# DedupBench: A Benchmarking Tool for Data Chunking Techniques

Alan Liu*, Abdelrahman Baba*, Sreeharsha Udayashankar*, Samer Al-Kiswany*#
*University of Waterloo, Canada
#Acronis Research, Canada
{i7liu, ababa, s2udayas, alkiswany}@uwaterloo.ca

***Abstract*—Data deduplication is a technique for reducing storage space by identifying and eliminating redundant data. The division of files into chunks is one of the key steps in the deduplication process and directly impacts deduplication effectiveness. Despite the numerous algorithms available for chunking, there is a limited understanding of their strengths and weaknesses in virtual machine backup environments.**

**We present DedupBench, a framework designed to assess the performance of different chunking algorithms for deduplication on user-specified data. DedupBench allows for the evaluation of chunking techniques by comparing their deduplication ratio and chunking throughput. DedupBench incorporates a generic design, allowing for the effortless integration of additional chunking techniques developed in the future.**

**We evaluate four widely used chunking algorithms using a VM-based dataset with DedupBench. Our evaluation contrasts earlier studies and demonstrates that Asymmetric Extremum (AE) has the best deduplication efficiency for VM-based datasets among the tested algorithms, highlighting the need to evaluate chunking techniques on user-specified data before designing deduplication systems.**

***Index Terms*—deduplication, chunking**

## I. Introduction

Data generation rates have grown explosively over the last decade [17]. Cloud storage providers have evolved numerous mechanisms to support this data growth, including large-scale distributed file systems [9, 15], novel storage architectures [11, 16] and data compression techniques [25].

Data deduplication is one such mechanism that has received industry-wide recognition. Previous studies have shown that a substantial amount of the data stored by providers such as Microsoft [14] and Dell EMC [27] is redundant. Data deduplication is a technique that identifies these redundant copies, eliminating the need to store or re-transmit them. Deduplication is especially effective in virtual machine (VM) backup workloads [27]. As VMs tend to change very little between consecutive backups, using deduplication techniques with these workloads can result in significant storage-cost savings of up to 80% [19].

Deduplication can be performed either at file or block-level granularities. While a file-level deduplication technique operates on complete files, block-level techniques typically divide them into multiple chunks using a *chunking algorithm* [19]. Block-level deduplication is preferred over file-level deduplication as it results in higher deduplication ratios and savings. After dividing files into chunks, the deduplication system hashes each chunk using a collision-resistant hashing algorithm such as SHA-256 [8]. The resulting hash, or *fingerprint*, is compared against a database of previously observed fingerprints, called a *fingerprint index* to detect whether the chunk is a duplicate. If a duplicate chunk is detected, it can be safely discarded without storing it.

Chunking algorithms can be of two kinds: fixed-size or content-defined [20]. Fixed-size chunking algorithms divide files into chunks of a pre-specified size while content-defined chunking algorithms examine the data within files to determine how they must be divided. The choice of chunking algorithm drastically affects the efficiency of a deduplication system. Although numerous chunking algorithms have been proposed in current literature [20, 28, 33, 34, 35], there is limited understanding of their comparative effectiveness on VM-based data.

In this paper, we present DedupBench, a framework designed to enable the comparison of chunking algorithms used in modern deduplication systems on user-specified data. DedupBench allows users to empirically evaluate these algorithms and select the best-performing technique for their requirements.

In our evaluation, we consider four chunking techniques that represent current categoires of chunking algorithms. We include fixed-size chunking, Asymmetric Extremum (AE) representing byte-value chunking, and Rabin and Gear representing hash-based chunking. We note DedupBench is extensible allowing the implementation of other chunking techniques.

The results show that the *deduplication efficiency* is highest and lowest when using Asymmetric Extremum (AE) [35] and Gear-based chunking [32] respectively for VM-based datasets. This contrasts earlier studies [32, 35] which showed that both algorithms performed similarly across a variety of datasets. We posit that the difference in our findings is primarily due to differing characteristics within VM-based datasets.

