance is critical, such as NetApp All Flash FAS [19] or Pure Storage [20], due to its high chunking speed. However, fixed-size chunking cannot address the issue of boundary shifting due to data insertions or deletions in a file. To this end, variable-size chunking, whose chunk boundaries are defined by file content, is proposed so that duplicate chunks can be identified even after file data shifting. In the so-called content-defined chunking (CDC) algorithm, a fixed-size rolling window is used to scan a file in a byte-by-byte manner to determine chunk boundaries. For the rolling window at any byte offset, a hash value is computed and compared to a predefined value. If they match, a chunk boundary is declared at the end of the window. Otherwise, the rolling window moves forward by one byte. And the process is repeated. To avoid generating too small or too large chunks, the *minimum-chunk-size* and *maximum-chunk-size* thresholds are defined. When a chunk boundary is declared, the rolling window skips the following minimum-chunk-size bytes. Meanwhile, a chunk boundary is immediately declared once the chunk size reaches the maximum-chunk-size.

It is noted that in the CDC chunking the process of determining a sequence of boundaries in a file is inherently sequential, as declaration of a new boundary is not only dependant on the hash value of the current rolling window, but also on the previous boundary's position. This places a challenge on its effective parallelization. Meanwhile, it is important to accelerate the chunking process as it is highly compute-intensive and can become the performance bottleneck of the system. There are two categories of efforts for accelerating chunking process, which are optimization of the rolling hashing and parallel chunking.

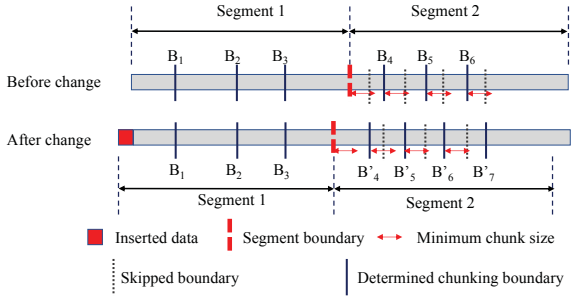


Figure 2: An example showing how deduplication opportunities are lost in existing segment-based CDC methods.

2.2 Optimizing Rolling Hashing

In CDC, a hash value is computed over the content of a rolling window at almost every byte offset of a file. As a result, the computation cost of the rolling hash function has a significant impact on the chunking speed. Lightweight hash functions have been proposed to reduce the cost. Gear [29] uses a more lightweight hash function requiring only one bit-shift, one add, and one table lookup, while Rabin fingerprint, such as CRC used in this paper, requires two bit-shift, two XOR operations, and two table lookups. FastCDC [30] proposed a few techniques to accelerate the Gear-based chunking process. AE [32] is a non-rolling-hash-based chunking algorithm that employs an asymmetric rolling window to identify extremums of data stream as boundaries. Yu et al. [31] use two functions, one lightweight and the other heavyweight, to select a chunk boundary. A simpler condition is tested first. Only when the condition is satisfied are additional computation steps performed. These techniques are orthogonal to SS-CDC, and many of them can be parallelized and accelerated using the SS-CDC technique.

2.3 Parallel Chunking and its Limitations

Another approach to speed up CDC is to parallelize and run the algorithm on parallel hardware. Many backup systems [7, 18, 28] have taken the approach to partition the input files into segments and use a thread to chunk a segment independently. With this approach, we can leverage multi-core processors to achieve parallel chunking. However, it does not guarantee chunking invariability and compromises the deduplication ratio. And it cannot fully exploit the parallelism on an SIMD hardware.

Regarding the impact on deduplication ratio, Figure 2 illustrates how a data insertion changes the segment boundaries and thus chunk boundaries, leading to the failure in detecting identical data in the second segment using the segment-based parallel chunking approach. Hash values for rolling windows at offsets B_1, B_2, ..., B_6, and B'_4, B'_5, B'_6, and B'_7 all match the predefined value and thus these offsets may potentially be chunk boundaries. Before the insertion, B'_4, B'_5, B'_6, and B'_7 are too close to their respective previous chunks (within the minimum chunk size) and thus were not selected as chunk boundaries. Instead, B_4, B_5, and B_6 were selected. After the insertion of a few bytes at the beginning of the file, as shown in Figure 2, the second segment shifts to the left by the same number of bytes. Now B'_4, B'_5, B'_6, and B'_7 become valid chunk boundaries and accordingly invalidate original chunk boundaries at offsets B_4, B_5, and B_6. For the second segment, the new chunks are unlikely to be identical to the old ones and cannot be deduplicated. In general, data insertions or deletions will shift segment boundaries, which can change chunk boundaries and reduce deduplication ratio. The root cause of the reduction is that segment boundaries are not defined by data content, though chunks within each segment are. In contrast, in a sequential CDC algorithm every chunk is content-defined without impact of boundaries defined by the segmentation.

There have been a few parallel chunking approaches. The chunking operation in P-Dedup uses a segment-based and multithreading approach [28]. It made efforts to achieve chunking invariability. After chunks within each segment are determined, a master thread computes additional rolling hashes for the data between any two adjacent segments and declares a new chunk boundary whenever it finds a matching hash value. However, this approach only produces additional small chunks and cannot ensure chunking invariability.

