# Accelerating Data Chunking in Deduplication Systems using Vector Instructions

SREEHARSHA UDAYASHANKAR, University of Waterloo, Canada
ABDELRAHMAN BABA, University of Waterloo, Canada
SAMER AL-KISWANY, University of Waterloo, Canada

Content-defined Chunking (CDC) algorithms dictate the overall space savings that deduplication systems achieve. However, due to their need to scan each file in its entirety, they are slow and often the main performance bottleneck within data deduplication. We present VectorCDC, a method to accelerate hashless CDC algorithms using vector CPU instructions, such as SSE / AVX. We analyzed the state-of-the-art chunking algorithms and discovered that hashless algorithms primarily use two data processing patterns to identify chunk boundaries: *Extreme Byte Searches* and *Range Scans*. VectorCDC presents a vector-friendly approach to accelerate these two patterns. Using VectorCDC, we accelerated three state-of-the-art hashless chunking algorithms: RAM, AE, and MAXP. Our evaluation shows that VectorCDC is effective on Intel, AMD, ARM, and IBM CPUs, achieving 8.35×−26.2× higher throughput than existing vector-accelerated algorithms, and 15.3×−207.2× higher throughput than existing unaccelerated algorithms. VectorCDC achieves this without affecting the deduplication space savings.

CCS Concepts: • **Information systems → Cloud based storage**; **Deduplication**; • **Networks → Cloud computing**; • **Computer systems organization → Single instruction, multiple data**.

Additional Key Words and Phrases: Content-defined chunking, SSE/AVX instructions, AVX-512, ARM NEON, IBM VSX

**ACM Reference Format:**
Sreeharsha Udayashankar, Abdelrahman Baba, and Samer Al-Kiswany. 2025. Accelerating Data Chunking in Deduplication Systems using Vector Instructions. 1, 1 (January 2025), 27 pages. https://doi.org/XXXXXXX.XXXXXXX

## 1 Introduction

The amount of data generated and stored on the Internet is growing at an exponential rate [1], and is expected to exceed 180 zettabytes per year in 2025. Storage capacity alone is not well positioned to handle this data influx, with the total installed storage capacity in 2020 only being 6.7 zettabytes [1]. Cloud storage providers instead support this data growth using alternatives such as novel storage paradigms [2, 3], distributed file systems [4, 5] and caches [6, 7], mechanisms such as data deduplication [8, 9], alongside additional investments in data protection [10].

Previous studies by Microsoft [11] and EMC [12] show that a large amount of redundancy exists in the data stored on the cloud, especially in file system backups [8, 11], virtual machine backups [12, 13] and shared documents [14]. The redundancy ratios in these workloads are significant, ranging from 50% to 75%. Mechanisms such as data deduplication [8] and compression [13] are used to conserve storage space by identifying redundant data, eliminating it, and minimizing its storage impact, thereby improving efficiency and reducing costs.

Authors' Contact Information: Sreeharsha Udayashankar, s2udayas@uwaterloo.ca, University of Waterloo, Waterloo, Canada; Abdelrahman Baba, ababa@uwaterloo.ca, University of Waterloo, Waterloo, Canada; Samer Al-Kiswany, alkiswany@uwaterloo.ca, University of Waterloo, Waterloo, Canada.

Data deduplication consists of four phases [9]; *Data Chunking*, *Chunk Fingerprinting*, *Metadata Creation*, and *Data Storage*. Data chunking and chunk fingerprinting are the most computationally intensive [8, 15] of these. While chunk fingerprinting has received significant optimization attention, with faster hashing algorithms [16, 17] and GPUs [18, 19], data chunking optimizations have not kept up (§3.1).

In the data chunking phase, the incoming data is divided into small chunks, typically of size 1 − 64 KB. Numerous data chunking algorithms exist in current literature [20–25] and can be broadly classified into hash-based and hashless algorithms [15]. As chunking occurs whenever new data is uploaded, this phase is on the critical path, and directly impacts system performance.

Previous efforts have explored accelerating chunking by using vector instructions. SS-CDC [26] uses vector instructions to accelerate hash-based data chunking algorithms, such as Rabin-Karp chunking [23] and Gear-based chunking [22]. Unfortunately, this approach only leads to modest improvements in chunking throughput, up to 3.13×, as shown in §3.2. Parallelizing hash-based chunking using vector instructions is complicated because these algorithms use the rolling hash of a sliding window of bytes to detect boundaries, inherently creating a computational dependency between adjacent bytes. Consequently, SS-CDC processes different regions of the data in parallel using slow `scatter` / `gather` vector instructions, limiting its performance (§3.2).

We posit that hashless chunking algorithms are better candidates for vector acceleration. Although they achieve slightly lower space savings compared to their hash-based counterparts (§6), they are up to 2× faster and use simple mathematical operations (e.g., finding a maximum value) that can be accelerated more efficiently using vector instructions. We analyzed state-of-the-art hashless algorithms to understand their design and identify opportunities to leverage vector instructions. We identified that all state-of-the-art hashless algorithms consist of two processing patterns. The first pattern involves finding local minima or maxima in a data region, which we call *Extreme Byte Search*, and the second pattern involves scanning a range of bytes to find values that are greater or lesser than a target value, which we call *Range Scan*. We found that, unlike rolling hash functions, these patterns can be efficiently accelerated using vector instructions.

Using these insights, we present VectorCDC, a technique for accelerating hashless chunking algorithms using vector instructions. VectorCDC uses a novel design to accelerate the two aforementioned patterns. We accelerate the extreme byte searches with a novel *tree-based search* that divides a region of bytes into multiple sub-regions, processes each region using vector instructions, and uses a tree-based approach to combine their results. We accelerate range scans with *packed scanning*, which packs multiple adjacent bytes into vector registers and compares them using a single vector operation.

We implemented VectorCDC using five different vector instruction sets: SSE-128, AVX-256, and AVX-512 on Intel / AMD CPUs; NEON-128 on ARM CPUs; and VSX-128 on IBM Power CPUs. We used VectorCDC to accelerate three state-of-the-art hashless chunking algorithms; *RAM* [24], *AE* [20], and *MAXP* [27], creating *VRAM*, *VAE*, and *VMAXP*, respectively. We compared the performance of our accelerated algorithms with that of state-of-the-art hash-based algorithms, hashless algorithms, and SS-CDC [26] accelerated algorithms using 10 diverse datasets.

Our evaluation (§6) shows that VectorCDC-based algorithms achieve 8.35×−26.2× higher chunking throughput than those accelerated with SS-CDC. *VRAM*, *VAE*, and *VMAXP* also achieve 5.51×−17.6× higher throughput compared to their unaccelerated hashless counterparts, without affecting deduplication space savings. Furthermore, they achieve 15.3×−207.2× higher throughput compared to unaccelerated hash-based algorithms. Finally, VectorCDC-accelerated algorithms retain their performance advantage across all five vector instruction sets.

We have made our code publicly available by integrating it with DedupBench[1] [15]. Due to the large sizes of our datasets (§6), we were unable to release all of them. Instead, we have publicly released one of our datasets (`DEB`) on Kaggle[2] [28] and provided detailed descriptions to facilitate the recreation of others for future experiments, similar to previous literature [20, 21, 24].

The rest of this paper is organized as follows: we discuss relevant background about deduplication and vector instructions in Section 2. Section 3 motivates our work by discussing deduplication performance bottlenecks and the inefficiencies encountered by previous work when accelerating hash-based CDC algorithms. Section 4 outlines VectorCDC's design while Section 5 discusses implementation challenges across vector instruction sets. Section 6 details our evaluation efforts on diverse datasets. We discuss related work in Section 7 and conclude our paper in Section 8.

## 2 Background

Data deduplication consists of four phases [9]:

- *Data Chunking*: Data is divided into small chunks typically of size $1 - 64$KB using a chunking algorithm. All chunking algorithms have configurable parameters that control the average size of generated chunks.
- *Fingerprinting and Comparison*: Chunks are hashed using a collision-resistant hashing algorithm such as MurmurHash3 [16] or SHA-256 [29] to generate *fingerprints*. Fingerprints are compared against those previously seen to identify duplicate chunks.
- *Metadata Creation*: Metadata, i.e., *file recipes* required to reconstruct the original data from stored chunks, are created.
- *Metadata and Chunk Storage*: Non-duplicate chunks and recipes are saved on the storage medium. Fingerprints are stored on the fingerprint database and cached in an in-memory index.

Data chunking and fingerprinting are typically the most computationally intensive phases in deduplication [8]. While fingerprinting has been accelerated up to 53× using GPUs [18, 19] and faster hashing algorithms [16, 30], data chunking acceleration has only received limited attention and adds significant overhead on the deduplication critical path (§3.1).

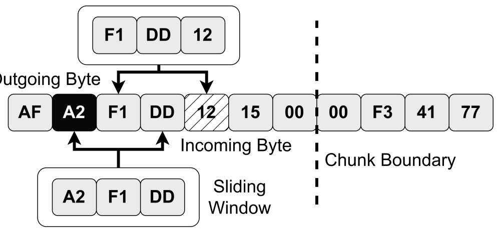
### 2.1 Data Chunking

Data chunking algorithms can generate *fixed-size* or *variable-sized* chunks. Dividing the data into fixed-size chunks is fast, but results in poor space savings on most datasets (§6). This is due to the *byte-shifting problem* [23], where adding a single byte causes all subsequent chunks to appear different, despite the data stream largely being unchanged. Thus, while traditional backup systems such as Venti [31] and OceanStore [32] use fixed-size chunks, modern deduplication systems employ content-defined chunking (CDC) algorithms [23] to generate variable-sized chunks.

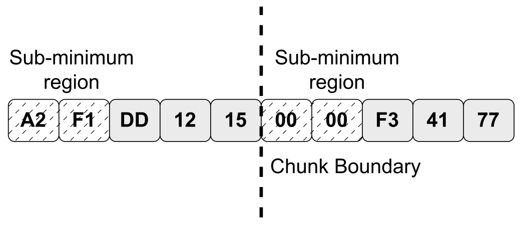
Chunk boundaries in CDC algorithms are derived from the data itself, i.e., they are *content-defined*. These boundaries are chosen such that most byte shifts cause them to shift by the corresponding amount, leaving subsequent chunks unaffected and preserving the ability to detect duplication. Numerous CDC algorithms have been proposed in previous literature [20–25, 27] and can be broadly classified into hash-based and hashless algorithms [15].

*2.1.1 Hash-based algorithms.* These algorithms [21–23, 25] slide a fixed-size window over the data. When the hash value of the window's contents matches a target mask, they insert a chunk boundary, creating a new data chunk lying

[1]https://github.com/UWASL/dedup-bench
[2]https://www.kaggle.com/datasets/sreeharshau/vm-deb-fast25



(a) Sliding window and rolling hash



(b) Minimum chunk sizes and sub-minimum skipping

Fig. 1. Hash-based chunking algorithms

between the current and previous chunk boundaries. Note that these hash-based CDC algorithms are only used during the *Data Chunking* phase and do not affect the *Fingerprinting and Comparison* phase.

Algorithms such as Rabin-Karp chunking [23] and CRC [26] slide a window over the source data and compute the hash of the window's contents using rolling hash algorithms. Figure 1a shows an example of data chunking with such algorithms. In the Rabin-Karp chunking algorithm [23], a chunk boundary is declared when the lower $k$ bits of the sliding window's hash value equals zero. If the current window's hash value does not meet this condition, the window is slid by a byte. To minimize the overhead of recomputing the hash value, the new value is calculated as a function of the old hash value, the incoming byte, and the outgoing byte (Figure 1a), i.e., a *rolling hash*. This creates a dependency between adjacent bytes, complicating acceleration efforts with SIMD instructions (§3.2). Additionally, despite the development of more lightweight rolling hash algorithms such as CRC [26] and Gear-Hash [22], hash-based chunking remains computationally expensive (§6.2).

Some hash-based algorithms like TTTD [25] and FastCDC [21] use minimum and maximum values to limit the chunk sizes. To improve chunking throughput, these algorithms skip scanning data lying before the minimum chunk size at the beginning of each chunk. Figure 1b shows an example of such algorithms with the sub-minimum regions highlighted using a dashed pattern. To offset the impact of skipping the sub-minimum regions and tighten chunk size distributions around the average, FastCDC [21] uses dynamically changing masks, i.e., relaxes the boundary detection condition by reducing $k$ when required. TTTD [25] uses two different boundary masks to do the same.

*2.1.2 Hashless algorithms.* Hashless CDC algorithms such as AE [20], RAM [24], and MAXP [27] treat bytes as individual values and use local minima/maxima to identify chunk boundaries. They also slide one or more windows over the source data but do not use rolling hashes and, as a result, are faster than most hash-based algorithms by 2–3×.

**AE.** Figure 2a shows an example chunk generated by the Asymmetric Extremum (AE) [20] algorithm. AE has two modes of operation: `AE-Min` and `AE-Max`, depending on whether it uses local minima or maxima; Figure 2a shows `AE-Max`. In each chunk, `AE-Max` tries to find a target byte that is greater than all the bytes before it. Once the target byte is identified, `AE-Max` scans a fixed-size window of bytes after the target to identify the maximum-valued byte among them. If the target byte is greater than this maximum-valued byte, it inserts a chunk boundary after the fixed-size window, as shown in Figure 2a.

