DESIGNING HIGHLY-EFFICIENT DEDUPLICATION SYSTEMS WITH OPTIMIZED COMPUTATION AND I/O OPERATIONS

by
FAN NI

Presented to the Faculty of the Graduate School of
The University of Texas at Arlington in Partial Fulfillment
of the Requirements
for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT ARLINGTON
May 2019

Copyright © by Fan Ni 2019
All Rights Reserved

ACKNOWLEDGEMENTS

I would like to express my thanks to my thesis advisor, Dr. Song Jiang, for his guidance in the completion of this thesis. During my Ph.D. study, Dr. Jiang has always be supportive to my research and given me a lot of help for my study and life. My thanks also extend to Drs. Hong Jiang, Jia Rao and Jiang Ming, my thesis committee members. I also want to thank Xing Lin who helped me a lot during my internship at NetApp, and also NetApp which will be my next stop for my career. Last on paper but first on my thoughts, I would like to give my most heartfelt thanks to my wife Qin Sun, who has always be so supportive and devoted her energy and time to take care of the whole family. I also want to thank my son Alex, who has given us so much fun and happiness.

April 17, 2019

ABSTRACT

DESIGNING HIGHLY-EFFICIENT DEDUPLICATION SYSTEMS WITH OPTIMIZED COMPUTATION AND I/O OPERATIONS

Fan Ni, Ph.D.
The University of Texas at Arlington, 2019

Supervising Professor: Song Jiang

Data deduplication has been widely used in various storage systems for saving storage space, I/O bandwidth, and network traffic. However, existing deduplication techniques are inadequate as they introduce significant computation and I/O cost. First, to detect duplicates the input data (files) are usually partitioned into small chunks in the chunking process. It can be very time consuming if the content-defined chunking (CDC) method is adopted, where the chunk boundaries are determined by checking the data content byte-by-byte, for detecting duplicates among modified files. Second, for each chunk generated in the chunking process, we need to apply a collision resistant hash function on it to generate a hash value (fingerprint). Chunks with the same fingerprint are deemed as having the same contents and only one copy of the data is stored on the disk. The fingerprinting process of calculating the collision-resistant hash value for each chunk is compute-intensive. Both the chunking and fingerprinting processes in existing deduplication systems introduce heavy computation burdens to the system and degrade the overall performance of the system. Third, in addition to the extra cost introduced by the chunking and fingerprinting processes, a deduplication system introduces extra I/O overheads for persisting and retrieving its metadata, which can significantly offset its advantage of saving I/O bandwidth.

To this end, a deduplication system demands efficient computation and I/O operations. In this dissertation, we made several efforts to reduce the computation and I/O overheads in deduplication systems. First, two efforts have been made to accelerate the chunking process in the CDC-based deduplication. We designed a new parallel CDC algorithm that can be deployed on the SIMD platform to fully exploit its instruction-level parallelism without compromising the deduplication ratio. Further, we designed a highly efficient CDC chunking method that removes the speed barrier imposed by the existing byte-by-byte chunk boundary detection technique through exploiting the duplication history. Second, we identified an opportunity to use fast non-collision-resistant hash func-

tions for efficient deduplication of journaled data in a journaling file system to achieve much higher file access performance without compromise of data correctness and reliability. Third, to avoid the performance degradation caused by the frequent writes of small metadata in primary deduplication systems, we proposed to opportunistically compress the fixed-size data block to make room for embedding the metadata. With the proposed method, in most cases the explicit metadata writes on the critical path can be avoided to significantly improve the I/O efficiency.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS iii
ABSTRACT iv
Chapter Page
1. INTRODUCTION 1
2. SS-CDC: A Two-stage Parallel Content-Defined Chunking for Deduplicating Backup Storage 5
   2.1 Introduction 5
      2.1.1 Using Deduplication to Improve Space Efficiency 5
      2.1.2 Accelerating CDC-based Deduplication 6
      2.1.3 Our Contribution 8
   2.2 Background and Related Work 8
      2.2.1 Fixed vs. Variable-size Chunking 8
      2.2.2 Optimizing Rolling Hashing 9
      2.2.3 Parallel Chunking and its Limitations 10
   2.3 The Design 11
      2.3.1 Decoupling Rolling Hashing from Chunk Boundary Determination 12
      2.3.2 Parallelizing Operations in SS-CDC 13
      2.3.3 SS-CDC on AVX Instructions 13
      2.3.4 Multi-threaded SS-CDC on Cores with AVX 15
   2.4 Evaluation 17
      2.4.1 Chunking Speed 18
      2.4.2 Deduplication Ratio 21
   2.5 Conclusions 23
3. RapidCDC: The Duplicate Locality and its Use to Accelerate Chunking in CDC-based Deduplication Systems 25
   3.1 Introduction 25
   3.2 The Duplicate Locality 27
   3.3 The Design of RapidCDC 30
      3.3.1 Quickly Reaching Next chunk's Boundary 30
      3.3.2 Accepting Suggested Chunk Boundaries 32
      3.3.3 Maintaining List of Next-chunk Sizes 33
   3.4 Evaluations 34
      3.4.1 The Systems in Evaluation 35

3.4.2 The Datasets 36
3.4.3 Results with Synthetic Datasets 36
3.4.4 Impact of Modification Count and Distribution 36
3.4.5 Results with Real-world Datasets 40
3.5 Related Work 45
3.5.1 Reducing Computation Cost in Chunking 45
3.5.2 Accelerating Chunking with Parallelism 46
3.6 Conclusion 47
4. WOJ: Enabling *W*rite-*O*nce Full-data *J*ournaling in SSDs by Using Weak-Hashing-based Deduplication 49
4.1 Introduction 49
4.1.1 Data Journaling is Necessary 49
4.1.2 SSD's Endurance is now a Barrier 50
4.1.3 Regular Deduplication is too Expensive 51
4.1.4 A Lightweight Built-in Solution 52
4.2 The Design of WOJ 53
4.2.1 SSD with File-system-level Knowledge 53
4.2.2 Deduplication with Non-collision-resistant Fingerprints 54
4.2.3 Metadata Supporting Movements of Physical Blocks 56
4.3 Evaluation 57
4.3.1 Experiment Methodology 57
4.3.2 Results with Write-only Micro Benchmarks 59
4.3.3 Results with Filebench Benchmarks 64
4.3.4 Results with Database Workload 65
4.3.5 Results with Workloads Using Real-world data 66
4.3.6 Memory Space Overheads 67
4.4 Conclusion 68
5. ThinDedup: An I/O Deduplication Scheme that Minimizes Efficiency Loss due to Metadata Writes 69
5.1 Introduction 69
5.2 The design of ThinDedup 71
5.2.1 Window-based Metadata Persistence 72
5.2.2 Zone-based Data Persistence 75
5.2.3 Service of Read Operations 77
5.3 Performance Evaluation 78
5.3.1 Experiment Setup 79
5.3.2 Experiment Results with Synthetic Workloads 80

5.3.3 Experiment Results with Real-world Workloads 84
5.4 Conclusion 88
6. CONCLUSIONS AND FUTURE WORK 90
6.1 Contributions 90
6.2 Future Work 91
REFERENCES 93

CHAPTER 1
INTRODUCTION

In the big data era, data from all sources have shown explosive growth. The rapid growth in data volumes poses a critical challenge to system designers for providing highly efficient storage infrastructure support to upper-layer software, which is a must to guarantee service quality for users.

Data deduplication, an important data reduction technique, has been introduced in both primary and backup storage systems for efficient data storage. And it has become a standard feature in commercial storage systems, including Dell-EMC Data Domain system and NetApp ONTAP system. With data deduplication, not only can the size of the data stored in the storage devices be significantly reduced, but also the I/O or network bandwidth can be saved. However, deploying deduplication technique in storage system is not free, and it can introduce extra computation and I/O cost compromising the system performance significantly. In order to perform data deduplication in a storage system, the input data (usually files) need first to be partitioned into small pieces or chunks before stored on the disk (if necessary) in a so-called chunking phase. Duplicates are detected by checking the uniqueness of the chunks' fingerprints. There the three sources of overheads introduced by the deduplication. First, the chunking process can be time-consuming and become a new performance bottleneck of the system if the content-defined chunking approach is used for high deduplication ratio. Second, calculation of fingerprints of data chunks is CPU-intensive task and can add significant computation cost. Third, maintaining necessary metadata for a deduplication system introduces extra requirements on data consistency and persistency, which can significantly compromise the I/O efficiency of the system if it is not processed in an efficient way.

Accordingly, the major work of the thesis focuses on improving the efficiency of performing data deduplication in a storage system. And several solutions have been proposed to reduce the computation and I/O cost for data deduplication for a highly-efficient deduplication system design.

To remove the performance bottleneck posed by the chunking process in CDC-based deduplication systems, two efforts have been made to accelerate the CDC process. The first effort is to leverage the Instruction-Level Parallelism (ILP) in modern processors, that has not been fully exploited in existing parallel CDC methods, to conduct chunking in

parallel in a core. To achieve the goal, a two-stage parallel CDC method (SS-CDC) was proposed, that enables (almost) full parallelism on chunking of a file. Moreover, different from existing parallel CDC methods that compromise deduplication ratio while achieving high chunking speedup, SS-CDC provides high chunking speedup without deduplication ratio reduction. The key idea is to first locate all potential chunk boundaries in parallel (in the first phase) and then make real chunking decision sequentially (in the second phase). As long as the first phase can be fully parallelized and the second phase only accounts for a small percentage of the chunking time, the parallel chunking speedup can be retained. Meanwhile, the chunking output can be identical to that in the sequential chunking method. As a case study, by using Intel AVX-512 instructions, SS-CDC consistently obtains super-linear 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 and retain the high deduplication ratio of the sequential CDC algorithm.

The other effort to accelerate CDC is to avoid the byte-by-byte detection whiling locating a valid chunk boundary. We identify existence of a property in the deduplicatable data, named duplicate locality. This locality reveals the fact that multiple deduplicatable chunks are likely to occur together. In other words, one deduplicatable chunk is likely to be immediately followed with a sequence of contiguous deduplicatable chunks. The longer the sequence, the stronger the locality is. After demonstrating evidence of existence of such locality in real-world data, a suite of chunking techniques that exploit the locality to remove almost all chunking cost for deduplicatable chunks in CDC-based deduplication systems are proposed. The consequent deduplication method, named RapidCDC, has two salient features. One is that its efficiency is positively correlated to the deduplication ratio. RapidCDC can be as fast as a fixed-size chunking method when it is applied on data sets with high data redundancy. The other feature is that its high efficiency does not depend on existence of strong duplicate locality. Actually its efficiency with weak locality can be as high as that with very strong locality. These attractive features make RapidCDC's effectiveness almost guaranteed for datasets with high deduplication ratio. Our experimental results with synthetic and real-world datasets show that RapidCDC's chunking speedup can be up to 33× over regular CDC chunking. Meanwhile, it maintains the same deduplication ratio.

To detect duplicates, the high computation cost of calculating a collision-resistant hash value for each chunk (block) to write seems unavoidable in general, however, we identified a unique opportunity to use non-collision-resistant for deduplication in the context of journaling file system to provide high-performance file access without compromis-

ing data correctness and reliability. we proposed a light-weighted file system journaling support technique inside SSDs to provide efficient file system journaling without compromising SSD's lifetime. Journaling is a commonly used technique in file systems to provide data reliability for applications. Full-data journaling, which stores all file system (data and metadata) updates in a journal before they are applied to their home locations, provides the strongest data reliability, reduces application developers' efforts on application-level crash consistency, and helps to remove most crash-consistency vulnerabilities. However, file system users usually hesitate to use it as it doubles the write traffic to the disk, leading to compromised performance. While fast SSDs have the potential to make full-data journaling affordable, its doubled writes threaten the devices' durability, which is their Achilles heel. While data deduplication has the potential to remove the second writes to the home locations, it can be too expensive to be a practical solution with its computing and caching of collision-resistant hashing values (fingerprints). The issue is especially serious for SSDs, which are becoming increasingly large and fast, and less tolerant of additional overhead in the I/O stack. Leveraging the fact that all writes to the home locations in a file system on the data-journaling mode are preceded by corresponding writes to the journal, we propose Write-Once data Journaling (WOJ), which uses a weak-hashing-based deduplication dedicated for removing the second writes in data journaling. WOJ can reduce regular deduplication's time and space overheads significantly without compromising the correctness. To further reduce metadata persistency cost, WOJ is integrated with SSD's FTL within the device.

Lastly, we proposed a new method to address the efficiency challenge introduced by frequent metadata flushes in data deduplication systems to provide highly efficient data storage. I/O deduplication is an important technique for reducing I/O volume and increasing effective storage capacity for both backup and primary storage systems. However, I/O deduplication requires a new level of indirection between logical addresses exposed to users of storage systems and physical addresses on storage devices, and consequently needs to maintain corresponding metadata. To meet requirements on data persistency and consistency, the metadata writing is likely to make deduplication operations much more expensive, in terms of amount of additional writes on the critical I/O path, than one might expect. One fundamental reason for the bloated cost is the incompatibility between writing of usually small metadata and storage devices' block interface. we propose a new deduplication scheme, named ThinDedup, to compress the data and insert metadata into data blocks to reconcile the incompatibility. Assuming that performance-critical data are usually compressible, we can mostly remove separate writes of metadata out of the critical path of servicing users' requests. In addition to metadata insertion, ThinDedup also

uses persistency of data fingerprints to evade enforcement of write order between data and metadata.

CHAPTER 2

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

2.1 Introduction

Backup storage is a critical infrastructure in protecting users from data loss events 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 a large amount of data within a tight time window, the system has to provide sufficiently high backup performance [106, 35]. 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 [23, 61], to prevent resource contention which could degrade the performance of single-stream backups.

2.1.1 Using Deduplication to Improve Space Efficiency

Along with the backup performance, space efficiency is also an important aspect for a backup storage system. Backup files usually contain a large amount of duplicate data, while changes between two consecutive backups can be small. Accordingly, data deduplication is often used in a backup storage system 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 [92] has demonstrated significant space saving from deduplication, achieving deduplication ratios from 2-14$\times$ in production deployments.

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

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

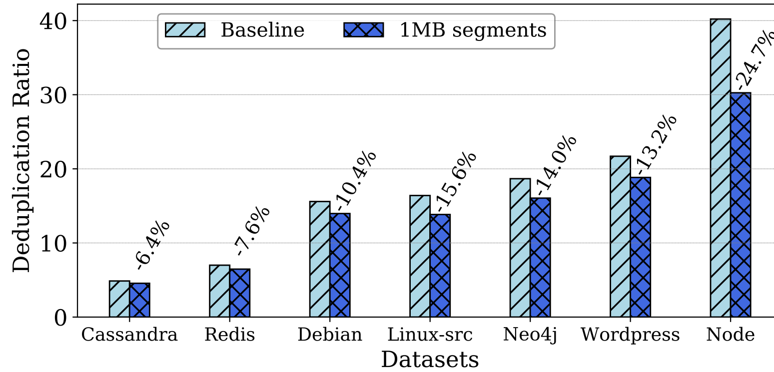


Figure 2.1: *Deduplication ratio reduction caused by existing parallel CDC approaches.*

(CDC) [57], 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 specific 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 [103, 2, 5].

2.1.2 Accelerating CDC-based Deduplication

To address the chunking 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* [97] or compatibility to the SIMD platforms, such as Advanced Vector Extensions (AVX) [20] 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 generate 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 gener-

[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.

ated 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 obtained by independently performing CDC within individual segments are different from those obtained 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 2.1 shows the comparison of deduplication ratio between sequential CDC and parallel CDC using 1MB segments, using 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 lowered 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, but cannot take advantage of instruction-level parallelism from SIMD platforms, such as AVX or GPUs. The reason is that SIMD requires simultaneous application 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 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.

2.1.3 Our Contribution

In this work, we identify the root cause of deduplication ratio degradation of existing parallel CDC 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 while provides 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 the AVX instructions for parallel chunking. Our experiments with real-world datasets show that compared to existing multithreading CDC, SS-CDC can detect up to about 47% more duplicate data, and achieve superlinear speedups (higher than the number of cores) [96], up to 3.3× more, by exploiting parallelism from AVX.

2.2 Background and Related Work

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

2.2.1 Fixed vs. Variable-size Chunking

A file can be partitioned into either fixed-size or variable-size chunks for deduplication. In fixed-size chunking, chunk boundaries are determined at offsets of multiple of a unit size. Since chunking is fast in fixed-size chunking, it is usually used in storage systems where performance is critical, such as NetApp All Flash FAS [63] or Pure Storage [71]. However, fixed-size chunking cannot address the issue of boundary shifting due to data insertion or deletion 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 rolling window of a fixed size 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 this process is repeated. To avoid generating too small or too big 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 automatically 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 depen-

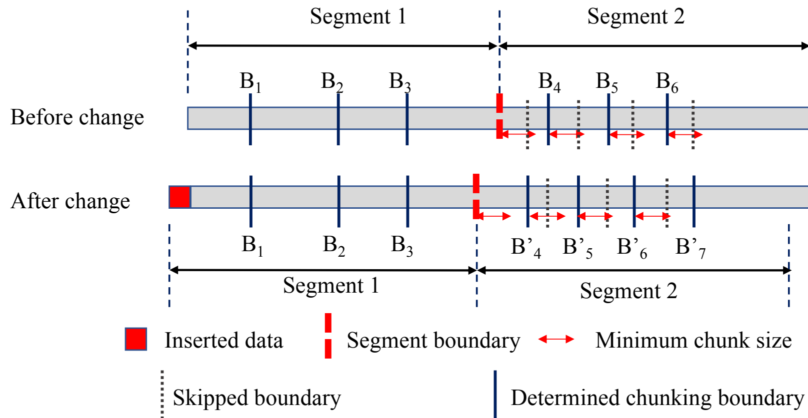


Figure 2.2: *An example showing how deduplication opportunities are lost in existing segment-based CDC methods.*

dant on hash value of the current rolling window, but also on the previous boundary's position. This places a challenge on its effective parallelization. In the meantime, because the process is highly compute intensive, it is necessary to accelerate the chunking process. To this end, there are two categories of efforts, which are optimization of the rolling hashing and parallel chunking.

2.2.2 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 [102] uses a lightweight hash function which requires 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 [103] proposed a few techniques to accelerate the Gear-based chunking process. AE [105] 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. [104] 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 SS-CDC.

2.2.3 Parallel Chunking and its Limitations

Another apprach to speed up CDC is to parallelize the algorithm and run it on a parallel hardware. Many backup systems [22, 61, 101] 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 2.2 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 2.2 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 *content-defined*, 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.

Next, we introduce a few existing parallel chunking approaches. The chunking operation in P-Dedup uses a segment-based and multithreading approach [101]. It made efforts to achieve chunking invariability. After chunks within each segment are determined, a master thread does additional rolling hash 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 still cannot ensure chunking invariability.

The only work we are aware of that guarantees chunking invariability and also provides multithreading chunking is MUCH [97]. 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. Neverthe-

Table 2.1: *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

less, 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 [5] 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 2.1 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.

2.3 The Design

SS-CDC is a new CDC approach which enables high parallelism for CDC chunking to utilize parallel hardware's computing power without compromise of 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.

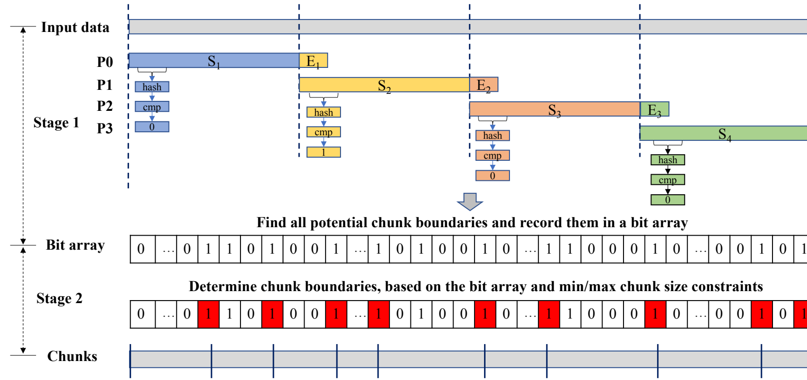


Figure 2.3: *The two stages of SS-CDC algorithm. Each thread continues the rolling hashing for extra (rolling_window_size - 1) bytes from the next segment, donated as E_1, E_2 and E_3 in the figure. P0, P1, P2, and P3 are processes (or threads) for chunking.*

2.3.1 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 checking both conditions at an offset together, 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 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 2.3 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), and each segment 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 2.3. 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 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.

2.3.2 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 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 [74] (or 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.

2.3.3 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 [74, 20], 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 [19, 18] and provide the benefit of parallel execution without requiring extra hardware support. We leave it as future work

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

Table 2.2: *Multi-threaded Chunking Implementations on Multi-cores with AVX*

	Sequential on Files	**Parallel on Files**
Sequential on Segments in a Core	SFSS (w/ dedup ratio loss)	MFSS
Parallel on Segments in a Core	SFMS	MFMS

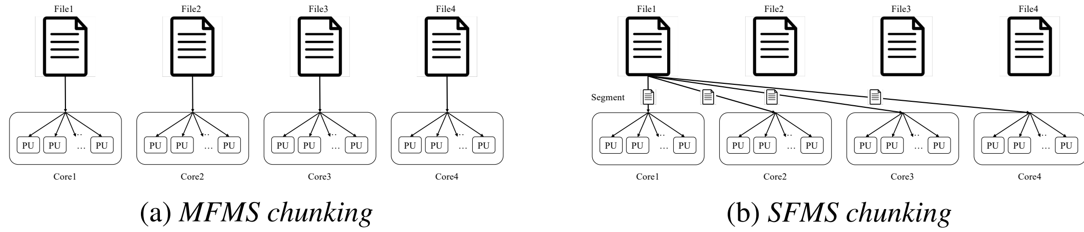
to port SS-CDC to other parallel hardware. For processors with AVX-512 instructions support, the extended registers are 512-bit long.

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 2.1 and 2.2. Like many other efficient CRC implementations [30], 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 2.1, the contribution of the leftmost byte leaving the window is removed, while in Equation 2.2 the contribution of the byte entering the window from the right is added to the value. We use a rolling window of 256 bytes.

$$hash_tmp = hash_old \oplus crcu[buf[offset - w]] \tag{2.1}$$

$$hash_new = (hash_tmp >> 8) \oplus crct[buf[offset]] \tag{2.2}$$

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



(a) *MFMS chunking* (b) *SFMS chunking*

Figure 2.4: *Illustrations of SS-CDC's MFMS (Multi-File Multi-Segment) and SFMS (Single-File Multi-Segment) multi-threaded designs. In either design, multi-segments are processed in multiple processing units (PUs) supporting AVX instructions in a core.*

One challenge in the implementation is 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. Instead, we 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 at a time 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 0s 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$\times$, making its running time account for only $\sim$2% of the total chunking time.

2.3.4 Multi-threaded SS-CDC on Cores with AVX

SS-CDC exploits parallelism among segments of a file on the AVX parallel hardware available on individual cores. In reality, the workload may consist of multiple files for deduplication on multiple cores. Assuming the parallelism across the cores is used in a parallel deduplication scheme, there are design choices to be made in the scheme. One is whether multiple files are processed in parallel. Or the choice is either single file (SF) or multiple files (MF) at a time. The other is whether the AVX parallelism within each core is used to process multiple segments simultaneously. Or the choice is either single

segment (SS) or multiple segments (MS) at a time in a core. For a multiple-thread SS-CDC on on multi-cores supporting AVX, there are two efficient options, which are multi-file multi-segment (MFMS) and single-file multiple segments (SFMS), as illustrated in Figure 2.4. In contrast, existing multi-threaded deduplication scheme is either MFSS or SFSS, as summarized in Table 2.2.

In the MFMS design, a file is processed by only one thread running at a core. In the thread, the file is partitioned into segments, which are then processed in parallel with the AVX-512 instruction. The single thread will process both chunking stages (filling the bit array and scanning it for true boundaries) at the core, as described in Section 3.3. If there are a sufficient number of files available to keep all cores busy, MFMS is an efficient implementation of multi-threaded SS-CDC. However, there may not be enough files ready for deduplication at a time. A backup file is usually very large, and a deduplication workload may contain a few big files received via the network at a time. Furthermore, a backup file may consume a large amount of memory. The amount of available memory may limit the number of backup files for parallel deduplication. In such scenarios, an SFMS design of the multi-threaded SS-CDC becomes necessary.

In an SFMS implementation, a backup file is first partitioned into fixed-size segments, which are placed into a segment queue. There is a chunking thread at each core, which 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 the reduced cost of calculating hash value for the rolling window, as an incremental hash function is used for the calculation. When all the segments of a file complete their first-stage processing, a barrier synchronization among the threads is required. 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 reduce the synchronization cost, 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. In the SFMS implementation, an apparent concern is the serial processing at the second stage. However, its impact is small. First, this stage involves very little computation accounting for about 2% of a file's total chunking time. Second, during the second-stage processing by one thread the other cores can be used to processing other file(s). The SS-CDC's SFMS implementation is necessary when deduplication of a particular file needs to be accelerated or only a small number of files are available at a time for

Table 2.3: *Real-world datasets used in the experiments. All the Docker images are downloaded from Docker Hub [24].*

Name	Size (GB)	# of files	Dedup Ratio	Description
Cassandra	14.2	40	5.0	Docker images of Apache Cassandra, an open-source storage system [9].
Redis	4.1	34	7.2	Docker images of the Redis key-value store database [77].
Debian	9.5	92	15.8	Docker images of Debian Linux distribution (since Ver. 7.11) [39].
Linux-src	570	1013	16.4	Uncompressed Linux source code (v3.0∼v4.9) downloaded from the website of Linux Kernel Archives [86].
Neo4j	46.0	140	19.0	Docker images of neo4j graph database [60].
Wordpress	181.7	501	22.0	Docker images of WordPress rich content management system [41]
Nodejs	800.0	1567	41.4	Docker images of JavaScript-based runtime environment packages [40]

chunking. It is noted that while SFSS can also accelerate a single file's chunking, it cannot exploit parallelism within a core for chunking multiple segments simultaneously.

2.4 Evaluation

We answer the following questions in the evaluation. First, how much speedup does 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 2.3. For the Linux source code, we downloaded all 1013 versions (from 3 to 4.9) from the Linux Kernel Archives [86]. Each version is converted into a file of the mtar format [51] for backup. The others are six groups of

Docker images, downloaded from Docker Hub [24], 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 [57].

2.4.1 Chunking Speed

With the instruction-level parallelism from using AVX, 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 the different datasets and see how the use of the AVX instructions improves the chunking performance. Figure 2.5 shows the speedups of SS-CDC's chunking performance over sequential CDC with one thread running on one core. The speedups are very consistently, at 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.

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 a few 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 see next, the minimum chunk size has a considerable impact on the speedups of SS-CDC. Secondly, while the chunking process is CPU intensive, it also includes substantial memory accesses which load data from the memory to registers for processing. 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 of SS-CDC 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 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

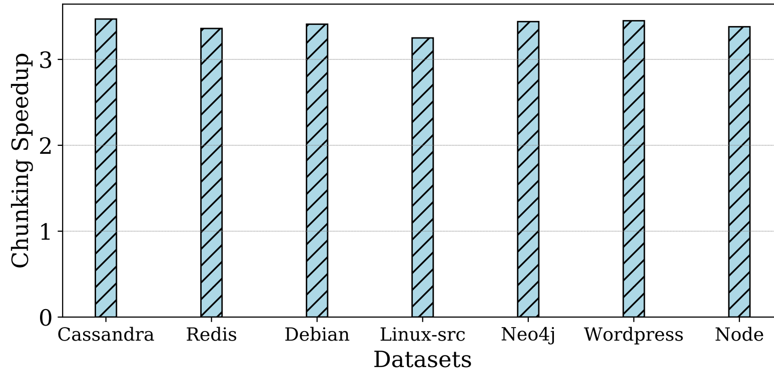


Figure 2.5: *Single-thread SS-CDC's chunking speedups over the sequential CDC at one core. The minimum, expected average and maximum chunk sizes are 2KB, 16KB, and 64KB, respectively.*

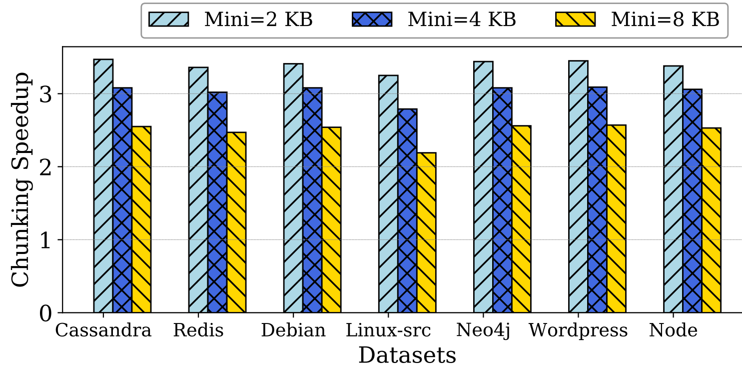
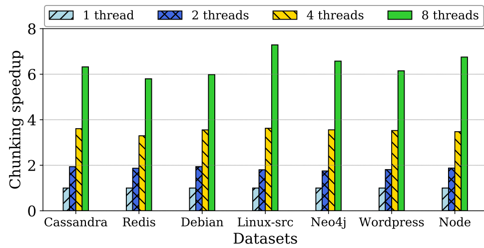


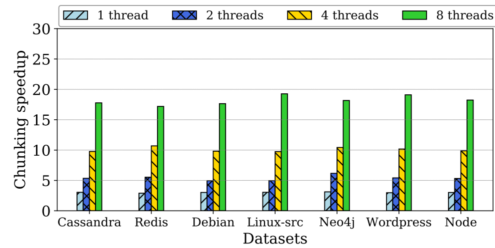
Figure 2.6: *Chunking speedups when different minimum chunk sizes are used. The expected average and maximum chunk sizes are 16KB and 64KB, respectively.*

are presented in Figure 2.6. 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) which do not use AVX. Their chunking speeds were normalized to the sequential CDC without using AVX, one file at a time, on one core.