The only work we are aware of that guarantees chunking invariability and also provides multithreading chunking is MUCH [27]. In MUCH, data is partitioned into segments and each segment is assigned to one thread for parallel chunking. To ensure chunking invariability, it introduces a chunk marshalling stage to additionally process chunks obtained within each segment, which includes coalescing chunks that are smaller than the minimum chunk size and splitting chunks that are larger than the maximum chunk size. Nevertheless, MUCH was designed as a multithreading chunking algorithm without consideration of running on an SIMD hardware. As a result, it cannot be applied on AVX or GPUs.

Shredder [2] is a parallel chunking scheme that is designed for running on GPUs. A major issue addressed in its design is to reduce the overhead of data transfer from the main memory to the GPU device's local memory and to minimize its performance impact on the chunking. However, because the window does not roll over segment boundaries, chunk boundaries in corresponding regions can be missed. Thus, Shredder does not guarantee chunking invariability.

Table 1: Comparison of existing parallel chunking algorithms with SS-CDC

	Chunking invariability	Multi-core	GPU	AVX
P-Dedup	No	Yes	No	No
MUCH	Yes	Yes	No	No
Shredder	No	No	Yes	Maybe
SS-CDC	Yes	Yes	Yes	Yes

Table 1 summarizes existing parallel chunking approaches and compares them with our approach, SS-CDC. SS-CDC is the only approach that guarantees chunking invariability while at the same time enables parallel chunking on multi-core processors, AVX, and GPUs.

3 THE DESIGN

SS-CDC is a new CDC approach which enables high parallelism for CDC chunking to utilize parallel hardware's computing power without compromising deduplication ratio. The key insight of the work is that the chunking process can be separated into two tasks. One task is for rolling window computation to generate all potential chunk boundaries, which is expensive but can be performed in parallel in different segments. The second task is to select chunk boundaries out of the candidate ones so that they meet the minimum and maximum chunk size requirements, whose execution has to be serialized across the segments but is lightweight. Accordingly, SS-CDC separates the chunking process into two stages, one for each task. As both the rolling window computation in the first stage and the searching for final chunk boundaries in the second stage are conducted in parallel, the CDC chunking is almost fully parallelized at any reasonably small granularity. Meanwhile, as the determination of chunk boundaries is performed sequentially, SS-CDC produces identical set of chunk boundaries and the same deduplication ratio as the sequential CDC.

3.1 Decoupling Rolling Hashing from Chunk Boundary Determination

For rolling window based CDC, a chunk boundary is declared at the end of the current window only when two conditions are met. First, the hash value of the contents within the rolling window matches a predefined value. Second, the size of the chunk is within the range of the minimum and maximum chunk sizes. Instead of simultaneously checking both conditions at an offset, SS-CDC separates them into two stages. Specifically, in the first stage a hash value for each rolling window is calculated and compared with the predefined value. A chunk boundary *candidate* is declared

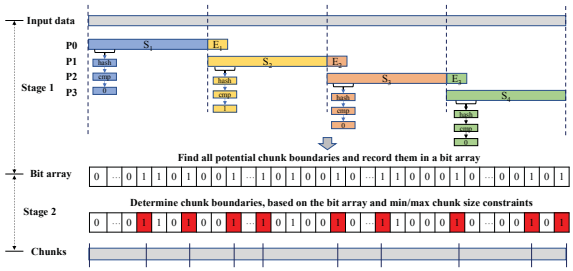


Figure 3: The two stages of SS-CDC algorithm. Each thread continues the rolling hashing for extra (rolling_window_size - 1) bytes from the next segment, donated as E_1, E_2 and E_3 in the figure. P0, P1, P2, and P3 are processes (or threads) for chunking.

at the end of the rolling window if the hash value matches. At the end of this stage, it produces a set of chunk boundary candidates which satisfy the first condition. During the second stage, final chunk boundaries are selected from the candidates, which meet the minimum and maximum chunk size constraints (the second condition).

Figure 3 illustrates operations involved in the two stages. The input data is first partitioned into equal-size segments (S_1, S_2, S_3, and S_4), each of which is assigned to a thread for identifying chunk boundary candidates. To determine chunk candidates at every offset, a thread working on a segment will include the extra ($rolling\_window\_size - 1$) bytes from the next segment for rolling window computation, which are illustrated as E_1, E_2, and E_3 in Figure 3. The hash value of each rolling window is calculated and compared to a predefined value and the result is recorded in a bit array, where each bit indicates whether there is a hash value match for the corresponding rolling window. Multiple bits in the bit array can be set simultaneously using SIMD instructions without using locks. For an input data with N bytes, the output bit array will be of N bits [3]. A bit '1' at the bit-offset k in the bit array indicates a chunk boundary candidate at the byte-offset k in the input file. After all candidates are identified, SS-CDC enters its second stage where the bit array produced by the first stage is scanned from its beginning, searching for the '1' bits that meet the minimum and maximum chunk size constraints. These offsets are the final chunk boundaries.

3.2 Parallelizing Operations in SS-CDC

To achieve parallel chunking performance, SS-CDC parallelizes both of its stages to take full advantage of the parallel hardware. It is straightforward to parallelize computation in the first stage. We assign each segment to a different thread, and the rolling window hashing and comparison are performed independently in each thread in parallel. However, the second stage must be performed by sequentially checking the bit array in a bit-by-bit manner to find the next chunk boundary which meets the minimum and maximum chunk size constraints. With the first stage being optimized, the runtime from the second stage becomes significant.

