RapidCDC: Leveraging Duplicate Locality to Accelerate Chunking in CDC-based Deduplication Systems

Fan Ni*
fan@netapp.com
NetApp, Inc.
Sunnyvale, CA

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

ABSTRACT

I/O deduplication is a key technique for improving storage systems' space and I/O efficiency. Among various deduplication techniques content-defined chunking (CDC) based deduplication is the most desired one for its high deduplication ratio. However, CDC is compute-intensive and time-consuming, and has been recognized as a major performance bottleneck of the CDC-based deduplication system.

In this paper we leverage the existence of a property in the duplicate data, named duplicate locality, that reveals the fact that multiple duplicate chunks are likely to occur together. In other words, one duplicate chunk is likely to be immediately followed by a sequence of contiguous duplicate chunks. The longer the sequence, the stronger the locality is. After a quantitative analysis of duplicate locality in real-world data, we propose a suite of chunking techniques that exploit the locality to remove almost all chunking cost for deduplicatable chunks in CDC-based deduplication systems. The resulting 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 applied on data sets with high data redundancy. The other feature is that its high efficiency does not rely on high duplicate locality strength. 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× higher than regular CDC. Meanwhile, it maintains (nearly) the same deduplication ratio.

CCS CONCEPTS

• **Information systems → Deduplication**.

KEYWORDS

storage systems, deduplication, CDC, content-defined chunking, locality

*This work was done when he was a Ph.D student at the CSE Department of UT Arlington.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SoCC '19, November 20–23, 2019, Santa Cruz, CA, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6973-2/19/11…$15.00
https://doi.org/10.1145/3357223.3362731

ACM Reference Format:
Fan Ni and Song Jiang. 2019. RapidCDC: Leveraging Duplicate Locality to Accelerate Chunking in CDC-based Deduplication Systems. In *ACM Symposium on Cloud Computing (SoCC '19), November 20–23, 2019, Santa Cruz, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3357223.3362731

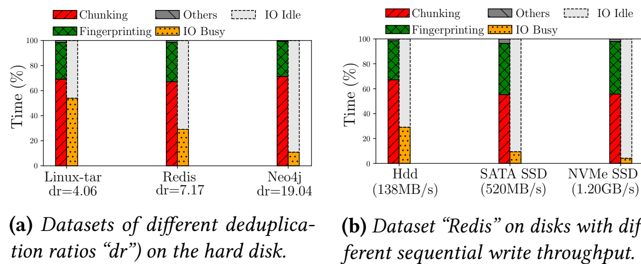
1 INTRODUCTION

With explosive growth of data volume and rising demand on high storage space efficiency and high performance, deduplication techniques have been widely deployed in various storage systems, including NetApp ONTAP system [20] and Dell EMC Data Domain [7].

In a storage system with deduplication technique deployed, when multiple pieces of input data share identical contents, only one copy is stored on the disk. To detect duplicates, the input data stream (often files) is first partitioned into pieces, or chunks. And accordingly the operation is called ***chunking***. For each data chunk, a collision-resistant hash function (e.g., SHA1) is applied on its contents to generate the chunk's fingerprint, an operation named ***fingerprinting***. Two chunks of data sharing a common fingerprint are considered as having identical contents. And one of the chunk is declared as a duplicate and is deduplicated. This is a common practice and widely used in real-world production systems [7, 20]. There are two metrics to assess a deduplication technique, namely deduplication ratio and deduplication speed.

The deduplication ratio is ratio of sizes of a data set before and after a deduplication operation. It measures a deduplication system's capability of detecting duplicates from the input data. For example, a deduplication ratio of 10 means 90% of the input data are redundant and can be kept from being stored on the disk.

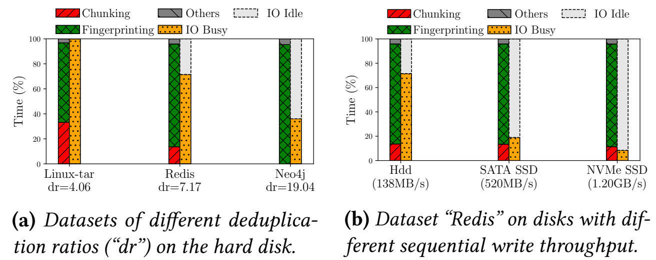
A deduplication technique's capability of detecting and removing redundancy mainly relies on its chunking method. For an input file, either fixed-size chunking (FSC) or content-defined chunking (CDC) methods can be used to partition it into chunks before their fingerprints are computed and compared to detect duplicates. With FSC, the file is partitioned into fixed-size chunks (e.g., 4KB) from its beginning regardless of the data contents. However, FSC suffers from the boundary-shift issue [18]. A small change (insertion or deletion) at the beginning of a stored file may keep almost all duplicate contents in the minimally changed file from being detected when it is written to the disk. CDC was proposed to addresses the issue by detecting chunk boundaries based on the file contents. With CDC, a chunk boundary is determined at a byte offset where the contents in the range between the previous boundary and the current offset satisfy a predefined chunking condition. This means as long as the contents in the ranges still satisfy the condition, the



(a) *Datasets of different deduplication ratios "dr") on the hard disk.*

(b) *Dataset "Redis" on disks with different sequential write throughput.*

Figure 1: Breakdown of CPU and disk times with ***regular CDC*** **deduplication of different datasets on different disks.**



(a) *Datasets of different deduplication ratios ("dr") on the hard disk.*

(b) *Dataset "Redis" on disks with different sequential write throughput.*

Figure 2: Breakdown of CPU and disk times with ***RapidCDC*** **deduplication of different datasets on different disks.**

chunk boundaries can be retained. Existing studies have shown that CDC can produce much higher deduplication ratio, often more than 10×, than FSC [29].