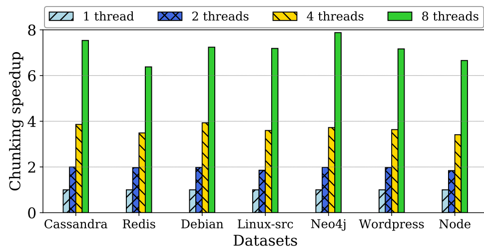


(a) *SFSS Regular CDC*

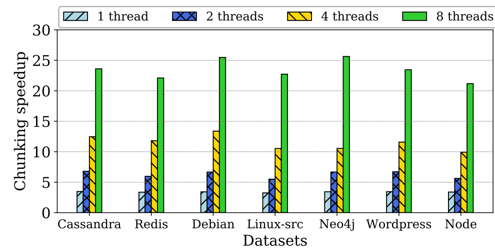


(b) *SFMS SS-CDC*

Figure 2.7: *Chunking speedups of multithreading regular CDC and multithreading SS-CDC over sequential CDC at one core with different datasets and thread/core counts when a file is processed by all threads.*



(a) *MFSS Regular CDC*



(b) *MFMS SS-CDC*

Figure 2.8: *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.*

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 2.7 for single file chunking and Figure 2.8 for multiple file chunking. From (a) in both figures (note the Y axes in (a) and (b) are using 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 2.8a) where each file is processed independently by a chunking thread. For example, for the

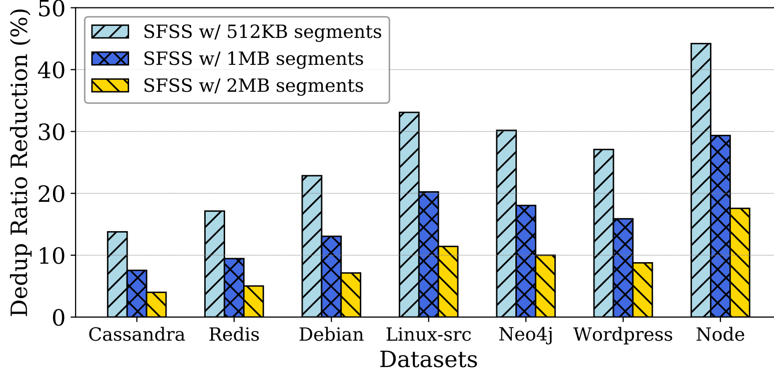


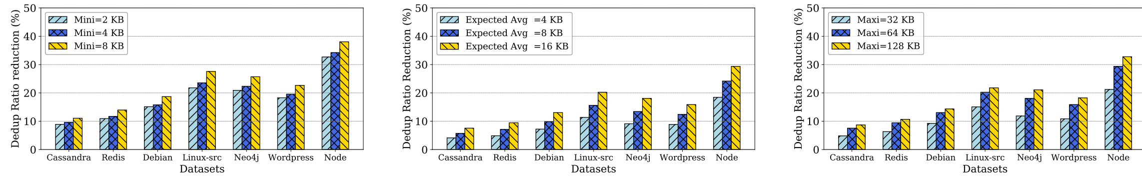
Figure 2.9: *Deduplication ratio reduction of the regular multithreading CDC chunking approach (SFSS), compared with that of SS-CDC's SFMS implementation (on 8 cores) when different segment sizes are used.*

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 $\sim 3\times$). 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 2.3. In spite of that, while SFMS cannot achieve the same speedups as MFMS, SFMS still achieves superlinear speedups. In the meantime, it unlocks the opportunity of exploiting intra-file chunking parallelism.

2.4.2 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-



(a) *Varying minimum chunk sizes*

(b) *Varying expected average chunk sizes*

(c) *Varying maximum chunk sizes*

Figure 2.10: *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.*

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 2.9 presents the deduplication ratio reduction from SFSS when we vary the segment size. The reduction is compared with the deduplication ratio from SS-CDC (and also sequential CDC). The results show that 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 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 2.10, 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 2.10a. 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 2.10b, 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 size, the maximum chunk size also affects deduplication ratio. Figure 2.10c 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.

2.5 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 level of deduplication 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.

CHAPTER 3

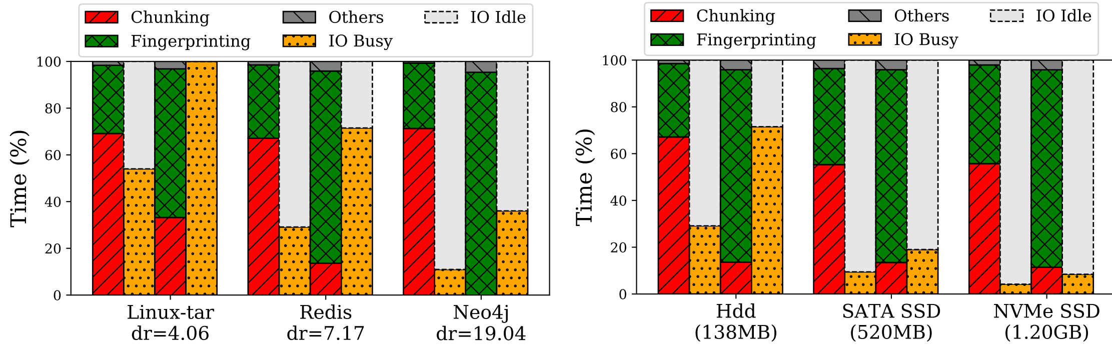
RapidCDC: The Duplicate Locality and its Use to Accelerate Chunking in CDC-based Deduplication Systems

3.1 Introduction

With explosive growth of data volume and rising demand on high storage space efficiency and high performance, deduplication (or dedup hereafter) techniques have been widely deployed in various storage systems, including Dell EMC Data Domain [22], and NetApp ONTAP system [62].

There are two metrics to assess a dedup technique, namely dedup ratio and dedup speed. The dedup ratio is the ratio between sizes of a data set before and after a dedup operation. For datasets of low redundancy, the dedup technique may not be employed. However, for those of sufficient redundancy, different dedup techniques may produce vastly different dedup ratio. A dedup technique's ability of detecting and removing redundancy for high dedup ratio largely relies on its chunking method. Assuming a file is to be deduplicated, one may use either fixed-size chunking (FSC) or content-defined chunking (CDC) methods to partition the file into chunks before their fingerprints are computed and compared to detect redundancy. Between the two methods, CDC often produces much higher dedup ratio than FSC, as it can accommodate various differences between stored files and files to be written. With FSC, a file is segmented into fixed-size chunks from its beginning. If any insertions or deletions are applied at the beginning of a stored file and their sizes are not exactly multiples of a predefined chunk size, all chunk boundaries in the file are shifted, causing boundary-shift issue [57]. Consequently, few chunks can be deduplicated, though a majority of the file is unchanged. CDC effectively addresses the issue by forming chunks based on the file content instead of at fixed file offsets.

However, CDC's operation of detecting chunk boundaries, or forming the chunks, can be very expensive, as it may have to scan a file byte-by-byte to avoid missing any potential chunk boundary and thus compromising dedup ratio. Specifically, a common practice of a CDC method is to use a fixed-size window to roll over the file. It applies a hash function on the byte sequence covered in the window and compares the hash value to a predefined value. If matched, a chunk boundary is declared at the end of the window. In principle, the window rolls forward by only one byte after every hash function computation and hash value comparison. In reality, CDC needs to avoid a chunk that is too small for



(a) *Datasets of different deduplication ratios (shown as "dr" with respective dataset names) on the hard disk.*

(b) *The "Redis" dataset on disks of different sequential write throughput (shown with respective disk types).*

Figure 3.1: *Breakdown of CPU time and disk time with CDC dedup of different datasets on different disks. In each bar group, the first two bars are for regular CDC chunking, and the last two are for the proposed RapidCDC.*

lower dedup metadata overhead or is too large for higher dedup ratio. To this end, a minimum chunk size and a maximum chunk size are set for the CDC method. When a chunk boundary is detected, or a chunk is formed, CDC moves the window forward by the minimum chunk size before it resumes its byte-by-byte rolling. When the window rolls away from the last boundary for the maximum chunk size without detecting a new boundary, the current window position is declared as a boundary (at the end of the window).

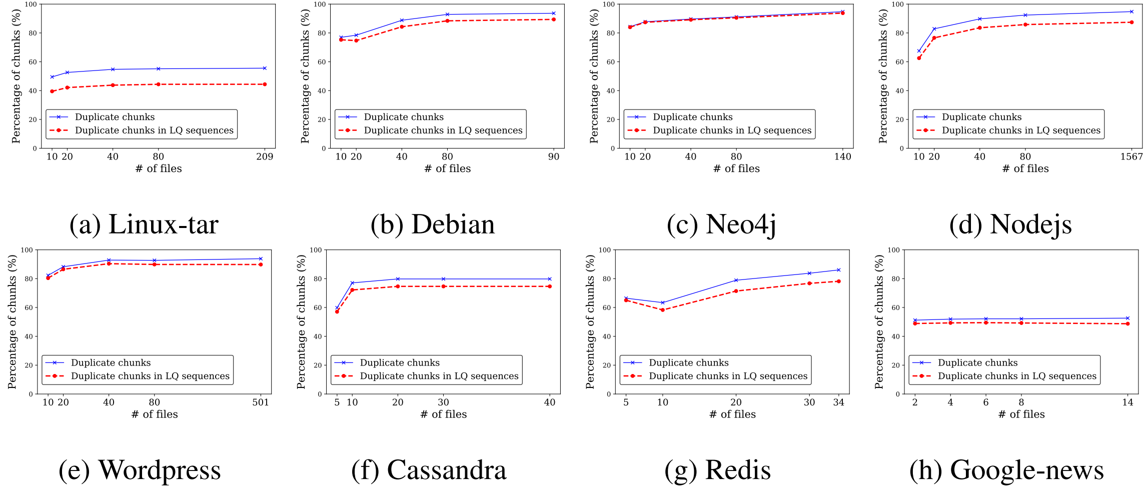
Even with introduction of the minimum chunk size, a significant portion of a file still has to be scanned with the hash function computation during the window rolling. For a CDC system whose minimum, maximum, average chunk sizes are 4KB, 12KB, and 8KB, respectively, about half of the bytes in a file are scanned [104]. For a storage system admitting data at the speed of 1GB/s, the CDC dedup component of the system must carry out around 500 million of such function computations per second so that the component itself does not become the storage system's performance bottleneck. However, it can be a serious challenge for the current CDC technique to make the I/O devices, such as hard disks and SSDs, instead of its own operations, be the performance bottleneck.

To illustrate whereabout of the bottleneck, we run the rolling-window-based CDC dedup on datasets of different dedup ratios and residing on the disks of different speeds. The datasets are described in Table 3.1. Detailed experiment setup is depicted in Section 4. In each experiment, duplication within each dataset is detected and removed by a CDC dedup system using Rabin as its hash function for rolling window and SHA-1 as its fingerprinting function. The minimum, expected average, and maximum chunk sizes are set at 4KB, 8KB and 12KB, respectively. Non-deduplicated data are asynchronously written to

the disks, so that the CPU time and disk time can be maximally overlapped. The first two bars in each bar group in Figure 3.1 show the breakdown of CPU time and disk time normalized over the corresponding dataset's total writing time. As non-deduplicated data are always written to a disk sequentially as a log, the percentage of disk busy time is calculated as the ratio of actual amount of written data and maximum amount of data that can be written to the disk under its peak sequential write throughput. From the figure we can see that the disk busy time percentage, or its utilization, is low. With either increasingly high dedup ratios (see Figure 3.1(a)) or increasingly high speed disks (see Figure 3.1(b)), the disk busy time holds accordingly smaller share of the disk time. In other words, the dedup's CPU time becomes an increasingly serious performance bottleneck. This is especially problematic for an archive storage system providing data backup service, where dedup is almost always deployed, as it usually has a high dedup ratio and uses a disk array consisting of a large number of disks for high throughput. Because collision-resistant hash function has to be employed on each chunk to generate its fingerprint, the fingerprinting cost can hardly be reduced. Fortunately, chunking is usually a more expensive CDC operation. As shown in the figure, chunking consistently represents over 60% of CPU time and therefore a great potential for CDC's performance improvement. In this paper, we aim to remove almost all of the chunking cost for workloads of high dedup ratio. To this end, we will first identify an important but currently much ignored property, named duplicate locality, and show how it may help accelerate chunking speed (Section 2). We will exploit the property to design a suite of chunking techniques with different trade-offs between performance risk and gain (Section 3). In Section 4, we extensively evaluate the CDC dedup strategies adopting these chunking techniques, named RapidCDC, with synthetic and real-world datasets to assess improvements of the chunking speed and entire dedup system's performance. The last two bars in each bar group in Figure 3.1 show the results with RapidCDC, where chunking time is significantly reduced and the disks are better utilized.

3.2 The Duplicate Locality

Existence of duplication is often a result of limited updating of an existing dataset. Naturally when an update operation, such as insertion, deletion, or overwrite, occurs at a file offset, there is a tendency that file contents around the offset are more likely to be updated. That is, the updates are likely to be unevenly distributed in a dataset. In other words, non-updated data, or duplicates, are likely to be contiguously laid out in a file. We name this layout of duplicate data *the duplicate locality*. In the context of dedup, this locality refers to the phenomenon that duplicate chunks are likely to stay together. When one such duplicate chunk is detected, the chunks around it are likely to be also deduplicatable.



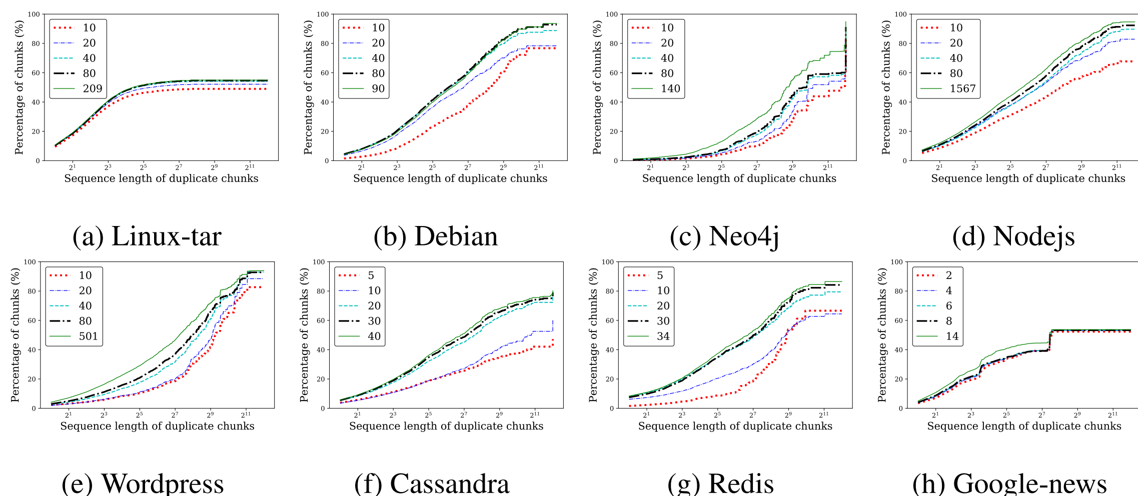
(a) Linux-tar (b) Debian (c) Neo4j (d) Nodejs

(e) Wordpress (f) Cassandra (g) Redis (h) Google-news

Figure 3.2: *Percentage of all duplicate chunks and percentage of duplicate chunks in the LQ sequences among all chunks when increasing number of files in a dataset are admitted into the system.*

Because chunks are not very large in practice (usually tens of KBs) for high dedup ratio, the duplicate locality at the chunk granularity tends to be strong. We use the number of contiguous deduplicatable chunks immediately following the first deduplicatable chunk to quantify the locality. These deduplicatable chunks constitute a chunk sequence, named locality-quantification sequence, or *LQ sequence* in short. Existence of such sequences motivates our proposed RapidCDC that may significantly reduce chunking cost. The longer the sequences are, the stronger the locality is. If the sequences' length is always 0, either there are not any duplicate chunks or individual duplicate chunks are all isolated by non-duplicate chunks. To investigate the existence and strength of the locality, we examine percentage of chunks in the sequences (Figure 3.2) and the sequence lengths' distribution (Figure 3.3) for a selected group of real-world datasets. As detailed in Table 3.1, the data sets cover various application domains, including Linux source code as tar files ("Linux-tar"), Linux distribution as Docker images ("Debian"), graph database ("Neo4j"), JavaScript-based runtime environment packages ("Nodejs"), WordPress container images ("Wordpress"), Apache Cassandra snapshot images ("Cassandra"), Redis key-value store backup images ("Redis"), and daily Google news archives ("Google-news"). As the investigation is on locality in terms of chunk sequences, we use a rolling-window-based CDC method to identify chunks in files. And still we use the 4KB-8KB-12KB configuration (for the minimum, expected average, and maximum chunk sizes, respectively) in the chunking.

In each experiment files in a dataset are sent to the CDC dedup system one at a time. When a certain number of files, such as 10, 20, up to the total file count in the dataset, are admitted into the system, we count all duplicate chunks and those duplicate chunks in the



(a) Linux-tar (b) Debian (c) Neo4j (d) Nodejs

(e) Wordpress (f) Cassandra (g) Redis (h) Google-news

Figure 3.3: *CDF (Cumulative Distribution Function) curves of LQ sequence lengths when a certain number of files in a dataset are admitted into the system. The number of currently admitted files is shown in the legend.*

LQ sequences, and show their respective percentages over total number of chunks at the time in Figure 3.2. The percentage of all duplicate chunks, shown as the upper lines in the figures and named accordingly as upper percentage, positively correlates to the dedup ratio. In contrast, the percentage of chunks in the LQ sequences, named lower percentage, exhibits existence of duplicate locality. The gap between the two percentages indicates the percentage of isolated duplicates, or duplicates without the said locality. As we observe the two percentages for each data set when different number of files are admitted in Figure 3.2, the gap is small (mostly less than 5% and up to 10%), and the majority of duplicate chunks are in the LQ sequences. Another observation is that these two percentages are correlated. A dataset of a higher dedup ratio has higher upper percentages. It accordingly has higher lower percentages, which makes RapidCDC more effective, as to be revealed in the next section.

To further see the strength of the duplicate locality, quantified by length of the LQ sequence, we show percentage of chunks that stay in an LQ sequences whose length is smaller than a certain threshold for different data setsat the time when different numbers of files are admitted in Figure 3.3. From the figure we can see that only a small percentage of chunks are in very short LQ sequences, such as those whose lengths are not more than 8 chunks, as a majority of chunks are duplicates. For some datasets of high dedup ratio, such as Debian and Wordpress, there can be more than 50% of chunks are in the LQ sequences whose lengths are longer than 64 chunks. When more files are added into the system, there tends to be more chunks in the shorter LQ sequences. Even so, across all the datasets and with various number of files are admitted a majority of duplicate chunks stay in relatively

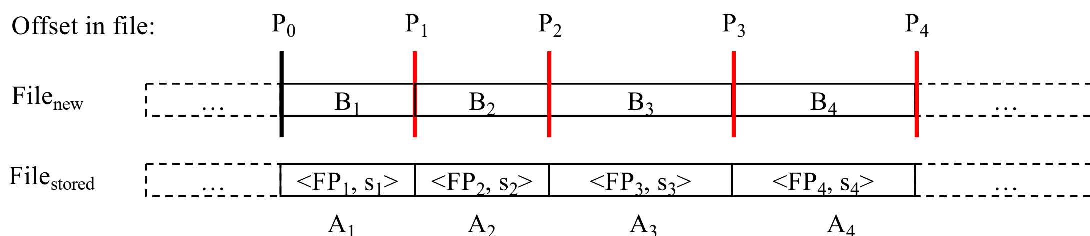


Figure 3.4: *An illustration of the idea of RapidCDC for determining chunk boundaries rapidly. A_{1-4} and B_{1-4} are chunks, FP_{1-4} are fingerprints of chunk A_{1-4}, respectively. s_{1-4} are size (in bytes) of chunk A_{1-4}, respectively.*

long sequences, demonstrating strong duplicate locality. This is an encouraging result. Interestingly, as we will show, RapidCDC's effectiveness is not sensitive to the locality's strength. Instead, it is correlated only to the percentage of chunks in LQ sequences of *any* lengths.

3.3 The Design of RapidCDC

The key technique of RapidCDC is to exploit the duplicate locality in the datasets to enable a chunking method that can detect chunk boundaries without a byte-by-byte window rolling. Due to existence of the locality, immediately following a current deduplicateable chunk, denoted B_1 in Figure 3.4, in a file named $File_{new}$, the B_2 chunk can also be a duplicate. The question is where the end boundary B_2 is. To this end, current CDC chunking method would take a window rolling over possibly a large number of bytes, one byte at a time with hash function calculation and comparison. The number is the difference between B_2's size and the minimum chunk size, which is usually a count of a few thousands of bytes or more.

3.3.1 Quickly Reaching Next chunk's Boundary

Let's assume the fingerprint of B_1 in $File_{new}$ matches fingerprint FP_1, which is currently in the dedup system and is associated with a unique (physical) chunk of data in the storage system, which is mapped to by at least one logical chunk. Assume one of the logical chunks is Chunk A_1 in a file named $File_{stored}$ that is currently stored in the storage, as illustrated in Figure 3.4. In a CDC dedup system each file has a recipe recording the mapping between each of its logical chunks and its mapped physical chunk for rebuilding file content [87, 28, 53, 79, 91, 92, 93]. Because the system maintains the mapping from fingerprints to their respective physical chunks, a file's recipe only needs to record its chunks' fingerprints along with their respective chunk sizes in the order of

their occurrence in the file. For example, $File_{stored}$'s recipe is composed of a sequence of records [..., (FP_1, s_1), (FP_2, s_2), ...], where s_1 and s_2 are corresponding chunks' sizes.

As B_1 in $File_{new}$ shares its fingerprint (and exact data content) with A_1 in $File_{stored}$, their respective next chunks, B_2 and A_2, and likely to have the same content. Accordingly, we may use s_2, Chunk A_2's size, in $File_{stored}$'s recipe as a hint to speculate B_2's size, and directly move the rolling window on $File_{new}$ to the corresponding file position, P_2, as shown in Figure 3.4. The key enabling operation is to obtain the s_2 size from the duplicate chunk A_1. A naive approach is to maintain a pointer for each fingerprint pointing to its corresponding recipe record (e.g., from Fingerprint FP_1 to (FP_1, s_1), the record in $File_{stored}$'s recipe). For example, we can following this relationship chain to obtain s_2 after knowing B_1 is a duplicate chunk: $B_1 \rightarrow FP_1 \rightarrow (FP_1, s_1) \rightarrow (FP_2, s_2) \rightarrow s_2$. While this approach works, it unnecessarily increases complexity and overhead by involving recipes in the operation. A recipe may be lost after its corresponding file is removed, making corresponding pointers invalid. In the meantime, a sequence of its fingerprints may remain in a different file's recipe with a sequence of duplicate chunks in the corresponding file, and the pointers need to be adjusted accordingly. It can be expensive to adjust the pointers in response to changes of recipes. Furthermore, as a fingerprint can be associated with multiple logical chunks, or multiple recipe records, it may be attached with multiple pointers. This may further increase cost of maintaining the pointers.

To address this issue, RapidCDC adopts a much more simple and efficient method. The CDC-based dedup process always starts from the beginning of a file and moves sequentially towards the end of the file. For any two consecutive chucks in a file, say A_1 and A_2, we record the size of A_2, say s_2, along with the fingerprint of A_1, say FP_1. In the example shown in Figure 3.4, the duplicate chunk B_1 can use a simpler relationship chain ($B_1 \rightarrow FP_1 \rightarrow s_2$) to obtain the suggested size (s_2) of its next chunk B_2 without involving recipes. While in different files, A_1 having Fingerprint FP_1 may be followed with chunks of different sizes, we allow a list of next-chunk sizes, named *size list*, to be attached to FP_1. Because the size of a chunk falls within a relatively small range (between the minimum and maximum sizes), the size can be efficiently represented (e.g., 2 bytes for a range of [2KB-64KB]). As a fingerprint itself needs tens of bytes for its storage (e.g., 20 bytes for a SHA-1 fingerprint), recording a few next-chunk sizes with a fingerprint is well affordable.

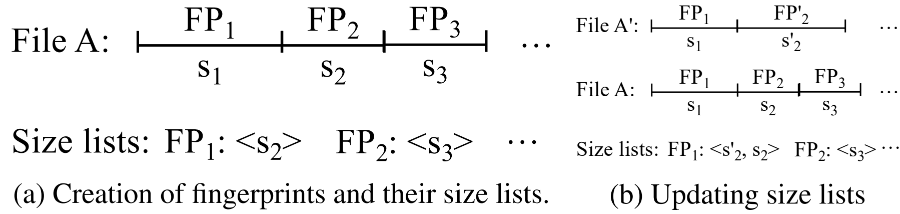
The next-chunk sizes attached to a fingerprint are actually hints of next chunk's boundary position. With such a hint, the rolling window can directly jump to the suggested position. If the position is accepted, RapidCDC avoids rolling the window one byte at a time for thousands of times to reach the next chunk boundary in regular CDC chunking. We will detail criteria of the acceptance in the next section. If not accepted, it will try another

next-chunk size in the size list of the duplicate chunk's fingerprint. Only when none of the sizes in the list is accepted, RapidCDC moves the window back to the position which is the minimum chunk size after the last chunk, or the position where a regular CDC would use after the last chunk. It then rolls the window byte-by-byte as the regular CDC does until a new duplicate chunk is found. As soon as a new duplicate chunk is found, RapidCDC's window jumps again attempting to take advantage of expected duplicate locality. In this way, RapidCDC can flexibly switch its window rolling between a fast forwarding mode and a byte-by-byte slow movement mode to maximally exploit duplicate locality and perform chunking as fast as possible.

3.3.2 Accepting Suggested Chunk Boundaries

After a duplicate chunk is detected, we retrieve the size list associated with its fingerprint, which provides hints for possible sizes of its next chunk, or accordingly its next chunk's possible end boundaries. We will check each of the sizes in the list in order until a chunk size (or the corresponding chunk boundary) is accepted or none of them can be accepted. There are four candidate acceptance criteria possibly adopted in RapidCDC, each with different trade-offs between performance gain and risk of performance penalty.

- FF (Fast-forwarding only). The suggested chunk boundary is always accepted. This is the most aggressive criterion for fast chunking. However, it may risk dedup ratio by choosing invalid chunks and missing true chunk boundaries. Our experiments with real-world datasets show the risk is very low due to RapidCDC's ability of switching to byte-by-byte detection whenever un-deduplicatable chunk is found (see Section 3.4.5).
- FF+RWT (Rolling Window Test). The suggested boundary is verified by one rolling window computation and comparison at the boundary position. Like the window used in a regular CDC chunking, its hash value is computed and compared to the pre-determined value. Only when they are equal and accordingly a valid chunk is confirmed, the boundary is accepted.
- FF+MT (Marker Test). Instead of computing the hash value of the window in the FF+RWT criterion, which can involve tens or hundreds of bytes, in FF+MT RapidCDC compares the last byte, treated as a marker, of the two chunks under consideration (e.g., A_2 and B_2 in Figure 3.4 after duplicate chunks A_1 and B_1 are found.). The boundary is accepted if the two bytes are equal. This criterion requires recording last byte of a chunk along with its size in the size list. This marker-byte comparison needs few instructions, and is faster than rolling window test.



(a) Creation of fingerprints and their size lists. (b) Updating size lists

Figure 3.5: *Maintenance of size lists*

- FF+RWT+FPT (Fingerprint Test). This is the most stringent criterion. After the boundary passes the rolling window test, FF+RWT+FPT additionally computes the fingerprint of the chunk delimited by the suggested boundary and tests whether the fingerprint currently exists (or whether the chunk is a duplicate). Only if the second test is also passed is the boundary accepted. By computing the fingerprint, this verification process is the most expensive one. However, if the fingerprint test is passed, the chunk is confirmed to be a duplicate chunk and can be readily deduplicated. Otherwise, it has a relatively high performance penalty.

3.3.3 Maintaining List of Next-chunk Sizes

As each fingerprint is associated with a size list providing hints on next chunk's end boundary, maintenance and use of the hints can be performance-critical. As illustrated in Figure 3.5(a), File A is sent to the CDC dedup system for storage. Assuming chunks of File A cannot be deduplicated. When its first chunk's fingerprint (FP_1) is added to the system's signature pool, its size list is created. When the file's second chunk is determined, its size (s_2) is added to FP_1's list as its first member. A list can grow. In Figure 3.5(b) File A$'$ also has a chunk whose fingerprint is FP_1. But the chunk is followed with a chunk of a different size ($s_2' \neq s_2$). s_2' is then added to the FP_1's list, which is now $< s_2', s_2 >$.

The size list is an ordered list. Once the fingerprint of a chunk in the file matches one already stored in the system, the size values in the fingerprint's size list will be checked one by one for next chunk boundary, until a boundary is accepted, in the order as they appear in the list. Consequently, the order can have an impact on the chunking performance. An acceptance of a file offset suggested by a size value as the next chunk boundary is termed a hit on the size value; otherwise, a miss on the value. Each miss may carry a miss penalty, depending on which of the four acceptance criteria is used. Therefore, we should place size values that are more likely to produce hits at the front of the list. To this end, we use the LRU (Least Recently Used) policy, which always places the most recently hit value at the front, including new value just added into the list. For example, in Figure 3.6 FP_1's

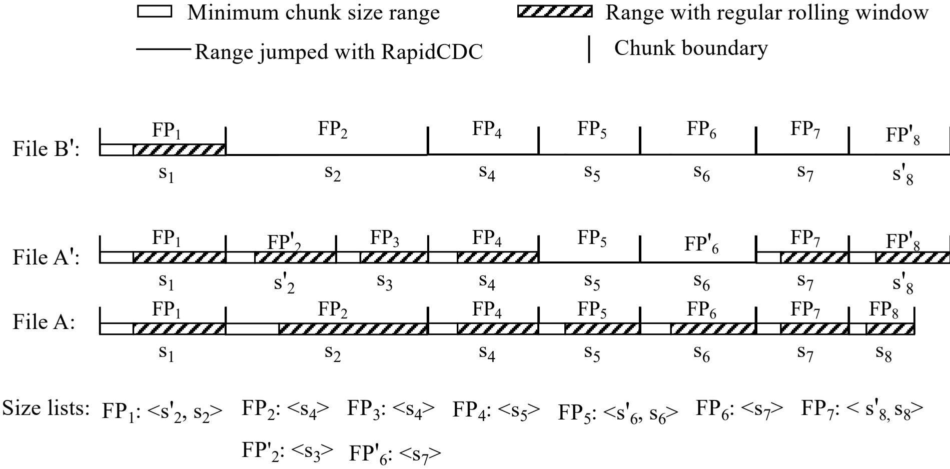


Figure 3.6: *Use of size lists to accelerate CDC chunking (The shown size lists reflect their contents after Files A and A' are stored and before File B is written.)*

list is $< s_2 >$ before File A' is stored. After File A' is stored, it becomes $< s_2', s_2 >$, where the newer size value s_2' is placed at the head. After File B is written, the list becomes $< s_2, s_2' >$ as s_2 is hit to help quickly find File B's second chunk. A common scenario is that a file is incrementally updated generating a sequence of file versions, each version resulting from updating of its previous one. Use of the LRU policy can maximize chance of hit at the first size value in a list. As shown in Figure 3.6, due to existence of duplicate locality, after sufficient history access sequences are available (Files A and A'), a new file (File B) can flexibly exploit the locality in multiple chunk sequences.