To address this issue, we observed that the bit array contains mostly '0' bits, with only a few '1' bits. For example, with an expected average chunk size of 4KB, there will be one '1' bit in every 4000 bits on average. The bit array can be represented as an array consisting of values of a longer data type, such as 32-bit integers. By comparing whether the next 32-bit integer is 0, we effectively check the next 32 bits in the bit array. When a non-zero value is found, a bit-by-bit checking is needed. Furthermore, SIMD instructions [21] (and multiple threads) can be used to check multiple values in parallel and we can skip the '0' bits and locate values with '1's in the array quickly.

3.3 SS-CDC on AVX Instructions

SS-CDC can be easily deployed on a wide range of parallel platforms, including multi/many-core systems, GPGPU platforms and others supporting SIMD instructions. As a case study, we implemented SS-CDC with Intel Advanced Vector Extensions 512 (AVX-512) instructions [6, 21], which are extensions to the x86 ISA for Intel and AMD processors, and provide vector operations in an SIMD manner for some instructions. They are generally available in today's mainstream processors [4, 5] and provide the benefit of parallel execution without requiring extra hardware support. We leave it as future work to port SS-CDC to other parallel hardware. For processors with AVX-512 instructions support, the extended registers are 512-bit long.

In the prototype, we use CRC-32 (with the polynomial 0xedb88320) as the rolling hash function for detecting chunk boundaries. The hash value of a rolling window with size w and ending at *offset* is calculated as shown in Equations 1 and 2. Like many other efficient CRC implementations [10], we pre-compute two static tables, *crcu* and *crct*, and use them to remove and add contribution of a byte in the rolling hash computation. In Equation 1, the contribution of the leftmost byte leaving the window is removed, while in Equation 2 the contribution of the byte entering the window from the right is added to the value. We use a rolling window of 256 bytes.

$$hash\_tmp = hash\_old \oplus crcu[buf[offset - w]] \quad (1)$$

$$hash\_new = (hash\_tmp >> 8) \oplus crct[buf[offset]] \quad (2)$$

In the first stage of SS-CDC, the input file, which are partitioned into 16 equal-size segments, are processed in parallel to identify all potential chunk boundaries. We use an AVX

[3]The additional memory to store the bit array can be freed or reused as soon as the chunking is completed.

register (named R_c) to store 16 CRC values for the current 16 rolling windows, one for each segment. The execution of an AVX-512 instruction can be viewed as 16 parallel threads performing the same operation. For the load operation, AVX-512 instructions support loading 16 32-bit values from 16 different locations in the memory to an AVX register. So the bytes entering and leaving a rolling window from a segment are loaded with 32 bits at once into registers. To remove the contribution of the byte leaving the window, for each segment the leftmost byte in the corresponding window is used as the index to retrieve a value from the *crcu* table. The 16 values from the table, one for each segment, are stored in an AVX register (named R_u). Then, the result of (R_c *xor* R_u) is stored in a register as *hash_tmp*. Similarly, we add the contribution of the byte entering the window for each segment by using the byte as the index to retrieve a value from the *crct* table, and *xor* the 16 values with *hash_tmp* right-shift by 8 bits, to obtain *hash_new*.

One challenge in the implementation is about the two table lookups in each iteration of the computation, which can be very time-consuming if performed sequentially because it would require 16 separated memory accesses for each lookup. One such example is to use the *_mm512_set_epi32* instruction to directly set an AVX register with 16 values from one of the tables. For higher efficiency, we instead accelerate the table lookup by performing parallel fetching with the *_mm512_i32gather_epi32* instruction, and reduce the time spent on the first stage by over 50%. The second stage checks 512 bits together using one AVX instruction by taking advantage of widespread '0' bits in the bit array. In most cases, we will have 512 consecutive '0' bits.

To detect a boundary, after skipping *minimum_chunk_size* bits, we load the following 512 bits (as 16 32-bit integers) from the bit array to a register, and use the *_mm512_cmpneq_epi32_mask* instruction to compare it with 16 '0's and generate a 16-bit mask indicating whether there are non-zero integers. SS-CDC will continue with the next 512 bits in the bit array unless non-zero integer(s) are found or the maximum chunk size is reached, which will declare a chunk boundary. Compared to scanning the bit array one bit at a time, using the AVX instructions accelerates the second stage by 30-40×, making its running time account for only ~2% of the total chunking time.

3.4 SS-CDC on Multiple Cores with AVX

By leveraging AVX instructions, our implementation of SS-CDC is able to do parallel chunking among segments from a single file on individual cores. However, parallel chunking from a single core may still not provide sufficient chunking bandwidth for a single file. In that case, we need to further explore multiple cores to further improve the chunking bandwidth. In addition, most backup systems need to support multiple concurrent backup jobs. Existing backup systems have exploited job-level parallelism and CPU core-level parallelism. With AVX, we can additionally exploit instruction-level parallelism. In this section, we will discuss how we scale SS-CDC chunking to multiple cores for single files and then how to handle multiple backup jobs (or backup files).

To scale chunking for a single file to multiple cores, the backup file is first partitioned into fixed-size segments, which are placed into a segment queue. The system allocates the number of cores for chunking for this backup job. A chunking thread is started at each core. It retrieves a batch of (N) segments each time from the head of the queue for the first-stage chunking. As a lock is required to enforce an exclusive access to the queue, a larger N is preferred to reduce the locking cost. Another benefit for a thread to retrieve and process multiple (contiguous) segments at a time is it only needs to calculate the first hash from scratch for each batch and the rest can be calculated incrementally, which is cheaper. A barrier synchronization is used at the end of the first-stage processing to synchronize the threads. After this, one of the threads proceeds to the second stage to select the final chunk boundaries according to the bit array produced in the first stage. To avoid high synchronization time among the first-stage threads due to an unbalanced load, a smaller segment batch (N) is preferred. To strike a balance between these two requirements on the batch size, for a segment size of about 0.5% of a file size using an N value of 2-8 generally leads to good performance. And our experiments find that the performance is not sensitive to the value in the range. Therefore, SS-CDC uses 4 as N's default value. Since the second stage is very lightweight, the performance is acceptable to do it sequentially. Besides, the other cores that have completed their first-stage chunking can be used to process other files.