Similarly, `AE-Min` tries to find a target byte that is less than all the bytes before it. When such a byte is identified, `AE-Min` scans a fixed-size window of bytes after the target to identify the minimum-valued byte within. If the target byte is lesser than this minimum-valued byte, it inserts a chunk boundary after the fixed-size window.

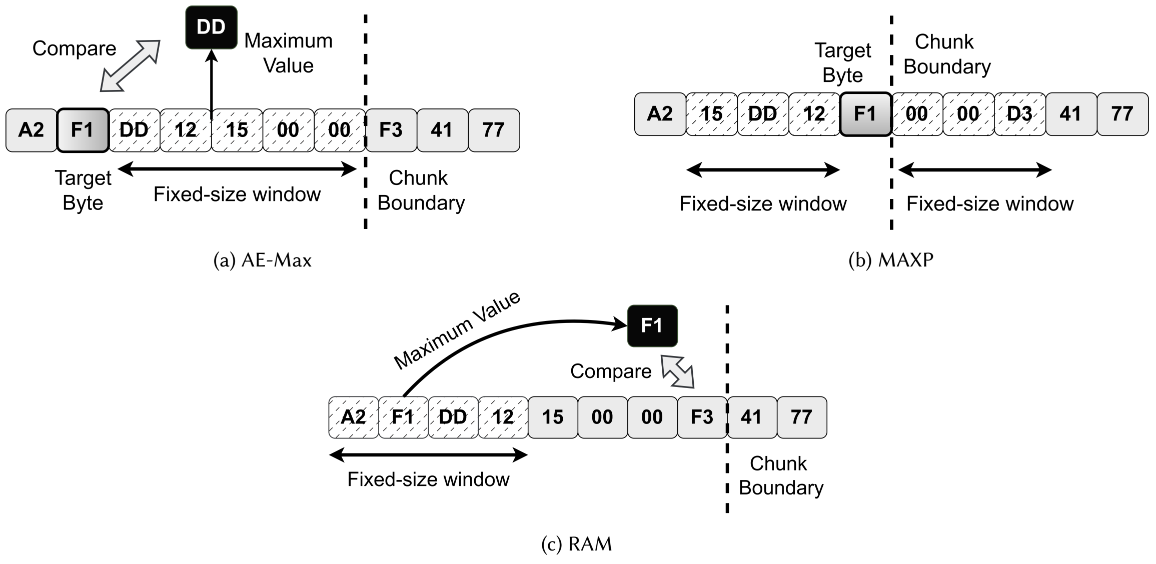


Fig. 2. Hashless chunking algorithms

**MAXP.** Figure 2b shows an example chunk generated by MAXP [27]. MAXP identifies target bytes in the data stream that are local maxima, i.e., they are greater than a fixed number of bytes before and after them. When such target bytes are found, chunk boundaries are inserted at their locations, as shown in the figure. Note that MAXP's window sizes are typically 70-80% smaller than AE [20] and RAM [24] to generate the same target average chunk size. MAXP has also been referred to as Local Maximum Chunking (LMC) in previous literature.

MAXP works by sliding two fixed-size windows over the data, tracking the maximum values from both windows. These windows are located one byte apart, as shown in Figure 2b, and the byte between them is the target byte. When the target byte's value is greater than the maximum value from both windows, a chunk boundary is inserted as the target byte is a local maximum.

**RAM.** Figure 2c shows an example chunk generated by the Rapid Asymmetric Maximum (RAM) [24] algorithm. RAM begins by scanning a fixed-size window at the beginning of each chunk to find the maximum valued byte (F1 in the figure). It then begins scanning at the first byte outside the window, serially comparing these bytes against this maximum value. A chunk boundary is inserted when the first byte that exceeds or equals the maximum is found, e.g., F3 in Figure 2c.

As they do not possess explicit dependencies between adjacent bytes, we argue that hashless algorithms are better candidates for SIMD acceleration efforts.

## 2.2 Deduplication Metrics

Previous literature [15, 21, 23, 33] outlines three important metrics for deduplication systems: *Space Savings*, *Chunk Size Distribution*, and *Chunking Throughput*. We describe these in detail in this section.

*2.2.1 Space savings.* Space savings [15, 33] is one of the primary metrics used to evaluate deduplication systems in production. It represents the overall disk space conserved by using the deduplication system, i.e. size of data stored on disk after deduplication. The space savings achieved are largely dictated by the choice of the data chunking algorithm and its associated parameters. It is defined as:

$$Space\ Savings\ (\%) = \frac{Original\ Data\ Size - Deduplicated\ Data\ Size}{Original\ Data\ Size} \times 100 \tag{1}$$

*2.2.2 Chunking throughput.* Chunking throughput is defined as the speed at which the deduplication system divides incoming data into chunks. As CDC algorithms are content-dependent, they need to scan every byte of an incoming data stream before making content-defined boundary decisions. Their speed depends on their computational complexity. Hash-based algorithms utilize expensive rolling-hash algorithms to determine chunk boundaries (§2.1), typically resulting in lower throughputs than their hashless counterparts. Chunking throughput is defined as:

$$Chunking\ Throughput = \frac{Original\ Data\ Size}{Time\ taken\ to\ generate\ all\ chunk\ boundaries} \tag{2}$$

*2.2.3 Chunk size distribution.* Content-defined chunking (CDC) algorithms generate variable-sized chunks of a target average chunk size. They try to ensure that the sizes of generated chunks are as close to the target average as possible. However, due to underlying algorithmic characteristics, each algorithm has a unique chunk size distribution pattern. For instance, FastCDC [21] exhibits two distinct smooth distributions, changing its pattern at the target average chunk size. This is because it switches masks past the target average size and relaxes boundary conditions. On the other hand, algorithms such as TTTD [25] exhibit a smooth distribution between their minimum and maximum specified chunk sizes. Chunk size distributions are typically represented using cumulative distribution function (CDF) plots.

Space savings are inversely proportional to the target average chunk size, i.e., the greater the average chunk size, the lower the space savings achieved in general [11]. This is because the probability of finding duplicate chunks is higher at smaller chunk sizes. The degree of space savings degradation with increasing chunk size depends on algorithmic and dataset characteristics.

All chunks generated by CDC algorithms are subsequently hashed using a collision-resistant algorithm [16] to generate fingerprints, as described above. The set of unique fingerprints observed thus far is stored in a *fingerprint database*. New incoming chunks are hashed, and their fingerprints are compared against this database to detect duplicates. Thus, smaller and more numerous chunks result in a larger database; specifically, the fingerprint database size is inversely proportional to the chosen average chunk size. A large number of small chunks can negatively impact system throughput, both due to the increased fingerprint database size and the random data accesses caused by these chunks. Thus, CDC algorithms in production typically target average chunk sizes between 2KB–64KB.

## 2.3 Vector Instructions

Vector instructions [34] are special Single-Instruction Multiple-Data (SIMD) instructions supported by most modern processors. They rely on special vector registers for their operations. These registers come in multiple sizes; depending on the width of the vector register they use, vector instructions can be classified into different families [34]. The most common vector register sizes are 128 bits, 256 bits, and 512 bits, i.e., 16, 32, and 64 bytes wide.

Vector instructions support the execution of an operation on multiple pieces of data by packing them into vector registers; for instance, eight 16-bit values *a*-*h* can be densely packed into a 128-bit vector register $V_1$. To add *a*-*h* to eight other values *i*-*p*, we can pack *i*-*p* into another register $V_2$. We can now add them pairwise with a single vector addition operation $VADD\ (V_1, V_2)$ using $V_1$ and $V_2$ as operands, instead of eight separate integer arithmetic operations.

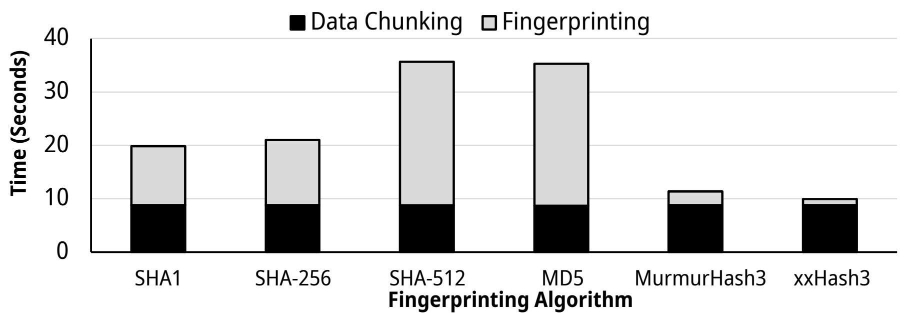


Fig. 3. Time taken for Data Chunking vs Fingerprinting while deduplicating randomized data with FastCDC [21] and an 8 KB average chunk size.

Vector instructions support various arithmetic operations [35], including pairwise addition, subtraction, multiplication, and maximum/minimum on packed values. Additionally, they support logical operations such as bitwise AND (&) and bitwise OR (|). They have been previously used to accelerate matrix multiplication [36], sorting [37], multimedia applications [38], fluid simulations [39], hash tables [40], and relational databases [41]. The supported vector instruction types and their relative performance vary across CPU manufacturers.

*2.3.1 Intel and AMD.* Vector instructions on x86 platforms can be classified into three families: SSE-128, AVX-256, and AVX-512 [34]. SSE-128 instructions use 128-bit registers and have been supported by Intel and AMD processors since 1999 [42] and 2003 [43], respectively. AVX-256 instructions were introduced by Intel and AMD in 2011 [42], and use 256-bit registers. Finally, only a handful of the newest Intel and AMD processors, since 2017 [44] and 2022 [45], which have 512-bit wide vector registers support AVX-512 instructions.

*2.3.2 ARM.* ARM processors have supported NEON-128 instructions, an equivalent to SSE-128, since 2011 [46]. Modern ARM processors also support vector widths of 256 bits and higher with the SVE/SVE2 instruction sets, which have been available since 2021 [47]. These two instruction sets differ in the kinds of instructions supported. For instance, NEON-128 does not support native *VMASK* operations, which are used to create integer masks by extracting one out of every $k$ bits in a vector register, while SVE / SVE2 does. This can lead to performance differences in applications that need the *VMASK* operation [40].

*2.3.3 IBM Power.* IBM's Power [48] architecture supports AltiVec / VSX-128 vector instructions [49], an equivalent to SSE-128, since the 1990s. This instruction set supports equivalents for most SSE instructions but lacks *VMASK* support.

## 3 Motivation

### 3.1 Performance bottlenecks in data deduplication

Although both data chunking and fingerprinting have traditionally been considered the main bottlenecks in deduplication [8], this has changed with the advent of new hashing algorithms and acceleration methods for fingerprinting. Figure 3 is a stacked bar plot showing the time taken by the data chunking and fingerprinting phases during deduplication. For fingerprinting, we use five different collision-resistant hashing algorithms [50–54]. For data chunking, we use FastCDC [21], the fastest unaccelerated chunking algorithm (§6.2), with an 8 KB average chunk size. We use 30 GB of randomized data and an Intel Emerald Rapids machine described in §6 for this experiment.

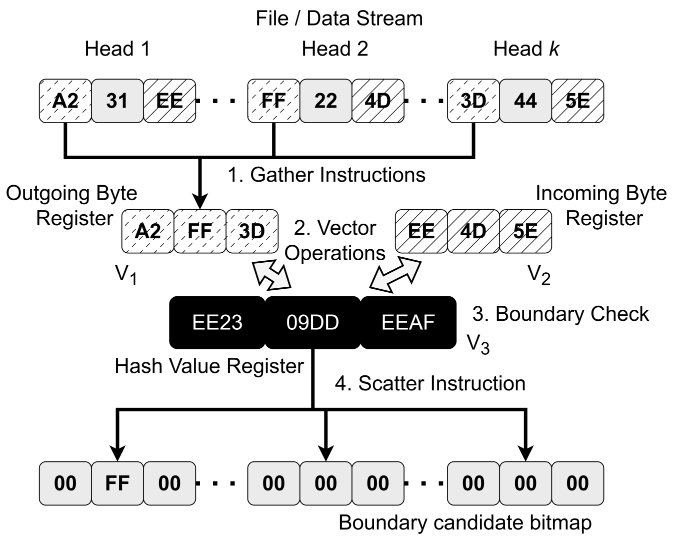


Fig. 4. SS-CDC [26]: Accelerating the rolling hash phase

Figure 3 shows that fingerprinting and data chunking take nearly equal time with traditional collision-resistant hashing algorithms, such as SHA1 [51] and SHA-256 [52]. Fingerprinting takes longer than data chunking with other algorithms, such as MD5 [50] and SHA-512 [52]. This indicates that both phases used to be the performance bottlenecks in deduplication, aligning with previous literature [8].

However, recent research has introduced faster hashing algorithms such as MurmurHash3 [53] and xxHash3 [55] that generate 128-bit digests equivalent to MD5 [56]. These hashing algorithms are being used for fingerprinting [17] or as weak fingerprints followed by a byte-by-byte comparison [57]. Figure 3 shows that fingerprinting takes significantly lower time than chunking with these new hashing algorithms, as they are $10\times -15\times$ faster than their counterparts on CPUs. Using GPUs can further accelerate fingerprinting speeds by up to 53× [18, 19].

