Performance Evaluation 79 (2014) 258–272

Contents lists available at ScienceDirect

Performance Evaluation

journal homepage: www.elsevier.com/locate/peva

Ddelta: A deduplication-inspired fast delta compression approach

Wen Xia[a], Hong Jiang[b], Dan Feng[a,*], Lei Tian[b], Min Fu[a], Yukun Zhou[a]

[a] *Wuhan National Laboratory for Optoelectronics, School of Computer, Huazhong University of Science and Technology, Wuhan, China*
[b] *University of Nebraska-Lincoln, Lincoln, NE, USA*

ARTICLE INFO

Article history:
Available online 10 July 2014

Keywords:
Delta compression
Deduplication
Content locality
Content-defined chunking
Fingerprinting

ABSTRACT

Delta compression is an efficient data reduction approach to removing redundancy among similar data chunks and files in storage systems. One of the main challenges facing delta compression is its low encoding speed, a worsening problem in face of the steadily increasing storage and network bandwidth and speed. In this paper, we present Ddelta, a deduplication-inspired fast delta compression scheme that effectively leverages the simplicity and efficiency of data deduplication techniques to improve delta encoding/decoding performance. The basic idea behind Ddelta is to (1) accelerate the delta encoding and decoding processes by a novel approach of combining Gear-based chunking and Spooky-based fingerprinting for fast identification of duplicate strings for delta calculation, and (2) exploit content locality of redundant data to detect more duplicates by greedily scanning the areas immediately adjacent to already detected duplicate chunks/strings. Our experimental evaluation of a Ddelta prototype based on real-world datasets shows that Ddelta achieves an encoding speedup of 2.5×–8× and a decoding speedup of 2×–20× over the classic delta-compression approaches Xdelta and Zdelta while achieving a comparable level of compression ratio.

1. Introduction

Data reduction has become increasingly important in storage systems due to the explosive growth of digital data in the world. As a result of this “data deluge”, effectively managing storage and reducing its costs have become one of the most challenging and important tasks in mass storage systems. One of the space-efficient approaches for storage management is delta compression that can effectively eliminate the redundancy among similar data blocks and files, which not only reduces data storage requirement [1–5], but also accelerates the network transmission of redundant data [6–10].

Broadly speaking, there are mainly three categories of lossless data reduction technologies in storage systems, namely, traditional compression (e.g., GZIP compression [11]), delta compression [12,13], and data deduplication [6,14]. Traditional compression approaches reduce data at the byte/string level based on classic algorithms such as Huffman coding [15] and dictionary coding [16,17]. On the other hand, deduplication approaches eliminate redundancy at the chunk/file level by dividing data streams into chunks with Content-Defined Chunking (CDC) [6,18] and identifying duplicate chunks by their secure SHA-1/MD5 fingerprints [14], which has been demonstrated to offer very high scalability in backup/archival storage systems [19,20].

* Corresponding author.
E-mail addresses: xia@hust.edu.cn (W. Xia), jiang@cse.unl.edu (H. Jiang), dfeng@hust.edu.cn (D. Feng), tian@cse.unl.edu (L. Tian), fumin@hust.edu.cn (M. Fu), ykzhou@hust.edu.cn (Y. Zhou).

http://dx.doi.org/10.1016/j.peva.2014.07.016
0166-5316/© 2014 Elsevier B.V. All rights reserved.

Delta compression, however, has been gaining increasing attention in recent years for its ability to remove redundancy among non-duplicate but very similar data files and chunks, for which the data deduplication technology often fails to identify and eliminate. Shilane et al. [9] propose a WAN optimized replication approach that combines deduplication, delta compression, and GZIP compression to maximally remove redundancy to accelerate the transmission of backup datasets. Difference Engine [2] and I-CASH [3] make full use of the delta compression technology to eliminate content redundancy in memory pages and SSD caches respectively. DropBox [21,22,10] employs delta compression to calculate the modified areas of the updated files to accelerate the file-upload operations.

One of the main challenges facing the delta compression technology is the time-consuming process of locating duplicate and similar data chunks and calculating the differences among similar data chunks, a worsening problem in face of the steadily increasing storage and network bandwidth and speed [3,4]. This is because the state-of-the-art delta compression techniques, such as Xdelta [12] and Zdelta [13], use a method similar to that of the traditional lossless compression approaches, namely, a byte-wise sliding window to identify matched (i.e., duplicate) strings between the source (or base) chunk and the target (or input) chunk for the delta calculation, which is very time-consuming.

Inspired by the high duplicate-detection scalability of the deduplication technology that employs Content-Defined Chunking (CDC) [18] to divide data stream into several independent and non-overlapping chunks, we believe that the CDC technique can be leveraged to help simplify delta compression's difference (delta) calculation by efficiently dividing the base and input chunks into smaller independent and non-overlapping strings and then detecting duplicates among these strings.

Unfortunately, there is a trade-off between the computation overhead and compression ratio [13]. While a CDC-based approach helps reduce the number of strings for delta calculation, the compression ratio often suffers because the CDC-based approach's identification of the chunking boundary can be inaccurate and thus lead to failure in identifying similar but non-duplicate strings (e.g., two very similar strings with only a few bytes being different). To address this problem, we are inspired by recent studies on content locality of redundant data [7,19,23,24] to fully exploit this property by searching regions immediately adjacent to the confirmed duplicate strings by the CDC approach to detect more duplicate content, thus improving the compression ratio.

In this paper we present Ddelta, a deduplication-inspired fast delta compression approach to simplifying and thus accelerating the process of duplicate identification between similar data files and chunks. The salient features of our Ddelta technique include:

- A Gear-based chunking algorithm that quickly divides similar data chunks into smaller, independent and non-overlapping strings. Gear-based chunking directly uses the re-hashed ASCII values of the data content with only two simple operations (i.e., one shift and one addition) for fast chunking, which cuts more than half of the calculation operations required by Rabin-based chunking [18,6] while achieving similar hashing and chunking efficacy.
- A weaker but faster Spooky-based fingerprinting approach to duplicate identification among strings. If the strings' Spooky hashes match, their content is further verified for duplicates by an in-RAM byte-wise comparison (i.e., using the C function memcmp() that compares two strings).
- A greedy byte-wise scanning approach to searching the areas adjacent to those already identified duplicate strings to hopefully find more redundancy. This is based on the known and observed content locality of data redundancy, i.e., areas/regions immediately adjacent to duplicates tend to have higher probability of being duplicates or similar themselves [23,24].

Our experimental results, based on real-world datasets, show that Ddelta achieves speedups of 2.5×–8× in delta encoding speed and 2×–20× in delta decoding throughput over Xdelta and Zdelta, while achieving a comparable level of compression ratio. In addition, the Gear-based chunking approach achieves a speedup of about 2.1× over the Rabin-based chunking, which helps accelerate both chunking and duplicate detection in Ddelta.

The rest of the paper is organized as follows. Section 2 presents background and motivation for this work. Section 3 describes in detail the novel delta encoding schemes of Ddelta. Section 4 presents our experimental evaluation of a Ddelta prototype and two case studies. Section 5 draws conclusions and provides directions for future work.

2. Background and motivation

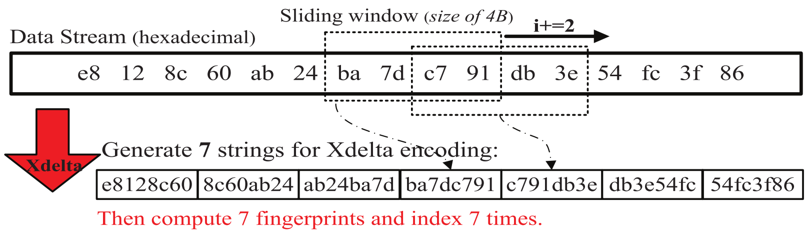
2.1. Delta compression and other data reduction approaches

Delta compression is becoming increasingly popular in storage systems as a space-efficient approach to data management. It can be more space-efficient than data deduplication in removing redundancy among similar and frequently modified data files and chunks. Table 1 shows a comparison among traditional lossless compression, delta compression, and data deduplication.

