Vectorized Sequence-Based Chunking for Data Deduplication

Sreeharsha Udayashankar †, Ali Assem Mahmoud §†, Samer Al-Kiswany †
† *University of Waterloo* § *National Research Council of Canada*
{s2udayas, ali.mahmoud, alkiswany}@uwaterloo.ca

***Abstract*—Data deduplication has gained wide acclaim as a mechanism to improve storage efficiency and conserve network bandwidth. Its most critical phase, data chunking, is responsible for the overall space savings achieved via the deduplication process. However, modern data chunking algorithms are slow and compute-intensive because they scan large amounts of data while simultaneously making data-driven boundary decisions.**

We present SeqCDC, a novel chunking algorithm that leverages lightweight boundary detection, content-defined skipping, and SSE/AVX acceleration to improve chunking throughput for large chunk sizes. Our evaluation shows that SeqCDC achieves $10\times$ higher throughput than unaccelerated and $1.2\times$–$1.35\times$ higher throughput than vector-accelerated data chunking algorithms while minimally affecting deduplication space savings.

***Index Terms*—Data storage, Data deduplication, SIMD, Cloud computing**

I. Introduction

Data generation rates have skyrocketed in recent years, leading to the explosion of the amount of data stored on the cloud [1]. Cloud storage providers employ numerous mechanisms to deal with this data influx, such as distributed file systems [2], [3], novel storage architectures [4], [5], data compression [6], [7] and data deduplication [8], [9].

Data deduplication has been widely employed in production by cloud storage providers such as Microsoft [10], EMC [11] and IBM [12]. A large percentage of the data stored by these providers is redundant [10]. Data deduplication helps identify and eliminate these redundant portions, reducing storage costs by up to 80% [11], [13]. Deduplication is performed at the *chunk-level*, after dividing files into *chunks* [14].

The division of files into chunks is achieved using data chunking algorithms [15], which dictate the space savings achieved by the deduplication system as a whole. Data chunking algorithms fall into two categories: fixed-size and content-defined chunking (CDC). Fixed-size chunking divides files into chunks of an equal pre-specified size. While this approach has been used by traditional backup systems such as Venti [13] and OceanStore [16], it achieves poor space savings due to its vulnerability to insertions and deletions (*byte-shifting*) [15].

To mitigate byte-shifting, modern deduplication systems instead resort to Content-Defined Chunking (CDC) algorithms [15], [17]–[22]. These algorithms make data-driven boundary decisions using the file's contents, effectively handling the byte-shifting problem. Numerous data chunking algorithms are in use today and can be broadly divided into hash-based [15], [19], [20], [22] and hashless algorithms [17], [21], [23]. Hash-based algorithms use rolling hash functions to derive chunk boundaries, while hashless algorithms treat each byte as a value and derive chunk boundaries using conditions based on local minima or maxima. Note that in either case, a *fingerprint* is generated using collision-resistant hash algorithms [24] after a chunk boundary is identified.

CDC algorithms suffer from four limitations that impact their throughput. First, they rely on expensive rolling hash functions [15], [20] and minima-maxima searches [17], [21] to determine chunk boundaries. Second, they sequentially scan the ingested data stream. This reduces throughput and increases end-to-end processing time, as real systems store terabytes of data. Third, they fail to utilize the SIMD capabilities of modern CPUs to accelerate data processing. Finally, they are designed to target datasets that benefit from smaller chunks of size 512 B – 4 KB. Such small chunks increase metadata overhead [8] and impact system throughput due to the random access and frequent transfer of small chunks.

State-of-the-art CDC techniques have aimed to solve some of these shortcomings. FastCDC [19] replaced the expensive Rabin's hash algorithm [15] with Gear hashing [20] to reduce boundary-detection overhead. AE [17] and RAM [21] reduce the overhead by avoiding hashing entirely, instead relying on minima/maxima to determine chunk boundaries. SS-CDC [18] and our previous work, VectorCDC [25], accelerate CDC algorithms with SSE/AVX instructions [26] offered by modern CPUs. However, these studies only address a few specific limitations and fail to do so effectively.

We present SeqCDC, a novel CDC algorithm that comprehensively addresses the limitations of modern CDC algorithms. SeqCDC uses three optimizations to improve chunking throughput: *lightweight boundary judgment*, *content-based data skipping*, and *vector acceleration*. Lightweight boundary judgment reduces boundary detection overhead by using monotonically increasing/decreasing sequences, avoiding complex hashing and minima-maxima searches (§III-A). To avoid scanning the entire source data, SeqCDC skips scanning selective data regions. However, to minimize the impact on deduplication efficiency, data skipping is regulated using content-based heuristics i.e. content-based skipping (§III-C). SeqCDC has been designed with vector acceleration in focus, and uses SSE/AVX instructions [26] to improve chunking throughput (§III-D). Finally, SeqCDC scales its throughput with chunk size, i.e., it offers higher throughput at the larger chunk sizes favored by deduplication systems (§VI-B).

Our evaluation compares SeqCDC to seven unaccelerated

and three vector-accelerated chunking algorithms using a variety of real-world datasets (§VI). We show that SeqCDC improves chunking throughput by 10× over unaccelerated and 1.2×–1.35× over vector-accelerated CDC algorithms, while achieving comparable deduplication space savings. Our code is publicly available with DedupBench[1] [27].

II. Background and Motivation

Data deduplication [10], [15] is used by cloud storage providers to detect duplicate data, allowing them to eliminate the costs associated with its storage and transmission. Data deduplication consists of the following steps [8]:

- *File Chunking:* Splitting a file into chunks using a data chunking algorithm is one of the primary steps in data deduplication. Deduplicating these chunks provides more space savings than file-level deduplication [15].
- *Chunk Hashing:* Each data chunk is hashed using a collision-resistant hashing algorithm, such as SHA-256 [24] or MD5 [28], to obtain a fingerprint.
- *Fingerprint Comparison:* The fingerprint is compared against a database of previously observed fingerprints. A duplicate fingerprint, i.e., one observed before, indicates an underlying duplicate chunk, which can be eliminated.
- *Data Storage:* Non-duplicate data chunks are saved on the storage medium, and their fingerprints are added to the fingerprint database.

Chunking is a critical part of this pipeline; it occurs on the critical path during data uploads and directly impacts the overall space savings and throughput associated with deduplication systems. Space savings represent the total space conserved by using deduplication, measured as:

$$Space\ savings = \frac{Original\ Size - Deduplicated\ Size}{Original\ Size} \tag{1}$$

The size of the fingerprint database is tied to the average chunk size. Smaller average sizes lead to more chunks and more fingerprints, thus increasing the size of the database and associated fingerprint comparison overheads. To minimize this overhead, systems in production favor larger chunk sizes.

A. Content-Defined Chunking (CDC) Algorithms

While fixed-size chunking has been used before [13], it is vulnerable to byte shifting, achieving poor space savings [15]. Content-Defined Chunking (CDC) algorithms [17]–[23] instead determine chunk boundaries based on data contents.

These algorithms slide a fixed-size window over the data within the source file. When the window's data meets pre-specified conditions, they insert a *chunk boundary* at the end of it. By repeating this across the entire file, they divide it into chunks. Each CDC algorithm has tunable parameters to change the average size of generated chunks. CDC algorithms can be classified into hash-based and hashless algorithms [27].

[1] https://github.com/UWASL/dedup-bench

Hash-based chunking algorithms, such as Rabin's chunking [15], insert chunk boundaries only when the hash value of the window's data matches a pre-specified mask. The hashing algorithms used here are typically not collision-resistant. For instance, with Rabin's Chunking (RC) [15], chunk boundaries are inserted when the lower order 13 bits of the hash value are zero. While Rabin's chunking achieves high deduplication ratios, it is very slow. TTTD [22] uses Rabin's hashing but improves space savings by simultaneously checking for two hash value conditions. The secondary condition is always computed but only used if a boundary is not found beyond a pre-specified size with the primary condition.

The CRC [18] and Gear [20] hashing functions have lower overheads than Rabin's hashing. FastCDC (FCDC) [19] uses Gear hashing and implements two optimizations to improve chunking throughput: sub-minimum skipping and chunk size normalization. Sub-minimum skipping skips scanning data up to the minimum chunk size at the beginning of each chunk. Chunk size normalization dynamically relaxes the boundary condition to ensure that generated chunks are close to the expected average chunk size.

Hashless algorithms such as Asymmetric Extremum (AE) [17] and Rapid Asymmetric Maximum (RAM) [21] also slide fixed-size windows over the source data. AE attempts to identify a window such that the starting byte's value is greater than all the bytes before it and not less than the other bytes within the window. When such a window is found, AE inserts a chunk boundary at the end of the window. AE is 4−5× faster than Rabin's chunking. Rapid Asymmetric Maximum [21] inserts chunk boundaries when a byte is found outside the window whose value is greater than the maximum-valued byte within the window. These algorithms avoid hashing when determining chunk boundaries, reducing boundary-detection overhead and achieving high chunking throughput [25], [29].

B. Chunking Throughput Analysis

These existing CDC algorithms suffer from high overheads during boundary detection. Hash-based algorithms [15], [19], [22] rely on *rolling hashes* to detect boundaries, while hashless algorithms rely on maximum/minimum values identified by an *Extreme Byte Search* [25], paired with *Range Scans*.

Boundary detection overhead. Figure 1 shows the total time spent in different phases across the deduplication pipeline when chunking 30 GB of random data. We use the Intel Icelake machine described in §VI for this experiment. We use MurmurHash3 [30] as the fingerprinting algorithm which generates a 128-bit digest, similar to MD5 [28]. We target an average chunk size of 16 KB for all CDC algorithms.

Figure 1 shows that the total time spent in the fingerprinting phase is only $2-3\%$ across all CDC algorithms, indicating that data chunking is the main bottleneck in the pipeline. Hash-based algorithms FastCDC [19], Rabin-Karp [15], and TTTD [22] spend $57-59\%$ of their time in the rolling hash phase and $38-39\%$ to check whether the generated hash matches the boundary condition. On the other hand, the hashless algorithm RAM [21] spends 96.32% of its time on extreme byte searches while another 0.8% is spent in the range scan phase to detect

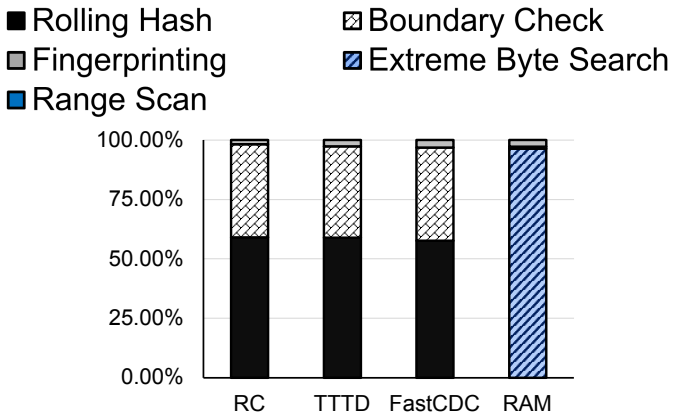


Fig. 1: Percentage of time spent by CDC algorithms in different phases while deduplicating 30 GB of randomized data.

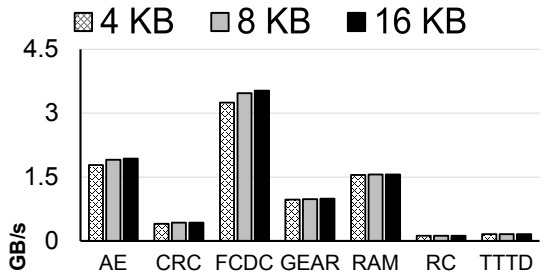


Fig. 2: Chunking throughput on randomized data across different chunk sizes

boundaries. These results show the high overheads involved in rolling-hashes and minima/maxima searches, motivating the need for a lightweight boundary detection condition.

Large chunk sizes. As noted above, deduplication systems in production prefer a smaller number of larger size chunks to minimize fingerprinting overheads. However, state-of-the-art CDC algorithms are designed for smaller chunk sizes, i.e., their throughput remains constant across chunk sizes.

Figure 2 shows the throughput achieved by AE [17], CRC [18], FastCDC (FCDC) [19], Gear-based chunking [20], RAM [21], Rabin's Chunking (RC) [15] and TTTD [22] when chunking a 1GB file containing random data.We compare the chunking throughput of these algorithms across three average chunk sizes: 4 KB, 8 KB, and 16 KB. This experiment was run on a machine with an Intel Icelake CPU, the details of which are in §VI. We use minimum and maximum chunk sizes of 0.5 and 2× the average chunk size for applicable algorithms, similar to previous studies [17], [19], [21], [27].