While keeping multiple size values in a size list accommodates different sequence patterns to minimize probability of switching backing to the window's slow movement mode, using a too-long list may incur excessive space overhead. As the list is managed with LRU, misses on the list can automatically evict the low-hit-ratio size values. However, if the list is too long, some low-hit-ratio size values may stay in the list for a long time period and experience misses. Any miss on a size value carries a penalty, because the cost of using the value for boundary acceptance check can be substantial, such as that for the FF+RWT+FPT criterion. Therefore, the list should be reasonably short to keep low-hit-ratio values out of the list.

3.4 Evaluations

To evaluate performance of RapidCDC, we conduct extensive experiments with both synthetic and real-world datasets.

Table 3.1: *Real-world datasets used in the experiments. All the Docker images are downloaded from Docker Hub [24].*

Name	Size (GB)	# of files	Dedup Ratio	Description
Google-news	7.2	14	2.1	Two weeks' data (10/17/2018∼10/31/2018) from news.google.com, one file for each day, collected by *wget* with a maximum retrieval depth of 3.
Linux-tar	37.2	209	4.1	Tar files of Linux source code (Ver. 4.0∼4.9.99) from kernel.org.
Cassandra	14.2	40	5.0	Docker images of Apache Cassandra, an open-source storage system [9].
Redis	100.4	34	7.2	Docker images of the Redis key-value store database [77].
Debian	9.5	92	15.8	Docker images of Debian Linux distribution (since Ver. 7.11) [39].
Neo4j	46.0	140	19.0	Docker images of neo4j graph database [60].
Wordpress	181.7	501	22.0	Docker images of WordPress rich content management system [41]
Nodejs	800.0	1567	41.4	Docker images of JavaScript-based runtime environment packages [40]

3.4.1 The Systems in Evaluation

We implement a prototype of RapidCDC that can be configured with any of the four chunk boundary acceptance criteria (FF, FF+RWT FF+MT, and FF+RWT+FPT). The rolling window size is set to 256 bytes. The default hash function applied on the content of the window to identify chunk boundaries is *Rabin* [73], which is an efficient rolling hash function that can reuse its hash computation of data that still remains in the window when the window shifts forward. This function has been widely used in CDC-based deduplications [57, 106, 47, 64, 100, 56]. In the evaluation, we also include a more lightweight hash, *Gear* [102]), to replace Rabin to reveal how the function's cost impacts RapidCDC's performance advantage. A CDC dedup system's minimum and maximum chunk sizes can be designated, and its expected average chunk size can be configured by adjusting parameters of the hash function. The default minimum/expected average/maximum chunk sizes used in the RapidCDC prototype are 4KB/8KB/12KB, respectively, which is the configuration adopted in the Dell-EMC Data Domain system [22]). In the evaluation we also test a configurations with a larger range of chunk size (2KB/16KB/64KB), which is used in LBFS, a low-bandwidth network file system using the CDC technique to reduce network traffic [57]. The prototype uses SHA-1 to compute a chunk's fingerprint. The implementation has about

2400 lines of C code, which is compiled by GCC 7.3.0 with "-O3" compiler optimization . By default, we use one thread. We will also conduct experiment with RapidCDC with multiple threads. And the default length of a fingerprint's size list is 2.

Each experiment includes a regular CDC dedup system as a counterpart of the RapidCDC system in comparison. All systems in a comparison are configured the same except stated otherwise, including the window size, the hash function, and chunk size setups. We run the systems 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. We use a hard disk as the default device for data storage. The hard disk has model number of WDC WD1003FZEX-00K3CA0 with sequential write and read bandwidths of 138MB/s and 150MB/s, respectively.

3.4.2 The Datasets

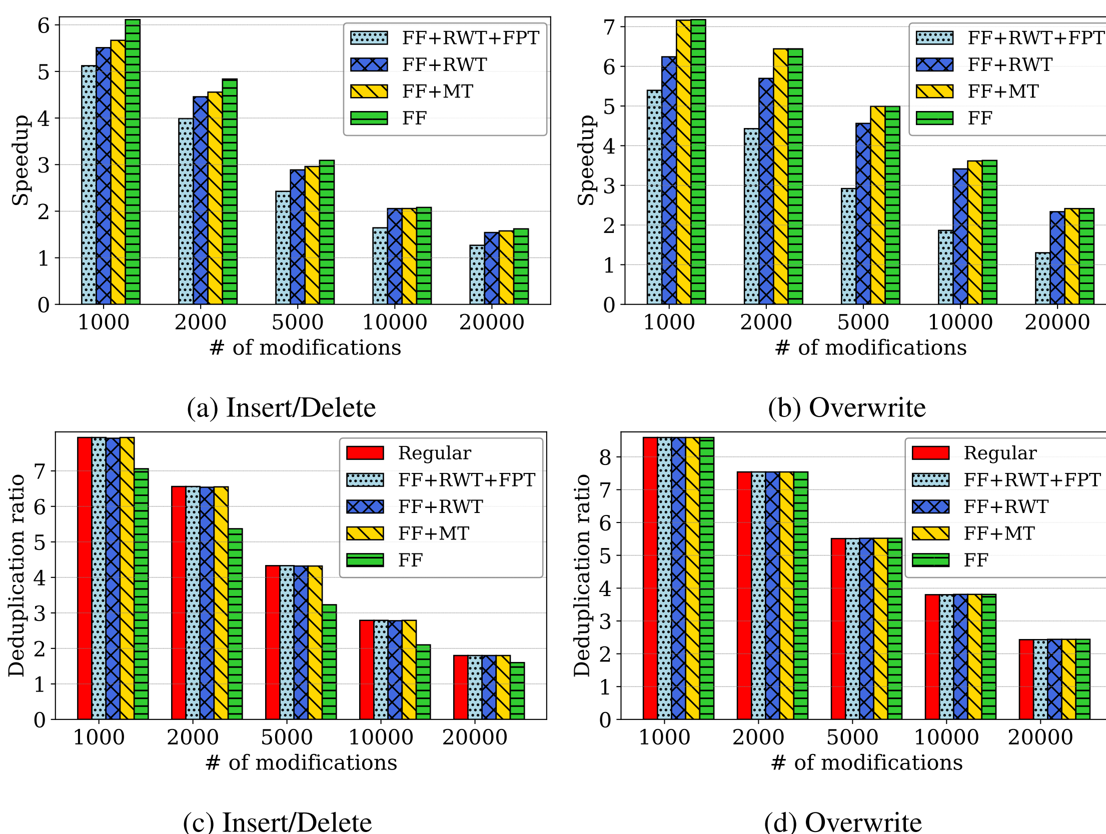
The datasets used in the evaluation include a series of synthetic datasets and eight real-world datasets as described in Table 3.1. Each synthetic dataset includes 10 files emulating a sequence of file versions with each produced after limited amount of modification over its previous version. The first version in the sequence is created with the *dd* command by copying 500MB randomly generated data from "/dev/urandom". A modification can be an insertion, deletion, or overwrite. To allow each modification to independently impact the dedup performance and ratio, we make sure each chunk receives at most one modification. The real-world datasets represent various workloads expected by a deduplication system, including the source code files, virtual machine images, database images, and internet news archives. Details of the datasets are listed in Table 3.1. The dedup ratios shown in the table are obtained by applying the aforementioned regular CDC dedup with its default configuration. As each dataset consists of a sequence of files, in a dedup experiment we write the files in a sequence, one at a time, to a dedup system and observe the deduplication speed and amount of data that can be deduplicated within the dataset itself.

3.4.3 Results with Synthetic Datasets

As mentioned, starting from the second one in the sequence of 10 files in a synthetic dataset, each chunk receives at most one 100-byte modification.

3.4.4 Impact of Modification Count and Distribution

Modifications can be categorized into two types. One may cause chunk boundary shift, including insert and delete. The other (overwrite) does not shift the boundary, and



(a) Insert/Delete

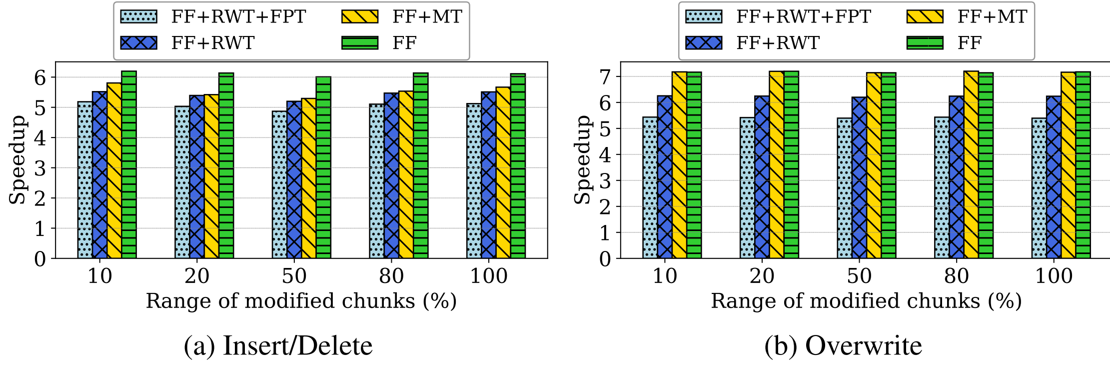
(b) Overwrite

(c) Insert/Delete

(d) Overwrite

Figure 3.7: *Chunking speed and dedup ratio for datasets with different numbers of modifications spread over an entire file.*

only changes a chunk's content. To generate a new version of file, we choose a modification type and a number of modifications and randomly apply them into a file. For the type of boundary-shift operations, we randomly choose either insert or delete. Figure 3.7 shows the chunking speed (reciprocal of total chunking time) and dedup ratio on a dataset generated with different types and different numbers of modifications for RapidCDC using different boundary acceptance criteria. The chunking speed is normalized to the speed of the regular CDC dedup on the same dataset, and is presented as speedup. As shown, RapidCDC with the FF criterion, which accepts a suggested boundary without any testing, has a consistently higher speedup. However, for Insert/Delete its advantage on the speedup comes at the cost of reduced dedup ratio. Because RapidCDC quickly switches to the slow window movement mode to look for the next legitimate boundary after a fingerprint mismatch, the reductions are limited. There is not such a reduction with datasets generated with overwrites, as they do not change the boundaries. Except with FF, RapidCDC has the same dedup ratio as the regular CDC. Across the various datasets it seems that FF+MT is a consistently well-performed choice in terms of both chunking speed and dedup ratio.

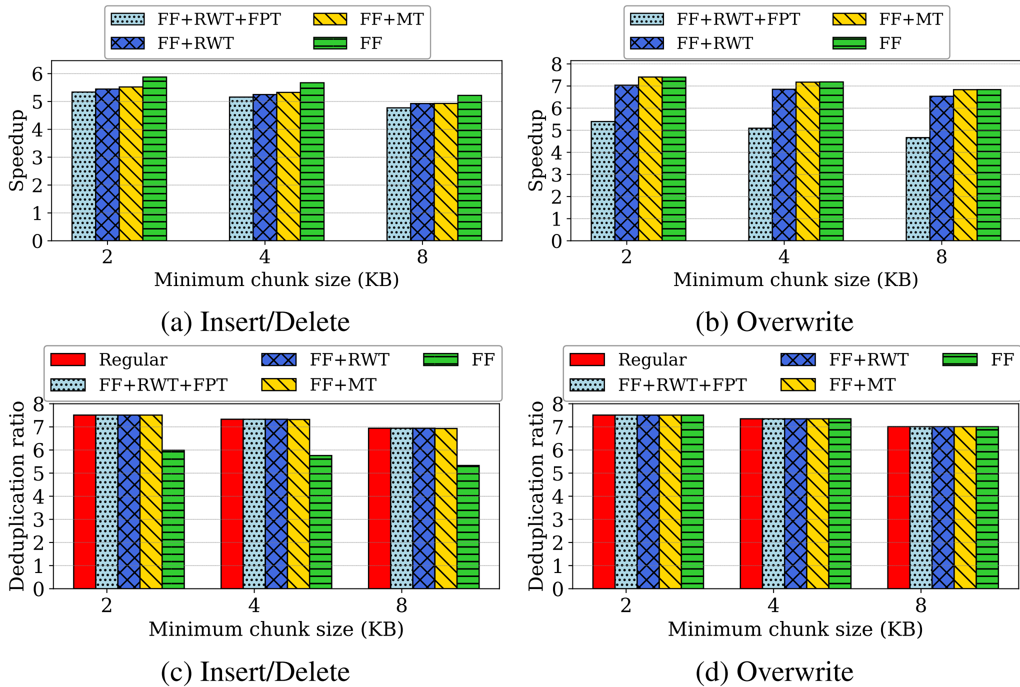


(a) Insert/Delete (b) Overwrite

Figure 3.8: *Chunking speedups for datasets where 1000 modifications are applied to different range of a file.*

Understandably the dedup ratio decreases with the increase of modification count. As RapidCDC takes advantage of duplicate sequence, which tends to become fewer and shorter with the increase, its chunking speedup is accordingly reduced. However, the speedup is always positively correlated with its corresponding dedup ratio, and is very close to the ratio. For example, for Insert/Delete and the FF RapidCDC, the dedup ratios are 7.9, 6.6, 4.3, 2.8, and 1.8 with modification counts of 1000, 2000, 5000, 10000, and 20000 in a file, respectively. And the respective speedups are 6.1, 4.8, 3.1, 2.1, and 1.6, which are close to the respective dedup ratios. RapidCDC can enable its fast forwarding mode to reach boundaries of any duplicate chunks in the LQ sequences. We speculate that the speedup can stay high regardless of distribution of LQ sequence lengths, as long as there are a sufficient number of duplicate chunks. This speculation is confirmed by the experiment results shown in Figure 3.8. Datasets used in the experiments are generated after applying the same number (1000) of modifications within the first certain percentage of a file. The speedup shows little change when the modifications are either made narrowly in a 10% file range or scattered over the entire file (100%). That is, RapidCDC's performance is not sensitive to LQ sequence length's distribution.

3.4.4.0.1 Impact of Minimum Chunk Size and Hash Function During chunking operations of a regular CDC, a rolling window also enters fast-forwarding mode to skip the minimum chunk size of bytes immediately after a chunk boundary is found. This optimization on the window rolling may have an impact on RapidCDC's relative benefit. To this end, we vary the minimum chunk size for a dedup on a dataset with 1000 insert/delete modifications randomly distributed in a file. To allow the minimum size can be varied in a larger range, we adopt a 2KB/16KB/64KB configuration, instead of the 4KB/8KB/12KB default one, for minimum/expected average/maximum chunk sizes. Figure 3.9 shows that



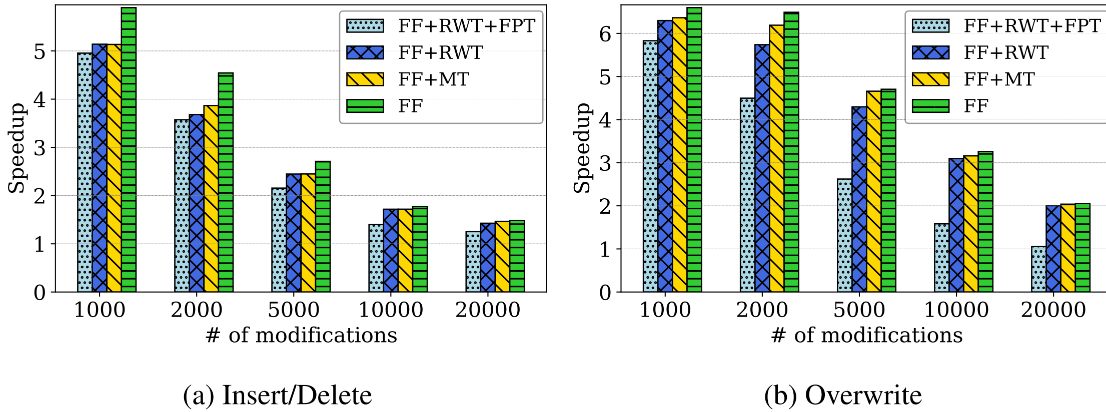
(a) Insert/Delete (b) Overwrite

(c) Insert/Delete (d) Overwrite

Figure 3.9: *Chunking speedup and dedup ratio with different minimum chunk sizes.*

the impact is small. With a decent dedup ratio, RapidCDC has removed most of the chunking time, leaving only a small number of window rollings in the slow movement mode. Therefore, the acceleration of the slow mode does not take away much of the RapidCDC's relative advantage. Furthermore, the small reduction of speedups are correlated to that of dedup ratio. A larger minimum size leads to large chunk sizes, which tends to reduce dedup ratio. RapidCDC's speedup becomes smaller with fewer duplicate chunks.

Another factor likely impacting RapidCDC's speedup is the hash function used for detecting chunk boundaries. Because the function is used very frequently in a regular CDC (at every rolling window position), chunking will become faster by using a faster hash function. The Gear [102] function used in FastCDC [102] can be 3× faster than Rabin. Figure 3.10 shows the speedups with the faster function used in both RapidCDC and regular CDC. The speedups are modestly smaller than those with the Rabin function (compared to Figure 3.7). For example, for RapidCDC's FF criterion they are 5.9, 4.5, 2.7, 1.8, and 1.5 with Gear for modification counts of 1000, 2000, 5000, 10000, and 20000, respectively. The corresponding speedups are 6.1, 4.8, 3.1, 2.1, and 1.6 with Rabin. While the window's slow rolling mode becomes faster, RapidCDC's relative advantage becomes smaller. These results also indicate that RapidCDC's performance strength is largely orthogonal to other optimization efforts on improving the CDC dedup.



(a) Insert/Delete (b) Overwrite

Figure 3.10: *Chunking speedups of RapidCDC on datasets with different modification counts and a faster hash function, Gear.*

3.4.4.0.2 RapidCDC's Worst-case Scenario RapidCDC's effectiveness relies on its successful acceptance of suggested next-chunk sizes. For any unaccepted size value, or a miss on the value, a miss penalty occurs for any acceptance criteria except FF. To gauge the impact of the miss on RapidCDC's chunking performance in the worst-case scenario, we change RapidCDC's prototype code to make none of the suggested boundaries accepted after a check using one of the criteria (FF+RWT FF+MT, or FF+RWT+FPT) and redo the experiment presented in Figure 3.7(a). The chunking speed slowdowns over the regular CDC are shown in Figure 3.11. Compared to multiple-times speedups observed in realistic datasets, the less-than 10% slowdowns are insignificant. With more modifications and accordingly lower dedup ratios, there are fewer size values to be checked and the slowdowns are smaller. Understandably such datasets are unlikely to occur.

3.4.5 Results with Real-world Datasets

We categorize the eight real-world datasets into the groups according to their redundancy quantified by dedup ratios shown in Table 3.1: a high data redundancy group (Debian, Neo4j, Wordpress, and Nodejs) and a low data redundancy group (Google-news, Linux-tar, Cassandra, and Redis), and use them to evaluate RapidCDC's dedup ratio and performance.

3.4.5.0.1 RapidCDC Performance Impact on Chunking and the Entire Dedup System. Figure 3.12 shows rapidCDC's chunking speedups with the datasets as well as the dedup ratios. For datasets of high redundancy, the dedup ratio can be over 40. But only RapidCDC with FF has speedups close to the high dedup ratios. Speedups of the other alternatives are much lower, especially for datasets of very high dedup ratios, such as Nodejs.

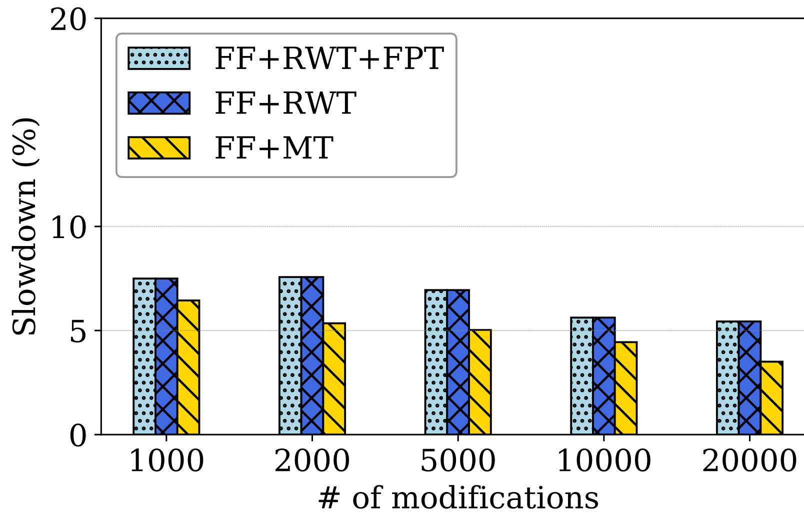
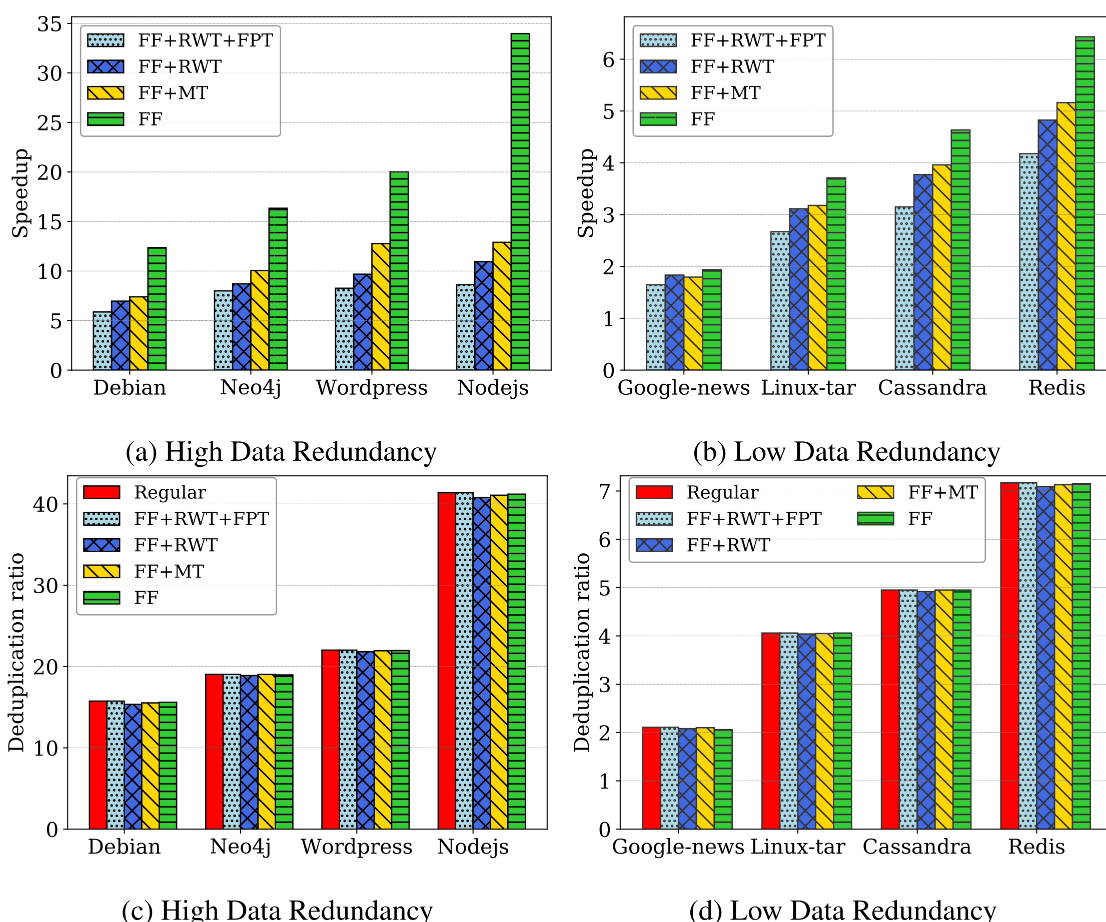


Figure 3.11: *Chunking speed slowdowns when all suggested next-chunk sizes are misses in RapidCDC.*

A speedup of around 10 is equivalent to a removal of 90% of chunking time from the regular CDC. Further improving the ratio, or removing the remaining time, requires elimination of even small operational costs, such as hashing for boundary acceptance and rolling the window back for a slow movement mode upon an unacceptance. Only FF mostly removes the costs and reaches the high ratio. Another interesting observation is that FF does not compromise the dedup ratio for the real-world datasets. To reveal the reason, we increase length of each fingerprint's size list to 4, and measure the hit ratio of each of the list's positions. A hit means the size at the position is accepted by the FF+RWT criterion. The results are shown in Figure 3.13. The first position has a very high hit ratio. For example, for the Nodejs dataset, the ratio is 99.24%, which means that in almost all cases the first sizes are the accepted ones. FF accepts the first sizes without any checking, and fortunately it indeed makes the right decision and does not sacrifice dedup ratio.

While the first position in the list has such a high ratio, we do not expect a long list could make a substantial difference on chunking performance. Figure 3.14 shows speedups of RapidCDC using FF-RWT with different size list lengths. While a miss on the entire list would cause RapidCDC to enter the slow window movement mode and have a sizable penalty, even a small hit ratio on positions other than the first one can contribute to the chunking performance. As shown, using a longer list often produces a visibly higher speedup. Meanwhile, the small contributions do not justify the space cost for keeping a long list. Therefore, its default length is 2.

To understand impact of RapidCDC's increased chunking speed on the CPU computation, we present speedups of the computation, whose two major components are chunk-



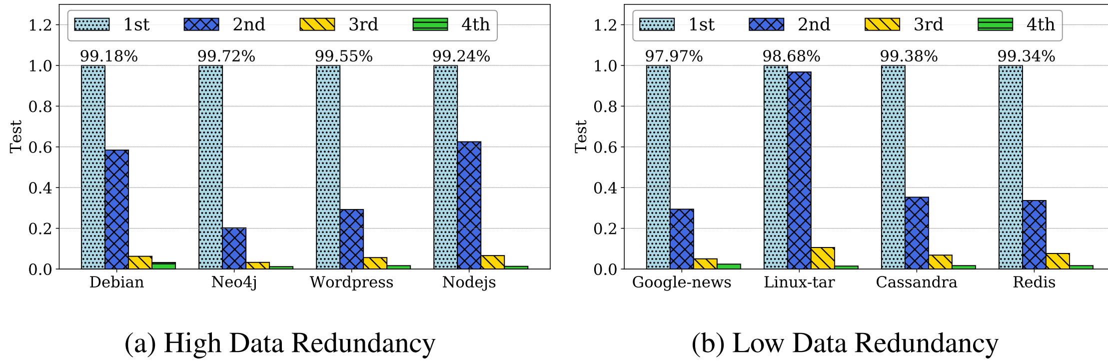
(a) High Data Redundancy (b) Low Data Redundancy

(c) High Data Redundancy (d) Low Data Redundancy

Figure 3.12: *Chunking speedup and dedup ratio for real-world datasets.*

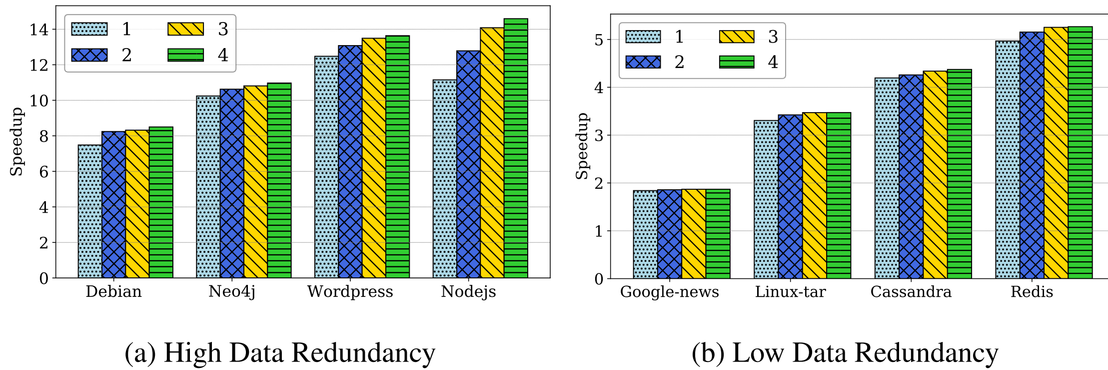
ing and calculation of fingerprints, in Figure 3.15. While these two components often take roughly similar amount of time in the regular CDC, and RapidCDC can remove significant portion of the chunking time, the speedup of the CPU computation is around 2. The speedup is less than 2 for low data-redundancy datasets as RapidCDC's chunking speedups are lower.