Thus, as a result of its computationally intensive nature and position on the critical path, *data chunking is a prime target for acceleration.*

## 3.2 Accelerating hash-based algorithms with vector instructions

To address the data chunking bottleneck, SS-CDC [26] proposed using AVX-512 instructions to accelerate hash-based CDC algorithms. They decouple the rolling hash and boundary detection phases, running the rolling hash on the entire source data to identify boundary candidates in the first phase, and determining boundaries sequentially in the second. This allows both stages to be independently accelerated with AVX instructions.

Figure 4 shows how SS-CDC [26] accelerates the first rolling hash phase of hash-based CDC algorithms. SS-CDC uses AVX-512 registers to create multiple *rolling-heads* (Head 1 - Head *k*), i.e., calculating the rolling hash on bytes from multiple regions of the file independently and in parallel. Each rolling head maintains its own hash value in the hash value register and independently calculates the contributions of incoming and outgoing bytes.

To use vector instructions, they first collect the outgoing bytes for each head into a vector register $V_1$. Similarly, they collect all the incoming bytes into another vector register $V_2$. This is shown in Step 1 in Figure 4 and uses

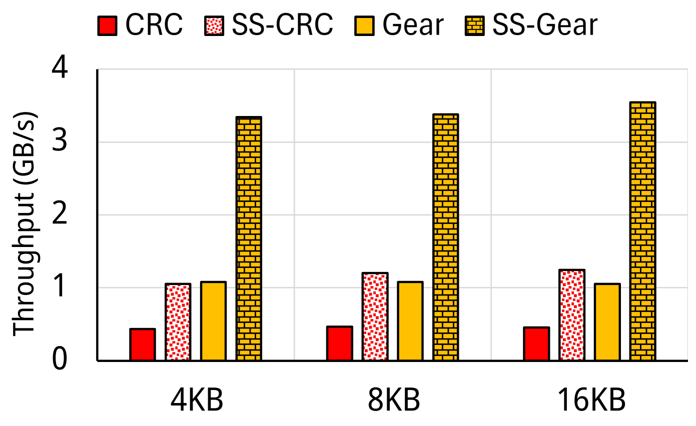


Fig. 5. SS-CDC [26] throughputs on randomized data with AVX-512 instructions

`gather` instructions. The hash values for each head are stored in a separate register $V_3$. In Step 2, SS-CDC removes the contributions of all outgoing bytes from the hash values with a single vector operation and adds the contributions of all incoming bytes with another.

Step 3 compares all the hash values against the pre-specified boundary condition (such as the lower $x$ bits being equal to zero in Rabin-Karp chunking [23]). Whenever any of the hash values match the boundary condition, they mark the current position as a boundary candidate in a separate bitmap in Step 4 using `scatter` instructions. This rolling hash phase is run on the entire incoming data stream/file. In the second phase, they scan the bitmap using vector instructions to determine the actual boundaries among all candidates, taking into account the minimum and maximum chunk sizes.

This approach introduces two problems. First, many hash-based algorithms, such as TTTD and FastCDC, skip scanning data up to the minimum chunk size to improve throughput (§2.1). Decoupling the rolling hash and boundary detection phases causes the rolling hash to be run on the entire incoming data stream, nullifying these optimizations.

Second, to load incoming and outgoing bytes from different regions in the file, SS-CDC [26] uses AVX `gather` instructions. To populate the candidate bitmap when boundary candidates are discovered, they use `scatter` instructions. These `scatter` and `gather` instructions are slow [58], limiting performance gains. Finally, `scatter` instructions are only available on processors supporting certain instruction sets [35], limiting SS-CDC's usage to a handful of the newest Intel and AMD processors (§2.3).

Figure 5 shows the chunking throughput obtained by running SS-CDC accelerated versions of CRC (*SS-CRC*) and Gear-based chunking (*SS-Gear*) [26] against their native unaccelerated counterparts. This experiment used randomized data, an Intel Emerald Rapids machine described in §6 and AVX-512 instructions. We ran each algorithm with chunk sizes of 4 – 16 KB. *SS-CRC* achieves 1.2 GB/s, a speedup of 2.58× over *CRC*. Similarly, *SS-Gear* achieves 3.3 GB/s, a speedup of 3.13× over *Gear*. These small speedups result from the challenges associated with hash-based algorithms that are described above.

Hashless algorithms do not possess explicit dependencies between adjacent bytes. They treat each byte as an independent value and use maximum / minimum values from data regions to determine chunk boundaries. VectorCDC chooses hashless algorithms over their hash-based counterparts as they are better candidates for SIMD acceleration.

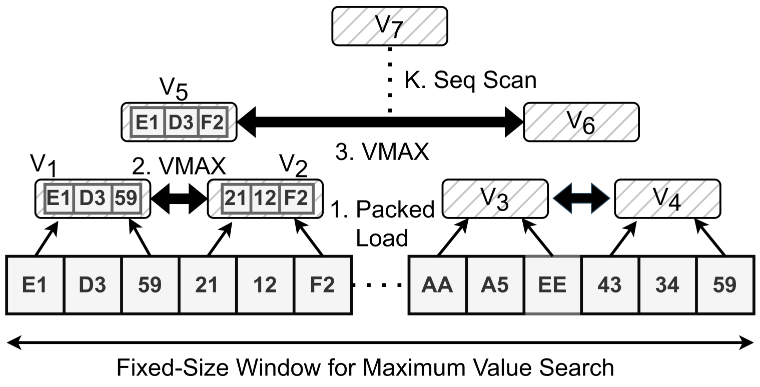

V7
K. Seq Scan
V5
E1 D3 F2
V6
3. VMAX
V1
2. VMAX
V2
E1 D3 59
21 12 F2
1. Packed Load
V3
V4
E1 D3 59 21 12 F2 AA A5 EE 43 34 59
Fixed-Size Window for Maximum Value Search

Fig. 6. Accelerating Extreme Byte Searches. Note that the byte values shown are in hexadecimal format.

## 4 VectorCDC Design

Hashless CDC algorithms such as *AE* [20], *RAM* [24], and *MAXP* [27] slide windows over the source data to determine chunk boundaries. We identify two common processing patterns across all hashless CDC algorithms: the *Extreme Byte Search* and *Range Scan*. We accelerate each of these patterns using different vector-based techniques, which are discussed in detail below.

While we use the AVX-512 instruction set as an example to describe our acceleration techniques in this section, they can be implemented on any CPU with a vector instruction set supporting *VMAX*, *VCMP* and *VMASK* operations. SSE-128 and AVX-256 instruction sets [35] fall under this umbrella, as do ARM and IBM processors with NEON-128 [46] and AltiVec / VSX-128 [48] instructions, respectively. Thus, VectorCDC is compatible with a wide range of processors, unlike SS-CDC [26], which relies on `scatter` instructions only available in AVX-512 instruction sets. Finally, while other minima/maxima-based hashless algorithms can also be accelerated using VectorCDC, their native versions are slower [20, 24, 27] than *AE*, *RAM*, and *MAXP* and have been omitted from the rest of our paper.

### 4.1 Tree-based Extreme Byte Search

Hashless CDC algorithms such as *AE* [20], *RAM* [24], and *MAXP* [27] all consist of a subsequence that identifies the *extreme byte* (maximum/minimum) in a fixed-size window. The size of this window depends upon the expected average chunk size and can be as large as 4 – 8KB. As this subsequence may need to be performed more than once per chunk, we propose accelerating it using a novel *tree-based search approach*. Let us consider the search for a maximum value using AVX-512 instructions (Figure 6). Note that the same method can be used with other vector instruction sets as well as to find minimum values.

We first divide the fixed-size window into smaller sub-regions, loading all the bytes into AVX-compatible `m512i` variables in *Step 1*. We load these bytes in a packed fashion i.e. each `m512i` variable contains 64 adjacent bytes. We then use vector `mm512_max` instructions to find the pairwise maximum among packed byte pairs (*Step 2*). For instance, among the bytes `0xE1` and `0x21`, byte value `0xE1` is the maximum. The resulting pairwise maximums are packed into a destination variable ($V_5$ in the figure).

*Step 3* compares these resulting variables $V_5$ and $V_6$ from *Step 2* using `mm512_max` instructions to find the pairwise maximums. We repeat this process, building a tree of `m512i` variables until we are left with a single variable $V_7$

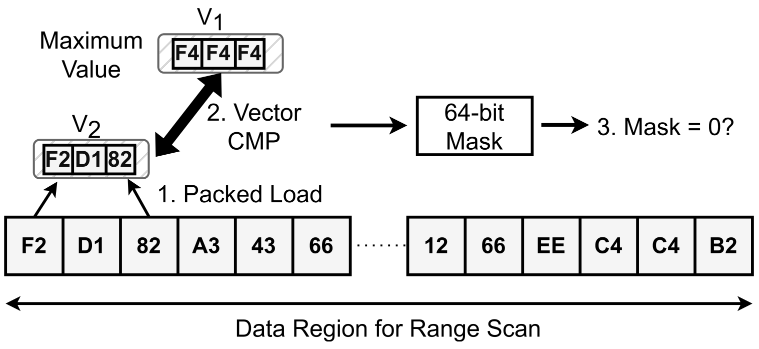


Fig. 7. Accelerating Range Scans. Note that the byte values shown are in hexadecimal format.

containing the maximum-valued 64 bytes from across the entire region. We scan these bytes sequentially in *Step K* to determine the maximum valued byte.

## 4.2 Packed Scanning for Range Scans

Hashless CDC algorithms also consist of a range scan subsequence, where bytes are serially compared against a target value. We propose to accelerate this scanning process using vector instructions. Let us consider a case where we compare bytes sequentially to see if they are greater than or equal to a target value (such as in *RAM* [24]). Figure 7 shows our proposal to accelerate this using *packed scanning* with AVX-512 instructions. Note that the same methods are applicable for other vector instruction sets as well.

We first load the target value (`0xF4` in Figure 7) into an AVX-compatible `m512i` variable $V_1$. We then pack 64 adjacent bytes from the scan region into another `m512i` variable $V_2$. We compare these 2 registers using `mm512_cmpge` vector compare instructions, which generate a 64-bit integer mask containing the comparison results. If this mask has a value greater than 0, a chunk boundary exists within the scanned 64 bytes. Its exact position is determined using the mask value. If the mask equals 0, no boundary exists within the scanned region and we proceed with loading the next 64 bytes into $V_2$ to repeat the process.

*Range Scans* can be run with one of five comparators: Greater-Than (*GT*), Lesser-Than (*LT*), Greater-Than or Equals (*GEQ*), Less-Than or Equals (*LEQ*), and exactly Equals (*EQ*). Each of these comparators uses a different vector compare instruction; for instance, the *GEQ* comparator uses `mm512_cmpge` instructions while the *LEQ* comparator uses `mm512_cmple` instructions. The same comparators also use different comparison instructions in different instruction sets; for instance, the *GEQ* comparator uses `mm512_cmpge` instructions with the AVX-512 instruction set while it uses `mm256_cmpge` with AVX-256.

It is worth noting that our *packed scanning* approach is compatible with sub-minimum skipping. Unlike SS-CDC's approach, chunk boundary detection and insertion can both occur in *Range Scans*, i.e., whenever a chunk boundary is detected, the next `minimum_chunk_size` bytes can be skipped.

## 4.3 Putting it together: AE-Max, AE-Min, MAXP, and RAM

*RAM* [24] first scans a fixed-size window at the beginning of the chunk to find a maximum value (Figure 2c). After this, it inserts a chunk boundary at the first byte outside the window, which is at least as large as the maximum valued byte (§2.1). With VectorCDC, we accelerate *RAM* as a combination of an *Extreme Byte Search* to find a maximum value, followed by a *Range Scan* with the *GEQ* comparator that compares this maximum value against bytes until a chunk boundary is found.

*AE-Max* [20] scans for a byte larger than all the bytes before it i.e., a target byte (Figure 2a). Once found, a fixed-size window after this byte is scanned to determine the maximum valued byte within. If the target byte is larger than the maximum valued byte, a chunk boundary is inserted; otherwise, scanning continues for a new target byte (§2.1). With VectorCDC, we accelerate *AE-Max* as a combination of multiple *Range Scans* with the *GT* comparator to find target bytes, each followed by a single *Extreme Byte Search* for a maximum value.

*AE-Min* [20] scans for a byte with lesser value than all those before it (§2.1). Once found, a fixed-size window after this byte is scanned to determine the minimum value within. If the target byte has a lesser value than the minimum valued byte, a chunk boundary is inserted; otherwise, scanning continues for a new target byte. Similar to *AE-Max*, we accelerate *AE-Min* as a combination of multiple *Range Scans* with the *LT* comparator to find target bytes, each followed by a single *Extreme Byte Search* for a minimum value.

Finally, *MAXP* [27] scans for a target local maxima that is exactly centered between two fixed-size windows (Figure 2b). A chunk boundary is inserted right after such a byte is found (§2.1). Thus, each chunk in *MAXP* can be represented as a combination of multiple *Range Scans* with the *GT* comparator, each followed by two *Extreme Byte Searches* for maximum values.

## 5 Implementation

We accelerate *AE* [20], *MAXP* [27], and *RAM* [24] using VectorCDC with 3000 lines of C++ code. We implemented SSE-128, AVX-256, AVX-512, NEON-128, and VSX-128 versions of all algorithms. We also implemented *Extreme Byte Searches* for minima and maxima, as well as *Range Scan* functionalities with the *GT*, *GEQ*, *LT*, *LEQ*, and *EQ* comparators on all five vector instruction sets. We have made our code publicly available with DedupBench[3] [15].