To handle multiple concurrent backup jobs, the system can use a policy to determine how many cores to allocate for each job for chunking. The system could allocate a single core for chunking for each job if that can provide sufficient chunking bandwidth or more cores to it if desired. The system could allocate cores for chunking evenly among backup jobs or based on priority. For a given number of cores and the number of jobs, the combination of how to allocate cores among jobs is very large. In our evaluation, we will instead focus on demonstrating SS-CDC's scalability with parallel chunking by either using all cores to chunk one file at a time or use each core to chunk a different file.

To make the presentation more concise, we follow the Flynn's taxonomy [25], which includes SIMD, to introduce the following terms. SFMS (Single File Multiple Segments) refers to parallel chunking of a *single* file at a time on multiple cores and parallel chunking of *multiple* segments within

each AVX-supported core. Similarly, SFSS (Single File Single Segment) refers to parallel chunking of a single file on multiple cores without using AVX for parallel chunking within a core. MFMS (Multiple Files Multiple Segments) is similar to SFMS except that multiple files, each at a different core, are processed at the same time. MFSS (Multiple Files Single Segment) is similar to SFSS except that multiple files, each at a different core, are processed concurrently without using AVX within each core. To show the additional parallelism from using AVX, we will compare the speedups between MFMS and MFSS and between SFMS and SFSS in our evaluation.

4 EVALUATION

We answer the following questions in the evaluation. First, how much speedup can SS-CDC provide by leveraging AVX at a single core? To answer this question, we compare the chunking time of SS-CDC with sequential CDC. Second, how does SS-CDC with AVX scale to multiple cores? We use the sequential CDC as the baseline and compare the speedups from SS-CDC with existing multithreading CDC, for both single-file chunking and multi-file chunking. Finally, we evaluate the deduplication ratio reduction (degradation) from existing segment-based multithreading CDC.

The experiments were conducted on a Dell-EMC PowerEdge T440 server with 2 Intel Xeon 3.6GHz CPUs, each with 4 cores and 16MB LLC. The server is equipped with 256GB DDR4 memory and installed with Ubuntu 18.04 OS. The processors support Intel AVX-512 instructions. The datasets are stored on the local disks. In the measurements, the chunking time only includes the time spent on determining chunking boundaries, and excludes the time for loading the data to memory before the chunking is performed. We use 7 real-world datasets as shown in Table 2. For the Linux source code, we downloaded all 1013 versions (from 3 to 4.9) from the Linux Kernel Archives [23]. Each version is converted into a file of the *mtar* format [15] for backup. The others are six groups of Docker images, downloaded from Docker Hub [9], where each image is a tar file. In the figures showing experiment results, measurements about the datasets are presented in the order of their deduplication ratios, from low to high. Unless noted, we use 1MB as the segment size for dispatching data to threads, and 2KB, 16KB, and 64KB, as the minimum, expected average, and maximum chunk size respectively, as those in LBFS [16].

4.1 Chunking Speed

With the instruction-level parallelism enabled by the AVX instructions, we expect to see speedups in chunking for both one core and multiple cores for SS-CDC.

Results on One Core. We first run the chunking process on one core with different datasets and see how the use of the AVX instructions improves the chunking performance. Figure 4 shows the chunking throughput of SS-CDC and sequential CDC with one thread running on one core. The speedups are very consistently, about 3.3×, though different datasets have different deduplication ratios. The speedups are achieved by leveraging the instruction-level parallelism within a single core and the deduplication ratio of a specific dataset does not impact the speedup, as SS-CDC always scans the complete dataset.

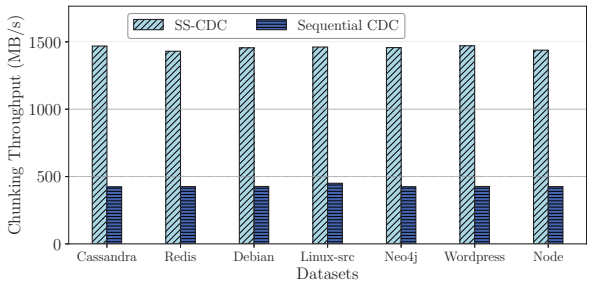


Figure 4: Chunking speed of single-threaded SS-CDC and sequential CDC with one core. The minimum, expected average and maximum chunk sizes are 2KB, 16KB, and 64KB, respectively.

Although the speedup is significant, it may be lower than one might expect in the light of parallelism provided by the AVX-512 instructions, where it processes 16 segments concurrently. There are several reasons. First, SS-CDC actually needs to read more data and do more rolling hash calculation than sequential chunking, as it does not skip the input data using the minimum chunk size. As we will show, the minimum chunk size has a considerable impact on the speedups of SS-CDC. Second, while the chunking process is CPU intensive, it also includes substantial memory accesses for loading data from the memory to registers. While SS-CDC leverages the AVX instructions and reduces number of instructions executed for chunking, it does not reduce the amount of data that needs to be loaded from the memory. Third, since we conduct chunking for 16 segments concurrently, data are accessed at 16 different memory addresses in parallel. Existing DRAM controllers and CPU caches may not be optimized to handle such workloads. Nevertheless, for all datasets we examined, we achieved more than 3× speedups over sequential CDC.

