The Design of Fast Delta Encoding for Delta Compression Based Storage Systems

Journal:	*Transactions on Storage*
Manuscript ID	TOS-2023-09-0045
Manuscript Type:	Long Paper
Date Submitted by the Author:	13-Sep-2023
Complete List of Authors:	Tan, Haoliang; Harbin Institute of Technology Shenzhen, ; Peng Cheng Laboratory, Xia, Wen; Harbin Institute of Technology Shenzhen, School of Computer Science; Peng Cheng Laboratory Zou, Xiangyu; Harbin Institute of Technology Shenzhen, School of Computer Science Deng, Cai; Harbin Institute of Technology Shenzhen Liao, Qing; Harbin Institute of Technology Shenzhen; Peng Cheng Laboratory Gu, Zhaoquan; Harbin Institute of Technology Shenzhen; Peng Cheng Laboratory
Keywords:	Data reduction, Compression, Delta encoding, data deduplication

The Design of Fast Delta Encoding for Delta Compression Based Storage Systems

HAOLIANG TAN, Harbin Institute of Technology, Shenzhen; Peng Cheng Laboratory, China
WEN XIA*, Harbin Institute of Technology, Shenzhen; Peng Cheng Laboratory, China
XIANGYU ZOU, Harbin Institute of Technology, Shenzhen, China
CAI DENG, Harbin Institute of Technology, Shenzhen, China
QING LIAO, Harbin Institute of Technology, Shenzhen; Peng Cheng Laboratory, China
ZHAOQUAN GU, Harbin Institute of Technology, Shenzhen; Peng Cheng Laboratory, China

Delta encoding is a data reduction technique capable of calculating the differences (i.e., delta) among very similar files and chunks and is thus widely used for optimizing synchronization replication, backup/archival storage, cache compression, etc. However, delta encoding is computationally costly because of its time-consuming word-matching operations for delta calculation. Existing delta encoding approaches either run at a slow encoding speed, such as Xdelta and Zdelta, or at a low compression ratio, such as Ddelta and Edelta. In this paper, we propose Gdelta, a fast delta encoding approach with a high compression ratio. The key idea behind Gdelta is the combined use of five techniques: (1) employing an improved Gear-based rolling hash to replace Adler32 hash for scanning overlapping words of similar chunks, (2) rolling two bytes each time to speed up the Gear-based rolling hash while scanning words of base chunks, (3) adopting a quick array-based indexing scheme for word-matching, (4) skipping unmatched words to accelerate delta encoding through non-redundant areas, and (5) last but not least, after word-matching, further batch compressing the remainder to improve the compression ratio. Our evaluation results driven by seven real-world datasets suggest that Gdelta achieves encoding/decoding speedups of 4X~16X over the classic Xdelta and Zdelta approaches while increasing the compression ratio by about 10%~260%.

CCS Concepts: • **Information systems** → **Data compression**; • **Theory of computation** → Data compression.

Additional Key Words and Phrases: Data reduction, compression, delta encoding, data deduplication

ACM Reference Format:
HAOLIANG TAN, Wen Xia, Xiangyu Zou, Cai Deng, Qing Liao, and Zhaoquan Gu. 2023. The Design of Fast Delta Encoding for Delta Compression Based Storage Systems. *ACM Trans. Storage* 1, 1 (September 2023), 29 pages. https://doi.org/XXXXXXX.XXXXXXX

*Correspoding Author

Authors' addresses: HAOLIANG TAN, haoliangtann@gmail.com, Harbin Institute of Technology, Shenzhen; Peng Cheng Laboratory, Shen Zhen, China; Wen Xia, Harbin Institute of Technology, Shenzhen; Peng Cheng Laboratory, Shen Zhen, China, xiawen@hit.edu.cn; Xiangyu Zou, Harbin Institute of Technology, Shenzhen, Shen Zhen, China, xiangyu.zou@hotmail.com; Cai Deng, Harbin Institute of Technology, Shenzhen, Shen Zhen, China, dengcai.319@gmail.com; Qing Liao, Harbin Institute of Technology, Shenzhen; Peng Cheng Laboratory, Shen Zhen, China, liaoqing@hit.edu.cn; Zhaoquan Gu, Harbin Institute of Technology, Shenzhen; Peng Cheng Laboratory, Shen Zhen, China, guzhaoquan@hit.edu.cn.

1553-3077/2023/9-ART $15.00
https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Efficient storage of a huge volume of digital data is an ongoing challenge for industry and academia, as evidenced in part by an estimated 33 zettabytes of data in 2018 and 175 zettabytes of data produced in the world in 2025 according to IDC [18], which suggests the digital data will continue to grow exponentially. As a result, space-efficient data reduction techniques for storage systems (especially for cluster storage) have been gaining increasing attention recently.

In general, there are three primary technology roadmaps for lossless data reduction, including general compression [4, 10], delta compression [23, 31, 33, 41], and data deduplication [13, 15, 17, 36, 42, 43]. General compression has been evolving for decades and is a popular technique for most users. General compression is designed for reducing redundant data at the string/byte level by using traditional algorithms, e.g., dictionary coding [38, 39] and Huffman coding [12]. Because of the fine-grained redundancy searching, general compression usually achieves a good compression ratio. But it also has several weaknesses: For large-scale storage systems, on the one hand, the compression/decompression speed of general compression is not fast enough for processing such amounts of data. On the other hand, due to the size limitation of the compression window, general compression can only find out data redundancy within the range of the compression window.

Because of the limitations of general compression in large-scale storage systems, data deduplication is a more popular option. Data deduplication, eliminating redundancy at the chunk level, divides the data stream into several independent chunks (e.g., size of 8KB) and then eliminates duplicate chunks. Data deduplication accelerates the processing (i.e., compressing) speed up to GB/s level [42] and makes it practical for large systems. Because the workflow of deduplication is coarse-grained, there is further room to improve the compression ratio.

To bridge the gap in compression ratio between general compression and data deduplication, delta compression is proposed, which focuses on removing redundancy among similar files or chunks. In data deduplication, the data streams are divided into coarse-grained chunks, and many chunks are very similar and compressible for delta compression. The resemblance detection approaches [3, 16, 22, 35, 40] (e.g., N-Transform [3]) are usually employed to find similar items in the storage system. Then, adding delta compression into data deduplication could significantly increase the compression ratio, and it has been studied in several use cases [2, 8, 11, 19, 20, 33]. However, compared with data deduplication, the speed of delta compression is much slower because of the computationally expensive process of delta encoding (i.e., calculating the differences between similar chunks or files). Shilane et al. [20] suggest that the average delta encoding speed of similar chunks is a potential bottleneck in delta-compression-based storage systems.

The classic delta encoding approaches, such as Xdelta [14] and Zdelta [21], follow the idea of general compression, which includes characterization on sliding windows and finding matches between source (or base) chunks and target (or input) chunks. Precisely, Xdelta and Zdelta will move the sliding window byte-by-byte, calculate the hash value of the sliding window (word) content, and search for the longest duplicate word between similar chunks, which is a time-consuming fine-grained matching process. Although Ddelta [26] and Edelta [27] are proposed to significantly improve the delta encoding speed by using Content-Defined Chunking (CDC) [29] and Greedy-based word-matching techniques, both sacrificing the compression ratio. We find that the decreased compression ratio is due to these two approaches' coarse-grained word-matching techniques. Concretely, these two methods will split chunks into several non-overlapping words by CDC and look for repeated words between similar chunks, significantly reducing the frequency of duplicated word matching without byte-by-byte operations. However, data modifications may cause the CDC not to split the same sub-chunks in some areas, resulting in failed word-matching and a decrease

in compression ratio. When we look back at Xdelta and Zdelta, their high compression ratio and slow speed result from their fine-grained word-matching techniques.

Through in-depth analyses of existing delta encoding methods, we carefully design our delta encoding method, Gdelta, to balance speed and compression ratio. Generally, the state-of-the-art approaches [14, 21, 26, 27] for delta encoding follow a similar framework to calculate differences from similar files or chunks. These approaches can be divided into three steps: *Characterizing*, *Matching*, and *Encoding*. ① In the *Characterizing* step, a series of content (i.e., a small word) characteristics of two similar items will be collected (i.e., hash values of all subchunks or all sliding windows). Then, words will be built indexes corresponding to the data structure (e.g., hash table and hash array). ② In the *Matching* step, the generated words in the target chunk will be searched in the characteristics set of the base chunk to find the repeated words. Ddelta and Edelta only generate a small set of words for chunks by introducing CDC in the *characterizing* step to simplify the *Matching* step. ③ In the *Encoding* step, the matched words between similar items will be encoded into instructions, and the unmatched words will be further compressed by entropy encoding (e.g., Huffman coding) and encoded into instructions also.

From our empirical and analytical studies, we obtain three important observations in three critical steps of delta encoding respectively:

(1) **The rolling hash bottleneck in the Characterizing step**. We observed that Xdelta uses an expensive rolling hash called Adler32 in the *Characterizing* step, which occupied 44.7% of the CPU time according to our experimental studies using the CPU performance analysis tool 'Perf'. Here, rolling hash is running in Xdelta that moves across files/chunks byte-by-byte to generate words for maximizing the duplicated word-matching.

(2) **The time-consuming fine-grained repeated word searching process in the Matching step**. We observe that both Xdelta and Zdelta adopt a byte-by-byte repeated-word searching scheme between similar chunks in the *Matching* step, as described above. Unfortunately, in the non-redundant area where there are no duplicate data, such a search process will introduce unnecessary hash computation, hash lookup, and content comparison costs.

(3) **The further compression opportunity in the Encoding step**. We observed that Zdelta achieves the highest compression ratio (about 5-30% higher) among the existing delta compression approaches since it detects duplicate words at a much smaller granularity of 3 bytes and then employs Huffman encoding to further compress the rest of the unmatched words after the delta calculation in the *Encoding* step.

Based on these observations, (1) firstly, and inspired by the previous fastest CDC algorithm, FastCDC [29], that employed a fast rolling hash called Gear to improve the content-defined chunking speed significantly, we believe that the Gear rolling hash can also help speed up the *Characterizing* step in delta encoding. (2) Secondly, according to the design of Ddella and Edelta, we find that reducing the frequency of searching repeated words between similar chunks can improve the speed of the *Matching step* compared with byte-by-byte operation. (3) Finally, the recently proposed fast compression tools, such as Zstd [4] and FSE encoding [1], also provide an opportunity to improve the delta compression ratio while maintaining a high compressing speed in the *Encoding step*. Therefore, motivated by the above observations and discussions, we propose Gdelta, a fast delta encoding approach with a high compression ratio. The contributions of this paper are six-fold:

- We introduce an improved Gear-based rolling hash to replace the Adler32 rolling hash to scan words of similar chunks. Gear hash runs much faster than Adler32 rolling hash while maintaining the same hashing efficiency.
- We further accelerate Gear rolling hash by the technique of rolling two bytes each time during indexing words of base chunks.

- We adopt the fast array-based indexing structure to save and look up the positions of words. This data structure is more efficient and has a similar compression ratio to that of the hash table in Gdelta.
- We introduce a skipping unmatched words mechanism to accelerate Gdelta pass through the non-redundant areas without byte-by-byte operations. This technique will not sacrifice the compression ratio much as Ddlta and Edelta.
- We introduce the Zstd and batch compression scheme to compress the rest after word-matching, further increasing the delta compression ratio.
- Evaluations suggest that Gdelta achieves encoding/decoding speedup of 4X∼16X over the classic Xdelta and Zdelta approaches while increasing the compression ratio by about 10%∼260%.

The rest of this paper is organized as follows. Section 2 introduces the background and related work. In Section 3, we discuss our essential observations on the classic Xdelta and Zdelta approaches, which motivate this work. Section 4 describes the design and implementation details of Gdelta. Section 5 presents the evaluation results of Gdelta, including comparisons with the known delta compression approaches, i.e., Ddelta, Edelta, Xdelta, and Zdelta. Finally, Section 6 concludes this paper and outlines future work.

2 BACKGROUND AND RELATED WORK

Table 1. A general comparison of three typical data reduction techniques: general compression, delta compression, and data deduplication. 'Appr. Dates' denotes the approximate dates of initial research for each technique.

	General compression	Delta compression	Data deduplication
Objects	All data	Similar data	Duplicate data
Compression Area	Local	Global and Local	Global
Granularity	String-/Byte-level	String-/Byte-level	Chunk-level
Rep. Techniques	Huffman coding [12]/ Dictionary coding	Copy/Insert-based Delta encoding	CDC & Secure signature
Appr. Dates	1970s	1990s	2000s
Rep. Prototypes	GZIP [10], Zlib [7], Zstd [4]	Xdelta [14], Zdelta [21], Ddelta [26]	Venti [17], LBFS , DDFS [37]

As a space-efficient technique for eliminating redundancy among very similar files or chunks, delta compression has gained increasing attention recently by reducing both the network and storage resource requirements in large-scale storage systems [8, 16, 19, 20, 24, 31, 33, 35, 40, 41]. In this section, we will first compare delta compression with other data reduction techniques and then introduce the latest work on delta compression.

There are many kinds of redundant data in storage systems. Thus, many data reduction techniques are proposed to process these redundant data respectively. Table 1 summarizes three typical data reduction techniques, also as detailed below:

- **General compression** (i.e., traditional lossless compression) methods, such as GZIP [10], focus on the internal compression of files or data chunks and eliminate byte- and bit-level redundant data by using Dictionary coding (e.g., LZ77 [38], LZ78 [39]) and Huffman coding [12], respectively. General compression methods are usually very time-consuming because of their fine-grained redundancy-eliminating schemes. They usually only compress data in a small region (e.g., a single file or a chunk), which means a local reducing redundancy area.

- **Delta compression** focuses on eliminating redundancy among similar files/chunks. Generally, it (e.g., Xdelta [14]) uses the Rabin-Karp-based string matching scheme to find repeated strings and then encodes matched (duplicate) strings with "Copy" instructions and unmatched strings with "Insert" instructions. In addition, Zdelta [21] further compresses unmatched strings at the byte-level by using Huffman coding [12]. Delta compression can be applied across an entire storage system by identifying similar files/chunks with resemblance detection techniques. The above-mentioned indicates that delta compression can search for redundancy globally and reduce it locally, combining the advantages of deduplication and general compression.
- **Data deduplication**, aiming at quickly and globally eliminating redundancy at a coarse granularity, is a chunk-level compression method for large-scale storage systems. Generally, it divides files into chunks of approximately equal length by Content-Defined Chunking (CDC) [15, 29, 34], and then a secure signature (or called fingerprint, e.g., SHA1) of each chunk is calculated for duplicate-matching. Two chunks will be considered duplicates if they have the same secure signature. The advantage of this method is simple, fast, and easy to implement in large-scale storage systems for global data reduction. But the disadvantage is that there is still a lot of redundant data between the similar but non-duplicate chunks/files, which can be removed by delta compression.

As discussed above, compared with general compression and data deduplication, delta compression is able to eliminate redundancy among very similar files/chunks, and it has been gaining increasing attention in recent years. Researchers in EMC [19] implemented delta compression on top of deduplication to further eliminate redundancy to accelerate the WAN replication of backup datasets, which suggests delta compression can help achieve about 2X higher compression ratio than the traditional ways. Yang et al. [32] employed the delta compression technique to eliminate redundancy among similar data blocks in SSDs, thus enlarging the logical space of SSD caches. Researchers in Microsoft [30, 31] proposed using delta compression techniques for similarity-based deduplication to improve the storage and network efficiency of online database management systems, which suggests delta compression can achieve 2X-5X higher compression ratio than the traditional deduplication-based approaches. Deng et al. [6] proposed a new delta encoder in its lossless deduplication system to compress similar JPEG images, which can achieve a 1.3X-1.6X higher compression ratio than traditional ways.

Generally, delta compression consists of two key stages: resemblance detection and delta encoding. **Resemblance detection** refers to the process of generating features and then searching similar chunks according to their features, which has been studied in many works [16, 20, 25, 28, 35, 40]. **Delta encoding** refers to the process of calculating the difference among very similar chunks/files (after resemblance detection), which determines the final compression ratio of delta compression. Delta encoding is time-consuming and is usually a bottleneck for delta compression-based systems, which has been gaining increasing attention recently [16, 27, 33, 41].

For delta encoding, Figure 1 shows a general example of the classical delta encoding method (e.g., Xdelta [14] and Zdelta [21]) on two similar chunks, i.e., input (target) chunk and its similar (base) chunk. The details are as follows.

(1) The *Characterize Step* first scans the base chunk to generate several overlapped words (e.g., strings with the length of 4 bytes) using a byte-wise rolling hash called Adler32 and then indexes the position of words in the hash table by Adler32 hash. It is worth noting that the hash table can store all identical words in one bucket (item) for the encoding method to find the best-matched words between similar chunks in the *Matching Step*.

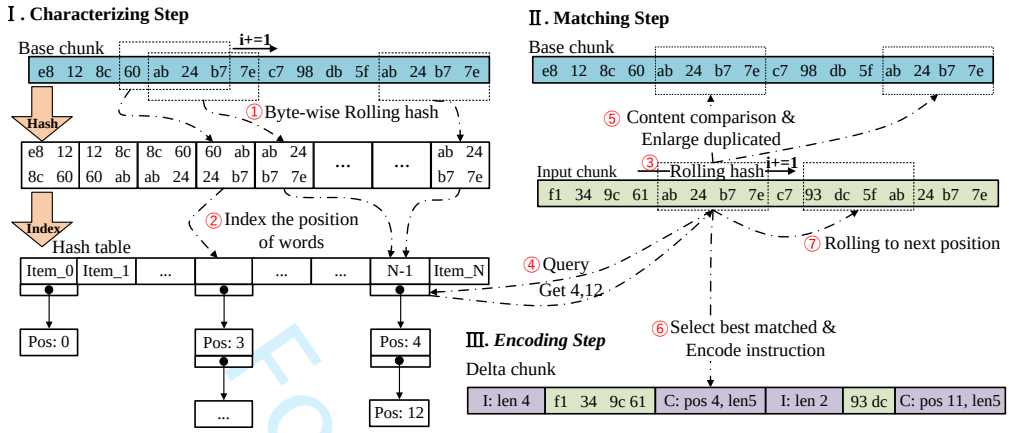


Fig. 1. A general example of the delta encoding process. 'C' and 'I' denote the 'Copy' and 'Insert' instructions in delta encoding, respectively.

Table 2. Comparison on different delta encoding approaches. Here inter match and inner match mean to detect the duplicate words between the base & input chunks and inside the input chunk, respectively.

Approa.	Chunking	Hashing	Moving Step	Indexing	Inter Match	Inner Match	Entropy Encoding
Xdelta	FSC	Adler32	1 Byte	Array	✓	✓	×
Zdelta	FSC	Adler32	1 Byte	Hash table	✓	✓	Huffman
Ddelta/ Edelta	CDC	Spooky/ xxHash	Sub-chunk size	Hash table	✓	×	×
Gdelta	FSC	Gear	Adaptive	Array	✓	✓	FSE

(2) The *Matching Step* uses the words in the base chunk as a "dictionary" and checks the words of the input chunk in the "dictionary" to identify redundancy. Firstly, the encoding method uses the same byte-wise rolling hash to generate words of the input chunk. Secondly, it queries the hash table to get the word positions of the base chunk by Adler32 hash. Thirdly, it compares the word content and enlarged duplicated words between the current position of the input chunk and different positions of the base chunk to find the longest matching word.