This difference, as well as other differing metrics such as *chunk size variance*, highlight the need to evaluate algorithms on user-specified data before making design decisions for deduplication systems.

## II. Background

### A. Data deduplication

Data deduplication is a widely used approach in backup systems that aims to reduce storage usage and network bandwidth

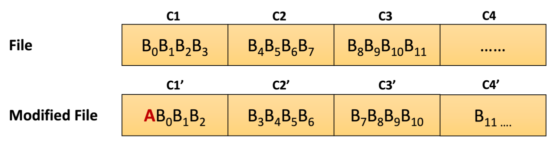


Fig. 1: Boundary-shift problem

by identifying and eliminating redundant data [19, 20]. Data deduplication consists of four fundamental steps [31]:

- `File Chunking`: Files to be deduplicated are divided into smaller units, called *chunks* by using a *chunking algorithm.*
- `Chunk Hashing`: Each chunk is hashed using a collision-resistant hashing function such as SHA-256 [8] to obtain a *fingerprint.*
- `Fingerprint Indexing`: Fingerprints are compared against a database of previously observed fingerprints, called a *fingerprint index* to check if the chunks are duplicates.
- `Chunk Storage`: Non-duplicate chunks are stored and their fingerprints are added to the fingerprint index.

### *B. File Chunking*

File Chunking is the first and most important step in a deduplication system [19]. In this step, files are divided into fixed or variable sized chunks based on the used chunking algorithm. Fixed-Size Chunking (FSC) and Content-Defined Chunking (CDC) are the main chunking methods used [18, 28] to achieve this.

*1) Fixed-Sized Chunking (FSC):* FSC divides files into predefined fixed size chunks and requires minimal computational calculations, as opposed to CDC. However, it fails to achieve high deduplication ratio due to the *boundary-shift* problem [20], where any change in the data (because of insertions or deletions) causes all subsequent chunks to change.

Figure 1 illustrates the boundary shift problem when using FSC. It shows after inserting a single byte (A) at the beginning of the data, all the subsequent chunks are altered. As a result, the deduplication system perceives these chunks as new, even though most of them differ only by a single byte from their previous versions.

*2) Content-Defined Chunking (CDC):* CDC algorithms are more resistant to boundary-shift problem [26], since chunks boundaries are defined using data characteristics rather than fixed size blocks, but more computational power is usually needed to calculate boundaries of data chunks. Additionally, research has shown that in CDC-based deduplication systems, CPU is the primary bottleneck rather than I/O, and that the file chunking stage alone is responsible for more than 60% of CPU time [34].

Most CDC algorithms use a sliding window approach to traverse the data stream. The chunk boundaries are then identified by calculating the rolling hash value of the bytes in the window. Rabin's rolling hash fingerprint [23] is one of the earliest and most commonly used hash function in deduplication systems [20]. Although the deduplication ratio obtained by

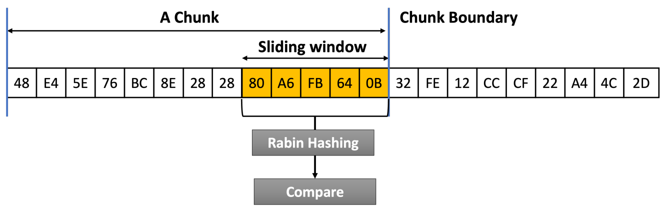


Fig. 2: Rabin-based chunking

Rabin is usually high, it requires high computational power to calculate [7]. Gear Hash (used in FastCDC [33]) is considered one of the fastest rolling hash algorithms at present. [33]. Other algorithms including MAXP [10], AE [35] and RAM [30] try to save computation power by avoiding hash calculations. Data attributes (such as byte values) are used to determine chunk boundaries instead.