While the speedups are not impacted by the deduplication ratio of a dataset, the minimum chunk size has a direct impact on the speedup of SS-CDC. The reason is that the sequential CDC skips the minimum chunk size of bytes after each new chunk boundary is detected while SS-CDC has to scan and calculate a hash for every byte. To understand

Table 2: Real-world datasets used in the experiments. All the Docker images are downloaded from Docker Hub [9].

Name	Size (GB)	# of files	Dedup Ratio	Description
Cassandra	14.2	40	5.0	Docker images of Apache Cassandra, an open-source storage system [3].
Redis	4.1	34	7.2	Docker images of the Redis key-value store database [22].
Debian	9.5	92	15.8	Docker images of Debian Linux distribution (since Ver. 7.11) [12].
Linux-src	570	1013	16.4	Uncompressed Linux source code (v3.0~v4.9) downloaded from the website of Linux Kernel Archives [23].
Neo4j	46.0	140	19.0	Docker images of neo4j graph database [17].
Wordpress	181.7	501	22.0	Docker images of WordPress rich content management system [14]
Nodejs	800.0	1567	41.4	Docker images of JavaScript-based runtime environment packages [13]

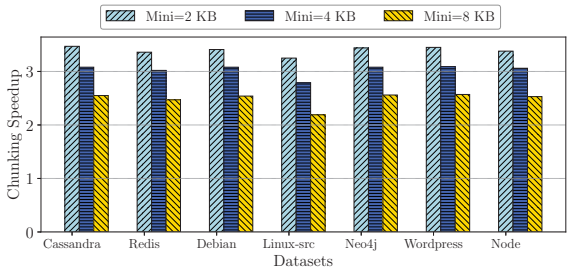


Figure 5: Chunking speedups when different minimum chunk sizes are used. The expected average and maximum chunk sizes are 16KB and 64KB, respectively.

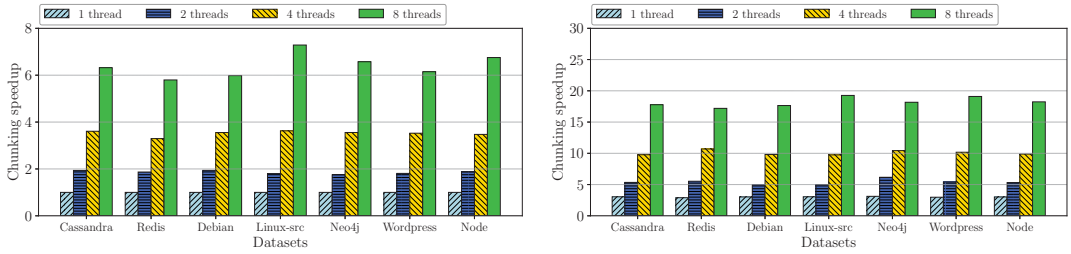
the impact of the minimum chunk size on SS-CDC's performance advantage, we measured the chunking speedups with different minimum chunk sizes. The results are presented in Figure 5. As expected, the chunking speedup is decreased when the minimum chunk size is increased. However, even with a large minimum chunk size (e.g., with a 8KB minimum chunk size and a 16KB average chunk size, 50% of the input data can be skipped in the sequential CDC baseline.), the chunking speedups are still substantial, about 2.5×. As the 2KB minimum chunk size is commonly used, we adopt it as the default value in the evaluation.

Results on Multiple cores. Next we evaluated the scalability of the chunking speed of SS-CDC on multiple cores. Specifically, we examined multithreading SFMS (scaling SS-CDC to multiple cores for single files) and MFMS (scaling SS-CDC to multiple cores for multiple files). We compared them with multithreading regular CDC methods (SFSS and MFSS) that do not use AVX. Their chunking speeds were normalized to the sequential CDC without using AVX, one file at a time, on one core.

To examine how SS-CDC scales for single file chunking, we look at how the chunking speedup increases when we use more threads for chunking a single file. After we establish SS-CDC scales for single file chunking, we examine multiple file chunking where one file is assigned to only one thread and we increase the number of concurrent files being chunked. While in a real deployment, there are many different ways to assign chunking threads among backup jobs, the experiments serve our purpose to demonstrate the scalability of SS-CDC for both single file chunking and multiple file chunking. For each dataset, we change the number of chunking threads from 1 to 8 and show their speedups over sequential chunking using one thread.

The results are shown in Figure 6 for single file chunking and Figure 7 for multiple file chunking. From Figures 6(a) and 7(a) (note the Y axes in the (a) and (b) figures use different ranges), we can see multithreading regular CDC receive a speedup approximately proportional to the number of cores (or threads). This is especially true for the MFSS case (Figure 7a) where each file is processed independently by a chunking thread. For example, for the Cassandra dataset, its MFSS speedups are 1.0, 2.0, 3.9, 7.5 on 1, 2, 4, and 8 cores, respectively. The speedups become smaller (1.0, 1.9, 3.6 and 6.3) with its SFSS implementation, where one file is partitioned into segments for the threads to process in parallel, resulting in overheads for using locks at the segment queue and waiting for all chunking threads to complete the first stage.