To see the impact of RapidCDC's chunking speedups on efficiency of an entire dedup storage system, we present throughput of the dedup system when various real-world datasets are written to the system in Figure 3.16. Disks of different speeds are used in the experiments. In addition to the default hard disk, a SATA SSD and NVMe SSD are used. The SATA SSD is Samsung SSD 860 EVO of 500GB and has write and read bandwidths of 520MB/s and 550MB/s, respectively. The NMVe SSD is Intel SSDPEDMW012T4 of 1.2TB and has write and read bandwidths of 1.2TB/s and 2.4TB/s, respectively. The RapidCDC uses FF as its chunk boundary acceptance criterion. In addition to RapidCDC, it also shows the results of a fixed size chunking (FSC) dedup system with the 8KB chunk size,



(a) High Data Redundancy (b) Low Data Redundancy

Figure 3.13: *Hit ratios of positions in the size lists. The first position's ratio is marked above its bar, which is not scale.*

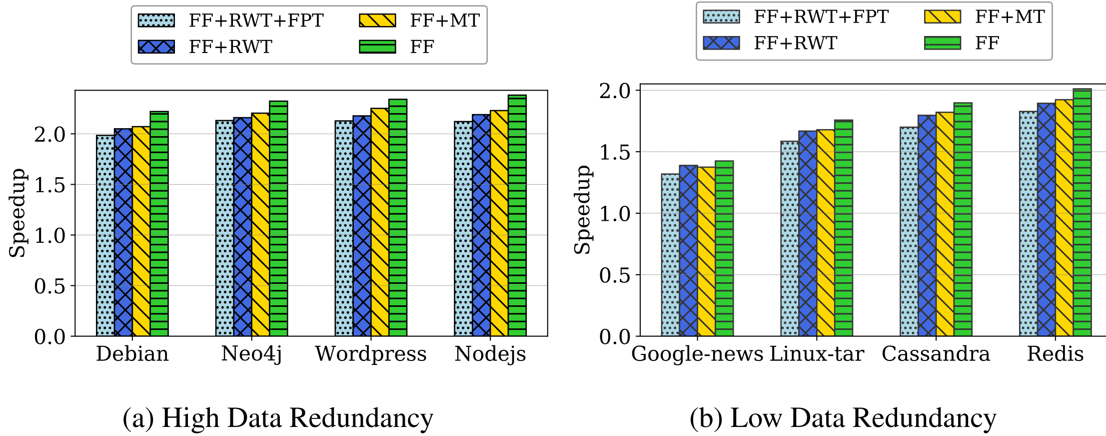


(a) High Data Redundancy (b) Low Data Redundancy

Figure 3.14: *Chunking speedups for real-world datasets with RapidCDC/FF+RWT of different lengths of the size list.*

those of the regular CDC, and a hypothetical dedup system. The hypothetical system represents an ideal CDC dedup, in which chunking time is completely removed. In this case, we obtain chunking boundaries offline.

There are several interesting observations in the figure. First, the disk can be a performance bottleneck when it has a low bandwidth and the dataset has a low dedup ratio that increases I/O demand. Figure 3.16(d) shows the datasets' dedup ratios. For FSC and RapidCDC, when their dedup ratios are low (e.g., for Google-news), using faster disks (SATA SSD and then NVMe SSD) increases the throughput. However, with a higher dedup ratio, the bottleneck shifts to the CPU, and faster disks do not lead to higher throughput. For example, RapidCDC has a dedup ratio of 7.2 on Redis and its throughput increases minimally. However, FSC's dedup ratio is 1.9, and its throughput keeps increasing (from 360MB/s, 745MB/s, to 760MB/s). Second, with a higher dedup ratio, RapidCDC's advantage over regular CDC becomes increasingly higher, because regular CDC's chunking speed is not correlated with the ratio. Its throughput is about 2.3X as high as that of regular CDC with Nodejs. Third, with low dedup ratio and fast disks, such as Google-news or



(a) High Data Redundancy (b) Low Data Redundancy

Figure 3.15: *Speedups of CPU computation (Chunking+fingerprinting) for real-world datasets.*

Linux-tar on NVMe SSD, FSC can have a higher throughput than RapidCDC as the bottleneck is on the CPU and RapidCDC cannot make the chunking fast enough to catch up with the speed of FSC. However, once RapidCDC can have a high dedup ratio, its chunking speed can be close to that of FSC. With its dedup ratio is usually much higher than that of FSC, its throughput is higher than FSC's in most cases. Fourth, once the dedup ratio is high or a fast disk is used, RapidCDC's throughput is close to that of Ideal as the bottleneck is on the CPU and RapidCD removes most of the chunking cost.

3.4.5.0.2 Throughput of Multi-threaded RapidCDC We implement a multi-threaded RapidCDC, each thread working on a different file in a dataset. Being challenged by high cost of CDC's chunking, researchers have proposed methods to use multi-core or even GPU to accelerate the operation [2, 31, 5, 100, 98]. RapidCDC can also leverage the parallelism in a multi-core system. To protect integrity of fingerprints and their size lists, we place the data items in a hash table and apply a lock on each hash bucket. For each dataset, an available thread picks up the next unprocessed file in the set. Figure 3.17 shows the chunking throughput (with disk I/O and fingerprinting operations excluded) of RapidCDC with FF and different thread counts. As shown, the system is well scaled and the throughput increases almost linearly. For high-data-redundancy datasets, the chunking throughput can reach tens of GB/s, making the chunking hardly be a performance bottleneck and making it be a fully addressed issue.

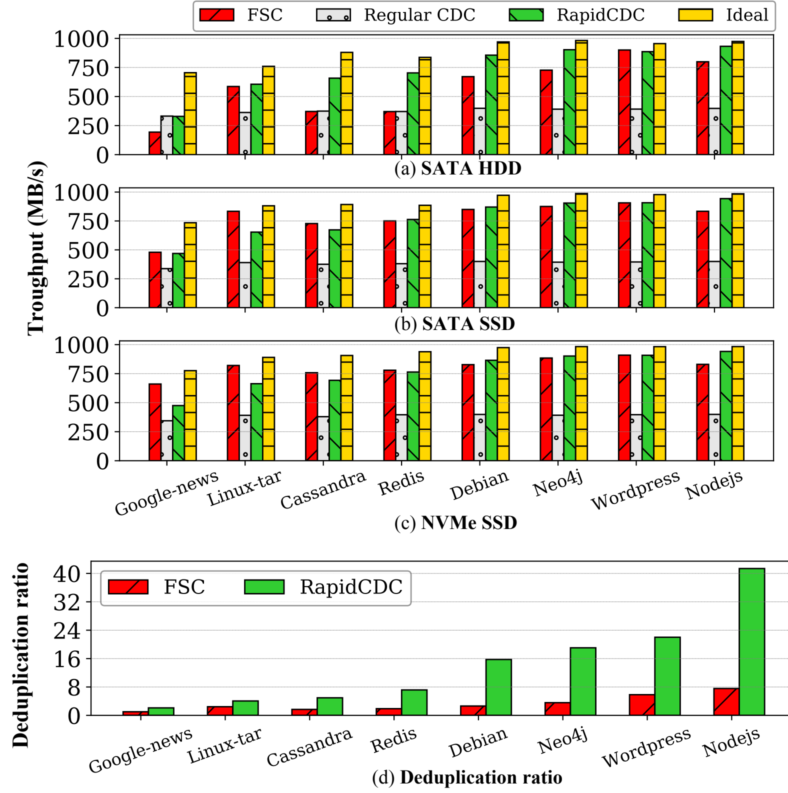


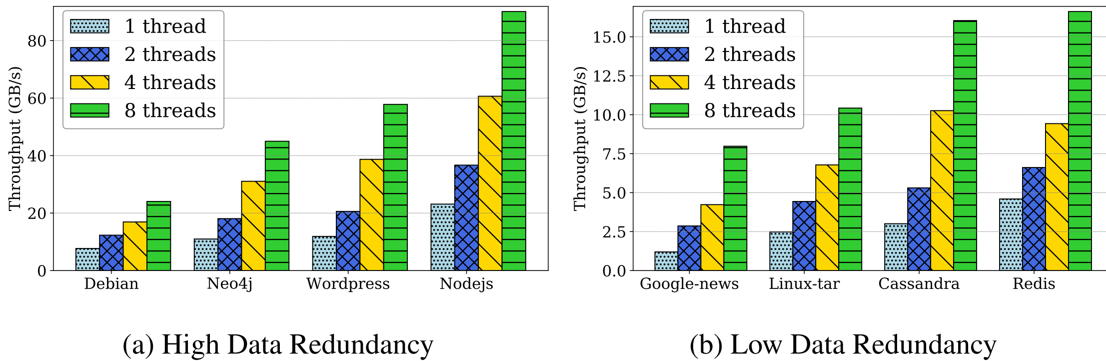
Figure 3.16: *Throughput and dedup ratio of the dedup systems.*

3.5 Related Work

Efforts on improving chunking performance in CDC-based deduplication have been made mainly on optimization of the rolling hash function and leveraging of parallel hardware to parallelize chunking operations.

3.5.1 Reducing Computation Cost in Chunking

In the CDC, a hash value is computed over the content of a rolling window at most of byte positions in a file, representing the major cost of the chunking operation. The Rabin fingerprint, or similarly CRC32, are commonly used as the hash function for determining chunk boundaries reported in literature [57, 106, 47, 64, 100, 56] and for production systems [22]. While chunking becomes a performance bottleneck of a CDC-based dedup system, many techniques have been proposed to reduce its cost. SimpleByte [1] was designed to provide fast chunking to eliminate fine-grained redundant data transmitted across networks. It uses a rolling window of only one byte to detect boundaries of chunks whose sizes are only 32-64 bytes. However, this approach is not likely to be used in regular stor-



(a) High Data Redundancy (b) Low Data Redundancy

Figure 3.17: *Chunking throughput with different number of threads, each on a different core.*

age systems as its condition to form a chunk boundary is too weak and will produce very small chunks, which significantly increase metadata management overhead. Gear [102] is a lightweight hash function requiring only one bit-shift, one add, and one table lookup in one hash computation. As reported, Gear-based chunking can be $3\times$ as fast as Rabin-based chunking [103]. Some issues with the Gear function, such as small rolling window size, are addressed in the FastCDC dedup design [103]. Instead of applying a hash function on a window of bytes to obtain a hash value, MAXP [7] and AE [105] treat bytes in a window as numerical values and find a local extremum to determine a chunk boundary more efficiently. Yu et al. use two functions, a lightweight one and a heavyweight one, to detect a chunk boundary [104]. A lightweight function is applied first to check if a condition is met. Only when it is not is the second function executed. All the efforts have been made to reduce the computational cost at individual window positions, instead of reducing number of file positions where the function has to be applied. RapidCDC takes a radically different approach. It minimizes number of file positions for detecting chunk boundaries. It makes chunking speed less sensitive to the cost of the hash function. And the efforts on reducing the function's computation cost further help with RapidCDC's efficiency.

3.5.2 Accelerating Chunking with Parallelism

StoreGPU [2, 31] and Shredder [5] leverage GPUs to accelerate the compute-intensive chunking and fingerprinting in deduplication. They focus on minimizing the data transfer cost between the host and GPU. P-Dedupe pipelines deduplication tasks and parallelizes chunking and fingerprinting process in each task with multiple threads and achieves high throughput [100]. MUCH is a multi-threaded chunking method, where a file is partitioned into segments for parallel chunking on different cores. It ensures the same set of chunks are generated as that from the sequential CDC chunking [98]. RapidCDC can remove most of

CDC chunking cost, making chunking a lightweight operation and making its parallelization less necessary.

3.5.2.0.1 Exploitation of Locality in Deduplication Locality in workloads has been recognized and exploited to improve various aspects of a dedup system. A sparse indexing structure has been proposed to reduce the in-memory chunk index by taking advantage of the locality of duplicate chunks [50]. The technique has been used in Hewlett-Packard backup products, and effectiveness of the method highly relies on existence of locality of duplicate chunks in segments. ChunkStash also leverages the locality to establish a compact in-memory chunk index and stores the full index on the flash to speedup the index lookup in a dedup system [21]. SiLo organizes a number of small files into a large data stream, so that long duplicate segments, or strong duplicate locality, can occur. This helps to improve dedup ratio and makes large chunks possible, and accordingly reduces amount of index structure [99]. To address the issue of file fragmentation in an FSC dedup system, iDedup [81] exploits the spatial locality in a duplicate data in a primary storage system to deduplicate only chunk sequences of sufficient lengths, so that random disk access can be minimized to efficiently serve read requests. After demonstrating widely available of duplicate locality in various real-world datasets, we propose RapidCDC to leverage the locality to accelerate chunking operations. To the best of our knowledge, this is the first work leveraging the locality to remove the major performance bottleneck of CDC deduplication.

3.6 Conclusion

In the paper we propose RapidCDC, a chunking technique for CDC-based deduplication, that can dramatically reduce chunking time. While chunking has been well recognized as a major performance bottleneck in a CDC dedup system and substantial efforts have been made to reduce its cost, RapidCDC represents a departure from existing optimization techniques for reducing chunking time. It innovatively leverages duplicate locality and access history. Instead of slowly scanning a file to search for chunk boundaries, it uses highly accurate hints on next chunk boundary to reach the boundary with only one verification operation. The RapidCDC technique can be incorporated into any existing CDC dedup systems with minor effort, which is to retrieve a suggested next-chunk size for every matched fingerprint and try to use it for accepting the next chunk boundary. Furthermore, its benefit is orthogonal to other optimizations made in a dedup system, including those for improving chunking performance.

We prototype RapidCDC and conduct extensive experiments comparing its chunking performance and deduplication ratio with other dedup systems. The results show that it can

provide up to 33$\times$ chunking speedup, which essentially removes the chunking performance bottleneck for datasets of high data redundancy. Meanwhile, it maintains the same dedup ratio as regular CDC systems.

CHAPTER 4

WOJ: Enabling *W*rite-*O*nce Full-data *J*ournaling in SSDs by Using Weak-Hashing-based Deduplication

4.1 Introduction

Journaling is a commonly used technique in today's file systems (e.g., ext3 [90] and ext4 [8]) to ensure data consistency in the face of a system failure. In a journaling file system, updates are first recorded in the journal (in the *commit* phase) and later applied to their home locations in the file system (in the *checkpoint* phase). In case of a system crash, all completed records (transactions) in the journal are replayed and applied, while uncompleted ones are discarded, to ensure the file system's consistency.

File system journaling can be in one of the two modes, *data* or *metadata* journaling, based on the contents written to the journal. In the data journaling mode, all file system (data and metadata) updates are written to the journal before being checkpointed to the files later on. In contrast, in the metadata journaling mode, only the updated metadata of the file system, such as inode and free block bitmap, are first written to and protected by the journal, while data are written directly to their home locations in the files. Due to the double write of both data and metadata in the data journaling mode, file system users are usually reluctant to use it and resort to metadata journaling for its fast speed. However, they have to tolerate compromised data reliability.

4.1.1 Data Journaling is Necessary

Metadata journaling has several limitations. First, as data are written directly to a file, without explicit synchronization control, updated data blocks can be mixed with un-updated ones in any order even with sequential write pattern if the system crashes during the file's overwriting [76]. Second, metadata journaling cannot guarantee even the consistency of metadata. For example, with the metadata journaling in the *ordered* mode in ext3 and ext4, the modify time ("mtime") of a file may remain unchanged after the file is updated. The anomaly is due to a critical rule, which is to write data before committing metadata to the journal in the ordered mode. This metadata inconsistency can raise an issue with applications relying on the *mtime* attribute to decide their next actions, including GNU make [85, 34] and file integrity checks using signatures [33, 84]. Last but not least, recent

research has revealed that metadata journaling is prone to introduce vulnerabilities to user-level applications as it reorders applications' write operations for performance [66, 67, 68]. To avoid the vulnerabilities, application developers have to be aware of write re-ordering in file systems and eliminate their side effects with extra efforts in their programming, such as inserting extra flushes between file system operations to enforce the right order. However, it is not easy to avoid the vulnerabilities at the application level even for expert programmers. These vulnerabilities are found in widely-used applications [67], including Google's LevelDB [32] and Linus Torvalds's Git [89].

As Linus Torvalds stated "*Filesystem people should aim to make 'badly written' code 'just work'*" [88], use of data journaling adheres to the belief. With data journaling, the file system maintains a total write order and preserves application order for both metadata and data updates, which provides the strongest data consistency support for applications. As shown in existing studies, most crash-consistency vulnerabilities in commonly-used applications can be avoided with the use of data journaling, and the remaining ones have minor consequences and are ready to be masked or fixed [66, 67]. With the clear advantages in providing data reliability and stronger file system consistency support for upper-level applications, application developers will prefer to use data journaling if the performance is not a concern. With the extensive use of SSDs and their ever-improving I/O performance, data journaling's higher demand on write bandwidth is likely to be accommodated, and the once-thought expensive journaling approach may become affordable.

4.1.2 SSD's Endurance is now a Barrier

While SSD can provide much higher write throughput to potentially support data journaling well, its endurance becomes a new barrier to the practical use of data journaling, which doubles write traffic to the disk. For most flash-based SSD devices, each flash memory cell can only be written several thousand times in its lifespan [38, 80, 52, 42]. While high-end SSDs can deliver GB/s-level throughput and TB-level capacity, their lifetime is not improved accordingly [78]. Actually, due to the adoption of MLC (multi-level cells) and TLC (triple-level cells) for larger capacity, SSDs' lifetime is worsening [38, 42]. For example, Intel 750 Series NVMe 400GB SSDs can provide up to 2.2GB/s and 900MB/s throughput for sequential read and write accesses, respectively, while the endurance is rated for up to a maximum of 127TB written (70GB per day) over the course of its 5-year limited warranty [45]. That is, even if the cells of the device are written evenly (an ideal scenario), it cannot surely admit new writes successfully after each flash cell is written about 317 times (127TB/400GB) on average. In other words, if the device keeps admitting writes at its full write speed, the total write time of the device can be only about 37 hours (127TB/900MB

Table 4.1: *Throughput of different hash functions under different CPU frequencies*

Hash functions	MiB/s (3.0GHz CPU)	MiB/s (1.2GHz CPU)
MD5	526.1	215.3
SHA-1	470.1	172.5
SHA-256	204.1	79.2
SHA-512	327.5	130.5
CRC32C	7631.3	3050.7
xxHash	5715.4	2285.2

seconds). While these numbers may be derived from highly conservative estimates on SSD's lifetime, endurance of SSDs is indisputably a major concern. If the write-twice issue could be addressed by efficiently removing the second write (for checkpointing) that follows the corresponding first one to the journal, data journaling would not be a concern to the SSD's endurance. As contents of the two writes are the same, block-level deduplication technique [106] is a potentially viable solution.

4.1.3 Regular Deduplication is too Expensive

Though deduplication can be a promising solution, it is too expensive to be effective for fast SSDs in terms of its computation, space, and synchronization overheads. Existing deduplication techniques rely on fingerprints, which are hash values computed on individual blocks' contents with a collision-resistant hash function, to identify duplicate data. Example collision-resistant hash functions include SHA-1 [27] and MD5 [75]. As write of every block requires computation of its fingerprint, the impact of the computation can be substantial. Table 4.1 shows throughput of computing hash values of 4KB blocks on a Dell server with Xeon E5-2680v3 CPUs (1.2GHz∼3.0GHz with DVFS technique) and 30MB LLC when different hash functions are used. As we can see, the collision-resistant hash functions, such as MD5, SHA-1, SHA-256, and SHA-512, have throughput higher than or comparable to that of hard disks or slow SSDs. These functions are affordable when the slow devices are used. However, the time of fingerprint computation can be larger than the access time of fast SSDs. For example, Intel 750 NVMe SSD has 900MB/s sequential write throughput [45], much higher than that for computing SHA-1 fingerprints (470MB/s). When the fingerprint computation becomes the performance bottleneck on the I/O path, deduplication is unlikely to be applied. Though non-collision-resistant hash functions, such as CRC32C and xxHash[15], have throughput higher by more than ten times (even on less powerful embedded CPUs [70], as illustrated in Table 4.1 showing throughput under lower CPU frequency), their use can compromise correctness. As we will show, deduplication is preferred to be implemented within a disk [10]. High-performance multi-core processors

are less likely to be used inside an SSD. Furthermore, today's high-performance SSDs, such as Intel Optane SSDs [46], have throughput as high as more than 2GB/s and demand any computation on their I/O paths to be very light.

To make matters even worse, regular deduplication techniques consume a large amount of memory for caching its metadata, including fingerprints, block address mappings, and reference counts, for its efficient operations. The space demand can be substantial. For example, assuming a disk of 4TB (1G 4KB blocks) and a fingerprint of 20B (a SHA-1 value) and each physical block is mapped to two logical blocks on average, the space demand for block address mapping and fingerprints can be 28GB (4B*1G*2+20B*1G). While fingerprints usually have weak locality, almost all of them have to be in the memory to achieve an optimal deduplication ratio.

Finally, deduplication requires periodical synchronization of its metadata, in particular, the block address mappings, onto the disk on a timely manner for data reliability and short response time. Additionally, different types of metadata should be persisted onto the disk in a particular order. All these require frequent use of expensive *flush* operations [11]. For deduplication not implemented in the disk, metadata synchronization operations can significantly degrade I/O throughput and offset deduplication's potential performance advantage.

4.1.4 A Lightweight Built-in Solution

In this work we propose a solution that is built in the SSD to transparently support **W**rite-**O**nce data **J**ournaling, named WOJ. While it still uses fingerprints to detect duplicate blocks and leverages deduplication technique to remove the second writes, it addresses all three issues with the regular deduplication technique. First, WOJ can use ultra-lightweight non-collision-resistant hashing to identify second writes of duplicated blocks without compromising correctness. Second, WOJ only needs to maintain a small amount of metadata, which is thousand times smaller than that for regular deduplication and can be fit in the SSD's internal memory. Third, WOJ integrates its block mapping with SSD's FTL and does not require frequent flushes from the host to the device.

Our contributions in the paper are threefold: (1) We investigate potential benefits and challenges of enabling data journaling in SSDs. We address the challenges with a design with little compromise on I/O performance even for high-end SSDs that are sensitive to even small overheads added into their I/O stack. Also, it protects SSDs' durability as well as regular deduplication does; (2) We prototype WOJ as a device mapper target supporting real file systems (ext3 and ext4) and perform I/O on real SSDs, instead of SSD simulators, to demonstrate its practicality and efficacy; (3) We extensively evaluate WOJ with micro-

benchmarks, widely-used file system benchmarks, database workload, and workloads with real-world data. The results show that WOJ removes about half of the writes in data journaling and provides significant performance improvement over full deduplication schemes.

4.2 The Design of WOJ

The design goal of WOJ is to address challenges on minimizing time and space costs that are required by the deduplication technique and are usually too high to fit in the I/O stack of high-performance SSDs. As we have mentioned, deduplication on the host needs to periodically and often frequently flush its metadata to the disk for their persistency and consistency [11]. This excessive overhead is unlikely to be removed on most of today's general-purpose computers as long as the deduplication is performed at the host side. In contrast, WOJ is situated within the SSD to leverage the non-volatile memory that is often found in today's SSDs in the form of DRAM protected by battery or super capacitor. An additional benefit of performing deduplication in the SSD is that its address mapping table overlaps with the SSD's existing mapping table (as part of the SSD's FTL), and additional space and maintenance costs for the table required by the deduplication can be avoided.

However, the resources available for deduplication, including computing power and memory, are highly constrained in the SSD. Accordingly, objectives of WOJ's design include (1) use of ultra-lightweight fingerprints without compromising correctness, and (2) very small demand on metadata space with its size decoupled from the SSD's capacity. WOJ achieves the design goals by taking advantage of a priori knowledge on the source of data duplication (checkpointing data that have been in the journal) and of very limited amount and lifetime of the data in the journal.

4.2.1 SSD with File-system-level Knowledge

To be simple and effective, WOJ performs deduplication at the block level. That is, it identifies and removes writes of duplicate data at the unit of blocks defined by the SSD's interface. WOJ is designed to remove the second writes in a journaling file system in the data journaling mode. It requires that the whole blocks (not only the updated portions) where (data or metadata) modifications take place are recorded in the journal and later written to the file system. This requirement is met by any file systems using physical journal [69, 94] (including ext3 and ext4), in which file system updates are logged in their original blocks of data and written to the journal. We are aware that file systems with logical journal (like XFS [95]) only record the deltas (the changes) made by the writes in the journal to reduce the amount of logged data. This is at the expense of increased

complexity. While WOJ can remove all the cost of second writes and enable a physical journal that is more efficient than logical journal, we believe that the simpler physical journal will become the preferred journaling approach and take the place of logical journal.

A unique feature of WOJ, as a block-level deduplication design, is to leverage knowledge that is only available at the upper level, i.e., file system level, to distinguish whether a block is written to the journal or not. With this knowledge WOJ can use non-collision-resistant fingerprints and maintain a very small set of metadata for lightweight deduplication without compromising correctness (details in the next subsection). As WOJ is implemented in the SSD, a journaling file system must inform WOJ that data journaling and WOJ functionality be enabled when the file system is mounted. In a journaling file system, such as ext3, ext4, and ReiserFS, a special file with contiguous space allocation at a fixed disk address and of fixed size is designated as the file system's journal to store records (transactions), which are then periodically checkpointed into the file system. The journal is used as a circular buffer, where space holding records that have been checkpointed can be reused. As long as the journal's address space, possibly in the form of its starting disk address and length, is disclosed by the file system to the disk, WOJ knows which of the writes to the disk are commit ones (*the first writes*) to the journal and which are checkpoint ones (*the second writes*) to the home locations. When there are more than one partition on the SSD, a file system needs to specify the address space the partition occupies, and WOJ distinguishes the two types of writes and performs its deduplication operations separately for individual WOJ-enabled partitions. WOJ can be enabled at the time of mounting a file system (with a *mount* system call). The journal's address space can be passed to the SSD via an SSD primitive similar to the *ptrim* and *exists* commands proposed by FusionIO [59]. When the WOJ functionality is not enabled or is turned off after being enabled, the SSD functions as a normal SSD device without deduplication.

4.2.2 Deduplication with Non-collision-resistant Fingerprints

WOJ does not pursue full deduplication, where all duplicate blocks are detected and only one copy of the duplicate blocks is physically stored in the disk. Instead, WOJ uses non-collision-resistant fingerprints to deduplicate the second writes (writes in the checkpoint phase) in a data-journaling file system with very low overheads. In the process it is required that (1) the correctness is not compromised; and (2) the collision rate is low so that deduplication ratio is negligibly affected. The key difference in its use of fingerprints from regular deduplication is that WOJ does not rely on fingerprints to determine the existence of duplication between a given block in the checkpoint phase and those that are recently committed to the journal and have not been checkpointed yet, as in the data jour-

naling mode *"all new data is written to the journal first, and then to its final location"* [48] and the duplication is guaranteed. Instead, a block's fingerprint is used only to identify which block in the journal has identical contents as the block under consideration in the checkpoint phase. As long as a fingerprint is not shared by more than one block in the journal, it can be used to identify the corresponding block in the checkpoint phase (even if non-collision-resistant fingerprints are used) and avoid writing it to the disk.

To facilitate the identification, WOJ maintains a fingerprint pool. A fingerprint will be inserted in the pool and used for detecting duplicate blocks in the checkpoint phase only when two conditions are satisfied. First, the fingerprint is computed over the contents of a block in the commit phase. Second, the fingerprint is unique in the pool. For each fingerprint in the pool, it is associated with a unique physical page address (PPA) indicating where the corresponding block of data is stored. We assume the block size exposed by the SSD's interface is identical to the flash memory page size inside SSD. If not, an adaptation is straightforward [6]. To this end, when a block is written into the journal (in the commit phase), its fingerprint is computed (Step 1.1 in Figure 4.1) and used as the key to search in the pool. If it is not found, the fingerprint is added into the pool (Step 1.3 in Figure 4.1). Otherwise, a collision occurs (Step 1.2 in Figure 4.1). A straightforward solution is to mark the fingerprint as invalid and abort the deduplication attempt on the blocks of this fingerprint. Note that when a fingerprint is designated as invalid, it is not immediately removed from the pool. Otherwise it can cause correctness issue as there can be further collisions on the fingerprint. An alternative is to introduce secondary fingerprints for the blocks in collision and enable a second chance after a collision. We do not take this option in the design as the probability of the collision is small. As disclosed in a recent study [4], the collision rate of CRC32 (and CRC32C) for typical storage workloads is lower than $8X10^{-5}$, and for *xxh64* (64-bit version of xxhash) 4 billion hashes have a 50% chance of getting one collision [13]. Additionally, the fingerprint pool is small, whose size is capped by number of blocks in a journal, whose size is usually configured as a few hundreds of megabytes to several gigabytes. Further, WOJ removes a fingerprint from the pool right after it is used for a successful removal of a second write. Otherwise, it will be removed when the space for its corresponding block is reclaimed in the journal. In either case the lifetime of a fingerprint in the pool is short and the collision is expected to be rare. It is noted that, no blocks in the commit phase will be deduplicated regardless of the uniqueness of their fingerprints.

For a block in the second write (in the checkpoint phase), its fingerprint is computed and searched in the fingerprint pool (Step 2.1 in Figure 4.1) for a match. Note that there always exists such a match. If it matches a valid fingerprint (one that has not experienced any collision), the block is deduplicated (Step 2.2 in Figure 4.1). Instead of actually writing

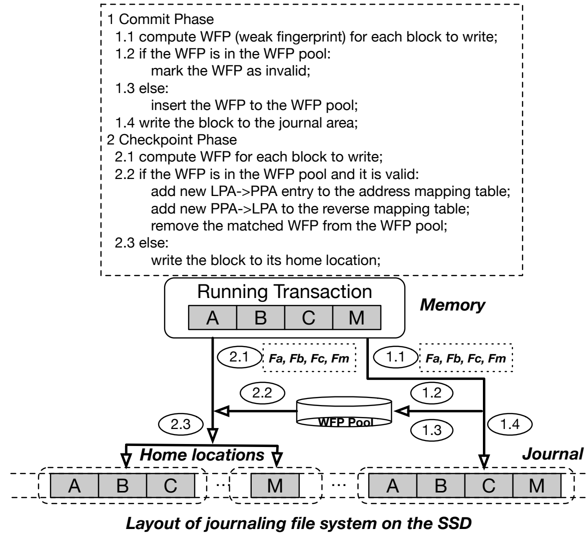


Figure 4.1: *Sequence of operations in WOJ for committing and checkpointing blocks. In the figure,* F_a*,* F_b*,* F_c*, and* F_m *refer to weak fingerprints of blocks* A*,* B*,* C*, and* M*, respectively.*

the data again to the disk, only an entry in the FTL's address mapping table (from a block's logical page address (LPA) to its logical page address (PPA)) is updated to reflect that the block's LPA is mapped to the PPA of the block with the matching fingerprint. Note that WOJ does not need to maintain a separate address mapping table. If the block matches an invalid fingerprint (Step 2.3 in Figure 4.1), it is written to the disk as usual, instead of being deduplicated.