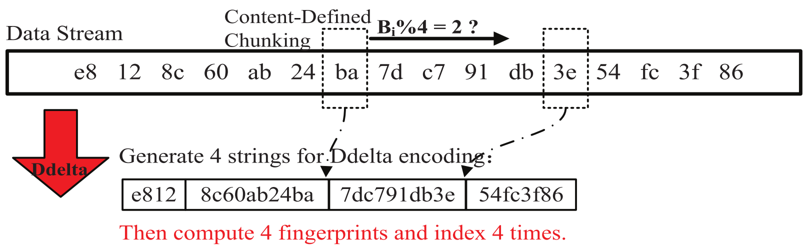
Since traditional lossless compression, delta compression, and data deduplication target different kinds of data for redundancy, they use different techniques to reduce data. As a typical lossless compression approach, the GZ compression [11,25] employs both the byte-level Huffman coding [15] and string-level dictionary coding (e.g., LZ77 [16], LZ78 [17]) to eliminate redundant data, which achieves a high compression efficiency but at the expense of long processing time. Delta compression [12,13] detects duplicates between similar chunks (i.e., the input and base chunks) and encodes the matched/unmatched strings of the input chunk as the instructions of “Copy”/“Insert” in a delta chunk. Delta compression

Table 1
A comparison among three lossless data reduction technologies.

	Traditional lossless compression	Delta compression	Data deduplication
Target	All data	Similar data	Duplicate data
Processing granularity	Byte/string	String	Chunk/file
Representative methods	Huffman coding/dictionary coding	KMP based Copy/Insert	CDC & Secure fingerprint
Scalability	Weak	Weak	Strong
Representative prototypes	GZIP [11], Zlib [25]	Xdelta, Zdelta	LBFS, DDFS


Sliding window (size of 4B)
i+=2
Data Stream (hexadecimal)
e8 12 8c 60 ab 24 ba 7d c7 91 db 3e 54 fc 3f 86
Xdelta
Generate 7 strings for Xdelta encoding:
e8128c60 8c60ab24 ab24ba7d ba7dc791 c791db3e db3e54fc 54fc3f86
Then compute 7 fingerprints and index 7 times.

(a) Byte-wise sliding window based chunking.


Content-Defined Chunking
B_i%4 = 2 ?
Data Stream
e8 12 8c 60 ab 24 ba 7d c7 91 db 3e 54 fc 3f 86
Ddelta
Generate 4 strings for Ddelta encoding:
e812 8c60ab24ba 7dc791db3e 54fc3f86
Then compute 4 fingerprints and index 4 times.

(b) Content-defined sliding window based chunking.

Fig. 1. A comparison between the classic Xdelta's byte-wise sliding window technique and Ddelta's deduplication-inspired content-defined sliding window technique for string matching in delta compression.

uses a byte-wise sliding window, similar to the GZ compression, to find the matched strings between similar chunks, which can be optimized by implementing the Rabin–Karp string matching algorithm [26,12]. Zdelta [13] incorporates the Huffman coding [15] for delta compression to maximally eliminate redundancy.

Deduplication [6,19] divides the data stream into independent chunks of approximately equal length by Content-Defined Chunking (CDC) (e.g., Rabin-based Chunking) algorithm [18] and then uses secure fingerprints (e.g., SHA-1) to uniquely identify these chunks. If any two secure fingerprints match, deduplication will consider their corresponding chunks also identical. Thus the deduplication approach simplifies the process of duplicate detection and scales well in mass storage systems, especially in backup/archival systems with abundant duplicate contents [20]. But deduplication only identifies duplicate chunks/files and thus fails to detect the non-duplicate but very similar chunks/files, which can be supplemented by the delta compression technology.

Consequently, delta compression has been gaining increasing attention in recent years. Shilane et al. [9] implement delta and GZ compression on top of deduplication to further eliminate redundancy to accelerate the WAN replication of backup datasets. Difference Engine [2] employs delta compression, called sub-page level sharing in their study, and LZO compression to save memory usage in VM environments. I-CASH [3] makes full use of the delta compression technique to save space and thus enlarges the logical space of SSD caches. Dropbox [21,22,10] implements delta compression to reduce the bandwidth requirement of uploading the updated files by calculating the difference between two revisions and sending only the delta updates.

2.2. *Content-defined chunking scheme for delta calculation*

As discussed in Section 2.1, while delta compression achieves a superior performance of data reduction among similar data chunks, the challenge of the time-consuming string matching stemming from delta encoding remains [2–4]. The study of Difference Engine shows that delta compression (i.e., page sharing) consumes a large amount of CPU time due to the overheads of resemblance detection and delta encoding [2]. Shilane et al. [4] suggest that the average delta encoding speed of similar chunks falls in the range of 30–90 MB/s, which may become a potential bottleneck in storage systems. This is because delta compression finds the matched strings by using a byte-wise sliding window. If the string of the sliding window does not find a match, the sliding window will move forward by only one or several bytes at a time [12], as shown in Fig. 1(a).

Inspired by deduplication that divides the data streams into independent and non-overlapping chunks to simplify the duplicate detection, we believe that the string matching process can also be simplified by the chunking concept of deduplication. This means that delta compression does NOT need to employ the byte-wise sliding window to find the matching strings. Instead, we divide the base and input chunks into smaller, independent, and non-overlapping strings by CDC and only identifies duplicates among strings. As shown in Fig. 1(b), Ddelta is motivated by deduplication to employ a simple content-based sliding window to find a chunking boundary to generate non-overlapping strings for duplicate identification in delta compression. As a result, Ddelta is able to generate fewer strings than Xdelta, which accelerates the process of string matching for delta calculation.

If the chunks/files are very similar, most of their chunked positions should be identical. This is because the same content in the sliding window will generate the same hash values (e.g., Rabin Hash [18,6,27,28]) while the chunked positions are decided if hash values of the sliding window are matched with a predefined value. In fact, based on our experimental observations and recently published studies [9,2,22,29], the delta compression candidates tend to be very similar and thus very compressible, meaning that the CDC-based approach can find enough duplicate candidates for delta compression as demonstrated in Section 5.

However, the CDC-based approach may generate non-duplicate but very similar strings (e.g., two very similar strings with only a few bytes being different) and lead to failure in identifying redundancy among them. We will address this problem of compromised compression ratio due to the possible inaccurate boundary identification of CDC-based chunking in Section 3.3, where we take inspirations from studies on content locality of redundant data [19,23,24] to search for likely more redundancy in the duplicate-adjacent areas/regions.

3. Design and implementation

3.1. Deduplication-inspired delta compression

The design goal of Ddelta is to accelerate the duplicate detection process in the encoding phase of delta compression for similar data files and chunks. Algorithm 1 describes the key workflow of deduplication-inspired Ddelta encoding that is designed to speedup encoding. For data chunks that are either identified to be similar via resemblance detection [1,4,5] or already known delta compression candidates [21,10,12], Ddelta is applied to fast delta encoding these data chunks as follows:

1. *Gear-based chunking*. Ddelta divides the similar chunks into independent and non-overlapping strings based on the Gear hashes of their content, which is implemented by the GEARCHUNKING( ) function detailed in Section 3.2 and in Algorithm 2.
2. *Spooky-based fingerprinting and duplicate identification*. Ddelta identifies the duplicate strings obtained from Gear-based chunking above by computing and then matching their Spooky hashes. If their Spooky hashes match, their content will be further verified for duplicates, as implemented in the FINDMATCH( ) function of Algorithm 1.
3. *Greedily scanning duplicate-adjacent areas*. To improve the duplicate detection and thus detect more duplicates, Ddelta byte-wisely scans non-duplicate areas immediately adjacent to the above confirmed duplicate strings exploiting content locality of redundant data as detailed in Section 3.3.
4. *Encoding*. Ddelta encodes the duplicate and non-duplicate strings as “Copy” and “Insert” instructions respectively, as detailed in Section 3.4.