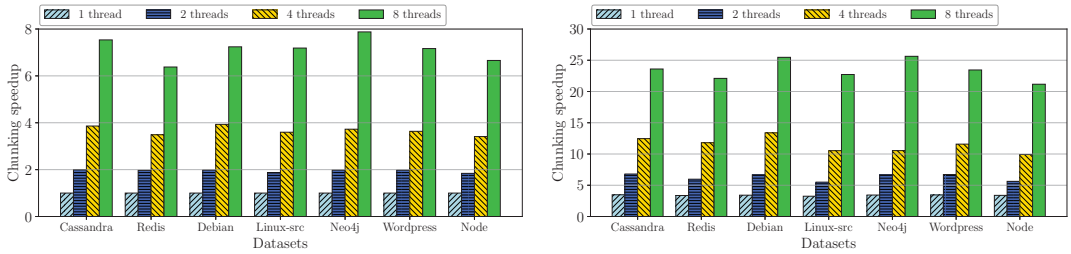
In contrast, with its use of the AVX instructions, multithreading SS-CDC achieves superlinear chunking speedups. Still take the Cassandra dataset as an example. Its MFMS speedups are 3.5, 6.8, 12.5, and 23.6 on 1, 2, 4, and 8 cores, respectively, which shows the extra speedup of using AVX instructions scale well with the number of cores (consistently ~3×). When scaling SS-CDC's performance to multiple cores for single-file chunking in SFMS, the speedups become smaller. For example, the SFMS speedups for the Cassandra dataset reduces to 3.1, 5.4, 9.8, and 17.8 on 1, 2, 4, and 8 cores, respectively. Multithreading SFMS suffers from the same bottlenecks as multithreading SFSS, such as the use of a lock at the segment queue, barrier synchronization at the end of the first stage, and serialization for the second stage. Besides, SFMS needs to do more rolling hash calculation for the extra bytes, as shown in E_1, E_2, or E_3 in Figure 3. In spite of this, SFMS still achieves superlinear speedups, though it cannot achieve the same speedups as MFMS. In the meantime, it unlocks the opportunity of exploiting intra-file chunking parallelism.



(a) SFSS Regular CDC

(b) SFMS SS-CDC

Figure 6: Chunking speedups of multithreading regular CDC and multithreading SS-CDC over sequential CDC at one core with different datasets and thread/core counts when a file is processed by all threads.



(a) MFSS Regular CDC

(b) MFMS SS-CDC

Figure 7: Chunking speedups of multithreading regular CDC and multithreading SS-CDC over sequential CDC at one core with different datasets and thread/core counts when each file is processed by one thread.

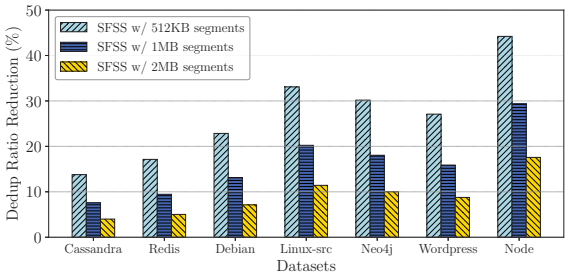
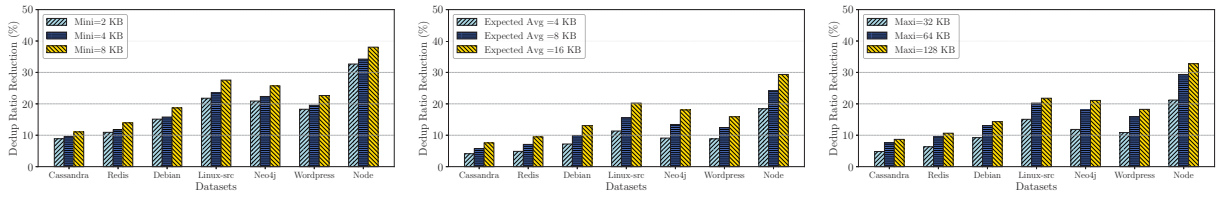


Figure 8: Deduplication ratio reduction of the regular multithreading CDC chunking approach (SFSS), compared with that of SS-CDC's SFMS implementation (on 8 cores) when different segment sizes are used.

4.2 Deduplication Ratio

SS-CDC is designed to provide chunking invariability. When used in either MFMS or SFMS, it can always achieve the same deduplication ratio as that of sequential CDC. In fact, during our development of SS-CDC, we compared the chunk boundaries from SS-CDC with sequential CDC, to verify our SS-CDC implementation is correct. However, existing segment-based single file parallel chunking (SFSS) cannot achieve this chunking invariability, and experience deduplication ratio reduction. To gauge significance of the reduction, we compare the deduplication ratios of SS-CDC's SFMS implementation with SFSS and conduct a quantitative study where we vary the segment size and the chunk size.

Figure 8 presents the deduplication ratio reduction from SFSS with different segment sizes. The baseline is the deduplication ratios from SS-CDC (and also sequential CDC). As shown, SFSS can suffer significant deduplication ratio reductions when using different segment sizes. For example, the reduction is about 45% when the segment size is 512KB for the *Node* dataset. The reduction decreases when increasing the segment size. However, for some datasets even when the segment size is large, the reduction can still be substantial. For example, for the *Node* dataset, the reduction is about 18% when the segment size is 2MB. In many scenarios, including execution at GPGPU's cores, it is necessary to avoid using very large segments to exploit sufficient parallelism or/and to accommodate the segments in the limited device local memory. Existing segment-based parallel chunking, as in SFSS, has the fundamental limitation that requires a user to make a tradeoff between fine-grain parallelism by using a small segment size and a high deduplication ratio with a



(a) Varying minimum chunk sizes **(b) Varying average chunk sizes** **(c) Varying maximum chunk sizes**

Figure 9: Reduction of deduplication ratio (in percentage) for multithreading CDC (SFSS) on 8 cores, compared to SS-CDC (multithreading SFMS) with different chunk size configurations. The segment size is 1MB.

large segment size. With SS-CDC, a user can use any segment size without deduplication ratio reduction.