(3) The *Encoding Step* encodes the matched words as "Copy" instructions with offset and length messages and encodes the new (unmatched) words as "Insert" instructions with the actual words (data). Note that adjacent matched or unmatched words will be merged into one bigger instruction, respectively (to minimize the instructions' overheads). Finally, the input chunk is compressed into a smaller delta chunk $\Delta_{(Input,Base)}$ with "Copy" and "Insert" instructions for space savings. With the delta chunk $\Delta_{(Input,Base)}$ and the base chunk, the delta encoding method can easily restore the input chunk according to the 'Copy' and 'Insert' instructions.

There are also many other delta encoding schemes that are different from Xdelta and Zdelta. We summarize four state-of-the-art delta encoding approaches (including Xdelta [14], Zdelta [21], Ddelta [26], and Edelta [27]) with their features as shown in Table 2.

Generally, the characteristic of high-compression-ratio delta encoding methods, such as Xdelta and Zdelta, can be concluded as follows:

(1) Uses Fix-Sized Chunking (FSC) to generate fix-sized and overlapped words for word matching. Concretely, the rolling hash Adler32 uses a byte-wise sliding window to roll on the chunks' contents to generate word hashes in the *Charaterizing* step.
(2) Uses the moving step of 1 Byte. It means that when a word of the input chunk can't be matched in the base chunk, both Xdelta and Zdelta will move the sliding window by 1 byte and repeat the hash calculation and content comparison between similar chunks in the *Matching* step.
(3) Zdelta uses a hash table to save words for finding the longest-matched content between similar chunks and employs entropy encoding (i.e., Huffman coding) to compress the rest after delta encoding for a higher compression ratio.
(4) Detects redundancy (duplicate words) between similar chunks and redundancy inside the input chunk (i.e., inter match and inner match).

Besides, the characteristic of high-encoding-speed delta encoding methods, such as Ddelta and Edelta, can be concluded as follows:

(1) Uses Content-Defined Chunking (CDC) to generate variable-sized and non-overlapping words to simplify word-matching. All words will generate a weak hash for indexing and matching.
(2) After each word-matching failure, the moving step is a sub-chunk, reducing the computation of hashing of words and content comparison, which is the core idea of the high speed of Ddelta and Edelta.
(3) Only detects redundancy between similar chunks and without entropy encoding.

In this paper, we focus on designing a high-speed delta encoding approach (called Gdelta) with a high compression ratio, i.e., accelerating the delta calculation process of the already-detected delta-compression candidates, which is a significant improvement over the previous work Ddelta/Edelta [26, 27]. In Table 2, we also list the features of our Gdelta, which is generally the same as Xdelta and Zdelta for a high compression ratio. However, we significantly change the hashing, indexing, moving step, and entropy encoding schemes in Gdelta, as explained later in the following sections, according to our observations of running the four state-of-the-art delta encoding approaches with seven real-world datasets. *The core idea of Gdelta is to explore the potential of fast delta encoding for a much higher compression ratio than state-of-the-art approaches.*

3 MOTIVATION AND OBSERVATIONS

As mentioned in Section 2, existing delta encoding approaches target either delta compression ratio (e.g., Xdelta [14] and Zdelta [21]) or delta encoding speed (e.g., Ddelta [26] and and Edelta [27]). Figure 2 shows the delta compression performance of the four state-of-the-art approaches on seven real-world datasets (whose characteristics are detailed in Table 4 in Section 4). Here compression ratio is measured by $\frac{Total\ size\ of\ input\ chunks}{Total\ size\ of\ delta\ chunks}$ (for the already-detected similar chunks). The results suggest that Ddelta and Edelta represent the high-speed delta compression approaches but at a low compression ratio. In contrast, Xdelta and Zdelta represent the classic delta compression approaches focusing on the compression ratio. Specifically, taking Xdelta and Edelta as an example, Xdelta achieves significantly higher compression ratios than Edelta, with improvements of 10.2X (TAR), 9.7X (LNX), 5.1X (WEB), 7.3X (VMA), 7.6X (VMB), 3.0X (RDB), and 6.8X (CHM). In contrast, Edelta achieves dramatically higher delta encoding speeds than Xdleta, with improvements of 3.2X (TAR), 3.7X (LNX), 5.1X (WEB), 4.1X (VMA), 4.6X (VMB), 5.0X (RDB), and 4.5X (CHM).

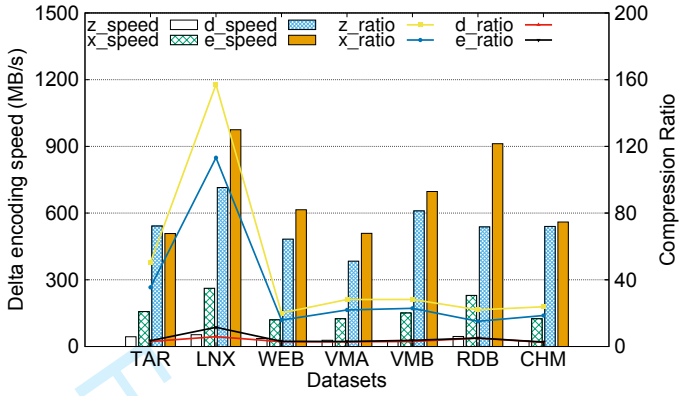


Fig. 2. Delta encoding speed and compression ratio of Xdelta, Zdelta, Ddelta, and Edelta on seven datasets.

As shown in figure 2, both Xdelta and Zdelta can remove over 90% of redundant data in the dataset, while Delta and Edelta can only approximately remove over 60% of it in most datasets. For Ddelta and Edelta, we find the decreased compression ratio is due to their coarse-grained word-matching techniques. To reduce the calculation of the *Mathching* step, the moving step (i.e., sub-chunk size in Table 2) is set up from 32B to 256B [26] in Ddelta and Edelta, which is much higher than Xdelta and Zdelta. It means that if a word of input chunk (8KB) fails to find duplicates in the base chunk, there are 256B unmatched data to be left at most. When we look back at Xdelta and Zdelta, their high compression ratio (about 3X-10X higher over Ddelta and Edelta) and the slow speed are from their fine-grained word-matching techniques, which provide three important observations for us as discussed below.

Observation ①: **The rolling hash bottleneck in the Characterizing step**. We find that Xdelta uses a time-consuming rolling hash called Adler32, which occupies 44.7% of the CPU time according to our experimental studies (running on seven datasets, also detailed in Section 5) using the CPU performance analysis tool 'Perf'. Here, the Adler32 rolling hash is used as the sliding window technique that moves on the similar chunks (files) byte-by-byte to generate words for maximizing the duplicate word matching in Xdelta. More concretely, Xdelta uses the sliding window size of 9 bytes to detect duplicate words between the base and input chunks and the sliding window size of 4 bytes to detect duplicate words inside the input chunks to detect redundancy as much as possible.

Observation ②: **The time-consuming fine-grained repeated word searching process in the Matching step**. We find that both Xdelta and Zdelta introduce a byte-by-byte sliding window to search repeated words between similar chunks to increase the compression ratio as much as possible. Unfortunately, in the non-redundant area where there are no duplicate data, such a search process will introduce a lot of additional computation costs such as hash computation, hash lookup, and string comparison without compression ratio improvement. Although Ddelta and Edelta introduce a big moving step to avoid byte-by-byte operations, they both sacrifice the compression ratio due to the coarse-grained word-matching.

Observation ③: **The further compression opportunity in the Encoding step**. We observe that Zdelta achieves the highest compression ratio (about 20-50% higher) among the existing delta compression approaches. This is because Zdelta uses a much smaller sliding window size of 3 bytes to detect the fine-grained redundancy for delta calculation among similar chunks (and also inside the input chunk) and then employs a Huffman encoding technique to further compress the remaining unmatched words (to increase the compression ratio).

For the problem in observation ①, we are inspired by the previous work on FastCDC [29] and Ddelta [26] that employs a fast rolling hash called Gear to improve the Content-Defined Chunking speed for data deduplication. We believe that Gear hash can also help speed up the *Characterizing Step* in delta encoding. To the best of our knowledge, Gear appears to be one of the fastest rolling hash algorithms for CDC at present since it uses many fewer calculation operations than others. In addition, we further accelerate the Gear rolling hash when scanning words of base chunks by the technique of rolling two bytes each time.

For the problem in Observation ②, To avoid byte-by-byte operations in Xdelta and a big moving step in Ddelta, Gdelta introduces a skipping unmatched words mechanism to change its moving step adaptively. Gdelta will first use a small moving step to detect duplicated words between similar chunks. When it fails to find duplicated words multiple times, it will increase its moving step gradually, which helps Gdelta quickly pass through areas without redundant data. Compared with Ddelta and Edelta, our method can still maintain the high ability to remove redundancy between similar chunks but also avoid byte-by-byte operations.

For the opportunity in Observation ③, We designed a two-stage compression strategy. In the first stage, techniques such as fast-rolling hashing, hash array indexing, and adaptive moving step were used to eliminate redundant data in similar chunks for a high encoding speed and reduce most input chunks' size. In the second stage, the recently proposed fast compression tools, such as Zstd [4] and FSE encoding [1], were used to eliminate internal redundancy in input chunks combining our batch compression scheme for fine-grained reducing data in the *Encoding Step*. The balanced two-stage compression strategy provides opportunities to improve the delta compression ratio without sacrificing fast compressing speed.

In conclusion, motivated by the three critical observations, we carefully design the workflow of Gdelta by taking into account the *Characterize Step*, *Matching Step*, and *Encoding Step* of delta encoding to achieve a balanced performance of speed and compression ratio.

4 DESIGN AND IMPLEMENTATION

4.1 Gdelta Overview

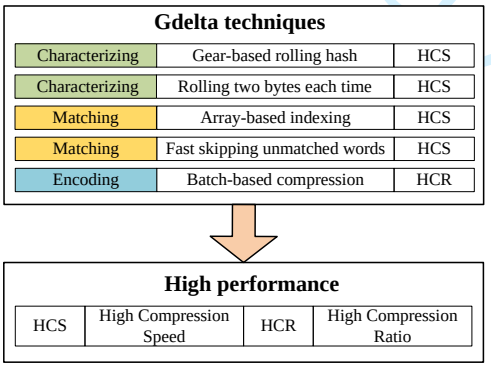


Fig. 3. The five key techniques used in Gdelta and their corresponding benefits for high performance.

Gdelta aims to provide high delta encoding performance for compressing similar chunks and files, including a high compression ratio and fast encoding/decoding speed. Generally, it is difficult,

if not impossible, to improve these two performance metrics simultaneously because they can be conflicting goals. For example, presetting a small minimum repeated word size usually leads to a higher compression ratio but at the cost of lower compression speed and higher metadata overheads. Thus, Gdelta is designed to strike a sensible trade-off among these two metrics to strive for high-performance delta encoding by combining the five techniques with their complementary features in Figure 3. The core idea of trade-off is the two-stage compression strategy. The first stage uses HCS technology in Figure 3 for fast delta compression between similar chunks, while the second stage uses HCR technology in Figure 3 for fine-grained internal compression of delta chunks generated from the first stage.

We will introduce the five technologies of Gdelta in the following order: (1) Gdelta employs the improved Gear rolling hash to replace the Adler32 rolling hash for generating overlapped words (detailed in subsection 4.2). (2) Gdelta employs the fast array-based indexing scheme (detailed in subsection 4.3) to replace the hash table for fast updating and searching index speed. (3) Gdelta uses the technique of rolling two bytes each time (detailed in subsection 4.4) to speed up the rolling hash while building the index of words for base chunks. (4) Gdelta introduces the fast skipping unmatched words mechanism (detailed in subsection 4.5) for word-matching to avoid byte-by-byte operations. (5) After word-matching between similar chunks, Gdelta uses compact encoding instructions to record the delta chunk (detailed in subsection 4.6) and further compresses the remaining bytes using Zstd and FSE to improve the compression ratio (detailed in subsection 4.7).

4.2 Gear-based Rolling Hash

Table 3. The comparison of two rolling hashes: Adler32 and Gear. Here 'a', 'b', and 'N' denote the contents of the first byte, the last byte, and the size of the sliding window, respectively. 'Gear' [26] denotes the predefined arrays, and 'fp' represents the fingerprint of the sliding window.

Name	Pseudocode	Speed
Adler32	$Low \mathrel{+}= b - a;$ $High \mathrel{+}= Low - a * N;$ $fp = (High << 16) \mid Low$	Slow
Gear	$fp = ((fp << 1) + Gear(b))$	Fast

As discussed in Section 3, Gear hash has the potential to replace the Adler32 used for generating overlapped words to maximize the duplicate-words matching. Thus we first introduce the Gear hash and then discuss our design of the improved Gear-based rolling hash used in Gdelta.

As shown in Table 3, the Gear rolling hash runs in two key ways: (1) It employs an array $Gear[]$ of 256 random 64-bit integers to map the values of the byte contents in the sliding window, and (2) The addition ("+") operation adds the new byte in the sliding window into Gear hashes while the left-shift ("<<") operation helps strip away the first byte of the last sliding window. This is because, after the '<<' and modulo operations, the first byte a will be calculated into the fp as $(Gear[a] << N)\ mod\ 2^N$, which will be equal to zero. Note that the '$mod\ 2^N$' operations can be eliminated if we set the fp as a 32- or 64-bit integer while using the sliding window size of 32 or 64 bytes accordingly. As a result, Gear rolls through the content using only three operations for each byte (i.e., "+", "<<", and an array lookup), enabling it to move quickly through the data content for the rolling hash function. It is also worth noting that in Content-Defined Chunking [29], the sliding window size can be adjusted by using a mask value for the hash judgment during chunking, which is distinct from the application scenario of this paper.

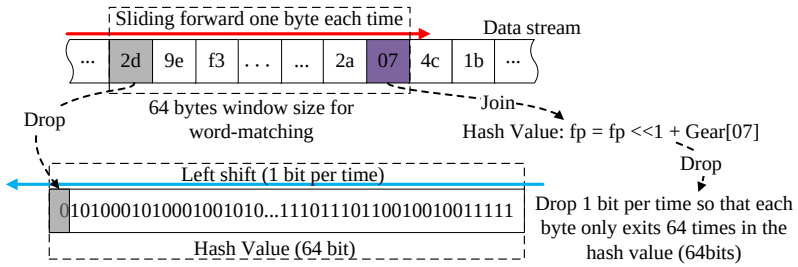


Fig. 4. A general example of Gear-based rolling hash in Gdelta.

Table 3 also shows the detailed implementation of Adler32 for comparison, which suggests Gear uses fewer calculations than Adler32. Thus, Gear is a good rolling hash candidate for Gdelta. Note that the sliding window size in Gear needs to be slightly adjusted to satisfy the fast-rolling hash requirements in Gdelta. More concretely, if we use the sliding window size of 8 bytes similar to Xdelta, we can implement Gear as $fp = ((fp << 1) + Gear(b)) \ mod \ 2^8$. Also, we can implement Gear as $fp = ((fp << 8) + Gear(b))$ for rolling while fp is a 64-bit integer, which eliminates the '$mod\ 2^8$' operation and thus further speeds up the rolling hashing process. Thus, the second implementation is more reasonable, and we choose the sliding window size of 64 bytes as our optimal configuration through our experimental results. Therefore, the formula in Table 3 is finally used in the remainder of this paper with a 64-bit integer fingerprint.

Figure 4 provides a detailed example of calculating the Gear hash in Gdelta while using the sliding window for word-matching in delta compression. The number of left-shift bits N can be calculated from the hash value bits V and the sliding window size s: $N = V/s$, which lets the hash value fp exactly represent the contents of the sliding window (used for word-matching) in delta encoding. More important, our evaluation results suggest that: By using the sliding window size of 64 bytes (Gear rolling hash is set as $fp = ((fp << 1) + Gear(b))$ and fp is a 64-bit integer) and combining other further compression techniques, Gear can obtain a higher compression ratio over Zdelta and Xdelta while achieving a much faster encoding speed (i.e., accelerating the duplicate word-matching process).

Although Gdelta, Ddelta, and Edelta have both chosen more oversized sliding windows, the moving step size of Gdelta does not directly skip the entire sub-chunk (i.e., sliding window). On the contrary, Gdelta chooses to increase the moving step progressively (detailed in Section 4.5) for a similar high encoding speed. Besides, this scheme ensures that Gdelta has a stronger ability to eliminate inter-redundancy than Ddelta and Edelta. Then, Gdelta improves the compression ratio through batch compression (detailed in Section 4.7). These techniques make Gdelta have the advantages of both Edelta (speed) and Xdelta (compression ratio).

Note that Gear hash has also been used in many studies [26, 29] for Content-Defined Chunking, which targets quickly getting the chunking positions (i.e., split files into several small and independent data chunks) in data deduplication-based systems. In our paper, we use Gear hash as a rolling hash for word-matching in delta encoding to replace the time-consuming Adler32 in Xdelta. And we slightly change the implementation of Gear (i.e., left-shifting more bits) to control its sliding window size to satisfy the requirement of the minimum word size used for word-matching. Through the experiment results of different word sizes, we choose the large word size (64 Bytes) as our final configuration for a higher compression ratio.

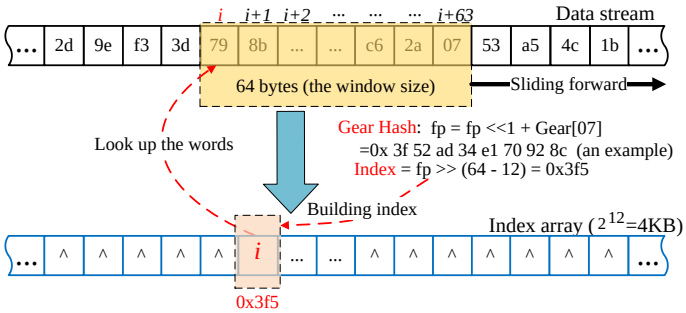


Fig. 5. A general example of the array-based indexing scheme combining with Gear-based rolling hash in Gdelta.

4.3 Array-based Indexing Scheme

As shown in Figure 5, in the *Characterizing* step, the word address of the base chunk will be indexed by the word hash (i.e., rolling hash or weak hash) for duplicated word matching (i.e., byte-by-byte string comparison). There are usually two types of hash data structures, namely hash array with open addressing method and hash table with chain address method, to solve hash collision. Zdelta adopts a hash table combined with a 3-byte word design, which has the advantage of finding all possible duplicate words between similar chunks and greedily selecting the best match. Gdleta finally chooses the hash array without open addressing to index address of words for the following reasons: ① Gdelta chooses a larger word length as the minimum length of the duplicated string between similar chunks. On the one hand, compared with the small words, the big words have less probability of hash collision. Besides, the Gear rolling hash has a low probability of hash collision (detailed in section 5.2). On the other hand, there is little probability that a chunk has many identical big words in different positions. ② Saving all small words in Zdelta is to find enough long duplicated words, while a large word setting already means significant matching benefits. ③ The update operation of the hash table and the content comparison of all candidate words are very time-consuming. Based on the above considerations, Gdelta adopts the hash array as the index structure for words and will directly update the address of words in place when they collide, which is consistent with Gdelta's fast compression scheme of similar chunks.

Specifically, after generating overlapped words by Gear hash, Gdelta uses a simple array-based indexing scheme (similar to Zstd [4] and Xdelta [14]) to look up the duplicate words according to their Gear hash. Take 64-byte words (i.e., the sliding window of 64 bytes) as an example to illustrate our indexing scheme in Figure 5, After running Gear hash, we get a 64-bit fingerprint for each word by computing $fp = ((fp << 1) + Gear(b))$. Thus, fp's 16 most-significant bits are calculated from all the bytes in the sliding window, while the 16 least-significant bits are just calculated from the latest sixteen bytes in the sliding window (since the last 48 bytes have been moved away here by using the "$<< 1$" operation many times). Therefore, we use the most-significant bits of fingerprint as the index of a hash array by the right-shift operation, which can effectively reduce the hash collision. Moreover, the number $maskBts$ of the most-significant bits used for the index can be minimized as $\lceil log(Chunk.Length) \rceil$ (e.g., $maskBts$ = 16 bits when the length of the base chunk is 64KB), which can also be used as the minimal size of the elements in the indexing array in Gdelta for memory space-saving.