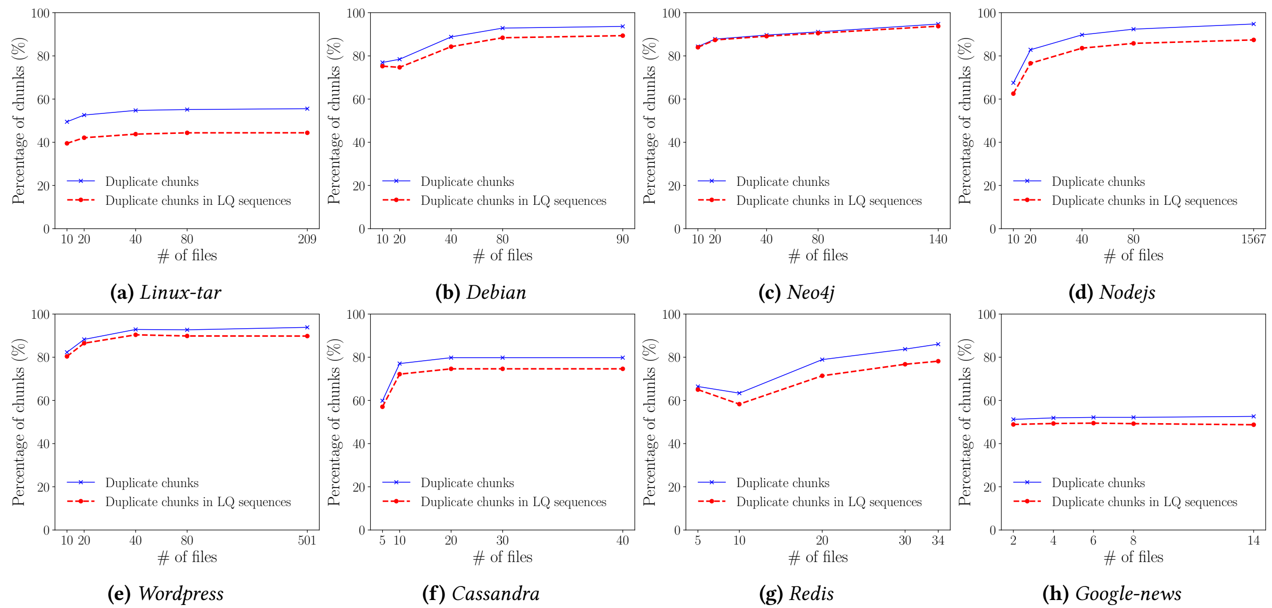
Although CDC-based deduplications may provide much higher deduplication ratio, its chunking process is very time-consuming. It essentially has to scan the entire file byte-by-byte to avoid missing any potential chunk boundary. Specifically, a common use of the CDC method is to roll a fixed-size window across the file. At a byte offset of the file it applies a hash function on the data covered in the window and compares the generated hash value to a predefined value. If matched, a chunk boundary is declared at the end of the window. In principle, the window stops at every byte for computing and comparing hash values. In reality, CDC needs to avoid a chunk that is too small for lower deduplication metadata overhead or that is too large for higher deduplication ratio. To this end, a minimum chunk size and a maximum chunk size are pre-determined. 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). It is noted that the hash function used for chunk boundary detection is different from the one used for fingerprinting. And it does not need to be collision-resistant.

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, expected average chunk sizes are 4KB, 12KB, and 8KB, respectively, about half of the bytes in a file are scanned [36]. As an example, for a storage system admitting data at the speed of 1GB/s the CDC deduplication subsystem must carry out around 500 million of such function computations per second so that the subsystem 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 deduplication on datasets of different deduplication ratios and residing on disks of different speeds. The datasets are described in Table 1. Detailed experiment setup is depicted in Section 4. In each experiment, duplication within each dataset is detected and removed by a CDC deduplication 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 overlapped as much as possible. As shown in Figure 1, the chunking phase consistently accounts for more than 60% of the CPU time in a deduplication system regardless of the input datasets, which limits the speed of data sent to the disk for storing and causes low utilization of the I/O bandwidth (about 30% for *Redis*) as shown in Figure 1(a) when the hard disk is used for data storage. The situation becomes worse when a fast SSD, which provides much higher write bandwidth, is used. As shown in Figure 1 (b), the utilization of the disk bandwidth drops to less than 10% for *Redis* when an NVMe SSD is used. The results show that for CDC-based deduplication systems, the system bottleneck is not at the disk but on the CPU, where the chunking operation contributes the most significant cost.

In the paper, we propose *RapidCDC*, a CDC acceleration technique that mostly removes the need of byte-by-byte window rolling in the determination of chunk boundaries. The technique represents a disruptive departure from existing chunking designs which have always deemed the byte-by-byte detection as necessary. RapidCDC leverages the duplicate locality, a phenomenon showing duplicate chunks are likely to appear together in the data, to quickly jump to potential chunk boundaries. In RapidCDC, a chunk's fingerprint is recorded along with the size of its next chunk in a file, which is used as a hint to locate the most likely boundary of the next chunk once a duplicate chunk is detected without byte-by-byte window rolling. Our evaluation results with synthetic and real-world datasets show that almost all of the chunking time can be removed for workloads of high deduplication ratios. Figure 2 show the results with RapidCDC, where chunking time is significantly reduced and the I/O bandwidths are better utilized.

The paper is organized as follows. In Section 2, we give a quantitative analysis of the duplicate locality, which has not yet been exploited for accelerating chunking speed, in real-world datasets. We then exploit the locality to design RapidCDC, which includes a suite of chunking techniques with different trade-offs between performance risk and gain (Section 3). In Section 4, we extensively evaluate the CDC deduplication strategies adopting these chunking techniques with synthetic and real-world datasets to assess improvements of the chunking speed and entire deduplication system's performance.



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

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

Figure 3: 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.

2 QUANTITATIVE ANALYSIS OF DUPLICATE LOCALITY

Duplicate data are often generated as a result of limited updates on existing data. 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 deduplication, this locality refers to the phenomenon that duplicate chunks are likely to stay together.

We are aware that duplicate locality has been identified and leveraged to improve various aspects of a deduplication system, including efforts on reducing the size of the in-memory chunk index [6, 15, 32] and on reducing file fragmentation and disk seeks for better read performance in a deduplication system [26]. However, the implication of the locality on improvement of chunking speed has not been considered at all. In particular, it's not well studied how the strength of the locality is affected by file updates, such as those in versions of software packages, which is a significant source of data duplication in backup storage systems. We address the issue in this section.

Because chunks are not very large in practice (usually tens of KBs) for high deduplication 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. There are two scenarios where the sequences' length is always 0. One is that there are not any duplicate chunks. And the other is that any duplicate chunk is isolated. To investigate the existence and strength of the locality, we examine the percentage of chunks in the sequences (Figure 3) and the sequence lengths' distribution (Figure 4) for a selected group of real-world datasets. As detailed in Table 1, the datasets 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 obtain the chunks. 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 deduplication 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 LQ sequences, and show their respective percentages over the total number of chunks at the time in Figure 3. The percentage of all duplicate chunks, shown as the the upper lines in the figures and named accordingly as ***upper percentage***, positively correlates to the deduplication ratio. In contrast, the percentage of chunks in the LQ sequences, named ***lower percentage***, exhibits the existence of duplicate locality. The gap between the two percentages indicates the percentage of isolated duplicates, or duplicates with zero duplicate locality. As we observe the two percentages for each data