We note that the throughputs of these algorithms do not scale with chunk size, as they always process the entire data stream. One of our motivating factors was to achieve higher throughput for large chunk sizes used in production. When targeting larger chunk sizes, most of the scanned data does not qualify as chunk boundaries. To minimize wasteful computation at larger chunk sizes, we propose skipping data regions during scanning. As random skipping can severely degrade the space savings achieved by deduplication, we use *content-defined data skipping* to skip scanning only certain regions (§III-C) using data-based heuristics. At larger chunk sizes, SeqCDC can afford to skip larger amounts of data without impacting space savings, leading to higher throughput.

C. Accelerating CDC algorithms with vector instructions

Vector instruction sets [26] are supported by most modern Intel and AMD CPUs. These instructions allow for the execution of arithmetic/logical operations simultaneously on multiple pieces of data, i.e., the *Single-Instruction Multiple-Data (SIMD)* paradigm. To do this, they rely on special vector registers provided within the CPU, packing multiple values into them and operating on all the values with a single operation such as an addition or subtraction. Depending on the amount of data they process at once, these instructions can be classified as SSE-128, AVX-256, and AVX-512, i.e., 128-bit, 256-bit, or 512-bit. While AVX-512 instructions are only supported by the newest Intel and AMD CPUs, SSE-128 and AVX-256 support has been available since 2003 and 2011.

Vector instructions have previously been used to accelerate mathematical operations [31], [32] and multimedia applications [33]. SS-CDC [18] previously attempted to accelerate hash-based CDC algorithms, such as CRC-32 and Gear [20], using AVX-512 instructions. They decouple the rolling hash and boundary detection phases to accelerate them separately. However, many hash-based algorithms such as FastCDC [19] and TTTD [22] use minimum chunk size skipping to improve throughput. Running the rolling hash phase on the entire source data and identifying boundaries later in a separate phase eliminates this throughput benefit. Additionally, due to the dependency between adjacent bytes when calculating hash values, accelerating rolling hash algorithms is complicated. To solve this, SS-CDC resorted to processing different regions of the data simultaneously with AVX-512 instructions, i.e, rolling with multiple heads. As this requires expensive vector `scatter/gather` instructions, the speedups achieved are limited, as shown by our previous work VectorCDC [25].

VectorCDC [25] instead accelerates hashless algorithms such as AE [17] and RAM [21] with vector instructions. It identifies two phases common to these algorithms, the *Extreme Byte Search* and *Range Scan* phases, accelerating them using different vector-based techniques. Using these techniques, VectorCDC [25] achieved orders of magnitude higher speedup for hashless algorithms than what SS-CDC [18] achieved for hash-based ones. While beneficial, vector-acceleration alone cannot improve chunking throughput at larger sizes (§VI-B).

Despite being hashless, SeqCDC does not follow the same paradigm as AE and RAM. SeqCDC uses content-defined skipping (§III-C). Additionally, instead of relying on minimum/maximum byte values to detect chunk boundaries, SeqCDC uses monotonically increasing/decreasing sequences (§III). Thus, VectorCDC's approach is incompatible with SeqCDC, motivating a new vector-compatible design.

III. SeqCDC's Design

SeqCDC is designed to insert chunk boundaries when fixed-length sequences of monotonically increasing/decreas-

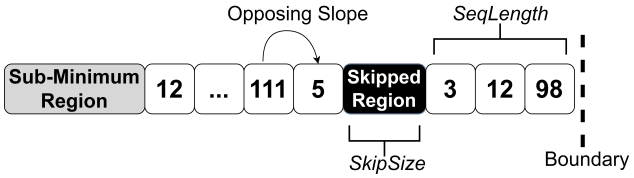


Fig. 3: An example of a chunk generated by SeqCDC

ing bytes are detected. SeqCDC can operate in either `Increasing` mode, i.e., targeting increasing order sequences or `Decreasing` mode targeting decreasing-order sequences. Note that these two modes are exclusive of each other. Figure 3 shows an example of SeqCDC's operation. §III-E discusses how SeqCDC is resistant to byte shifting.

SeqCDC utilizes three parameters: *SeqLength*, *SkipTrigger* and *SkipSize*, each described in detail in the following subsections. SeqCDC includes four optimizations we discuss in detail: lightweight boundary detection, ignoring data at the beginning of a chunk, content-based data skipping, and acceleration with vector instructions. Finally, to combat pathological data patterns, SeqCDC uses minimum and maximum chunk sizes, similar to existing CDC algorithms [19], [22].

A. Lightweight Boundary Detection

To avoid complex hashing operations, SeqCDC treats each byte within the data stream as an independent value similar to existing hashless CDC algorithms [17], [21]. However, to improve chunking throughput even further, SeqCDC reduces the overheads associated with boundary detection by avoiding minima/maxima searches.

Instead, SeqCDC looks for *fixed-length monotonically increasing/decreasing* sequences of bytes and inserts chunk boundaries at their end, whenever such sequences are detected. The sequence must have a length of *SeqLength* to be considered a boundary candidate.

Modes of operation. SeqCDC can be used in `Increasing` or `Decreasing` mode. Both these modes are exclusive of each other. While in `Increasing` mode, SeqCDC targets monotonically increasing sequences. On the other hand, it targets monotonically decreasing sequences in `Decreasing` mode. Depending upon dataset characteristics, one mode may be more effective than the other.

Figure 3 shows an example of SeqCDC operating in `Increasing` mode with a *SeqLength* of 3. A chunk boundary is inserted after the byte with value `98`, as it forms an increasing sequence with the bytes preceding it.

B. Ignoring Sub-minimum Regions

SeqCDC utilizes the concept of ignoring data at the beginning of each chunk introduced within previous literature [19], [22] to increase chunking throughput. SeqCDC skips scanning data of size (*minimum_chunk_size* - *SeqLength*) at the beginning of chunks (`"Sub Minimum Region"` in Figure 3).

Increasing the minimum chunk size allows SeqCDC to skip over larger amounts of data at the beginning of each chunk, increasing chunking throughput. However, when performed excessively, this may negatively impact space savings on some datasets. The minimum chunk size for SeqCDC is 25-50% the average chunk size, similar to existing algorithms [19], [22].

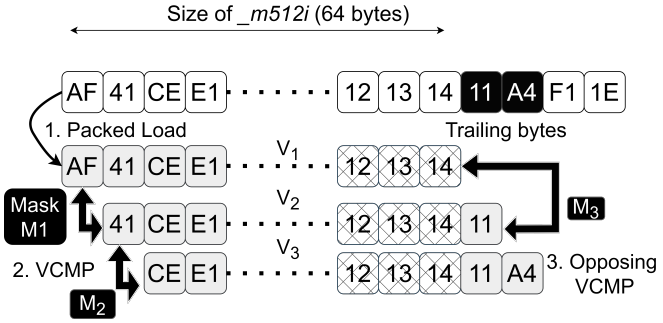


Fig. 4: Accelerating SeqCDC with AVX-512 instructions. Note that all byte values shown are in hexadecimal format.

C. Content-based Data Skipping

SeqCDC additionally improves chunking throughput by skipping scanning certain data regions when looking for chunk boundaries (`"Skipped Region"` in Figure 3). Randomly skipping data regions can lead to missed boundaries, lowering byte-shifting resistance and negatively affecting space savings. To avoid this, SeqCDC adopts a novel content-based data skipping mechanism, i.e., data regions are skipped over only when skip conditions are met.

SeqCDC skips scanning data within *unfavorable regions*, i.e., data regions with byte sequences in an order opposing the target sequence. For instance, regions with decreasing order sequences are considered unfavorable in `Increasing` mode. When *SkipTrigger* pairs of bytes in opposing order are detected, SeqCDC decides that the current region is unfavorable and skips scanning the next few bytes, hoping to land in a favorable region with bytes in the chosen order. For instance, in Figure 3, the skip condition is triggered after the byte with a value of 5, causing the next *SkipSize* bytes to be ignored. The *SkipSize* is kept small at 100-700 bytes, to avoid skipping over large sections of data. After a skip is triggered, SeqCDC resets its counters and resumes scanning for boundaries.

Larger *SkipSizes* improve chunking throughput. While larger *SkipSizes* are feasible for larger chunks, they may result in a disproportionately high amount of data skipped within smaller chunks, negatively affecting space savings. SeqCDC overcomes this by adjusting the *SkipSize*, depending on the expected average chunk size.

Data skipping can potentially impact byte-shifting resistance. SeqCDC therefore trades off a small reduction in space savings for higher chunking throughput. §III-E discusses this trade-off in greater detail. Additionally, in our evaluation (§VI-A), we show that this design decision minimally impacts space savings in real datasets.

D. Accelerating SeqCDC with vector instructions

VectorCDC [25] demonstrated the throughput benefits that can be obtained by using vector instructions to accelerate

data chunking algorithms. However, VectorCDC's approach cannot be directly applied to accelerate SeqCDC (§II-C), as SeqCDC relies on detecting monotonically increasing/decreasing sequences for chunk boundaries. We propose an alternate vector-based method to accelerate SeqCDC in this section.

Figure 4 shows an example of accelerating SeqCDC operating with a *SeqLength* of 3 and `Increasing` mode using AVX-512 instructions [26]. The figure shows a byte range with byte values `0xAF`–`0x1E` that need to be scanned for boundaries. A chunk boundary sequence starting at byte `12` exists in this region and is shown with a cross-stitched pattern. Let us assume scanning begins at byte `0xAF`.

Scan Procedure. We start by loading the 64 bytes `0xAF`–`0x14` into a vector register V_1 in packed fashion (Step 1). We then load bytes at an offset of 1 from this position, i.e., `0x41`–`0x11` into another vector register V_2. We repeat this process for a total of *SeqLength* vector registers, i.e., until V_3 in this case.

In Step 2, we run a vector comparison operation `mm512_cmpgt` between V_2 and V_1. This operation compares pairwise bytes in both registers to see if the byte from V_2 is greater than its counterpart from V_1. For instance, byte `0x41` from V_2 is compared against byte `0xAF` from V_1 as they are both the first bytes in their respective registers. The operation generates a 64-bit mask M_1, containing set bits for all positions where the byte from V_2 is greater than that of V_1. We repeat this operation between registers V_3 and V_2 as well to generate mask M_2.

In Step 3, we run a single vector comparison operation between registers V_2 and V_1. This is a `mm512_cmplt`, which compares pairwise bytes, checking for bytes in V_2 *lesser* than those in V_1. The operation generates a bit mask M_3. Note that this is the opposite comparison operation to the one from Step 2.

Following this, we check for boundaries and opposing byte pairs (detailed below) before moving the scan position by 64 bytes, i.e., to byte `0x11`. Effectively, we are scanning for chunk boundaries and content-defined skips 64 bytes at a time. Finally, note that while trailing bytes `0x11` and `0xA4` are used in Steps 1 and 2, we have not scanned for boundary sequences beginning at these bytes yet. Thus, they are used again when scanning moves ahead.

Boundary Detection. To detect boundaries, we use the masks obtained in Step 2. Each of these masks contains set bits corresponding to increasing byte pairs. If we combine all the masks using a bitwise `AND` operation, the resulting mask only contains set bits in positions with increasing bytes from all pairwise vector comparisons. If a bit at index *k* is set within M_1, it indicates that the byte at index *k* in V_2 is greater than the one at index *k* from V_1. Similarly, a set bit index *k* in M_2 indicates that the byte at index *k* in V_3 is greater than the byte at index *k* from V_2.

For instance, in Figure 4, the bit at index *61* in mask M_1 represents a comparison between byte `0x13` from V_2 and byte `0x12` from V_1 and will have a set bit. Similarly, the bit with index *61* in M_2 will be set as it compares byte `0x14` from V_3 with byte `0x13` from V_2. Thus, the resulting mask obtained using $M_1 \& M_2$ will have a set bit at index *61*.

Boundaries can be detected by examining the resulting combined mask $M_1 \& M_2$. If the mask contains any set bits (has a non-zero value), a chunk boundary is declared at *SeqLength* bytes ahead of the first set bit's position.

Content-defined skipping. To detect opposing pairs of bytes, we use mask M_3 obtained in Step 3. This mask contains set bits at all positions where a byte from V_2 is lesser than its counterpart from V_1. Thus, the total opposing byte pairs observed in the current scanned region equals the number of set bits in M_3. We keep a running total of the number of these opposing byte pairs. When the running total exceeds *SkipTrigger*, a content-defined skip of *SkipSize* bytes is initiated as described in §III-C. The exact position to jump from is determined using the first set bit that causes the total to exceed *SkipTrigger*.