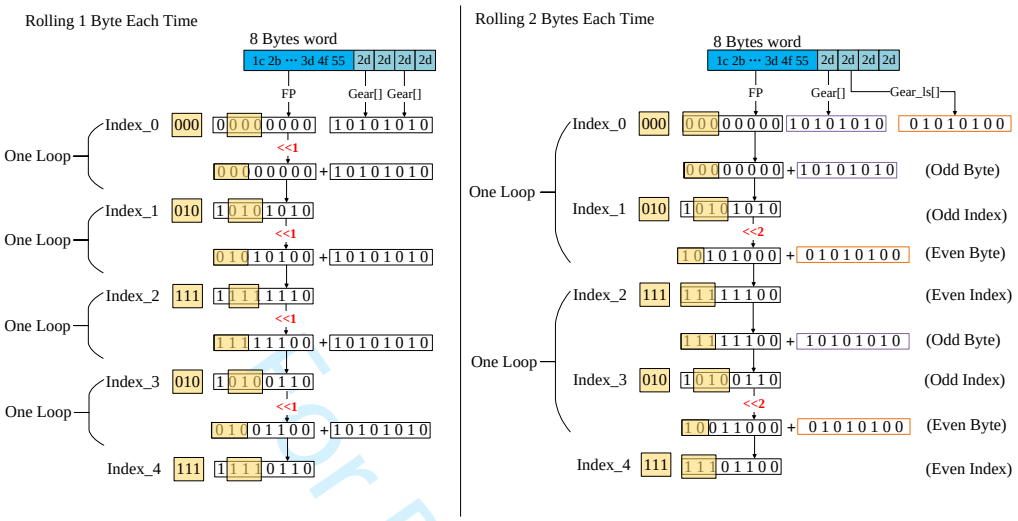


Fig. 6. A simple example of rolling two bytes each time when building word index of base chunks in Gdelta ($Gear\_ls[] = Gear[] << 1$).

According to our observation, the traditional hash table's memory cost is about 2-3 times more than this array-based index since the hash table needs to store additional data points and the offset information for each indexing unit [5]. Meanwhile, our performance evaluation of Gdelta suggests that compared with hash tables, this simple array-based indexing scheme achieves a comparable compression ratio and a fast look-up speed, as discussed later in Section 5.

4.4 Rolling Two Bytes Each Time

In addition to the techniques in the above subsections, we also propose an independent technique called "rolling two bytes each time" to speed up Gear rolling hash during indexing words from the base chunk. As shown in algorithm 2, the core idea of this technique is that the fingerprint fp rolls two bytes in each looping (i.e., $fp << 2$, merging two move-bits into one operation), in contrast to the traditional way of rolling one byte each time (see algorithm 1).

Figure 6 shows a simple example to explain the "rolling two bytes each time" technique. For the convenience of understanding, assuming the Gear fingerprint is an 8-bit integer, and the number of slots in the hash array is 8. Therefore, we can determine the index value by taking three digits in fp. In this example, to adapt to the design of "rolling two bytes each time", we take the 2nd to 4th high bits of the fingerprint as the index (as shown in rolling 1 byte each time in Figure 6). Specifically, "rolling two bytes each time" implements two index calculations in a function loop, merging the two left shift operations into one. To achieve this goal, We shift the random number table $Gear[]$ to the left one bit in advance to obtain a new random number table $Gear\_ls[]$. And in one loop, the index is obtained from two different positions of fp.

Logically, the word index still takes the highest bits of the fingerprint but discards the highest bit in this technique. In terms of implementation, ① for odd bytes, the index still gets the highest number of bits except for the highest ones, ② while for even bytes, we use the $Gear\_ls$ table ($Gear\_ls$ contains elements from the Gear table, which are all left shift one bit), and because the

fingerprint has been shifted more in advance, the index can directly obtain the highest number of bits.

This technique is simple but very useful for the Gear rolling hash since it introduces a loop unwinding optimization to reduce the number of function cycles and conditional judgments by half when rolling two bytes compared with the traditional approach (rolling one byte each time in Algorithm 1). Our technique ensures exactly the same indexing results and increases the speed of Gdelta by 10%~25% according to our evaluation results discussed in the next section.

Algorithm 1: Base Chunk Indexing Word

Input: Base data, bas[]
Output: hash array, arryInx[]

```
wordSze ← 64;
arryInx ← empty;
maskBts ← ⌈log(bas.Len)⌉;
fp ← 0;
for i = 0; i < wordSze; i ++; do
    fp ← (fp ≪ 1) + Gear[bas[i]];
Inx ← fp ≫ (64 − maskBts);
arryInx[Inx] ← i;
for ; i + wordSze < bas.Len; i ++; do
    fp ← (fp ≪ 1) + Gear[bas[i]];
    Inx ← fp ≫ (64 − maskBts);
    arryInx[Inx] ← i; //Build index for the Base words;
return arryInx;
```

Algorithm 2: Base Chunk Indexing Word By Rolling Two Bytes Each Time

Input: Base data, bas[]
Output: hash array, arryInx[]

```
wordSze ← 64;
arryInx ← empty;
maskBts ← ⌈log(bas.Len)⌉;
fp ← 0;
for i = 0; i < wordSze; i ++; do
    fp ← (fp ≪ 1) + Gear[bas[i]];
fp ← fp ≪ 1;
for ; i + wordSze < bas.Len; i+=2; do
    Inx ← fp ≫ (64 − maskBts); //Even Index;
    arryInx[Inx] ← i; //Build index for the Base words;
    fp ← fp + Gear[bas[i]]; //Odd byte;
    Inx ← (fp ≪ 1) ≫ (64 − maskBts); //Odd Index;
    arryInx[Inx] ← i + 1;
    fp ← (fp ≪ 2) + Gear_ls[bas[i + 1]]; //Even byte;
return arryInx;
```

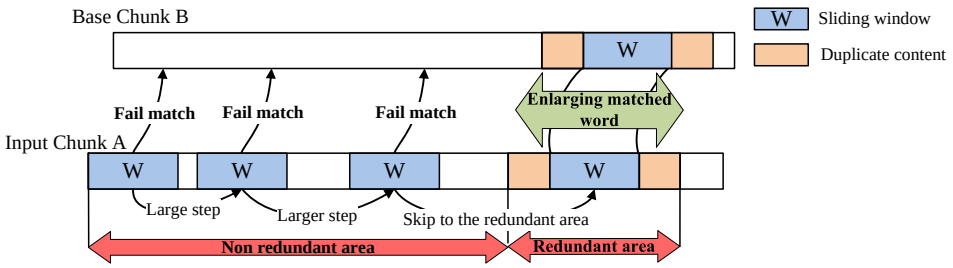


Fig. 7. A general example of skipping unmatched words mechanism in Gdelta.

4.5 Fast skipping unmatched words scheme

As mentioned in Section 3, Xdelta and Zdelta divide the input chunk into several overlapping words by a sliding window that moves byte-by-byte, and they search for repeated words in base chunks to improve the compression ratio as much as possible. However, this search mechanism will introduce unnecessary computing overhead (i.e., hash computing, hash lookup, and string comparison) when these two methods perform word-matching in areas without duplicate data. On the contrary, Ddelta and Edelta use the CDC method to split the chunk into large and non-overlapping words and skip the entire word when failing to find duplicated words, thus reducing the word-matching frequency and achieving a high delta encoding speed. The delta encoding method using CDC must ensure that the chunking points on both ends of the CDC are the same and the content of the sub-chunk is consistent to eliminate duplicate sub-chunk. However, data modification can lead to inaccurate chunking points in CDC (the wrong beginning or ending chunking point) or the same chunking points but different contents inside. Therefore, the compression ratio of Ddetla and Edelta will decrease due to the problem of chunking offset by CDC. Because a big moving step (i.e., 32-256B) will make a failure of word-matching significant loss for removing redundancy.

Therefore, our method adopts the same idea as that of Ddelta and Edelta to reduce the frequency of repeated word searches. When Gdelta identifies an area without duplicate content, Gdelta will quickly skip the area instead of computing and searching byte-by-byte. At the same time, we have taken into account that a large moving step may lead to a significant decrease in the compression ratios of Ddelta and Edelta. Therefore, in the initial search process, the sliding window of Gdelta will move in a small step (i.e., 1 byte) to ensure the compression ratio until Gdelta fails to search for repeated words multiple times. In conclusion, Gdelta will change the moving step adaptively.

As shown in Equation 1, Gdelta will record the last matching duplicate data endpoint $lastMatchPos$, and then subtract this value from the current position $Current$ to get a searched distance, and then divide this distance by a preset parameter N to get the next movement step. In practical implementation, we replace the division calculation with the right shift operation for speed. When Gdelta encounters an area where it is difficult to find duplicate words, Gdelta will move through the area by a large step rather than byte-by-byte, thus reducing needless computing operations. Note that the smaller the N obtained in Formula 1, the larger the step of the Gdelta movement. Finally, we set the parameter N equal to 2, the largest moving step, and test it in Section 5.5.

$$Step = (LastMatchPos - Current) >> N + 1 \quad (1)$$

Figure 7 shows the mechanism of skipping unmatched words for searching repeated words between similar chunks in Gdelta. In the area without redundant data, Gdelta skips over the area in

a larger step instead of byte-by-byte moving, which means an adaptive moving step in Gdelta. When the sliding window of word-matching falls into the redundant area, it can find all redundant data by greedily enlarging the matched word on the front and back sides. Therefore, the fast skipping non-redundant area mechanism will not significantly sacrifice the compression ratio because of the chunking offset problem and entire word moving in the CDC delta encoding method. Our experimental results show that the speed of Gdelta encoding is improved by 30%~70% with slight compression loss with our adaptive moving step mechanism.

4.6 Encoding and Decoding

Algorithm 3: Gdelta Encoding

Input: Base data, bas[]; Input data, ipt[];
Output: Delta data, dlt[];

```
1  dlt.Len ← 0;
2  anchor ← 0;
3  wordSze ← 64;
4  moveBts ← 64/wordSze;
5  arryInx ← empty;
6  maskBts ← ⌈log(bas.Len)⌉;
7  fp ← 0;
8  arryInx ← BaseChunkIndexingWord(bas[]);
9  for j = 0; j < wordSze; j ++; do
10     fp ← (fp ≪ moveBts) + Gear[ipt[i]];
11 for ; j + wordSze < ipt.Len; j ++; do
12     Inx ← fp ≫ (64 − maskBts);
13     ofst ← arryInx[Inx]; //Look up the input words;
14     for mLen ← 0; ; mLen ++; do
15         if bas[ofst+mLen]!=ipt[j-wordSze+mLen] then
16             break;
17     if mLen ≥ wordSze then
18         instIns.Len ← j − anchor; //Insert instruction;
19         copyIns ← (mLen, ofst); //Copy instruction;
20         memcpy(dlt + dlt.Len, instIns, size(instIns));
21         dlt.Len ← dlt.Len + size(instIns));
22         memcpy(dlt+dlt.Len, ipt+anchor,instIns.Len);
23         dlt.Len ← dlt.Len + instIns.Len;
24         memcpy(dlt + dlt.Len, copyIns, size(copyIns));
25         dlt.Len ← dlt.Len + size(copyIns);
26         anchor ← j + mLen;
27         j ← j + mLen − wordSze;
28         for ; j < anchor; j ++; do
29             fp ← (fp ≪ movebts) + Gear[ipt[j]];
30     else
31         fp ← (fp ≪ movebts) + Gear[ipt[j]];
32 return dlt.Len;
```

Algorithm 4: Gdelta Decoding

Input: Delta data, dlt[], Base data, bas[]
Output: Decompressed data, ipt[]

```
readLen ← 0;
ipt.Len ← 0;
while readLen < dlt.Len do
    Ins ← read(dlt + readLen, size(Ins));
    readLen ← readLen + size(Ins);
    if Ins == insertIns then
        memcpy(ipt + ipt.Len, dlt + readLen, Ins.Len);
        readLen ← readLen + Ins.Len;
        ipt.Len ← ipt.Len + Ins.Len;
    else if Ins == copyIns then
        memcpy(ipt + ipt.Len, bas + Ins.ofst, Ins.Len);
        ipt.Len ← ipt.Len + Ins.Len;
return ipt.Len;
```

Algorithms 3 and 4 provide the detailed pseudo code of the Gdelta encoding and decoding processes, also including the rolling hash computation (using the sliding window size of 64 bytes and 64-bits *fp*) and duplicate-words matching. Gdelta also uses the "Copy" and "Insert" instructions to record the duplicate and non-duplicate words of the input chunk, which is similar to Xdelta and Zstd (see Figure 1). Thus, for Gdelta encoding, the delta chunk is recorded by the "Copy" and "Insert" instructions in turn. Note that for the duplicate-words matching, greedy-based matching used in Edelta (called 'word-enlarging') is also applied in Gdelta, which can further reduce the rolling hash computation and duplicate-words matching operations (just by directly matching forward to detect the duplicate words as long as possible). For Gdelta decoding, the input chunk can be easily restored according to the "Copy" and "Insert" instructions in the delta-encoded chunk, which is the same as Xdelta and Ddelta.

4.7 Batch-based Further Compression

As mentioned in Section 3, we designed a two-stage compression strategy, with the first stage using fast delta encoding techniques (such as Gear hash, hash array, adaptive skipping search, and "rolling two bytes each time") to quickly eliminate redundant data between similar chunks. These are all aimed at improving the speed of delta encoding and ensuring a significant reduction in the size of input chunks. The experiment (detailed in Section 5.4) shows that even with a fast delta compression scheme, the size of the input chunks is basically reduced by less than 10%, which allows us to use fine-grained compression mechanisms in the second stage to reduce the internal redundant data of the data chunk (i.e., inner match).

Therefore, in the second stage, we will further compress the remainder after the Gdelta encoding given above in Algorithm 3. Two recently proposed compression approaches are employed here, Zstd [4], and FSE encoding [1]. Finite State Entropy (FSE) [1] is a fast encoding algorithm based on the recently proposed Asymmetric Numeral Systems (ANS) theory [9], which can represent a symbol with a fractional part statistically (i.e., smaller than one bit), which is more space-efficient than Huffman encoding for redundancy elimination at the bit-level. Zstandard is also a fast general

compression approach that eliminates redundancy at both the word level (default size of 8 bytes) and bit level, recently proposed by Facebook.

To improve the compression efficiency in Gdelta, we compress the section of "Copy" and "Insert" instructions at the bit-level using FSE and compress the data part in the "Insert" instructions at both the word- and bit-level using Zstd. Moreover, to maximize the compression ratio of Gdelta, we put several delta chunks altogether into a buffer with a size of 4MB. This is because batch-based processing efficiently reduces the metadata overheads for data compression, such as the indexing arrays, dictionaries, etc., used in FSE and Zstd for each delta chunk. Note that the data section and the instructions section in a buffer are batch-compressed individually and then stored separately. Because there is almost no redundancy between them, and putting them all together for batch compression will increase the compression metadata overhead. We will discuss the batch size parameter in Section 5.4 by experiment.

5 EVALUATION

5.1 Experimental Setup

To evaluate delta encoding performance, we implement a prototype of the delta compression system on the Ubuntu 18.04.1 operating system running on an Intel Xeon(R) Gold 6130 processor at 2.1GHz, with 128GB of memory. Two known delta encoding approaches with a high delta compression ratio, Xdelta [14] and Zdelta [21] (we use the latest Xdelta 3.1 and Zdelta 2.1), are used with their default configurations as the baselines for evaluating Gdelta. On the other hand, Gdelta is configured using the word size of 64 bytes (the smallest matching unit) and a Zstd compression level of 5 (for fine-grained inner match). In addition, the delta compression prototype consists of approximately 6,000 lines of C code, which is compiled by GCC 7.5.0 with the "-O3" compiler option to maximize the speed of the resulting executable. Note that Edelta [27] and Ddelta [26] have a much lower compression ratio than other approaches (see Figure 2 in Section 2), we only compare Gdelta with their encoding speed and compression ratio in this section, instead of treating them as baselines.

Three main metrics for delta compression are used for this evaluation, namely, *Compression ratio (CR)* measured in terms of $\frac{\textit{Total size of input chunks}}{\textit{Total size of delta chunks}}$ by running a given delta compression scheme on the detected similar chunks, *encoding speed* and *decoding speed* recorded by the response time of delta calculation and restore, respectively, in memory by a given evaluated delta encoding scheme. Note that for the *encoding* and *decoding speed*, we run each experiment five times to get stable and average results.

The datasets used in the evaluation are generated from seven real-world workloads in various use cases, as shown in Table 4. These datasets are typical workloads for delta compression and have been used in many delta compression or data deduplication related papers [26, 27, 29, 34, 35, 43]. Four of these datasets TAR[1], LNX[2], WEB[3], and VMA[4] are publicly available. To get similar chunks for delta encoding, we first apply Content-Defined Chunking (CDC) on these datasets to generate data chunks and detect similar chunks with a super-feature based resemble detection method [19]. The CDC method used here is FastCDC [29], and the minimum chunk size, average chunk size, and maximum chunk size are configured as 2KB, 8KB, and 64KB, respectively. The specific resemblance detection method is to compute 12 features from each chunk and group them into three Super-Features. Two chunks are considered to be similar if they share one or more Super-Features [19, 35].

[1]https://ftp.gnu.org
[2]https://www.kernel.org
[3]https://news.sina.com.cn
[4]http://www.thoughtpolice.co.uk

Table 4. Workload characteristics of the seven datasets used in the performance evaluation.

Name	Size	Workload descriptions
TAR	54 GB	215 tarred files from several open source projects such as GCC, GDB, Emacs, etc.
LNX	116 GB	260 versions of Linux kernel source code. Each version is packaged as a tar file
WEB	278 GB	120 days of snapshots of the website: news.sina.com.cn
VMA	138 GB	90 virtual machine images of different OS release versions, including Fedora, CentOS, Debian, etc.
VMB	278 GB	18 backups of an Ubuntu 12.04 VM image
RDB	431 GB	66 backups of the redis key-value store database
CHM	279 GB	Source code of Chromium project from v82.0.4066 to v85.0.4165

To better evaluate Gdelta, we implement three versions of Gdelta with different configurations of compression techniques (with the default array-based indexing) as introduced below:

- Gdelta_A: only detect duplicate words between similar chunks and use Gear hash with the default sliding window size of 64 bytes. Thus, Gdelta_A only uses inter match (without inner match), as shown in Algorithm 3 in Section 4.6.
- Gdelta_B: After Gdelta_A, further compress each delta chunk using FSE and Zstd to compress instructions and unmatched words, respectively.
- Gdelta_C: After Gdelta_A, group delta chunks into a buffer (with the default size of 4MB), and then further compress the buffer for a much higher compression ratio by using FSE and Zstd.