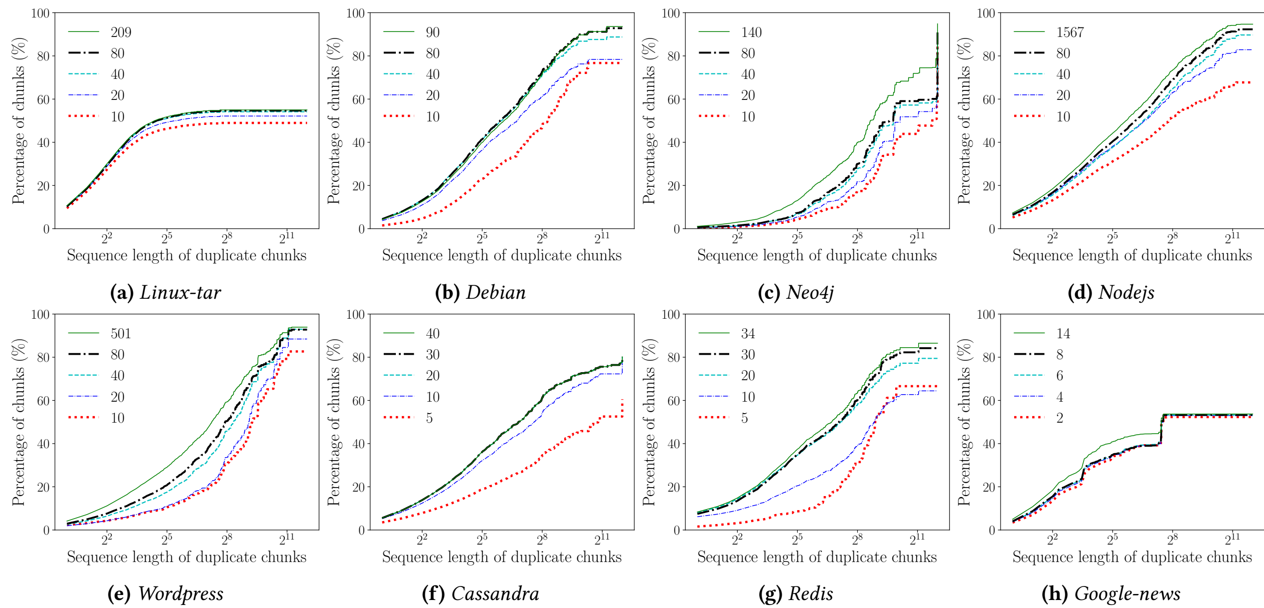


Figure 4: CDF (Cumulative Distribution Function) curves of LQ sequence lengths when a certain number of files in a dataset are admitted into the system. Numbers of currently admitted files are shown in the legend.

set when different number of files are admitted in Figure 3, their 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 deduplication 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 the length of the LQ sequence, we show the percentage of chunks that stay in an LQ sequence whose length is smaller than a certain threshold for different datasets at the time when different numbers of files are admitted in Figure 4. From the figure we can see that only a small percentage of chunks are in very short LQ sequences (e.g., those of 8 or fewer chunks). For some datasets of high deduplication 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. Across all the datasets and with various number of admitted files a majority of duplicate chunks stay in relatively 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 THE DESIGN OF RAPIDCDC

The key technique of RapidCDC is to exploit the duplicate locality in the datasets to enable a chunking method which detects chunk boundaries without a byte-by-byte window rolling. Due to the existence of the locality, immediately following a current deduplicateable chunk, denoted B_1 in Figure 5, in a file named $File_{new}$, the next chunk (B_2) is likely also to be a duplicate. The question is

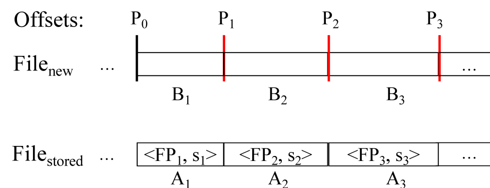


Figure 5: An illustration of the idea of RapidCDC for rapidly determining chunk boundaries. A_k and B_k, where k = 1, 2, or 3, are chunks. FP_k are fingerprints of chunk A_k. And s_k are size (in bytes) of chunk A_k.

where the end boundary of B_2 is. To this end, current CDC methods would take a window rolling over a potentially 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.

3.1 Quickly Reaching Next chunk's Boundary

Let's assume the fingerprint of Chunk B_1 in File $File_{new}$ matches Fingerprint FP_1, which is currently recorded and associated with a unique (physical) chunk of data in the storage system, that 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 on the disk, as illustrated in Figure 5. In a CDC deduplication system each file has a recipe recording the mapping between each of its logical chunks and its mapped physical chunk for rebuilding file content [9, 16, 25, 27–30]. 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 $File_{new}$ has the same fingerprint (and exact data content) as that of A_1 in $File_{stored}$, their respective next chunks, B_2 and A_2, are 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 of B_2's size, and directly move the rolling window on $File_{new}$ to the corresponding file offset, P_2, as shown in Figure 5. The key enabling operation is to obtain the size s_2 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 follow this relationship chain to obtain s_2 after learning that 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. Furthermore, some of its fingerprints may appear in other files' recipes. The pointers to the fingerprints in the file whose recipe will be removed may need to be adjusted. It can be expensive to make the adjustments in response to changes of recipes. While a fingerprint can be associated with multiple logical chunks, or multiple recipe records, it may be attached with multiple pointers. This can further increase the cost of maintaining the pointers.

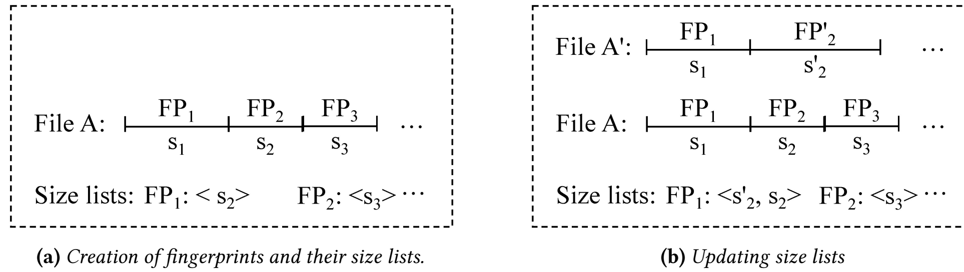
To address this issue, RapidCDC adopts a much simpler and more efficient method. The CDC-based deduplication process always starts from the beginning of a file and moves sequentially towards the end of the file. For any two consecutive chucks in a file, say A_1 and A_2, we record the size of A_2, say s_2, along with the fingerprint of A_1, say FP_1. In the example shown in Figure 5, 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 knowledge about recipes. Chunks with a certain fingerprint may appear in different files, and accordingly be followed with chunks of different sizes. Therefore, we allow a list of next-chunk sizes, named *size list*, to be attached to a fingerprint. 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. 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 last chunk boundary plus the minimum chunk size, or the position that a regular CDC would use after the last chunk boundary is detected. It then rolls the window byte-by-byte as the regular CDC does until a new duplicate chunk is found. Once a new duplicate chunk is found, RapidCDC's window jumps again attempting to take advantage of the 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.

3.2 Accepting Suggested Chunk Boundaries

After a duplicate chunk is detected, we retrieve the size list attached to its fingerprint, which provides hints for possible sizes of its next chunk, or its next chunk's possible end boundaries. We will check each of the sizes in the order in which they appear in the list 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. A RapidCDC deduplication scheme may incorporate any one of the criteria. And a suggested chunk boundary is accepted when it satisfies the criterion.

- FF (Fast-forwarding only). The suggested chunk boundary is always accepted without further checking at the file offset. This is the most aggressive criterion for fast chunking. However, it may cause loss of deduplication opportunities by choosing boundaries that do not satisfy the predefined chunking condition and thus produce unique chunks, which could hurt deduplication ratio. However, our experiments with real-world datasets indicate the risk is very low due to RapidCDC's ability of switching to the byte-by-byte detection whenever non-deduplicatable chunk is found (see Section 4.3).
- FF+RWT (Rolling Window Test). The suggested boundary is verified by checking the contents in the rolling window that ends at the suggested boundary position. As a regular CDC does, the hash value of the contents in the window is computed and compared to the pre-determined value. Only when they are equal and accordingly a valid chunk is formed, the boundary is accepted.
- FF+MT (Marker Test). Instead of computing 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 5 after duplicate chunks A_1 and B_1 are found.). The boundary is accepted if the two bytes are the same. This criterion requires recording last byte of a chunk along with its size in the size list. This marker-byte comparison needs only a few instructions, and is faster than the rolling window test.
- FF+RWT+FPT (Fingerprint Test). This is the most stringent criterion. After the boundary passes the rolling window test as that in FF+RWT, 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



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

Figure 6: Maintenance of size lists

duplicate chunk and can be readily deduplicated. Otherwise, it has a relatively high performance penalty.