Accelerated x86 Intrinsics. Intel and AMD CPUs support many other hardware-accelerated intrinsics. Forward Scan (`builtin_ffs`) is used to find the first set bit in 32/64-bit integers. Parallel Bit Deposit (`pdep`) is used to deposit contiguous low bits into a destination integer. Trailing Zero Count (`tzcnt`) is used to count the number of trailing zeros in an integer. Population Count (`popcnt`) is used to count the number of set bits in an integer. While these are hardware-accelerated, they fall under other CPU instruction sets, i.e., they are not vector instructions.

Boundary detection needs to identify the first set bit in a mask, while content-defined skips need to identify the first set bit that causes the running total to exceed *SkipTrigger*, i.e., the n^{th} set bit. We use hardware-accelerated x86 intrinsics [34] for both of these; `builtin_ffs` for boundary detection, and a combination of `pdep` and `tzcnt` for content-defined skipping. In addition, we use `popcnt` within content-defined skipping to count the total number of set bits in the opposing slope mask. Note that the performance of these intrinsics varies across CPU architectures.

E. Discussion - Impact of insertions and deletions

Inserting or deleting bytes from the middle of a file causes the data bytes to shift, resulting in changes to certain chunk boundaries. Byte shifting can span one or more bytes and take the form of insertions or deletions. In general, sub-minimum and content-defined skipped regions affect SeqCDC's byte-shifting resistance. Figure 5 shows three chunks with four boundary sequences B_1 - B_4. Each chunk has a corresponding sub-minimum region (M_1 - M_3) at the beginning of the chunk. The figure also shows two regions V_1 and V_2 skipped via SeqCDC's content-defined skipping i.e., using *SkipTrigger*, and five byte shifts $S_1 - S_5$.

Skipped region impact: When byte-shifts occur, boundaries may be moved in and out of skipped regions (both sub-minimum and content-defined). This occurs with large byte-shifts when boundaries are close to these regions. For instance, consider S_1 in Figure 5 which occurs in the sub-minimum region M_1. If byte-shift S_1 causes a boundary previously hidden within M_1 to be pushed outside, a new chunk may be created, thus splitting Chunk 1. This may in turn lead B_2 to be pushed into M_2, affecting a few subsequent chunks

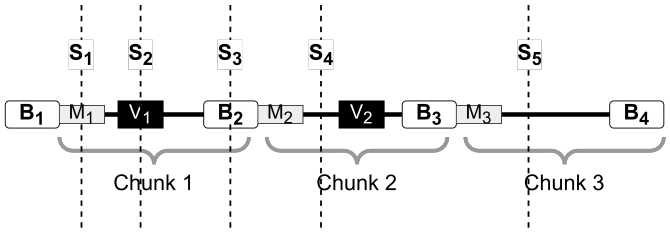


Fig. 5: Handling byte-shifts with SeqCDC

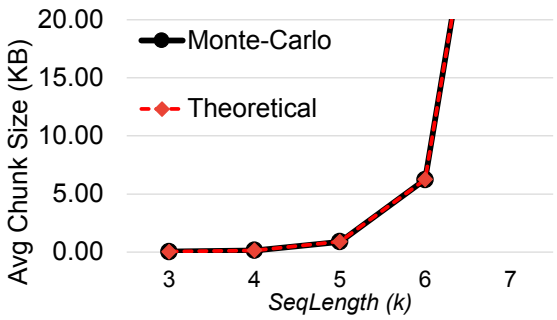


Fig. 6: Average chunk sizes for various *SeqLength*s on randomized data

as well. Similarly, a deletion may cause B_2 to be hidden within M_1, leading to chunk mergers. Additionally, boundaries may be moved in and out of regions previously skipped with *SkipTrigger* and *SkipSize* by shifts such as S_2, causing chunk splits and mergers.

While this behavior can theoretically impact a large number of chunks, it only impacts a limited number of chunks within real datasets. This is why many CDC algorithms, such as FastCDC [19] and TTTD [22] use sub-minimum skips in production. SeqCDC also minimizes the impact caused by content-defined skipping by keeping the *SkipSize* under 705 bytes. Thus, despite data skipping, SeqCDC achieves competitive space savings with other CDC algorithms, as shown in §VI-A. Finally, we note that all CDC algorithms have pathological data patterns i.e. data engineered to ensure that they are ineffective.

In the rest of this section, we focus on the more common kind of byte-shifting i.e. those that do not result in new boundaries being uncovered or hidden, simply being shifted instead.

Within the boundary sequence. The probability of byte shifting occurring within a boundary sequence is rare in real datasets, as *SeqLength* typically ranges from 3 to 7 bytes (§IV-B). If byte shifting occurs within a sequence, the boundary may no longer exist. Thus, scanning will continue until the next sequence is detected. In Figure 5, S_3 may cause B_2 to exist no longer. Thus, the next boundary will be after B_3, causing Chunks 1 and 2 to be merged while subsequent chunks are unaffected.

Between sequences and outside skipped regions. Byte-shifts such as S_4 are the ones most commonly seen in real datasets. They occur outside of skipped regions and do not drastically change the chunk structure. If S_4 does not create a new boundary sequence, the existing boundary sequence B_3 shifts. Thus, Chunk 2 changes while subsequent chunks are unaffected. On the other hand, if S_4 does create a new boundary sequence, a boundary is inserted after it, changing Chunk 2. Depending on the shift position, the next boundary detected may be B_3 or B_4, resulting in a changed Chunk 3 as well. Thus, Chunks 2 and 3 are affected while others are unaffected.

Maximum chunk size. If a shift S_5 causes the maximum chunk size to be reached, a boundary is inserted at the maximum chunk size, similar to existing algorithms [17], [19], [21]. This will cause the chunk to split into multiple chunks. For instance, if a boundary is inserted after S_5 in Figure 5, Chunk 3 will be split into two if no subsequent boundary sequences are hidden.

IV. Deriving Parameter Values

A. Mathematical analysis without data skipping

Let us consider the base SeqCDC algorithm without minimum chunk size or content-defined data skipping. SeqCDC uses k-length subsequences to identify chunk boundaries; thus, the average chunk size depends on the value of k. Note that we frame our discussion around strictly increasing k-length subsequences, but the calculations remain the same for strictly decreasing ones as well.

In mathematical terms, we have a sequence $S = (x_i)_{i=1}^{n}$ of integer terms from $\{0, 1, \ldots, 255\}$. S simulates a stream of random bytes and hence is an independent identically distributed sequence [35]. The problem is to estimate, on scanning S, how often we encounter a k-length strictly increasing contiguous subsequence. There is a constraint; we only count non-overlapping sequences.

In this setting, S can be viewed as a concatenation of *components*, each of which is a maximally strictly increasing subsequence. Note that a component can still be a singleton.

Let λ_k be the rate of k-length strictly increasing subsequences occurring in stream S. λ_k can be calculated as:

$$\lambda_k = \frac{\text{Expected } k\text{-length subsequences per component}}{\text{Expected length of component}}.$$

In Appendix E, we calculate that

$$\lambda_k = \frac{m+1}{m} \sum_{j=1}^{\lfloor m/k \rfloor} \frac{1}{m^{kj}} \binom{m}{kj} \left(1 + \frac{1}{kj}\right)^{-1},$$

where m is the total number of possible values for a single byte i.e., $m = 256$.

Using λ_k, the total number of k-length subsequences in S can be calculated as $\lambda_k \times length(S)$. Assuming each subsequence is a chunk boundary i.e., no minimum chunk size skipping, the average chunk size for each k can be denoted as:

$$Avg\ Chunk\ Size\ (k) = \frac{length(S)}{\lambda_k \times length(S)} = \frac{1}{\lambda_k} \quad (2)$$

Figure 6 shows the average chunk sizes obtained using Monte-Carlo simulations on random data, compared against those obtained theoretically using λ_k, for different values of

Dataset	Size	Information	XC
DEB	40 GB	65 Debian VM Images from VMware Marketplace [36]	9.59%
DEV	230 GB	100 backups of a Rust [37] nightly build server	69.37%
DKR	110 GB	200 Docker Images [38] for storage and database solutions	10.21%
LNX	65 GB	160 Linux kernel distributions in TAR format [39]	9.65%
RDS	122 GB	100 Redis [40] snapshots with `redis-benchmark` runs	33.45%
TPCC	106 GB	25 snapshots of a MySQL [41] VM running TPC-C [42].	22.25%

TABLE I: Dataset information. Note that *XC* is the space savings achieved using 16 KB fixed-size chunks.

k. Note that k is the same as *SeqLength* and that our Monte-Carlo results are the averages of 50 runs. We observe that the actual and expected average chunk sizes only differ by a maximum of 3% across all k values.

B. Deriving parameter values for real datasets

While the theoretical analysis above provides some guidelines on how to choose *SeqLength*, it is not always accurate due to the following reasons:

- Real datasets have patterns which are vastly different from randomized data. As a result, the parameter values derived by many previous CDC algorithms [15], [17], [21] using random data do not match those used in production.
- Content-defined skipping influences the average chunk size and is mathematically complex to model. (§III-E)
- SeqCDC uses minimum and maximum chunk sizes, which influence the average chunk size.

To address these issues, we propose a dataset-specific process to obtain parameter values. SeqCDC has three configurable parameters apart from its mode: *SeqLength*, *SkipTrigger* and *SkipSize*. To obtain the parameter value combination needed to generate a given average chunk size, we follow the steps below. We have automated these steps and released the associated code with our implementation (§V)

1) We first perform Monte-Carlo simulations [43] on randomized data streams with data skipping enabled, to identify the maximum and minimum values for each parameter that result in a chunk size close to the target. For example, to identify the parameter values for an average size of 16 KB, we identify candidate combinations resulting in average chunk sizes of 14 – 18 KB. For instance, the *SkipSize* for a 16 KB target size varied between 320 bytes – 704 bytes.
2) We use the parameter values obtained here as the bounds for a parameter grid-search on each dataset. The increments chosen for each parameter dictate the computational cost of the parameter search. For instance, we varied *SkipSize* from 320 – 704 bytes in increments of 32 bytes. The limits and increments in parameter values that we chose resulted in a total of 480 combinations for each chunk size. Note that these can be increased or reduced, depending on available computational resources.

Dataset	*SeqLength*	*SkipTrigger*	*SkipSize*
DEB	5	40	640
DEV	5	35	576
DKR	5	35	384
LNX	6	55	320
RDS	6	60	384
TPCC	5	45	704

TABLE II: SeqCDC parameter values for 16 KB chunks

3) Following this, for each dataset, we perform a parameter grid-search with all combinations between the chosen limits and increments to determine the best configuration. As parameter grid searches are computationally expensive, we uniformly sample 10% of the dataset and use this sample for the grid search. Our underlying assumption is that duplication patterns do not change significantly in different dataset regions, which holds true for all real-world datasets we analyzed.
4) We choose the top five parameter combinations from the grid search that result in the closest average chunk size to the target size, while achieving the best combination of space savings and throughput. We run each of these on the entire dataset, and report the results from the best combination in §VI.

Table II shows our final chosen combination for each dataset to generate chunks with 16 KB average size. Note that the entire parameter search is a one-time procedure in production for any given dataset.

V. Implementation

We implemented a native (unaccelerated) version of SeqCDC using approximately 250 lines of C++ code. We optimized the computation of *SeqLength* and *SkipTrigger* using the `std::signbit` function to reduce the number of branch conditions. We have automated the parameter search process described in §IV-B to run with any given dataset, using 150 lines of Python and Shell code. We have released the code for these publicly with DedupBench [27].

We implemented the vector acceleration described in §III-D using an additional 350 lines of C++ code. We have implemented SSE-128, AVX-256 and AVX-512 compatible accelerations for SeqCDC. SeqCDC is compatible with all file and data formats.

x86 intrinsics performance. As noted in §III-D, the performance of `popcnt`, `pdep` and `tzcnt` may vary across CPU architectures. In order to estimate their performance on a given CPU, we implemented an x86 intrinsics micro-benchmark using approximately 250 lines of C++ code.

VI. Evaluation

In this section, we evaluate SeqCDC's space savings, chunk size distribution, and chunking throughput and compare it to the state-of-the-art CDC algorithms.

Testbed. We use machines with Intel Skylake and Icelake CPUs from CloudLab Wisconsin [44] for our evaluation. The

Skylake machine (*c220g5*) consists of two 10-core Intel Xeon Silver 4114 CPUs with hyperthreading, 192 GB of RAM, and a 10 GBps Intel NIC, The Icelake machine (*sm220u*) consists of two 16-core Intel Xeon Silver 4314 CPUs with hyperthreading, 256 GB of RAM and a 100 GBps Mellanox NIC. Unless otherwise mentioned, we show results from the Icelake machine. All our results are the average of 5 runs with a standard deviation of less than 5%.