To simplify duplicate detection, Ddelta fast computes the fingerprints of strings and builds a small index table “*Sindex*” for lookup, by using a weaker but faster hash scheme, called Spooky hash [30], in place of a time-consuming secure fingerprinting approach (e.g., SHA-1). Here, Ddelta employs the 64-bit Spooky hash for string fingerprinting to minimize the hash collisions.

Since the data objects being processed by Ddelta compression are just two similar chunks or files (the input and base) and thus can all easily fit in RAM, a fast byte-wise comparison for duplicates becomes possible. If two strings' fingerprints match, Ddelta checks their contents by an in-RAM byte-wise comparison (i.e., memcmp( ) in C language) whose overhead is negligible relative to chunking and fingerprinting. Note that other fast hash approaches like Murmur [31] and xxHash [32] can also be employed for fast Ddelta fingerprinting.

3.2. Gear-based fast chunking

As suggested by recently published studies [27,28] and our experimental observations, the pure Rabin-based chunking is time-consuming and thus may be a source of potential performance bottleneck for delta encoding. To address this problem, we propose a faster hash-based chunking algorithm, called Gear-based chunking, to accelerate the CDC scheme for Ddelta by using the re-hashed ASCII values of data content of similar chunks, as shown in Fig. 2. In Gear-based chunking, the hash value of the sliding window is cumulatively and quickly calculated from the previous value by the *GearHash* in the form of “$H_i = (H_{i-1} \ll 1) + \mathit{GearTable}[B_i]$”.

A good hash function must have a uniform distribution of hash values regardless of the hashed content. GearHash achieves this in two key steps: (1) GearHash employs an array of 256 random 32-bit integers to map with the ASCII values

Algorithm 1 Deduplication Inspired Ddelta Encoding

```
Input: base chunk, $src$; input chunk, $tgt$
Output: delta chunk, $dlt$;
function ComputDelta($src$, $tgt$)
    $dlt \leftarrow empty$
    $Slink \leftarrow$ GearChunking($src$)
    $Tlink \leftarrow$ GearChunking($tgt$)
    $Sindex \leftarrow$ initMatch($Slink$)
    $str \leftarrow Tlink$; $len \leftarrow size(str)$
    while $str! = NULL$ do
        $pos \leftarrow$ findMatch($src$, $Sindex$, $str$, $len$)
        if $pos < 0$ then                                  ▷ No matched string
            $dlt+ =$Instruction($Insert\ str, len$)           ▷ Add the unmatched string into the delta chunk
        else                                               ▷ Find a matched string
            $dlt+ =$Instruction($Copy\ pos, len$)             ▷ Add the matched info into the delta chunk
        end if
        $str \leftarrow (str \rightarrow next)$; $len \leftarrow size(str)$
    end while
    return $dlt$
end function

function initMatch($Slink$)
    $str \leftarrow Slink$; $pos \leftarrow 0$
    $Sindex \leftarrow empty$
    while $str! = empty$ do
        $f \leftarrow$ Spooky($str$, $size(str)$)
        $Sindex[hash[f]] \leftarrow pos$
        $pos+ = size(str)$
        $str \leftarrow (str \rightarrow next)$
    end while
    return $Sindex$
end function

function findMatch($src$, $Sindex$, $str$, $len$)
    $f \leftarrow$ Spooky($str$, $len$)
    if $Sindex[hash[f]] = empty$ then
        return $-1$                                        ▷ No matched fingerprint
    end if
    $pos \leftarrow Sindex[hash[f]]$
    if memcmp($src + pos$, $str$, $len$)$= 0$ then
        return $pos$                                       ▷ Find a matched string
    else
        return $-1$                                        ▷ Spooky hash collision
    end if
end function
```

of the data bytes in the sliding window; and (2) The addition ("+") operation adds the least-significant byte in the content-defined sliding window into Gear hashes while the left-shift ("$\ll$") operation helps strip away the most-significant byte of the content-defined sliding window. As a result, GearHash generates uniformly distributed hash values by using only three operations (i.e., "+", "$\ll$", and an array lookup) while quickly moves through the data content for the purpose of chunking.

Algorithm 2 describes the workflow of Gear-based chunking for delta encoding. Gear-based chunking determines the chunking boundary by checking whether the value of the x most-significant bits (MSB) of Gear hash is equal to a predefined value or not. Note that this MSB-based way of finding the boundary by Gear-based chunking is different from Rabin-based chunking in that the latter determines the chunking boundary by checking if the Rabin hash value mod M, i.e., the value of the x least-significant bits (LSB), is equal to a predefined value, where M is the average chunk size. Note that the number x of MSB or LSB bits is determined by the average chunks size M as $x = \log_2(M)$. For example, to generate chunks of average 8 KB size, Gear-based chunking would choose the 13 most-significant bits (i.e., $\log_2(8192)$) of the Gear hash value to determine the chunking boundary, whereas Rabin-based chunking would choose the 13 least-significant bits.

Table 2 shows a comparison of pseudocode and computation overhead between Rabin-based chunking [27,33,6] and Gear-based chunking. By effectively employing a random-integer table and incorporating with one addition ("+") and

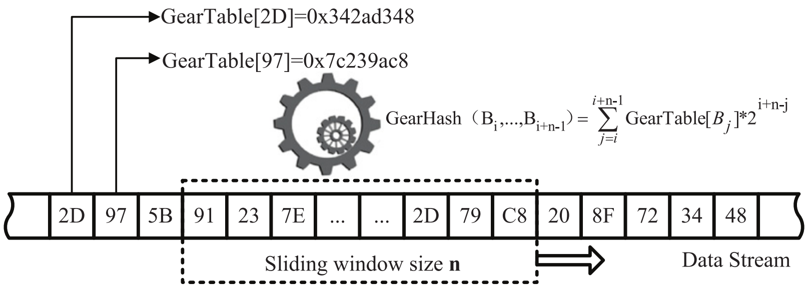

GearTable[2D]=0x342ad348
GearTable[97]=0x7c239ac8
GearHash (B_i,...,B_{i+n-1}) = Σ_{j=i}^{i+n-1} GearTable[B_j]*2^{i+n-j}
2D 97 5B 91 23 7E 2D 79 C8 20 8F 72 34 48
Sliding window size n
Data Stream

Fig. 2. Gear-based fast chunking.

Algorithm 2 Gear-based Chunking for Ddelta Encoding

```
Input: chunk, src;
Output: strings linklist, Slink;
1:  Predefined values: mask bits for the average string size, mask; matched value, xxx
2:  function GEARCHUNKING(src)
3:      $fp \leftarrow 0$; $pos \leftarrow 0$; $last \leftarrow 0$
4:      $SLink \leftarrow empty$
5:      while $pos < size(src)$ do
6:          $fp = (fp << 1) + GearTable[src[pos]]$
7:          if $fp$ & $mask = xxx$ then                         ▷ Get the MSB of fp
8:              $str = src[last \rightarrow pos]$
9:              InsertLinkList($str$, $Slink$)
10:             $last \leftarrow pos$
11:         end if
12:         $pos \leftarrow pos + 1$
13:     end while
14:     return SLink
15: end function
```

one left-shift ("≪") operation, Gear-based chunking eliminates more than half of calculation overhead from Rabin-based chunking while achieving a comparable hashing and chunking efficacy as demonstrated in Section 4.2.

3.3. Greedily scanning the duplicate-adjacent areas

While the Gear-based CDC approach enables Ddelta to effectively reduce the number of strings for duplicate identification, the challenge of decreased compression ratio due to CDC remains for Ddelta. *This is because the CDC-based approach* [6,18] *cannot always accurately find the boundary between the changed and duplicate regions among similar data and, instead, simply determines the chunk boundary if the hash (e.g., Rabin, Gear) of the CDC sliding window matches a predefined value*. Consequently, for some non-duplicate but nearly identical strings (i.e., with only one or two bytes being different), Ddelta may generate totally different Spooky fingerprints and thus miss the opportunity to identify redundancy among these strings.