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 6(a), File A is sent to the CDC deduplication 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 fingerprint pool, its size list is created. When the file's second chunk is determined, its size (s_2) is added to FP_1's size list as its first member. A list can grow. In Figure 6(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 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.

Figure 7 shows size lists after Files A and B are stored and before File C is stored. FP_1's list is $< s_2 >$ before File B is stored. After File B is stored, it becomes $< s'_2, s_2 >$, where the newer size value s'_2 is placed at the head. After File C is stored, the list becomes $< s_2, s'_2 >$ as s_2 is hit. A common scenario is that a file is incrementally updated generating a sequence of file versions, with each version resulting from updating of its previous one. Use of the LRU policy can maximize the chance of hit at the first size value in a list. As shown in Figure 7, due to existence of duplicate locality, after sufficient history access sequences are available (Files A and B), a new file (File C) can flexibly exploit the locality in multiple chunk sequences.

Keeping multiple size values in a size list can track different sequence patterns to minimize the probability of switching back to the window's slow movement mode. As the list is managed with LRU, a low-hit-ratio size value can be replaced out of the list following admitting a new value. However, the list should not be too long. In addition to potentially excessive space overhead, it may cause low-hit-ratio size values to 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, such as that for the FF+RWT+FPT criterion, can be substantial. Therefore, the list should be reasonably short to keep low-hit-ratio values out of the list.

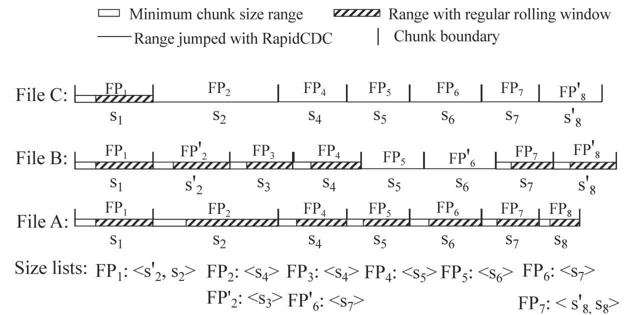


Figure 7: Use of size lists to accelerate CDC (the shown size lists reflect their contents after Files *A* and *B* are stored and before File *C* is stored.)

3.4 Likeness of chunks produced by RapidCDC and regular CDC

By opportunistically jumping to the next chunk boundaries, RapidCDC may produce chunk boundaries that are different from those produced by the regular CDC that always uses byte-by-byte window rolling. This does not compromise correctness of the deduplication storage system, as deduplication of any chunks requires matching of their fingerprints with those in the system. However, the difference of the chunk set produced by RapidCDC may impact the chance of successful matching, or the deduplication ratio. To understand the impact, we use the set of chunks produced by the regular CDC as the baseline and investigate the likeness of chunks produced by RapidCDC and regular CDC.

In the investigation, We would like to see the sequence of chunk boundaries produced by RapidCDC is (almost) the same as that by the regular CDC. Let's first assume use of the FF+RWT+FPT boundary acceptance criterion in RapidCDC. With this criterion,

RapidCDC does not add any fingerprints obtained in its fast forwarding mode to the system's fingerprint pool. That is, all fingerprints in the pool are for chunks identified by a regular rolling window test, the same as the regular CDC. Therefore, a chunk accepted by the FF+RWT+FPT criterion is exactly the same as that identified by the regular CDC. That is, if we use the FF+RWT+FPT criterion, we can guarantee RapidCDC produces the same sequence of chunks as the regular CDC. And the deduplication ratio of such a RapidCDC is the same as that of the regular CDC.

Now let's assume the FF+RWT criterion is adopted in RapidCDC to chunk a file. Suppose that the current chunk is a duplicate and ends at the boundary at P_1. And Suppose that the P_1 boundary is identified by both RapidCDC and the regular CDC. Further we assume that the next chunk boundary identified by the regular CDC is P_2. And the next chunk boundary identified by the RapidCDC under its fast forwarding mode is P_2'. Let's see how likely P_2' is different from P_2. First. P_2' cannot appear before P_2. Otherwise, the regular CDC would choose P_2', instead of P_2, as its next chunk boundary. Second, every size value in a RapidCDC's size list is obtained by window rolling. For the rolling window to produce a size from P_1 to P_2', the file content in the window at P_2 must be updated so that the window test at P_2 fails, and the test at P_2' must succeed. In the current file both tests at P_2 and P_2' succeed. This means the content of the window at P_2 is different from that where the size from P_1 to P_2' is obtained. And a new chunk demarcated by P_1 and P_2' is unlikely to be a duplicate chunk. Having a non-duplicate chunk will force RapidCDC to switch back to its regular byte-by-byte sliding window mode, and to produce chunk boundaries same as those by regular CDC. That is, chunks different from those produced by the window rolling are occasional and much less likely to see their duplicates. Furthermore, generation of such chunks discontinues once the byte-by-byte slow movement mode kicks in.

The analysis with the FF+MT criterion is similar. For the FF criterion, we will experimentally evaluate the impact of its aggressive chunking approach on the deduplication ratio.

4 EVALUATIONS

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

4.1 Experimental Setup

The Systems in Evaluation. In RapidCDC's implementation we choose the rolling-window-based CDC as its slow movement mode, which is used in production systems. The RapidCDC prototype can be configured with one of the four chunk boundary acceptance criteria (FF, FF+RWT, FF+MT, or FF+RWT+FPT). The default hash function applied on the content of the window to identify chunk boundaries is *Rabin* [23], 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 deduplication system implementations [14, 17, 18, 21, 33, 38]. The rolling window size is set to 48 bytes. In the evaluation, we also include a more lightweight hash, *Gear* [34]), to replace Rabin to reveal how the function's cost impacts RapidCDC's performance advantage. 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 [7]. In the evaluation we also evaluate a configuration with a larger range of chunk size (2KB/16KB/64KB), which is used in LBFS, a low-bandwidth network file system using the CDC-based deduplication technique to reduce network traffic [18]. The prototype uses SHA-1 to compute a chunk's fingerprint. The implementation has about 2400 lines of C code. By default, we use one thread. We also conduct experiments with RapidCDC using multiple threads. The default length of a fingerprint's size list is 2.

Each experiment uses a regular CDC-based deduplication system (denoted as *regular* in figures) as a counterpart of the RapidCDC system for comparison. All systems in a comparison are configured the same except stated otherwise, including choices of window size, the hash function, and chunk size. 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 WDC WD1003FZEX-00K3CA0 with sequential write and read bandwidths of 138MB/s and 150MB/s, respectively.