Alternatives. We evaluate the following hash-based CDC algorithms:

- *CRC*: A chunking algorithm using CRC-32 [18].
- *FCDC*: FastCDC [19] with a normalization level of 2.
- *GEAR*: Gear-based chunking [20].
- *RC*: A hash-based CDC using Rabin's fingerprinting algorithm [15].
- *SS-CRC*: AVX-512 version of CRC, accelerated with SS-CDC [18].
- *SS-Gear*: AVX-512 version of Gear, accelerated with SS-CDC [18].
- *TTTD*: Two-Threshold Two-Divisor Algorithm, based on Rabin's fingerprinting with a backup divisor [22].

We also evaluate the following hashless CDC algorithms:

- *AE*: The Asymmetric Extremum [17] algorithm.
- *RAM*: Rapid Asymmetric Maximum [21].
- *VRAM*: SSE-128, AVX-256, and AVX-512 versions of RAM accelerated with VectorCDC [25].
- *SEQ*: Native version of SeqCDC. We only report the results for SeqCDC in `Increasing` mode. The results for `Decreasing` mode are similar.
- *VSEQ*: SSE-128, AVX-256, and AVX-512 versions of SeqCDC in `Increasing` mode.

We use minimum and maximum chunk sizes of $\frac{1}{2}\times$ and $2\times$ the expected average chunk size, in line with previous studies [19], [22]. The only exception is that for a small average chunk size of 4 KB, we use a minimum size of 1 KB.

SeqCDC uses a smaller context window of size *SeqLength* to determine chunk boundaries while hash-based algorithms use larger windows, typically between $32-64$ bytes. A common concern is whether hash-based algorithms can achieve throughput similar to SeqCDC with smaller context windows. We have experimented with reduced context windows for hash-based algorithms in Appendix B. The results show that hash-based algorithms' throughputs are unaffected by window size, i.e., reducing the window sizes of hash-based algorithms with smaller windows does not improve their throughput, as the bottleneck is in the rolling-hash process.

Datasets. Table I shows the datasets used in our evaluation as well as the space savings achieved by using fixed-size chunking (XC) with an average size of 16 KB. The datasets represent diverse workloads such as database backups, docker images, VMs, and Linux kernel code. We have made the `DEB` dataset publicly available[2] [45].

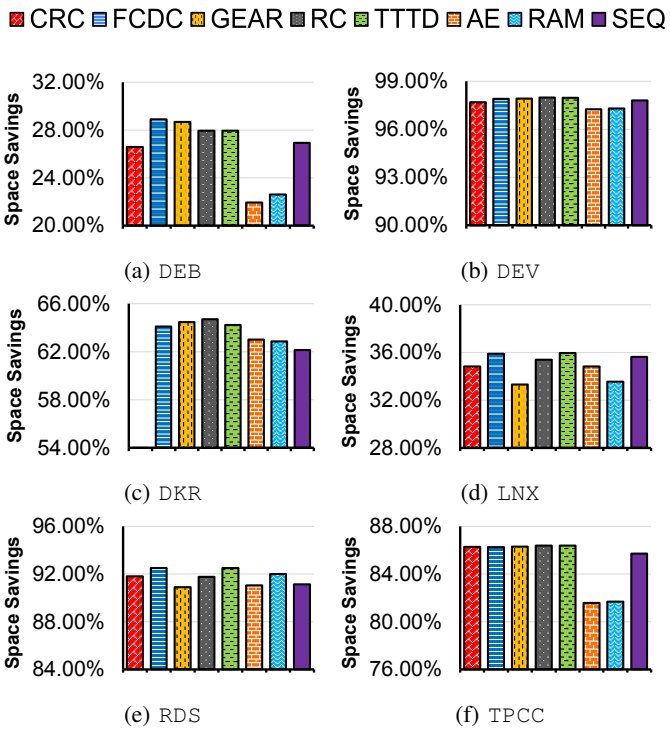


Fig. 7: Space Savings with 16 KB chunks. Note that *SEQ* is unaccelerated SeqCDC.

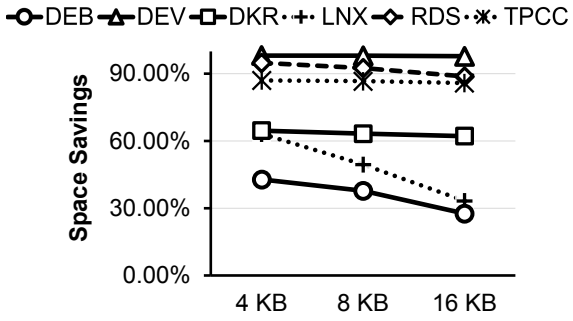


Fig. 8: SeqCDC's Space Savings vs Chunk Size across datasets

A. Space Savings

Figure 7 shows the space savings achieved by all the alternatives across datasets with 16 KB chunks. Appendix C shows the detailed average chunk sizes exhibited by all algorithms. By comparing the space savings achieved by fixed-size chunking (*XC*) on all datasets (Table I) to those achieved by the CDC algorithms (Figure 7), we note that CDC algorithms achieve superior space savings to *XC*. For instance, *XC* achieves a space savings of only 22.25% on `TPCC`, while CDC algorithms achieve 81-87%. The detailed results for all chunk sizes, algorithms, and dataset combinations are in the Appendix in Table III.

CDC comparison. From Figure 7, we see that SeqCDC (*SEQ*) achieves similar space savings to all the other CDC

[2] https://www.kaggle.com/datasets/sreeharshau/vm-deb-fast25

algorithms on these datasets with 16 KB chunks. The best algorithm for space savings varies depending on the dataset. For instance, *RC*, *FCDC*, and *TTTD* achieve the best space savings on DEV, RDS, and TPCC respectively. The detailed results for all chunk sizes are in the Appendix A. *On all datasets and chunk sizes, SeqCDC either is the best or achieves space savings within* 6% *of the best performer.*

We note that *CRC* suffers from poor space savings of 20.88% on DKR, while other CDC algorithms achieve 62–64%. This is due to the dataset characteristics, resulting in *CRC* being unable to find boundaries. As shown in Appendix C, this also causes *CRC* to exhibit a much higher average chunk size on DKR compared to other CDC algorithms.

Finally, *Deduplication Elimination Ratio (DER)*, a metric alternative to space savings, has been sometimes used in previous literature to compare CDC algorithms [17], [23]. While they are different forms of the same metric, space savings is easier to interpret [46] and is more commonly used [19], [20], [25], [27]. We analyzed the *DER* exhibited by all CDC algorithms across datasets and found that the trends were similar to those shown in Figure 7. We omit those results from this section, but provide them in Appendix D for completeness.

Dataset characteristics. Figure 8 shows the space savings variation across chunk sizes for all our datasets using SeqCDC; the results are similar for other CDC algorithms. The space savings achieved decrease with increasing chunk size across datasets.

The space savings degradation between 4 KB and 16 KB average chunk sizes on the DEV, DKR, RDS, and TPCC datasets is 0.5–6%. However, as the total number of chunks at 16 KB is far lower than that at 4 KB, the size of the fingerprint database and fingerprinting overheads are significantly lower. Similarly, the best chunk size configuration for the DEB dataset is 8 KB. *This demonstrates why deduplication systems favor larger chunk sizes on some datasets.*

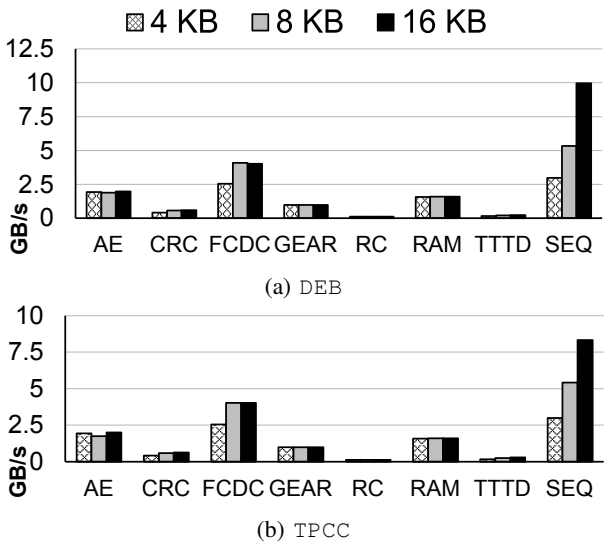
On the other hand, the LNX dataset presents a case favoring smaller chunk sizes. The space savings degradation for LNX in Figure 8 moving from average chunk sizes of 4 KB to 16 KB is 29.87%. This far outweighs any gains within fingerprint indexing.

Vector acceleration. We observed that vector acceleration minimally impacts the space savings achieved by CDC algorithms, in line with previously observed results [18], [25]. For instance, *SS-CRC* and *CRC* achieve the same space savings values. We have excluded these results from Figure 7 for clarity. Similarly, vector acceleration does not impact SeqCDC's space savings.

B. Chunking Throughput: Unaccelerated CDC

Figure 9 shows the chunking throughput for all unaccelerated CDC algorithms on DEB and TPCC. The results for other datasets are similar. When examining chunking throughputs across average chunk sizes, we note that the throughput minimally scales with size for all algorithms other than *SEQ*, similar to our analysis in §II-B.

Among the hash-based algorithms, *RC* [15] and *TTTD* [22] are the slowest, only achieving 0.12 − 0.22 GB/s. *TTTD* is slightly faster than Rabin's chunking due to sub-minimum skipping (§II-A). The poor chunking throughput of both algorithms is due to the high computational cost of Rabin's hashing, as pointed out in previous literature [19]. *GEAR* chunking [20] uses the faster gear hash algorithm allowing it to reach 0.95 − 0.98 GB/s. FastCDC (*FCDC*) fares significantly better and achieves 2.5 − 3.5 GB/s, largely due to sub-minimum skipping and using the Gear hashing algorithm. *FCDC*'s throughput increases between average chunk sizes of 4 KB to 8 KB due to the increased ratio of the sub-minimum region size to chunk size (25% to 50%).



(a) DEB

(b) TPCC

Fig. 9: Chunking throughput of native CDC algorithms. Note that *SEQ* is unaccelerated SeqCDC.

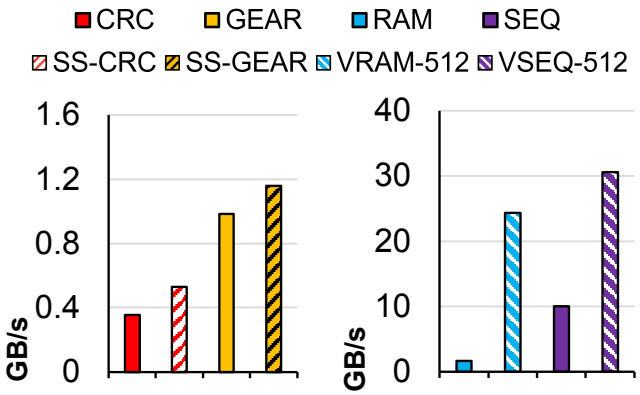
Hashless algorithms such as *AE* [17] and *RAM* [21] achieve higher chunking throughput than all hash-based algorithms except *FCDC*. They both achieve throughputs of 1.5−1.9 GB/s across datasets. FCDC is only faster than *AE* and *RAM* because it uses sub-minimum skipping; they do not.

Unaccelerated SeqCDC (*SEQ*) consistently achieves higher chunking throughput than all other CDC algorithms at average chunk sizes of 8 KB and 16 KB. At a chunk size of 8 KB, it achieves 5.3 − 5.4 GB/s, 1.3× and 2.8× better than *FCDC* and *AE* respectively. At a chunk size of 16 KB, it achieves a chunking throughput of ˜8.8−10 GB/s, 2.15× and 4.6× better than *FCDC* and *AE* respectively. *SEQ*'s increase in throughput from 8 KB to 16 KB is primarily due to a larger amount of skipped data, i.e., lower *SkipTrigger*s and higher *SkipSize*s.