To address this challenge, we take inspirations from the studies of Bimodal Chunking [23] and SiLo [24] on the content locality of redundant data, which suggests that the neighboring data of duplicate chunks should be considered good deduplication candidates due to the data-stream content locality. Therefore, Ddelta also searches the areas immediately adjacent to known duplicates to lend themselves to easy duplicate detection by the following two steps.

- *Chunk-level search in the duplicate-adjacent areas*. For those resemblance-detected chunks [1,2,4] (i.e., the delta compression candidates), Ddelta will directly search and scan from both ends of similar chunks toward centers for duplicate matching until byte-wise comparison fails to find a match. For example, given contiguous chunks {A, B, C} and {A′, B′, C′} from two data streams where A′ & C′ are detected to be duplicate to A & C respectively and B′ is resemblance-detected to be very similar to B, Ddelta can search the duplicate-adjacent areas, namely, the beginnings and the endings of B′ and B, to detect more duplicate contents due to the content locality of data stream.
- *String-level search in the duplicate-adjacent areas*. For those duplicate-detected strings, Ddelta byte-wisely searches and scans their immediately adjacent non-duplicate strings to identify more duplicates in a way similar to above chunk-level search.

Section 3.5 summarizes the deduplication-inspired delta encoding with these two steps incorporated, which helps Ddelta detect more duplicates for delta calculation as demonstrated in Section 4.3.

Table 2
A comparison between the implementations of the Gear- and Rabin-based CDC. Here '*a*' and '*b*' denote contents of the first and the last byte of the sliding window respectively, '*N*' is the length of the content-defined sliding window, and '*U*' & '*T*' denote two predefined arrays for the computation of Rabin-based chunking [33,6].

Name	Pseudocode & overhead analysis
Rabin	$hash = ((hash \hat{} U(a)) \ll 8) \vert b \hat{} T[hash \gg N]$ Total overhead: 1 OR, 2 XORs, 2 SHIFTs, 2 ARRAY LOOKUPs
Gear	$hash = (hash \ll 1) + GearTable[b]$ Total overhead: 1 ADD, 1 SHIFT, 1 ARRAY LOOKUP

3.4. Encoding and decoding

Combining the techniques of deduplication-inspired delta encoding and greedily scanning for more duplicates, Ddelta encodes the matched and unmatched regions of the input chunk as a delta chunk by the two instructions of "Copy[offset,length]" and "Insert[data,length]" respectively. For those matched or unmatched contiguous data regions, Ddelta will merge them into a single "Copy" or "Insert" instruction to simplify both the delta encoding and decoding operations.

To restore the input chunk, Ddelta decodes the instructions in the delta chunk sequentially. For a "Copy" instruction, Ddelta reads the data from the base chunk according to the information of offset and length. For an "Insert" instruction, Ddelta directly reassembles the data from the delta chunk into the restored chunk.

3.5. Putting it all together

To put things together and in perspective, Fig. 3 shows the Ddelta encoding workflow by way of an example. For each pair of resemblance-detected similar chunks or already known delta compression candidates, Ddelta goes through the following four key steps.

- **Step 1.** Chunk-level greedily scanning for duplicates as illustrated in Fig. 3(a).[1]
- **Step 2.** Duplicate-string identification by Gear-based chunking and Spooky-based fingerprinting as illustrated in Fig. 3(b).
- **Step 3.** String-level greedily scanning for duplicates as illustrated in Fig. 3(c).
- **Step 4.** Generate delta chunk based on results of duplicate identification from Steps 1–3 as illustrated in Fig. 3(d).

4. Performance evaluation

4.1. Experimental setup

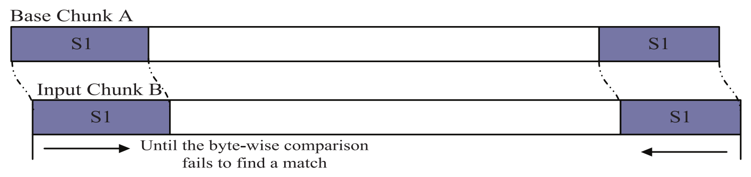
Experimental platform. We implement and evaluate a Ddelta prototype on the Ubuntu 12.04.2 operating system running on a quad-core Intel i7 processor at 2.8 GHz, with a 16GB RAM, two 1TB 7200RPM hard disks, and a 120GB SSD of Kingston SVP200S37A120G.

Configurations for data reduction. For data deduplication, the average, maximum, and minimum chunk sizes are 8 KB, 64 KB, and 2 KB respectively, which is the same configuration as the one in LBFS [6]. For delta compression, two well-known open-source projects, Xdelta [12] and Zdelta [13], are used in the evaluation as the baselines for the proposed Ddelta compression. In addition, we use the GZIP compression [11] (short for GZ) to assess the post-delta-compression data reduction performance.

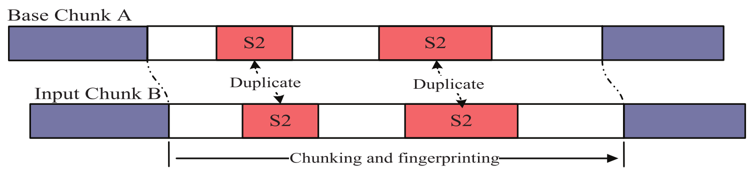
Evaluation metrics. *Compression ratio (CR)* is measured in terms of the percentage of data reduced by any of the evaluated data-reduction schemes, namely, deduplication, delta compression (Xdelta, Zdelta, and Ddelta), and GZ compression. *Compression factor* (*CF*) is measured by the ratio of data sizes before and after data reduction by any of the evaluated schemes. Thus, $CR = \frac{CF}{(CF+1)}$. *Encoding/decoding speeds* (throughputs) are recorded by dividing the amount of encoded/decoded data by the response time of the given evaluated delta compression approach.

Evaluation datasets. Six datasets are used in the evaluation of Ddelta, with their key characteristics summarized in Table 3 and detailed as follow.

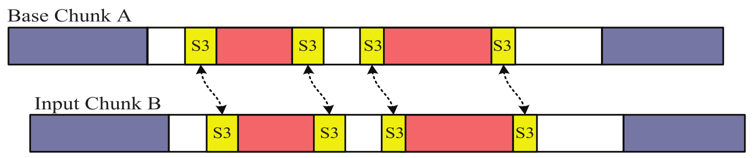
[1] Byte-wise comparison is frequently used in our algorithm, thus it plays an important role in the overall performance of Ddelta. To avoid the naive byte-by-byte comparison that can be slow, we perform one 64-bit XOR operation for each pair of 8-byte strings at a time. If the XOR result is zero, it means that all 8 bytes are identical, and we can move the next 8 bytes. Otherwise, we need to perform right shift operations on the XOR result and obtain the byte offset that causes the difference.



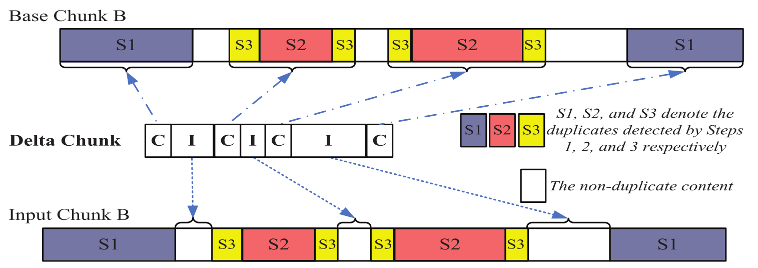
(a) Step 1: Scanning from both ends of two known similar chunks (i.e., A & B that are adjacent to confirmed duplicate chunks (not shown here)) toward the center in search of duplicate content by byte-wise comparison, until match fails.