Note that while ARM processors support *VCMP* and *VMAX* operations, they lack native support for *VMASK* instructions, which are used during range scans to generate a single mask containing the comparison results. This is a common issue encountered by ARM developers trying to port x86 code [59]. We chose an efficient alternative implementation [59] to work around the lack of native *VMASK* support. However, this alternative implementation uses multiple slow NEON-128 instructions, such as `vshrn` and `vreinterpretq`, as opposed to a single x86 `mm_movemask` instruction. As shown in §6.4, this causes accelerated algorithms to achieve lower speedups on ARM CPUs compared to Intel and AMD.

IBM processors also support *VCMP* and *VMAX* operations, but lack native *VMASK* support. However, the same functionality can be achieved using one `vec_bperm` and two `vec_extract` instructions. As these instructions are relatively inexpensive, they are an efficient alternative to *VMASK*. As shown in §6.4, this allows IBM processors to achieve speedups equivalent to or greater than Intel and AMD processors when using VectorCDC.

## 6 Evaluation

In this section, we evaluate VectorCDC against the state-of-the-art CDC algorithms.

**Testbed.** We run all our experiments using machines from the Cloudlab [60] platform. We pick five machines with diverse vector instruction set support; Table 1 shows the vector instruction sets supported by each machine. The details of each machine are as follows:

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

| CPU / CPU Family | SSE-128 | AVX-256 | AVX-512 | NEON-128 | VSX-128 |
|---|---|---|---|---|---|
| Intel Emerald Rapids | ✓ | ✓ | ✓ | – | – |
| Intel Skylake | ✓ | ✓ | ✓ | – | – |
| AMD EPYC Rome | ✓ | ✓ | – | – | – |
| ARM v8 Atlas | – | – | – | ✓ | – |
| IBM Power 8 | – | – | – | – | ✓ |

Table 1. Vector instruction sets supported by the different machines in our testbed.

- *Intel Emerald Rapids:* We use a *c6620* machine from CloudLab Utah, which has a 28-core Intel Xeon Gold 5512U with hyperthreading at 2.1 GHz, 128 GB of RAM, and one Intel NIC each of 25 GBps and 100 GBps. It supports the SSE-128, AVX-256, and AVX-512 vector instruction sets.
- *Intel Skylake:* We use a *c240g5* machine from CloudLab Wisconsin, which has two 10-core Intel Xeon Silver 4114 CPUs with hyperthreading at 2.2 GHz, 192 GB of RAM, one Mellanox 25 GBps NIC, and one onboard Intel 1 GBps NIC. It supports the SSE-128, AVX-256, and AVX-512 vector instruction sets.
- *AMD EPYC Rome:* We use a *c6525-25g* machine from CloudLab Utah, which has a 16-core AMD 7302P CPU with hyperthreading at 3.0 GHz, 128 GB of RAM, and two Mellanox 25 GBps NICs. It supports the SSE-128 and AVX-256 vector instruction sets.
- *ARM v8 Atlas:* We use a *m400* machine from CloudLab Utah, which has an 8-core ARM Cortex A-57 CPU at 2.4 GHz, 64 GB of RAM, and a 10 GBps Mellanox NIC. It supports the NEON-128 vector instruction set.
- *IBM Power 8:* We use an *ibm8335* machine from CloudLab Clemson, which has dual 10-core IBM Power8NVL CPUs at 2.86 GHz with 8 hardware threads per core, 256 GB of RAM, and a 10 GBps Broadcom Xtreme II NIC. It supports the VSX-128 vector instruction set.

While some ARM CPUs released after 2022 support higher vector widths with SVE instructions (§2.3), we could not obtain such a machine for our experiments. Note that all our runs are on the Intel Emerald Rapids machine unless otherwise specified. Our throughput results are the averages of 5 runs, and the standard deviation was less than 5%.

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

- *CRC:* Native (unaccelerated) version of the CRC-32 chunking algorithm from SS-CDC [26].
- *FCDC:* Native version of FastCDC [21].
- *Gear:* Native version of the Gear-hash based chunking algorithm [22].
- *RC:* Rabin's chunking algorithm from LBFS [23].
- *SS-CRC:* AVX-512 version of CRC accelerated using SS-CDC [26].
- *SS-Gear:* AVX-512 version of Gear accelerated using SS-CDC [26].
- *TTTD:* Two-Threshold Two-Divisor algorithm [25].

We also evaluate the following hashless CDC algorithms:

- *AE:* Native version of the Asymmetric Extremum algorithm [20]. We evaluate both *AE-Max* and *AE-Min*.
- *MAXP:* Native version of the MAXP algorithm [27].
- *RAM:* The native Rapid Asymmetric Maximum [24] algorithm.

- *VAE*: Accelerated versions of *AE-Max* and *AE-Min* with VectorCDC.
- *VMAXP*: Accelerated versions of MAXP with VectorCDC.
- *VRAM*: Accelerated versions of RAM with VectorCDC.

Note that for each hashless algorithm accelerated with VectorCDC, we evaluate their SSE-128, AVX-256, AVX-512, NEON-128, and VSX-128 versions on supporting CPU platforms from our testbed (Table 1).

**Datasets.** We use 10 diverse datasets to evaluate VectorCDC; Table 2 shows their details. The datasets represent diverse workloads such as VM backups, database and map backups, web snapshots, and source code. Some datasets, such as `FLOW` and `WIKI`, are similar to those used by previous studies [74]. We have publicly released the `DEB` dataset [4] [28].

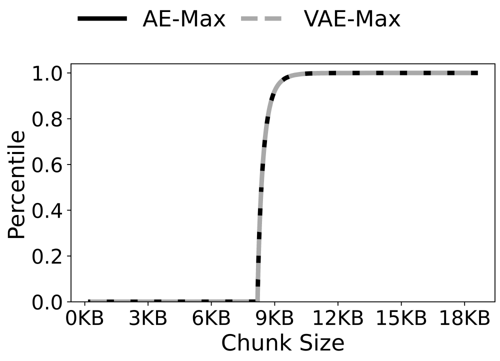
We note that the selected datasets have diverse characteristics. They have varying sizes, ranging from 1 GB for `WIKI` to 981 GB for `MAPS`. They have different file counts; datasets such as `MAPS` and `NEWS` consist of a few large files, while others, such as `FLOW` and `KUBE`, consist of a large number of small files. We include files with varying formats, such as *OSM* [75], *RDB* [76], *TAR* [67], *VMDK / OVA* [77], text files, and binary files across these datasets for comprehensive coverage.

Finally, Table 2 shows the space savings achieved by using fixed-size chunking (*XC*) and the median of those achieved by CDC algorithms (*Median CDC*) on these datasets with 8KB chunks. By comparing *XC* against *Median CDC*, we note that the datasets possess varying degrees of byte-shifting. The difference in space savings between *XC* and *Median CDC* in `FLOW` and `KUBE` is small (less than 6% ), indicating a smaller number of byte-shifts. `DEV` has a moderate amount of byte-shifting, as shown by the ~15% difference between *XC* and *Median CDC*. Finally, CDC algorithms achieve a median

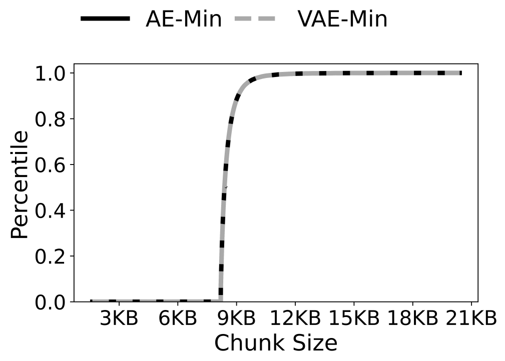
[4] https://www.kaggle.com/datasets/sreeharshau/vm-deb-fast25

| Dataset | Size | Files | Dataset Information | XC | Median CDC |
|---|---|---|---|---|---|
| **DEB** | 40 GB | 65 | Debian [61] VM Images obtained from the VMware Marketplace [62] | 18.98% | 34.64% |
| **DEV** | 230 GB | 100 | Nightly backups of a Rust [63] build server | 83.17% | 98.05% |
| **FLOW** | 8 GB | 630341 | C++ source code for 25 versions of TensorFlow [64] | 90.69% | 91.98% |
| **KUBE** | 1.5 GB | 117344 | Go source code for 5 versions of Kubernetes [65] | 64.52% | 69.42% |
| **LNX** | 65 GB | 160 | Linux kernel distributions [66] in TAR format [67] | 19.87% | 45.62% |
| **MAPS** | 981 GB | 15 | OpenStreetMap [68] backups of Canada extracted using GeoFabrik [69] | 0.10% | 68.57% |
| **NEWS** | 478 GB | 47 | Complete snapshots of a news website across 47 consecutive days in TAR [67] format | 38.95% | 73.80% |
| **RDS** | 122 GB | 100 | Redis [70] snapshots between redis-benchmark runs | 33.54% | 92.94% |
| **TPCC** | 106 GB | 25 | 25 snapshots of a MySQL [71] VM running TPC-C [72] | 37.39% | 86.64% |
| **WIKI** | 1 GB | 3134 | Snapshots of the largest Wikipedia article [73] across multiple days, chosen for extreme versioning. | 1.31% | 72.37% |

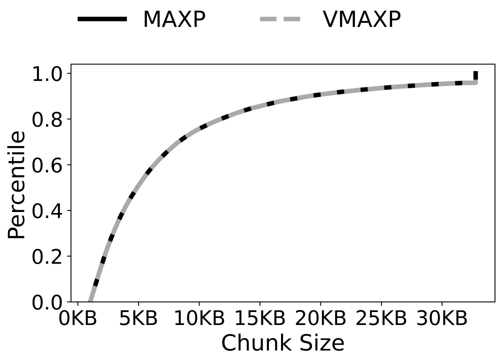
Table 2. Dataset Information. Note that *XC* represents the space savings achieved by fixed-size chunking with 8KB chunks while *Median CDC* is the median space savings achieved by CDC algorithms with an 8KB average chunk size.



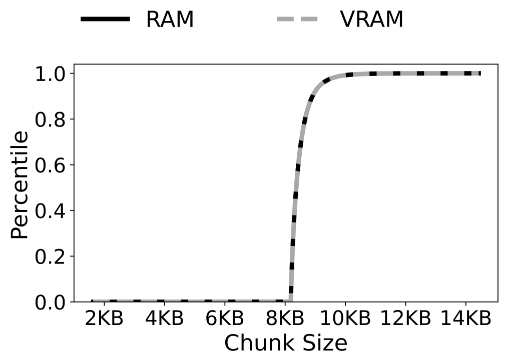
(a) *AE-Max / VAE-Max*



(b) *AE-Min / VAE-Min*



(c) *MAXP / VMAXP*



(d) *RAM / VRAM*

Fig. 8. Chunk size CDFs of hashless algorithms and their AVX-512 accelerated versions on TPCC with an 8KB average chunk size

of more than 2× higher space savings than *XC* on DEB, LNX, MAPS, NEWS, RDS, TPCC, and WIKI, indicating that these data sets have a large degree of byte-shifting.

**Metrics.** We evaluate the space savings, chunk size distribution, and chunking throughput achieved by each alternative on all the described datasets.

## 6.1 Space Savings and Chunk Size Distributions

Figures 9a - 9j show the space savings achieved by all alternatives with 8KB chunks across datasets. We omit the results for other chunk sizes as the trends were similar.

*6.1.1 Vector-acceleration Impact.* Vector-acceleration does not impact the space savings achieved by CDC algorithms. Consequently, for clarity, we omit the space savings results for vector-accelerated algorithms from Figure 9. This aligns with the results previously observed for *SS-CRC* and *SS-GEAR* [26].

AVX-512 acceleration does not impact the chunks generated by hashless algorithms. We compared the generated chunks of vector-accelerated algorithms with their native counterparts and verified that they were identical. We present only the chunk size distribution comparison in this paper due to space constraints.