Therefore, based on the above experimental setup, we would like to answer the following questions in our evaluation:

- *How well does the Gear hash used for delta encoding in Gdelta compared with Adler32*? We answer this question in subsection 5.2 and study the hashing speed, hash collision ratio of Gear-based rolling hash, and its effect on the delta encoding performance.
- *What is the compression ratio difference between the hash array-based and hash table-based indexing schemes in delta encoding*? We answer this question in subsection 5.3 by evaluating the metrics of delta encoding speed and compression ratio.
- *How effective is the batch-based compression scheme in Gdelta*? We answer this in subsection 5.4 by evaluating Gdelta performance, enabling and disabling batch-based compression.
- *How much do the two techniques (i.e., adaptive moving step mechanism and rolling two bytes each time when building hash indexes for base chunks) improve the speed of Gdelta*? We answer this question in subsections 5.5 and 5.6 by evaluating Gdelta encoding speed, enabling and disabling these two approaches. Besides, we will compare the encoding speed of Gdelta, Ddelta, and Edelta to verify the efficiency of these techniques.
- *What is the final performance of Gdelta compared with the classic Xdelta and Zdelta*? We answer this in subsection 5.7 by studying the metrics of the encoding/decoding speed and the compression ratio by running these three approaches on the seven datasets.

Table 5. Hashing speed and collision ratio comparison of Gear and Adler32, with different sliding window sizes: 8B, 12B, and 16B.

Approaches	Thpt(MB/s)	Collision Ratio		
		10MB	20MB	50MB
Adel32(8B)	478.3	2.4E-10	2.5E-10	2.3E-10
Adel32(12B)	473.2	2.4E-10	2.5E-10	2.3E-10
Adel32(16B)	478.1	2.3E-10	2.2E-10	2.2E-10
Gear(+, 8B)	1738.8	2.4E-10	2.5E-10	2.4E-10
Gear(+,12B)	1741.5	2.4E-10	2.5E-10	2.4E-10
Gear(+,16B)	1741.3	2.3E-10	2.3E-10	2.3E-10

5.2 Gdelta Using Gear for Hashing

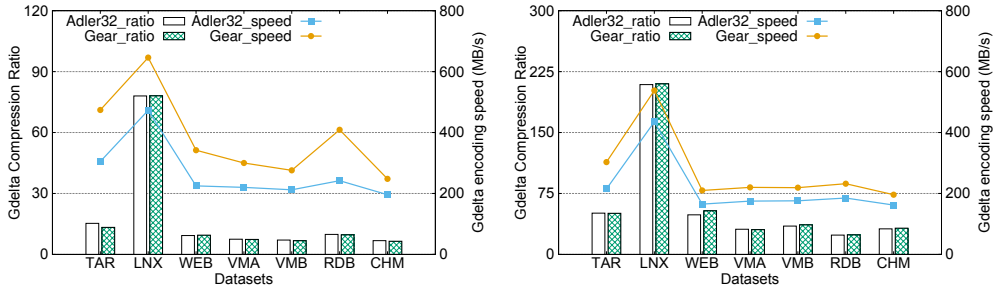
Hashing speed and collision ratio are the two most important metrics for evaluating the effectiveness of weak hashes Gear and Adler32. In this subsection, to evaluate the hash collision, we generate five test files consisting of random numbers for the given size of 10MB, 20MB, and 50MB (totaling 15 files). As introduced in Section 4.2, the Gear hash used in Gdelta has been slightly changed by left-shifting several bits instead of one bit in the original Gear [26], which is due to the necessary of different sliding window size (Gdelta uses 64 bytes) used for word-matching in delta encoding.

Table 5 shows the average hashing speeds and collision ratios of Adler32 and Gear. The evaluation is to calculate the hashes of words with different and small sliding window sizes (i.e., 8, 12, 16 bytes) on the random number workloads, which is often used for evaluating hash efficiency. The metric collision ratio listed in Table 5 is calculated according to Equation 2 as introduced below, which is the total combination of two unique words with an identical weak hash fingerprint divided by the combination of picking any two words from all the words in the test files.

$$P_{collision} = \frac{\sum_{i=2}^{n} n_i C_i^2}{C_n^2} \tag{2}$$

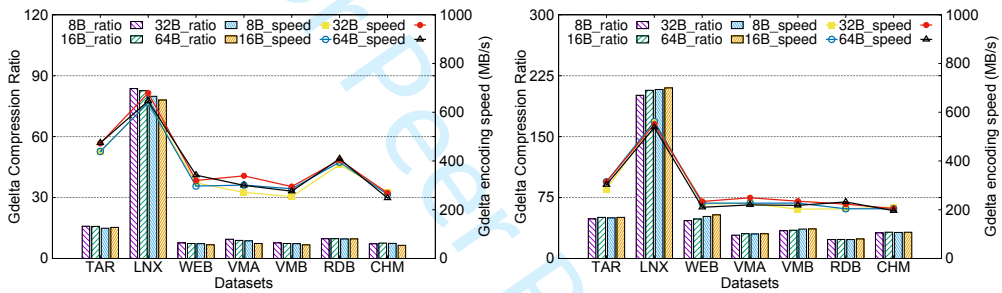
Here i represents the number of collision occurrences for a given weak hash while n_i corresponds to the number of such i-times colliding weak hash. For example, if ten different words share the same weak hash, and there are seven such weak hash, then i equals 10 and n_i equals 7). The results prove Gear achieves nearly the same hashing efficiency as Adler32. Essentially, picking two colliding words is equivalent to finding a new word that has the same weak hash fingerprint as the given one. Therefore, the probability is supposed to be $1/2^{64}$ (assuming that the hash value is a 64-bit integer) if the output of the hash function is uniformly distributed, which is consistent with our experiments. For example, for a target chunk, say 8KB (~8000 words in total and according to table 5, the collision probability is 2.5×10^{-10} at maximum), the maximum collision is calculated as only 0.016 (according to Equation 2, we can get the biggest collision case when $n_i = 0$ if $i \neq 2$) when using Gear to hash the words, indicating a quite low possibility for collision occurrences.

Table 5 also suggests that under the different file sizes, the hash speed of Gear reaches 1740 MB/s, about 3.5× of Adler32. The fast speed of Gear is because Gear consumes fewer instructions for the rolling hash compared with Adler32 [29]. For the collision ratio issue, we can observe that Gear and Adler32 have almost identical hash collision ratios, while the sliding window size has almost no influence on the collision ratio. It is worth noting that the hash collision ratio is acceptably low since our delta encoding candidates are usually smaller than 64KB, and a collision case would be detected when comparing bytes between the base and target chunks.



(a) Gdelta_A (with only inter match) (b) Gdelta_C (with both inter and inner match)

Fig. 8. Compression ratio and encoding speed of Gdelta (Gdelta_A and Gdelta_C) using Gear hash and Adler32 for fingerprinting, respectively.



(a) Gdelta_A (with only inter match) (b) Gdelta_C (with both inter and inner match)

Fig. 9. Compression ratio and encoding speed of Gdelta using different sliding window sizes for Gear fingerprinting.

To further study the hashing effect in delta encoding, we evaluate Gdelta performance when using Gear to replace Adler32, as shown in Figure 8. Here we configure Gdelta with array-based indexing and with/without batch-based compression. Specifically, Gdelta evaluated in Figure 8(a) only detects duplicate words among similar chunks, while Figure 8(b) configures Gdelta using batch-based compression. Generally, we can conclude from Figure 8:

- For the *compression ratio*, the two hash approaches achieve nearly the same compression ratio as discussed earlier in this subsection: the hash collision ratios of Gear and Adler32 are all very low. Note that Gdelta_C achieves a much higher compression ratio than Gdelta_A by using batch-based compression, which will be compared and discussed in Section 5.4.
- For the *delta encoding speed*, when only detecting the duplicate words among similar chunks, Gear achieves 20%-70% higher delta encoding speed over Adler32. while adding the technique of batch-based compression into the delta encoding process, Gear achieves 20% higher delta encoding speed over Adler32.

Figure 9 also studies how the sliding window size of Gear hash (i.e., the minimum unit used for word-matching) impacts the delta encoding speed and compression ratio in both Gdelta_A and Gdelta_C. Generally, we can conclude from Figure 9:

- For the *compression ratio*: a smaller sliding window size (i.e., 8 bytes) results in a higher compression ratio when Gdelta_A only removes redundancy among the similar chunks, as

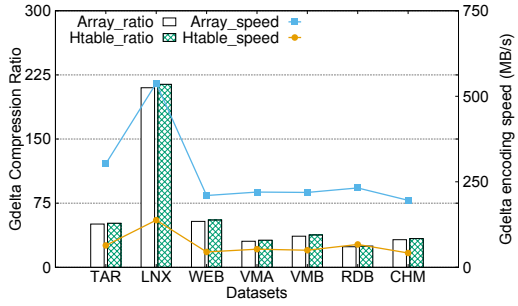


Fig. 10. Delta encoding speed and compression ratio of Gdelta (Gdelta_C) using an array or a table for indexing.

shown in Figure 9(a). But Figure 9(b) suggests Gdelta_C (with batch-based compression) achieves a higher compression ratio when using the sliding window size of 64 bytes, which is because we observed that a larger sliding window size leaves larger unmatched sequences of bytes, and compression algorithms tend to compress better on larger amounts of data.

- For the *encoding speed*: The influence of window size above 8B on speed is relatively small because we find that the real factor affecting speed is the moving step size after word-matching failure, which is the core idea of high encoding speed in Ddelta and Edelta.

In summary, Gear hash is shown to be 3.5X faster than Adler32 while keeping a very low hash collision ratio. It also achieves a very high compression ratio using the sliding window size of 64 bytes in Gdelta while combining with the technique of batch-based Zstd compression. Therefore, Gear is shown to be an excellent rolling hash function used for delta encoding in Gdelta.

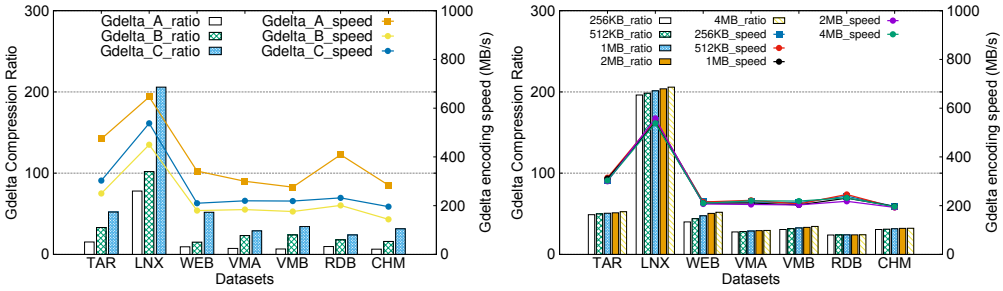
5.3 Gdelta Using Array-Based Indexing

In this subsection, we evaluate the Gdelta performance when using the technique of array-based indexing (i.e., quickly find a match), as shown in Figure 10. Here we configure Gdelta with Gear-based hashing and batch-based compression (i.e., Gdelta_C). Here the approach 'Htable' means Gdelta uses a hash table as used in Zdelta [21]: When searching for matches, a greedy strategy is used; All possible matches in the base and target chunks are examined and the best one (i.e., the longest duplicate words) is selected. Generally, we can conclude from Figure 10:

- For *compression ratio*, the hash table-based approach only achieves a slightly higher compression ratio. This means array-based indexing used in Gdelta works sufficiently well for word-matching in delta encoding when combined with batch-based compression.
- For *delta encoding speed*, array-based indexing achieves a much higher encoding speed because of its simple design, as shown in Figure 5 in subsection 4.3. On the other hand, the hash table-based approach takes much more time to find the best match for delta encoding while only achieving a minor improvement of the compression ratio on the seven datasets.

5.4 Gdelta Using Batch-based Compression

In this subsection, we evaluate Gdelta's (i.e., Gdelta_C) performance when using the technique of batch-based compression (i.e., putting many delta chunks together in a buffer for batch compression). Here we also evaluate Gdelta_A and Gdelta_B for comparison, as shown in Figure 11(a). Besides, we evaluated the impact of different buffer sizes (i.e., the size of grouped delta chunks) on the



(a) Comparison of Gdelta_A, Gdelta_B, and Gdelta_C (b) Comparison of Gdelta_C with different batched sizes

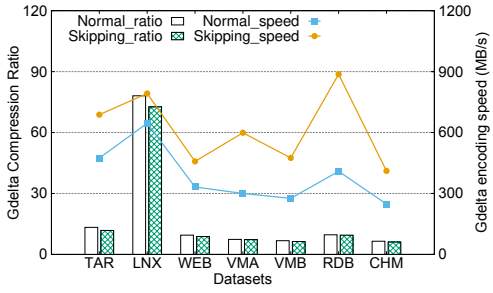
Fig. 11. Delta encoding speed and compression ratio of Gdelta_A, Gdelta_B, and Gdelta_C (with different batched sizes).

compression speed and compression ratio of Gdelta_C, as shown in Figure 11(b). Generally, we can conclude from Figure 11:

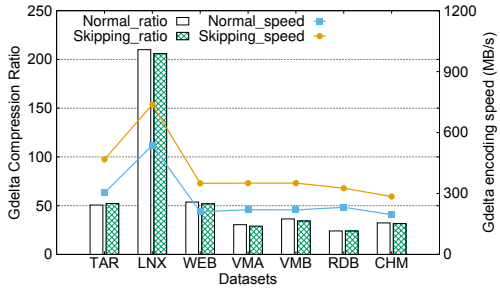
- For the *compression ratio*, the batch-based compression technique (i.e., Gdelta_C) improves the compression ratio by 25%~240% compared with only compressing the delta chunks individually (i.e., Gdelta_B). This demonstrates that there is still a lot of redundant data among the delta chunks after delta encoding between the target and base chunks. Meanwhile, Gdelta_B also achieves a much higher compression ratio than Gdelta_A, which suggests that inner match is also necessary after inter-match in delta encoding.
- For the *delta encoding speed*, the batch-based approach not only does not increase the time overhead but also achieves about 10% higher encoding speed than the approach that only compresses the delta chunks individually. This is because a 4MB buffer consists of hundreds of delta chunks, and this batch compression could reduce most of the initialization operations of Zstd and FSE techniques for the small delta chunks. Besides, the encoding speed of Gdelta_A decreases by 15%~40% after batch compression because Gdelta_C will perform fine-grained inner match and FSE encoding on the remained data of Gdelta_A for a high compression ratio.
- For the *batched sizes parameter*, on one hand, we find that the encoding speed has low relativity to the batched size when Gdelta_C has grouped the delta chunks together to compress. On the other hand, the compression ratio of Gdelta_C slightly increases as the batched size increases. Because the compression window of Zstd is 4MB, which is larger than our experimental batched size, generally, placing more data in the same compression window can achieve a higher compression ratio. The experiment shows that a 4MB batched size can improve the compression ratio by 5%~30% compared to the batched size of 256KB. We ultimately set the batched size of 4MB in Gdelta, which is aligned to the compression window of Zstd for higher compression ability.

5.5 Gdelta Using Fast Skipping Unmatched Words Mechanism

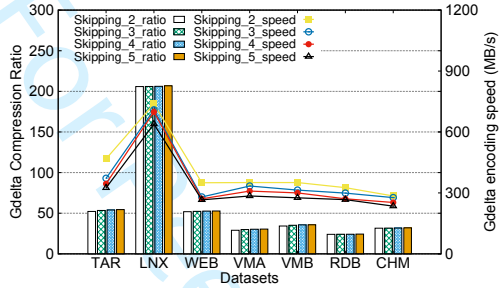
In this subsection, we evaluate Gdelta's (i.e., Gdelta_A and Gdelta_C) performance with the technique of fast skipping the non-redundancy area when searching for repeated words between similar chunks. We compare it with the traditional approach (i.e., byte-by-byte searching), as shown in Figure 12. Generally, we can conclude from Figure 12:



(a) Gdelta A (with only inter)



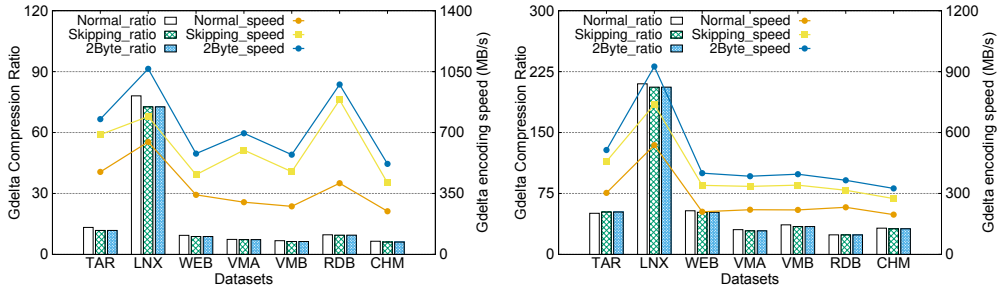
(b) Gdelta_C (with both inter and inner match)



(c) Gdelta_C with different skipping step (skippinng_N means that N is the preset parameter in Equation 1, and the step size decreases as N increases)

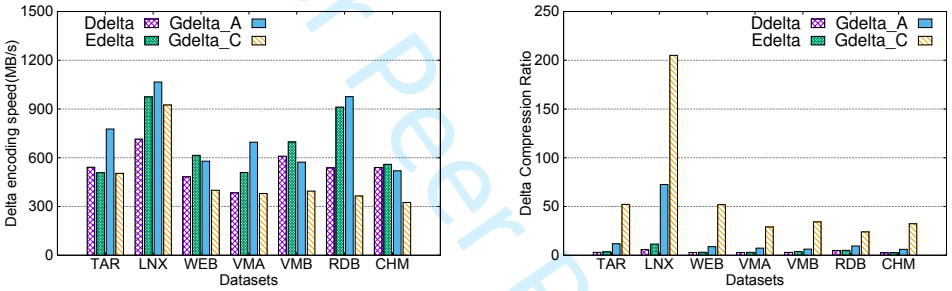
Fig. 12. Delta encoding speed and compression ratio of Gdelta_A and Gdelta_C.

- For the *delta encoding speed*, the adaptive moving step approach improves the speed of Gdelta_A by 30%~200% and Gdelta_C by 30%~70% on multiple datasets, compared with traditional byte-by-byte searching for repeated words. This is because this approach can identify the area without redundant data and skip the area with a large step, which greatly reduces unnecessary computational overhead.
- For the *compression ratio*, compared with byte-by-byte searching for repeated words, our approach only loses 3%~5% compression ratio. It is worth noting that Ddelta and Edelta speed up delta encoding through CDC, but their compression ratio is sacrificed because of the big moving step. However, our fast skipping unmatched words approach can only be performed smartly in areas without redundancy, which has little impact on the compression ratio and improves the encoding speed at the same time.
- For the *skip step parameter*, as the skipping step size decreases (i.e., the parameter N in Equation1 is set larger), the loss of compression ratio reduces, and the speed also decreases. As the step size decreases, it means that Gdelta performs a more fine-grained word-matching search, so the compression ratio is higher, but it introduces more computational overhead. Finally, we set the parameter N to 2, and Gdelta achieved the fastest speed with only a slight loss in compression ratio.



(a) Gdelta_A (with only inter match) (b) Gdelta_C (with both inter and inner match)

Fig. 13. Delta encoding speed and the compression ratio of Gdelta_A and Gdelta_C with rolling two bytes each time.



(a) Encoding speed (b) Compression Ratio

Fig. 14. Delta encoding speed and the compression ratio of Ddelta, Edelta, Gdelta_A, and Gdelta_C.