At a chunk size of 4 KB, SeqCDC constrains the amount of data skipped by using higher *SkipTrigger*s and lower *SkipSize*s. As expected, this results in lower throughputs. However, SeqCDC still achieves 2.9 GB/s, 1.16× and 1.5× faster than *FCDC* and *AE* respectively. Thus, *native SeqCDC is faster than other native CDC algorithms by* 1.5×–2.8× *at larger chunk sizes and* 16% *at smaller chunk sizes.*

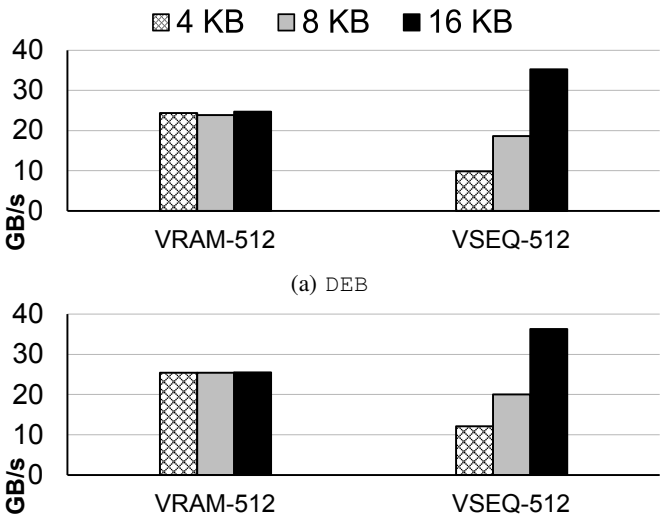
C. Chunking Throughput: Vector-accelerated CDC

Figure 10 shows the throughput of native algorithms with their respective AVX-512 accelerated versions; *CRC* and *Gear*



(a) CRC and Gear with SS-CDC (b) SeqCDC and VRAM

Fig. 10: Chunking speedups on DEB with 16 KB chunks



(a) DEB

(b) TPCC

Fig. 11: Chunking throughput of vector-accelerated CDC algorithms. Note that *VSEQ* is vector accelerated SeqCDC.

accelerated with SS-CDC [18], *RAM* accelerated with VectorCDC [25] and SeqCDC accelerated with the method described in §III-D. Figure 10b shows that *VSEQ-512 achieves the highest throughput among all vector-accelerated algorithms*, at 30.5 GB/s. It is also 10× faster than *FCDC*, the fastest non-vector alternative.

Vector speedups. *SS-CRC* and *SS-Gear* (Figure 10a) achieve low speedups over their native counterparts due to the use of scatter / gather instructions to overcome dependencies between adjacent bytes (§II-C). Hashless vector-based algorithms such as *VRAM* and *VSEQ* do not suffer from these limitations. It is important to note that while *VSEQ* achieves the highest throughput overall, its speedup over its native counterpart is lower than that of *VRAM* (Figure 10b). *VSEQ-512* achieves a 3.05× speedup over *SEQ* while *VRAM-512* achieves a 16× speedup over *RAM*. Thus, *SeqCDC is not as vector-friendly as RAM.*

Accelerating FastCDC. SS-CDC's [18] proposed method can accelerate any hash-based algorithm using AVX-512 instructions (§II-C). However, they decouple the rolling hash and boundary detection phases to accomplish this. FastCDC uses the Gear rolling hash algorithm [19] to detect boundaries, and optimizes its throughput via sub-minimum skipping, i.e., skipping data regions. SS-CDC's decoupling causes the rolling hash to be run on the entire incoming data stream, nullifying the throughput benefit of sub-minimum skipping. We observed no throughput benefits when accelerating FastCDC with vector instructions, similar to the results in earlier studies [25].

For the rest of our evaluation, we use only *VSEQ* and *VRAM*.

VSEQ vs VRAM. Figure 11 compares the two best performing alternatives *VSEQ-512* and *VRAM-512* across different chunk sizes on DEB and TPCC. The figure shows that *VRAM* cannot scale its throughput with increasing chunk sizes, similar to its native counterpart *RAM*. Thus, *vector-acceleration alone cannot fix the inability of CDC algorithms to scale their throughput with chunk size*. *VSEQ* can accomplish this due to its changing *SkipSize*, similar to *SEQ*.

At lower chunk sizes such as 4 KB, we note that *VRAM* outperforms *VSEQ*. However, at 8 KB this performance gap quickly closes. At 16 KB, *VSEQ* outperforms *VRAM* by 1.23–1.35×. This gap increases further at larger chunk sizes. Thus, while *VRAM* is faster at smaller chunk sizes, *VSEQ is* 1.23 – 1.35× *faster than VRAM at the larger chunk sizes favored by deduplication systems.*

Throughput breakdown. Figure 12 shows the impact of each optimization (§III) on SeqCDC's throughput. We use 16 KB chunks for this experiment.

BASE represents native SeqCDC running with only lightweight boundary judgement and sub-minimum skipping i.e. without content-defined skipping. *SEQ* enables content-defined skipping. *VBASE-512* represents an AVX-512 accelerated SeqCDC without content-defined skipping. Finally, *VSEQ-512* uses both AVX-512 acceleration and content-defined skips.

We note that content-defined skipping is beneficial on all datasets without AVX-512 acceleration, albeit to different extents. For instance, content-defined skipping allows *SEQ* to achieve 1.8× higher throughput than *BASE* on RDS. On the other hand, it only achieves 1.39× on DEB.

When pairing content-defined skips with AVX-512 acceleration, the landscape changes. We notice three kinds of behavior between *VBASE-512* and *VSEQ-512* in Figure 12. On RDS, the benefits are still significant, with *VSEQ-512* achieving a

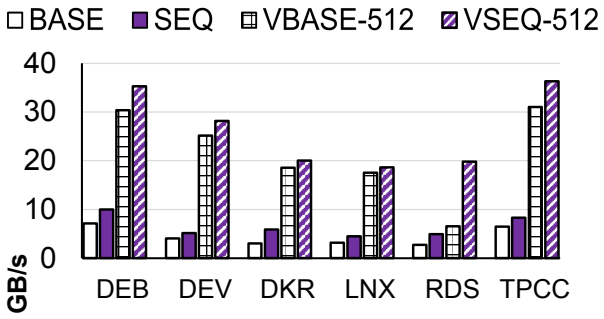
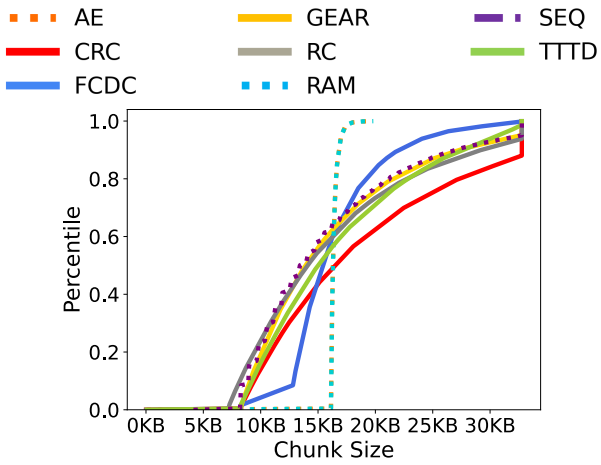


Fig. 12: SeqCDC's throughput breakdown at 16 KB with native and AVX-512 instructions



(a) DEB

(b) TPCC

Fig. 13: Chunk size distribution (CDF) at an average chunk size of 16 KB

3.03× higher throughput than *VBASE-512*. On DEV, DEB, and TPCC, the speedups are slightly lower at 1.11×–1.41×. Finally, on DKR and LNX, skipping is not that beneficial, improving throughput only by 1.05×–1.06×.

Despite being hardware-accelerated, keeping track of opposing slopes and jump positions does not require vector instructions; it uses Bit Manipulation Instructions (BMI and BMI-2) [47]. Such intermingling of vectorized and non-vectorized code regions leads to lower performance gains [48], and may outweigh the benefits of content-defined skips on certain datasets. Thus, while SeqCDC's *content-defined skipping is almost always beneficial without vector instructions, its benefits depend on dataset characteristics when paired with vector instructions.*

D. Backward compatibility with AVX-256 and SSE-128

Only a handful number of newer Intel and AMD processors support AVX-512 instructions. However, SSE-128 and AVX-256 instructions have been supported by Intel and AMD since 2003 and 2011 respectively. Thus, most CPUs available today support one of the three classes of vector instructions.

While SeqCDC's design described in §III-D uses AVX-512 instructions, SeqCDC can be accelerated with SSE-128/AVX-256 instructions as well. Figures 14a and 14c show the benefits of accelerating SeqCDC with SSE-128, AVX-256, and AVX-512 instructions on two different CPU architectures: Intel Skylake and Intel Icelake. We use the DEB dataset.

We note that *VSEQ-128* and *VSEQ-256* achieve significant speedups over *SEQ* on both architectures. For instance with 16 KB chunks, *VSEQ-128* achieves a 1.57×–2.19× speedup on both architectures. Similarly, *VSEQ-256* at 16 KB achieves a 2.52×–3.34× speedup. Thus, *SeqCDC retains most of its throughput benefits when accelerated with SSE-128 and AVX-256 instructions, and is compatible with most modern CPUs.*

Note that the speedups slightly vary across architectures. For instance, *VSEQ-512* achieves a 5.8× speedup over *SEQ* on the Skylake while it achieves 3.05× on the Icelake machine. This is largely tied to CPU microarchitecture and x86 intrinsic performance as described in §III-D.

SeqCDC vs VRAM. Figures 14b and 14d compare *VSEQ* with the best performing vector accelerated alternative, *VRAM*, using SSE-128, AVX-256 and AVX-512 instructions on Intel Skylake and Icelake architectures. We use DEB with 16 KB chunks for this experiment. We see that *VSEQ* maintains its performance advantage over *VRAM* with all vector instruction families, further cementing its effectiveness and backward compatibility.

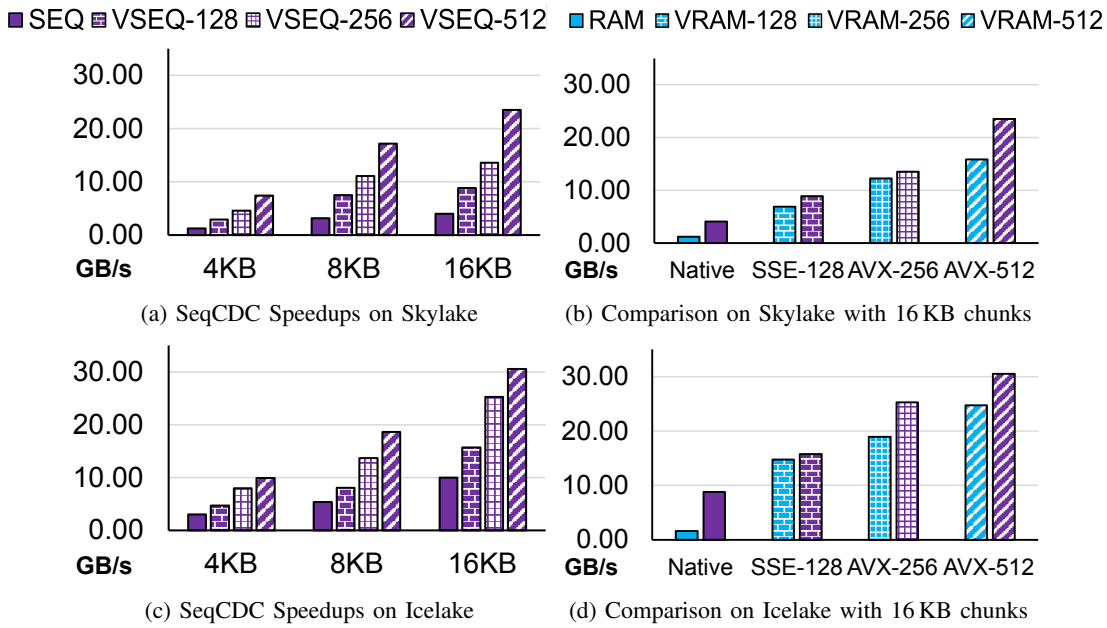
E. Chunk Size Distribution

CDC algorithms are expected to achieve a uniform chunk size distribution, loosely centered around the average chunk size. Figure 13 shows a CDF of chunk sizes from all algorithms at an average size of 16 KB on the DEB and TPCC datasets. Hash-based algorithms' distributions are shown using solid lines, while hashless algorithms use patterned lines.

The exhibited average chunk size needs to be similar for a fair comparison of deduplication efficiency. From Figure 13, we note that the average chunk sizes exhibited by all CDC algorithms are similar. The only exception is *CRC* which is very sensitive to parameter changes and exhibits a slightly larger average chunk size on all datasets. Excluding CRC, the maximum difference between the exhibited average chunk sizes of all algorithms is 5.8% and 2.6% of the target average chunk size, on DEB and TPCC, respectively. The detailed average chunk sizes for all algorithms across datasets can be found in Appendix C. Note that the details for other target sizes were similar, but have been omitted from Appendix C due to space constraints.

Hash-based algorithms exhibit uniform distributions between the minimum and maximum chunk sizes. Rabin's Chunking (*RC*) and *TTTD* exhibit similar distributions since *TTTD* only differs from *RC* by the use of a backup divisor. *CRC* and *GEAR* exhibit similar smooth patterns as well. FastCDC (*FCDC*) is an exception, exhibiting a split pattern centered around the average chunk size. This is because it switches masks and relaxes boundary conditions past the average chunk size i.e., chunk size normalization (§II-A). *AE* and *RAM* exhibit nearly identical tighter distributions when compared to hash-based algorithms.

Despite being hashless, SeqCDC exhibits a chunk size distribution similar to hash-based algorithms. We observed similar results across all our datasets and chunk sizes.