4.2.3 Metadata Supporting Movements of Physical Blocks

As mentioned above, to serve writes and reads from users WOJ only needs to use FTL's address mapping table and maintains a very small fingerprint pool (tens of KB for a journal of a few hundreds of MB) due to the small number of much shorter fingerprints (8B for xxhash vs. 20B for SHA-1). The pool can be easily held in the SSD's memory.

However, WOJ needs to maintain a reverse address mapping table (from PPA to LPA) to support SSD's internal operations such as garbage collection and static wear leveling. When a physical block at a PPA is migrated during the operations, one has to know the LPA(s) that are mapped to the PPA, so that the address mapping table ($LPA \rightarrow PPA$) can be properly updated. When a block in the checkpoint phase is deduplicated, and its logical page address (LPA_2) is mapped to the PPA of a matching block in the journal, whose logical page address is (LPA_1), the PPA is mapped to two LPAs ($PPA \rightarrow LPA_1$ and $PPA \rightarrow LPA_2$). The first one is automatically recorded in the physical page's OOB (out-

of-band) area when the page is written. The second one is recorded into a reverse mapping table. Note that a PPA is mapped to no more than two LPAs in WOJ, and a reference count maintained for each PPA in the regular deduplication is not necessary. To reclaim a physical block, both block's OOB area and the reverse table are checked to ensure no LPAs are still mapped to its PPA .

Most of the reverse mapping table will be stored on the flash. The organization of the table can be similar to the directory-based one for the address mapping table in DFTL [36]. As updates of the table have strong spatial locality due to usually sequential writes to PPAs in the journal, they can be done efficiently in batches. Though lookups into the table can be expensive by involving flash reads, they may not be on the critical path of servicing user requests. More importantly, they are often followed by erase operations, which can be hundreds of times more expensive than the lookups and make their impact negligible.

4.3 Evaluation

We extensively evaluate WOJ with a variety of workloads on a WOJ prototype to reveal its performance insights. In particular, we will answer the following questions: 1) can WOJ retain data journaling's performance on fast SSDs, and to what extent can WOJ reduce the performance overheads introduced by deduplication? 2) in terms of reducing writes to the disks, can WOJ address the write-twice issue in data journaling by removing the duplicate writes to the disk?

4.3.1 Experiment Methodology

As an SSD with built-in WOJ is not available yet, we prototyped a virtual SSD with WOJ functionality enabled. In the virtual SSD, the WOJ functionality is implemented in Dmdedup [82], an open-source deduplication framework, as a device mapper target at the generic operating system block device layer in Linux kernel 4.12.4. All the data written to the virtual disk are first processed in the target and those to be persisted are directed to a real SSD. Block size of the device is set to be the file system's default page size, which is 4KB. Since WOJ is to be implemented inside SSDs, where the metadata can be cached in the NVM space (e.g., battery-backed RAM) to avoid frequent data persistence, we choose the *INRAM* backend of Dmdedup to manage the deduplication metadata. With the INRAM backend, the deduplication metadata is only stored in DRAM, rather than writing to the disk. In the prototype, WOJ can be enabled/disabled by using the "*dmsetup message*" command. An the ext3 file system is installed on the virtual SSD, and data journaling mode (*data = journal*) is enabled with a 256MB journal. In the prototype we implemented the

Table 4.2: *SSDs used in the experiments*

Type	Fast SSD	Moderate SSD	Slow SSD
Disk Size	400GB	240GB	80GB
Model Family	Intel 750 Series [45]	Intel 520 Series [44]	Intel X18-M/X25-M [43]
Device Model	SSDPEDMW400G4	SSDSC2CW240A3	SSDSA2M080G2GC
Sequential Read	2200MB/s	550MB/s	250MB/s
Sequential Write	900MB/s	520MB/s	70MB/s
Interface	PCI-Express 3.0 X4	SATA 3.0, 6.0GB/s	SATA 2.6, 3.0Gb/s
Endurance Rating	70GB/day for 5 years	20GB/day for 5 years	100GB/day for 5 years

deduplication operations as well as their supporting data structures including an address mapping table, a reverse mapping table, and a fingerprint pool. It is noted that, using Dmdedup framework adds additional overheads (compared to an in-SSD implementations) to WOJ and its counterparts, which are described in the below. Because WOJ itself is more lightweight than its counterparts, the overhead represents a larger percentage of its service time, and accordingly WOJ's reported relative performance advantages are conservative.

The experiments are conducted on a Dell R630 server with two Xeon E5-2680v3 2.50GHz CPUs, each with twelve cores and 30MB last-level cache. The server is equipped with 128GB DDR4 memory. Three SSDs with different performance (as listed in Table 4.2) are used in the experiments to create different evaluation scenarios.

In the evaluation, the file system is configured with one of following three configurations:

- *WOJ_X*: data journaling mode with WOJ enabled. "*X*" indicates the hash function for fingerprinting. In the experiments, we use *xxHash* [15], a fast non-cryptographic hash algorithm, to generate weak fingerprints as a representative of non-collision-resistant hash functions[1]. Specifically, we applied the xxHash function over the first 64 bytes of a 4KB block to generate a 64-bit fingerprint for the block. Accordingly this WOJ is named *WOJ_xxh64*. In addition, to show the impact of hash functions, we also use collision-resistant hash function, namely, SHA-1 and MD5, in WOJ (*WOJ_sha1* and *WOJ_md5*, respectively.)
- *No_Dedup_DJ*: data journaling mode without using any deduplication. For a fair comparison, the I/O requests are also directed to the Dmdedup framework, and then sent to the device without deduplication.

[1]Using crc32c for fingerprinting can provide even better performance (see Table 4.1) and reduce space overhead, but its high speed depends on special Intel instructions that are not available in most embedded processors.

- *Dmdedup_sha1*: data journaling mode with *Dmdedup* enabled for full deduplication using SHA-1 values as fingerprints, where any blocks with identical fingerprints are detected and deduplicated.

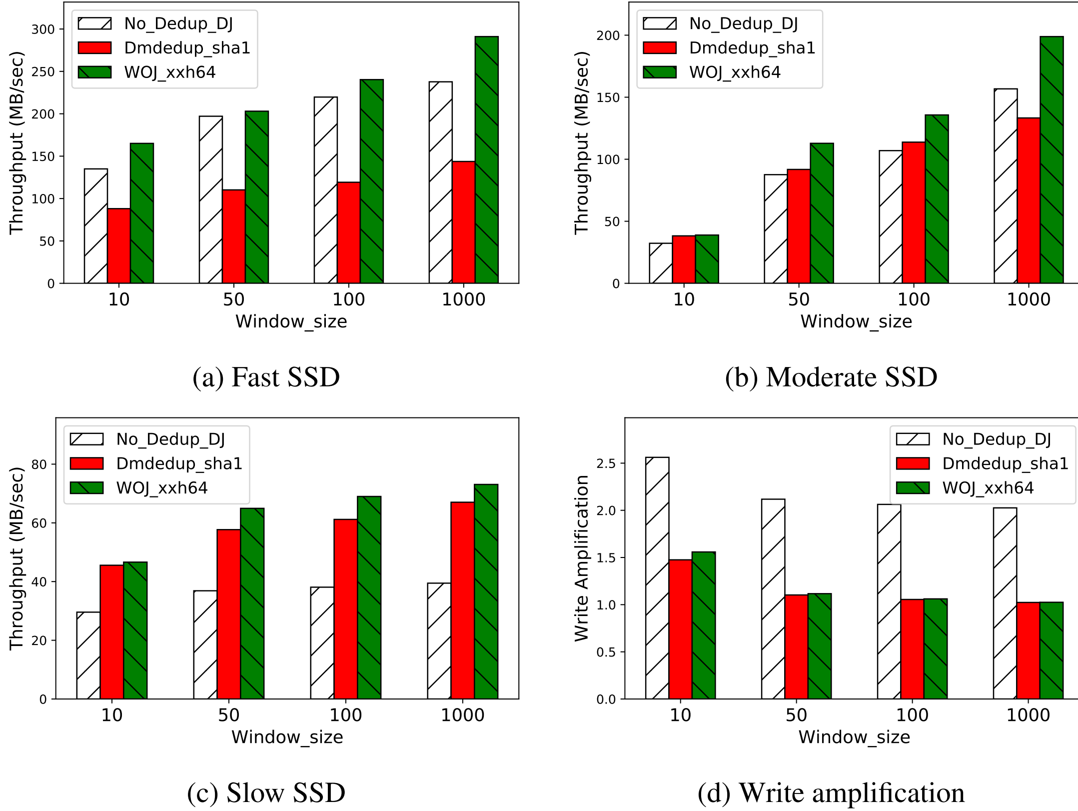
We use four types of workloads to evaluate WOJ:

- Running write-only micro benchmarks. We continuously write 4KB data to a file. Both sequential and random writes are tested.
- Running Filebench benchmarks. We conduct experiments on write-intensive benchmarks in Filebench [55], a widely-used file system and storage benchmark suite, which includes common file operations, such as create, open, close, delete, read, and write. We fill the data blocks in the write operations with randomly-generated contents.
- Hosting database workloads. In the above two workloads, each user file access had been aligned to 4KB blocks before being sent to the disk. In the database workloads, we choose a popular KV store (LevelDB [32]) and run the off-the-shelf benchmark (*db_bench*) on it. In the tests, users' data are usually re-organized before they are written to the disk, which may have implication on WOJ's deduplication ratio.
- Serving workloads with real-world data. In this experiment we conduct some common file system operations (file creations, reads, and writes) on real-world data.

4.3.2 Results with Write-only Micro Benchmarks

In these workloads write requests of 4KB data are issued continuously to a file, and a flush operation (issued with the *fsync* system call) is issued after a given number of writes for data persistency. We name this number the *flush window size*. Using a smaller flush window reduces chance of losing data during a system crash but may suffer a higher performance penalty.

We first perform sequential writes to a new file until it grows to 8GB. Because of use of journaling and file system operations, the actual amount of data written to the disk is larger than the amount of data requested by user programs for writing. The ratio between these two amounts is named *write amplification*, or *WA*. Figure 4.2 shows the throughput and WA of the sequential writes with different flush window sizes on different SSDs. The results reveal a number of insights. First, WOJ_xxh64 achieves the highest throughput among the three configurations as shown in Figures 4.2a, 4.2b and 4.2c. Second, throughput difference among the three configurations varies on different SSDs. For the fast SSD, Dmdedup_sha1 shows much lower throughput than No_Dedup_DJ. For example, its throughput is only about 60% of No_Dedup_DJ's throughput when the window size is 1000. Although applying deduplication removes almost half of the writes to the disk in data journaling as



(a) Fast SSD

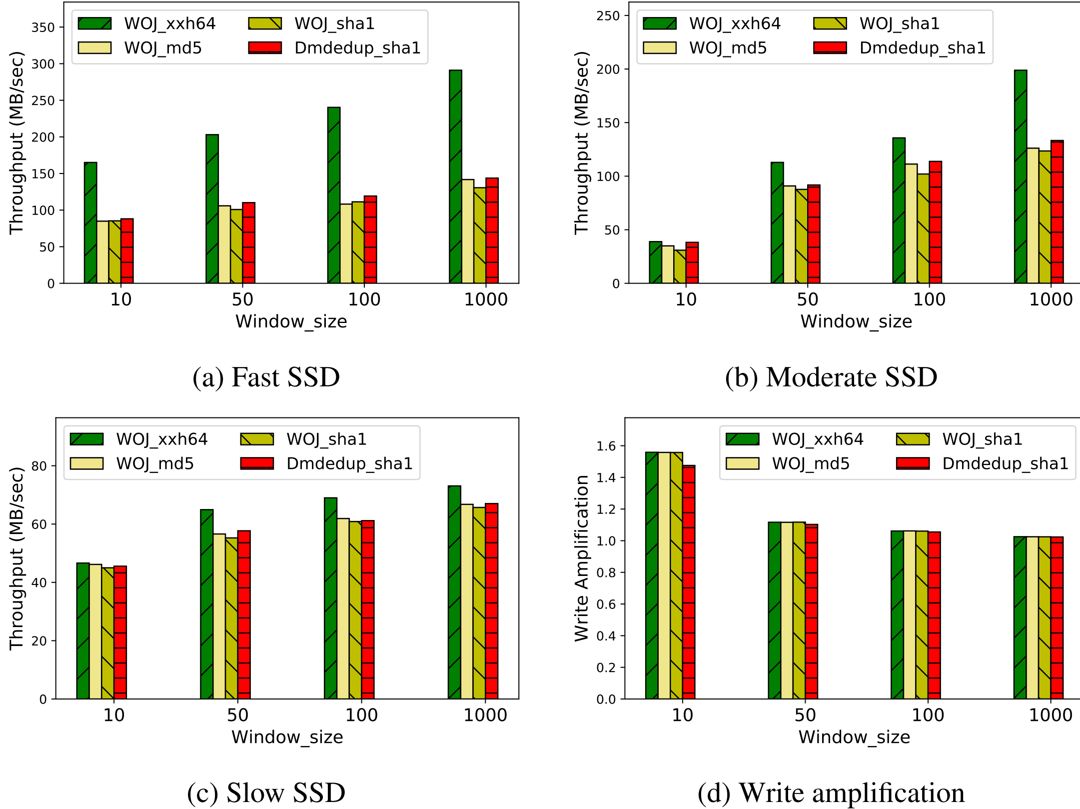
(b) Moderate SSD

(c) Slow SSD

(d) Write amplification

Figure 4.2: *Throughput and write amplification for sequential write accesses with different window sizes on the three SSDs.*

shown in Figure 4.2d, it introduces fingerprinting overheads, which is too expensive for the fast SSD as indicated in Tables 4.1 and 4.2. These results confirm our belief that regular deduplication schemes are too expensive for high-performance SSDs. For moderate and slow SSDs, since their I/O performance is much lower (as shown in Table 4.2), their overhead of fingerprinting becomes less significant, making Dmdedup_sha1 provide comparable (as shown in Figure 4.2b) or higher (as shown in Figure 4.2c) throughput than that with No_Dedup_DJ. Third, there is an anomaly in the performance difference between No_Dedup_DJ and Dmdedup_sha1 in Figure 4.2b when the window size is 1000. Although the throughput of fingerprinting with SHA-1 (470MB/s) is a little lower than the write throughput of the moderate SSD (550MB/s) as shown in Tables 4.1 and 4.2, Dmdedup_sha1 achieves higher throughput when the window size is small as the frequently issued flush operations add extra cost to the I/O operations. However, when the window size becomes larger, No_Dedup_DJ has higher performance than Dmdedup_sha1. This is because the overhead of the expensive flush operations is amortized by more write opera-



(a) Fast SSD

(b) Moderate SSD

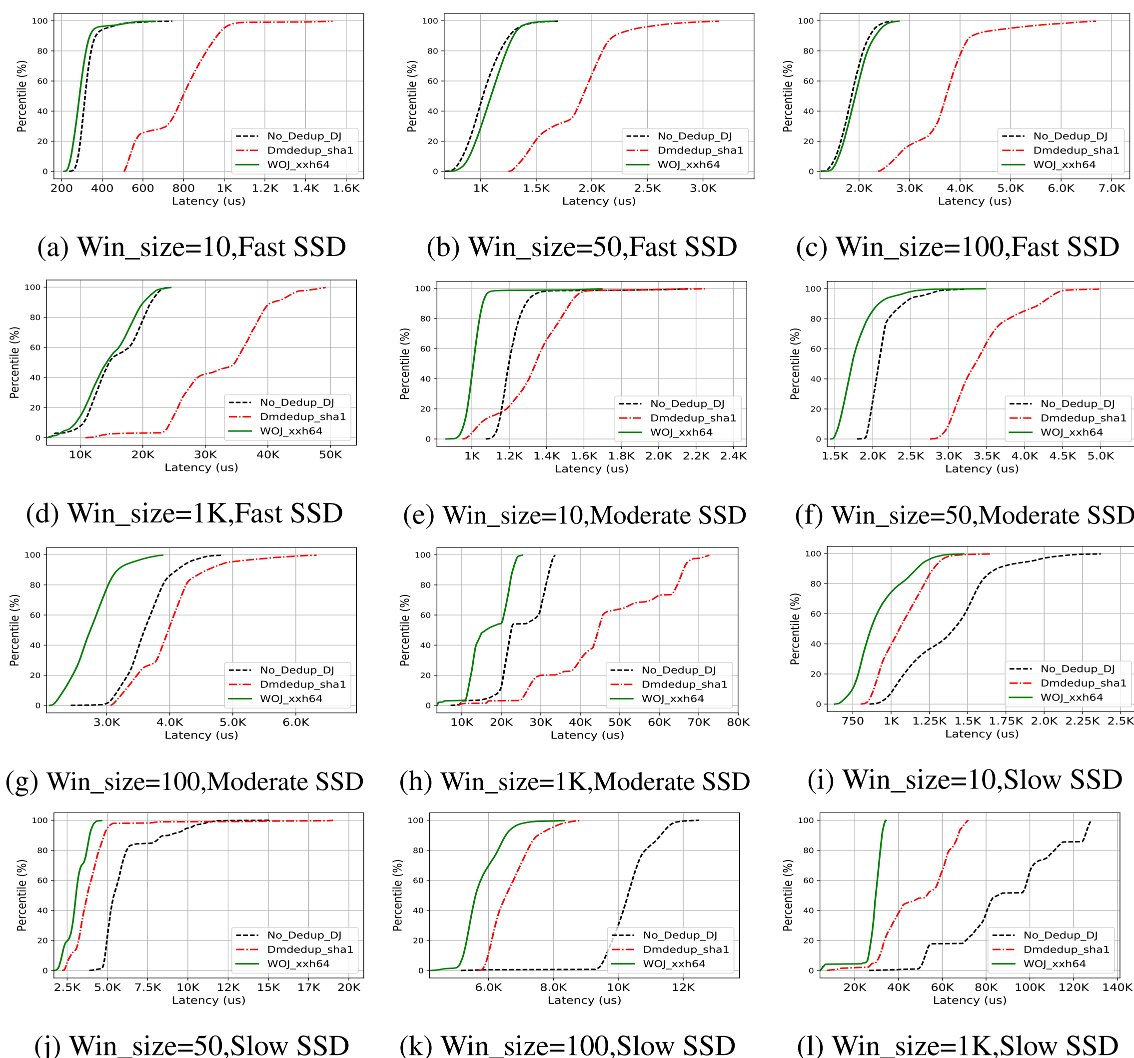
(c) Slow SSD

(d) Write amplification

Figure 4.3: *Throughput and write amplification of WOJ for sequential write accesses with different hash functions for different window sizes.*

tions, which improves the I/O performance and makes the overhead of fingerprinting more significant. In general, WOJ improves the throughput by about 1.85X to 2.03X compared to Dmdedup_sha1 for the the fast SSD.

As shown in Figure 4.2d, WOJ_xxh64 removes about half of the data written to the disk as Dmdedup_sha1 does. The results show that hash collisions due to the use of weak fingerprint are rare in WOJ. We also see that WA is lower when the window size is larger. For example, WA for No_Dedup_DJ is about 2 when the window size is 1000, while it increases to 2.6 when the window becomes 10. There are two reasons for this. First, with a larger window size file system metadata updates to the same metadata blocks in the same window can be merged and thus fewer metadata blocks are written to the disk. Second, with larger window size flush operations are less frequently issued, the file system can fit more file system updates into a single transaction, which will improve file system efficiency and reduce the journal metadata blocks (e.g., descriptor blocks and commit blocks).



(a) Win_size=10,Fast SSD (b) Win_size=50,Fast SSD (c) Win_size=100,Fast SSD

(d) Win_size=1K,Fast SSD (e) Win_size=10,Moderate SSD (f) Win_size=50,Moderate SSD

(g) Win_size=100,Moderate SSD (h) Win_size=1K,Moderate SSD (i) Win_size=10,Slow SSD

(j) Win_size=50,Slow SSD (k) Win_size=100,Slow SSD (l) Win_size=1K,Slow SSD

Figure 4.4: *Sequential write latency with different window sizes on the three SSDs.*

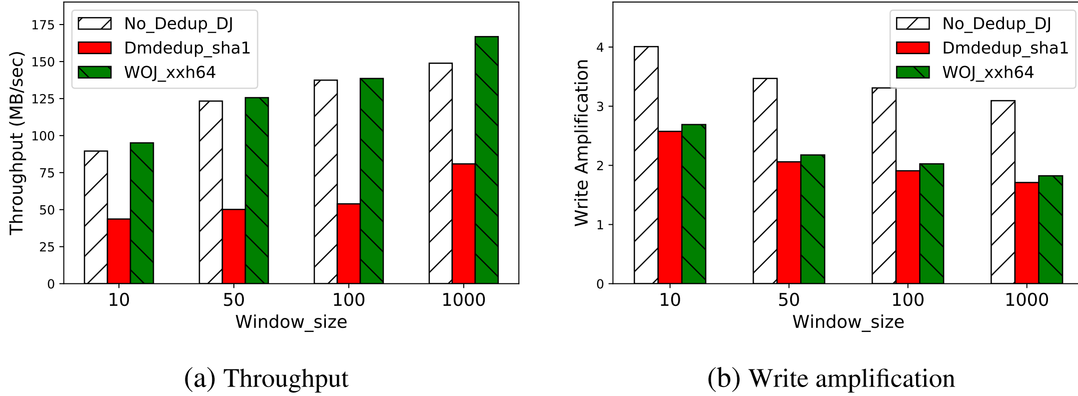
To understand the contribution of the use of a weak hash function in WOJ to its performance improvement, we replace the hash function with collision-resistant ones for generating fingerprints in WOJ. The results for WOJ_xxh64, WOJ_md5, and WOJ_sha1 are shown in Figure 4.3. There are several observations. First, WOJ_xxh64 provides much better performance than the other two, as *xxhash* is much more lightweight than the other two as shown in Table 4.1. Second, the throughput difference between the three is more significant on the fast SSD, where a larger proportion of run time is spent on computing fingerprints. Third, WOJ_sha1 has a little lower performance than Dmdedup_sha1. This is because WOJ_sha1 detects and removes duplicates only in the checkpoint phase, while Dmdedup_sha1 achieves full deduplication, which may reduce a little more writes to the

disk as shown in Figure 4.3d. Last, comparing Figures 4.3a and 4.3b, we can find that with strong hash functions, the throughput of WOJ on the fast SSD and moderate SSD does not show significant difference for large window size even though the raw performance of the SSDs is quite different. The reason is that the computation of strong fingerprints becomes performance bottleneck for fast SSDs and offsets the performance advantage of the fast SSD. In general, with faster hash functions, such as xxHash, WOJ provides up to 2.24X throughput improvement over that using slow hash functions such as SHA-1.

For file system users, I/O request latency is also a critical performance metric. Figure 4.4 shows the CDF curves of write latencies of the three schemes on the three SSDs with different window sizes. The latency increases when the window grows for all the three schemes as a request is acknowledged only when the flush operation at the end of the window completes, which ensures all data blocks in the window are persisted on the flash. On the fast SSD, the latency with WOJ_xxh64 is similar to that with No_Dedup_DJ, and much lower than that with Dmdeup_sha1 for all window sizes. For the fast SSD with high bandwidth, the doubled amount of writes poses a modest impact on the write latency for No_Dedup_DJ. For WOJ_xxh64, although half of the writes are removed, it has write latency similar to that of No_Dedup_DJ. On the moderate SSD, the benefit of reducing half of the write traffic outweighs the overhead of computing weak fingerprints. Accordingly, WOJ_xxh64 provides lower latency than No_Dedup_DJ. In contrast, computing strong fingerprint is still too expensive. Thus, the latency of Dmdedup_sha1 is the highest. On the slow SSD with low bandwidth, the doubled write traffic in No_Dedup_DJ poses a significant overhead, causing No_Dedup_DJ to have the highest latency. For the fast SSD, the 90th percentile latency of *WOJ_xxh64* can be reduced to around 50% of that with *Dmdedup_sha1*.

To understand impact of different access patterns, we also conduct experiments with 4KB writes at random locations in a 8GB file. The experiments demonstrate similar performance trends, though random access has lower performance than sequential access. Figure 4.5 shows experiment results on the fast SSD. Compared to Figure 4.2, we see two significant differences. First, the throughput is much lower than that with sequential access. For example, for a window size of 1000 writes, the throughput of *WOJ_xxh64* is about 165MB/s for random writes, while it is about 290MB/s for sequential writes. Second, the WA is higher for random access, which contributes to its throughput degradation. For random writes, the metadata updated in a flush window are less likely to be merged as they usually scatter in different metadata blocks, which causes more metadata blocks to be written to the disk.

Results with sequential and random writes show that WOJ reduces almost the same amount of duplicated data as that in full deduplication schemes with strong fingerprints.



(a) Throughput (b) Write amplification

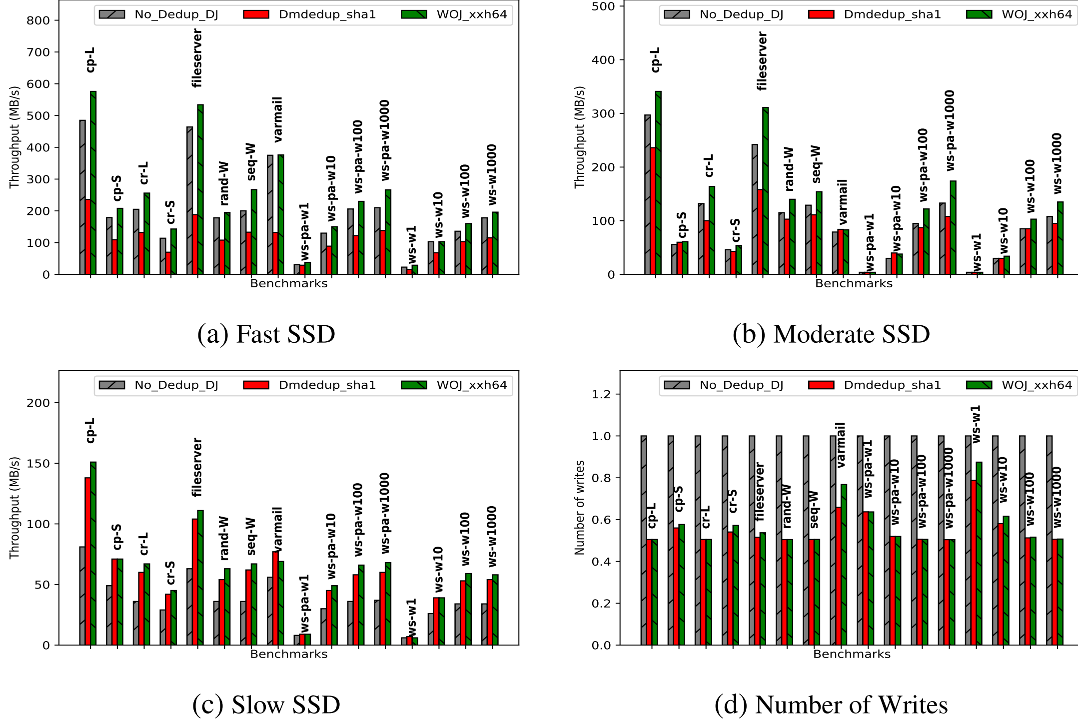
Figure 4.5: *Throughput and write amplification for random write accesses with different window sizes on the fast SSD.*

Meanwhile, WOJ reduces the performance overhead to significantly improve both throughput and latency.

4.3.3 Results with Filebench Benchmarks

The Filebench benchmarks include commonly-used file system operations, such as create, open, read, append, overwrite, close, and delete. Figure 4.6 shows the throughput and amount of data written to the disk for selected benchmarks on the three SSDs. For almost all the benchmarks, WOJ_xxh64 provides the highest performance. On the fast SSD, WOJ_xxh64 achieves much higher throughput than Dmdedup_sha1 for benchmarks with large flush window sizes, such as *cp_L*, *cr_L*, *ws_w1000* and *ws_pa_w1000*. The improvement can be up to 2.44X. For the benchmarks *varmail* and *fileserver*, WOJ_xxh64 achieves high throughput improvement for a different reason. These two benchmarks generate about the same amount of read and write operations and the I/O time dominates their execution time. In such workloads, read and write requests compete for the disk's bandwidth. While WOJ removes about half of the writes and makes the corresponding disk bandwidth available, read throughput also increases. Although Dmdedup_sha1 can remove about the same amount of writes, its high fingerprinting overhead largely offsets this benefit.

In a case with *varmail* shown in Figure 4.6c, the throughput of Dmdedup_sha1 is a little higher than WOJ_xxh64 on the slow SSD. In *varmail* about 10% more data written to the disk are removed by Dmdedup_sha1 than WOJ_xxh64 (shown in Figure 4.6d). WOJ only removes duplicate data in the checkpoint phase, while Dmdedup implements full deduplication. WOJ misses some deduplication opportunities hidden within the journal area and takes more I/O time.