(b) Step 2: Using Gear-based chunking to divide the remaining areas (i.e., non-duplicate portions of A & B after Step 1) into smaller and non-overlapping strings and identifying duplicate strings by their Spooky fingerprints & content.



(c) Step 3: Scanning the areas immediately adjacent to duplicate strings identified in Step 2 for additional duplicate content.


Base Chunk B
S1
S3
S2
S3
S3
S2
S3
S1
Delta Chunk
C
I
C
I
C
I
C
S1
S2
S3
S1, S2, and S3 denote the duplicates detected by Steps 1, 2, and 3 respectively
The non-duplicate content
Input Chunk B
S1
S3
S2
S3
S3
S2
S3
S1

(d) Step 4: Encoding duplicate and non-duplicate data regions into the delta chunk, where C and I stand for the “Copy” and “Insert” instructions respectively.

Fig. 3. The workflow of the Ddelta approach.

- **GCC** and **Linux** are two well-known open-source projects that are available from the websites [34,35]. These two datasets represent workloads of typical large software source code.
- **VM-A** is a VM archive that consists of 78 virtual machine images of different OS release versions from the website [36], including 23 Centos images, 18 Fedora images, 17 Ubuntu images, 12 FreeBSD images, and 6 Debian images. This datasets represent the workload with low deduplication factor.
- **VM-B** is a VM backup dataset containing 177 full backups of an Ubuntu 12.04 virtual machine in use, which is a common use-case for data reduction in the real world.
- **RDB** is a dataset collected from the Redis key–value store database [37]. We backup the dump.rdb files as the snapshots of the Redis database, and collect 200 backups, which represents a typical database workload for data reduction.
- **Bench** is a benchmark dataset generated from the snapshots of a personal cloud storage benchmark [21]. We simulate common operations of file systems, such as file create/delete/modify on the snapshots of this benchmark according to existing approaches [38,39].

4.2. An empirical study of Gear hash

In this subsection, we evaluate the efficiency and efficacy of Gear hash for CDC with an average chunk size of 8 KB. Fig. 4 shows the distribution of hash values of the Rabin and Gear algorithms under three typical workloads, Linux, VM-A, and RDB, by chunking one image each of these datasets using these algorithms. To estimate the hash distribution, throughout the entire chunking process we count the total number of occurrences of hashing that result in a given masked hash value. Thus, each dot in Fig. 4 indicates the total number of occurrences of hashing (Y axis) that result in the same masked hash

Table 3
Workload characteristics of the six datasets used in the performance evaluation. Here deduplication factor is measured by Rabin-based chunking.

Dataset name	# of Images/versions	Total size	Deduplication factor
GCC	43	14.1 GB	6.71
Linux	258	104 GB	44.7
VM-A	78	114 GB	1.65
VM-B	117	1.78 TB	25.8
RDB	211	1.15 TB	7.22
Bench	200	1.54 TB	35.0

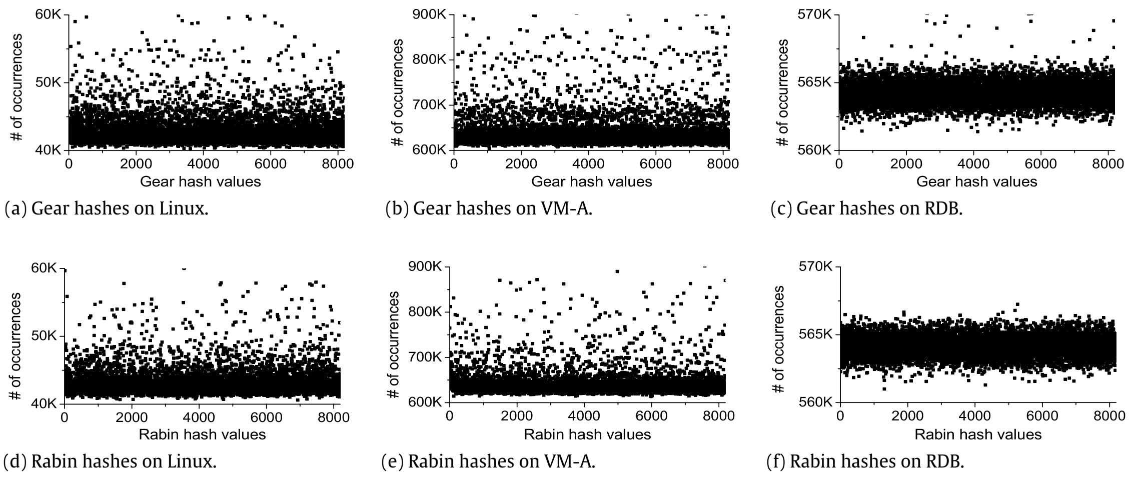


Fig. 4. A comparison of hash distribution between the Rabin-based and Gear-based chunking schemes. Here the X axis shows the masked hash values, that is, the 13 most/least significant bits of the Gear/Rabin hash values for CDC with an average chunk size of 8 KB in a deduplication system. The Y axis indicates the total number of occurrences of hashing that result in the same masked hash value.

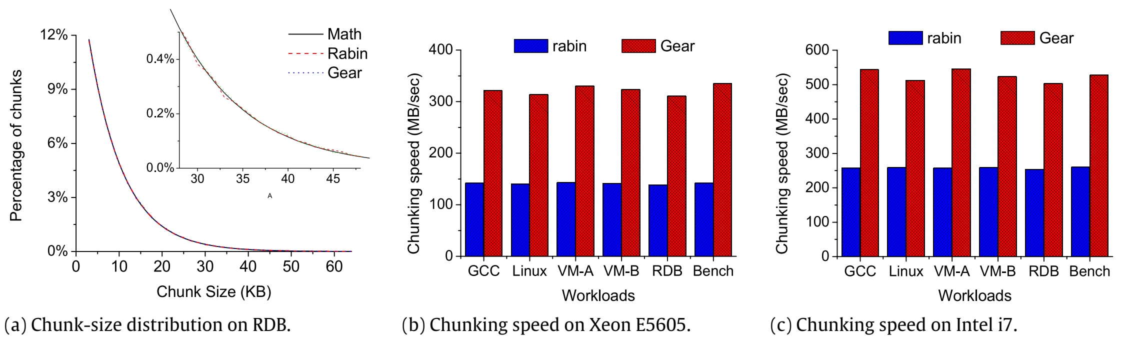


Fig. 5. Comparisons of chunk-size distribution and chunking speed between Rabin-based and Gear-based chunking approaches. Note that chunk-size distributions of other datasets are similar to and consistent with results of RDB in (a), and thus are omitted here.

value on the X axis. With this plotting, the closer the dots clutter around a horizontal band (i.e., hash values being more uniformly distributed), the more desirable the hash algorithm is. The results in Fig. 4 suggest that the Gear hash algorithm achieves a similar uniform distribution as the Rabin hash algorithm under the three workloads, indicating that the former achieves its low compute overhead without sacrificing the hash quality for CDC, as discussed in Section 3.2.

Fig. 5(a) shows the distribution of chunk-size generated by the Gear-based and Rabin-based chunking approaches on the Linux dataset. In fact, the chunk-size distributions of the Gear-based and Rabin-based approaches are nearly identical and follow the exponential distribution (i.e., $F(x) = (1 - e^{-\frac{x-2048}{8192}})$, where x is the chunk size) as denoted by the "math" curve in Fig. 5(a). On the other hand, Figs. 5(b) and (c) suggest that the Gear-based chunking achieves average speedups of about 2.04× and 2.28× over the Rabin-based chunking on the Intel i7 and Xeon E5605 processors, respectively. The results in Fig. 5 are consistent with the overhead analyzed in Table 2 of Section 3.2, which demonstrates that Gear hash can be used as a replacement of the Rabin hash algorithm for faster CDC.