**Rabin's Fingerprint.** Rabin's fingerprinting (Figure 2) determines boundaries by passing a fixed-size sliding window over the data stream. Rabin's hash [23] is calculated for all the bytes in the window. A chunk boundary is inserted at the end of the window when the lowest `K` bits of the hash value match a predefined value. Otherwise, the window slides by one byte until this condition is satisfied. When sliding the window, the new hash can be recomputed from the previous hash using the following equation where $\alpha$ is the window size and `P` is a random irreducible polynomial:

$$Rabin(B_2, B_3, ..., B_{\alpha+1}) = \{[Rabin(B_1, ..., B_\alpha) - B_1 P^{\alpha-1}]p + B_{\alpha+1}\} \; mod \; S \tag{1}$$

The expected average chunk size is proportional to `K` and is equal to $2^K$. Rabin's fingerprinting has a high computational overhead and struggles to find boundaries in low entropy strings, which can lead to high variance in chunk sizes as shown in our evaluation (§IV-D) . Maximum and minimum thresholds can be applied to chunk size to overcome this problem [20].

**Gear-based Chunking.** Gear-based Chunking algorithms use gear hash to calculate the fingerprint, which was designed to be computationally inexpensive compared to Rabin's hashing. Due to its lower computational overhead, gear hashing has been used by many chunking algorithms including FastCDC [33], SuperCDC [28] and RapidCDC [21].

Gear uses an array of 256 random large integers to map bytes to hash values. This reduces the number of operations required to recompute the hash upon sliding the window. Table I shows the operations required to slide the window by a single byte with both algorithms. The following equation is used to recompute the hash where G is the array of large integers:

$$Gear(B_2, B_3, ..., B_{\alpha+1}) = (Gear(B_1, ..., B_\alpha) << 1) + G[B_{\alpha+1}] \tag{2}$$

A drawback in Gear is its limited window size compared to Rabin-based hashing, The window size in Gear is limited to the number of bits used by the mask value. The mask plays an important role in determining the boundaries. It

| Algorithm | ADD | LOOKUP | OR | SHIFT | XOR |
|---|---|---|---|---|---|
| Rabin | 0 | 2 | 1 | 2 | 2 |
| Gear | 1 | 1 | 0 | 1 | 0 |

TABLE I: Sliding window computational cost comparison

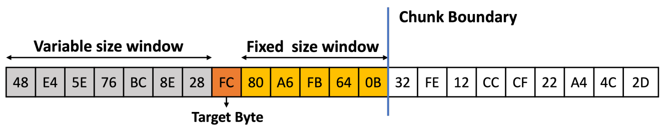

Chunk Boundary
Variable size window
Fixed size window
48 E4 5E 76 BC 8E 28 FC 80 A6 FB 64 0B 32 FE 12 CC CF 22 A4 4C 2D
Target Byte

Fig. 3: Asymmetric Extremum (AE) chunking

is a specific sequence of bits that is used to perform a bitwise AND operation with the fingerprint. When the result of this operation is equal to a predetermined constant, a chunk boundary is identified [33]. For instance, to achieve an average chunk size of 8KB the mask value would be $2^{13}$, resulting in a 13 byte window size for Gear-based chunking, much smaller compared to the 48 byte window in Rabin-based chunking. This smaller window size decreases the deduplication ratio, as it causes more collisions.

*3) Asymmetric Extremum:* Asymmetric Extremum (AE) [35] is one of a family of non rolling-hash based chunking algorithms such as MAXP [10] and RAM [30]. Figure 3 depicts the chunk boundary identification process within AE. AE searches for extremums in the data stream to identify chunk boundaries. It uses both variable and fixed size windows and the byte located between them is known as the *target byte*.

AE inserts chunk boundaries at the end of the fixed size window when the *target byte* is an extremum i.e. the value of this byte must be larger / smaller than the values of all bytes in the fixed sized window. Otherwise, the variable sized window expands by one byte until this condition is satisfied.