(a) Fast SSD

(b) Moderate SSD

(c) Slow SSD

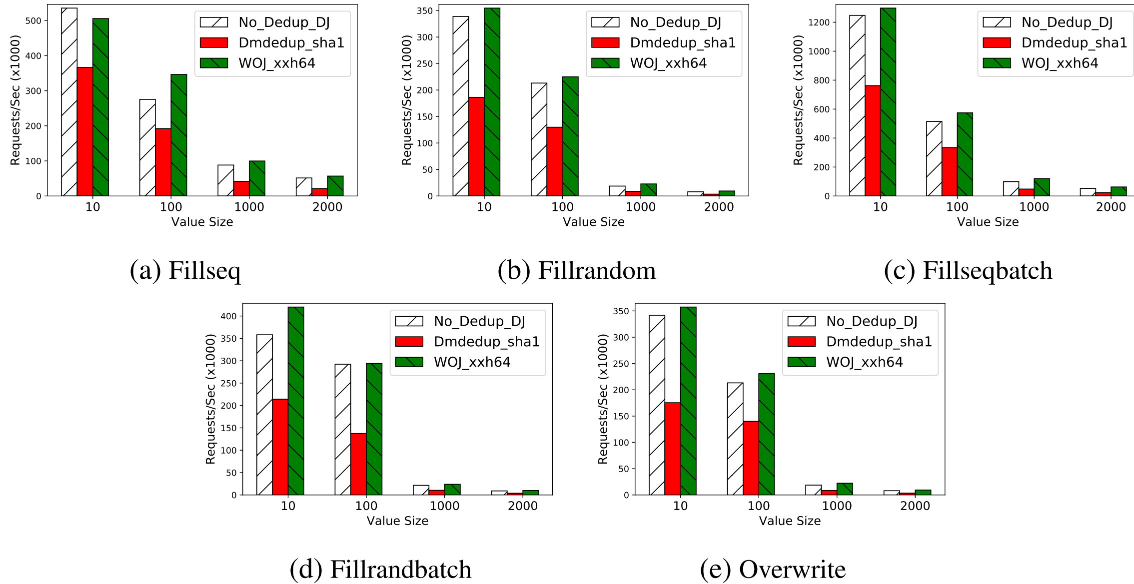
(d) Number of Writes

Figure 4.6: *Throughput and normalized number of writes on the three SSDs with filebench benchmarks. Meaning of the abbreviations: cp=copy; cr=create; L=Large file (1GB); S=Small file (64KB); W=Write; ws=Write with sync; pa=Preallocate file.*

The results with Filebench workloads also show that in most cases WOJ can perform as well as full deduplication schemes in reducing redundant data, as shown in Figure 4.6d. Except for *varmail* and *ws-w1*, where many blocks are journal metadata and less likely to be deduplicated even by full deduplication schemes, WOJ can remove about 38% to 50% blocks written to the disk, which will substantially help to improve the disk's lifetime.

4.3.4 Results with Database Workload

User applications usually rely on the database for data storage and retrieval, and expect it to be highly reliable as a crash in a database can affect all upper-layer applications that use its service. As databases, like LevelDB [32], are usually built on top of file systems and store their data in files, the development of database programs can be much easier if the underlying file systems provide stronger data consistency support and preserve write orders by using data journaling [66]. In this experiment we mount the ext3 file system with the data journaling mode enabled. We then extensively conduct experiments on PUT workloads with different access patterns on LevelDB. We use all the default configurations of LevelDB



(a) Fillseq (b) Fillrandom (c) Fillseqbatch

(d) Fillrandbatch (e) Overwrite

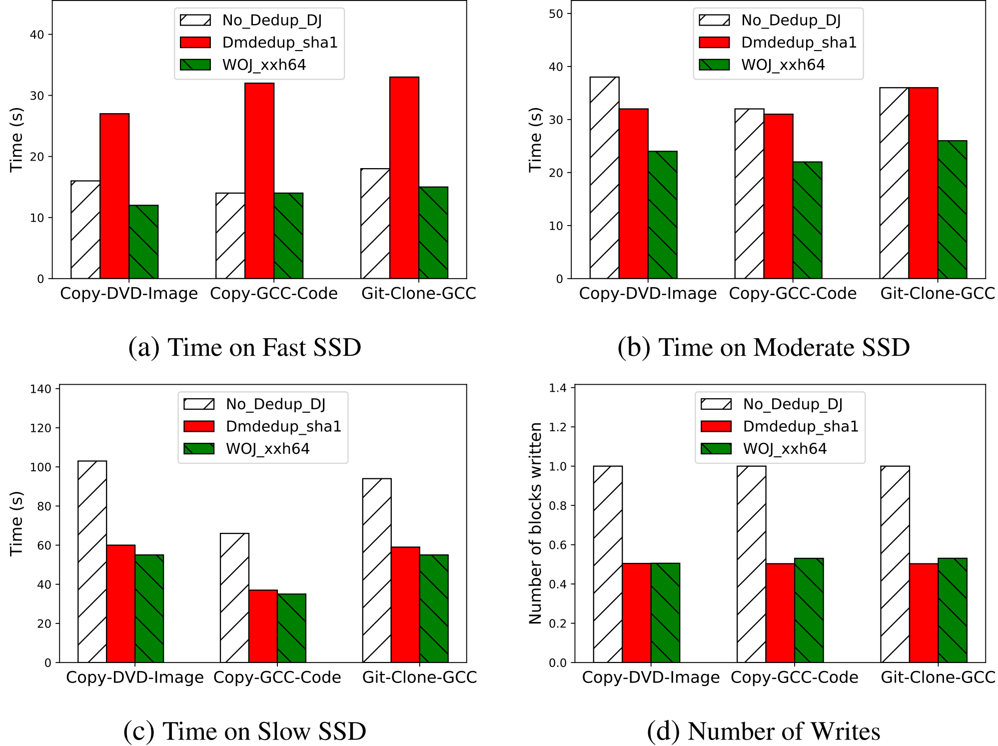
Figure 4.7: *Throughput for PUT operations with different value sizes in LevelDB on fast SSD. The key size is 16 bytes.*

during the experiments. In particular, the SSTable's size is 2MB. We run *db_bench* released with the LevelDB code [12]. Due to space constraint, we only show experiment results on the fast SSD, which poses significant performance challenges to existing deduplication schemes.

Figure 4.7 shows the throughput for LevelDB benchmarks on the fast SSD. For all the benchmarks, WOJ_xxh64 provides better performance than Dmdedup_sha1. The improvement is about 1.38X to 2.7X, and higher improvements are achieved with larger value sizes. Given a fixed number of requests, requests of larger value sizes will generate more blocks for writing and more I/O operations. On the contrary, with smaller value sizes more requests' data (key-value items) can fit in a single block and written to the disk together, leading to reduced I/O operations, and the advantage of using WOJ to reduce write traffic is weakened.

4.3.5 Results with Workloads Using Real-world data

In this section, we evaluate WOJ with workloads with real-world data, rather than randomly generated data, in the I/O operations. In the first experiment (*Copy-DVD-Image*), we copy a Debian DVD image [84] (about 3.7GB) from the tmpfs (/tmp) to the tested file system. In the second experiment (*Copy-GCC-Code*), we copy a compressed gcc source code [65], which includes more than 4000 files and has a size of about 3.1GB, from the



(a) Time on Fast SSD

(b) Time on Moderate SSD

(c) Time on Slow SSD

(d) Number of Writes

Figure 4.8: *Run time and number of writes of workloads with real-world data on the three SSDs. The number of writes is normalized based on the number of blocks written with No_Dedup_DJ.*

tmpfs (/tmp) to the tested file system. In the third experiment (*Git-Clone-GCC*), we use *git clone* to clone a gcc repository to the tested file system. The results are shown in Figure 4.8.

Consistent with observations on other workloads, WOJ shows substantial performance advantages in the three experiments on the three SSDs. Compared to Dmdedup_-sha1, WOJ_xxh64 reduces the execution time by about 55.6%, 56.3%, and 54.5% in the three experiments on the fast SSD, respectively. As expected, the improvements are less significant on the slow SSD, which are about 8.3%, 5.4%, and 9.3%, respectively. As shown in Figure 4.8d, WOJ_xxh64 performs almost as well as Dmdedup_sha1 in terms of write reduction due to deduplication. In all the experiments, WOJ removes about 50% of data blocks being written to the disk, demonstrating that use of non-collision-resistant hash function has negligible impact on the deduplication effectiveness.

4.3.6 Memory Space Overheads

Reducing memory space overhead is critical to effectively enable WOJ inside SSDs. In WOJ, three main data structures are maintained in memory: an address mapping table,

a fingerprint pool, and a reverse mapping table. For the address mapping table WOJ does not require any extra space beyond that currently available in the SSD's FTL.

The size of the fingerprint pool is proportional to the journal size. In our evaluation, we have configured the ext3 file system with journals of different sizes (from 32MB to 1GB). The performance results show little difference. So we choose a moderate journal size, which is 256MB. With this journal size, the signature pool size is capped at 768KB[2]. For the reverse mapping table, only the segment where the PPAs mapped by current journal blocks needs to stay in memory, and has a size of 512KB ($256MB/4KB * 8$). Therefore, WOJ requires only about 1.28MB memory in the SSD for its operations.

In contrast, if a full deduplication scheme with collision-resistant hashing is deployed in an SSD, a few gigabytes of metadata have to be maintained assuming a 400GB SSD [10], like the 400GB fast disk listed in Table 4.2. In the scheme even a fingerprint pool can be larger than 1GB. Specifically, in the pool each item can be of 24 bytes (a 20-byte SHA-1 value and a 4-byte PPA) and total size can be 2.4GB ($400GB/4KB * 24B$). For efficient and effective deduplication, it is often expected that its metadata are mostly cached in the memory. Otherwise, additional flash reads for cache miss and writes for metadata persistency are required, which compromises its performance. In the aforementioned experiment, we conservatively assume a sufficiently large non-volatile memory for it to hold all the metadata.

4.4 Conclusion

In this paper we describe WOJ, a weak-hashing-based deduplication scheme for deployment inside SSDs to address the write-twice issue of data journaling with negligible performance and space overheads. With a prototype implemented as a device mapper target, we extensively conducted experiments with a variety of workloads, and the results show that the ext3 file system with data journaling on WOJ-enabled SSDs can achieve up to 2.7X higher throughput than that with regular deduplication, and it also removes about half of the writes to the SSD.

[2]Each item in the pool is 12B, including an 8B fingerprint (an *xxh64* hash value) and a 4B PPA. There are at most $256MB/4KB$ items.

CHAPTER 5

ThinDedup: An I/O Deduplication Scheme that Minimizes Efficiency Loss due to Metadata Writes

5.1 Introduction

Data deduplication has been widely used in various storage systems to reduce storage space [10, 72, 81, 106]. With data growth at an explosive rate, deduplication plays an important role at various computing environments, including data centers, portable devices, and cyberphysical systems. Currently the technique finds its most successful uses in archival and backup systems [106, 25, 35] as well as with offline deduplication during primary storage system's idle time [3, 17]. However, there are apparent advantages of incorporating data deduplication on primary storage. According to an IDC report, deduplication in primary storage has "cascading benefits across all tiers", contributing to reduction of network and I/O loads to other storage tiers and of space demand on all the tiers [26]. In addition, advantages of inline deduplication are also well recognized [81]. It removes unnecessary consumption of disk space and disk bandwidth in the first place. It does not incur additional reads and writes, which interfere with the foreground workloads at a later offline deduplication.

However, even with these clear benefits inline deduplication in primary storage systems is rarely deployed in production systems [81]. Primary storage systems are usually performance sensitive. There are two main concerns on impact of the deduplication on service quality experienced by foreground users. One is degraded read performance due to compromised locality. When data are deduplicated on the hard disks, sequentiality of data layout can be disrupted, leaving one or even multiple disk seeks during originally sequential reads. This issue has been well addressed by the iDedup scheme by performing selective deduplication to retain data spatial locality [81]. The other concern, which can be more challenging to tackle, is additional writes for persistency of deduplication metadata.

As deduplication introduces a layer of indirection, it has to maintain mappings from logical address space exposed to users of the storage system to the physical address space supported by the storage devices[1]. In particular, as we limit this work on the block-level deduplication, the mapping is from a *logical block number* (LBN) to a *physical block num-*

[1]In this context the physical address is distinct from the one internal to the storage devices. It refers to a logical address in the linear address space exposed by the device(s)

ber (PBN). To service a request for synchronously writing a data block, its corresponding address mapping must be persisted onto the disk even when the data block itself does not need to be due to its deduplication. In addition to the address mapping, there are other metadata whose persistency can be expensive, including that recording mapping between a data block's fingerprint and its physical address, and a data block's reference count indicating number of logical block addresses mapped to it. Furthermore, frequency of updating the metadata is high. For example, with every write corresponding signature(s) have to be updated. As another example, in many deduplication system implementations and designs, out-of-place block writing is used to enable efficient maintenance of consistency between a block's content and its fingerprint [83, 81, 11]. In addition, because the out-of-place writes are made into a log, slow random accesses can be turned into fast sequential ones. These benefits come at the cost of high metadata maintenance cost – every write causes a new LBA to PBA address mapping.

For high persistency and strong consistency of the system, immediate persistency of the metadata is required, which poses significant challenges to use of inline deduplication in the primary storage. First, the metadata are small compared to the data block size. Writing them through the block interface of disks can make the disk bandwidth substantially underutilized, or the seemingly small persistency cost can be greatly amplified. Second, many today's applications prefer to quickly persist user data to minimize chances of losing them in a trend where concern on user experience exceeds that on consumption of resources [37]. This may neutralize the effort of collectively writing metadata through batched service of requests for high I/O efficiency and make metadata I/O even more expensive. Third, metadata may be retained in non-volatile memory without being immediately persisted on the disks. However, this requires special hardware support, such as PCM (Phase Change Memory) or battery- or supercapacitor-backed RAM, which may not be available. It is desired that a general-purpose solution does not assume availability of such supports while still achieving similar performance and persistency. Fourth, to maintain crash consistency between metadata and data on the disk, one has to pay extra cost to apply approaches such as journaling, shadowing-based atomic write, or flush-ordered writes. These approaches incur expensive disk flush operations and/or additional writes.

In this work we propose a deduplication scheme, named *ThinDedup*, to address the challenges without requiring any special hardware supports. The idea is to compress the data in the data blocks to make room for holding (critical) metadata. Because the metadata are hidden in the data blocks, they can be immediately persisted with data without concerns of amplified write cost, increased latency, and frequent use of flushes. To further reduce use of flushes for keeping data and metadata order, we always store address mapping and signature about the same data block together to enable non-ordered writes of data and

metadata. We note that in the workloads of an online primary storage it is more common to have data blocks that can be compressed than those of an archival or backup storage [16]. In addition, as we will show, to be effective ThinDedup only needs a small percentage of the data blocks to be slightly compressible.

In a summary, we made three contributions in the work.

- We address the metadata issue, which is a major concern of using the deduplication technique in the inline primary storage system, with an innovative and effective approach. This is achieved in the proposed ThinDedup scheme by (1) hiding the critical address mapping information in the compressed data blocks, and (2) collocating two types of metadata, address mapping and corresponding fingerprint, to minimize use of flushes.
- We design ThinDedup that can take advantage of compressibility of data blocks with minimally added compression cost and leaving only a relatively small percentage of data blocks in their compressed format.
- We have implemented a ThinDedup prototype as a Linux device mapper target and extensively evaluated it. Experiment results show that ThinDedup can provide up to 3X throughput compared to state-of-the-art deduplication schemes. Meanwhile, the latency can be reduced by up to 88% without compromising the throughput.

5.2 The design of ThinDedup

A major design goal of ThinDedup is to remove extra metadata writes out of critical path of inline deduplication in primary storage system and to minimize use of flushes without compromising consistency requirement. While insertion of metadata into compressed data blocks is an appealing idea, there are a number of challenges to address in the design to make it truly effective. First, compression consumes CPU cycles, and reading compressed data requires decompression operation. While these costs are negligible compared to the access times of hard disks, they might become a performance issue when faster devices, such as SSD, are used. The design should ensure that overhead of data block compression and decompression is sufficiently low compared to the operation cost of both hard disk and SSD. Second, to reduce (de)compression cost only necessary data blocks are compressed. With compressed and uncompressed data blocks co-existing on the disk, these two kinds of data blocks should be able to be identified with minimal metadata support and overhead, and be managed efficiently. Third, when a data block is deduplicated (not to be stored on the disk) or is incompressible, its metadata cannot be inserted into the block itself. The design has to make sure that success of metadata insertion does not strongly rely on small deduplication ratio or large percentage of compressible blocks.

5.2.1 Window-based Metadata Persistence

In a deduplication system, there are two critical issues that must be addressed in its design, which are persistency and consistency. For persistency, a synchronous write request is acknowledged only after it is ensured that the data in the request will not be lost even with a system crash afterwards. For consistency, with a loss of in-memory data structure, possibly due to a system crash, at any time, all data and metadata on the disk must remain consistent to each other. While immediate persistency and consistency can be too expensive to achieve, we use window-based batch persistency and fingerprint-assisted consistency in ThinDedup.

In ThinDedup, all incoming write requests are buffered in the memory, until the total number of data blocks involved in the writes reaches a pre-defined number threshold or the interval since the first request arrives in the buffer reaches a time threshold. The window size is defined by either of the thresholds, whichever reached earlier. At the end of the window, we calculate total amount of metadata (except reference count of physical blocks) to be updated due to the writes, and select one data block for compression. If the block can be compressed and the extra space made available by the compression is large enough to hold the metadata, all the metadata are inserted into the compressed block. If the extra space is not large enough, only part of them is inserted. Whenever there are still metadata remaining to be inserted, another block is selected for its data compression. This process is repeated until all the metadata are inserted, or all the data blocks have been tried.

After this, all the dirty blocks about the writes in the window are written to the disk without using flushes to enforce a particular order between them for high disk efficiency. The dirty blocks include data blocks that are compressed and hold metadata and that are not compressed. If there still exist some metadata that cannot be inserted into data blocks, they will stay in their respective metadata blocks and be written to the disk. After submitting the block writes to the disk, ThinDedup issues a flush command to the disk to make sure all the blocks are persisted. Only after the flush is complete, the write requests that had been buffered during the window are returned with acknowledgement indicating the data have been safely written.

Figures 5.1 and 5.2 show how ThinDedup works to serve write requests in a flush window for avoiding extra writes and flushes due to metadata persisting. Based on the data compressibility and deduplication ratio of the blocks in a flush window, there are mainly four cases on scheduling and servicing the requests. In the first case shown in Figure 5.1a, some data blocks (e.g., A and D) in the window are deduplicated and some of the other blocks (e.g., B and C) are selected for compression to make room to hold metadata for data blocks in the window. In this case, compression of Block B produces sufficiently large extra space to hold all the metadata. All the metadata are then inserted in the block, next

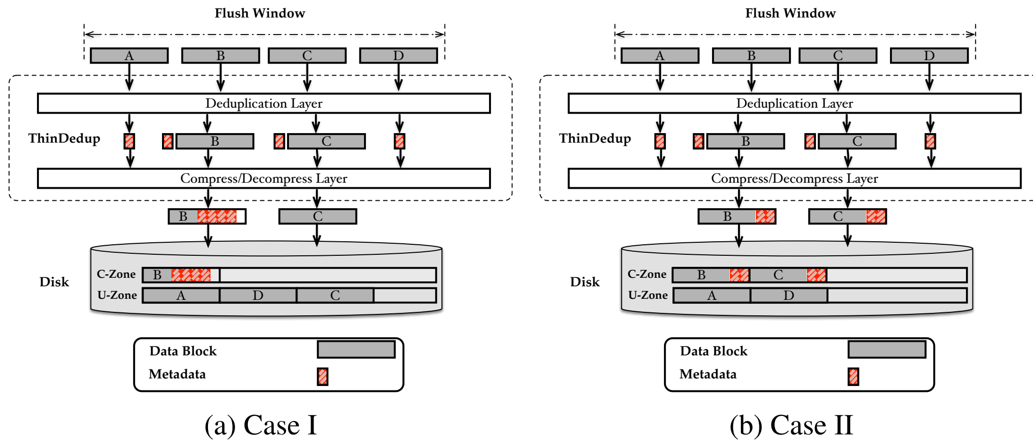
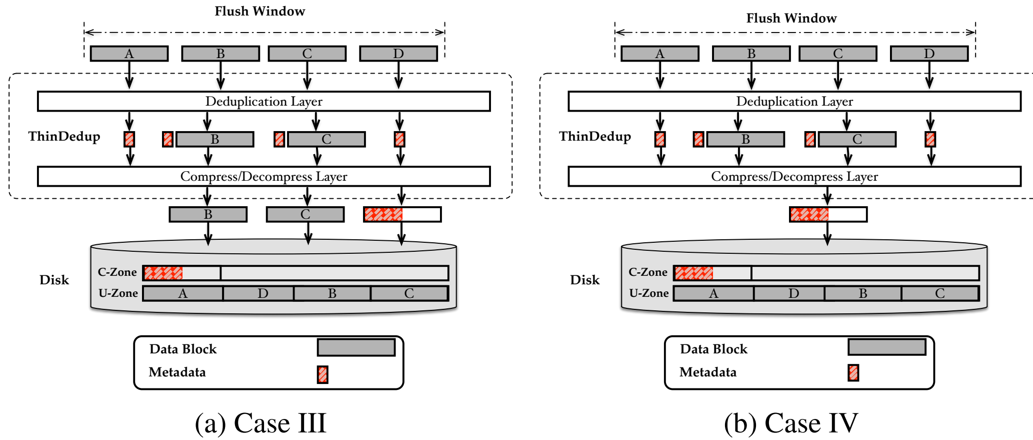


Figure 5.1: *Cases of serving write requests in a flush window in ThinDedup where explicit metadata persisting is removed. The two cases differ due to different deduplication ratio and the compressibility of blocks in a window. In the figure, the four data blocks (A, B, C, and D) are in the same window.*

to the compressed data, and data block C is left without being compressed. After that, the compressed and uncompressed blocks (B and C) are written to the disk and stored in different regions as explained in Section 5.2.2. According to existing studies on primary storage workloads' deduplication ratio and compressibility characteristics [54], this case represents the most common scenario in practice. Figure 5.1b illustrates the second case where the metadata have to be inserted in more than one compressed data blocks. The case usually happens when the window size is very large (e.g., 100 or larger), or the compression ratio of the data blocks is very low. In both of the two cases, ThinDedup can avoid explicitly writing metadata by embedding them into compressed data blocks. More importantly, enforcement of ordering between data and metadata persisting, which can be very expensive, is removed.

If the compressed data block(s) cannot make enough space to hold all metadata, as shown in Figure 5.2a, or there are not any data blocks to be written to disk at all due to a complete deduplication, as shown in Figure 5.2b, extra blocks containing the metadata must be explicitly written to the disk for metadata persistency. Actually, Figure 5.2a represents a less likely case where none of the data blocks to be written can be compressed. In these two cases, ThinDedup does not remove metadata writes. However, it can still remove requirement on write ordering between metadata and data, which is also performance-critical. This is due to the batching of fingerprints with LPA-to-PBA mappings.

This design has several efforts on reducing I/O costs. First, persistency with a window of data blocks improves write locality exploited by the I/O scheduler. Second, with



(a) Case III (b) Case IV

Figure 5.2: *Cases of serving write requests in a flush window in ThinDedup where explicit metadata writes are still needed. The two cases differ due to different deduplication ratio and the compressibility of blocks in a window. In the figure, the four data blocks (A, B, C, and D) are in the same window.*

such a window of data blocks, it would be relatively easy to find one or some blocks that can be compressed to receive metadata, even when deduplication ratio is high. Third, when data blocks are well compressible, we can compress only one or a few blocks for receiving all metadata associating with the writes in the window and leave most blocks in a window uncompressed. This has the potential to significantly reduce the compression cost for higher write performance as well as the decompression cost for higher read performance. Fourth, address mapping and fingerprint about the same data block are stored together, either inserted into a compressed data block or left in a metadata block. Therefore, they will be atomically written to the disk. Usually, a data block has to be written (persisted) before its corresponding address mapping. This is often enforced with a flush between the writes, a soft-update-style approach for crash consistency [29]. However, by storing the data block's fingerprint together with the mapping entry ThinDedup allows them to be persisted in any order without using a flush. This doesn't lead to a consistency issue. Assume that a system crash happens when the mapping info is persisted but the corresponding data block is not yet. As the mapping is stored with the fingerprint, ThinDedup uses the fingerprint to verify the data block at the address pointed to by the mapping and can detect that the data block is not a valid one. Therefore, wrong data will not be returned.

Note that the address mappings and fingerprints inserted into data blocks on the disk are not ready for online use as they may not yet be reflected in on-disk data structures regarding the deduplication's metadata. However, for a substantially long time period they are still in memory for serving I/O requests before being forced out of memory. Further-

more, after their persistency with the data blocks, they will incrementally be committed to the data structure.

In a deduplication system a reference count is maintained for each physical block tracking number of logical blocks mapped to the physical block. Once the count is decremented to 0, the physical block is likely to be reclaimed for receiving new data. Whenever a data block is updated, the old physical block's count cannot be decremented before incrementing new physical blocks's count and persistency of new data to prevent data loss. Enforcing such an order for every update needs many flushes. Instead, ThinDedup enforces the consistency requirement in a much larger scale. For every certain (large) number of windows ThinDedup first writes all increased reference counts during the windows to the disk, followed with a flush, and then writes all decreased counts. In this way, the flush's cost can be well amortized. A crash during the process may leave inconsistency between reference counts and address mappings. This is resolved by scanning the address mappings touched during the windows.

5.2.2 Zone-based Data Persistence

ThinDedup has two types of data blocks to store on the disk (compressed and uncompressed). However, a data block itself cannot reveal its own type. Therefore, ThinDedup sets up two types of space zone, each for holding blocks of corresponding type. Specifically, *C-Zone* is to hold compressed blocks with metadata inserted in them, and *U-Zone* is to hold uncompressed (regular) data blocks. Figures 5.1 and 5.2 show examples how compressed and uncompressed data blocks are stored in the C-Zone and U-Zone.

A zone holds a large number of blocks (e.g., a 4MB zone can hold 1024 4KB blocks). On the disk there is a zone bitmap, where each zone's status is represented by two bits. There are four possible statues: unallocated, U-Zone, uncommitted C-Zone, and committed C-Zone. ThinDedup periodically (usually not during the period when the system is fully loaded) writes the metadata embedded in the compressed blocks in a C-Zone to the well-structured metadata area on the disk. After the commitment, the C-Zone changes its status from 'uncommitted' to 'committed'. Note that this commitment usually does not involve reading the compressed blocks into the memory, as recently used metadata are always retained in the memory for high performance. Because a zone is relatively large, its status does not change frequently and the cost for maintaining its status is very small.

The structured metadata area include two arrays (an array of logical-to-physical block address mapping entries and an array of physical-address-to-reference-count entries) and a B+ tree for indexing fingerprints to their corresponding physical addresses. However, they may not be up to date, as the newest updates may be still only permanently stored in

the C-Zones and in the volatile memory. After an expected system crash, the volatile updates are lost. Although the updates are also available in the C-Zone(s), they are not yet well indexed and readily usable. To recover the metadata, ThinDedup needs to scan all compressed blocks that contain not-yet-committed metadata to extract the metadata and commit them to the metadata structure. Because zone statuses are synchronously persisted, this scanning only needs to cover those zones whose statuses are marked as 'uncommitted C-Zones'. Because C-Zones are periodically committed, the zones that have to be scanned during the recovery period is of small number and ThinDedup's impact on the recovery time is minimal.

The allocation of zone space is generally conducted in a manner similar to that of a log-structured file. Whenever a recently allocated zone is filled to its capacity with blocks of the same type (either compressed or not), a new zone of the same type is allocated. Zones are allocated sequentially and the new zone is appended at the end of the log. Similar to a log-structure file system, when serving a write to LBA that has been mapped to a PBA ThinDedup does not overwrite data in the PBA. Instead, it performs an out-of-place write to a new PBA to ease the maintenance of metadata, in particular, the consistency of a data block's content and its fingerprint. Correspondingly, reference count of the original PBA block is decremented by one. When the count is reduced to zero, space occupied by the corresponding block is available for reclamation. While the disk capacity is limited, the space has to be reclaimed and reused in a later time. The common practice in log-structure file systems for performing garbage collection in a selected segment (equivalent to the zone in ThinDedup) is to copy all live blocks out of the segment and make the entire segment available for new allocations. However, ThinDedup does not take the strategy to minimize metadata maintenance cost. Migration of each live block can lead to a number of updates on metadata, some of which may not be committed yet. These metadata include LBA-to-PBA mappings, PBA-to-reference-count entry, and fingerprint-to-PBA entry. As a major design objective of ThinDedup is to minimize metadata write cost, we leave the live blocks in a zone in place during space reclamation and re-allocate the available space in the zone. A downside of the strategy is to spatial locality may be compromised for sequential writes. While the impact is small for SSD, it can potentially degrade I/O performance on the hard disk. To this end, ThinDedup first selects zones with a large number of contiguous idle blocks for space reclamation. In the future, we will consider introducing operations for defragmenting scattered idle spaces when the system is not loaded. In the reuse of idle blocks in a zone, there are two issues that have to be addressed effectively.

The first issue is about efficient maintenance of block status. To determine whether a block is idle, we maintain a bitmap for a zone, each bit for a block. The bitmap needs to be up-to-date so that allocated blocks are not re-allocated until their reference counts

reach zero. For correctness and space efficiency the bitmap needs to be updated whenever a block is allocated or a block's reference count turns into zero. The bitmaps are small enough to be kept in memory and keeping the maps in the memory up-to-date is efficient. However, immediately updating them on the disk is not affordable in ThinDedup. In theory, after a loss of up-to-date bitmaps in the memory due to a power failure or system crash, the bitmaps on the disk can be brought up to date by scanning all the block mappings. However, such a recovery process can be too long. To reduce number of zones that have to be scanned in the recovery, we assign each zone a one-bit flag indicating whether the zone's in-memory bitmap is consistent to the one on the disk ('clean') or not ('dirty'). When a clean bitmap is to be updated, ThinDedup changes its status to 'dirty' and synchronously write it to the disk. The following updates on the bitmap will not incur any I/O operation until the bitmap is scheduled to be persisted. Because of spatial locality in the block write and allocation, the number of dirty bitmaps would be limited. When an idle block in a committed C-Zone is re-allocated and receives a new block of data, the zone must change its status to 'uncommitted' to reflect the fact that there are embedded and uncommitted metadata in the zone. While this status change also needs to be persisted, we co-locate the bitmap and the commitment status about a zone and persist them together to save an I/O operation.