Table 4 compares the compression performance between Rabin-based CDC and Gear-based CDC with an expected average chunk size of 8KB for data deduplication. The comparisons of CR and CF on the six datasets clearly show that

Table 4
Comparisons of compression ratio, compression factor, and the real average chunk size between Rabin-based chunking and Gear-based chunking for data deduplication with the expected average chunk size of 8 KB.

Dataset	Compression ratio (*CR*)			Compression factor (*CF*)			Average chunk size		
	Rabin	Gear	Difference	Rabin	Gear	Difference	Rabin (KB)	Gear (KB)	Difference (KB)
GCC	85.09%	85.09%	0.003%	6.71	6.71	0.001	4.19	4.20	0.011
Linux	97.76%	97.76%	0.001%	44.69	44.71	0.015	5.80	5.83	0.033
VM-A	39.50%	39.83%	0.329%	1.65	1.66	0.009	12.4	11.3	1.144
VM-B	96.13%	96.15%	0.035%	25.84	25.99	0.156	10.9	10.5	0.395
RDB	86.15%	86.18%	0.023%	7.22	7.24	0.018	9.55	9.54	0.008
Bench	97.14%	97.15%	0.009%	35.03	35.13	0.106	9.95	9.93	0.019

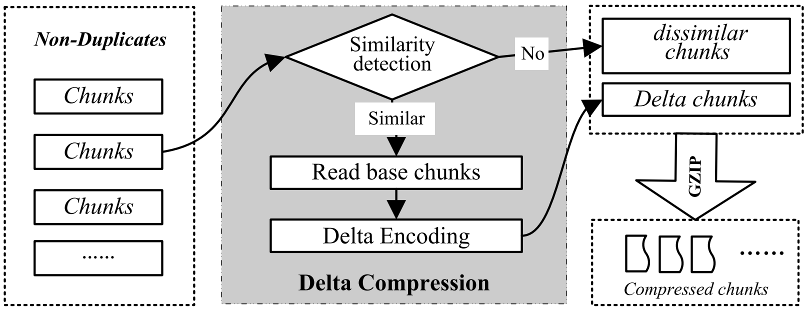


Fig. 6. The workflow of a post-deduplication data reduction (delta +GZ compression) system.

the difference in duplicate identification on the sets of data chunks generated by Rabin-based chunking and Gear-based chunking is practically negligible. The difference in the real average chunk size between these two approaches is very insignificant, which is consistent with the observations made from Figs. 4 and 5(a). Note that the real average chunk size generated by the Rabin and Gear approaches are very different from the expected value of 8 KB. The reasons are twofold. First, we configure the minimum chunk size to be 2 KB, which is the same as LBFS [6]. Second, the actual chunk size is determined when the CDC process reaches the end of the file.

Based on the results in Figs. 4 and 5, and Table 4, we believe that Gear-based chunking is comparable to Rabin-based chunking in terms of the uniformity of chunk-size distribution and the data reduction efficiency, while achieving a hashing speed twice as fast.

4.3. Case study I: delta compression of resemblance-detected similar chunks

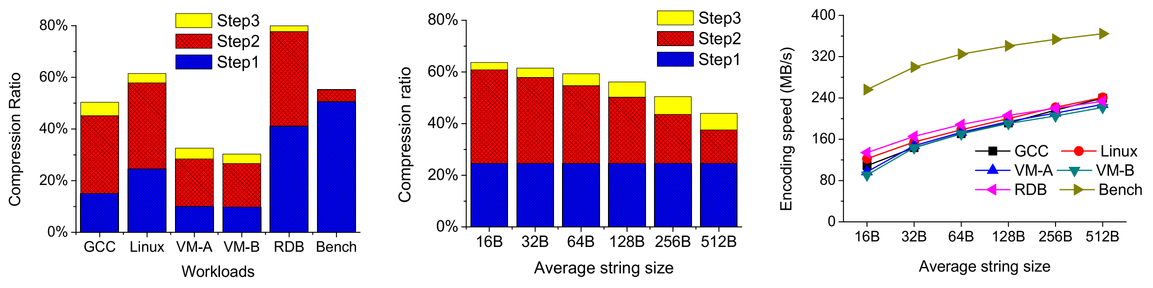
In this subsection, we evaluate the compression efficiency and delta encoding/decoding speed of Ddelta by means of a post-deduplication data reduction system that implements delta and GZ compression on the non-duplicate chunks (i.e., data deduplication is already done) as shown in Fig. 6.

Generally speaking, implementing delta compression on top of data deduplication entails three key functional components as indicated in Fig. 6, (1) resemblance detection, (2) reading the base chunk, and (3) delta encoding the input chunk with the base chunk. Here we employ a widely used super-feature based approach [1,4,9,5] to detecting resembling chunks among the non-duplicate data chunks resulting from data deduplication for delta compression. The super-feature approach generates features of chunks by their Rabin fingerprints and group these features into super-features, which has been used widely to successfully identify similar web pages, files, and chunks [1]. If an input chunk was resemblance-detected by matching a super-feature, the system will read the base chunk that has the matched super-feature for delta encoding by Xdelta, Zdelta, and Ddelta respectively. GZ compression will be applied to further reduce redundancy when delta compression has finished encoding all similar chunks.

In this case study, we use a configuration of 3 super-features and 2 features per super-feature for resemblance detection in our evaluation of delta compression. As the resemblance detection may detect multiple resembling chunks for the input chunks, we adopt the “First-Fit” approach proposed in REBL [1]. GZ compression is applied in a 128 KB write buffer to achieve a good trade-off between compression ratio and speed [20]. Since reading base chunks for delta encoding inevitably causes random IOs for the on-disk data [4], we store the dissimilar chunks and delta chunks on SSD to examine the maximum performance of delta encoding/decoding of the Ddelta, Xdelta, and Zdelta approaches.

Table 5 shows the *CF* achieved by data deduplication, delta compression, and GZ compression respectively. Delta compression achieves a *CF* of more than 3 on the Linux and RDB datasets and *CF* of 1.5×–2.5× on the other four datasets. GZ compression achieves a *CF* of more than 2× on the first four datasets but does relatively poorly on the last two datasets that contain a large amount of random-byte content.

Based on the post-deduplication data reduction system described in Fig. 6 and results shown in Table 5, we conduct a sensitivity study of Ddelta compression efficiency according to its two key ideas of reducing compute overhead by Gear-based

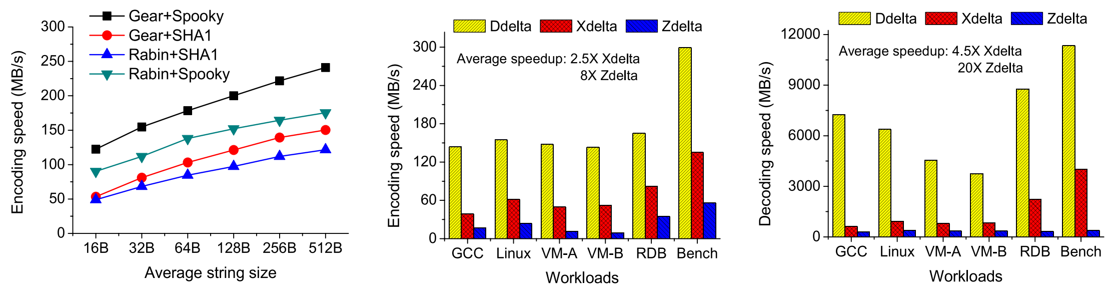


(a) Contributions to *CR* by the three duplicate-identification steps of Ddelta.

(b) *CR* as a function of the average string size on the Linux dataset.

(c) Encoding speed as a function of the average string size.

Fig. 7. A closer look at Ddelta, its three key steps for duplicate identification, i.e., Steps 1–3 of Fig. 3: (1) chunk-level greedily scanning for duplicates, (2) Gear-based duplicate-string identification, and (3) string-level greedily scanning for duplicates, and its sensitivity to the average string size for Gear-based chunking.