(a) SeqCDC Speedups on Skylake

(b) Comparison on Skylake with 16 KB chunks

(c) SeqCDC Speedups on Icelake

(d) Comparison on Icelake with 16 KB chunks

Fig. 14: Chunking Throughput on DEB with SSE-128, AVX-256 and AVX-512 Instructions

Vector acceleration. Chunk size distributions remain unaffected by vector acceleration for all CDC algorithms.

VII. Additional Related Work

Accelerating deduplication. Numerous efforts have been made to accelerate the other phases involved in data deduplication. StoreGPU [49] uses GPUs, Silo [50] uses locality-based optimizations and other work [51], [52] focuses on using faster hashing algorithms to improve fingerprint indexing. History-aware rewriting [53] and FG-DEFRAG [54] improve restore performance by reducing fragmentation in stored chunks. HYDRAstor [55] and Extreme Binning [56] improve scalability by building deduplication on top of distributed storage.

While SeqCDC is compatible with many of these approaches, they are orthogonal to ours as we focus on the file chunking phase within deduplication.

Other chunking optimizations. RapidCDC [57] and QuickCDC [58] use locality-based optimizations to speed up chunking for duplicate chunks. MUCH [59] and P-Dedupe [60] parallelize chunking using multiple threads. SeqCDC is compatible with any of these techniques as they all rely on implementing optimizations on top of existing CDC algorithms.

MII [61] uses a sequence-based approach to chunk data but their approach results in inflexible chunk sizes and low throughput. Our previous work [62] discussed SeqCDC briefly. However, it does not present a methodology to accelerate SeqCDC with vector instructions or compare its performance with other vector-accelerated CDC algorithms.

Decentralized deduplication. Some deduplication systems adopt a decentralized approach [63], [64], adopting a distributed index on top of distributed file systems such as HDFS [3]. These efforts focus on improving coordination between nodes, distributing load among nodes, and managing a large-scale fingerprint index. They all still use the data chunking techniques we discussed in §II. These efforts are orthogonal to SeqCDC, and SeqCDC is compatible with these systems.

Secure deduplication systems. Several efforts build end-to-end deduplication systems for encrypted data [65]. They mainly target encryption schemes [66], [67] for the underlying data or focus on reducing attacks on the system [68]. SeqCDC is compatible with all these approaches.

VIII. Conclusion

Deduplication systems in production employ larger chunk sizes due to reduced fingerprinting overheads. However, state-of-the-art CDC algorithms are designed to target smaller average chunk sizes, suffering from poor chunking throughput at larger sizes.

We present SeqCDC, a CDC algorithm that achieves higher chunking speeds than the state-of-the-art. SeqCDC leverages hashless lightweight boundary judgement, content-based data skipping, and SSE/AVX acceleration to improve chunking throughput by $10\times$ over unaccelerated and $1.2\times - 1.35\times$ over vector accelerated CDC algorithms while achieving similar deduplication space savings. We hope that our work inspires a new generation of vector-friendly data chunking algorithms to accelerate data deduplication.

IX. Future Work

SeqCDC currently uses fixed parameters to generate chunks for specific target sizes. A future direction is to explore dynamic parameters that can be relaxed or tightened. This could be tied to the amount of data scanned within the current chunk, the distance of the current position from the target chunk size, and the sizes of previously generated chunks.

SeqCDC relies on Bit Manipulation Instructions (BMI) to accelerate content-defined skipping. Other CPU architectures, such as ARM [69], [70] and IBM Power [71], support vector

instructions but not equivalents for BMI instructions. An alternative vector-compatible design for these CPUs is necessary.

Acknowledgments

We thank Abdelrahman Baba for his contributions to an earlier version of this paper. We thank Lori Paniak for helping us enable our experiments. This research was supported by grants from the National Cybersecurity Consortium (NCC), Natural Sciences and Engineering Research Council of Canada (NSERC) (ALLRP-561423-20 and RGPIN-2025-03332), and Acronis. Sreeharsha is supported by the Cheriton Graduate Scholarship and the Ontario Graduate Scholarship.

References

[1] A. Holst, "Volume of data/information created, captured, copied, and consumed worldwide from 2010 to 2025," *Statista, June*, 2021.

[2] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google File System," *SIGOPS Operating Systems Review*, vol. 37, p. 29–43, oct 2003.

[3] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop Distributed File System," in *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pp. 1–10, 2010.

[4] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson, "RAID: High-performance, reliable secondary storage," *ACM Computing Surveys (CSUR)*, vol. 26, no. 2, pp. 145–185, 1994.

[5] G. A. Gibson and R. Van Meter, "Network attached storage architecture," *Communications of the ACM*, vol. 43, no. 11, pp. 37–45, 2000.

[6] D. A. Lelewer and D. S. Hirschberg, "Data compression," *ACM Computing Surveys (CSUR)*, vol. 19, no. 3, pp. 261–296, 1987.

[7] D. Salomon, *Data compression*. Springer, 2002.

[8] W. Xia, H. Jiang, D. Feng, F. Douglis, P. Shilane, Y. Hua, M. Fu, Y. Zhang, and Y. Zhou, "A comprehensive study of the past, present, and future of data deduplication," *Proceedings of the IEEE*, vol. 104, no. 9, pp. 1681–1710, 2016.

[9] N. Mandagere, P. Zhou, M. A. Smith, and S. Uttamchandani, "Demystifying data deduplication," in *The ACM/IFIP/USENIX Middleware'08 Conference Companion*, pp. 12–17, 2008.

[10] D. T. Meyer and W. J. Bolosky, "A study of practical deduplication," *ACM Transactions on Storage (ToS)*, vol. 7, no. 4, pp. 1–20, 2012.

[11] G. Wallace, F. Douglis, H. Qian, P. Shilane, S. Smaldone, M. Chamness, and W. Hsu, "Characteristics of backup workloads in production systems," in *10th USENIX Conference on File and Storage Technologies (FAST 12)*, (San Jose, CA), USENIX Association, Feb. 2012.

[12] L. Coyne, S. Moulton, and C. Alvarez, *IBM System Storage N Series Data Compression and Deduplication: Data ONTAP 8.1 Operating in 7-mode*. IBM Redbooks, 2012.

[13] S. Quinlan and S. Dorward, "Venti: A New Approach to Archival Data Storage," in *Conference on File and Storage Technologies (FAST 02)*, (Monterey, CA), USENIX Association, Jan. 2002.

[14] Q. He, Z. Li, and X. Zhang, "Data deduplication techniques," in *2010 international conference on future information technology and management engineering*, vol. 1, pp. 430–433, IEEE, 2010.

[15] A. Muthitacharoen, B. Chen, and D. Mazieres, "A low-bandwidth network file system," in *The eighteenth ACM symposium on Operating systems principles*, pp. 174–187, 2001.

[16] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, *et al.*, "Oceanstore: An architecture for global-scale persistent storage," *ACM SIGOPS Operating Systems Review*, vol. 34, no. 5, pp. 190–201, 2000.

[17] Y. Zhang, H. Jiang, D. Feng, W. Xia, M. Fu, F. Huang, and Y. Zhou, "AE: An asymmetric extremum content defined chunking algorithm for fast and bandwidth-efficient data deduplication," in *2015 IEEE Conference on Computer Communications (INFOCOM)*, pp. 1337–1345, IEEE, 2015.

[18] F. Ni, X. Lin, and S. Jiang, "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, (New York, NY, USA), p. 86–96, Association for Computing Machinery, 2019.

[19] W. Xia, Y. Zhou, H. Jiang, D. Feng, Y. Hua, Y. Hu, Q. Liu, and Y. Zhang, "FastCDC: A fast and efficient content-defined chunking approach for data deduplication," in *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pp. 101–114, 2016.

[20] W. Xia, H. Jiang, D. Feng, L. Tian, M. Fu, and Y. Zhou, "Ddelta: A deduplication-inspired fast delta compression approach," *Performance Evaluation*, vol. 79, pp. 258–272, 2014. Special Issue: Performance 2014.

[21] R. N. Widodo, H. Lim, and M. Atiquzzaman, "A new content-defined chunking algorithm for data deduplication in cloud storage," *Future Generation Computer Systems*, vol. 71, pp. 145–156, 2017.

[22] K. Eshghi and H. K. Tang, "A framework for analyzing and improving content-based chunking algorithms," *Hewlett-Packard Labs Technical Report TR*, vol. 30, no. 2005, 2005.

[23] N. Bjørner, A. Blass, and Y. Gurevich, "Content-dependent chunking for differential compression, the local maximum approach," *Journal of Computer and System Sciences*, vol. 76, no. 3-4, pp. 154–203, 2010.

[24] A. W. Appel, "Verification of a cryptographic primitive: SHA-256," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 37, no. 2, pp. 1–31, 2015.

[25] S. Udayashankar, A. Baba, and S. Al-Kiswany, "VectorCDC: Accelerating Data Deduplication with Vector Instructions," in *USENIX Conference on File and Storage Technologies*, 2025.

[26] J. E. Smith, G. Faanes, and R. Sugumar, "Vector instruction set support for conditional operations," *ACM SIGARCH Computer Architecture News*, vol. 28, no. 2, pp. 260–269, 2000.

[27] A. Liu, A. Baba, S. Udayashankar, and S. Al-Kiswany, "Dedupbench: A Benchmarking Tool for Data Chunking Techniques," in *2023 IEEE Canadian Conference on Electrical and Computer Engineering (CCECE)*, pp. 469–474, 2023.

[28] R. L. Rivest, "The MD5 Message-Digest Algorithm." RFC 1321, 1992.

[29] M. A. Jarah, S. Udayashankar, A. Baba, and S. Al-Kiswany, "The Impact of Low-Entropy on Chunking Techniques for Data Deduplication," in *2024 IEEE 17th International Conference on Cloud Computing (CLOUD)*, pp. 134–140, 2024.

[30] N. Tan, J. Luettgau, J. Marquez, K. Teranishi, N. Morales, S. Bhowmick, F. Cappello, M. Taufer, and B. Nicolae, "Scalable incremental checkpointing using gpu-accelerated de-duplication," in *Proceedings of the 52nd International Conference on Parallel Processing*, pp. 665–674, 2023.

[31] S. A. Hassan, M. M. Mahmoud, A. Hemeida, and M. A. Saber, "Effective implementation of matrix-vector multiplication on intel's avx multicore processor," *Computer Languages, Systems & Structures*, vol. 51, pp. 158–175, 2018.

[32] S. Gueron and V. Krasnov, "Fast quicksort implementation using AVX instructions," *The Computer Journal*, vol. 59, no. 1, pp. 83–90, 2016.

[33] R. L. Bocchino Jr and V. S. Adve, "Vector LLVA: a virtual vector instruction set for media processing," in *Proceedings of the 2nd international conference on Virtual execution environments*, pp. 46–56, 2006.

[34] Intel, "Intel® Intrinsics Guide." https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html.

[35] K. Jacobs, "Independent identically distributed (iid) random variables," in *Discrete Stochastics*, pp. 65–101, Springer, 1992.

[36] VMWare, "VMWare marketplace." https://marketplace.cloud.vmware.com/services, 2023.

[37] Rust, "GitHub - rust-lang/rust: Empowering everyone to build reliable and efficient software.." https://github.com/rust-lang/rust, 2023.

[38] DockerHub, "Container image repository." https://hub.docker.com/.

[39] Linux, "The Linux Kernel Archives." https://www.kernel.org/, 2023.

[40] Redis, "Redis." https://redis.io/, 2023.

[41] MySQL, "MySQL." https://www.mysql.com/, 2023.

[42] T. P. Council, "TPC-C Overview." https://www.tpc.org/tpcc/detail5.asp, 2023.

[43] C. Z. Mooney, *Monte carlo simulation*. Sage, 1997.

[44] D. Duplyakin, R. Ricci, A. Maricq, G. Wong, J. Duerig, E. Eide, L. Stoller, M. Hibler, D. Johnson, K. Webb, A. Akella, K. Wang, G. Ricart, L. Landweber, C. Elliott, M. Zink, E. Cecchet, S. Kar, and P. Mishra, "The Design and Operation of CloudLab," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, (Renton, WA), pp. 1–14, USENIX Association, July 2019.

[45] S. Udayashankar, A. Baba, and S. Al-Kiswany, "VM Images for Deduplication." https://www.kaggle.com/dsv/10561721, 2025.

[46] M. Dutch, "Understanding data deduplication ratios," in *SNIA Data Management Forum*, vol. 7, 2008.

[47] D. Kanter, "Intel's Haswell CPU microarchitecture," *Real World Technologies*, vol. 17, 2012.