The Datasets. The datasets for the evaluation include a series of synthetic datasets and eight real-world datasets as described in Table 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 insert, delete, or overwrite. Modifications are randomly distributed within either an entire file or a selected region of the file. 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 1. The deduplication ratios shown in the table are obtained by applying the aforementioned regular CDC deduplication with its default configuration. As each dataset consists of a sequence of files, in an experiment we write the files in a sequence, one at a time, to a deduplication system and measure the deduplication speed and amount of data that can be deduplicated within the dataset itself.

4.2 Results with Synthetic Datasets

In a synthetic dataset, starting from the second one in the sequence of 10 files a chunk in a file receives at most one 100-byte modification unless stated otherwise.

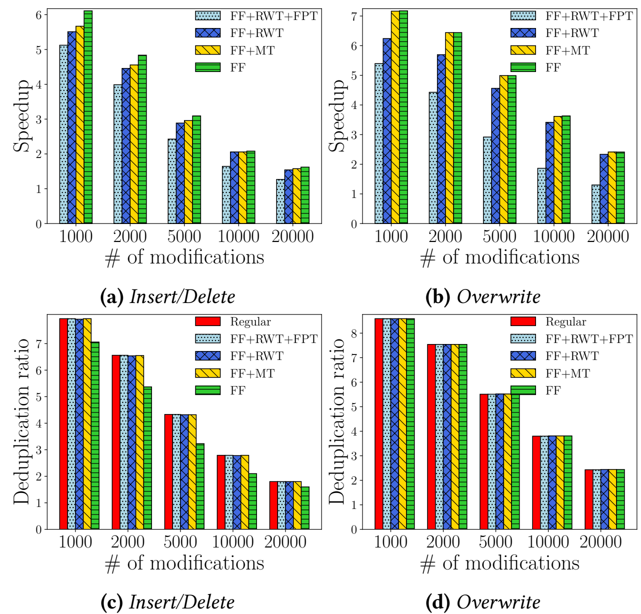
Impact of Modification Count and Distribution. Modifications can be categorized into two types. One may cause chunk boundary shift, including *insert* and *delete*. The other (*overwrite*) does not shift the boundary, and only changes a chunk's content. To generate a new version of file, we choose a 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 8 shows the chunking speed (reciprocal of total chunking time) and deduplication ratio on a dataset

Table 1: Real-world datasets used in the experiments. All the Docker images are downloaded from Docker Hub [8].

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 [5].
Redis	100.4	34	7.2	Docker images of the Redis key-value store database [24].
Debian	9.5	92	15.8	Docker images of Debian Linux distribution (since Ver. 7.11) [11].
Neo4j	46.0	140	19.0	Docker images of neo4j graph database [19].
Wordpress	181.7	501	22.0	Docker images of WordPress rich content management system [13]
Nodejs	800.0	1567	41.4	Docker images of JavaScript-based runtime environment packages [12]

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 deduplication 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 deduplication ratio. Because RapidCDC quickly switches to the slow window movement mode to look for the next valid 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 (almost) the same deduplication 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 deduplication ratio.

Understandably the deduplication ratio decreases with the increase of modification count. As RapidCDC takes advantage of duplicate sequences, which tend to become fewer and shorter when modifications increase, its chunking speedup is accordingly reduced. However, the speedup is always positively correlated with its corresponding deduplication ratio, and is very close to the ratio. For example, for Insert/Delete and the FF RapidCDC, the deduplication 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 deduplication 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 9. 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.

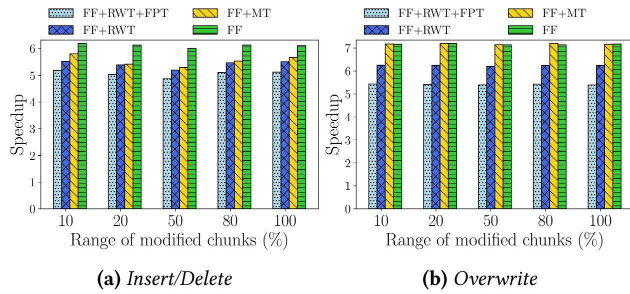


(a) *Insert/Delete* **(b)** *Overwrite*

(c) *Insert/Delete* **(d)** *Overwrite*

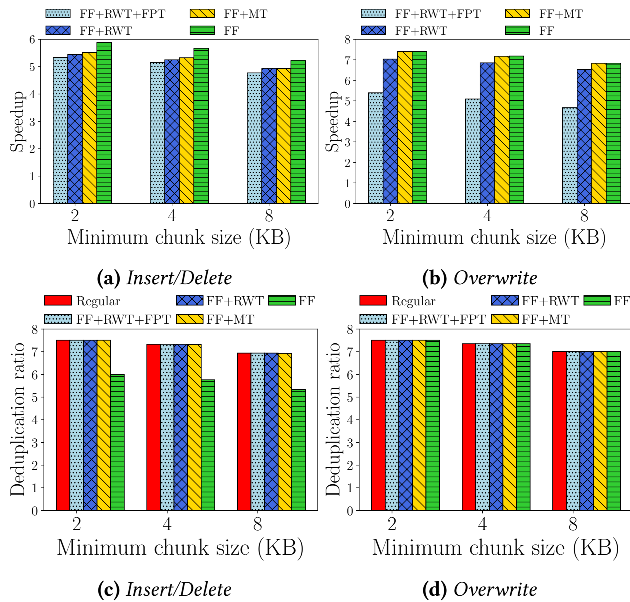
Figure 8: Chunking speed and deduplication ratio for datasets with different numbers of modifications spread over an entire file.

Impact of Minimum Chunk Size and Hash Function. During chunking operations of a regular CDC, a rolling window may also enter its fast-forwarding mode by skipping the minimum chunk size of bytes immediately after a detected chunk boundary. This optimization on the window rolling may have an impact on RapidCDC's relative benefit. To this end, we vary the minimum chunk size during the deduplication of a dataset with 1000 insert/delete modifications randomly distributed in a file. To allow the minimum size to change in a larger range, we adopt a 2KB/16KB/64KB configuration, instead of the 4KB/8KB/12KB default one, as minimum/expected average/maximum chunk sizes. Figure 10 shows that the impact is small. With a decent deduplication ratio, RapidCDC has removed most of the chunking time, leaving only a small number of window



(a) *Insert/Delete* **(b)** *Overwrite*

Figure 9: Chunking speedups for datasets where 1000 modifications are applied to different range of a file.



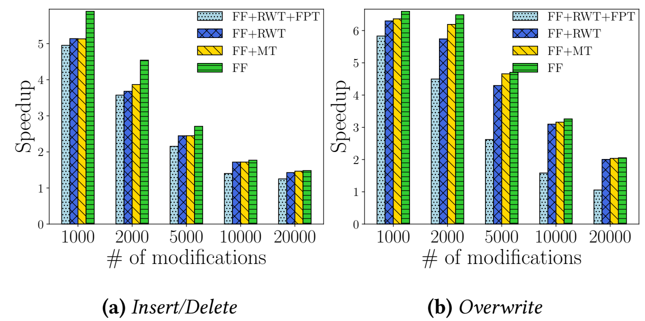
(a) *Insert/Delete* **(b)** *Overwrite*

(c) *Insert/Delete* **(d)** *Overwrite*

Figure 10: Chunking speedup and deduplication ratio with different minimum chunk sizes.

rollings in the slow movement mode. Therefore, the acceleration due to the use of a larger minimal chunk size does not take away much of RapidCDC's relative advantage. Furthermore, the small reduction of speedups are correlated to that of deduplication ratio. A larger minimum size leads to large chunk sizes, which tends to reduce deduplication ratio. RapidCDC's speedup becomes smaller with fewer duplicate chunks.