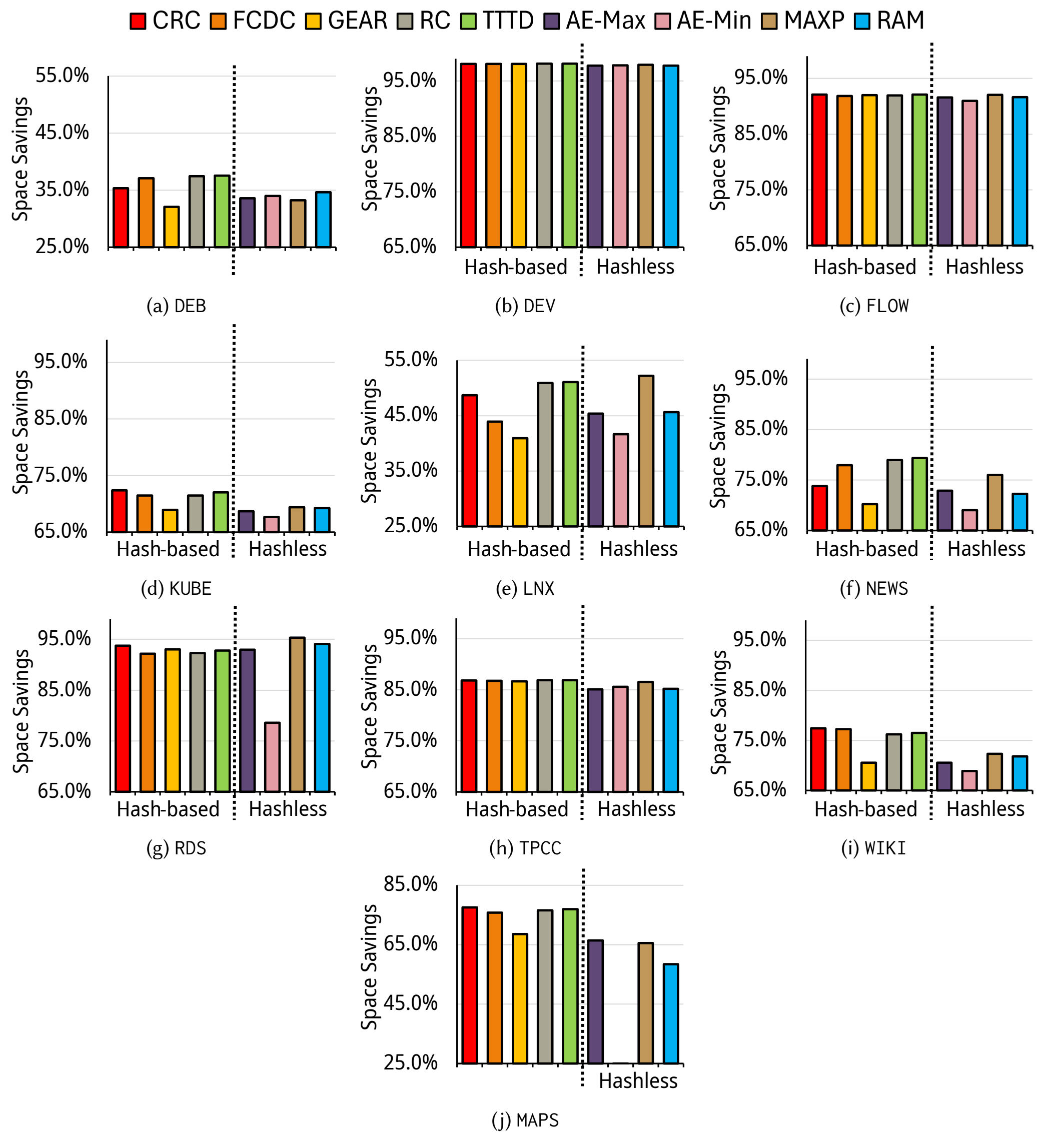


Fig. 9. Space Savings with 8KB chunks. Note that the legend entries are in the same order as the plot bars.

Figure 8 shows the chunk size distributions exhibited by *AE-Max*, *AE-Min*, *MAXP*, and *RAM* compared against their AVX-512 versions accelerated with VectorCDC. Note that each figure is a cumulative frequency (CDF) [78] plot. We use a target average chunk size of 8KB and the TPCC dataset for this experiment. The results for other datasets and chunk sizes were similar and have been omitted for clarity.

*6.1.2 Hash-based vs Hashless.* Hashless algorithms are generally competitive with hash-based ones in space savings. The best among the hashless algorithms achieves slightly lower space savings than the best hash-based algorithm on some datasets, such as DEB and NEWS (Figures 9a and 9f). On the other hand, the best hashless algorithm outperforms all hash-based algorithms on other datasets, such as LNX and RDS (Figures 9e and 9g). Overall, the best hashless algorithms achieve space savings values within 11% of the best hash-based ones across all datasets and chunk sizes.

*6.1.3 Hashless algorithm comparison.* The performance of the hashless algorithms depends on the dataset's characteristics and the average chunk size. For instance, *RAM* achieves the highest space savings on DEB (Figure 9a) while *MAXP* does so on TPCC (Figure 9h). This shows that *accelerating all hashless algorithms is important, as the performance of each algorithm depends on the dataset's characteristics.*

Notably, *AE-Min* is adversely affected by the byte-shifting pattern in MAPS, causing it to achieve only 8.89% in space savings while other CDC algorithms achieve 58%-78%.

Finally, while *MAXP* achieves higher space savings than *RAM* and both *AE* variants on many datasets, the space savings difference between it and the next best hashless algorithm is small.

*6.1.4 Differences among datasets.* Hashless algorithms perform equivalent to or better than their counterparts on virtual machine and database backups, such as DEV, RDS, and TPCC. Source-code datasets demonstrate mixed results, with hash-based algorithms slightly edging out hashless ones on KUBE, equivalence on FLOW, and hashless algorithms being better on LNX. File formats largely do not influence space savings.

## 6.2 Chunking Throughput

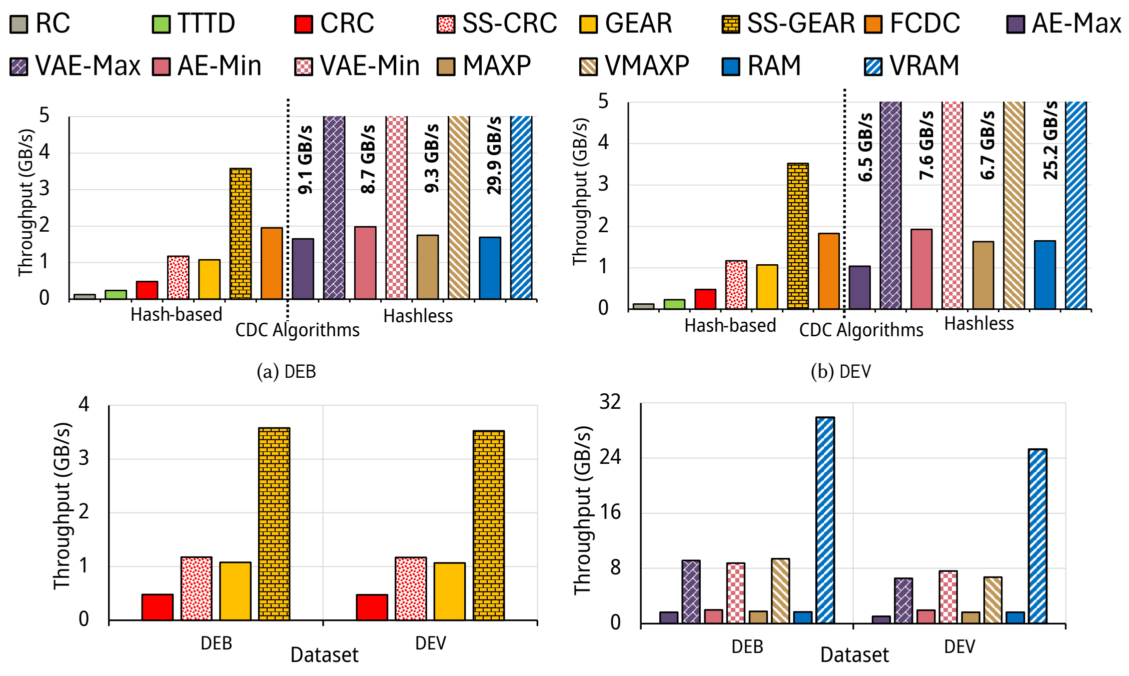
Figures 10a and 10b show the throughput achieved by all algorithms on DEB and DEV with a chunk size of 8KB. Note that vector-accelerated algorithms are shown with patterned bars and that we have cropped the y-axis to 5 GB/s to avoid the figures being skewed by *VRAM*. The results on other datasets and chunk sizes had similar trends and have been omitted for clarity.

*6.2.1 Throughput Comparison.* Figures 10a and 10b show that hashless algorithms accelerated with VectorCDC achieve 4× to 15× higher throughput than all accelerated CDC algorithms. *VRAM*, the fastest accelerated hashless algorithm, achieves 8.35× and 15.3× higher throughput than *SS-GEAR* and *FastCDC*, the fastest accelerated and unaccelerated hash-based algorithms, respectively. Additionally, *VRAM* achieves 207.2× higher throughput than *RC*, a popular but slow hash-based CDC algorithm.

Among unaccelerated hash-based algorithms, *Gear* [22], *CRC* [26], and *FastCDC* [21] are the fastest. We accelerated each of these using SS-CDC [26]; *SS-GEAR* achieves 3 × higher throughput compared to its unaccelerated version, and *SS-CRC* achieves 2 × higher throughput that unaccelerated CRC. We did not observe any speedup when accelerating *FastCDC* [21] with SS-CDC [26]. One of the main throughput optimizations used by *FastCDC* is sub-minimum skipping (§2.1). However, as noted in §3.2, decoupling the rolling-hash phase from the boundary identification phase eliminates the throughput benefits of minimum chunk size skipping, nullifying any speedup provided by vector-acceleration.

*6.2.2 Vector-acceleration benefits.* Figures 10c and 10d compare the throughput benefits of accelerating hash-based and hashless algorithms with AVX-512 accelerated algorithms on DEB and DEV.

Accelerating hash-based algorithms (Figure 10c) using SS-CDC achieves a speedup of 2.45 – 3.32×. On the other hand, the hashless algorithms *VAE-Max*, *VAE-Min*, *VMAXP*, and *VRAM* achieve speedups of 5.1×, 4.43×, 5.36×, and 17.69× over their respective native counterparts, achieving throughputs in the range of 6.5 GB/s–29.9 GB/s. Thus,



(a) DEB

(b) DEV

(c) Hash-based algorithms with SS-CDC [26]

(d) Hashless algorithms with VectorCDC

Fig. 10. Chunking Throughput with AVX-512 instructions and 8KB chunks. Note the different scales in Figures 10c and 10d, and that the legend entries are in the same order as the plot bars from Figures 10a and 10b.

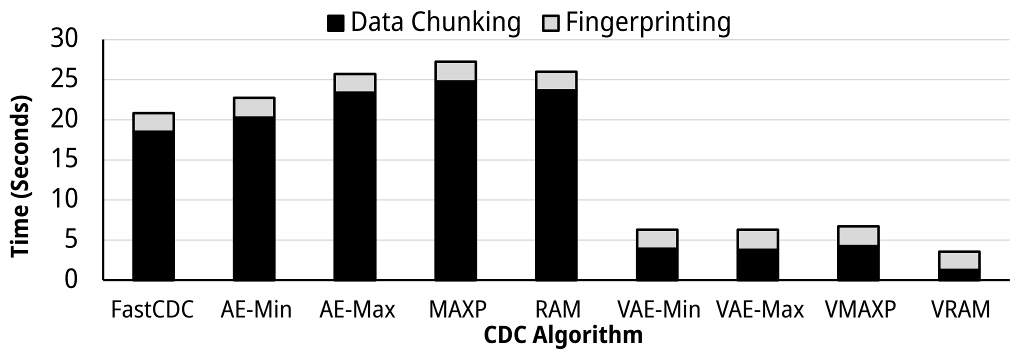
vector instructions can be leveraged far more efficiently for hashless algorithms, *proving that hashless algorithms are better candidates for vector-acceleration than their hash-based counterparts*.

Figure 10d shows that *VRAM* achieves higher throughputs than *VAE-Max*, *VAE-Min*, and *VMAXP*. This is because *VAE* requires multiple iterations of *Range Scan* per chunk, each followed by an *Extreme Byte Search*, while *VRAM* only requires one iteration of each (§4). Similarly, *VMAXP* requires multiple *Range Scans*, each followed by two *Extreme Byte Searches*. For a given target average chunk size, the size of the *Extreme Byte Search* regions in *MAXP* is 70 − 80% smaller than the search region in *AE*. This allows *VMAXP* to achieve speeds similar to *VAE-Max* and *VAE-Min* despite needing an extra *Extreme Byte Search*.

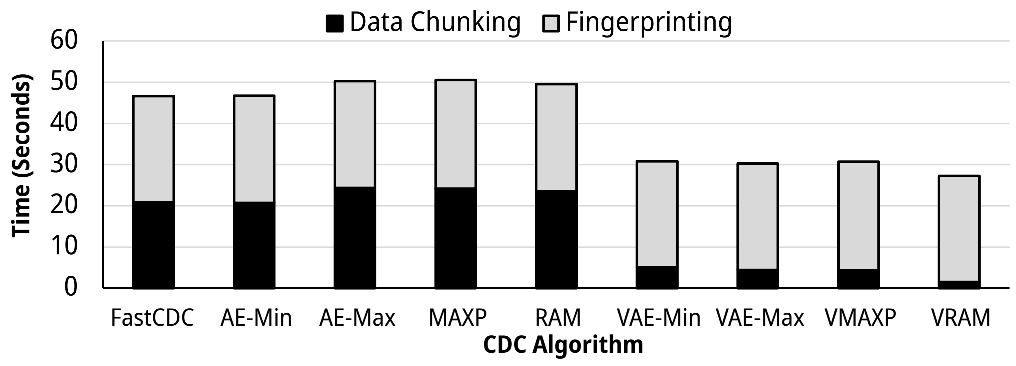
Thus, *RAM is inherently more vector-friendly than AE and MAXP*. However, note that *VAE* and *VMAXP* are still faster than every other CDC algorithm.

*6.2.3 Deduplication performance bottlenecks.* Figure 11 shows the time taken by the chunking and hashing phases in the deduplication pipeline on DEB with an 8KB average chunk size. We omit the results for other datasets as they were similar. We used two fingerprinting algorithms; xxHash3, the fastest but generates a 128-bit digest, and SHA-256, slower but offers higher collision resistance with a 256-bit digest (§3.1). We ran this experiment on the Intel Emerald Rapids machine. We use AVX-512 versions of hashless CDC algorithms, accelerated with VectorCDC.