(a) Ddelta vs. deduplication.

(b) Encoding speed.

(c) Decoding speed.

Fig. 8. Comparisons in delta encoding/decoding speed among deduplication, Ddelta, Xdelta, and Zdelta. (a) Combines one of the two chunking schemes, Rabin and Gear, with one of the two fingerprinting schemes, SHA1 and Spooky, to compare Ddelta (Gear+Spooky) with deduplication (Rabin+SHA1). (b) and (c) Compare Ddelta with Xdelta and Zdelta in encoding/decoding speed, where Ddelta is configured with an average string size of 32B.

Table 5
Compression factors of deduplication and post-deduplication data reduction. Here we use the Xdelta as the representative approach for delta compression. Note that $CF_{Total} = CF_{Dedupe} \times CF_{Delta} \times CF_{GZ}$.

Dataset	Compression factor (*CF*)				Final size
	Dedupe	Delta	GZ	Total	
GCC	6.71	2.58	2.90	50.2	287 MB
Linux	44.7	3.14	3.09	434.5	245 MB
VM-A	1.65	1.60	2.51	6.63	17.2 GB
VM-B	25.8	1.55	2.35	93.7	19.0 GB
RDB	7.22	5.29	1.47	56.3	20.4 GB
Bench	35.5	2.23	1.00	78.4	19.6 GB

chunking and Spooky-based fingerprinting, and of improving compression ratio by greedily scanning the duplicate-adjacent areas.

Fig. 7 shows the Ddelta compression performance with a breakdown of contributions by its three key duplicate-identification steps (i.e., Steps 1–3 of Fig. 3) and as a function of the average string size for Gear-based chunking on the six datasets. Fig. 7(a) shows that Step 1 of Ddelta detects about one third of redundancy by greedily scanning from two ends of similar chunks toward their centers without chunking and fingerprinting. Fig. 7(b) suggests that redundancy eliminated by Ddelta decreases as the average string size on the Linux dataset increases while Step 3 helps detect more duplicates by scanning the areas adjacent to duplicate strings detected in Step 2. Fig. 7(c) indicates that the Ddelta encoding speed increases with the average string size. This is because when using a smaller average string size, Ddelta generates more strings and thus computes more fingerprints for duplicate identification. Note that Ddelta achieves the highest delta encoding speed on the Bench dataset in Fig. 7(c) because its Step 1 contributes to more than 80% of duplicate detection as shown in Fig. 7(a).

Fig. 8(a) shows a comparison in encoding speed between the deduplication (Rabin +SHA1) and Ddelta (Gear +Spooky) solutions as a function of the chunked string size. Ddelta accelerates the deduplication-based delta encoding by a factor of about 1.7× via Spooky-based fingerprinting and by a factor of about 1.4× via Gear-based chunking. This indicates that the Gear-based and Spooky-based approaches contribute significantly to Ddelta's ability to reduce the compute overhead for delta encoding. Fig. 8(b) and (c) compare the delta compression performance of Xdelta, Zdelta, and Ddelta (with an average string size of 32B) on the six datasets. Ddelta is shown in Fig. 8(b) to achieve an average delta encoding speedup of 2.5× and 8× over Xdelta and Zdelta respectively, by chunking the similar data into independent and non-overlapping

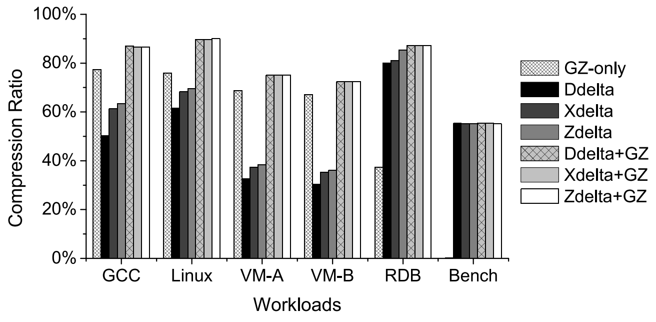


Fig. 9. Comparison in *CR* (compression ratio) among seven post-deduplication data reduction schemes.

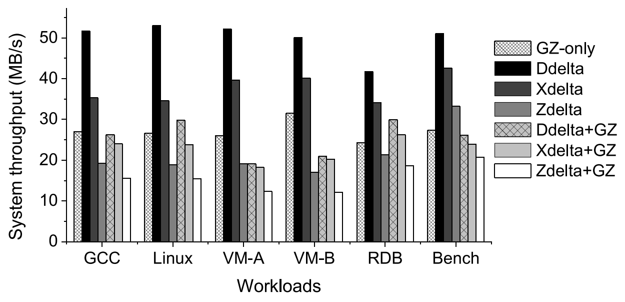


Fig. 10. Comparison in compressing throughput among seven post-deduplication data reduction schemes.

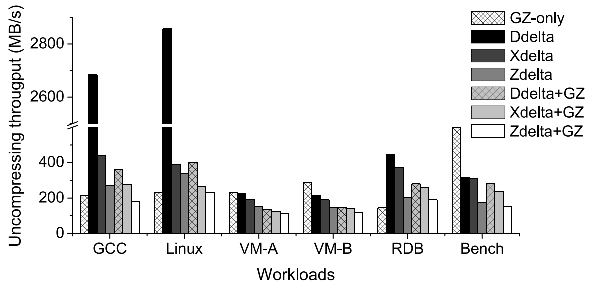


Fig. 11. Comparison in uncompressing throughput among seven post-deduplication data reduction schemes.

strings to simplify the duplicate detection. Meanwhile, Ddelta also achieves a decoding speedup of 4.5×–20× over Xdelta and Zdelta by using Gear-based chunking to generate non-overlapping strings for duplicate identification. Note that Zdelta has the lowest encoding and decoding speeds because of its addition of the Huffman coding for delta compression, which improves redundancy elimination slightly but at the expense of longer processing time.

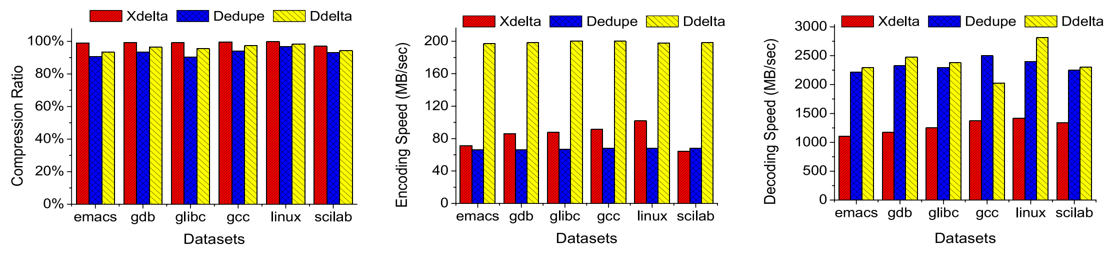
Fig. 9 shows the *CR* of different combinations of various data reduction approaches. Ddelta (with an average string size of 32B) achieves nearly 90% of compression ratio of Xdelta and Zdelta when the post-deduplication data reduction system only applies delta compression on top of deduplication. But when the system combines Delta and GZ compression together for data reduction, the Ddelta+GZ approach achieves nearly the same *CR* as the Xdelta+GZ and Zdelta+GZ solutions. GZ compression achieves better data reduction performance than delta-compression-only approaches on the first four datasets but their combined GZ+Delta solution obtains a superior performance of *CR* to the GZ-only approach. Meanwhile, GZ compression performs relatively poorly on reducing the last two datasets that contain significant random-byte content (*CR* of GZ-only approach on the Bench dataset is nearly equal to zero), which were already well compressed by delta compression solutions.

Fig. 10 further examines the system compressing throughput under different data reduction solutions. Generally speaking, Ddelta achieves the highest throughput of compressing the deduplicated data with or without combining GZ compression. Note that the compressing throughput of Ddelta is much lower than its delta encoding speed due to the overhead of computing super-features for resemblance detection and reading base chunks for delta encoding, as indicated in Fig. 6.