Another factor likely impacting RapidCDC's speedup is the rolling function for detecting chunk boundaries. Because the function is used at almost every rolling window position in a regular CDC, a faster function can accelerate the chunking process. The Gear [34] function used in FastCDC [34] can be 3× faster than Rabin. Figure 11 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 8). 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,



(a) *Insert/Delete* **(b)** *Overwrite*

Figure 11: Chunking speedups of RapidCDC on datasets with different modification counts and a faster hash function, ***Gear.***

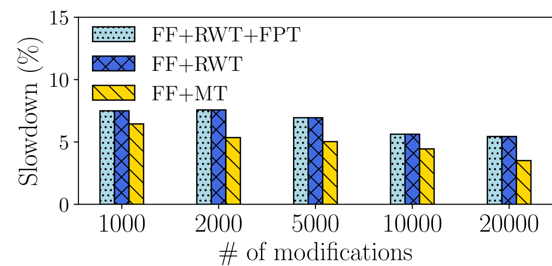


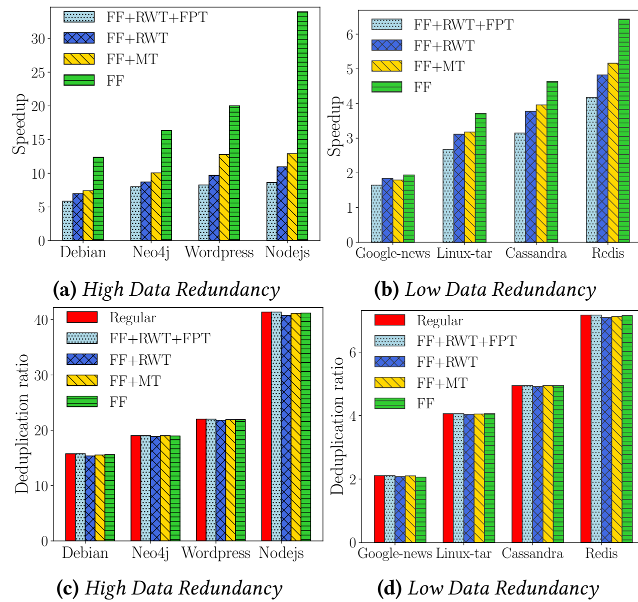
Figure 12: Chunking speed slowdowns when all suggested next-chunk sizes are misses in RapidCDC.

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 accelerating CDC process, including chunking method optimization like FastCDC [35] that uses the Gear function.

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 8(a). The chunking speed slowdowns over the regular CDC are shown in Figure 12. Compared to tens of times speedups observed in realistic datasets as shown in Section 4.3, the less-than 10% slowdowns are insignificant. With more modifications and accordingly lower deduplication ratios, there are fewer size values to be checked and the slowdowns are smaller. Understandably such datasets are unlikely to occur. And even the small slowdowns due to the hypothetical dataset can hardly materialize.

4.3 Results with Real-world Datasets

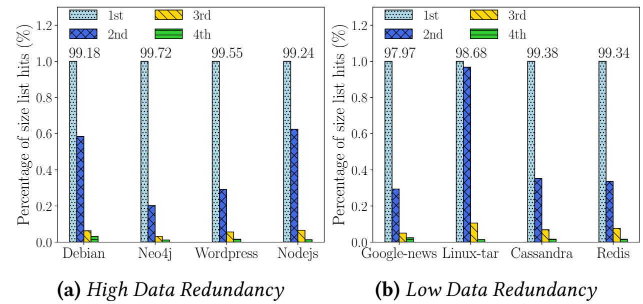
We categorize the eight real-world datasets in Table 1 into two groups according to their redundancy quantified by deduplication ratios: a high data redundancy group (Debian, Neo4j, Wordpress,



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

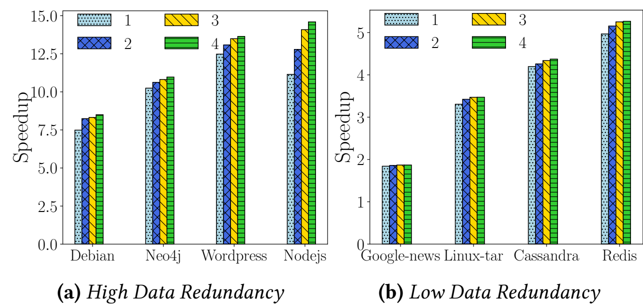
(c) *High Data Redundancy* **(d)** *Low Data Redundancy*

Figure 13: Chunking speedup and deduplication ratio for real-world datasets.



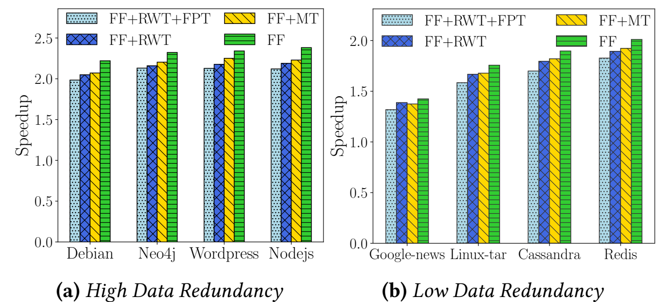
(a) *High Data Redundancy* **(b)** *Low Data Redundancy*

Figure 14: Hit ratios of positions in the size lists under the FF+RWT criterion. The first position's ratio is marked above its bar, which does not scale.



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

Figure 15: Chunking speedups for real-world datasets with RapidCDC/FF+RWT of different lengths of the size list.



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

Figure 16: Speedups of CPU computation (Chunking+fingerprinting) for real-world datasets.

and Nodejs) and a low data redundancy group (Google-news, Linux-tar, Cassandra, and Redis), and use them to evaluate RapidCDC's deduplication ratio and performance.

RapidCDC Performance Impact on Chunking and the Entire deduplication System. Figure 13 shows RapidCDC's chunking speedups and deduplication ratios with the datasets. For datasets of high redundancy, the deduplication ratio can be about 40. But only RapidCDC with FF has speedups close to the high deduplication ratios. Speedups of the other alternatives are much lower, especially for datasets of very high deduplication ratios, such as Nodejs. 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 the slow movement mode upon a non-acceptance. Only FF mostly removes the costs and reaches the high ratio. Another interesting observation is that FF does not compromise the deduplication 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 under the FF+RWT criterion. A hit means the size at the position is accepted. The results are shown in Figure 14. The first position has a very high hit ratio. For example, for the Debian dataset, the ratio is 99.18%, which means that in almost all cases the first sizes are accepted. FF accepts the first sizes without any checking, and fortunately it indeed makes the right decision and does not sacrifice deduplication ratio.