Figure 11 shows that with xxHash3 (Figure 11a), data chunking takes significantly longer than fingerprinting with unaccelerated algorithms. On the other hand, *VAE-Min*, *VAE-Max*, *VMAXP*, and *VRAM* show data chunking times similar



(a) xxHash3 - 128-bit digest



(b) SHA256 - 256-bit digest

Fig. 11. Time taken for Data Chunking vs Fingerprinting on DEB with an 8 KB average chunk size, and AVX-512 instructions for acceleration.

to or lower than fingerprinting. For instance, with *VRAM*, data chunking takes 1.29 seconds while fingerprinting takes 2.27 seconds.

With SHA-256 (Figure 11b), we observe that fingerprinting takes as long as data chunking with unaccelerated CDC algorithms. On the other hand, *VAE-Max*, *VAE-Min*, *VMAXP*, and *VRAM* take significantly lower time to run.

These results show that *VectorCDC effectively alleviates the data chunking bottleneck in the deduplication pipeline.*

## 6.3 Throughput breakdown - Extreme Byte Search vs Range Scan

The throughput impact of each processing pattern depends on algorithmic and dataset characteristics. Figure 12 shows the individual impact of accelerating *Extreme Byte Search* and *Range Scan* using *VRAM* on the DEB and LNX datasets with an 8KB chunk size. *VRAM-EBS* and *VMAXP-EBS* represent *RAM* and *MAXP* running with only *Extreme Byte Search* acceleration, while *VRAM-512* and *VMAXP-512* use both accelerated patterns.

Figure 12a shows that on DEB, *VRAM-EBS* achieves a throughput of 18.5 GB/s compared to *RAM* at 1.7 GB/s. Accelerating *Range Scan* provides an additional speedup of 11.4 GB/s. On the other hand on LNX, *VRAM-EBS* only achieves 2.7 GB/s compared to *RAM* at 2 GB/s. Accelerating *Range Scan* provides an additional speedup of 27.6 GB/s.

Thus, each pattern has a balanced impact on *VRAM*'s throughput on `DEB`, while *Range Scan* primarily contributes to throughput on `LNX`, indicating that dataset characteristics affect the throughput breakdown.

The throughput breakdown also varies across algorithms; for instance, accelerating *Extreme Byte Searches* has differing impacts on the throughputs of *RAM* and *MAXP*. While Figure 12a shows that *VRAM-EBS* achieves significantly higher throughput than *RAM* on `DEB`, Figure 12b shows that *VMAXP-EBS* only achieves small speedups over *MAXP*, i.e., *Range Scan* acceleration contributes more to throughput on *VMAXP* than it does on *VRAM*.

These results are directly tied to the number of bytes processed by the algorithms on both datasets. Figure 13 shows the percentage shares of bytes processed by *Extreme Byte Searches* and *Range Scans*, for all hashless algorithms on `DEB` and `LNX`. As seen in Figure 13a, the percentage shares differ across algorithms. For instance, *RAM* processes 96.70% and 3.30% of bytes on `DEB` with *Extreme Byte Search* and *Range Scan*, respectively. On the other hand, *MAXP* processes 10.26% and 89.74% of bytes with *Extreme Byte Search* and *Range Scan*, respectively. Additionally, this percentage varies across datasets, as seen by the differences between Figures 13a and 13b.

Thus, *accelerating both phases using vector instructions is crucial to performance, as the impact of each phase depends on dataset and algorithmic characteristics.*

## 6.4 VectorCDC across different vector instruction sets

VectorCDC is compatible with a large range of platforms that support vector instructions such as SSE-128, AVX-256, NEON-128, and VSX-128 (§2.3). This is unlike SS-CDC [26] which requires CPUs with `scatter/gather` instruction support. Such CPUs are present only in a small percentage of datacenter nodes today.

While §4 discusses VectorCDC's design using AVX-512 instructions, the same methods can be applied to any vector instruction set that supports `VCMP`, `VMAX`, and `VMASK` operations. In this section, we evaluate VectorCDC's performance with other such vector instruction sets. We ran this experiment using the `DEB` dataset and an average chunk size of 8 KB.

*6.4.1 AMD EPYC Rome.* Figure 14a shows the throughputs achieved by hashless algorithms accelerated with VectorCDC on an AMD EPYC Rome machine. As shown in Table 1, the AMD machine only supports SSE-128 and AVX-256 instructions. All four hashless algorithms in Figure 14a show speedups over their native versions with both instruction sets. For instance, *AE-Max* achieves 2.12× and 3.43× speedups with SSE-128 and AVX-256 instructions, respectively. Similar to the results in §6.2 with AVX-512 instructions, *RAM* achieves the highest throughput of all algorithms with both SSE-128 and AVX-256 instructions.

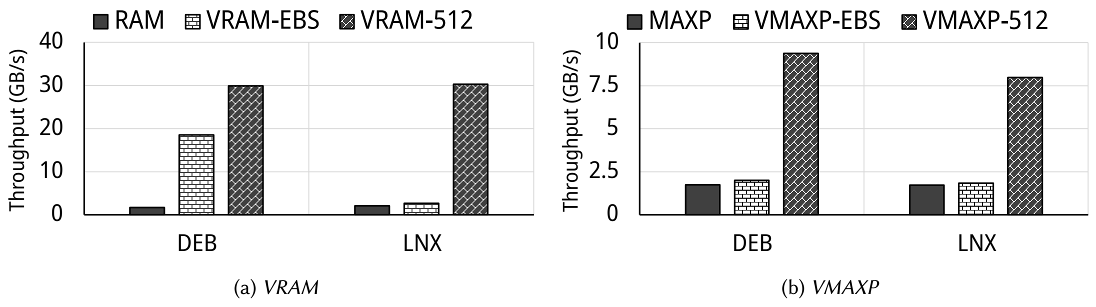


Fig. 12. Throughput Breakdown with AVX-512 instructions. Note that *VRAM-EBS* and *VMAXP-EBS* represent *VRAM* and *VMAXP* with only Extreme Byte Search accelerated.

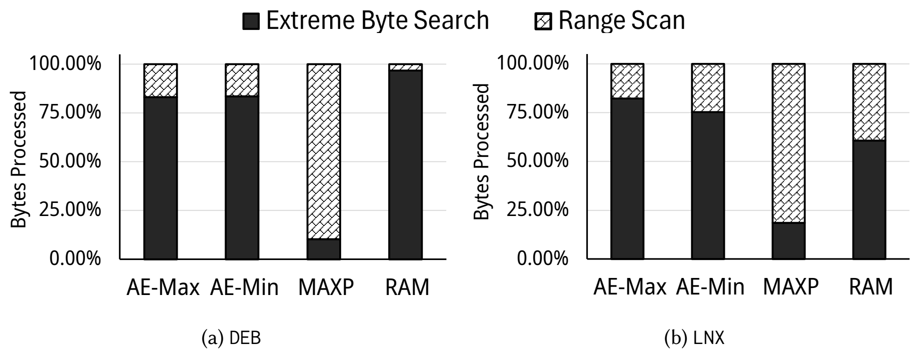


Fig. 13. Percentage share of bytes processed using *Extreme Byte Search* and *Range Scan* by hashless CDC algorithms on DEB and LNX

*6.4.2 Intel Emerald Rapids and Skylake.* Figures 14b and 14c show the throughputs achieved by hashless algorithms accelerated with VectorCDC on Intel Emerald Rapids and Skylake machines. As shown in Table 1, these machines support SSE-128, AVX-256, and AVX-512 instructions. All four hashless algorithms in Figures 14b and 14c achieve speedups over their unaccelerated versions with all instruction sets. For instance, in Figure 14c, *AE-Max* achieves 2.29×, 4.91×, and 6.71× speedups with SSE-128, AVX-256, and AVX-512 instructions, respectively. Similar to the results in §6.2 with AVX-512 instructions, *RAM* achieves the highest throughput of all algorithms with both SSE-128 and AVX-256 instructions.

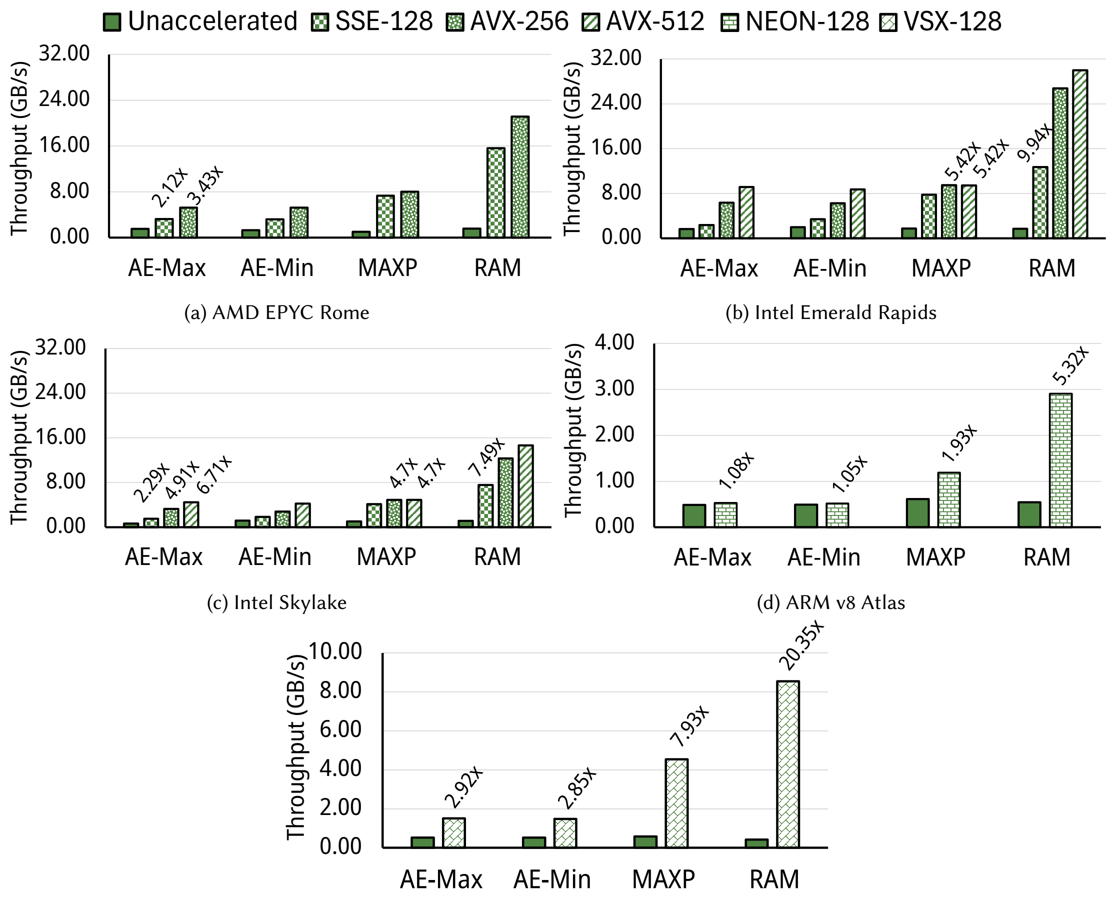
On both platforms, all algorithms also benefit from increasing vector widths; that is, higher vector widths lead to higher throughput. The only exception is *MAXP*, which does not gain as much as the other algorithms with AVX-512 instructions over AVX-256. This is related to the small window sizes used by *MAXP* for its *Extreme Byte Search* phases, which do not benefit from high vector widths. However, *MAXP* still achieves 4.7× and 5.42× speedups with AVX-512 instructions over its unaccelerated version, on the Skylake and Emerald Rapids machines, respectively.

*6.4.3 ARM v8 Atlas.* Figure 14d shows the throughputs achieved by hashless algorithms accelerated with VectorCDC on an ARM v8 Atlas machine. As shown in Table 1, the machine only supports NEON-128 instructions, an ARM equivalent to SSE-128. While the instruction set supports *VMAX* and *VCMP* operations, it lacks native support for *VMASK* operations (§5). *RAM* achieves the highest throughput among all accelerated hashless algorithms at 2.91 GB/s.

All hashless algorithms achieve lower speedups on ARM with NEON-128 instructions, when compared to SSE-128 instructions on Intel and AMD machines. *AE-Max* and *AE-Min* are especially affected, achieving only 1.08× and 1.05× speedups, i.e. 8% and 5% gains with NEON-128 over their unaccelerated versions. This is largely due to the lack of native *VMASK* support, which affects *Range Scans*. While our implementation uses an alternative method to achieve the same functionality, it uses four NEON-128 instructions instead of a single SSE-128 *VMASK* instruction.

However, *MAXP* and *RAM* still achieve 1.93× and 5.32× speedups, respectively, showing that VectorCDC remains beneficial on ARM platforms with NEON-128 support. Note that these numbers are expected to improve in ARM platforms supporting SVE/SVE2 instructions [47], as they offer native *VMASK* support. However, as we could not obtain such a platform for our evaluation, we leave a detailed SVE/SVE2 performance review as future work.