## III. Tool Design and Implementation

To effectively evaluate the techniques described in §II-B, we have designed and implemented DedupBench. DedupBench is a deduplication framework which focuses on empirically evaluating chunking algorithms using quantifiable performance metrics.

### A. Architecture

DedupBench is designed to facilitate measuring the performance of different chunking techniques on large amounts of data. DedupBench is designed into two separate tools: a chunking tool and an analysis tool. The chunking tool takes a set of files, chunks the data, and computes a chunk fingerprint. It then writes each chunk's size and hash to an intermediary file. To speed up operations, the chunking tool can be run in parallel on each storage node in a cluster. The analysis tool takes all the intermediary files produced by the chunking tool and analyzes them to determine chunk information and deduplication ratios.

After reading the configuration details, DedupBench performs the following steps for each chunking algorithm:

1) Read the contents of each file into memory buffers.
2) Chunk the contents of the buffers and calculate the chunk hash.

Once the chunking tool has been run on every node, the analysis tool calculates performance metrics including Deduplication Elimination Ratio (DER) and Chunk Size Variance and reports them.

### B. Extensibility

DedupBench is designed to be easily extensible. To this end, abstract interfaces for hashing and chunking (`Hashing_Technique` and `Chunking_Technique`) are provided to allow for the easy integration of chunking and hashing techniques developed in the future. Figure 4 shows the UML diagram for DedupBench.

*1) Chunking_Technique:* This abstract interface models a chunking algorithm and exposes the following functions for implementation by child classes:

- `chunk_stream()`: Chunk the data provided by a stream buffer.
- `chunk_file()`: Chunk the file located at the specified file path.

These functions should return a list of `File_Chunks`, a structure which models a chunk of data up for deduplication. This structure contains the data for the chunk with some additional metadata such as the `chunk_size` in bytes and the `chunk_hash` value.

*2) Hashing_Technique:* This abstract interface models a hashing algorithm and exposes the following function for implementation by child classes:

- `hash_chunk()`: Accepts a `File_Chunk` as input and modifies its `chunk_hash` to hold the value obtained by hashing the chunk.

This interface can be inherited to implement an SHA-1 class as shown in Figure 4. This interface has been used to

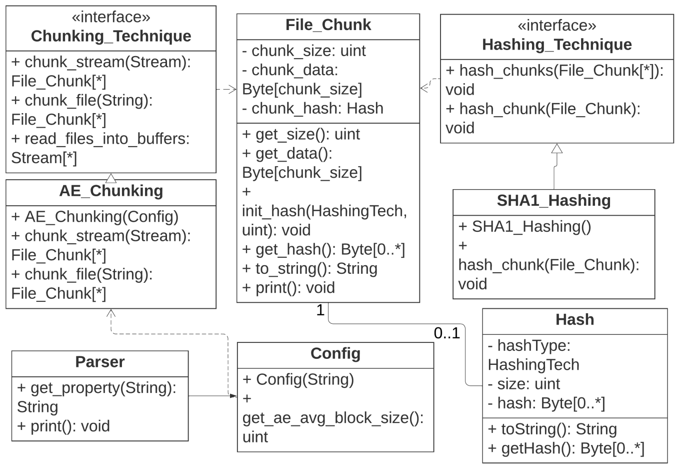

«interface»
Chunking_Technique
+ chunk_stream(Stream): File_Chunk[*]
+ chunk_file(String): File_Chunk[*]
+ read_files_into_buffers: Stream[*]
File_Chunk
- chunk_size: uint
- chunk_data: Byte[chunk_size]
- chunk_hash: Hash
+ get_size(): uint
+ get_data(): Byte[chunk_size]
+ init_hash(HashingTech, uint): void
+ get_hash(): Byte[0..*]
+ to_string(): String
+ print(): void
«interface»
Hashing_Technique
+ hash_chunks(File_Chunk[*]): void
+ hash_chunk(File_Chunk): void
AE_Chunking
+ AE_Chunking(Config)
+ chunk_stream(Stream): File_Chunk[*]
+ chunk_file(String): File_Chunk[*]
SHA1_Hashing
+ SHA1_Hashing()
+ hash_chunk(File_Chunk): void
1
0..1
Hash
- hashType: HashingTech
- size: uint
- hash: Byte[0..*]
+ toString(): String
+ getHash(): Byte[0..*]
Parser
+ get_property(String): String
+ print(): void
Config
+ Config(String)
+ get_ae_avg_block_size(): uint