5.6 Rolling Two Bytes Each Time

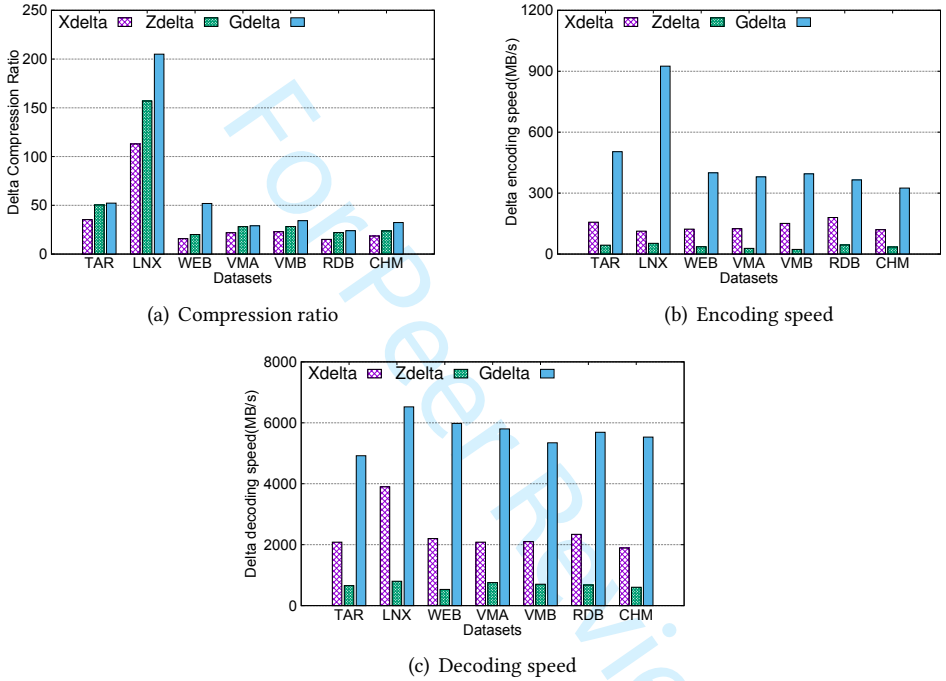
In this subsection, we evaluate the performance of Gdelta (i.e., Gdelta_A and Gdelta_C) by combining the technique of "rolling two bytes each time" when indexing the words of base chunks. In this experiment, except the normal version, other Gdelta versions adopt the skipping unmatched words mechanism. Generally, we can conclude from Figure 13 that:

- The technique of rolling two bytes each time further increases the Gdelta encoding speed by 10~25 percent since it further uses a loop unrolling optimization when calculating the Gear rolling hash, which can increase the concurrency of the CPU.
- Combining the skipping word match mechanism and the technique of rolling two bytes each time can improve the encoding speed of Gdelta by 70~90 % compared with the normal version.

In addition, we compare the delta encoding speeds of Gdelta (combining all techniques) with Ddelta and Edelta to validate all of our speed optimization techniques. As shown in Figure 14, Gdelta_ A and Edelta maintain the highest encoding speed in almost all datasets. In the TAR, LNX, VMA, and RDB datasets, Gdelta_ A achieves the fastest delta encoding speed, indicating that using techniques such as adaptive moving size to skip non-redundant area and "rolling two bytes each time" can achieve the same simplified word-matching effect as CDC delta encoding. Besides, Gdelta_ A has a 2X~6X higher delta compression ratio than Ddelta and Edelta in seven datasets. It suggests that Gdelta not only has a fast delta encoding speed but also has a better ability to eliminate

redundant data of similar chunks. After batch compression, the encoding speed of Gdelta_C is decreased by 10%~50% than Gdelta_A, while the compression ratio is improved by 3X~5X. But in TAR and LNX datasets, Gdelta_C still remains similar encoding speed to that of Ddelta and Edelta, which suggests that Gdelta obtains a high compression ratio without significantly sacrificing the computing performance. This is attributed to our balanced two-stage compression strategy.

5.7 Overall Performance



(a) Compression ratio

(b) Encoding speed

(c) Decoding speed

Fig. 15. Overall performance comparison of Xdelta, Zdelta, and Gdelta (i.e., the final version of Gdelta_C using five key techniques).

In this subsection, three approaches are evaluated for the overall performance comparison, including Xdelta, Zdelta, and Gdelta (i.e., Gdelta_C with all the techniques used: Gear-based hashing, array-based indexing, skipping unmatched words, rolling two bytes each time, and batch-based compression).

From Figure 15, we can find that compared with Xdelta, Zdelta always achieves a higher compression ratio but a much slower encoding/decoding speed since it uses Huffman encoding to further compress after its fine-grained word-matching. At the same time, Gdelta always detects more redundancy than Zdelta, especially on the WEB dataset, with about 2.6× higher compression ratio. This is because Gdelta uses batch-based compression with FSE and Zstd techniques to compress the instructions and data, which can not only remove the redundancy of the chunk itself at a fine-grained level but also eliminate the adjacent chunks' redundancy.

Meanwhile, as shown in Figures 15(b) and (c), Gdelta achieves much higher encoding and decoding speed. The reasons are four-fold. ① Gdelta uses the techniques of Gear-based rolling hash, rolls two bytes each time, and array-based indexing to accelerate building words index for similar

chunks in the *Characterizing* step. ② Gdelta employs the skipping unmatched words mechanism to avoid the byte-by-byte operations in the *Matching* step. ③ Gdelta uses batch-based compression and decompression, which helps accelerate batch delta encoding and decoding. ④ Gdelta also has a higher decoding speed than Xdelta because it uses a big word of 64 bytes.

In summary, Gdelta combines all the techniques mentioned in Section 4 and outperforms Xdelta and Zdelta on all three metrics, achieving a much higher compression ratio and throughput.

6 CONCLUSION AND FUTURE WORK

In this paper, we propose Gdelta and explore the potential of a fast delta encoding approach to achieve a high compression ratio, by taking into account the *Characterize*, *Matching*, and *Encoding* step of delta encoding. More concretely, it improves the delta encoding speed by employing an improved Gear-based rolling hash, a quick array-based indexing scheme, rolling two bytes each time, and an adaptive moving step scheme. Then, after word-matching, it further batch compresses the rest to improve the compression ratio using the latest Zstd and FSE techniques. The balanced two-stage compression scheme is the core idea behind Gdelta. Experimental results show that Gdelta speeds up the delta encoding/decoding by 4X∼16X over the classic Xdelta and Zdelta approaches while increasing the compression ratio by about 10%∼260%.

In the future, we plan to improve the batch compression technique in Gdelta to support local decompression and further study the trade-off between the delta encoding speed and compression ratio in Gdelta.

REFERENCES

[1] 2018. New generation entropy codecs : Finite State Entropy and Huff0. https://github.com/Cyan4973/FiniteStateEntropy

[2] Deepak R Bobbarjung, Suresh Jagannathan, and Cezary Dubnicki. 2006. Improving duplicate elimination in storage systems. *ACM Transactions on Storage (TOS)* 2, 4 (2006), 424–448.

[3] Andrei Z Broder. 1997. On the resemblance and containment of documents. In *Proceedings of Compression and Complexity of Sequences (SEQUENCES'97)*. IEEE, Washington, DC, USA, 21–29.

[4] Y Collet and M Kucherawy. 2018. Zstandard Compression and the application/zstd Media Type. *RFC 8478* (2018).

[5] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms, Third Edition*.

[6] Cai Deng, Qi Chen, Xiangyu Zou, Erci Xu, Bo Tang, and Wen Xia. 2022. imDedup: A Lossless Deduplication Scheme to Eliminate Fine-grained Redundancy among Images. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE, 1071–1084.

[7] Peter Deutsch and J-L Gailly. 1996. Zlib compressed data format specification version 3.3. *RFC Editor, http://tools.ietf.org/html/rfc1950* (1996).

[8] Idilio Drago, Marco Mellia, Maurizio M Munafo, et al. 2012. Inside dropbox: understanding personal cloud storage services. In *Proceedings of the 2012 ACM conference on Internet measurement conference (IMC'12)*. ACM Association, Boston, MA, USA, 481–494.

[9] Jarek Duda. 2013. Asymmetric numeral systems: entropy coding combining speed of Huffman coding with compression rate of arithmetic coding. *arXiv preprint arXiv:1311.2540* (2013).

[10] JL Gailly and M Adler. 1991. The GZIP compressor. http://www.gzip.org/.

[11] Diwaker Gupta, Sangmin Lee, Michael Vrable, et al. 2008. Difference engine: harnessing memory redundancy in virtual machines. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI'08)*. USENIX Association, San Diego, CA, USA, 309–322.

[12] David A Huffman. 1952. A method for the construction of minimum redundancy codes. *Proceedings of the IRE: Institute of Radio Engineers* 40, 9 (1952), 1098–1101.

[13] Lifang Lin, Yuhui Deng, Yi Zhou, and Yifeng Zhu. 2023. Inde: An inline data deduplication approach via adaptive detection of valid container utilization. *ACM Transactions on Storage (TOS)* 19, 1 (2023), 1–27.

[14] J. MacDonald. 2000. File system support for delta compression. Masters thesis. Department of Electrical Engineering and Computer Science, University of California at Berkeley.

[15] Athicha Muthitacharoen, Benjie Chen, and David Mazières. 2001. A Low-Bandwidth Network File System. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles (SOSP'01)* (Banff, Alberta, Canada). Association for

Computing Machinery, New York, NY, USA, 174–187. https://doi.org/10.1145/502034.502052

[16] Jisung Park, Jeonggyun Kim, Yeseong Kim, Sungjin Lee, and Onur Mutlu. 2022. DeepSketch: A new machine Learning-Based reference search technique for Post-Deduplication delta compression. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*. 247–264.

[17] S. Quinlan and S. Dorward. 2002. Venti: a new approach to archival storage. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST'02)*.

[18] David Reinsel, John Gantz, and John Rydning. 2018. The digitization of the world from edge to core. *IDC White Paper* (2018).

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

[20] Philip Shilane, Grant Wallace, Mark Huang, and Windsor Hsu. 2012. Delta compressed and deduplicated storage using stream-informed locality. In *Proceedings of USENIX Conference on Hot Topics in Storage and File Systems (HotStorage'12)*.

[21] Dimitre Trendafilov, Nasir Memon, and Torsten Suel. 2002. Zdelta: An efficient delta compression tool. *Technical report, Department of Computer and Information Science at Polytechnic University* (2002).

[22] Tony Wong, Smriti Thakkar, Kao-Feng Hsieh, Zachary Tom, Hetaben Saraiya, and Philip Shilane. 2023. Dataset Similarity Detection for Global Deduplication in the DD File System. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. IEEE, 3322–3335.

[23] Suzhen Wu, Zhanhong Tu, Yuxuan Zhou, Zuocheng Wang, Zhirong Shen, Wei Chen, Wei Wang, Weichun Wang, and Bo Mao. 2023. FASTSync: a FAST Delta Sync Scheme for Encrypted Cloud Storage in High-Bandwidth Network Environments. *ACM Transactions on Storage (TOS)* (2023).

[24] Wen Xia, Hong Jiang, Dan Feng, et al. 2011. Silo: a similarity-locality based near-exact deduplication scheme with low ram overhead and high throughput. In *Proc. USENIX ATC*.

[25] Wen Xia, Hong Jiang, Dan Feng, and Lei Tian. 2014. Combining Deduplication and Delta Compression to Achieve Low-Overhead Data Reduction on Backup Datasets. In *Proceedings of IEEE Data Compression Conference (DCC'14)*. IEEE Computer Society Press, Snowbird, Utah, USA, 203–212.

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

[27] Wen Xia, Chunguang Li, Hong Jiang, Dan Feng, Yu Hua, Leihua Qin, and Yucheng Zhang. 2015. Edelta: A Word-enlarging Based Fast Delta Compression Approach. In *the 7th USENIX conference on Hot Topics in Storage and File Systems (HotStorage'15)*. USENIX Association, Santa Clara, CA, 1–5.

[28] Wen Xia, Lifeng Pu, Xiangyu Zou, Philip Shilane, Shiyi Li, Haijun Zhang, and Xuan Wang. 2023. The Design of Fast and Lightweight Resemblance Detection for Efficient Post-Deduplication Delta Compression. *ACM Transactions on Storage (TOS)* 19, 3 (2023), 1–30.

[29] Wen Xia, Yukun Zhou, Hong Jiang, Dan Feng, Yu Hua, Yuchong Hu, Qing Liu, and Yucheng Zhang. 2016. FastCDC: A Fast and Efficient Content-Defined Chunking Approach for Data Deduplication. In *USENIX Annual Technical Conference (ATC'16)*. USENIX Association, Denver, CO, 101–114.

[30] Lianghong Xu, Andrew Pavlo, Sudipta Sengupta, and Gregory R. Ganger. 2017. Online Deduplication for Databases. In *Proceedings of ACM International Conference on Management of Data (SIGMOD'17)*. Chicago, IL, USA, 1355–1368.

[31] Lianghong Xu, Andrew Pavlo, Sudipta Sengupta, Jin Li, and Gregory R Ganger. 2015. Reducing replication bandwidth for distributed document databases. In *Proceedings of the Sixth ACM Symposium on Cloud Computing (SoCC'15)*. ACM Association, Big Island, Hawaii, USA, 222–235.

[32] Qing Yang and Jin Ren. 2011. I-CASH: Intelligently coupled array of SSD and HDD. In *Proceedings of the 17th IEEE International Symposium on High Performance Computer Architecture (HPCA'11)*. IEEE Computer Society Press, San Antonio, TX, USA, 278–289.

[33] Yucheng Zhang, Hong Jiang, Dan Feng, Nan Jiang, Taorong Qiu, and Wei Huang. 2023. LoopDelta: Embedding Locality-aware Opportunistic Delta Compression in Inline Deduplication for Highly Efficient Data Reduction. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. 133–148.

[34] Yucheng Zhang, Hong Jiang, Dan Feng, Wen Xia, Min Fu, Fangting Huang, and Yukun Zhou. 2015. AE: An asymmetric extremum content defined chunking algorithm for fast and bandwidth-efficient data deduplication. In *IEEE Conference on Computer Communications (INFOCOM'15)*. IEEE, 1337–1345.

[35] Yucheng Zhang, Wen Xia, Dan Feng, Hong Jiang, Yu Hua, and Qiang Wang. 2019. Finesse: fine-grained feature locality based fast resemblance detection for post-deduplication delta compression. In *Proceedings of USENIX Conference on File and Storage Technologies (FAST'19)*. 121–128.

[36] Zihao Zhang, Huiqi Hu, Zhihui Xue, Changcheng Chen, Yang Yu, Cuiyun Fu, Xuan Zhou, and Feifei Li. 2021. Slimstore: A cloud-based deduplication system for multi-version backups. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 1841–1846.

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

[38] Jacob Ziv and Abraham Lempel. 1977. A universal algorithm for sequential data compression. *IEEE Transactions on information theory* 23, 3 (1977), 337–343.

[39] Jacob Ziv and Abraham Lempel. 1978. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory* 24, 5 (1978), 530–536.

[40] Xiangyu Zou, Cai Deng, Wen Xia, Philip Shilane, Haoliang Tan, Haijun Zhang, and Xuan Wang. 2021. Odess: Speeding up resemblance detection for redundancy elimination by fast content-defined sampling. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 480–491.

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

[42] Xiangyu Zou, Jingsong Yuan, Philip Shilane, Wen Xia, Haijun Zhang, and Xuan Wang. 2021. The dilemma between deduplication and locality: Can both be achieved?. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. 171–185.

[43] Xiangyu Zou, Jingsong Yuan, Philip Shilane, Wen Xia, Haijun Zhang, and Xuan Wang. 2022. From Hyper-Dimensional Structures to Linear Structures: Maintaining Deduplicated Data's Locality. *ACM Transactions on Storage (TOS)* 18, 3 (2022), 1–28.

Dear editors and reviewers of ACM Transactions on Storage:

The submitted manuscript, titled "The Design of Fast Delta Encoding for Delta Compression Based Storage Systems," presents our latest research results that have not been published in or submitted to a journal. The preliminary results (i.e., Exploring the potential of fast delta encoding: Marching to a higher compression ratio.) of this research were published in the proceedings of the IEEE International Conference on Cluster Computing (CLUSTER'20, 11 pages). This new manuscript (27 pages) proposes a new approach (called Gdelta) that is provided with two new techniques: "fast skipping unmatched words" and "rolling with multiple bytes." These two techniques are based on our observations of inefficient operations in detecting redundancies and are designed to address these issues. With these two benefits, the computation complexity of the new approach is hugely reduced without sacrificing the compression ratio (compared to the old version). Evaluations suggest that the new approach could achieve a 70% ~ 90% higher encoding speed than the previous version with nearly the same compression ratio.

We list the significant improvements of the manuscript as follows:

1. **We solve the limitations of the previous research**: Delta encoding has many steps, such as splitting base/input chunks into several words, word hashing, building hash index for words, searching duplicated words between similar chunks, and encoding instructions. In the conference version, we only focused on the impact of rolling hashing and the data structure of hash indexing on the speed of delta encoding but ignored the calculation of searching repeated words between similar chunks. In the latest article, we consider the whole workflow and optimize the process of searching repeated words between similar chunks and building word indexes for base chunks.
2. **We propose two new techniques to improve the speed of delta encoding by 70% ~ 90%:** The first is the "fast skipping unmatched words mechanism", which enables Gdelta to pass through the area without duplicate data faster. The second is the technique called "rolling two bytes each time", which uses the loop unwinding to accelerate the Gear rolling hash. Our experiments show that the delta encoding speed of the latest version is improved by 70% ~ 90% compared with the conference version.
3. **We rerun the experiments with new techniques, new baselines, and new datasets:** Compared with the conference version, (1) We add experiment results to verify the new techniques. (2) We add experiment results that compare our method with the fastest encoding method (i.e., Ddelta and Edelta). (3) We add two new datasets, RDB (Backup of Redis database) and CHM (Source code of Chrome project), which are used in the related work of the deduplication storage system.

The detailed improvements are as follows:

1. We update the descriptions of existing methods with more details in Section 2.
   (1) We update Figure 1 to show the mechanism of existing delta encoding methods in more detail.
   (2) We add some descriptive and summary sentences to analyze existing delta encoding methods in more detail.

2. We add a new motivation in Section 1 and Section 3.
   (1) By testing the performance of existing delta encoding methods (see Figure 2) and analyzing their algorithm mechanism (see observation 2 in section 3), we conclude the reasons for the large performance differences of existing methods. We find that classical delta encoding methods introduce too many byte-by-byte operations. Based on this motivation, we propose a new method to reduce the performance gap of these methods, and our method will not cause a significant reduction in compression ratio.
3. We add descriptions of two new techniques in Section 4.
   (1) We add Section 4.4, which describes the technique of "rolling two bytes each time". We add Figure 6, Algorithm 1, and Algorithm 2 to illustrate the design of this technique.
   (2) We add Section 4.5, which describes the fast skipping unmatched words mechanism. We add Figure 7 and Equation 1 to illustrate the design of this technique.
4. We update all the experiments in Section 5 based on the new datasets.
   (1) In Section 5.4, we add the experiment that evaluates the performance of Gdelta with different batched sizes.
   (2) In Section 5.5, we analyze the performance improvement of "fast skipping unmatched words mechanism" on Gdelta.
   (3) In Section 5.6, we show the performance improvement of "rolling two bytes each time" on Gdelta.
   (4) In Section 5.6, we compare the performance of Gdelta with the fastest encoding methods, which shows that Gdelta has a high delta compression speed.
   (5) In Section 5.7, we compare the performance of the Gdelta with other existing delta encoding methods that have the highest compression ratio.
5. We also update the sections of the Abstract and Conclusion accordingly.

We sincerely hope that our manuscript satisfies the submission requirements. We will carefully improve the paper's quality based on your comments.

Thank you very much for your kind consideration.