*6.4.4 IBM Power 8.* Figure 14e shows the throughputs achieved by hashless algorithms accelerated with VectorCDC on an IBM Power 8 machine. As shown in Table 1, this machine only supports VSX-128 instructions, an IBM equivalent



(a) AMD EPYC Rome

(b) Intel Emerald Rapids

(c) Intel Skylake

(d) ARM v8 Atlas

(e) IBM Power 8

Fig. 14. Accelerating hashless algorithms with VectorCDC across processor architectures, on DEB at an 8KB average chunk size. Data labels show speedups over the respective native algorithm for the specific cases discussed in text. Note the different y-axis scale on Figures 14d and 14e.

to SSE-128. This instruction set lacks support for native *VMASK* operations as well (§5). *RAM* achieves the highest throughput among all accelerated hashless algorithms, at 8.54 GB/s.

Unlike ARM, all hashless algorithms exhibit considerable speedups on IBM Power 8 with VectorCDC. *AE-Max* and *AE-Min* achieve speedups of 2.92× and 2.85× respectively. *MAXP* and *RAM* achieve speedups of 7.93× and 20.35× respectively. Furthermore, all hashless algorithms accelerated with VSX-128 instructions achieve speedups equivalent to or greater than their counterparts accelerated with SSE-128 on Intel and AMD machines. For instance, *RAM* achieves a speedup of 20.35× with VSX-128 on IBM Power 8 while it achieves a speedup of 7.49× and 9.94× with SSE-128 on Intel Emerald Rapids and AMD EPYC Rome, respectively.

This is because, despite the lack of native *VMASK* instruction support, the alternative implementation using `vec_bpermq` is efficient and uses just two fast VSX-128 instructions.

## 6.5 Evaluation Summary

To summarize, the main takeaways from our evaluation are the following:

- VectorCDC-based hashless algorithms achieve 15.3×–207.2× and 8.35×–26.2× higher throughput than unaccelerated and vector-accelerated hash-based algorithms respectively, showing that hashless algorithms are better candidates for vector acceleration (§6.2).
- VectorCDC effectively alleviates the data chunking performance bottleneck in the deduplication pipeline (§6.2.3).
- Accelerating both *Extreme Byte Search* and *Range Scan* is important because their individual impact depends on dataset and algorithmic characteristics (§6.3).
- VectorCDC provides benefits across different processor architectures, and is compatible with a wide range of vector instruction sets (§6.4).
- Accelerating hashless algorithms with VectorCDC does not impact their space savings and generates chunks identical to their unaccelerated counterparts.
- Hashless algorithms achieve space savings values comparable to or better than those of their hash-based counterparts on real-world datasets (§6.1). The best performing hashless algorithm varies by dataset, showing that accelerating all of them is equally important.

## 7 Related Work

*7.0.1 Chunking optimizations.* Many efforts have been made to optimize data chunking. MUCH [79] and P-Dedupe [80] use multiple threads to accelerate chunking. RapidCDC [81] sometimes skips data chunking by predicting the next chunk boundary based on historical data, but requires maintaining additional metadata. Bimodal Chunking [82] initially splits the data into large chunks, and then divides duplicate adjacent chunks into smaller ones, to enhance space savings. VectorCDC is compatible with all of these approaches, as they build on top of existing CDC algorithms.

Previous work [83] that analyzes the characteristics of chunks generated by CDC algorithms, is orthogonal to VectorCDC, as vector acceleration does not affect generated chunks.

Our previous paper at USENIX FAST 2025 [84] presented VectorCDC's design, but does not discuss accelerating *MAXP* [27] or VectorCDC's performance on varying CPU architectures. Additionally, it does not present a comprehensive evaluation of VectorCDC's capabilities.

*7.0.2 Deduplication optimizations.* Several other efforts exist to optimize the other phases of the deduplication pipeline. StoreGPU [19] and GPU-Dedup [18] accelerate chunk hash computation using GPUs. SiLo [85], Sparse Indexing [86] and Extreme Binning [87] optimize hash indexing. HYDRAStor [88] is a distributed deduplication system that focuses on data placement. Several studies incorporate delta compression after deduplication to further compress similar but non-duplicate chunks [89–91]. These efforts are orthogonal to ours as we accelerate the data chunking phase.

*7.0.3 Accelerating other storage systems.* Vector instructions have been widely used to accelerate other storage systems. MinervaFS [92] accelerates the computation of transform and basis functions in generalized deduplication with AVX instructions. ICID [93] records memory-copy operations in a B-Tree for fine-grained deduplication, accelerating tree searches with AVX instructions. AVX-512 conflict detection instructions have been used to accelerate lightweight data compression algorithms [94]. Numerous works attempt to accelerate collision-resistant hashing algorithms used across storage systems with vector instructions [95, 96]. These efforts are orthogonal to ours as we focus on using vector instructions to accelerate CDC algorithms for block-level deduplication.

*7.0.4 Secure deduplication systems.* Several efforts build end-to-end deduplication systems for encrypted data [97]. They mainly target encryption schemes [98, 99] for the underlying data or focus on reducing attacks on the system [100, 101]. Some target specific applications, such as distributing encrypted docker images [102] and encrypted videos [103]. As all of these efforts layer encryption atop existing data chunking algorithms, VectorCDC is compatible with all these approaches.

## 8 Conclusion

We present VectorCDC, a methodology for accelerating content-defined chunking using vector instructions. VectorCDC avoids the pitfalls of previous work that accelerates CDC algorithms by choosing hashless CDC algorithms instead. VectorCDC accelerates these algorithms using novel *tree-based search* and *packed scanning* methods. Our evaluation shows that VectorCDC achieves 8.35×-26.2× higher throughput than existing vector-accelerated CDC algorithms and 15.3×-207.2× higher throughput than unaccelerated algorithms. We have made our code publicly available by integrating it with DedupBench [15], and published one of our datasets on Kaggle [28].

## Acknowledgments

We thank the anonymous reviewers of USENIX FAST 2025 and ACM Transactions on Storage for their feedback. We thank Lori Paniak for his technical assistance throughout the project, and Mu'men Al-Jarah for his feedback on an earlier version of this work. The research team was supported by grants from the National Cybersecurity Consortium (NCC), Natural Sciences and Engineering Research Council of Canada (NSERC), and the Ontario Research Fund's Research Excellence Program (ALLRP-561423-20, RGPIN-2025-03332, and ORF-RE012-051). The team was also supported by research grants from Acronis, Oracle Research Labs, and Rogers Communications. Sreeharsha is supported by the Cheriton Graduate Scholarship and the Ontario Graduate Scholarship.

## References

[1] Statista. Worldwide data created from 2010 to 2025, 2024.
[2] Mark Carlson, Alan Yoder, Leah Schoeb, Don Deel, Carlos Pratt, Chris Lionetti, and Doug Voigt. Software Defined Storage. *Storage Networking Industry Association Working Draft*, pages 20–24, 2014.
[3] Peter M Chen, Edward K Lee, Garth A Gibson, Randy H Katz, and David A Patterson. RAID: High-performance, reliable secondary storage. *ACM Computing Surveys (CSUR)*, 26(2):145–185, 1994.
[4] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop distributed file system. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10. Ieee, 2010.
[5] Sage Weil, Scott A Brandt, Ethan L Miller, Darrell DE Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th Conference on Operating Systems Design and Implementation (OSDI'06)*, pages 307–320, 2006.
[6] Brad Fitzpatrick. Distributed caching with memcached. *Linux Journal*, 2004(124):5, 2004.
[7] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, et al. TAO: Facebook's distributed data store for the social graph. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 49–60, 2013.
[8] Dutch T Meyer and William J Bolosky. A study of practical deduplication. *ACM Transactions on Storage (ToS)*, 7(4):1–20, 2012.
[9] Wen Xia, Hong Jiang, Dan Feng, Fred Douglis, Philip Shilane, Yu Hua, Min Fu, Yucheng Zhang, and Yukun Zhou. A comprehensive study of the past, present, and future of data deduplication. *Proceedings of the IEEE*, 104(9):1681–1710, 2016.
[10] Deyan Chen and Hong Zhao. Data security and privacy protection issues in cloud computing. In *2012 International Conference on Computer Science and Electronics Engineering*, volume 1, pages 647–651. IEEE, 2012.
[11] Ahmed El-Shimi, Ran Kalach, Ankit Kumar, Adi Ottean, Jin Li, and Sudipta Sengupta. Primary Data Deduplication — Large scale study and system design. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 285–296, 2012.
[12] Grant Wallace, Fred Douglis, Hangwei Qian, Philip Shilane, Stephen Smaldone, Mark Chamness, and Windsor Hsu. Characteristics of backup workloads in production systems. In *USENIX Conference on File and Storage Technologies (FAST)*, volume 12, pages 4–4, 2012.

[13] Phlip Shilane, Mark Huang, Grant Wallace, and Windsor Hsu. Wan-optimized replication of backup datasets using stream-informed delta compression. *ACM Transactions on Storage (ToS)*, 8(4):1–26, 2012.

[14] Sarah Henderson. Document duplication: How users (struggle to) manage file copies and versions. *Proceedings of the American Society for Information Science and Technology*, 48(1):1–10, 2011.

[15] Alan Liu, Abdelrahman Baba, Sreeharsha Udayashankar, and Samer Al-Kiswany. DedupBench: A Benchmarking Tool for Data Chunking Techniques. In *2023 IEEE Canadian Conference on Electrical and Computer Engineering (CCECE)*, pages 469–474. IEEE, 2023.

[16] Jingwei Li, Zuoru Yang, Yanjing Ren, Patrick PC Lee, and Xiaosong Zhang. Balancing storage efficiency and data confidentiality with tunable encrypted deduplication. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–15, 2020.

[17] Nigel Tan, Jakob Luettgau, Jack Marquez, Keita Teranishi, Nicolas Morales, Sanjukta Bhowmick, Franck Cappello, Michela Taufer, and Bogdan Nicolae. Scalable incremental checkpointing using gpu-accelerated de-duplication. In *Proceedings of the 52nd International Conference on Parallel Processing*, ICPP '23, page 665–674, New York, NY, USA, 2023. Association for Computing Machinery.

[18] Kiatchumpol Suttisirikul and Putchong Uthayopas. Accelerating the cloud backup using GPU based data deduplication. In *2012 IEEE 18th International Conference on Parallel and Distributed Systems*, pages 766–769. IEEE, 2012.

[19] Samer Al-Kiswany, Abdullah Gharaibeh, Elizeu Santos-Neto, George Yuan, and Matei Ripeanu. StoreGPU: Exploiting Graphics Processing Units to Accelerate Distributed Storage Systems. In *Proceedings of the 17th International Symposium on High Performance Distributed Computing*, HPDC '08, page 165–174, New York, NY, USA, 2008. Association for Computing Machinery.

[20] Yucheng Zhang, Hong Jiang, Dan Feng, Wen Xia, Min Fu, Fangting Huang, and Yukun Zhou. AE: An asymmetric extremum content defined chunking algorithm for fast and bandwidth-efficient data deduplication. In *2015 IEEE Conference on Computer Communications (INFOCOM)*, pages 1337–1345. IEEE, 2015.

[21] Wen Xia, Yukun Zhou, Hong Jiang, Dan Feng, Yu Hua, Yuchong Hu, Qing Liu, and Yucheng Zhang. FastCDC: A fast and efficient content-defined chunking approach for data deduplication. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 101–114, 2016.

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

[23] Athicha Muthitacharoen, Benjie Chen, and David Mazieres. A low-bandwidth network file system. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles (SOSP)*, pages 174–187, 2001.

[24] Ryan NS Widodo, Hyotaek Lim, and Mohammed Atiquzzaman. A new content-defined chunking algorithm for data deduplication in cloud storage. *Future Generation Computer Systems*, 71:145–156, 2017.

[25] Kave Eshghi and Hsiu Khuern Tang. A framework for analyzing and improving content-based chunking algorithms. *Hewlett-Packard Labs Technical Report TR*, 30(2005), 2005.

[26] Fan Ni, Xing Lin, and Song 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*, pages 86–96, 2019.

[27] Nikolaj Bjørner, Andreas Blass, and Yuri Gurevich. Content-dependent chunking for differential compression, the local maximum approach. *Journal of Computer and System Sciences*, 76(3-4):154–203, 2010.

[28] Sreeharsha Udayashankar, Abdelrahman Baba, and Samer Al-Kiswany. VM Images for Deduplication. https://www.kaggle.com/dsv/10561721, 2025.

[29] Dian Rachmawati, JT Tarigan, and ABC Ginting. A comparative study of Message Digest 5 (MD5) and SHA256 algorithm. In *Journal of Physics: Conference Series*, volume 978, page 012116. IOP Publishing, 2018.

[30] Chunlin Song, Xianzhang Chen, Duo Liu, Jiali Li, Yujuan Tan, and Ao Ren. Optimizing the Performance of Consistency-Aware Deduplication Using Persistent Memory. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2023.

[31] Sean Quinlan and Sean Dorward. Venti: A new approach to archival data storage. In *USENIX Conference on File and Storage Technologies*, 2002.

[32] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, et al. Oceanstore: An architecture for global-scale persistent storage. *ACM SIGOPS Operating Systems Review*, 34(5):190–201, 2000.

[33] Mike Dutch. Understanding data deduplication ratios. In *SNIA Data Management Forum*, volume 7, 2008.