As 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 15 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 loss of 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 set at 2.

To understand the impact of RapidCDC's increased chunking speed on the CPU computation, we present speedups of the computation, whose two major components are chunking and calculation of fingerprints, in Figure 16. 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 the overall performance of a deduplication storage system, we present throughput of the deduplication system when various real-world datasets are written to the system in Figure 17. 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 with 500GB capacity, whose write and read bandwidth is 520MB/s and 550MB/s, respectively. The NMVe SSD is Intel SSDPEDMW012T4 of 1.2TB and has write and read bandwidths of 1.2GB/s and 2.4GB/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) deduplication system with the 8KB chunk size, results of the regular CDC, and results of a hypothetical deduplication system. The hypothetical system represents an *ideal* CDC-based deduplication implementation, in which chunking time is completely removed. In this case, we obtain chunking boundaries offline.

There are several interesting observations from Figure 17. First, the disk can be a performance bottleneck when it has a low bandwidth and the dataset has a low deduplication ratio that increases I/O bandwidth demand. Figure 17(d) shows the datasets' deduplication ratios. For FSC and RapidCDC, when their deduplication ratios are low (e.g., for Google-news), using faster disks (SATA SSD and then NVMe SSD) increases the throughput. However, with a higher deduplication ratio, the bottleneck shifts to the CPU, and faster disks do not lead to higher throughput. For example, RapidCDC has a deduplication ratio of 7.2 on Redis and its throughput increases minimally. However, FSC's deduplication ratio is 1.9, and its throughput keeps increasing (from 360MB/s, 745MB/s, to 760MB/s). Second, with a higher deduplication ratio, RapidCDC's advantage over the regular CDC becomes increasingly higher, because the regular CDC's chunking speed is not correlated with the ratio. Its throughput is about 2.3× as high as that of the regular CDC with Nodejs. Third, with low deduplication ratio and fast disks, such as Google-news or 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 deduplication ratio, its chunking speed can be close to that of FSC. While its deduplication ratio is usually much higher than that of FSC, its throughput is higher than FSC's in most cases. Fourth, once the deduplication ratio is high or a fast disk is used, RapidCDC's throughput is close to that of the *ideal* system as the bottleneck is on the CPU and RapidCDC removes most of the chunking cost.

Throughput of Multi-threaded RapidCDC. We implement a multi-threaded RapidCDC, each thread working on a different file in a dataset. Being challenged by CDC's high cost, researchers have proposed methods to use multi-core or even GPU to accelerate the operation [2, 3, 10, 31, 33]. 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 18 shows the chunking throughput (excluding disk I/O and fingerprinting operations) of RapidCDC with FF and different thread counts. As shown, the performance scales well and the throughput increases almost linearly. For datasets with high redundancy, the throughput can reach tens of GB/s, making the chunking hardly be a performance bottleneck and making it be a completely addressed issue.

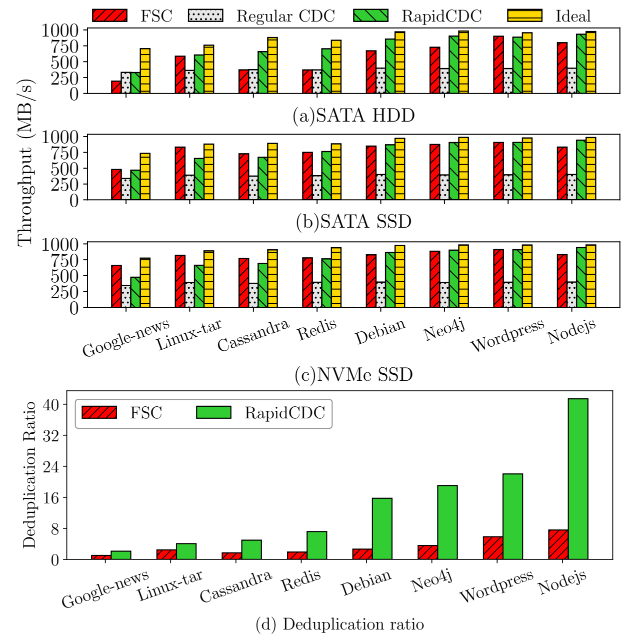
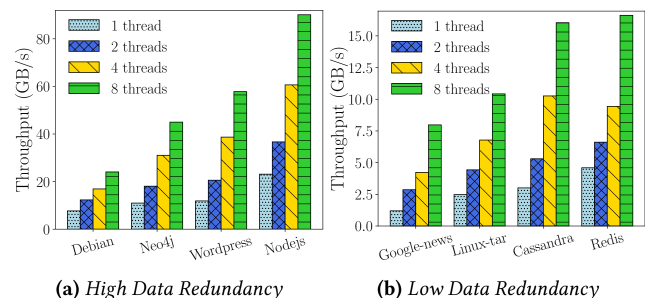


Figure 17: Throughput and deduplication ratio of the deduplication systems.



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

Figure 18: Chunking throughput with different number of threads, each on a different core.

5 RELATED WORK

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

Reducing Computation Cost in Chunking. In 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 as reported in literature [14, 17, 18, 21, 33, 38] and for production systems [7]. While chunking becomes a performance bottleneck of a CDC-based deduplication system, many techniques have been proposed to reduce its cost. SimpleByte was designed to provide fast chunking to eliminate fine-grained redundant data transmitted across networks [1]. 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 storage systems as its condition to form a chunk boundary is too weak and will produce very small chunks, which can significantly increase metadata management overhead. Gear [34] is a lightweight hash function requiring only one bit-shift, one add, and one table lookup in a hash computation. As reported, the Gear-based chunking can be 3× as fast as the Rabin-based chunking [35]. Some issues with the Gear function, such as generating too small chunks, are addressed in the FastCDC deduplication design [35]. Instead of applying a hash function on a window of bytes to obtain a hash value, MAXP [4] and AE [37] 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 [36]. A lightweight function is applied first to check whether a condition is met. Only when it is satisfied is the second function executed. All the efforts were 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.

Accelerating Chunking with Parallelism. StoreGPU [2, 10] and Shredder [3] leverage GPUs to accelerate the chunking and fingerprinting process 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 to achieve high throughput [33]. 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 to be generated as that from the sequential CDC [31]. SS-CDC was proposed to use Intel-AVX instructions to accelerate CDC, which guarantees the produced chunk boundaries will be identical to those with regular sequential CDC [22]. All these parallel chunking efforts requires additional or specialized hardware supports, which adds extra cost to existing storage systems or may not be available. Furthermore, introduction of parallel hardware for CDC acceleration may introduce substantial software development cost. And existing deduplication systems may need to be redesigned to optimize the communication and to synchronize multiple chunking instances. RapidCDC can remove most of CDC cost, making chunking a lightweight operation and making its parallelization essentially unnecessary.

Exploitation of Locality in Deduplication. Locality in workloads has been recognized and exploited to improve various aspects of a deduplication system. A sparse indexing structure has been proposed to reduce the in-memory chunk index by taking advantage of the locality of duplicate chunks [15]. 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 speed up the index lookup in a deduplication system [6]. 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 deduplication ratio and makes large chunks possible, and accordingly reduces amount of index structure [32]. To address the issue of file fragmentation in an FSC deduplication system, iDedup [26] 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 high availability 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.