Sincerely yours,
Haoliang Tan, Wen Xia, Xiangyu Zou, Cai Deng, Qing Liao, and Zhaoquan Gu

Exploring the Potential of Fast Delta Encoding: Marching to a Higher Compression Ratio

Haoliang Tan[1], Zhiyuan Zhang[1], Xiangyu Zou[1], Qing Liao[1,2], and Wen Xia[1,2]

[1] School of Computer Science and Technology, Harbin Institute of Technology, Shenzhen, China
[2] Cyberspace Security Research Center, Peng Cheng Laboratory, Shenzhen, China

***Abstract*—Delta compression (or called delta encoding) is a data reduction technique capable of calculating the differences (i.e., delta) among the very similar files and chunks, and is thus widely used for optimizing synchronization replication, backup/archival storage, cache compression, etc. However, delta compression is costly because of its time-consuming word-matching operations for delta calculation. Existing delta encoding approaches, are either at a slow encoding speed, such as Xdelta and Zdelta, or at a low compression ratio, such as Ddelta and Edelta. In this paper, we propose Gdelta, a fast delta encoding approach with a high compression ratio, that improves the delta encoding speed by employing an improved fast Gear-based rolling hash for scanning fine-grained words, and a quick array-based indexing scheme for word-matching, and then, after word-matching, further batch compressing the rest to improve the compression ratio. Our evaluation results driven by six real-world datasets suggest that Gdelta achieves encoding/decoding speedups of 2X∼4X over the classic Xdelta and Zdelta approaches while increasing the compression ratio by about 10%∼120%.**

***Index Terms*—Data reduction, compression, delta encoding**

I. Introduction

Nowadays, efficient storage of a huge volume of digital data becomes a big challenge for industry and academia, as evidenced in part by an estimated amount of about 33 zettabytes and will be 175 zettabytes of data produced in the world respectively in 2018 and 2025 according to IDC [15], which suggests the digital data will continue to grow exponentially. As a result, the space-efficient data reduction techniques for storage systems (especially for cluster storage) have been gaining increasing attention recently.

In general, there are mainly three technology roadmaps for lossless data reduction, including general compression [9], delta compression [12], [19], and data deduplication [13], [14]. General compression, which has been evolving for decades, is a popular technique for most users. General compression is designed for reducing redundancy data at the byte/string level by using traditional algorithms, e.g., dictionary coding [30], [31] and Huffman coding [11]. And because of the fine-grained redundancy searching, general compression usually achieves a good compression ratio. But it also has several weaknesses: For large-scale storage systems, on one hand, the compression/decompression speed of general compression is not quick enough for processing such amounts of data; And on the other hand, because of the size limit of compression window, general compression can not find out data redundancies which are far away from each other.

Because of the limitations of general compression in large-scale storage systems, data deduplication is proposed. Data deduplication, eliminating redundancy at the chunk level, always follows the idea that dividing data streams into several independent chunks, and then eliminates duplicate chunks. Data deduplication accelerates the processing (i.e., compressing) speed up to GB/s level, and makes it practical for large systems. But because the workflow of redundancy elimination is coarse-grained, it scarifies the compression ratio.

To bridge the gap in compression ratio (between data deduplication and general compression), delta compression is proposed, which focuses on removing redundancy among similar files or chunks. In data deduplication, the data streams are divided into coarse-grained chunks (e.g., size of 8KB), and there are many chunks which are not totally duplicate but very similar and compressible for delta compression. Therefore, introducing delta compression into data deduplication could significantly increase the compression ratio, and many previous works [7], [10], [16], [17], [25] follow this idea to improve several systems in different domains.

But compared with data deduplication, the delta encoding (i.e. calculate the differences between similar chunks or files) speed of delta compression is much slower, and becomes the main challenge in systems which introduces delta compression techniques. The classic delta compression approaches, such as Xdelta [12] and Zdelta [19], follow the idea of general compression, which includes characterization on sliding windows and finding matches on source (or base) chunks and target (or input) chunks, which is very time-consuming.

Although Ddelta [21] and Edelta [22] are proposed to significantly improve the delta encoding speed by using Content-Defined Chunking and Greedy-based word-matching techniques, both of them sacrifice the compression ratio. For these two approaches, we find that the decreased compression ratio is due to their coarse-grained word-matching techniques. When we look back at Xdelta and Zdelta, their high compression ratio and the slow speed results from their fine-grained word-matching techniques. Specifically, our empirical

978-1-7281-6677-3/20/$31.00 ©2020 IEEE
DOI 10.1109/CLUSTER49012.2020.00030

1) **The rolling hash bottleneck**. We observed Xdelta uses an expensive rolling hash called adler32, which occupied 44.7% of the CPU time according to our experimental studies using the CPU performance analysis tool 'Perf'. Here rolling hash is running in Xdelta that moves on the similar files/chunks byte-by-byte to generate words for maximizing the duplicate-word-matching.
2) **The further compression opportunity**. We observed Zdelta achieves the highest compression ratio (about 5-30% higher) among the existing delta compression approaches since it detects duplicate word at a much smaller granularity of 3 Bytes and then employs Huffman to further compress the rest of unmatched words after delta calculation.

Inspired by our previous work FastCDC [23] that employed a fast rolling hash called Gear to improve the content-defined chunking speed significantly, we believe that Gear hash can also help speed up the word-matching process in delta encoding. In addition, the recently proposed fast compression tools such as Zstd [4] and FSE encoding [1], also provide an opportunity to improve the delta compression efficiency while maintaining a high compressing speed.

Therefore, motivated by the above observations and discussions, we propose Gdelta, a fast delta encoding approach with a high compression ratio. Specifically, Gdelta improves the delta encoding speed by employing the improved Gear-based rolling hash for quickly scanning words and applying a fast array-based indexing scheme for word-matching; Gdelta improves the compression ratio by further carefully batch compressing the rest after word-matching. Our evaluation results suggest that Gdelta achieves encoding/decoding speedup of 2X∼4X over the classic Xdelta and Zdelta approaches while increasing the compression ratio by about 10%∼120%.

The rest of this paper is organized as follows. Section II introduces the background and related work. In Section III, we discuss our important observations on the classic Xdelta and Zdelta approaches, which motivate this work. Section IV describes the design and implementation details of Gdelta. Section V presents the evaluation results of Gdelta, including comparisons with the known delta compression approaches, Xdelta and Zdelta. Finally, Section VI concludes this paper and outlines future work.

II. Background and Related Work

As a space-efficient technique for eliminating redundancy among the very similar files or chunks, delta compression has gained increasing attention recently by effectively and efficiently reducing both the network and storage resource requirements in large-scale storage systems [2], [3], [7], [10], [16]–[18], [25]. In this section, we will first compare delta compression with other data reduction techniques, and then introduce the latest work on delta compression.

There are many kinds of redundant data in storage systems, thus many data reduction techniques are proposed to process these redundant data respectively. Table I summarizes three typical data reduction techniques, also as detailed below:

TABLE I
A general comparison of three typical data reduction techniques: general compression, delta compression, and data deduplication. 'Appr. Dates' denotes the approximate dates of initial research for each technique.

	General compression	Delta compression	Data deduplication
Objects	All data	Similar data	Duplicate data
Granularity	String-/Byte-level	String-/Byte-level	Chunk-level
Rep. Techniques	Huffman coding [11]/ Dictionary coding	Copy/Insert-based Delta encoding	CDC & Secure signature
Appr. Dates	1970s	1990s	2000s
Rep. Prototypes	GZIP [9], Zlib [6], Zstd [4]	Xdelta [12], Zdelta [19], Ddelta [21]	Venti [14], LBFS , DDFS [29]

- **General compression** (i.e., traditional lossless compression) methods, such as GZIP [9], focus on the internal compression of files or data chunks, and eliminate byte- and bit-level redundant data by using Dictionary coding (e.g., LZ77 [30], LZ78 [31]) and Huffman coding [11], respectively. General compression methods are usually very time-consuming because of their fine-grained redundancy eliminating schemes, and usually only compress data in a small region (e.g., a single file or a chunk).
- **Delta compression** focuses on eliminating redundancy among similar files/chunks. Generally, it (e.g., Xdelta [12]) uses the Rabin-Karp-based string matching scheme to find repeated strings, and then encodes matched (duplicate) strings with "Copy" instructions and unmatched strings with "Insert" instructions. In addition, Zdelta [19] further compresses unmatched strings at the byte-level by using Huffman coding [11].
- **Data deduplication**, aiming at quickly eliminating redundancy at a coarse granularity, is a chunk-level compression method for large-scale storage systems. Generally, it divides files into chunks of approximately equal length by Content-Defined Chunking (CDC) [13], and then the secure signature (or called fingerprint, e.g., SHA1) of each chunk is calculated for duplicate-matching. Two chunks will be considered duplicate if they own the same secure signature. The advantage of this method is simple, fast, and easy to be implemented in large-scale storage systems. But the disadvantage is that there is still a lot of redundant data between the very similar but non-duplicate chunks/files, which can be solved by delta compression.

As discussed above, compared with general compression and data deduplication, delta compression is able to eliminate redundancy among the very similar files/chunks, and it has been gaining increasing attention in recent years. Researchers in EMC [17] implement delta compression on top of deduplication to further eliminate redundancy to accelerate the WAN replication of backup datasets, which suggests delta compression can help achieve about 2X higher compression ratio than the traditional ways. Yang et al. [26] employ the delta compression technique to eliminate redundancy among similar data blocks in SSDs and thus enlarges the logical space of SSD

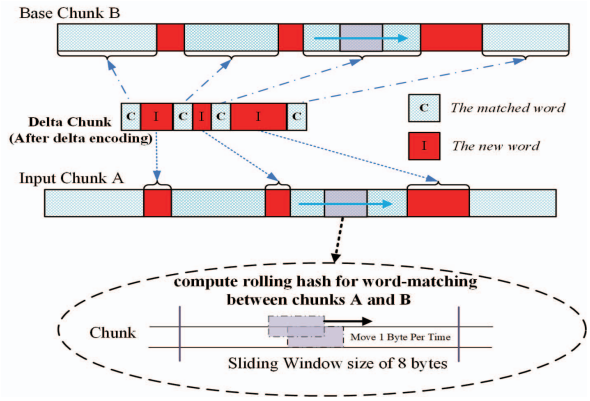


Fig. 1. A general example of delta encoding process in Xdelta. 'C' and 'I' denote the 'Copy' and 'Insert' instructions in delta encoding, respectively.

caches. Researchers in Microsoft [24], [25] propose using delta compression techniques for similarity-based deduplication to improve the storage and network efficiency of on-line database management systems, which suggests delta compression can achieve 2X-5X higher compression ratio than the traditional deduplication-base approaches.

Generally, delta compression consists of two key stages: resemblance detection and delta encoding. **Resemblance detection** refers to the process of generating features and then searching similar chunks according to their features, which has been studied in many works [16], [20]. **Delta encoding** refers to the process of calculating the delta among the very similar chunks/files (after resemblance detection), which determines the final compression ratio of delta compression. Delta encoding is time-consuming, usually a bottleneck of the delta compression-based systems, which thus has been gaining increasing attention recently [24], [25].

For delta encoding, Figure 1 shows a general example of Xdelta [12] on two similar chunks, i.e., input (new) chunk A and its similar (base) chunk B. Xdelta first scans the two chunks to generate several overlapped words (e.g., strings with the length of 8 bytes) using a rolling hash called Adler32, and then finds the duplicate words by indexing the Adler32 hashes of the words. Therefore, Xdelta uses the words in the base chunk B as "dictionary" and indexes the words of chunk A in the "dictionary" to identify redundancy. Then, the matched words will be encoded as "Copy" instructions with offset and length messages and the new (unmatched) words are encoded as "Insert" instructions with the actual words (data). Note that adjacent matched or unmatched words will be merged into one bigger instruction, respectively (to minimize the instructions overhead). Finally, the input chunk is compressed into a smaller delta chunk $\Delta_{(A,B)}$ with "Copy" and "Insert" instructions for space savings. With the delta chunk $\Delta_{(A,B)}$ and the base chunk B, Xdelta can easily restore the input chunk A according to the 'Copy' and 'Insert' instructions.

There also many other delta encoding schemes that are different from Xdelta. We summarize four state-of-the-art delta encoding approaches (including Xdelta [12], Zdelta [19], Ddelta [21], and Edelta [22]) with their features as shown in Table II. Generally, to maximize the duplicate-word matching, Xdelta and Zdelta use Fix-Sized Chunking (FSC) to generate fix-sized and overlapped words for matching, and also detect the duplicate words inside the input chunk. On the other hand, Ddelta/Edelta use Content-Defined Chunking (CDC) to generate variable-sized and independent (non-overlapped) words for simplifying matching, which results in high encoding speed. Moreover, Zdelta uses a hash table to detect the best-matched words and employs entropy encoding (i.e., Huffman coding) to further compress the rest after word-matching, thus achieving the highest compression ratio.

TABLE II
COMPARISON ON DIFFERENT DELTA ENCODING APPROACHES. HERE INTER MATCH AND INNER MATCH MEAN DETECT THE DUPLICATE WORDS BETWEEN THE BASE & TARGET CHUNKS AND INSIDE THE TARGET CHUNK, RESPECTIVELY.

Approa.	Chunking	Hashing	Indexing	Inter Match	Inner[1] Match	Entropy Encoding
Xdelta	FSC[2]	Adler32	Array	✓	✓	×
Zdelta	FSC	Adler32	Hash table[3]	✓	✓	Huffman
Ddelta/ Edelta	CDC[4]	Spooky/ xxHash	Hash table	✓	×	×
Gdelta	FSC	Gear	Array	✓	✓	FSE

1 Inter match means detecting redundancy (duplicate words) between similar chunks while inner match means detecting redundancy inside the input chunk after inter match.
2 FSC means using a byte-wise sliding window to roll on the chunks' contents to generate fix-sized and overlapped words for maximizing duplicate-word matching.
3 Hash table used in Zdelta is to detect the best match when indexing.
4 CDC means dividing the chunks into several variable-sized and non-overlapped words for simplifying the duplicate-word matching.

In this paper, we focus on designing a high-speed delta encoding approach (called Gdelta) with a high compression ratio, i.e., accelerating the delta calculation process of the already-detected delta-compression candidates, which is very different from our previous work Ddelta/Edelta [21], [22]. In Table II, we also list the features of our Gdelta, which is generally the same as Xdelta and Zdelta for a high compression ratio. But we significantly change the hashing, indexing, and entropy encoding schemes in Gdelta, as explained later in the next sections according to our observations of running the four state-of-the-art delta encoding approaches with six real-world datasets. *The main contribution of Gdelta is to explore the potential of fast delta encoding for a much higher compression ratio than state-of-the-art approaches.*

III. MOTIVATION AND OBSERVATIONS

As mentioned in Section II, existing delta encoding approaches target either delta compression ratio (e.g., Xdelta [12] and Zdelta [19]) or delta encoding speed (e.g., Ddelta [21] and and Edelta [22]). Figure 2 shows the delta compression performance of the four state-of-the-art approaches on six real-world datasets (whose characteristics are detailed in Table IV in Section IV). Here compression ratio is measured by $\frac{\textit{Total size of input chunks}}{\textit{Total size of delta chunks}}$ (for the already-detected similar chunks). The results suggest that Ddelta and Edelta represent the high-

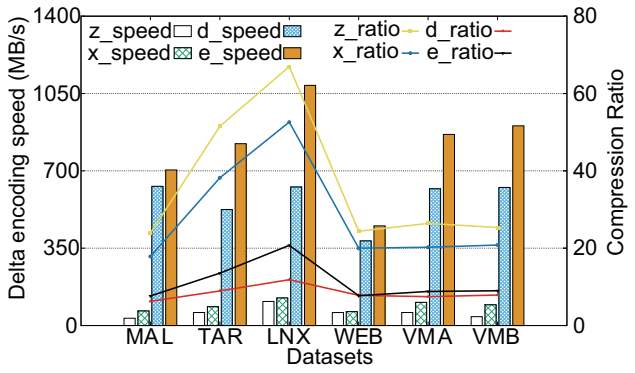


Fig. 2. Delta encoding speed and compression ratio of Xdelta, Zdelta, Ddelta, and Edelta on six datasets.

speed delta compression approaches but at a low compression ratio while Xdelta and Zdelta represent the classic delta compression approaches that focus on the compression ratio. For Ddelta and Edelta, we find the decreased compression ratio is due to their coarse-grained word-matching techniques. When we look back at Xdelta and Zdelta, their high compression ratio (about 2X-6X higher over Ddelta and Edelta) and the slow speed are from their fine-grained word-matching techniques, which provide two important observations for us as discussed below.

Observation ①: **The rolling hash bottleneck**. We find that Xdelta uses a time-consuming rolling hash called Adler32, which occupies 44.7% of the CPU time according to our experimental studies (running on six datasets, also detailed in Section V) using the CPU performance analysis tool 'Perf'. Here the rolling hash is used as the sliding window technique that moves on the similar chunks (files) byte-by-byte to generate words for maximizing the duplicate-word-matching in Xdelta. More concretely, Xdelta uses the sliding window size of 9 bytes to detect duplicate words between the base and input chunks and the sliding window size of 4 bytes to detect duplicate words inside the input chunks, in order to detect redundancy as much as possible.

Observation ②: **The further compression opportunity**. We observe that Zdelta achieves the highest compression ratio (about 5-30% higher) among the existing delta compression approaches. This is because Zdelta uses a much smaller sliding window size of 3 bytes to detect the fine-grained redundancy for delta calculation among similar chunks (and also inside the input chunk), and then employs Huffman encoding technique to further compress the remaining unmatched words (to increase the compression ratio).

For the problem in observation ①, inspired by our previous work FastCDC [23] and Ddelta [21] that employs a fast rolling hash called Gear to greatly improve the Content-Defined Chunking speed for data deduplication, we believe that Gear hash can also help speed up the word-matching process in delta encoding. To the best of our knowledge, Gear appears to be one of the fastest rolling hash algorithms for CDC at present

TABLE III
THE COMPARISON OF TWO ROLLING HASHES: ADLER32 AND GEAR. HERE 'a', 'b', AND 'N' DENOTE CONTENTS OF THE FIRST BYTE, THE LAST BYTE, AND THE SIZE OF THE SLIDING WINDOW RESPECTIVELY, AND 'GEAR' [21] DENOTE THE PREDEFINED ARRAYS, AND 'fp' REPRESENTS THE FINGERPRINT OF THE SLIDING WINDOW.

Name	Pseudocode	Speed
Adler32	$Low \mathrel{+}= b - a;$ $High \mathrel{+}= Low - a * N;$ $fp = (High << 16) \mid Low$	Slow
Gear	$fp = ((fp << 1) + Gear(b))$	Fast

since it uses much fewer calculation operations than others. For the problem in Observation ②, the recently proposed fast compression tools such as Zstd [4] and FSE encoding [1], also provide opportunities to improve the delta compression ratio without sacrificing the fast compressing speed.

Therefore, motivated by the two critical observations, we propose Gdelta in this paper using Gear hash to march to a higher compression ratio over the classic Xdelta and Zdelta approaches while maintaining a fast encoding speed.

IV. Design and Implementation

A. Gdelta Overview

Gdelta aims to provide high delta encoding performance for compressing very similar chunks and files, including the high compression ratio and fast encoding/decoding speed. To achieve that target, Gdelta first employs an improved Gear rolling hash for generating overlapped words (detailed in subsection IV-B), applies a fast array-based indexing scheme for word-matching (detailed in subsection IV-C), and then, after word-matching, uses the compacted encoding instructions to record the delta chunk (detailed in subsection IV-D) and further compresses the rest using Zstd and FSE to improve the compression ratio (detailed in subsection IV-E).