When new blocks are written into a committed C-Zone, their embedded metadata are not yet committed and metadata in the existing blocks have been committed. The second issue is how to efficiently differentiate them so that metadata are not unnecessarily re-committed and only new metadata are committed to overwrite old ones. To this end, ThinDedup maintains a clock for each type of metadata. The clock ticks when a new metadata entry of the type is generated, and current clock reading becomes a timestamp attached to the entry. When each metadata entry has its unique timestamp, only an entry with a larger timestamp can overwrite one with a smaller timestamp during the commitment. When a C-Zone completes its commitment and changes its status to 'committed', for each type of metadata it records the largest timestamp among all of its blocks as the zone's timestamp for this type. In this way, new metadata can be easily recognized as their timestamps are larger than the zone's. When the zone is committed again, only the new ones are considered for being committed. Because there is a possibility that the metadata entries in the structured metadata area can be newer ones, ThinDedup always compares timestamps during the commitment to make sure that old entries do not overwrite newer ones.

5.2.3 Service of Read Operations

In ThinDedup, a read request is served by first looking up in the LBA-to-PBA mapping table in the main memory with the LBA of the request as the key. This process works

as any other deduplication systems and will return a PBA indicating where the data block is stored in the disk. With the PBA, a read operation is carried out at the disk to retrieve the data block. In ThinDedup, the data block fetched from the disk cannot be directly returned to the upper-level software as it may be a compressed block and needs to be decompressed. Because normal (uncompressed) and compressed data blocks are organized into different zones (U-Zone or C-Zone) in the disk, to distinguish whether the block is compressed or not we need to figure out which zone the block is stored in. The zone id can be extracted from the block's PBA. For example, if the 4MB zone is used and the block size is 4KB, the zone id can be calculated as ($PBA >> 10$). Using the zone id to index into the zone bitmap, we can get the status of the zone, which will indicate whether it is a U-Zone or C-Zone. If the block belongs to a U-Zone, it can be returned directly to the upper-level software. Otherwise, we extract the header of the block to obtain offset and size of the compressed data in the block, and use the information to decompress the data and then return the resulting data. In the process of serving a read request, the metadata inserted in the compressed block (shown in Figures 5.1 and 5.2) are not used because the metadata are either still in the main memory or have been asynchronously spilled into the on-disk metadata area. The inserted metadata are only used to restore the system to a consistent state when the following two conditions are met. First, the system is recovering from a system failure. Second, the metadata belong to a uncommitted C-Zone. The inserted metadata in a committed C-Zone are not needed for recovery as they have been already be checkpointed to the on-disk metadata area. In ThinDedup, the service of read requests for normal data blocks is almost as efficient as existing deduplication system as only very little additional computation overhead is involved. The performance of reading compressed data is a little lower as an extra decompression operation is required. However, compared to the relatively slow I/O operations, the decompression operation can be considered as light weight. On our experiment platform as shown in Section 5.3.1, the throughput of single-threaded 4KB-block decompression is over 2GB/s, much higher than the SSD's throughput, which is around 500MB/s or lower.

5.3 Performance Evaluation

To evaluate ThinDedup's performance, we implemented a prototype at the generic operating system block device layer as a device mapper target in Linux kernel 4.7.1. The design follows the basic block read/write interfaces provided by the Dmdedup [83] framework. We extensively conducted experiments to reveal insights of its performance behaviors.

5.3.1 Experiment Setup

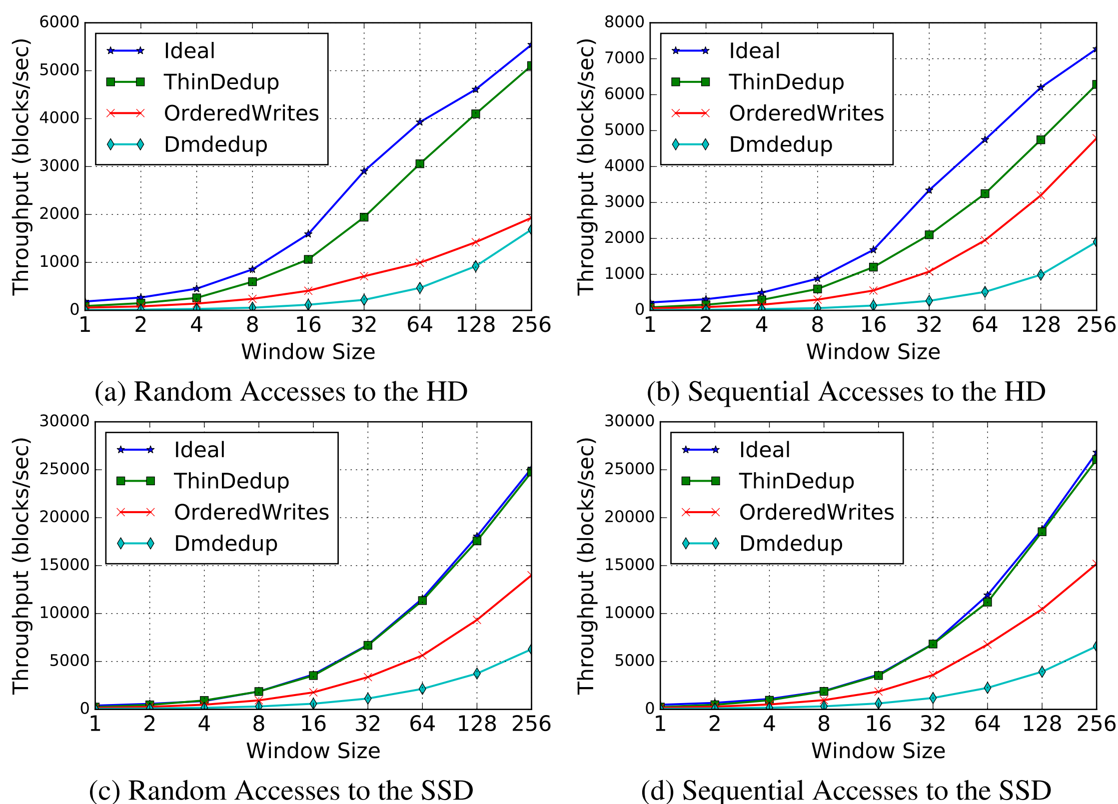
As a block-layer deduplication scheme, ThinDedup uses 4KB block as its block unit and calculates a 128-bit MD5 fingerprint for any blocks of new data. The LBA and PBA are 8 bytes, respectively, and an 16-byte timestamps are used to serialize the metadata entries. Thus, the metadata about a block write to be inserted in a compressed data block is 44 Bytes (16B LBA->PBA+16B fingerprint+16B timestamp), which is only about 1% of the block size. We use a fast compression algorithm (LZ4 [14]) for data (de)compression. On the server used in the evaluation, the algorithm can produce about 700MB/s and 2.45GB/s throughputs for compressing and decompressing 4KB blocks, respectively. Regarding window size, we test windows of different blocks in the evaluation. As write requests are continuously fed into the system without an interval, we use number of blocks requested for writing to define window size. Zone size is 4MB. C-Zones start to be committed when there are two or more uncommitted C-Zones in the system.

The experiments are conducted on a server with two Xeon E5-2680v3 2.50GHz CPUs, each has twelve cores and 30MB last-level cache. The server equips with 128GB DDR4 memory, a 320GB Western Digital Caviar Blue SATA disk and a 1TB Crucial/Micron SSD. For the hard disk, its 90th percentile latencies of sequential and random writes of 4KB block are 8.33ms and 13.36ms, respectively. For the SSD device, the latencies are 5.67ms and 5.71ms, respectively.

In the evaluation, we compare ThinDedup with three other block-layer deduplication schemes. Among them, *ideal* represents an idea (but unrealistic) scenario for producing the optimal performance, in which all metadata always stay only in the memory and only data blocks are written to the disk.

OrderedWrites is similar to ThinDedup except that its metadata are not inserted into data blocks. Instead, at the end of a window, all dirty data blocks are batch written to the disk, followed by a flush to establish the order, and then all critical metadata (mainly the block mappings) are batch written to the metadata structure on the disk. This scheme represents the upper-bound performance of the OrderMergeDedup [11] scheme, which uses flushes to enforce order and uses I/O delay and merging to exploit write locality. *OrderedWrites* limits the number of flushes for ordering to only one and allows requests within a window to be scheduled freely. However, OrderedWrites does not always ensure metadata consistency on the disk. After a crash, all metadata are restored to a consistent state by scanning the persisted critical metadata.

Dmdedup is an open-source deduplication system using page shadowing to maintain on-disk metadata consistency. It uses Linux's on-disk Copy-on-Write (COW) B-tree implementation to organize the metadata, and each metadata update involves several meta-



(a) Random Accesses to the HD

(b) Sequential Accesses to the HD

(c) Random Accesses to the SSD

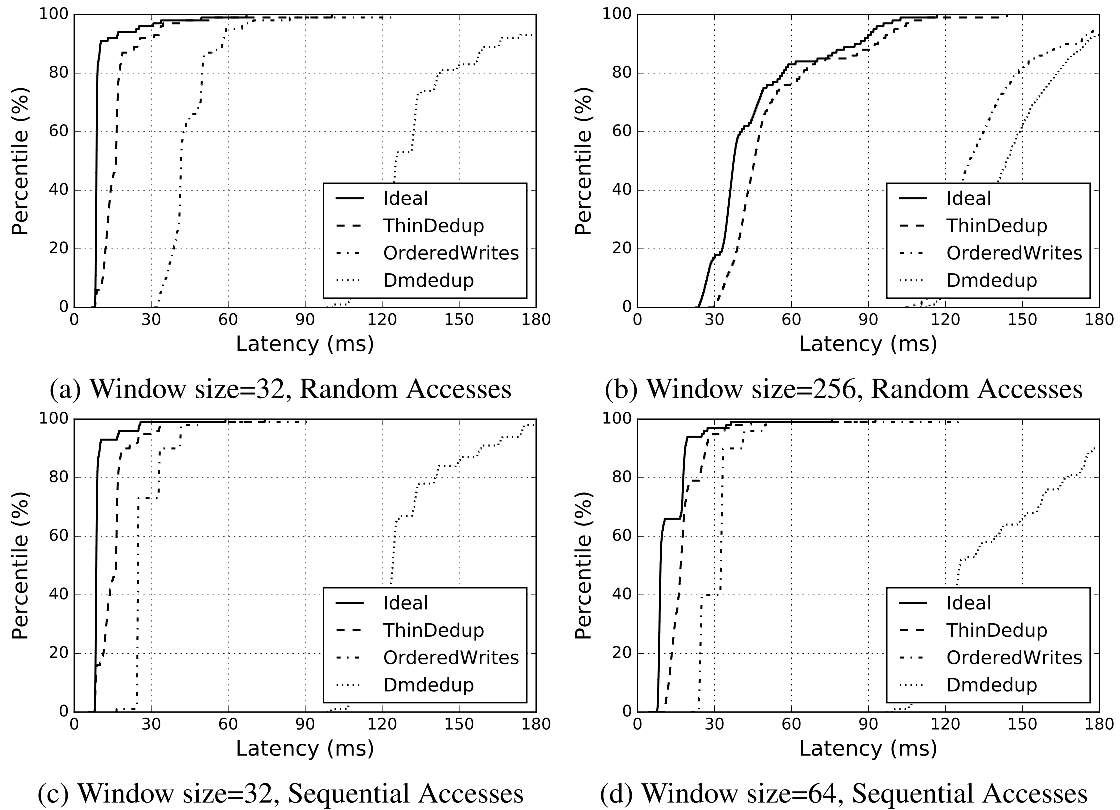
(d) Sequential Accesses to the SSD

Figure 5.3: *Write throughput for varying window sizes. The deduplication ratio is 2 and compressibility ratio is 80%.*

data page write operations and flushes. This scheme can keep the metadata and data always consistent, and does not require a recovery for consistency after a crash.

5.3.2 Experiment Results with Synthetic Workloads

In the section we generate synthetic block write traces and issue them continuously to each of the deduplication systems. For the traces there are several parameters we may change. Besides window size, we vary deduplication ratio, which is the ratio of numbers of data blocks written to the disk before and after the deduplication. We also vary compressibility ratio, which is the ratio of data blocks that can be compressed among all blocks in a window to be written to the disk. If a block can be compressed, we assume a compression ratio of 1.25, or about 20% of the block space can be made available to hold metadata after a compression. We also test two different access patterns. One is random access, where data blocks' logic address is randomly distributed in a 120GB address space, and the other



(a) Window size=32, Random Accesses
(b) Window size=256, Random Accesses
(c) Window size=32, Sequential Accesses
(d) Window size=64, Sequential Accesses

Figure 5.4: *Write latency on the* ***hard disk*** *with different window sizes. The deduplication ratio is 2 and compressibility ratio is 80%.*

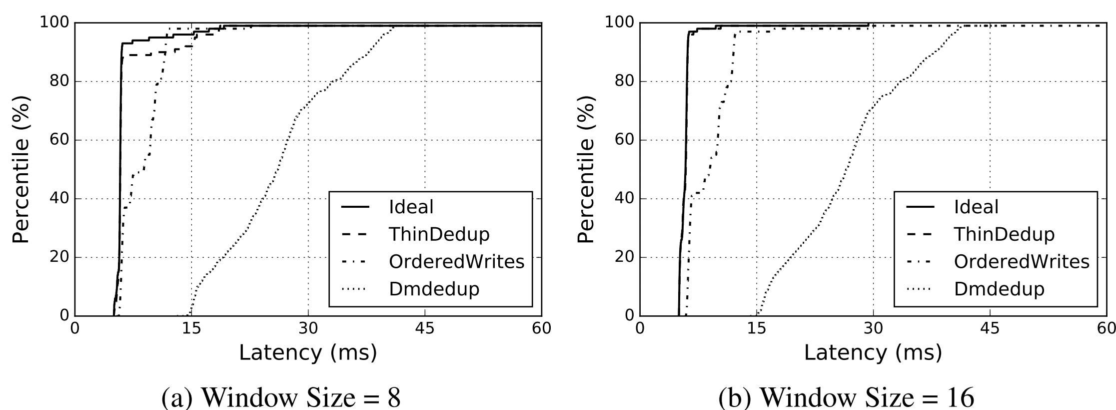
is sequential access. In the evaluation, we use both hard disk (HD) and SSD as the storage devices.

Figure 5.3 shows the write throughput with the four deduplication schemes using different window sizes with various access patterns and storage devices. As shown in all the test scenarios ThinDedup produces a throughput significantly higher than that of OrderedWrites and Dmdedup. In particular, with random accesses ThinDedup has a larger improvement than sequential ones due to the fact that random accesses cause more metadata page writes and ThinDedup avoids most of them by inserting the metadata in compressed data blocks. Generally, using larger window size helps increase relative performance advantage of ThinDedup as it is more likely to find data blocks for compression to avoid expensive explicit metadata writes. However, when the window is too large, for example 256 in Figure 5.3, the improvement can become smaller as all benefits from compression have been exploited by ThinDedup while OrderedWrites and Dmdedup favor large windows. For example, ThinDedup achieves the highest improvement when window size is 64, which is about 3X compared to that of OrderedWrites, and the improvement is 2.73X and 2.88X

when the window size is 32 and 128 respectively. With sequential access on the SSD, the improvements reduce to 1.9X and 1.77X for 32 and 128 window size, respectively. While data blocks are written sequentially, random access in the logical address space does lead to random access on the disk. Furthermore, sequential access helps reduce amplification of metadata write, as multiple metadata entries are more likely located in the same metadata blocks. This is why even on SSD, which is less sensitive to access pattern, sequential access receives a higher throughput than random access. In comparison, Dmdedup consistently has the lowest throughput, as it involves the largest number of metadata writes and flush operations for strong consistency. Throughput of ThinDedup is close to that of the *Ideal*'s deduplication. This is especially the case with the use of SSD, as our measurements show that ThinDedup removes more than 95% of the metadata and the remaining metadata writes introduces relatively low overhead. On the hard disk, write of the small amount of leftover metadata produces a larger performance loss.

It is obvious that increasing window size can help substantially increase throughput in all the systems. However, it also deteriorates request latency, which measures the period from the time when a request enters a scheduling window to the time when the window of requests are serviced. Figure 5.4 shows CDF curves of request latency with different window sizes for the four systems on the hard disk. For a given window size, ThinDedup has a latency much lower than OrderedWrites and Dmdedup, and close to that of Ideal. With a larger window size the latency can be significantly increased (though throughput also greatly increases). For example, the 80th percentile latencies of ThinDedup and OrderedWrites with a 32-block window and random access are increased to 3.5X and 3.2X, respectively, when the window size increases to 256. However, Figure 5.3a shows that ThinDedup has about the same throughput (about 2000 blocks/sec) at a 32-block window as that of OrderedWrites at a 256-block window. That indicates that ThinDedup can achieve higher throughput without having to significantly increase the window and compromising latency. For sequential access, Figure 5.3b show that ThinDedup at a 32-block window has a throughput (about 2000 blocks/sec) similar to that of OrderedWrites at 64-block window. However, it can have a lower latency, as seen in Figures 5.4c and 5.4d.

While throughputs of random access on SSD for ThinDedup with a 8-block window and OrderedWrites with a 16-block window are similar (about 2000 blocks/second), we show their latency in Figure 5.5. We find that this increase of window size does not significantly impact the latency. SSD has a much higher speed than the hard disk. The I/O time spent for a window of requests is mainly dominated by the flush operations. While increasing window size does not require more flushes, the impact is small. However, because ThinDedup reduces one flush with its fingerprint-assisted consistency for each window, its latency is significantly smaller than OrderedWrites.



(a) Window Size = 8 (b) Window Size = 16

Figure 5.5: *Random write latency on* ***SSD*** *with different window sizes. Deduplication ratio is 2 and compressibility ratio is 80%.*

To investigate how deduplication ratio affects throughput of the systems, we change the ratio in traces of different access patterns on the hard disk and SSD. The results are shown in Figure 5.6. As shown, the Ideal system keeps increasing its throughput with increase of the ratio as more writes of data blocks are removed. For other systems, deduplication of a data block only removes write of data blocks, and writes of metadata remain. For OrderedWrites and Dmdedup, higher deduplication ratio does not help increase throughput. As data blocks are sequentially written, the improvement due to reduction of number of data blocks is limited. While ThinDedup maintains a higher throughput than these two systems, interestingly its throughput has minimal increase with the increase of deduplication ratio, and even has small decrease. With a high deduplication ratio, many data blocks in a window are removed, and it is hard to remove metadata by finding compressible data blocks to insert them, and more metadata have to be explicitly written to the disk. Figure 5.7 shows the percentage of metadata blocks that have to be explicitly written to the disk due to inability of being inserted into data blocks (compared to number of written metadata blocks with zero insertion). As we can seen, if the deduplication ratio is relatively small (a ratio of around 2 is common [54]) or the window size is large, almost all writes of metadata blocks can be removed. Figure 5.6 also shows that ThinDedup's throughput is almost the same as that of the Ideal system on the SSD. On an SSD, write of a few leftover metadata blocks in ThinDedup has minimal impact on its performance. Meanwhile, its removal of a flush per window does give its advantage over the other two systems.

To understand the impact of data compressibility, we show ThinDedup's throughput on the hard disk under different data compressibility ratios in Figure 5.8. As we can see, with a reasonably large window sizes (16 or 32), as long as the ratio is 40% or higher, or 40% or more of data blocks in a window can be compressed, ThinDedup can receive its full

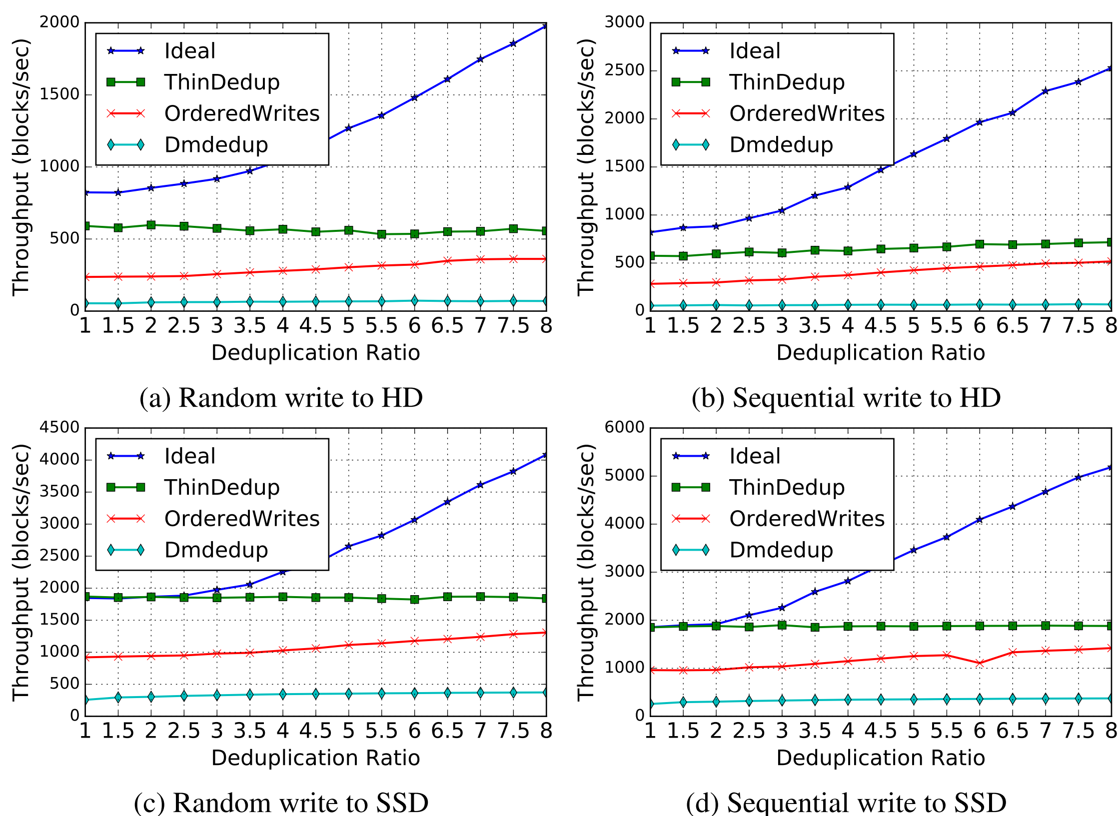
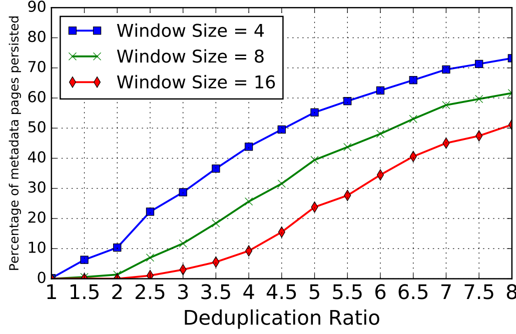


Figure 5.6: *Throughput under different deduplication ratios. The window size is 8 and compressibility ratio is 80%.*

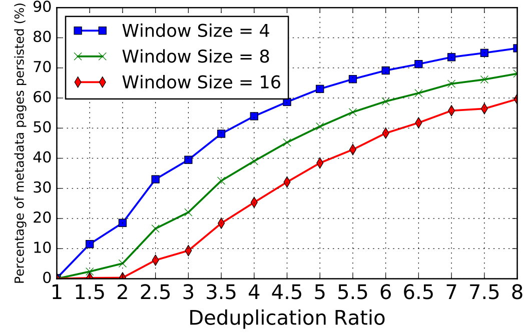
benefit. With a high compressibility ratio (e.g., 80%) and window size (e.g., 32), a high data block compression ratio (e.g, 1.5), and a moderate deduplication ratio (e.g., 2), only about 10% of the data blocks need to be compressed and only a small percentage of zones on the disk are C-Zones (e.g., 10%). Considering the high throughput of the (de)compression operations and the low percentage of C-Zone, the performance degradation caused by the (de)compression operations is negligible.

5.3.3 Experiment Results with Real-world Workloads

To evaluate ThinDedup's performance under realistic workloads, we use publicly available FIU traces, which includes three traces collected on production systems at Florida International University: Web, Mail, and Homes [49]. Each trace covers requests in 21 continuous days. Access pattern in the trace is relatively consistent across the duration of 21 days. However, the deduplication ratio varies substantially across the days. To highlight correlation of the trace characteristic with its running performance, we choose three write-

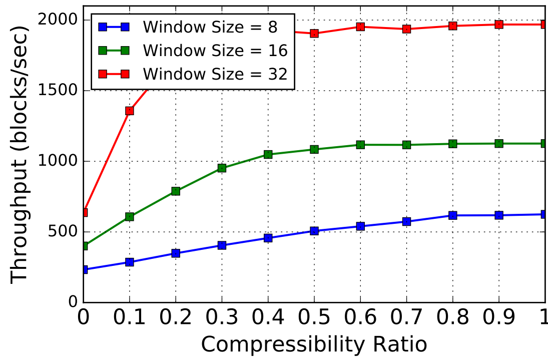


(a) Random access

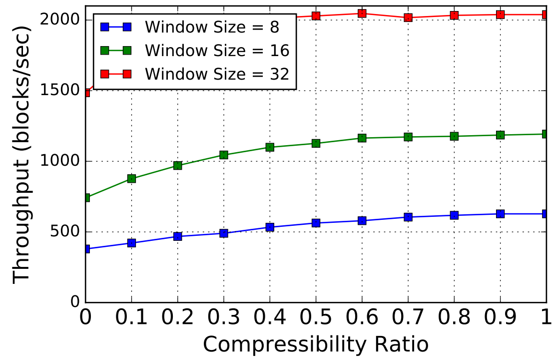


(b) Sequential access

Figure 5.7: Percentage of metadata blocks written to the disk in thinDedup with different deduplication ratio. The compressibility ratio is 80%.



(a) Random Access



(b) Sequential Access

Figure 5.8: *ThinDedup's throughput on the hard disk with different data compressibility. The deduplication ratio is 2.*

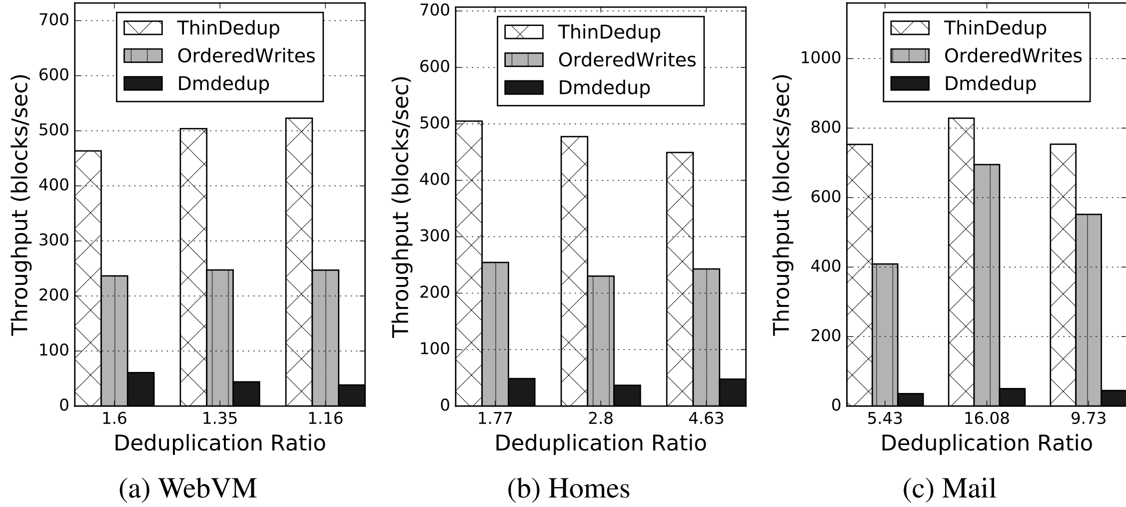
intensive segments from each trace, each segment representing one-day-long accesses of distinct deduplication ratio. In an experiment, the system is warmed up with requests preceding the tested segment of requests.

Table 5.1 lists the days on which segments are chosen as well as their respective deduplication ratios. The traces only contain data fingerprints but no real data. We assume a compressibility ratio of 80% (80% of blocks are compressible) and a compression ratio of 1.25 (each compressible block can contribute 20% of its space). According to [92], this is a conservative and reasonable assumption. The window size is 8.

Figures 5.9 and 5.10 show the throughput of trace segments on the hard disk and on SSD. Across the test cases ThinDedup produces the highest throughput, while Dmdedup's throughput is much worse than the two systems. Compared to OrderedWrites, ThinD-

Table 5.1: *Characteristics of FIU traces*

Trace	Selected Day	Dedup. Ratio (respectively)
WebVM	4,8,12	1.60, 1.35, 1.16
Homes	11,12,13	1.77, 2.80 , 4.63
Mail	1, 2, 5	5.43, 16.08, 9.73

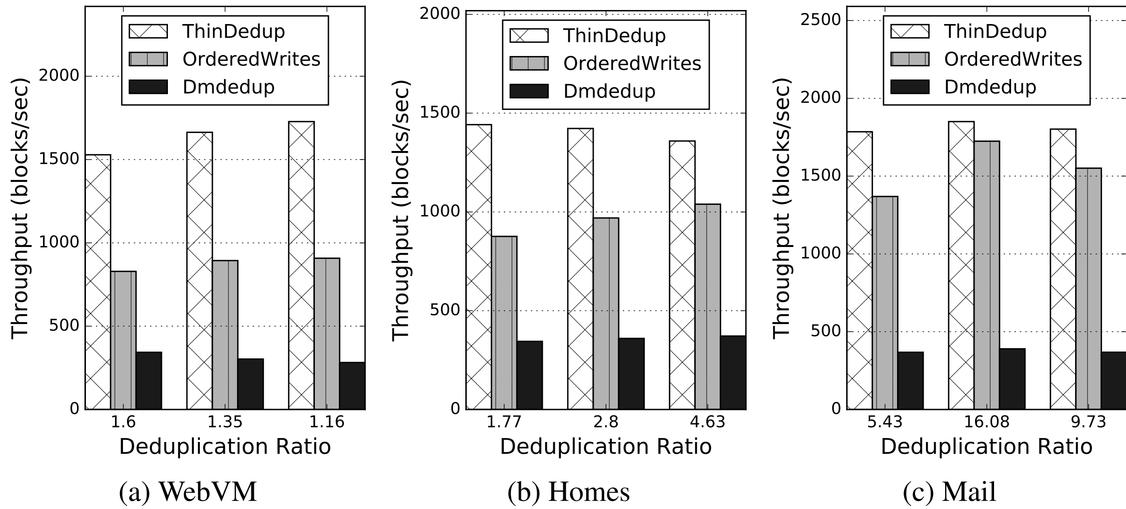


(a) WebVM (b) Homes (c) Mail

Figure 5.9: *Throughput of different trace segments on the hard disk.*

edup improves throughput by 19% to 112% on the hard disk and 7% to 91% on the SSD. Consistent to observations on synthetic workloads, a very high deduplication ratio reduces ThinDedup's advantage (see throughput of the Mail trace with the ratios of 16.08 and 9.73 in Figure 5.9c and 5.10c, respectively). A higher ratio reduces opportunity of finding compressible blocks. As we expected, improvements of ThinDedup on the SSD is smaller than those on the hard disk with the same trace segment as SSD's performance is less sensitive to random writes. However, SSD's performance is still heavily affected by flushes, which ThinDedup manages to reduce. Also, considering the amount of data blocks written to the disk when the deduplication ratio is extremely high (e.g. 16.08), about 27.8% of the metadata writes can be avoided. This is important for SSD, which has limited lifespan. We can also see that a high deduplication ratio (e.g., 16.08 and 9.73 in Figure 5.9c and 5.10c, respectively) does not necessarily lead to substantial improvement of throughput, though many writes of data blocks can be removed. As data blocks are sequentially written, cost of persisting metadata can dominate the I/O cost in the deduplication. This highlights the importance of addressing the metadata issue in an online primary deduplication.