6 CONCLUSION

In the paper we propose RapidCDC, a chunking technique for CDC-based deduplication systems, that can dramatically reduce the chunking time. While chunking has been well recognized as a major performance bottleneck in a CDC-based deduplication system and substantial efforts have been made to reduce its cost, RapidCDC represents a departure from existing optimization techniques for reducing the chunking time. It uniquely 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-based deduplication system with minor effort, which is to retrieve a suggested next-chunk size for every matched fingerprint and try to use it for locating the next valid chunk boundary. Furthermore, its benefit is orthogonal to other optimizations made in a deduplication system, including those for improving chunking performance.

We prototype RapidCDC and conduct extensive experiments comparing its chunking performance and deduplication ratio with other deduplication systems. The results show that it can provide up to 33× chunking speedup, which essentially removes the chunking performance bottleneck for datasets of high data redundancy. Meanwhile, it maintains (almost) the same deduplication ratio as regular CDC systems.

The source code of our RapidCDC implementation is available at *https://github.com/moking/RapidCDC*.

7 ACKNOWLEDGMENTS

We are grateful to the paper's shepherd Dr. Xiangyao Yu and anonymous reviewers who helped to improve the paper's quality. This work was supported by US National Science Foundation under Grants CNS-1527076 and CCF-1815303.

REFERENCES

[1] Bhavish Aggarwal, Aditya Akella, Ashok Anand, Athula Balachandran, Pushkar Chitnis, Chitra Muthukrishnan, Ramachandran Ramjee, and George Varghese. 2010. EndRE: An End-system Redundancy Elimination Service for Enterprises. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation (NSDI'10)*. USENIX Association, Berkeley, CA, USA, 28–28. http://dl.acm.org/citation.cfm?id=1855711.1855739

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

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

[4] Nikolaj Bjrner, Andreas Blass, and Yuri Gurevich. 2010. Content-dependent Chunking for Differential Compression, the Local Maximum Approach. *J. Comput. Syst. Sci.* 76, 3-4 (May 2010), 154–203. https://doi.org/10.1016/j.jcss.2009.06.004

[5] Apache Cassandra. 2014. Apache cassandra. *Available online at http://planetcassandra.org/what-is-apache-cassandra* (2014), 13.

[6] Biplob K Debnath, Sudipta Sengupta, and Jin Li. 2010. ChunkStash: Speeding Up Inline Storage Deduplication Using Flash Memory.. In *USENIX annual technical conference*. 1–16.

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

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

[9] Kave Eshghi, Mark Lillibridge, Lawrence Wilcock, Guillaume Belrose, and Rycharde Hawkes. 2007. 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)*. USENIX Association, Berkeley, CA, USA, 22–22. http://dl.acm.org/citation.cfm?id=1267903.1267925

[10] Abdullah Gharaibeh, Samer Al-Kiswany, Sathish Gopalakrishnan, and Matei Ripeanu. 2010. A GPU Accelerated Storage System. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing (HPDC '10)*. ACM, New York, NY, USA, 167–178. https://doi.org/10.1145/1851476.1851497

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

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

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

[14] Himshai Kambo and Bharati Sinha. 2017. 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*. IEEE, 400–404.

[15] Mark Lillibridge, Kave Eshghi, Deepavali Bhagwat, Vinay Deolalikar, Greg Trezise, and Peter Camble. 2009. Sparse Indexing: Large Scale, Inline Deduplication Using Sampling and Locality. In *Proccedings of the 7th Conference on File and Storage Technologies (FAST '09)*. USENIX Association, Berkeley, CA, USA, 111–123. http://dl.acm.org/citation.cfm?id=1525908.1525917

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

[17] Dirk Meister and André Brinkmann. 2009. Multi-level Comparison of Data Deduplication in a Backup Scenario. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference (SYSTOR '09)*. ACM, New York, NY, USA, Article 8, 12 pages. https://doi.org/10.1145/1534530.1534541

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

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

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

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

[22] Fan Ni, Xing Lin, and Song Jiang. 2019. SS-CDC: A Two-stage Parallel Content-defined Chunking for Deduplicating Backup Storage. In *Proceedings of the 12th ACM International Conference on Systems and Storage (SYSTOR '19)*. ACM, New York, NY, USA, 86–96. https://doi.org/10.1145/3319647.3325834

[23] Michael O Rabin. 1981. Fingerprinting by random polynomials. *Technical report* (1981).

[24] Salvatore Sanfilippo and Pieter Noordhuis. 2015. Redis. *http://redis.io* (2015).

[25] Philip Shilane, Grant Wallace, Mark Huang, and Windsor Hsu. 2012. Delta Compressed and Deduplicated Storage Using Stream-Informed Locality.. In *HotStorage*.

[26] Kiran Srinivasan, Tim Bisson, Garth Goodson, and Kaladhar Voruganti. 2012. iDedup: Latency-aware, Inline Data Deduplication for Primary Storage. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST'12)*. USENIX Association, Berkeley, CA, USA, 24–24. http://dl.acm.org/citation.cfm?id=2208461.2208485

[27] Niraj Tolia, Michael Kozuch, Mahadev Satyanarayanan, Brad Karp, Thomas C Bressoud, and Adrian Perrig. 2003. Opportunistic Use of Content Addressable Storage for Distributed File Systems.. In *USENIX Annual Technical Conference, General Track*, Vol. 3. 127–140.

[28] Cristian Ungureanu, Benjamin Atkin, Akshat Aranya, Salil Gokhale, Stephen Rago, Grzegorz Calkowski, Cezary Dubnicki, and Aniruddha Bohra. 2010. 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)*. USENIX Association, Berkeley, CA, USA, 17–17. http://dl.acm.org/citation.cfm?id=1855511.1855528

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

[30] Jiansheng Wei, Hong Jiang, Ke Zhou, and Dan Feng. 2010. 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)*. IEEE Computer Society, Washington, DC, USA, 1–14. https://doi.org/10.1109/MSST.2010.5496987

[31] Youjip Won, Kyeongyeol Lim, and Jaehong Min. 2015. MUCH: Multithreaded Content-Based File Chunking. *IEEE Trans. Comput.* 64, 5 (2015), 1375–1388.

[32] Wen Xia, Hong Jiang, Dan Feng, and Yu Hua. 2011. 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)*. USENIX Association, Berkeley, CA, USA, 26–28. http://dl.acm.org/citation.cfm?id=2002181.2002207

[33] Wen Xia, Hong Jiang, Dan Feng, and Lei Tian. 2012. 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 (FASTâĂŹ12 Poster)*. 1–2.

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

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

[36] Chuanshuai Yu, Chengwei Zhang, Yiping Mao, and Fulu Li. 2015. Leap-based content defined chunkingâĂŤtheory and implementation. In *Mass Storage Systems and Technologies (MSST), 2015 31st Symposium on*. IEEE, 1–12.

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

[38] Benjamin Zhu, Kai Li, and Hugo Patterson. 2008. Avoiding the Disk Bottleneck in the Data Domain Deduplication File System. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST'08)*.