Fig. 4: DedupBench Architectural Diagram

implement SHA-1 [13], SHA-256 [8] and MD5 hashing [24] within DedupBench.

*3) Configuration:* Each chunking technique can have its own set of parameters specified within a configuration file. The `Config` class is responsible for creating key-value mappings from this file for retrieval and use by the chunking techniques.

## C. Implementation

We have implemented DedupBench in C++ 17 in $\sim$ 3200 lines of code. DedupBench currently supports three hashing algorithms: SHA-1 [13], SHA-256 [8] and MD5 [24] using the OpenSSL [22] library. DedupBench supports the four chunking algorithms outlined in §II-B. DedupBench reports the performance metrics discussed in detail in §IV.

The code for DedupBench has been made publicly available on GitHub [3] under the `UWASL/dedup-bench` repository [29].

## IV. Evaluation

In this section, we evaluate the four chunking techniques discussed using a virtual-machine (VM) based dataset.

**Testbed.** Our evaluation has been carried out on a platform within CloudLab [12]. The machine had an Intel Xeon CPU with 8 hyperthreaded cores, 12M Last-Level-Cache and 64GB of main memory. We measure the performance of each chunking algorithm independent of the underlying file system. To facilitate this, DedupBench reads the data into in-memory buffers before the chunking process is started.

**Dataset.** We use a collection of fifteen virtual machine images built for various applications from the VMware Marketplace [6] to create a VM-based dataset with a total size of 10.66 GB. The images are packaged by Bitnami [2] and are Debian-based. Each image is pre-packaged with a target application such as Jenkins [4], Apache Kafka [1], and MySQL [5]. The complete list of VM images can be found on the project's GitHub repository [29].

**Chunking algorithms.** We compare the following chunking algorithms described in §II-B:

- *Fixed*: Fixed-size chunking algorithm
- *Rabin*: Content-defined chunking algorithm using Rabin's hashing [20] to identify chunk boundaries.
- *Gear*: Content-defined chunking algorithm using Gear hashing [32] to identify chunk boundaries.
- *AE-Max*: Asymmetric Extremum [35] operating in maximum mode i.e. the *target byte* is a local maximum.
- *AE-Min*: Asymmetric Extremum [35] operating in minimum mode i.e. the *target byte* is a local maximum.

**Metrics.** We use the following metrics for our evaluation.

- *Chunking Throughput*: The average rate at which data chunking occurs for the specified chunking algorithm.
- *Deduplication Elimination Ratio (DER)*: This metric represents the data savings obtained by a system using the specified chunking algorithm. A value of 1 indicates no deduplication at all while higher values indicate better deduplication efficiency. It is calculated as :