B. Gear-based Rolling Hash

As discussed in Section III, Gear hash has the potential to replace the Adler32 used for generating overlapped words to maximize the duplicate-words matching. Thus we first introduce Gear hash and then discuss our design of the improved Gear-based rolling hash used in Gdelta.

As shown in Table III, Gear rolling hash is running in two key ways: (1) It employs an array $Gear[]$ of 256 random 64-bit integers to map the values of the byte contents in the sliding window; and (2) The addition ("+") operation adds the new byte in the sliding window into Gear hashes while the left-shift ("<<") operation helps strip away the first byte of the last sliding window. This is because, after the '<<' and modulo operations, the first byte a will be calculated into the fp as $(Gear[a] << N)\ mod\ 2^N$, which will be equal to zero. Note that the '$mod\ 2^N$' operations can be eliminated if we set the fp as a 32- or 64-bits integer while using the sliding window size of 32 or 64 bytes accordingly. As a result,

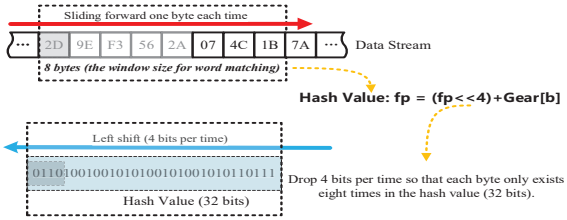


Fig. 3. A general example of Gear-based rolling hash in Gdelta.

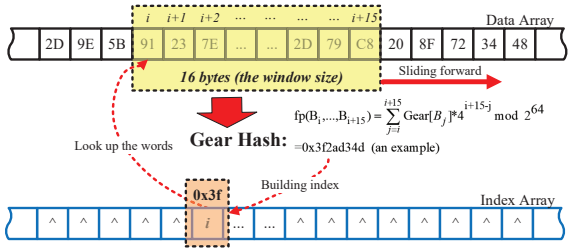


Fig. 4. A general example of the array based indexing scheme combining with Gear-based rolling hash in Gdelta.

Gear rolls on the content using only three operations each byte (i.e., "+", "<<", and an array lookup), enabling it to move quickly through the data content for rolling hash function. It is also worth noting that in Content-Defined Chunking [23], the sliding window size can be adjusted by using a mask value for the hash judgment during chunking, which is very different from the application scenario of this paper.

Table III also shows the detailed implementation of Adler32 for comparison, which suggests Gear uses much less calculation operations than Adler32, thus being a good rolling hash candidate for Gdelta. Note that the sliding window size in Gear needs to be slightly adjusted to satisfy the fast-rolling hash requirements in Gdelta. More concretely, if we use the sliding window size of 8 bytes similar to Xdelta, we can implement Gear as $fp = ((fp << 1) + Gear(b)) \ mod \ 2^8$. Also, we can implement Gear as $fp = ((fp << 8) + Gear(b))$ for rolling while *fp* is a 64-bits integer, which eliminates the '$mod\ 2^8$' operation and thus further speeds up the rolling hashing process. Thus, the second implementation is more reasonable and is finally used in the remainder of this paper.

Figure 3 provides a detailed example of calculating the Gear hash in Gdelta while using the sliding window for word-matching in delta compression. The number of left-shift bits N can be calculated from the hash value bits V and the sliding window size s: $N = V/s$, which lets the hash value fp exactly represents the contents of the sliding window (used for word-matching) in delta encoding. More important, our evaluation results suggest that: By using the sliding window size of 16 bytes (Gear rolling hash is improved as: $fp = ((fp << 4) + Gear(b))$ and *fp* is a 64-bits integer) and combining other further compression techniques, Gear can obtain a higher compression ratio over Zdelta and Xdelta while achieving a much faster encoding speed (i.e., accelerate the duplicate word-matching process).

Note that Gear hash has also been used in many papers [21], [23] for Content-Defined Chunking, which targets at quickly getting the chunking positions (i.e., to split files into several small and independent data chunks) in data deduplication-based systems. In our paper, we use Gear hash as a rolling hash for word-matching in delta encoding, to replace the time-consuming Adler32 in Xdelta. And we slightly change the implementation of Gear (i.e., left-shifting more bits) to control its sliding window size to satisfy the requirement of the minimum word size used for matching in delta encoding.

C. Array-based Indexing Scheme

After generating overlapped words by Gear, Gdelta uses a simple array-based indexing scheme (similar to Zstd [4] and Xdelta [12]) to look up the duplicate words according to their Gear hash. Take 16-bytes words (i.e., the sliding window of 16 bytes) as an example to illustrate our indexing scheme in Figure 4, After running Gear hash, and we get a 64-bits fingerprint for each word by computing $fp = ((fp << 4) + Gear(b))$. Thus, *fp*'s 16 most-significant bits are calculated from all the sixteen bytes in the sliding window, while the 16 least-significant bits are just calculated from the latest four bytes in the sliding window (since the last 12 bytes have been moved away here by using the "<< 4" operation many times). Therefore, we use the most-significant bits of fingerprint as the index of a hash array by the right-shift operation, which can effectively reduce the hash collision. Moreover, the number $maskBts$ of the most-significant bits used for the index can be minimized as $\lceil log(Chunk.Length) \rceil$ (e.g., $maskBts$ = 16 bits when the length of the base chunk is 64KB), which can also be used as the minimal size of the elements in the indexing array in Gdelta for memory space saving.

According to our observation, the traditional hash table's memory cost is about 2-3 times more than this array-based index since the hash table needs to store additional data points and the offset information for each indexing unit [5]. Meanwhile, our performance evaluation of Gdelta suggests that comparing with hash tables, this simple array-based indexing scheme achieves a comparable compression ratio and a fast looking up speed, as discussed later in Section V.

D. Encoding and Decoding

Algorithms 1 and 2 provide the detailed pseudo code of the Gdelta encoding and decoding processes, also including the rolling hash computation (using the sliding window size of 16 bytes and 64-bits *fp*) and duplicate-words matching. Gdelta also uses the "Copy" and "Insert" instructions to record the duplicate and non-duplicate words of the input chunk, which is similar to Xdelta and Zstd (see Figure 1). Thus, for Gdelta encoding, the delta chunk is recorded by the "Copy" and "Insert" instructions in turn. Note that for the duplicate-words matching, the greed-based matching used in Edelta (or called 'word-enlarging') is also applied in Gdelta, which can

Algorithm 1: Gdelta Encoding

Input: Base data, bas[]; Input data, ipt[];
Output: Delta data, dlt[];
$dlt.Len \leftarrow 0$;
$anchor \leftarrow 0$;
$wordSze \leftarrow 16$;
$moveBts \leftarrow 64/wordSze$;
$arryInx \leftarrow empty$;
$maskBts \leftarrow \lceil log(bas.Len) \rceil$;
$fp \leftarrow 0$;
for $i = 0; i < wordSze; i$ ++; **do**
  $fp \leftarrow (fp \ll moveBts) + Gear[bas[i]]$;
for $; i + wordSze < bas.Len; i$ ++; **do**
  $fp \leftarrow (fp \ll moveBts) + Gear[bas[i]]$;
  $Inx \leftarrow fp \gg (64 - maskBts)$;
  $arryInx[Inx] = i$; //Build index for the Base words;
$fp \leftarrow 0$;
for $j = 0; j < wordSze; j$ ++; **do**
  $fp \leftarrow (fp \ll moveBts) + Gear[ipt[i]]$;
for $; j + wordSze < ipt.Len; j$ ++; **do**
  $Inx \leftarrow fp \gg (64 - mask)$;
  $ofst \leftarrow arryInx[Inx]$; //Look up the input words;
  for $mLen \leftarrow 0; ; mLen$ ++; **do**
    if *bas[ofst + mLen] != ipt[j - wordSze + mLen]* **then**
      $break$;
  if $mLen \geq wordSze$ **then**
    $instIns.Len \leftarrow j - anchor$; //Insert instruction;
    $copyIns \leftarrow (mLen, ofst)$; //Copy instruction;
    $memcpy(dlt + dlt.Len, instIns, size(instIns))$;
    $dlt.Len \leftarrow dlt.Len + size(instIns))$;
    $memcpy(dlt+dlt.Len, ipt+anchor, instIns.Len)$;
    $dlt.Len \leftarrow dlt.Len + instIns.Len$;
    $memcpy(dlt + dlt.Len, copyIns, size(copyIns))$;
    $dlt.Len \leftarrow dlt.Len + size(copyIns)$;
    $anchor \leftarrow j + mLen$;
    $j \leftarrow j + mLen - wordSze$;
    for $; j < anchor; j$ ++; **do**
      $fp \leftarrow (fp \ll movebts) + Gear[ipt[j]]$;
  else
    $fp \leftarrow (fp \ll movebts) + Gear[ipt[j]]$;
return $dlt.Len$;

Algorithm 2: Gdelta Decoding

Input: Delta data, dlt[], Base data, bas[]
Output: Decompressed data, ipt[]
$readLen \leftarrow 0$;
$ipt.Len \leftarrow 0$;
while $readLen < dlt.Len$ **do**
  $Ins \leftarrow read(dlt + readLen, size(Ins))$;
  $readLen \leftarrow readLen + size(Ins)$;
  if $Ins == insertIns$ **then**
    $memcpy(ipt + ipt.Len, dlt + readLen, Ins.Len)$;
    $readLen \leftarrow readLen + Ins.Len$;
    $ipt.Len \leftarrow ipt.Len + Ins.Len$;
  else if $Ins == copyIns$ **then**
    $memcpy(ipt + ipt.Len, bas + Ins.ofst, Ins.Len)$;
    $ipt.Len \leftarrow ipt.Len + Ins.Len$;
return $ipt.Len$;

further reduce the rolling hash computation and duplicate-words matching operations (just by directly matching forward to detect the duplicate words as long as possible). For Gdelta decoding, the input chunk can be easily restored according to the "Copy" and "Insert" instructions in the delta chunk and also the base chunk, which is the same as Xdelta and Ddelta.

E. Container-based Further Compression

As mentioned in Section III, we will further compress the rest after Gdelta encoding given above in Algorithm 1. The two recently proposed compression approaches are employed here, Zstd [4] and FSE encoding [1]. Finite State Entropy (FSE) [1] is a fast encoding algorithm based on the recent proposed Asymmetric Numeral Systems (ANS) theory [8], which can represent a symbol with fractional part statistically (i.e., smaller than one bit), which is more space-efficient than Huffman encoding for redundancy elimination at the bit-level. Zstandard is also a fast general compression approach that eliminates redundancy at both the word-level (default size of 8 bytes) and bit-level, recently proposed by Facebook.

To improve the compression efficiency in Gdelta, we compress the section of "Copy" and "Insert" instructions at the bit-level using FSE and compress the data part in the "Insert" instructions at both the word- and bit-level using Zstd. Moreover, to maximize the compression ratio of Gdelta, we put several delta chunks altogether in a container with a size of 2MB or 4MB. This is because the container-based batch processing efficiently reduces the metadata overheads for data compression, such as the indexing arrays, dictionaries, etc., used in FSE and Zstd for each delta chunk. Note that the data section and the instructions section in a container are batch-compressed individually and then stored separately, because there is almost no redundancy between them, and putting them all together for batch compression will increase the compression metadata overhead.

V. Evaluation

A. Experimental Setup

To evaluate delta encoding performance, we implement a prototype of the delta compression system on the Ubuntu 18.04.1 operating system running on an Intel Xeon(R) Gold 6130 processor at 2.1GHz, with 128GB of memory. Two known delta encoding approaches with the high delta compression ratio, Xdelta [12] and Zdelta [19] (we use the latest Xdelta 3.1 and Zdelta 2.1), are used with their default configurations as the baselines for evaluating Gdelta. On the other hand, Gdelta is configured using the word size of 16 bytes (the smallest matching unit) and Zstd compression-level of 10. In addition, the delta compression prototype consists of approximately 6, 000 lines of C code, which is compiled by GCC 7.5.0 with the "-O3" compiler option to maximize the speed of the resulting executable. Note that Edelta [22] and Ddelta [21] have a much lower compression ratio than other

approaches (see Figure 2 in Section III), we do not compare Gdelta with them in this section.

Three main metrics for delta compression are used for this evaluation, namely, *Compression ratio (CR)* measured in terms of $\frac{\textit{Total size of input chunks}}{\textit{Total size of delta chunks}}$ by running a given delta compression scheme on the detected similar chunks, *encoding speed* and *decoding speed* recorded by the response time of delta calculation and restore respectively in memory by a given evaluated delta encoding scheme. Note that for the *encoding* and *decoding speed*, we run each experiment five times to get the stable and the average results.

The datasets used in the evaluation are generated from six real-world workloads in various use cases, as shown in Table IV. These datasets are typical workloads for delta compression and have been used in many delta compression or data deduplication related papers [21]–[23], [27], [28]. Four of these datasets TAR[5], LNX[6], WEB[7], and VMA[8] are publicly available. To get similar chunks for delta encoding, we first apply Content-Defined Chunk (CDC) on these datasets to generate data chunks, and detect similar chunks with a super-feature based resemble detection method [17]. The CDC method used here is FastCDC [23], and the minimum chunk size, average chunk size, and maximum chunk size are configured as 2KB, 8KB, and 64KB, respectively. The specific resemble detection method is to compute 12 features from each chunk, and grouping them into three Super-Features. Two chunks are considered to be similar if they share one Super-Feature or more [17], [28].

To better evaluate Gdelta, we implement three versions of Gdelta with different configurations of compression techniques (with the default array-based indexing) as introduced below:

- Gdelta_A: only detect duplicate words between similar chunks and use Gear hash with the default sliding window size of 16 bytes. Thus, Gdelta_A reflects Gdelta only uses inter match (without inner match), as shown in Algorithm 1 in Section IV-D.
- Gdelta_B: further compress each delta chunk generated from Gdelta_A, i.e., uses FSE and Zstd to compress instructions and unmatched words, respectively.
- Gdelta_C: After Gdelta_A, put several delta chunks into a container (with the default size of 4MB), and then further compress the container for a much higher compression ratio by using FSE and Zstd. Thus, Gdelta_C reflects Gdelta using all the techniques listed in Section IV.

Therefore, based on the above experimental setup, we would like to answer the following questions in our evaluation:

- *How well does the Gear hash used for delta encoding in Gdelta compared with Adler32*? We answer this question in subsection V-B and study the hashing speed, hash collision ratio of Gear-based rolling hash, and its effect on the delta encoding performance.

[5]https://ftp.gnu.org
[6]https://www.kernel.org
[7]https://news.sina.com.cn
[8]http://www.thoughtpolice.co.uk

TABLE IV
WORKLOAD CHARACTERISTICS OF THE SIX DATASETS USED IN THE PERFORMANCE EVALUATION.

Name	Size	Workload descriptions
MAL	15 GB	a backup of E-Mails in personal use
TAR	54 GB	215 tarred files from several open source projects such as GCC, GDB, Emacs, etc.
LNX	116 GB	260 versions of Linux kernel source code. Each version is packaged as a tar file
WEB	278 GB	120 days of snapshots of the website: news.sina.com.cn
VMA	138 GB	90 virtual machine images of different OS release versions, including Fedora, CentOS, Debian, etc.
VMB	278 GB	18 backups of an Ubuntu 12.04 VM image

TABLE V
HASHING SPEED AND COLLISION RATIO COMPARISON OF GEAR AND ADLER32, WITH DIFFERENT SLIDING WINDOW SIZE: 8B, 12B, AND 16B.

Approaches	Thpt(MB/s)	Collision Ratio		
		10MB	20MB	50MB
Adel32(8B)	478.3	2.4E-10	2.5E-10	2.3E-10
Adel32(12B)	473.2	2.4E-10	2.5E-10	2.3E-10
Adel32(16B)	478.1	2.3E-10	2.2E-10	2.2E-10
Gear(+, 8B)	1738.8	2.4E-10	2.5E-10	2.4E-10
Gear(+,12B)	1741.5	2.4E-10	2.5E-10	2.4E-10
Gear(+,16B)	1741.3	2.3E-10	2.3E-10	2.3E-10

- *What is the compression ratio difference between the array-based and hash table-based indexing schemes in delta encoding*? We answer this question in subsection V-C by evaluating the metrics of delta encoding speed and compression ratio.
- *How effective is the container-based further compression scheme in Gdelta*? We answer this in subsection V-D by evaluating Gdelta performance, enabling and disabling the container-based further compression.
- *What is the final performance of Gdelta compared with the classic Xdelta and Zdelta*? We answer this in subsection V-E by studying the metrics of the encoding/decoding speed, and the compression ratio by running these three approaches on the six datasets.

B. Gdelta Using Gear for Hashing

Hashing speed and collision ratio are the two most important metrics for evaluating the effectiveness of weak hash Gear and Adler32. In this subsection, to evaluate the hash collision, we generate five test files consisting of random numbers for the given size of 10MB, 20MB, and 50MB (totaling 15 files). As introduced in Section IV-B, Gear hash used in Gdelta has been slightly changed by left-shifting several bits instead of one bit in the original Gear [21], which is due to our control of the small sliding window size (usually use size of 16 bytes in Gdelta) used for word-matching in delta encoding.

Table V shows the averaged hashing speeds and collision ratios of Adler32 and Gear. The evaluation is to calculate the hashes of words with different sizes of the sliding window (i.e., 8, 12, 16 bytes) on the random number workloads, which is often used for evaluating hash efficiency. Accurately, the metric collision ratio listed in Table V is calculated

according to Equation 1 as introduced below, which is the total combination of two unique words with identical weak hash fingerprint divided by the combination of picking any two words from all the words in the test files.

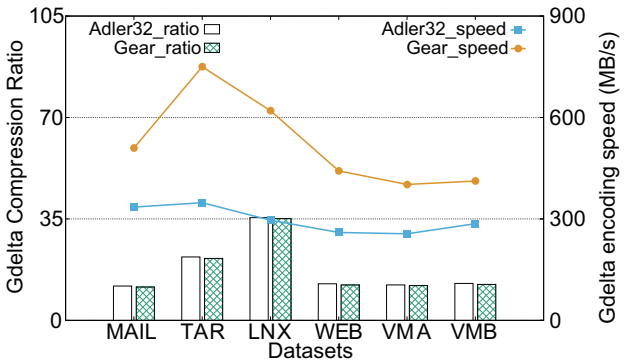
$$P_{collision} = \frac{\sum_{i=2}^{n} n_i C_i^2}{C_n^2} \quad (1)$$

Here i represents the number of collision occurrences for a given weak hash while n_i corresponds to the number of such i-times colliding weak hash fingerprints (e.g., if ten different words are sharing the same weak hash and there are seven such weak hash fingerprints, then i equals to 10 and n_i equals to 7). The results exactly prove Gear achieves nearly the same hashing efficiency as Adler32. Essentially speaking, picking two colliding words is equivalent to finding a new word that has the same weak hash fingerprint as the given one. Therefore, the probability is supposed to be $1/2^{64}$ (assuming that the hash value is a 64-bits integer) if the output of the hash function is uniformly distributed, which is consistent with our experiments. For example, for a target chunk, say 8KB (∼8000 words in total and according to table V, colliding probability is 2.5×10^{-10} at maximum), the maximum colliding word number is calculated as only 0.016 (according to Equation 1, we can get the biggest colliding-word number when $n_i = 0$ if $i \neq 2$) when using Gear to hash the words for indexing, indicating a quite low possibility for collision occurrence.