Besides FIU traces, we also evaluate ThinDedup with the public block-level I/O traces collected on a storage cluster and released by Microsoft Research Cambridge [58].

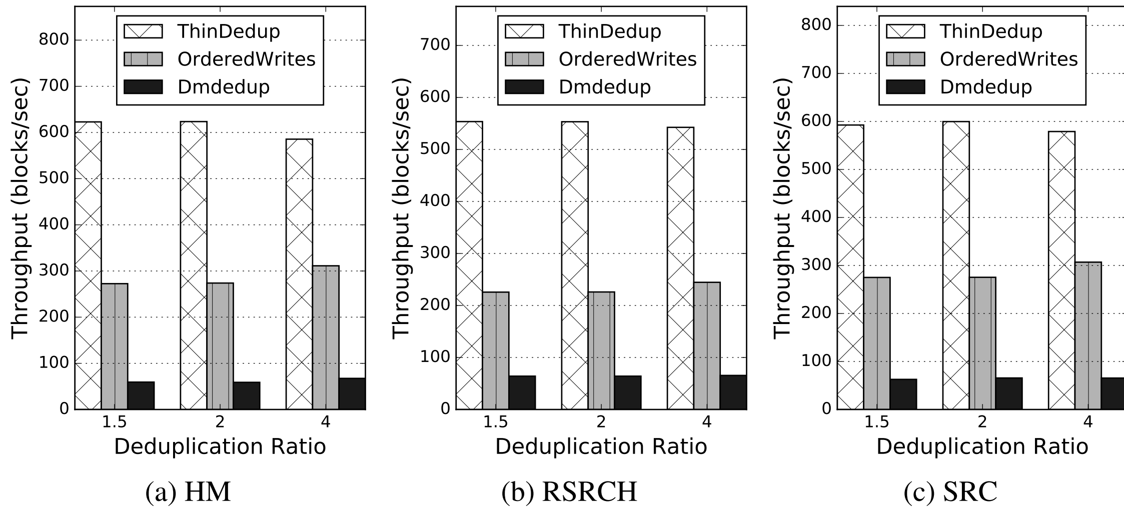


(a) WebVM (b) Homes (c) Mail

Figure 5.10: *Throughput of different trace segments on the SSD.*

The traces are captured on 36 volumes of 179 disks located in 13 servers. We randomly select three traces from three different servers (hardware monitor, research, and source control) in our experiments. MSR Cambridge traces do not expose any content information (data or fingerprint) of the requests, so we have to generate contents for the requests. According to previous publications [54], the deduplication ratio for primary storage workloads are mainly around 2. In our experiments, we choose three different deduplication ratios (1.5, 2 and 4) for the evaluation. We also assume a compressibility ratio of 80% (80% of blocks are compressible) and a compression ratio of 1.25, the same as that for the FIU trace evaluation. The window size is still 8.

Figures 5.11 and 5.12 show the throughput of the three traces with different deduplication ratio settings. ThinDedup achieves much higher throughput in all test cases than OrderedWrites and Dmdedup as expected. Compared to OrderedWrites, ThinDedup improves throughput from 88% to 145% for the hard disk and 58% to 96% for the SSD. When comparing Figures 5.11a and 5.11b, we can see that even with the same deduplication ratio, the throughput can be quite different for different workloads. The reason is that these two workloads have very different access patterns. Compared to *HM*, *RSRCH* touches about 25% more metadata pages in our experiments. The random access feature of *RSRCH* introduces more metadata writes and thus degrades the throughput. For SSD, however, the performance difference is not significant as SSD is not so sensitive to random access.



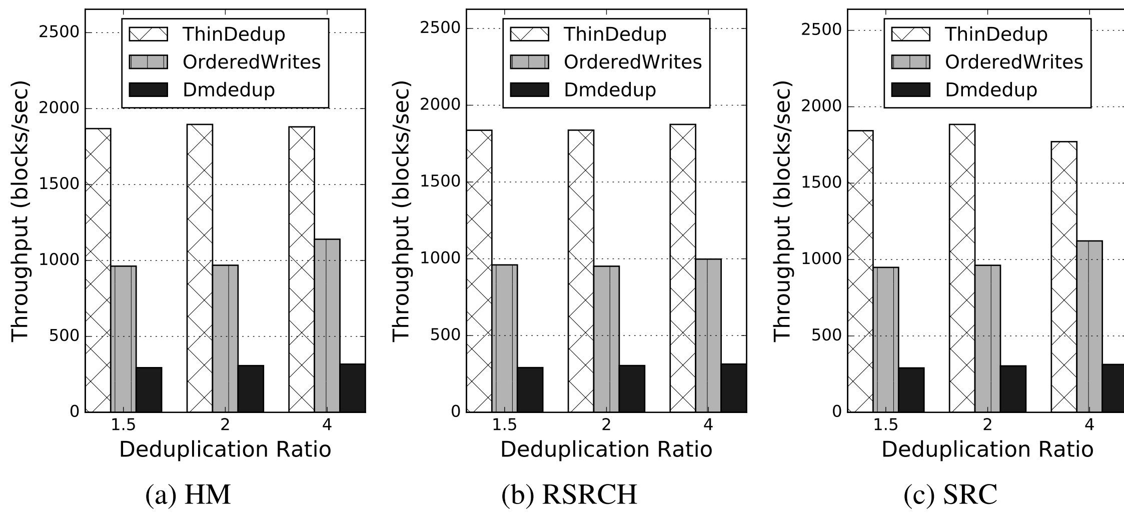
(a) HM (b) RSRCH (c) SRC

Figure 5.11: *Write throughput on the* ***hard disk*** *under different deduplication ratio for MSR Cambridge traces. The window size is 8 and data compressibility ratio is 0.8.*

5.4 Conclusion

In this paper we describe ThinDedup, an efficient deduplication scheme designed to minimize the performance loss due to metadata persistency. By embedding metadata into the compressed data blocks, ThinDedup removes most of metadata persisting operations out of the critical path. It also uses window-based batch persistency to amortize the high flush overhead. To provide crash consistency, ThinDedup stores fingerprint together with block mapping to remove requirement on the write ordering.

Our extensive experiments with micro benchmarks and real-world workloads demonstrate that ThinDedup can significantly improve performance of deduplication systems. Compared to other state-of-the-art approaches, it provides up to 3X throughput. Meanwhile, the write latency can be reduced by up to 88% without compromising the throughput.



(a) HM (b) RSRCH (c) SRC

Figure 5.12: *Write throughput on the* ***SSD*** *under different deduplication ratio for the MSR traces. The window size is 8 and data compressibility ratio is 0.8.*

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

I/O Deduplication, a key technique for data reduction, has become a standard feature in many commercial storage systems. For a storage system with deduplication feature enabled, the deduplicatio performance has a significant impact on the overall performance of the system. However, existing deduplication techniques can significantly degrade the performance due to its high computation cost in the chunking and fingerprinting phase and high I/O overheads introduced by the frequent write and flush operations for metadata persistency. The major work of this dissertation has been focused on proposing solutions to improve the performance of deduplication systems. The efforts covers all phases of data deduplication. With all the efforts, we can design a more efficient deduplication system.

6.1 Contributions

The contributions of this dissertation can be summarized as follows:

- We quantify the impact of the existing parallel CDC methods on deduplication ratio, and proposed a two-phase CDC method (SS-CDC) that can provide high chunking speedup as regular parallel CDC approaches, and achieve the same deduplication ratio as the sequential CDC method does. Also, since most of the chunking work acts in an SIMD way, it can fully exploit the ILP computation power of modern processors.
- Although SS-CDC can significantly improve the chunking speed, it relies on the computation power of specific hardware, and it does not reduce the computation in the chunking process. Fortunately, we disclosed the existence of deduplicate locality in the datasets of deduplication systems. With the locality, we can leverage the deduplication history to guide the chunking process, which helps to remove the byte-by-byte detection for a valid chunk boundary in most cases. Accordingly, we designed a very fast CDC method, named RapidCDC, that is totally different from existing chunking acceleration methods. The evaluation results with real-world datasets show that RapidCDC can provide tens of times chunking speedup with one chunking threads and the performance scales with the number of chunking threads, which totally re-

moves the performance bottleneck posed by the chunking process in CDC-based deduplication systems.

- Existing data deduplication techniques rely on collision-resistant hash functions (such as MD5 or SHA1) to generate strong fingerprints for detecting duplicates. The calculation of fingerprints is time-consuming, and for systems where data are stored on a fast disk (i.e., NVMe SSD), it has become a new performance bottleneck of the system. We identify an opportunity in journaling file system where fast non-collision-resistant hash functions can be used to generate weak fingerprints for detecting duplicates and thus avoiding the write-twice issue for the data journaling mode without compromising data correctness and reliability.
- Deduplication systems need to maintain extra metadata, which must be persisted to disk periodically. However, the persistency of the metadata can be very expensive as it will introduce many small writes and frequent use of the expensive flush operations. We propose a method that opportunistically compresses the data block to make room to insert the small metadata. With a sophisticated design, in most cases the small metadata writes on the critical path can be removed, which significantly improve the I/O efficiency of the system.

6.2 Future Work

In the cloud environment, the cloud platform stores data from many different users in the cloud storage systems. And a file can be uploaded and stored on the cloud multiple times by different users, which produces many redundant data. The application of deduplication technique in cloud storage systems can help to detect the duplicates and avoid saving the same files on the cloud multiple times, which significantly reduces the expense on storage devices. Nowadays, data deduplication in cloud storage systems has become a practice, and commercial cloud companies, including EMC and Dropbox, have already deployed deduplication technique in their cloud products for saving storage space and network bandwidth. Compared to local storage systems, the deployment of deduplication on cloud systems faces a big challenge, the data confidentiality. In the cloud environment, the data are stored remotely on the cloud and users usually have no control of the physical devices where the data are stored, which leads to extra concern of the data confidentiality. A commonly used approach to protect data confidentiality in an untrusted environment is data encryption. However, to retain the deduplicatability of the data after encryption, there are some limitations on the encryption methods. How to efficiently perform deduplication without sacrificing data confidentiality for cloud storage systems is increasingly attracting

attentions from both academia and industry. My future work will focus on this topic and try to propose solutions for deduplication of secure data in the cloud.

REFERENCES

[1] B. Aggarwal, A. Akella, A. Anand, A. Balachandran, P. Chitnis, C. Muthukrishnan, R. Ramjee, and G. Varghese. Endre: An end-system redundancy elimination service for enterprises. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, NSDI'10, pages 28–28, Berkeley, CA, USA, 2010. USENIX Association.

[2] S. Al-Kiswany, A. Gharaibeh, E. Santos-Neto, G. Yuan, and M. Ripeanu. Storegpu: Exploiting graphics processing units to accelerate distributed storage systems. In *Proceedings of the 17th International Symposium on High Performance Distributed Computing*, HPDC '08, pages 165–174, New York, NY, USA, 2008. ACM.

[3] C. Alvarez. Netapp deduplication for fas and v-series deployment and implementation guide. *Technical ReportTR-3505*, 2011.

[4] J. An and D. Shin. Offline deduplication-aware block separation for solid state disk, 2013.

[5] P. Bhatotia, R. Rodrigues, and A. Verma. Shredder: Gpu-accelerated incremental storage and computation. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, FAST'12, pages 14–14, Berkeley, CA, USA, 2012. USENIX Association.

[6] M. Bjørling, J. González, and P. Bonnet. LightNVM: The linux open-channel SSD subsystem. In *Proceedings of the 15th Usenix Conference on File and Storage Technologies*, FAST'17, pages 359–373, Berkeley, CA, USA, 2017. USENIX Association.

[7] N. Bjrner, A. Blass, and Y. Gurevich. Content-dependent chunking for differential compression, the local maximum approach. *J. Comput. Syst. Sci.*, 76(3-4):154–203, May 2010.

[8] M. Cao, S. Bhattacharya, and T. Ts'o. Ext4: The next generation of Ext2/3 filesystem., 2007.

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

[10] 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 Stroage Technologies*, FAST'11, pages 6–6, Berkeley, CA, USA, 2011. USENIX Association.

[11] Z. Chen and K. Shen. OrderMergeDedup: Efficient, Failure-consistent Deduplication on Flash. In *Proceedings of the 14th Usenix Conference on File and Storage Technologies*, FAST'16, pages 291–299, Berkeley, CA, USA, 2016. USENIX Association.

[12] H. Chu. Database benchmarks. `https://github.com/hyc/leveldb/tree/benches`, 2014.

[13] Y. Collet. xxHash wider: assessing quality of a 64-bits hash function. `http://fastcompression.blogspot.com/2014/07/xxhash-wider-64-bits.html`, 2014.

[14] Y. Collet. lz4. http://cyan4973.github.io/lz4/, 2015.

[15] Y. Collet. xxHash–Extremeley fast hash algorithm. `http://cyan4973.github.io/xxHash/`, 2016.

[16] C. Constantinescu, J. Glider, and D. Chambliss. Mixing deduplication and compression on active data sets. In *2011 Data Compression Conference*, pages 393–402. IEEE, 2011.

[17] I. Corporation. IBM white paper: IBM storage tank – a distributed storage system, Jan. 2002.

[18] I. Corporation. Intel xeon phi processors. https://www.intel.com/content/www/us/en/products/processors/xeon-phi/xeon-phi-processors.html, 2013.

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

[20] I. Corporation. Intel architecture instruction set extensions programming reference. https://software.intel.com/sites/default/files/managed/c5/15/architecture-instruction-set-extensions-programming-reference.pdf, 2018.

[21] B. K. Debnath, S. Sengupta, and J. Li. Chunkstash: Speeding up inline storage deduplication using flash memory. In *USENIX annual technical conference*, pages 1–16, 2010.

[22] Dell EMC. Data Domain - Data Backup Appliance, Data Protection, 2019.

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

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

[25] W. Dong, F. Douglis, K. Li, R. H. Patterson, S. Reddy, and P. Shilane. Tradeoffs in scalable data routing for deduplication clusters. In *FAST*, volume 11, pages 15–29, 2011.

[26] L. DuBois, M. Amaldas, and E. Sheppard. Key considerations as deduplication evolves into primary storage. *White Paper*, 223310, 2011.

[27] D. Eastlake 3rd and P. Jones. US secure hash algorithm 1 (SHA1), 2001.

[28] K. Eshghi, M. Lillibridge, L. Wilcock, G. Belrose, and R. Hawkes. Jumbo store: Providing efficient incremental upload and versioning for a utility rendering service. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies*, FAST '07, pages 22–22, Berkeley, CA, USA, 2007. USENIX Association.

[29] G. R. Ganger, M. K. McKusick, C. A. Soules, and Y. N. Patt. Soft updates: a solution to the metadata update problem in file systems. *ACM Transactions on Computer Systems (TOCS)*, 18(2):127–153, 2000.

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

[31] A. Gharaibeh, S. Al-Kiswany, S. Gopalakrishnan, and M. Ripeanu. A gpu accelerated storage system. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 167–178, New York, NY, USA, 2010. ACM.

[32] S. Ghemawat and J. Dean. Leveldb. *URL: https://github. com/google/leveldb,% 20http://leveldb. org*, 2011.

[33] J. Graham-Cumming. Rebuilding when a file's checksum changes. https://www.cmcrossroads.com/article/rebuilding-when-files-checksum-changes, 2006.

[34] J. Graham-Cumming. *The GNU make book*. No Starch Press, 2015.

[35] F. Guo and P. Efstathopoulos. Building a high-performance deduplication system. In *USENIX Annual Technical Conference*, 2011.

[36] A. Gupta, Y. Kim, and B. Urgaonkar. *DFTL*: A flash translation layer employing demand-based selective caching of page-level address mappings. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, pages 229–240, New York, NY, USA, 2009. ACM.

[37] T. Harter, C. Dragga, M. Vaughn, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. A file is not a file: understanding the i/o behavior of apple desktop applications. *ACM Transactions on Computer Systems (TOCS)*, 30(3):10, 2012.

[38] HP Workstations. Technical white paper: SSD endurance. `http://h20195.www2.hp.com/v2/getpdf.aspx/4AA5-7601ENW.pdf?ver=1.0`, 2015.

[39] D. Inc. debian: Docker Official Images. `https://hub.docker.com/_/debian/`, 2018.

[40] D. Inc. Node: Docker Official Images. `https://hub.docker.com/_/node/`, 2018.

[41] D. Inc. wordpress: Docker Official Images. `https://hub.docker.com/_/wordpress/`, 2018.

[42] L. E. Inc. How to properly calculate write endurance. `http://h20195.www2.hp.com/v2/getpdf.aspx/4AA5-7601ENW.pdf?ver=1.0`, 2015.

[43] Intel. Intel X25-M and X18-M Mainstream SATA Solid-State Drives. `https://www.intel.com/content/dam/support/us/en/documents/ssdc/hpssd/sb/Intel_SSD_50nm_X25-M_X18M_Series_Product-brief.pdf`, 2008.

[44] Intel. Intel solid-state drive 520 series product specification. `https://www.intel.com/content/dam/support/us/en/documents/ssdc/hpssd/sb/Intel_SSD_520_Series_Product_specification.pdf`, 2012.

[45] INTEL. Intel Solid State Drive 750 Series Product Specification, Revision 004. `https://www.intel.com/content/www/us/en/products/\memory-storage/solid-state-drives/gaming-enthusiast-ssds/750-series/750-400gb-2-5-inch-20nm.html`, 2015.

[46] Intel. Intel Optane SSD DC P4800X Series. `https://www.intel.com/content/www/us/en/products/memory-storage/solid-state-drives/data-center-ssds/optane-dc-p4800x-series/p4800x-750gb-2-5-inch.html`, 2017.

[47] H. Kambo and B. Sinha. Secure data deduplication mechanism based on rabin cdc and md5 in cloud computing environment. In *Recent Trends in Electronics, Information & Communication Technology (RTEICT), 2017 2nd IEEE International Conference on*, pages 400–404. IEEE, 2017.

[48] L. Kernel. Ext4 Filesystem. `https://www.kernel.org/doc/Documentation/filesystems/ext4.txt`, 2017.

[49] R. Koller and R. Rangaswami. I/O Deduplication: Utilizing Content Similarity to Improve I/O Performance. *Trans. Storage*, 6(3):13:1–13:26, Sept. 2010.

[50] M. Lillibridge, K. Eshghi, D. Bhagwat, V. Deolalikar, G. Trezise, and P. Camble. Sparse indexing: Large scale, inline deduplication using sampling and locality. In *Proccedings of the 7th Conference on File and Storage Technologies*, FAST '09, pages 111–123, Berkeley, CA, USA, 2009. USENIX Association.

[51] X. Lin, F. Douglis, J. Li, X. Li, R. Ricci, S. Smaldone, and G. Wallace. Metadata considered harmful ... to deduplication. In *Proceedings of the 7th USENIX Conference on Hot Topics in Storage and File Systems*, HotStorage'15, pages 11–11, Berkeley, CA, USA, 2015. USENIX Association.

[52] R.-S. Liu, C.-L. Yang, C.-H. Li, and G.-Y. Chen. Duracache: A durable SSD cache using mlc nand flash. In *Proceedings of the 50th Annual Design Automation Conference*, DAC'13, pages 166:1–166:6, New York, NY, USA, 2013. ACM.

[53] Y. Lokeshwari, B. Prabavathy, and C. Babu. Optimized cloud storage with high throughput deduplication approach. In *Proceedings of the International Conference on Emerging Technology Trends (ICETT)*. Citeseer, 2011.

[54] M. Lu, D. Chambliss, J. Glider, and C. Constantinescu. Insights for data reduction in primary storage: a practical analysis. In *Proceedings of the 5th Annual International Systems and Storage Conference*, page 17. ACM, 2012.

[55] R. McDougall and J. Mauro. Filebench. `http://www.nfsv4bat.org/Documents/nasconf/2004/filebench.pdf(Citedonpage56.)`, 2005.

[56] D. Meister and A. Brinkmann. Multi-level comparison of data deduplication in a backup scenario. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, SYSTOR '09, pages 8:1–8:12, New York, NY, USA, 2009. ACM.

[57] A. Muthitacharoen, B. Chen, and D. Mazières. A low-bandwidth network file system. *SIGOPS Oper. Syst. Rev.*, 35(5):174–187, Oct. 2001.

[58] D. Narayanan, A. Donnelly, and A. Rowstron. Write off-loading: Practical power management for enterprise storage. *ACM Transactions on Storage (TOS)*, 4(3):10, 2008.

[59] D. Nellans, M. Zappe, J. Axboe, and D. Flynn. ptrim ()+ exists (): Exposing new ftl primitives to applications, 2011.

[60] Neo Technology. Neo4j Graph Database Platform. `https://neo4j.com/`, 2018.

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

[62] NetApp inc. ONTAP Data Management Software: ONTAP Data Management Software. `https://www.netapp.com/us/products/data-management-software/ontap.aspx`, 2018.

[63] NETAPP inc. 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`, 2019.

[64] A. Neumann. Fast implementation of Content Defined Chunking (CDC) based on a rolling Rabin Checksum in C. `https://github.com/fd0/rabin-cdc`, 2018.

[65] G. operating system. Mirror of all gcc svn branches and tags. `https://gcc.gnu.org/git/?p=gcc.git;a=summary`, 2017.

[66] T. S. Pillai, R. Alagappan, L. Lu, V. Chidambaram, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Application crash consistency and performance with ccfs. In *Proceedings of the 15th Usenix Conference on File and Storage Technologies*, FAST'17, pages 181–196, Berkeley, CA, USA, 2017. USENIX Association.

[67] T. S. Pillai, V. Chidambaram, R. Alagappan, S. Al-Kiswany, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. All file systems are not created equal: On the complexity of crafting crash-consistent applications. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 433–448, Berkeley, CA, USA, 2014. USENIX Association.

[68] T. S. Pillai, V. Chidambaram, R. Alagappan, S. Al-Kiswany, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Crash consistency. *Commun. ACM*, 58(10):46–51, Sept. 2015.

[69] V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Analysis and evolution of journaling file systems. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATC '05, pages 8–8, Berkeley, CA, USA, 2005. USENIX Association.

[70] A. Pranckevicius. Performance tests of various non-cryptographic hash functions, on various cpus. `http://aras-p.info/blog/2016/08/09/More-Hash-Function-Tests/`, 2016.

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

[72] S. Quinlan and S. Dorward. Venti: A new approach to archival storage. In *FAST*, volume 2, pages 89–101, 2002.

[73] M. O. Rabin. Fingerprinting by random polynomials. *Technical report*, 1981.

[74] J. Reinders. Intel avx-512 instructions, 2013.

[75] R. Rivest. The md5 message-digest algorithm. `https://tools.ietf.org/html/rfc1321`, 1992.

[76] D. Robbins. Advanced filesystem implementor's guide, part 8. `https://www.ibm.com/developerworks/library/l-fs8/index.html`, 2001.

[77] S. Sanfilippo and P. Noordhuis. Redis. `http://redis.io`, 2015.

[78] B. Schroeder, R. Lagisetty, and A. Merchant. Flash reliability in production: The expected and the unexpected. In *Proceedings of the 14th Usenix Conference on File and Storage Technologies*, FAST'16, pages 67–80, Berkeley, CA, USA, 2016. USENIX Association.

[79] P. Shilane, G. Wallace, M. Huang, and W. Hsu. Delta compressed and deduplicated storage using stream-informed locality. In *HotStorage*, 2012.

[80] G. Soundararajan, V. Prabhakaran, M. Balakrishnan, and T. Wobber. Extending SSD lifetimes with disk-based write caches. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies*, FAST'10, pages 8–8, Berkeley, CA, USA, 2010. USENIX Association.

[81] K. Srinivasan, T. Bisson, G. R. Goodson, and K. Voruganti. idedup: latency-aware, inline data deduplication for primary storage. In *FAST*, volume 12, pages 1–14, 2012.

[82] V. Tarasov, D. Jain, G. Kuenning, S. Mandal, K. Palanisami, P. Shilane, S. Trehan, and E. Zadok. Dmdedup: Device mapper target for data deduplication, 2014.

[83] V. Tarasov, D. Jain, G. Kuenning, S. Mandal, K. Palanisami, P. Shilane, S. Trehan, and E. Zadok. Dmdedup: Device mapper target for data deduplication. In *2014 Ottawa Linux Symposium*, 2014.

[84] Debian Project. Debian CD Image Download page. `https://cdimage.debian.org/debian-cd/current/amd64/iso-cd/`, 2017.

[85] GNU Operating System. Timestamp resolution and make. `https://www.gnu.org/software/autoconf/manual/autoconf-2.61/html_node/Timestamps-and-Make.html`, 2008.

[86] The Linux Kernel Organization, Inc. The Linux Kernel Archives, 2019.

[87] N. Tolia, M. Kozuch, M. Satyanarayanan, B. Karp, T. C. Bressoud, and A. Perrig. Opportunistic use of content addressable storage for distributed file systems. In *USENIX Annual Technical Conference, General Track*, volume 3, pages 127–140, 2003.

[88] L. Torvalds. Linux 2.6.29. `https://lwn.net/Articles/326505/`, 2009.

[89] L. Torvalds and J. Hamano. Git: Fast version control system. `http://git-scm.com`, 2010.

[90] S. Tweedie. Ext3, journaling filesystem. In *Ottawa Linux Symposium*, pages 24–29, 2000.

[91] C. Ungureanu, B. Atkin, A. Aranya, S. Gokhale, S. Rago, G. Calkowski, C. Dubnicki, and A. Bohra. Hydrafs: A high-throughput file system for the hydrastor content-addressable storage system. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies*, FAST'10, pages 17–17, Berkeley, CA, USA, 2010. USENIX Association.

[92] G. Wallace, F. Douglis, H. Qian, P. Shilane, S. Smaldone, M. Chamness, and W. Hsu. Characteristics of backup workloads in production systems. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, FAST'12, pages 4–4, Berkeley, CA, USA, 2012. USENIX Association.

[93] J. Wei, H. Jiang, K. Zhou, and D. Feng. Mad2: A scalable high-throughput exact deduplication approach for network backup services. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10, pages 1–14, Washington, DC, USA, 2010. IEEE Computer Society.

[94] Wikipedia. Wiki Page: Journaling file system. `https://en.wikipedia.org/wiki/Journaling_file_system`, 2017.

[95] Wikipedia. Wiki Page: XFS-Journaling. `https://en.wikipedia.org/wiki/XFS#Journaling`, 2017.

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

[97] Y. Won, K. Lim, and J. Min. Much: Multithreaded content-based file chunking. *IEEE Transactions on Computers*, 2015.

[98] Y. Won, K. Lim, and J. Min. Much: Multithreaded content-based file chunking. *IEEE Transactions on Computers*, 64(5):1375–1388, 2015.

[99] W. Xia, H. Jiang, D. Feng, and Y. Hua. Silo: A similarity-locality based near-exact deduplication scheme with low ram overhead and high throughput. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'11, pages 26–28, Berkeley, CA, USA, 2011. USENIX Association.

[100] W. Xia, H. Jiang, D. Feng, and L. Tian. Accelerating data deduplication by exploiting pipelining and parallelism with multicore or manycore processors. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST'12 Poster)*, pages 1–2, 2012.

[101] W. Xia, H. Jiang, D. Feng, L. Tian, M. Fu, and Z. Wang. P-dedupe: Exploiting parallelism in data deduplication system. In *2012 IEEE Seventh International Conference on Networking, Architecture, and Storage*, pages 338–347, Xiamen, China, 2012. IEEE, IEEE.

[102] W. Xia, H. Jiang, D. Feng, L. Tian, M. Fu, and Y. Zhou. Ddelta: A deduplication-inspired fast delta compression approach. *Performance Evaluation*, 79, 2014.

[103] W. Xia, Y. Zhou, H. Jiang, D. Feng, Y. Hua, Y. Hu, Y. Zhang, and Q. Liu. 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, pages 101–114, Berkeley, CA, USA, 2016. USENIX Association.

[104] C. Yu, C. Zhang, Y. Mao, and F. Li. Leap-based content defined chunking – theory and implementation. In *the 31st Symposium on Mass Storage Systems and Technologies (MSST)*, Santa Clara, CA, 2015. IEEE.

[105] Y. Zhang, D. Feng, H. Jiang, W. Xia, M. Fu, F. Huang, and Y. Zhou. A fast asymmetric extremum content defined chunking algorithm for data deduplication in backup storage systems. *IEEE Transactions on Computers*, 66(2), 2017.

[106] B. Zhu, K. Li, and H. Patterson. 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, pages 18:1–18:14, Berkeley, CA, USA, 2008. USENIX Association.