$$\frac{Original\ data\ size}{Deduplicated\ data\ size} \tag{3}$$

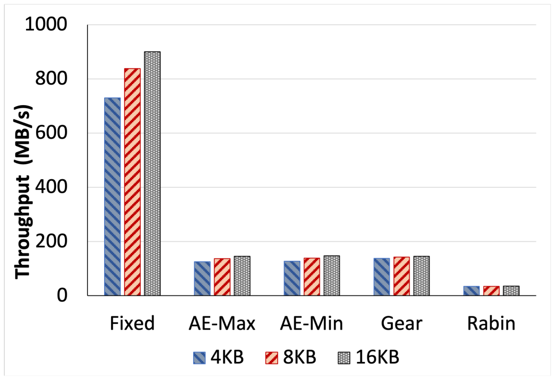


Fig. 5: Chunking Throughput Comparison

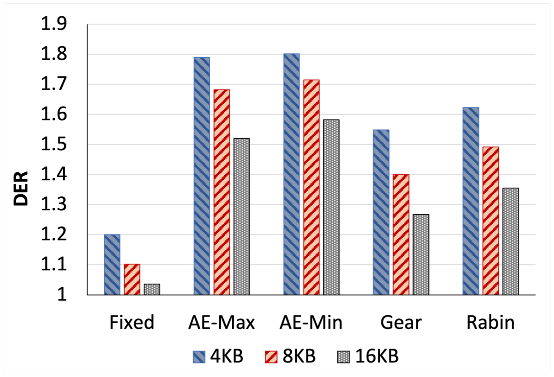


Fig. 6: Deduplication Elimination Ratio (DER) Comparison

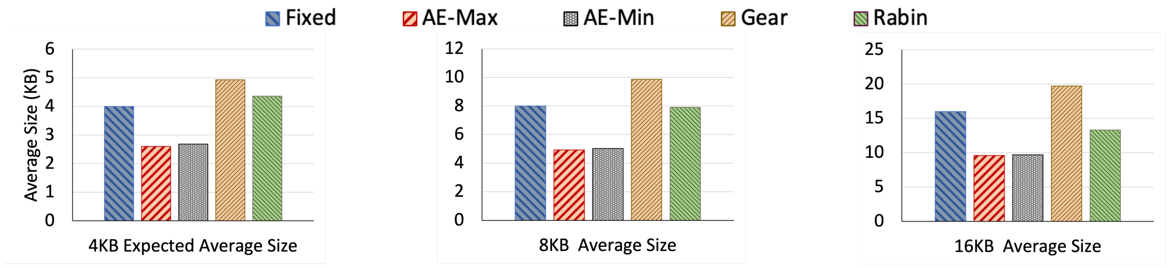
- *Average Chunk Size*: The average size of all chunks obtained by running the specified chunking algorithm. With content-defined chunking algorithms, the average observed chunk size can differ from the expected chunk size.
- *Chunk Size Variance*: The variance in chunk sizes obtained by running the specified chunking algorithm.

## A. Chunking Throughput

Figure 5 compares the average chunking throughput among all algorithms. Fixed-size chunking attains a significantly higher throughput compared to all other algorithms as it does not involve scanning the contents of the data. The throughput of fixed-size chunking increases with larger chunk sizes, as the total number of chunks is reduced.

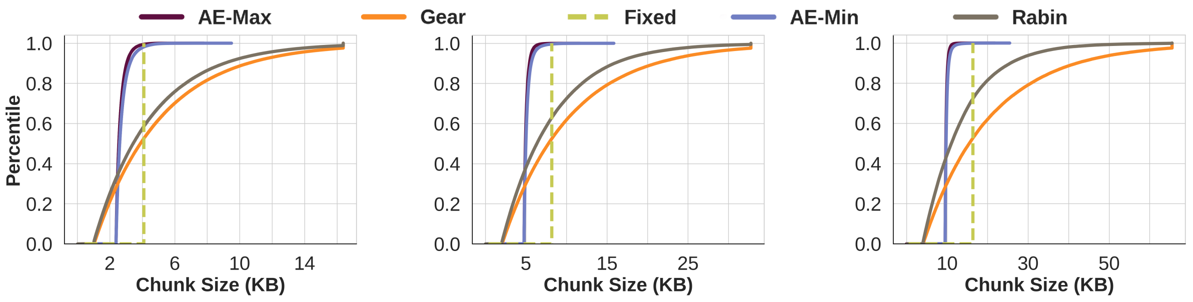
By contrast, Rabin-based chunking exhibits the lowest throughput at around 30MB/s regardless of chunk size. This is because Rabin hashing using a sliding window is the most computationally expensive of all the evaluated chunking algorithms. The throughput of Gear-based chunking spans a range of 125-150 MB/s. Gear hashing's lower computational cost is one of the major reasons for this improvement over Rabin-based hashing.