[48] M. Gottschlag, T. Schmidt, and F. Bellosa, "AVX overhead profiling: How much does your fast code slow you down?," in *Proceedings of the 11th ACM SIGOPS Asia-Pacific Workshop on Systems*, APSys '20, (New York, NY, USA), p. 59–66, Association for Computing Machinery, 2020.

[49] S. Al-Kiswany, A. Gharaibeh, E. Santos-Neto, G. Yuan, and M. Ripeanu, "StoreGPU: Exploiting Graphics Processing Units to Accelerate Distributed Storage Systems," in *The 17th International Symposium on High Performance Distributed Computing*, HPDC '08, (New York, NY, USA), p. 165–174, Association for Computing Machinery, 2008.

[50] W. Xia, H. Jiang, D. Feng, and Y. Hua, "Similarity and Locality Based Indexing for High Performance Data Deduplication," *IEEE Transactions on Computers*, vol. 64, no. 4, pp. 1162–1176, 2015.

[51] J. Li, Z. Yang, Y. Ren, P. P. Lee, and X. Zhang, "Balancing storage efficiency and data confidentiality with tunable encrypted deduplication," in *Proceedings of the Fifteenth European Conference on Computer Systems*, pp. 1–15, 2020.

[52] C. Song, X. Chen, D. Liu, J. Li, Y. Tan, and A. Ren, "Optimizing the Performance of Consistency-Aware Deduplication Using Persistent Memory," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2023.

[53] M. Fu, D. Feng, Y. Hua, X. He, Z. Chen, W. Xia, F. Huang, and Q. Liu, "Accelerating restore and garbage collection in deduplication-based backup systems via exploiting historical information," in *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, (Philadelphia, PA), pp. 181–192, USENIX Association, June 2014.

[54] Y. Tan, B. Wang, J. Wen, Z. Yan, H. Jiang, and W. Srisa-an, "Improving Restore Performance in Deduplication-Based Backup Systems via a Fine-Grained Defragmentation Approach," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 10, pp. 2254–2267, 2018.

[55] C. Dubnicki, L. Gryz, L. Heldt, M. Kaczmarczyk, W. Kilian, P. Strzelczak, J. Szczepkowski, C. Ungureanu, and M. Welnicki, "HYDRAstor: A scalable secondary storage.," in *FAST*, vol. 9, pp. 197–210, 2009.

[56] D. Bhagwat, K. Eshghi, D. D. E. Long, and M. Lillibridge, "Extreme Binning: Scalable, parallel deduplication for chunk-based file backup," in *2009 IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pp. 1–9, 2009.

[57] F. Ni and S. Jiang, "RapidCDC: Leveraging Duplicate Locality to Accelerate Chunking in CDC-Based deduplication systems," in *The ACM Symposium on Cloud Computing*, SoCC '19, (New York, NY, USA), p. 220–232, Association for Computing Machinery, 2019.

[58] Z. Xu and W. Zhang, "QuickCDC: A Quick Content Defined Chunking Algorithm Based on Jumping and Dynamically Adjusting Mask Bits," in *2021 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom)*, pp. 288–299, 2021.

[59] Y. Won, K. Lim, and J. Min, "MUCH: Multithreaded Content-Based File Chunking," *IEEE Transactions on Computers*, vol. 64, no. 5, pp. 1375–1388, 2015.

[60] 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*, pp. 338–347, IEEE, 2012.

[61] C. Zhang, D. Qi, Z. Cai, W. Huang, X. Wang, W. Li, and J. Guo, "MII: A Novel Content Defined Chunking Algorithm for Finding Incremental Data in Data Synchronization," *IEEE Access*, vol. 7, pp. 86932–86945, 2019.

[62] S. Udayashankar, A. Baba, and S. Al-Kiswany, "SeqCDC: Hashless Content-Defined Chunking for Data Deduplication," in *Proceedings of the 25th International Middleware Conference*, Middleware '24, (New York, NY, USA), p. 292–298, Association for Computing Machinery, 2024.

[63] P. Bartus and E. Arzuaga, "Gdedup: Distributed file system level deduplication for genomic big data," in *2018 IEEE International Congress on Big Data (BigData Congress)*, pp. 120–127, IEEE, 2018.

[64] A. T. Clements, I. Ahmad, M. Vilayannur, J. Li, *et al.*, "Decentralized deduplication in san cluster file systems.," in *USENIX annual technical conference*, vol. 9, pp. 101–114, 2009.

[65] Y. Shin, D. Koo, and J. Hur, "A Survey of Secure Data Deduplication Schemes for Cloud Storage Systems," *ACM Computing Surveys*, vol. 49, Jan 2017.

[66] M. Bellare, S. Keelveedhi, and T. Ristenpart, "Message-locked encryption and secure deduplication," in *Annual international conference on the theory and applications of cryptographic techniques*, pp. 296–312, Springer, 2013.

[67] J. Liu, N. Asokan, and B. Pinkas, "Secure Deduplication of Encrypted Data without Additional Independent Servers," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, (New York, NY, USA), p. 874–885, Association for Computing Machinery, 2015.

[68] D. Harnik, B. Pinkas, and A. Shulman-Peleg, "Side Channels in Cloud Services: Deduplication in Cloud Storage," *IEEE Security and Privacy*, vol. 8, no. 6, pp. 40–47, 2010.

[69] M. Jang, K. Kim, and K. Kim, "The performance analysis of arm neon technology for mobile platforms," in *Proceedings of the 2011 ACM Symposium on Research in Applied Computation*, pp. 104–106, 2011.

[70] R. Shi, G. Schieffer, M. Gokhale, P.-H. Lin, H. Patel, and I. Peng, "Arm sve unleashed: Performance and insights across hpc applications on nvidia grace," in *European Conference on Parallel Processing*, pp. 33–47, Springer, 2025.

[71] M. Gschwind, "Workload acceleration with the ibm power vector-scalar architecture," *IBM Journal of Research and Development*, vol. 60, no. 2-3, pp. 14–1, 2016.

Appendices: Vectorized Sequence-Based Chunking for Data Deduplication

Sreeharsha Udayashankar †, Ali Assem Mahmoud §†, Samer Al-Kiswany †
† *University of Waterloo* § *National Research Council of Canada*
{s2udayas, ali.mahmoud, alkiswany}@uwaterloo.ca

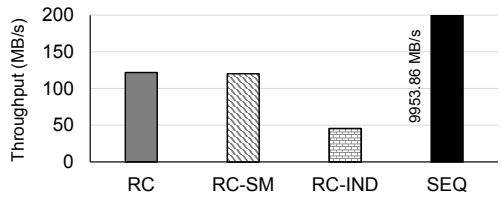


Fig. 1: Chunking throughput on DEB with 16 KB average chunk size. Note that *SEQ* is SeqCDC without vector-acceleration.

Appendix A
Space Savings across chunk sizes

Table I shows the space savings achieved by all the alternatives described in our evaluation on all our datasets. The best-performing algorithm for each dataset and chunk size combination has been highlighted using bold text. Note that there may sometimes be multiple best performers.

SeqCDC (*SEQ*) achieves space savings within 4% of the best performer on all datasets except LNX and within 6% of the best performer on LNX.

Dataset	CDC	4KB	8KB	16KB
DEB	AE	41.99%	33.69%	21.94%
	CRC	41.65%	35.23%	26.59%
	FCDC	43.83%	36.10%	**28.90%**
	GEAR	39.49%	33.12%	28.68%
	RC	44.38%	36.16%	27.94%
	RAM	42.98%	34.21%	22.61%
	TTTD	**45.06%**	37.13%	27.94%
	SEQ	42.77%	**37.76%**	26.97%
DEV	AE	98.00%	97.75%	97.26%
	CRC	98.15%	98.09%	97.70%
	FCDC	98.17%	98.06%	97.91%
	GEAR	98.14%	98.07%	97.92%
	RC	98.21%	**98.12%**	**97.98%**
	RAM	98.05%	97.79%	97.31%
	TTTD	**98.22%**	97.97%	97.97%
	SEQ	98.13%	98.03%	97.80%
LNX	AE	59.41%	45.35%	34.84%
	CRC	62.43%	48.70%	34.84%
	FCDC	59.16%	43.92%	35.89%
	GEAR	57.80%	40.94%	33.32%
	RC	67.02%	50.90%	35.40%
	RAM	57.94%	45.62%	33.56%
	TTTD	**68.46%**	**51.06%**	**35.96%**
	SEQ	63.13%	49.46%	35.63%
RDS	AE	94.66%	92.94%	91.04%
	CRC	94.66%	93.72%	91.81%
	FCDC	93.82%	92.17%	**92.50%**
	GEAR	92.44%	93.01%	90.90%
	RC	94.31%	92.27%	91.76%
	RAM	**95.67%**	**94.09%**	91.99%
	TTTD	95.2%	92.80%	92.49%
	SEQ	94.86%	92.54%	91.13%
TPCC	AE	86.58%	84.96%	81.58%
	CRC	87.23%	86.79%	86.27%
	FCDC	87.18%	86.74%	86.26%
	GEAR	86.96%	86.64%	**86.30%**
	RC	87.24%	86.80%	86.25%
	RAM	86.71%	85.21%	81.68%
	TTTD	**87.29%**	**86.84%**	86.28%
	SEQ	87.04%	86.68%	85.72%

TABLE I: Space savings of CDC techniques. Note that the best algorithm at each configuration is shown using bold text and that *SEQ* is native (unaccelerated) SeqCDC.

Appendix B
Reducing the window size for hash-based algorithms

One of the concerns associated with our evaluation was that hash-based algorithms (such as Rabin-Karp chunking [1]) use large context windows while SeqCDC bases its boundary decisions on the last *SeqLength* bytes, which are typically smaller. This section examines whether hash-based algorithms can achieve better throughputs with small context windows.

Figure 1 shows the throughput achieved by Rabin-Karp chunking with three different configurations and SeqCDC on DEB with a 16KB average chunk size. Note that the y-axis has been cropped to 200 MB/s to avoid SeqCDC skewing the graph. We use an Intel Icelake machine (*sm220u*) from CloudLab [2] for this experiment. The alternatives are described below:

- *RC*: Rabin-Karp [1] chunking.
- *RC-SM*: Rabin-Karp chunking [1] with a small context window of 6 bytes, equal to SeqCDC's *SeqLength*.
- *RC-IND*: Rabin-Karp chunking [1] with no rolling hashes and a small context window of 6 bytes, i.e, independently calculating each window's hash.
- *SEQ*: SeqCDC without vector-acceleration.

Figure 1 shows that there are no throughput gains achieved by reducing the context window of Rabin-Karp chunking [1] to 6 bytes (*RC-SM*). This is because the performance bottleneck is the *sliding operation*, which moves one byte at a time across the input stream, regardless of the context window size. Furthermore, avoiding rolling hashes and calculating each window's hash independently (*RC-IND*) results in a throughput drop of 62.5%, showing that rolling hashes are far more

Algorithm	Avg Chunk Size	DER	Space Savings
AE	16407	1.281	21.94%
CRC	18563	1.362	26.59%
FastCDC	16367	1.408	28.90%
Gear	16034	1.402	28.68%
Rabin	16202	1.388	27.94%
RAM	16407	1.292	22.61%
TTTD	16713	1.388	27.94%
SeqCDC	15760	1.369	26.94%

TABLE II: DEB with 16 KB target average chunk size

Algorithm	Avg Chunk Size	DER	Space Savings
AE	16401	36.5	97.26%
CRC	19102	44.48	97.70%
FastCDC	16351	47.88	97.91%
Gear	16035	47.99	97.92%
Rabin	16254	49.6	97.98%
RAM	16401	37.16	97.31%
TTTD	16648	49.58	97.97%
SeqCDC	15932	45.39	97.80%

TABLE III: DEV with 16 KB target average chunk size

efficient than their independent counterparts, aligning with previous literature [1], [3]. For comparison, SeqCDC achieves 9.66 GB/s on the same dataset and configuration, while the best throughput achieved by all Rabin-Karp configurations is 0.12 GB/s.

Appendix C
Average Chunk Sizes

Tables II - VII show the actual average chunk sizes exhibited by all CDC algorithms on all datasets when targeting an average chunk size of 16 KB. Note that we have omitted the results for other target chunk sizes due to space constraints. We note that SeqCDC exhibits average chunk sizes close to those of other algorithms with our chosen parameters, making deduplication efficiency comparisons fair. We discuss a few specific cases below.

CRC [4] is very sensitive to parameter changes, with small parameter changes resulting in up to 4 KB changes in the exhibited average chunk size. We have chosen parameters for CRC resulting in the closest average chunk size to the target size of 16 KB on all datasets. However, as shown in Tables II - VII, *CRC* exhibits a slightly larger chunk size than other CDC algorithms.