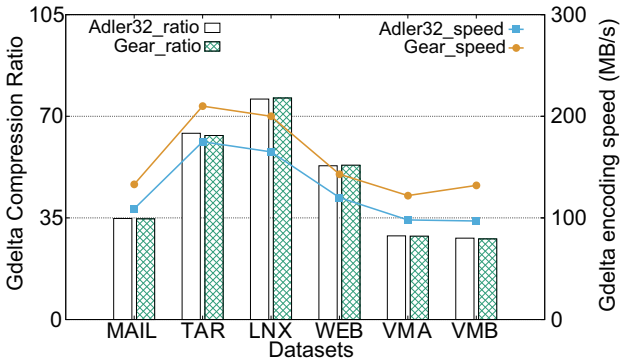
Table V also suggests that under the different file sizes, the hash speeds of Gear reaches 1740 MB/s, about 3.5× of Adler32. The fast speed of Gear is due to that Gear consumes much fewer instructions for the rolling hash comparing with Adler32 [23]. For the collision ratio issue, we can observe that Gear and Adler32 have almost identical hash collision ratios, while the sliding window size has almost no influence on the collision ratio. It is worth noting that the hash collision ratio is acceptably low since our delta encoding candidates are usually smaller than 64KB.

To further study the hashing effect in delta encoding, we evaluate the Gdelta performance when using Gear to replace Adler32, as shown in Figure 5. Here we configure Gdelta with array-based indexing and with/without container-based further compression. Specifically, Gdelta evaluated in Figure 5(a) only detects duplicate words among similar chunks, while Figure 5(b) configures Gdelta using container-based further compression. Generally, we can conclude from Figure 5 as:

- For the *compression ratio*, the two hash approaches achieves nearly the same compression ratio as discussed earlier in this subsection: the hash collision ratios of Gear and Adler32 are all very low. Note that Gdelta_C achieves a much higher compression ratio than Gdelta_A by using container-based further compression, which will be explained and discussed in Section V-D.
- For the *delta encoding speed*, when only detecting the duplicate words among similar chunks, Gear achieves 1.5X-2.4X higher delta encoding speed over Adler32. while adding the technique of contain-based further compression into the delta encoding process, Gear achieves 20% higher delta encoding speed over Adler32.



(a) Gdelta_A (with only inter match)



(b) Gdelta_C (with both inter and inner match)

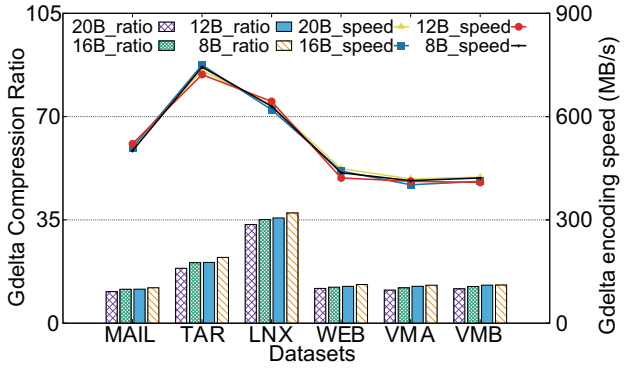
Fig. 5. Compression ratio and encoding speed of Gdelta (Gdelta_A and Gdelta_B) using Gear hash and Adler32 for fingerprinting, respectively.

Figure 6 also studies how the sliding window size of Gear hash (i.e., the minimum unit used for word-matching) impacts the delta encoding speed and compression ratio in both Gdelta_A and Gdelta_C. Generally, smaller sliding window size (i.e., 8 bytes) results in a higher compression ratio when Gdelta_A only removes redundancy among the similar chunks, as shown in Figure 6(a). But Figure 6(b) suggests Gdelta_C (with container-based further compression) achieves a higher compression ratio when using the sliding window size of 16 bytes, which is because we observed that a larger sliding window size provides more opportunities for our further compression after word-matching among similar chunks.

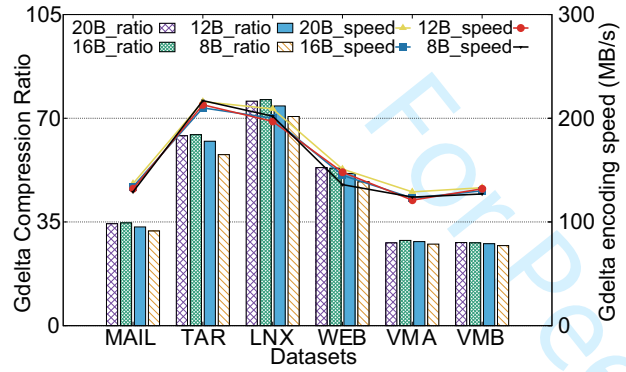
In summary, Gear hash is shown to be 3.5X faster than Adler32 while keeping a very low hash collision ratio. It also achieves a very high compression ratio using the sliding window size of 16 bytes in Gdelta while combining with the technique of container-based further Zstd compression. Therefore, Gear is shown to be an excellent rolling hash function used for delta encoding in Gdelta.

C. Gdelta using Array-based Indexing

In this subsection, we evaluate the Gdelta performance when using the technique of array-based indexing (i.e., quickly



(a) Gdelta_A (with only inter match)



(b) Gdelta_C (with both inter and inner match)

Fig. 6. Compression ratio and encoding speed of Gdelta using different sliding window size for Gear fingerprinting.

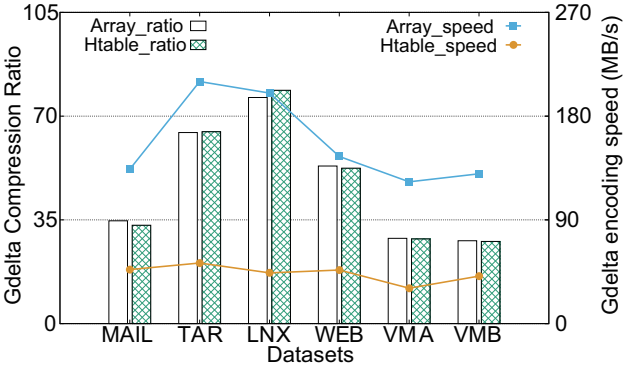


Fig. 7. Delta encoding speed and compression ratio of Gdelta (Gdelta_C) using an Array or a table for Indexing.

find a match), as shown in Figure 7. Here we configure Gdelta with Gear-based hashing and container-based further compression (i.e., Gdelta_C). Here the compared approach 'Htable' means Gdelta uses a hash table as used in Zdelta [19]: When searching for matches, a greedy strategy is used; All possible matches in the base and target chunks are examined and the best one (i.e., the longest duplicate words) is selected. Generally, we can conclude from Figure 7 as:

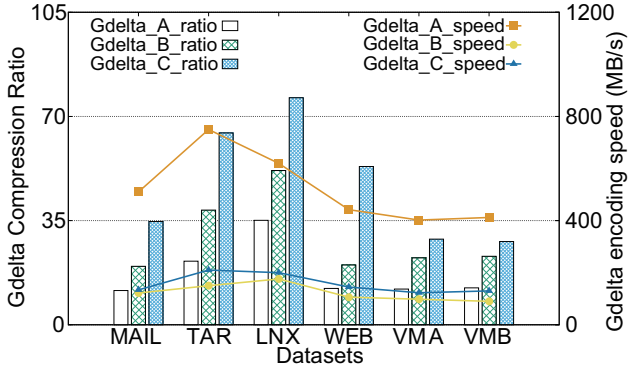


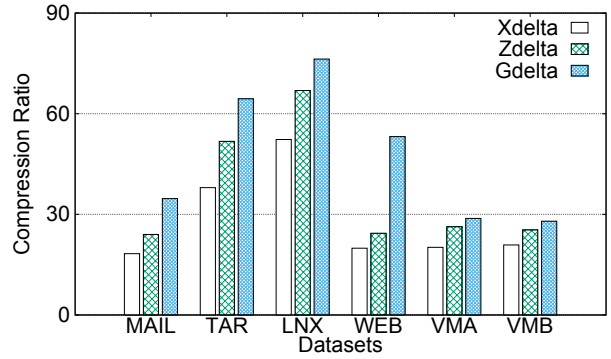
Fig. 8. Delta encoding speed and compression ratio of Gdelta_A, Gdelta_B, and Gdelta_C (with different compression schemes).

- For the *compression ratio*, hash table based approach only achieves a slightly higher compression ratio. This means array-based indexing used in Gdelta works sufficiently well for word-matching in delta encoding while combining with the container-based further compression, and sometimes even achieves a higher compression ratio (e.g., on dataset MAIL) than hash table based approach.
- For the *delta encoding speed*, array-based indexing achieves a much higher encoding speed because of its simple design, as shown in Figure 4 in subsection IV-C. On the other hand, the hash table based approach takes much more time to find the best match for delta encoding while only achieving the minor and even no improvement of the compression ratio on the six datasets.

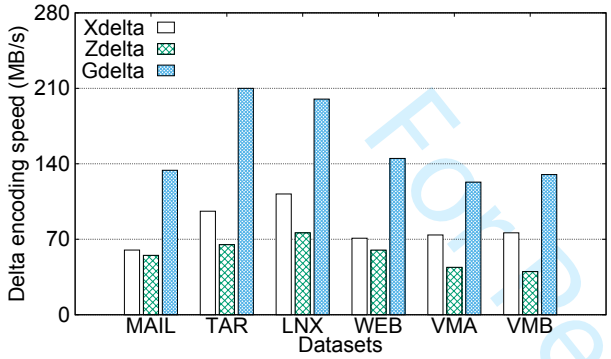
D. Gdelta using Container-based Further Compression

In this subsection, we evaluate Gdelta (i.e., Gdelta_C) performance when using the technique of container-based further compression (i.e., putting many delta chunks altogether for batch compression), as shown in Figure 8. Here we also evaluate Gdelta_A and Gdelta_B for comparison. Generally, we can conclude from Figure 8 as:

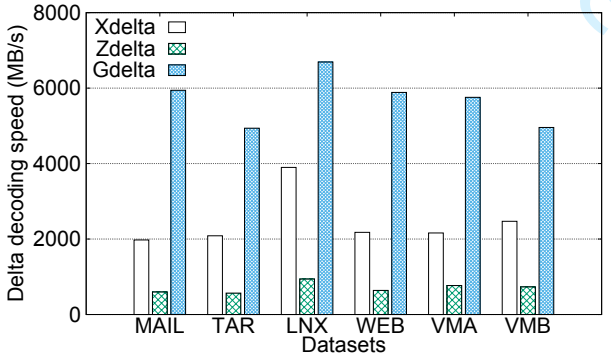
- For the *compression ratio*, container-based further compression technique (i.e., Gdelta_C) improves the compression ratio by 10% ∼90% comparing with only compressing the delta chunks individually (i.e., Gdelta_B). This demonstrates that there are still lots of redundant data among the delta chunks (after delta encoding between the target and base chunks). Meanwhile, Gdelta_B also achieves a much higher compression ratio than Gdelta_A, which suggests that inner match is also necessary after inter match in delta encoding.
- For the *delta encoding speed*, container-based approach not only does not increase the time overhead, but also achieves about 20% higher encoding speed than the approach that only compresses the delta chunks individually. This is because a 4MB-container consists of hundreds of delta chunks, and this batch compression



(a) Compression ratio



(b) Encoding speed



(c) Decoding speed

Fig. 9. Overall performance comparison of Xdelta, Zdelta, and Gdelta (i.e., the final version of Gdelta_C).

could reduce most of the initialization operations of Zstd and FSE techniques for the small delta chunks.

E. Overall Performance

In this subsection, three approaches are evaluated for the overall performance comparison, including Xdelta, Zdelta, and Gdelta (i.e., Gdelta_C with all the techniques used: Gear-based hashing, array-based indexing, and container-based further compression).

From Figure 9, we can find that compared with Xdelta, Zdelta always achieves a higher compression ratio but a much slower encoding/decoding speed since it uses Huffman to further compress the rest after its fine-grained word-matching. At the same time, Gdelta always detects more redundancy than Zdelta, especially on the WEB dataset, with about 2.2× higher compression ratio. This is because Gdelta uses the container-based batch compression with FSE and Zstd techniques to compress the instructions and data, which dramatically reduces the compression metadata for indexing (such as tables, arrays, etc.) in the small delta chunks.

Meanwhile, as shown in Figures 9(b) and (c), Gdelta achieves much higher encoding and decoding speed. The reasons are three-folds. ① Gdelta uses the techniques of Gear-based rolling hash and the array-based indexing, which significantly accelerates the fine-grained word-matching process. ② Gdelta uses container-based batch compression and decompression, which helps accelerate batch delta encoding and decoding. ③ Gdelta also has a higher decoding speed over Xdelta due to it uses a big word of 16 bytes.

In summary, Gdelta using all the techniques mentioned in Section IV outperforms Xdelta and Zdelta on all the three metrics, especially achieving a much higher compression ratio.

VI. Conclusion and Future Work

In this paper, we propose Gdelta, explore the potential of a fast delta encoding approach for marching to a high compression ratio. More concretely, it improves the delta encoding speed by employing an improved Gear-based rolling hash and a fast array-based indexing scheme for fine-grained word-matching. And then, after word-matching, it further batch compresses the rest to improve the compression ratio using the latest Zstd and FSE techniques. Experimental results show that Gdelta speeds up the delta encoding/decoding by 2X∼4X over the classic Xdelta and Zdelta approaches while increasing the compression ratio by about 10%∼120%.

In the future, we plan to improve the container-based batch compression technique in Gdelta to support local decompression, and further study the trade-off between the delta encoding speed and compression ratio in Gdelta.

Acknowledgements

We are grateful to the anonymous reviewers for their insightful comments and constructive suggestions on this work. This research was partly supported by NSFC No. 61972441 and No. 61702134, Key-Area R&D Program for Guangdong Province under Grant No. 2019B01013600, the Shenzhen Science and Technology Program under Grant No. JCYJ20190806143405318. Wen Xia is the Corresponding author (xiawen@hit.edu.cn).

References

[1] New generation entropy codecs : Finite state entropy and huff0. https://github.com/Cyan4973/FiniteStateEntropy/.

[2] Bhavish Aggarwal, Aditya Akella, Ashok Anand, et al. EndRE: an end-system redundancy elimination service for enterprises. In *Proceedings of the 7th USENIX Symposium on Networked Systems Design and Implementation (NSDI'10)*, San Jose, CA, USA, April 2010. USENIX Association.

[3] Lior Aronovich, Ron Asher, Eitan Bachmat, et al. The design of a similarity based deduplication system. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, pages 1–12, Haifa, Israel, May 2009. ACM Association.

[4] Y Collet and M Kucherawy. Zstandard compression and the application/zstd media type. *RFC 8478*, 2018.

[5] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. 2009.

[6] Peter Deutsch and J-L Gailly. Zlib compressed data format specification version 3.3. *RFC Editor, http://tools.ietf.org/html/rfc1950*, 1996.

[7] Idilio Drago, Marco Mellia, Maurizio M Munafo, et al. Inside dropbox: understanding personal cloud storage services. In *Proceedings of the 2012 ACM conference on Internet measurement conference (IMC'12)*, pages 481–494, Boston, MA, USA, November 2012. ACM Association.

[8] Jarek Duda. Asymmetric numeral systems: entropy coding combining speed of huffman coding with compression rate of arithmetic coding. *arXiv preprint arXiv:1311.2540*, 2013.

[9] JL Gailly and M Adler. The gzip compressor. http://www.gzip.org/, 1991.

[10] Diwaker Gupta, Sangmin Lee, Michael Vrable, et al. Difference engine: harnessing memory redundancy in virtual machines. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI'08)*, pages 309–322, San Diego, CA, USA, December 2008. USENIX Association.

[11] David A Huffman. A method for the construction of minimum redundancy codes. *Proceedings of the IRE: Institute of Radio Engineers*, 40(9):1098–1101, 1952.

[12] J. MacDonald. File system support for delta compression. Masters thesis. Department of Electrical Engineering and Computer Science, University of California at Berkeley, 2000.

[13] Athicha Muthitacharoen, Benjie Chen, and David Mazières. A low-bandwidth network file system. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles (SOSP'01)*, page 174–187, New York, NY, USA, 2001. Association for Computing Machinery.

[14] S. Quinlan and S. Dorward. Venti: a new approach to archival storage. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST'02)*, 2002.

[15] David Reinsel, John Gantz, and John Rydning. The digitization of the world from edge to core. *IDC White Paper*, 2018.

[16] Philip Shilane, Grant Wallace, Mark Huang, and Windsor Hsu. Delta compressed and deduplicated storage using stream-informed locality. In *Proceedings of USENIX Conference on Hot Topics in Storage and File Systems (HotStorage'12)*, 2012.

[17] 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, November 2012.

[18] Neil T Spring and David Wetherall. A protocol-independent technique for eliminating redundant network traffic. *ACM SIGCOMM Computer Communication Review*, 30(4):87–95, 2000.

[19] Dimitre Trendafilov, Nasir Memon, and Torsten Suel. Zdelta: An efficient delta compression tool. *Technical report, Department of Computer and Information Science at Polytechnic University*, 2002.

[20] Wen Xia, Hong Jiang, Dan Feng, and Lei Tian. Combining deduplication and delta compression to achieve low-overhead data reduction on backup datasets. In *Proceedings of IEEE Data Compression Conference (DCC'14)*, pages 203–212, Snowbird, Utah, USA, March 2014. IEEE Computer Society Press.

[21] 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.

[22] Wen Xia, Chunguang Li, Hong Jiang, Dan Feng, Yu Hua, Leihua Qin, and Yucheng Zhang. Edelta: A word-enlarging based fast delta compression approach. In *the 7th USENIX conference on Hot Topics in Storage and File Systems (HotStorage'15)*, pages 1–5, Santa Clara, CA, 2015. USENIX Association.

[23] 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 *USENIX Annual Technical Conference (ATC'16)*, pages 101–114, Denver, CO, 2016. USENIX Association.

[24] Lianghong Xu, Andrew Pavlo, Sudipta Sengupta, and Gregory R. Ganger. Online deduplication for databases. In *Proceedings of ACM International Conference on Management of Data (SIGMOD'17)*, pages 1355–1368, Chicago, IL, USA, 2017.

[25] Lianghong Xu, Andrew Pavlo, Sudipta Sengupta, Jin Li, and Gregory R Ganger. Reducing replication bandwidth for distributed document databases. In *Proceedings of the Sixth ACM Symposium on Cloud Computing (SoCC'15)*, pages 222–235, Big Island, Hawaii, USA, 2015. ACM Association.

[26] Qing Yang and Jin Ren. I-cash: Intelligently coupled array of ssd and hdd. In *Proceedings of the 17th IEEE International Symposium on High Performance Computer Architecture (HPCA'11)*, pages 278–289, San Antonio, TX, USA, February 2011. IEEE Computer Society Press.

[27] 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 *IEEE Conference on Computer Communications (INFOCOM'15)*, pages 1337–1345. IEEE, 2015.

[28] Yucheng Zhang, Wen Xia, Dan Feng, Hong Jiang, Yu Hua, and Qiang Wang. Finesse: fine-grained feature locality based fast resemblance detection for post-deduplication delta compression. In *Proceedings of USENIX Conference on File and Storage Technologies (FAST'19)*, pages 121–128, 2019.

[29] Benjamin Zhu, Kai Li, and R Hugo Patterson. Avoiding the disk bottleneck in the Data Domain Deduplication File System. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST'08)*, San Jose, CA, USA, 2008.

[30] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on information theory*, 23(3):337–343, 1977.

[31] Jacob Ziv and Abraham Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24(5):530–536, 1978.