Next, we turn to look at how the chunk size impacts deduplication ratio for SFSS. We vary all three parameters controlling the chunk size, including the minimum, the expected average, and the maximum chunk sizes, one at a time. The deduplication ratios are compared with SS-CDC.

Across all three figures in Figure 9, in general when a dataset has a higher deduplication ratio, the reduction of deduplication ratio from SFSS is more significant. When there are more duplicates in the dataset, SFSS is more likely to turn a duplicate chunk into a unique one due to chunk boundary shifts because of file segmentation. The deduplication ratio reduction is most significant when we vary the minimum chunk size, ranging from 10% to 38% as shown in Figure 9a. Furthermore, when we increase the minimum chunk size, the reduction becomes larger. With a larger minimum chunk size, it increases the possibility of finding a matching rolling hash value in that window and having different chunks between sequential CDC and SFSS. This can leave the use of SFSS in a dilemma where a larger minimum chunk size can skip more bytes for better chunking performance while a smaller minimum chunk size can avoid substantial deduplication ratio reduction.

The impacts of the average chunk size on deduplication ratio for SFSS are more complicated. On one hand, with a larger average chunk size, there are fewer chunks and thus fewer chunk boundaries. We have a smaller probability to find candidate chunk boundaries in the minimum chunk size window that could lead to different chunks in segment-based SFSS. On the other hand, with a larger average chunk size, it takes more bytes for SFSS to synchronize back to chunk boundaries as those in sequential CDC as there will be fewer candidate chunk boundaries. To investigate which factor has a larger impact on the deduplication ratio in SFSS, we conduct experiments by varying the expected average chunk size. As shown in Figure 9b, with a larger average chunk size the deduplication ratio reduction is more significant, which indicates the second factor has a bigger impact on deduplication ratio reduction.

In addition to the minimum and the average chunk sizes, the maximum chunk size also affects deduplication ratio. Figure 9c shows the deduplication ratio reduction when varying the maximum chunk size. With a larger maximum chunk size, the deduplication reduction is more significant. By using a larger maximum chunk size, the size of the chunks are more likely larger as more bytes can be scanned when deciding the next chunk boundary. So once a unique chunk is generated due to the segmentation, it can potentially make a larger range of bytes not deduplicated.

To summarize, existing parallel chunking using segments suffers significant deduplication ratio reduction when they exploit segment-based parallelism. SS-CDC guarantees chunking invariability and achieves parallel chunking performance without impacting deduplication ratios.

5 CONCLUSIONS

In this paper, we presented SS-CDC, a new parallel CDC that takes full advantage of the parallel computing power of the underlying hardware for high chunking speed without compromising deduplication ratio. SS-CDC separates the chunking process into a stage that is compute intensive but easy to parallelize and a second stage that is sequential but with low runtime cost. It can achieve almost full parallel chunking performance and the same deduplication ratio as sequential CDC. With a prototype based on AVX-512, we demonstrated SS-CDC can be implemented on an SIMD platform and evaluated that we can achieve parallel chunking performance for both single file chunking and multi-file chunking. With SS-CDC, we can now offload compute-intensive CDC to SIMD platforms to exploit extra instruction-level parallelism to accelerate chunking and get high deduplication ratios.

ACKNOWLEDGMENTS

We are grateful to the anonymous reviewers, for their valuable comments that improved the paper. This work was supported in part by NSF grants CNS-1527076 and CCF-1815303.

REFERENCES

[1] Samer Al-Kiswany, Abdullah Gharaibeh, Elizeu Santos-Neto, George Yuan, and Matei Ripeanu. 2008. StoreGPU: Exploiting Graphics Processing Units to Accelerate Distributed Storage Systems. In *Proceedings of the 17th International Symposium on High Performance Distributed Computing (HPDC '08)*. ACM, New York, NY, USA, 165–174. https://doi.org/10.1145/1383422.1383443

[2] Pramod Bhatotia, Rodrigo Rodrigues, and Akshat Verma. 2012. Shredder: GPU-accelerated Incremental Storage and Computation. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST'12)*. USENIX Association, Berkeley, CA, USA, 14–14. http://dl.acm.org/citation.cfm?id=2208461.2208475

[3] Apache Cassandra. 2014. Apache cassandra. http://planetcassandra.org/what-is-apache-cassandra.

[4] Intel Corporation. 2013. Intel Xeon Phi processors. https://www.intel.com/content/www/us/en/products/processors/xeon-phi/xeon-phi-processors.html.

[5] Intel Corporation. 2015. Intel Skylake Processors. https://ark.intel.com/products/codename/37572/Skylake.

[6] Intel Corporation. 2018. Intel Architecture Instruction Set Extensions Programming Reference. https://software.intel.com/sites/default/files/managed/c5/15/architecture-instruction-set-extensions-programming-reference.pdf.

[7] Dell EMC. 2019. Data Domain - Data Backup Appliance, Data Protection. https://www.dellemc.com/en-us/data-protection/data-domain-backup-storage.htm

[8] DELL EMC inc. 2018. Supported Stream Counts for Data Domain OS 5.7. https://community.emc.com/docs/DOC-63282.