Both AE-Max and AE-Min and possess chunking throughputs similar to that of Gear-based hashing. Their throughput slightly increases with chunk size as well.



(a) 4 KB (b) 8 KB (c) 16 KB

Fig. 7: Observed Average Chunk Size for 4K, 8K and 16K expected average chunk sizes



(a) 4 KB (b) 8 KB (c) 16 KB

Fig. 8: Chunk Size Variance for 4K, 8K and 16K expected average chunk sizes

## *B. Deduplication Efficiency*

We use the Deduplication Elimination Ratio (DER) to evaluate the deduplication efficiency of each chunking algorithm. Figure 6 illustrates the DER for each algorithm with varying expected chunk sizes. DER decreases with increasing expected chunk size for all chunking algorithms. This is because the common chunks across the dataset reduce with increasing chunk size.

AE-Min achieves the highest DER of all the chunking algorithms, slightly higher than AE-Max. Rabin-based hashing performs better than gear-based hashing in terms of DER. While previous studies [35] have shown that AE and Rabin-based hashing have similar deduplication ratios, our results show that AE is significantly better than Rabin-based hashing for virtual-machine based data.

Gear-based hashing has the lowest deduplication ratio among all algorithms for our virtual-machine dataset. This contrasts earlier studies [33] which showed that Gear-based hashing has similar DERs as Rabin-based hashing.

Our evaluation shows that the deduplication efficiency is highly dependent on the underlying chunking algorithm. Chunking algorithms can also significantly differ in performance with changing data characteristics. Thus, DedupBench highlights the value of evaluating chunking algorithms on user-specified data to allow for informed decisions when designing deduplication systems.

## *C. Average Chunk Size*

Figure 7 shows the actual average chunk sizes for expected average chunk sizes of 4KB, 8KB and 16KB. Fixed-size chunking always results in the actual average chunk size matching the expected average, as all chunks are equal to the pre-specified size.

However, the actual average chunk sizes for content-defined chunking algorithms differ from their expected values. For instance, when using Gear-based chunking with an expected average chunk size of 16KB (Figure 7c), the actual average chunk size is 25% higher, at 20 KB instead. A key point to note from Figure 7 is that AE-Min and AE-Max always result in a lower actual average size than expected. On the other hand, rolling hash based chunking algorithms such as Rabin-based and Gear-based chunking always result in a higher actual average size than expected.

## *D. Chunk Size Variance*

Figure 8 shows the CDF of observed chunk sizes for expected average chunk sizes of 4KB, 8KB and 16KB. Fixed-size chunking has the least variance as all the chunk sizes are equal to the expected chunk size, with the exception of the last chunk whose size depends on the file size.

Among the content-defined chunking algorithms, AE-Min and AE-Max have the lowest variance. Rabin-based and Gear-based chunking however exhibit large variances in chunk size. For instance, in Figure 8a the maximum and minimum chunk sizes for Rabin-based chunking are 16KB and 1KB respectively, while the expected average chunk size is 4KB. Gear-based chunking exhibits similar variance as well.

This variance in the chunk size partially explains why Rabin-based and Gear-based chunking have lower DERs than AE-Min and AE-Max, as large chunk variance tends to be detrimental to deduplication performance.

## V. Conclusion

We present DedupBench, a framework to enable the smooth evaluation of popular chunking algorithms on user-specified data. DedupBench is designed to be generic and easily extensible to chunking algorithms developed in the future. Our evaluation shows that key metrics such as deduplication efficiency and chunk size variance are highly dependent on characteristics of the data up for deduplication. This highlights the need to evaluate these chunking algorithms on user-specified data before making critical design decisions for large-scale deduplication systems.