On RDS (Table VI), hash-based algorithms are unable to find chunk boundaries, resulting in up to 10 KB higher chunk sizes than expected. We could not find suitable parameters for these algorithms to address this issue. While using *FastCDC* with a higher normalization factor controls the chunk size, it results in a severe drop in space savings and was not used. To make comparisons fair, we have chosen a configuration for SeqCDC that exhibits an average chunk size close to that of *FastCDC* [5] on this dataset. Note that SeqCDC achieves higher space savings with its actual best configuration.

Finally, on LNX (Table V), while *FastCDC* exhibits a slightly larger average chunk size than expected, other hash-based algorithms such as *Rabin* [1] and *TTTD* [3] do not. Hence, we have chosen SeqCDC's parameters such that its average chunk size is close to that of these algorithms.

Algorithm	Avg Chunk Size	DER	Space Savings
AE	16142	2.704	63.02%
CRC	20335	1.264	20.88%
FastCDC	16765	2.786	64.10%
Gear	16509	2.816	64.48%
Rabin	16549	2.834	64.72%
RAM	16501	2.693	62.87%
TTTD	16282	2.796	64.24%
SeqCDC	16827	2.643	62.16%

TABLE IV: DKR with target 16 KB average chunk size

Algorithm	Avg Chunk Size	DER	Space Savings
AE	16366	1.535	34.84%
CRC	20813	1.534	34.84%
FastCDC	18329	1.56	35.89%
Gear	16671	1.5	33.32%
Rabin	15826	1.548	35.40%
RAM	16155	1.505	33.56%
TTTD	16437	1.562	35.96%
SeqCDC	16404	1.55	35.63%

TABLE V: LNX with 16 KB target average chunk size

Appendix D
Deduplication Elimination Ratio

Deduplication Elimination Ratio (*DER*) has been used to compare the efficiency of CDC algorithms by a few previous studies [6], [7]. It is defined as:

$$DER = \frac{Original\ Size}{Deduplicated\ Size} \quad (1)$$

While this is a comparable metric to *Space Savings* used by our paper, it is harder to interpret [8]. As a result, *Space Savings* has been used by many previous studies [1], [5], [9]–[12]. While we use *Space Savings* as the primary metric everywhere in our paper, we provide the corresponding *DER* values for all datasets and CDC algorithms at a target chunk size of 16 KB in Tables II - VII for reference.

Figure 2 shows the *DER*s achieved by all CDC algorithms on DEB with a target average chunk size of 16 KB. The figure shows that SeqCDC achieves a *DER* of 1.369, higher than *AE* [6] and *RAM* [13], but slightly lower than *Rabin* [1] and

Algorithm	Avg Chunk Size	DER	Space Savings
AE	17013	11.16	91.04%
CRC	23526	12.21	91.81%
FastCDC	21590	13.34	92.50%
Gear	26001	11.2	90.90%
Rabin	23831	12.13	91.76%
RAM	17013	12.48	91.99%
TTTD	21756	13.32	92.49%
SeqCDC	20755	11.24	91.13%

TABLE VI: RDS with 16 KB target average chunk size

Algorithm	Avg Chunk Size	DER	Space Savings
AE	16435	5.428	81.58%
CRC	17413	7.286	86.27%
FastCDC	16410	7.278	86.26%
Gear	16088	7.3	86.30%
Rabin	16350	7.274	86.25%
RAM	16435	5.457	81.68%
TTTD	16135	7.287	86.28%
SeqCDC	16001	7.003	85.72%

TABLE VII: TPCC with 16 KB target average chunk size

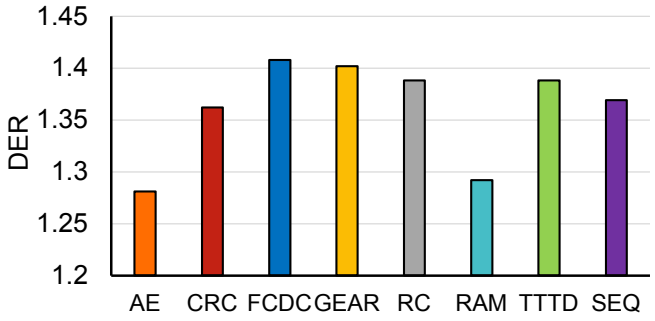


Fig. 2: Deduplication Elimination Ratio on DEB with a 16 KB target average chunk size. Note that *SEQ* is SeqCDC.

FastCDC [5]. This corroborates the trend demonstrated by *Space Savings*. The results are similar across datasets and chunk sizes. Finally, Tables II - VII show that the trends exhibited by *Space Savings* and *DER* are the same across all other datasets as well.

APPENDIX E
DERIVING THE VALUE OF λ_k

A. Renewal-theory Viewpoint

A *renewal process* is a counting process for which the inter-arrival intervals are positive, independent, and identically distributed random variables (i.i.d) [14]. These processes are called renewal processes because the process probabilistically starts over at each arrival epoch. In particular, our process of scanning the stream and reporting contiguous maximal increasing subsequences (components) is thereby a renewal process. Each component is an interval of consecutive indices where the sequence is strictly increasing. Every time an increase fails, a component "terminates" and a new one "restarts." To our practical use, it will be important to learn that renewal processes have the properties of the strong law of large numbers, as this will be fundamental in calculating expectations.

We can start by calculating the probability $p_{\texttt{start}}$ that a given index $i \in \{1, 2, \ldots, n\}$ is the start of a component by noticing that this occurs if and only if $x_{i-1} \geq x_i$, or that $i = 1$. Thus, assuming $n \to \infty$,

$$\begin{aligned} p_{\texttt{start}} &= \frac{1}{n} + (1 - \frac{1}{n}) \Pr(x_{i-1} \geq x_i) \approx \Pr(x_{i-1} \geq x_i) \\ &= \sum_{k=0}^{m-1} \Pr(x_{i-1} = k) \Pr(x_i \leq k) \\ &= \frac{1}{m^2} \sum_{k=0}^{m-1} (k+1) = \frac{m+1}{2m}. \end{aligned} \tag{2}$$

By renewal theory and the strong law of large numbers, the average number of inter-arrival intervals (starts) equals the reciprocal of the mean interval length. Therefore, in our case, the expected component length is the reciprocal of the start (renewal) probability:

$$\mathbb{E}[L] \ = 1/p_{\text{start}} = \ \frac{2m}{m+1}. \tag{3}$$

We now proceed to calculate $\mathbb{E}\big[\lfloor L/k \rfloor\big]$, the expected number of non-overlapping k-blocks per component. At the end we are to use this towards calculating $\lambda_k = \mathbb{E}\big[\lfloor L/k \rfloor\big]/\mathbb{E}[L]$. This relation is justified, since the average event rate per element is

$$\text{Event rate} = \frac{\text{Expected number of events per component}}{\text{Expected number of elements per component}},$$

where the event here is finding a k-length increasing/decreasing contiguous subsequence.

B. Tail Probability

Starting at some random position, let us calculate the probability R_s that, moving forward, we witness s increasing terms in S. Well, the number of ways for choosing s elements is generally m^s. However, increasing terms have to be distinct and uniquely ordered, hence we have $m(m-1)\cdots(m-s+1)/s!$ selections. Thus, for any $s \geq 1$ the probability is given by

$$R_s = \frac{m(m-1)\cdots(m-s+1)}{m^s \, s!} = \frac{\binom{m}{s}}{m^s}. \tag{4}$$

We will use R_s to calculate the probability $T_s = \Pr(L \geq s)$, where L is the length of a component. The connection between the two viewpoints is a size-bias relation. Namely, for each $s \geq 1$ we have

$$R_s \ = \ \frac{1}{\mathbb{E}[L]} \sum_{u=s}^{\infty} T_u,$$

since the fraction of indices whose forward chain length is at least s equals (expected number of such indices per component)/(expected component length). From this relation one can recover T_s from the more easily calculated R_s:

$$T_s \ = \ \mathbb{E}[L] \, (R_s - R_{s+1}). \tag{5}$$

C. Expected Number of Non-overlapping k-blocks

Let k be the block length of interest (in our initial case $k = 5$). We can use the indicator decomposition to apply linearity of expectation:

$$\mathbb{E}\left\lfloor \frac{L}{k} \right\rfloor \ = \mathbb{E} \sum_{j \geq 1} \mathbf{1}\{L \geq kj\} = \sum_{j \geq 1} \Pr(L \geq kj) = \sum_{j \geq 1} T_{kj}. \tag{6}$$

Finally, to get the event rate per element we can now calculate

$$\begin{aligned} \lambda_k \ &= \ \frac{\mathbb{E}\big[\lfloor L/k \rfloor\big]}{\mathbb{E}[L]} = \frac{1}{\mathbb{E}[L]} \sum_{j=1}^{\lfloor m/k \rfloor} T_{kj} \ = \ \sum_{j=1}^{\lfloor m/k \rfloor} (R_{kj} - R_{kj+1}) \\ &= \ \sum_{j=1}^{\lfloor m/k \rfloor} \frac{\binom{m}{kj}}{m^{kj}} - \frac{\binom{m}{kj+1}}{m^{kj+1}} \ = \ \sum_{j=1}^{\lfloor m/k \rfloor} \frac{\binom{m}{kj}(kj)(m+1)}{(kj+1)\, m^{kj+1}} \\ &= \frac{m+1}{m} \sum_{j=1}^{\lfloor m/k \rfloor} \frac{1}{m^{kj}} \binom{m}{kj} \left(1 + \frac{1}{kj}\right)^{-1}. \end{aligned} \tag{7}$$

REFERENCES

[1] A. Muthitacharoen, B. Chen, and D. Mazieres, "A low-bandwidth network file system," in *The eighteenth ACM symposium on Operating systems principles*, pp. 174–187, 2001.

[2] D. Duplyakin, R. Ricci, A. Maricq, G. Wong, J. Duerig, E. Eide, L. Stoller, M. Hibler, D. Johnson, K. Webb, A. Akella, K. Wang, G. Ricart, L. Landweber, C. Elliott, M. Zink, E. Cecchet, S. Kar, and P. Mishra, "The Design and Operation of CloudLab," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, (Renton, WA), pp. 1–14, USENIX Association, July 2019.

[3] T.-S. Moh and B. Chang, "A running time improvement for the two thresholds two divisors algorithm," in *The 48th Annual Southeast Regional Conference*, pp. 1–6, 2010.

[4] F. Ni, X. Lin, and S. Jiang, "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, (New York, NY, USA), p. 86–96, Association for Computing Machinery, 2019.

[5] W. Xia, Y. Zhou, H. Jiang, D. Feng, Y. Hua, Y. Hu, Q. Liu, and Y. Zhang, "FastCDC: A fast and efficient content-defined chunking approach for data deduplication," in *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pp. 101–114, 2016.

[6] Y. Zhang, H. Jiang, D. Feng, W. Xia, M. Fu, F. Huang, and Y. Zhou, "AE: An asymmetric extremum content defined chunking algorithm for fast and bandwidth-efficient data deduplication," in *2015 IEEE Conference on Computer Communications (INFOCOM)*, pp. 1337–1345, IEEE, 2015.

[7] N. Bjørner, A. Blass, and Y. Gurevich, "Content-dependent chunking for differential compression, the local maximum approach," *Journal of Computer and System Sciences*, vol. 76, no. 3-4, pp. 154–203, 2010.

[8] M. Dutch, "Understanding data deduplication ratios," in *SNIA Data Management Forum*, vol. 7, 2008.

[9] A. Liu, A. Baba, S. Udayashankar, and S. Al-Kiswany, "Dedup-bench: A Benchmarking Tool for Data Chunking Techniques," in *2023 IEEE Canadian Conference on Electrical and Computer Engineering (CCECE)*, pp. 469–474, 2023.

[10] M. A. Jarah, S. Udayashankar, A. Baba, and S. Al-Kiswany, "The Impact of Low-Entropy on Chunking Techniques for Data Deduplication," in *2024 IEEE 17th International Conference on Cloud Computing (CLOUD)*, pp. 134–140, 2024.

[11] S. Udayashankar, A. Baba, and S. Al-Kiswany, "SeqCDC: Hashless Content-Defined Chunking for Data Deduplication," in *Proceedings of the 25th International Middleware Conference*, Middleware '24, (New York, NY, USA), p. 292–298, Association for Computing Machinery, 2024.

[12] S. Udayashankar, A. Baba, and S. Al-Kiswany, "VectorCDC: Accelerating Data Deduplication with Vector Instructions," in *USENIX Conference on File and Storage Technologies*, 2025.

[13] R. N. Widodo, H. Lim, and M. Atiquzzaman, "A new content-defined chunking algorithm for data deduplication in cloud storage," *Future Generation Computer Systems*, vol. 71, pp. 145–156, 2017.

[14] R. G. Gallager, "Renewal processes," *Springer US*, pp. 57–102, 1996.