[9] Docker, Inc. 2016. Official repositories on Docker Hub. https://hub.docker.com/.

[10] Gary S. Brown. 1986. CRC32 code in FreeBSD derived from work by Gary S. Brown. http://web.mit.edu/freebsd/head/sys/libkern/crc32.c.

[11] Fanglu Guo and Petros Efstathopoulos. 2011. Building a High-performance Deduplication System. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference (USENIXATC'11)*. USENIX Association, Berkeley, CA, USA, 25–25. http://dl.acm.org/citation.cfm?id=2002181.2002206

[12] Docker Inc. 2018. debian: Docker Official Images. https://hub.docker.com/_/debian/.

[13] Docker Inc. 2018. Node: Docker Official Images. https://hub.docker.com/_/node/.

[14] Docker Inc. 2018. wordpress: Docker Official Images. https://hub.docker.com/_/wordpress/.

[15] Xing Lin, Fred Douglis, Jim Li, Xudong Li, Robert Ricci, Stephen Smaldone, and Grant Wallace. 2015. Metadata Considered Harmful ... To Deduplication. In *Proceedings of the 7th USENIX Conference on Hot Topics in Storage and File Systems (HotStorage'15)*. USENIX Association, Berkeley, CA, USA, 11–11. http://dl.acm.org/citation.cfm?id=2813749.2813760

[16] Athicha Muthitacharoen, Benjie Chen, and David Mazières. 2001. A Low-bandwidth Network File System. *SIGOPS Oper. Syst. Rev.* 35, 5 (Oct. 2001), 174–187. https://doi.org/10.1145/502059.502052

[17] Neo Technology. 2018. Neo4j Graph Database Platform. https://neo4j.com/.

[18] NETAPP inc. 2015. NetApp® AltaVault® Cloud Integrated Storage 4.0: Installation and Service Guide for Physical Appliances. goo.gl/wj2Y4K.

[19] NETAPP inc. 2019. AFF A-Series All Flash Arrays: Leads the market with new performance benchmark results. https://www.netapp.com/us/products/storage-systems/all-flash-array/aff-a-series.aspx.

[20] Pure Storage, Inc. 2019. Pure Unifies Cloud: Your Hybrid Cloud Journey Just Got A Lot Easier. https://www.purestorage.com/.

[21] James Reinders. 2013. Intel AVX-512 instructions. https://software.intel.com/en-us/blogs/2013/avx-512-instructions

[22] Salvatore Sanfilippo and Pieter Noordhuis. 2015. Redis. http://redis.io.

[23] The Linux Kernel Organization, Inc. 2019. The Linux Kernel Archives. https://www.kernel.org

[24] Grant Wallace, Fred Douglis, Hangwei Qian, Philip Shilane, Stephen Smaldone, Mark Chamness, and Windsor Hsu. 2012. Characteristics of Backup Workloads in Production Systems. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST'12)*. USENIX Association, Berkeley, CA, USA, 4–4. http://dl.acm.org/citation.cfm?id=2208461.2208465

[25] Wikipedia. 2019. Flynn's taxonomy. https://en.wikipedia.org/wiki/Flynn%27s_taxonomy

[26] Wikipedia. 2019. Speedup. https://en.wikipedia.org/wiki/Speedup#Super-linear_speedup.

[27] Y. Won, K. Lim, and J. Min. 2015. MUCH: Multithreaded Content-Based File Chunking. *IEEE Trans. Comput.* 64, 5 (May 2015), 1375–1388. https://doi.org/10.1109/TC.2014.2322600

[28] Wen Xia, Hong Jiang, Dan Feng, Lei Tian, Min Fu, and Zhongtao Wang. 2012. P-dedupe: Exploiting parallelism in data deduplication system. In *2012 IEEE Seventh International Conference on Networking, Architecture, and Storage*. IEEE, IEEE, Xiamen, China, 338–347.

[29] Wen Xia, Hong Jiang, Dan Feng, Lei Tian, Min Fu, and Yukun Zhou. 2014. Ddelta: A deduplication-inspired fast delta compression approach. *Performance Evaluation* 79 (2014), 258–272.

[30] Wen Xia, Yukun Zhou, Hong Jiang, Dan Feng, Yu Hua, Yuchong Hu, Yucheng Zhang, and Qing Liu. 2016. FastCDC: A Fast and Efficient Content-defined Chunking Approach for Data Deduplication. In *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '16)*. USENIX Association, Berkeley, CA, USA, 101–114. http://dl.acm.org/citation.cfm?id=3026959.3026969

[31] C. Yu, C. Zhang, Y. Mao, and F. Li. 2015. Leap-based Content Defined Chunking âĂŤ Theory and Implementation. In *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE, Santa Clara, CA, 1–12. https://doi.org/10.1109/MSST.2015.7208290

[32] Yucheng Zhang, Dan Feng, Hong Jiang, Wen Xia, Min Fu, Fangting Huang, and Yukun Zhou. 2017. A Fast Asymmetric Extremum Content Defined Chunking Algorithm for Data Deduplication in Backup Storage Systems. *IEEE Trans. Comput.* 66, 2 (Feb 2017), 199–211. https://doi.org/10.1109/TC.2016.2595565

[33] Benjamin Zhu, Kai Li, and Hugo Patterson. 2008. Avoiding the Disk Bottleneck in the Data Domain Deduplication File System. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST'08)*. USENIX Association, Berkeley, CA, USA, Article 18, 14 pages. http://dl.acm.org/citation.cfm?id=1364813.1364831