## References

[1] Apache kafka. https://kafka.apache.org/, Accessed on 2023-04-30.

[2] Bitnami. https://bitnami.com/, Accessed on 2023-04-30.

[3] Github. https://github.com/.

[4] Jenkins. https://www.jenkins.io/, Accessed on 2023-04-30.

[5] Mysql. hhttps://www.mysql.com/, Accessed on 2023-04-30.

[6] Vmware marketplace. https://marketplace.cloud.vmware.com/services, Accessed on 2023-04-30.

[7] S. Al-Kiswany, A. Gharaibeh, E. Santos-Neto, G. Yuan, and M. Ripeanu. Storegpu: exploiting graphics processing units to accelerate distributed storage systems. In *Proceedings of the 17th international symposium on High performance distributed computing*, pages 165–174, 2008.

[8] A. W. Appel. Verification of a cryptographic primitive: Sha-256. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 37(2):1–31, 2015.

[9] D. Beaver, S. Kumar, H. C. Li, J. Sobel, P. Vajgel, et al. Finding a needle in haystack: Facebook's photo storage. In *OSDI*, volume 10, pages 1–8, 2010.

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

[11] 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)*, 26(2):145–185, 1994.

[12] 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)*, pages 1–14, Renton, WA, July 2019. USENIX Association.

[13] D. Eastlake 3rd and P. Jones. Us secure hash algorithm 1 (sha1). Technical report, 2001.

[14] A. El-Shimi, R. Kalach, A. Kumar, A. Ottean, J. Li, and S. Sengupta. Primary data deduplication-large scale study and system design. In *USENIX Annual Technical Conference*, volume 2012, pages 285–296, 2012.

[15] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 29–43, 2003.

[16] G. A. Gibson and R. Van Meter. Network attached storage architecture. *Communications of the ACM*, 43(11):37–45, 2000.

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

[18] G. Lu, Y. Jin, and D. H. Du. Frequency based chunking for data de-duplication. In *2010 IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 287–296. IEEE, 2010.

[19] D. T. Meyer and W. J. Bolosky. A study of practical deduplication. *ACM Transactions on Storage (ToS)*, 7(4):1–20, 2012.

[20] A. Muthitacharoen, B. Chen, and D. Mazieres. A low-bandwidth network file system. In *Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 174–187, 2001.

[21] F. Ni and S. Jiang. Rapidcdc: Leveraging duplicate locality to accelerate chunking in cdc-based deduplication systems. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 220–232, 2019.

[22] I. OpenSSL Foundation. Openssl. https://www.openssl.org/.

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

[24] R. Rivest. The md5 message-digest algorithm. Technical report, 1992.

[25] D. Salomon. *Data compression: the complete reference*. Springer Science & Business Media, 2004.

[26] W. Tian, R. Li, Z. Xu, and W. Xiao. Does the content defined chunking really solve the local boundary shift problem? In *2017 IEEE 36th International Performance Computing and Communications Conference (IPCCC)*, pages 1–8. IEEE, 2017.

[27] G. Wallace, F. Douglis, H. Qian, P. Shilane, S. Smaldone, M. Chamness, and W. Hsu. Characteristics of backup workloads in production systems. In *FAST*, volume 12, pages 4–4, 2012.

[28] B. Wan, L. Pu, X. Zou, S. Li, P. Wang, and W. Xia. Supercdc: A hybrid design of high-performance content-defined chunking for fast deduplication. In *2022 IEEE 40th International Conference on Computer Design (ICCD)*, pages 170–178. IEEE, 2022.

[29] WASL. Uwasl/dedup-bench: Code for the dedupbench project. https://github.com/UWASL/dedup-bench.

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

[31] 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*, 104(9):1681–1710, 2016.

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

[33] 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)*, pages 101–114, 2016.

[34] 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)*, pages 288–299. IEEE, 2021.

[35] 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)*, pages 1337–1345. IEEE, 2015.