[34] James E Smith, Greg Faanes, and Rabin Sugumar. Vector instruction set support for conditional operations. *ACM SIGARCH Computer Architecture News*, 28(2):260–269, 2000.

[35] Intel. Intel® Intrinsics Guide. https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html, 2024.

[36] Somaia A Hassan, Mountasser MM Mahmoud, AM Hemeida, and Mahmoud A Saber. Effective implementation of matrix–vector multiplication on Intel's AVX multicore processor. *Computer Languages, Systems & Structures*, 51:158–175, 2018.

[37] Shay Gueron and Vlad Krasnov. Fast quicksort implementation using AVX instructions. *The Computer Journal*, 59(1):83–90, 2016.

[38] Robert L Bocchino Jr and Vikram S Adve. Vector LLVA: a virtual vector instruction set for media processing. In *Proceedings of the 2nd International Conference on Virtual Execution Environments*, pages 46–56, 2006.

[39] Jorge Francés, Sergio Bleda, Andrés Márquez, Cristian Neipp, Sergi Gallego, Beatriz Otero, and Augusto Beléndez. Performance analysis of SSE and AVX instructions in multi-core CPUs and GPU computing on FDTD scheme for solid and fluid vibration problems. *The Journal of Supercomputing*, 70:514–526, 2014.

[40] Maximilian Böther, Lawrence Benson, Ana Klimovic, and Tilmann Rabl. Analyzing Vectorized Hash Tables across CPU Architectures. *Proceedings of the VLDB Endowment*, 16(11):2755–2768, July 2023.

[41] Markus Dreseler, Jan Kossmann, Johannes Frohnhofen, Matthias Uflacker, and Hasso Plattner. Fused Table Scans: Combining AVX-512 and JIT to Double the Performance of Multi-Predicate Scans. In *2018 IEEE 34th International Conference on Data Engineering Workshops (ICDEW)*, pages 102–109, 2018.

[42] Intel. Intel® Instruction Set Extensions Technology. https://www.intel.com/content/www/us/en/support/articles/000005779/processors.html.

[43] Advanced Micro Devices. Revision Guide for AMD Athlon 64 and AMD Opteron$^{TM}$ Processors. https://www.amd.com/content/dam/amd/en/documents/archived-tech-docs/revision-guides/25759.pdf, 2003.

[44] WikiChip. Skylake Server - Microarchitectures - Intel. https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(server), 2017.

[45] WikiChip. Zen 4 - Microarchitectures - AMD. https://en.wikichip.org/wiki/amd/microarchitectures/zen_4, 2022.

[46] ARM. ARM NEON Architecture Overview. https://developer.arm.com/documentation/dht0002/a/Introducing-NEON/NEON-architecture-overview/NEON-instructions, 2013.

[47] Ruimin Shi, Gabin Schieffer, Maya Gokhale, Pei-Hung Lin, Hiren Patel, and Ivy Peng. ARM SVE Unleashed: Performance and Insights Across HPC Applications on Nvidia Grace. *European Conference on Parallel Processing*, 2025.

[48] B. Sinharoy, J. A. Van Norstrand, R. J. Eickemeyer, H. Q. Le, J. Leenstra, D. Q. Nguyen, B. Konigsburg, K. Ward, M. D. Brown, J. E. Moreira, D. Levitan, S. Tung, D. Hrusecky, J. W. Bishop, M. Gschwind, M. Boersma, M. Kroener, M. Kaltenbach, T. Karkhanis, and K. M. Fernsler. IBM POWER8 processor core microarchitecture. *IBM Journal of Research and Development*, 59(1):2:1–2:21, 2015.

[49] Frederic P. Miller, Agnes F. Vandome, and John McBrewster. *AltiVec.* Alpha Press, 2010.

[50] Ronald Rivest. RFC 1321: The MD5 message-digest algorithm. Technical report, Network Working Group, 1992.

[51] D Eastlake 3rd. RFC 3174: US secure hash algorithm 1 (SHA1). Technical report, Network Working Group, 2001.

[52] D. Eastlake 3rd and T. Hansen. RFC 4634: US Secure Hash Algorithms (SHA and HMAC-SHA). Technical report, Network Working Group, 2006.

[53] Austin Appleby. MurmurHash3. 2011.

[54] Lianhua Chi and Xingquan Zhu. Hashing techniques: A survey and taxonomy. *ACM Computing Surveys (Csur)*, 50(1):1–36, 2017.

[55] xxHash. xxHash - Extremely fast non-cryptographic hash algorithm. https://xxhash.com/, 2020.

[56] Austin Appleby. SMHasher. 29:2016, 2016.

[57] Fan Ni, Xingbo Wu, Weijun Li, Lei Wang, and Song Jiang. Woj: Enabling write-once full-data journaling in ssds by using weak-hashing-based deduplication. *Performance Evaluation*, 127-128:56–69, 2018.

[58] Patrick Lavin, Jeffrey Young, Richard Vuduc, Jason Riedy, Aaron Vose, and Daniel Ernst. Evaluating Gather and Scatter Performance on CPUs and GPUs. In *Proceedings of the International Symposium on Memory Systems*, MEMSYS '20, page 209–222, New York, NY, USA, 2021. Association for Computing Machinery.

[59] Danila Kutenin. Porting x86 vector bitmask optimizations to Arm NEON. https://community.arm.com/arm-community-blogs/b/servers-and-cloud-computing-blog/posts/porting-x86-vector-bitmask-optimizations-to-arm-neon, 2022.

[60] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of CloudLab. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 1–14, Renton, WA, July 2019. USENIX Association.

[61] Debian. Debian – The Universal Operating System. https://www.debian.org/, 2025.

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

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

[64] Bo Pang, Erik Nijkamp, and Ying Nian Wu. Deep learning with tensorflow: A review. *Journal of Educational and Behavioral Statistics*, 45(2):227–248, 2020.

[65] Marko Luksa. *Kubernetes in action.* Simon and Schuster, 2017.

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

[67] GNU. GNU tar 1.35: Basic Tar Format. https://www.gnu.org/software/tar/manual/html_section/Standard.html, 2023.

[68] Mordechai Haklay and Patrick Weber. OpenStreetMap: User-Generated Street Maps. *IEEE Pervasive Computing*, 2008.

[69] GeoFabrik. GEOFABRIK. https://www.geofabrik.de/, 2025.

[70] Redis. Redis. https://redis.io/, 2023.

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

[72] Transaction Processing Council. TPC-C Overview. https://www.tpc.org/tpcc/detail5.asp, 2023.

[73] Wikipedia. List of films based on actual events. https://en.wikipedia.org/wiki/List_of_films_based_on_actual_events, 2022.

[74] Owen Randall and Paul Lu. Predicting deduplication performance: An analytical model and empirical evaluation. In *2022 IEEE International Conference on Big Data (Big Data)*, pages 319–328, 2022.

[75] OpenStreetMap. OSM file formats - OpenStreetMap Wiki. https://wiki.openstreetmap.org/wiki/OSM_file_formats, 2025.

[76] Jan-Erik Rediger. RDB File Format. https://rdb.fnordig.de/file_format.html, 2015.

[77] DMTF. Open virtualization format white paper. https://www.dmtf.org/sites/default/files/standards/documents/DSP2017_1.0.0.pdf, 2009.

[78] Irving W Burr. Cumulative Frequency Functions. *The Annals of Mathematical Statistics*, 13(2):215–232, 1942.

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

[80] Wen Xia, Hong Jiang, Dan Feng, Lei Tian, Min Fu, and Zhongtao Wang. P-Dedupe: Exploiting Parallelism in Data Deduplication System. In *2012 IEEE Seventh International Conference on Networking, Architecture, and Storage*, pages 338–347, 2012.

[81] Fan Ni and Song Jiang. RapidCDC: Leveraging Duplicate Locality to Accelerate Chunking in CDC-Based Deduplication Systems. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '19, page 220–232, New York, NY, USA, 2019. Association for Computing Machinery.

[82] Erik Kruus, Cristian Ungureanu, and Cezary Dubnicki. Bimodal content defined chunking for backup streams. In *Fast*, pages 239–252, 2010.

[83] Mu'men Al Jarah, Sreeharsha Udayashankar, Abdelrahman Baba, and Samer Al-Kiswany. The Impact of Low-Entropy on Chunking Techniques for Data Deduplication. In *2024 IEEE 17th International Conference on Cloud Computing (CLOUD)*, pages 134–140, 2024.

[84] Sreeharsha Udayashankar, Abdelrahman Baba, and Samer Al-Kiswany. {VectorCDC}: Accelerating data deduplication with vector instructions. In *23rd USENIX Conference on File and Storage Technologies (FAST 25)*, pages 513–522, 2025.

[85] Wen Xia, Hong Jiang, Dan Feng, and Yu Hua. Similarity and Locality Based Indexing for High Performance Data Deduplication. *IEEE Transactions on Computers*, 64(4):1162–1176, 2015.

[86] Mark Lillibridge, Kave Eshghi, Deepavali Bhagwat, Vinay Deolalikar, Greg Trezis, and Peter Camble. Sparse indexing: Large scale, inline deduplication using sampling and locality. In *USENIX Conference on File and Storage Technologies (FAST)*, volume 9, pages 111–123, 2009.

[87] Deepavali Bhagwat, Kave Eshghi, Darrell DE Long, and Mark Lillibridge. Extreme binning: Scalable, parallel deduplication for chunk-based file backup. In *2009 IEEE International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems*, pages 1–9. IEEE, 2009.

[88] Cezary Dubnicki, Leszek Gryz, Lukasz Heldt, Michal Kaczmarczyk, Wojciech Kilian, Przemyslaw Strzelczak, Jerzy Szczepkowski, Cristian Ungureanu, and Michal Welnicki. HYDRAstor: A scalable secondary storage. In *USENIX Conference on File and Storage Technologies (FAST)*, volume 9, pages 197–210, 2009.

[89] Phlip Shilane, Mark Huang, Grant Wallace, and Windsor Hsu. WAN-optimized replication of backup datasets using stream-informed delta compression. *ACM Transactions on Storage (ToS)*, 8(4):1–26, 2012.

[90] Xiangyu Zou, Wen Xia, Philip Shilane, Haijun Zhang, and Xuan Wang. Building a high-performance fine-grained deduplication framework for backup storage with high deduplication ratio. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 19–36, 2022.

[91] Yucheng Zhang, Hong Jiang, Dan Feng, Nan Jiang, Taorong Qiu, and Wei Huang. {LoopDelta}: Embedding locality-aware opportunistic delta compression in inline deduplication for highly efficient data reduction. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 133–148, 2023.

[92] Lars Nielsen, Dorian Burihabwa, Valerio Schiavoni, Pascal Felber, and Daniel E. Lucani. MinervaFS: A User-Space File System for Generalised Deduplication: (Practical experience report). In *2021 40th International Symposium on Reliable Distributed Systems (SRDS)*, pages 254–264, 2021.

[93] Haikun Liu, Xiaozhong Jin, Chencheng Ye, Xiaofei Liao, Hai Jin, and Yu Zhang. I/O Causality Based In-Line Data Deduplication for Non-Volatile Memory Enabled Storage Systems. *IEEE Transactions on Computers*, 73(5):1327–1340, 2024.

[94] Annett Ungethum, Johannes Pietrzyk, Patrick Damme, Dirk Habich, and Wolfgang Lehner. Conflict Detection-Based Run-Length Encoding - AVX-512 CD Instruction Set in Action. In *2018 IEEE 34th International Conference on Data Engineering Workshops (ICDEW)*, pages 96–101, 2018.

[95] Daniel Lemire and Owen Kaser. Faster 64-bit universal hashing using carry-less multiplications. *Journal of Cryptographic Engineering*, 6:171–185, 2016.

[96] Tony C Pan, Sanchit Misra, and Srinivas Aluru. Optimizing high performance distributed memory parallel hash tables for DNA k-mer counting. In *2018 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 135–147. IEEE, 2018.

[97] Youngjoo Shin, Dongyoung Koo, and Junbeom Hur. A Survey of Secure Data Deduplication Schemes for Cloud Storage Systems. *ACM Computing Surveys*, 49(4), Jan 2017.

[98] Mihir Bellare, Sriram Keelveedhi, and Thomas Ristenpart. Message-locked encryption and secure deduplication. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 296–312. Springer, 2013.

[99] Jian Liu, N. Asokan, and Benny 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, page 874–885, New York, NY, USA, 2015. Association for Computing Machinery.

[100] Danny Harnik, Benny Pinkas, and Alexandra Shulman-Peleg. Side Channels in Cloud Services: Deduplication in Cloud Storage. *IEEE Security and Privacy*, 8(6):40–47, 2010.

[101] Yanjing Ren, Jingwei Li, Zuoru Yang, Patrick P. C. Lee, and Xiaosong Zhang. Accelerating Encrypted Deduplication via SGX. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 957–971. USENIX Association, July 2021.

[102] Tong Sun, Bowen Jiang, Borui Li, Jiamei Lv, Yi Gao, and Wei Dong. {SimEnc}: A {High-Performance}{Similarity-Preserving} encryption approach for deduplication of encrypted docker images. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, pages 615–630, 2024.

[103] Yifeng Zheng, Xingliang Yuan, Xinyu Wang, Jinghua Jiang, Cong Wang, and Xiaolin Gui. Toward Encrypted Cloud Media Center With Secure Deduplication. *IEEE Transactions on Multimedia*, 19(2):251–265, 2017.