Fig. 11 shows the uncompressing throughput of seven data reduction schemes. Ddelta also outperforms Xdelta and Zdelta in uncompressing speed with or without combining GZ compression. The much lower uncompressing speed of the delta compression approaches than their corresponding decoding speed of Fig. 8(c) is due to the time-consuming stage of reading

Table 6
The workloads of case study II.

Base file	Size (MB)	Updated file	Size (MB)
emacs-22.1.tar	133	emacs-22.2.tar	134
gdb-6.5a.tar	106	gdb-6.6a.tar	106
glibc-2.10.1.tar	99	glibc-2.11.1.tar	101
gcc-4.3.4.tar	387	gcc-4.3.5.tar	387
linux-3.0.10.tar	431	linux-3.0.11.tar	431
scilab-5.0.1.tar	333	scilab-5.0.2.tar	334



(a) Compression ratio. (b) Encoding speed. (c) Decoding speed.

Fig. 12. Delta compression performance of updated tarred files by Xdelta, Deduplication, and Ddelta.

base chunks for delta decoding. Although we have stored the delta compressed data on SSD, the speed of reading the base chunks [4] is still much lower than the delta decoding speed of Fig. 8(c). This is why Ddelta achieves an uncompressing throughput of about 2 GB/s on the Linux and GCC datasets whose delta-compressed-size is about 700MB that is small enough to be cached in RAM by the filesystem. GZ compression does not need to read base chunks and thus provides a stable performance of uncompressing speed of about 200 MB/s, except for the Bench dataset for which GZ compression fails to reduce data.

Based on the results from Figs. 8–11, Ddelta achieves a superior performance of delta encoding and decoding but at a cost of slightly lower compression ratio, which can be supplemented by GZ compression as shown in Fig. 9. The combined Delta and GZ compression achieves a superior performance of data reduction but at the cost of longer processing time. It is in this combination that Ddelta can be an excellent substitute to the Xdelta and Zdelta approaches.

4.4. Case study II: delta compression for updated tarred files

This subsection evaluates the delta compression performance of the updated similar tarred files, namely, the updated files of the six well-known open source projects (available from the websites [34,35,40]), shown in Table 6. We compare Ddelta with Xdelta and Deduplication (i.e., the classic Rabin+SHA1 approach [19]) in Fig. 10. Since we observe that the compression ratio degrades slightly when we apply a larger average string size for Ddelta encoding, both Ddelta and Deduplication use an average string size of 128B for Content-Defined Chunking based delta calculation.

Fig. 12(a) suggests that Ddelta achieves a comparable level of data reduction on the six updated files to Xdelta but detects more redundancy than Deduplication due to the contribution of Ddelta's Step 3 (see Fig. 3) that greedily scans the duplicate-adjacent areas. By using the time-efficient Gear-based chunking and Spooky-based fingerprinting, Ddelta significantly outperforms the classic Deduplication approach, by a factor of 3×, on the delta encoding speed as shown in Fig. 12(b). Fig. 12(c) shows that Ddelta also achieves a superior performance on delta decoding speed to Xdelta, by a factor of 2×–3×, due to Ddelta's CDC based, more efficient encoding/decoding scheme. In this case, resemblance detection is unnecessary and all the base and updated files can be fit in memory, thus Ddelta can provide a much stabler performance of delta compression.

5. Conclusion and future work

In this paper, we present Ddelta, a deduplication-inspired fast delta compression scheme that effectively leverages the principles of deduplication to improve delta encoding and decoding speeds without sacrifice to compression ratio. Our experimental results suggest that Ddelta achieves an encoding speedup of 2.5×–8× and a decoding speedup of 2×–20× over the classic Xdelta and Zdelta approaches, while achieving a comparable level of data reduction. The empirical study of Gear-based chunking shows it improves the Rabin-based Content-Defined Chunking process by a factor of about 2.1× while achieving nearly the same hashing and chunking efficacy for data reduction.

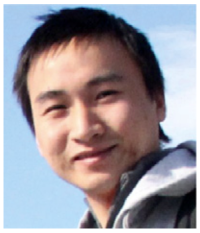
We plan to further improve Ddelta, an on-going project, in its compression ratio and optimize the performance of other stages of delta compression, such as resemblance detection and reading the base-chunks. In addition, we will study and improve data restore performance of the data reduction system that combines deduplication, delta compression, and GZIP compression as a future work.

Acknowledgments

This work was supported in part by National Basic Research 973 Program of China under Grant 2011CB302301; National Science Fund for Distinguished Young Scholars of China under Grant 61025008; National Natural Science Foundation of China (NSFC) under Grants 61173043 and 61232004; The National High Technology Research and Development Program ("863" Program) of China under Grant No. 2013AA013203; US NSF under Grants IIS-0916859, CCF-0937993, CNS-1116606, and CNS-1016609. This work was also supported by Key Laboratory of Information Storage System, Ministry of Education, China.

References

[1] P. Kulkarni, F. Douglis, J. LaVoie, J. Tracey, Redundancy elimination within large collections of files, in: USENIX Annual Technical Conference. USENIX Association, 2004.
[2] D. Gupta, S. Lee, M. Vrable, S. Savage, A.C. Snoeren, G. Varghese, G.M. Voelker, A. Vahdat, Difference Engine: Harnessing memory redundancy in virtual machines, in: Proceedings of the 8th USENIX Symposium on Operating System Design and Implementation. USENIX Association, 2010.
[3] Q. Yang, J. Ren, I-CASH: intelligently coupled array of ssd and hdd, in: Proceedings of IEEE 17th International Symposium on High Performance Computer Architecture (HPCA), IEEE, 2011, pp. 278–289.
[4] P. Shilane, G. Wallace, M. Huang, W. Hsu, Delta compressed and deduplicated storage using stream-informed locality, in: Proceedings of the 4th USENIX conference on Hot Topics in Storage and File Systems. USENIX Association, 2012.
[5] L. Aronovich, R. Asher, E. Bachmat, et al., The design of a similarity based deduplication system, in: SYSTOR 2009: The Israeli Experimental Systems Conference, ACM Association, Haifa, Israel, May 2009, pp. 1–12.
[6] A. Muthitacharoen, B. Chen, D. Mazieres, A low-bandwidth network file system, in: Proceedings of ACM Symposium on Operating Systems Principles, ACM, 2001.
[7] N.T. Spring, D. Wetherall, A protocol-independent technique for eliminating redundant network traffic, ACM SIGCOMM Comput. Commun. Rev. 30 (4) (2000) 87–95.
[8] B. Agarwal, A. Akella, A. Anand, A. Balachandran, P. Chitnis, C. Muthukrishnan, R. Ramjee, G. Varghese, Endre: An end-system redundancy elimination service for enterprises, in: 7th USENIX Symposium on Networked Systems Design and Implementation, NSDI'10, 2010, pp. 419–432.
[9] P. Shilane, M. Huang, G. Wallace, W. Hsu, Wan optimized replication of backup datasets using stream-informed delta compression, in: Proceedings of USENIX Conference on File and Storage Technologies, USENIX Association, 2012.
[10] I. Drago, M. Mellia, M.M. Munafò, A. Sperotto, R. Sadre, A. Pras, Inside dropbox: understanding personal cloud storage services, in: Proceedings of the 12th ACM SIGCOMM Conference on Internet Measurement, ser. IMC'12, 2012, pp. 481–494.
[11] J. Gailly, M. Adler, The gzip compressor, 1991, http://www.gzip.org/.
[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] D. Trendafilov, N. Memon, T. Suel, Zdelta: An efficient delta compression tool, Technical report, Department of Computer and Information Science at Polytechnic University, 2002.
[14] S. Quinlan, S. Dorward, Venti: a new approach to archival storage, in: Proceedings of USENIX Conference on File and Storage Technologies, USENIX Association, 2002.
[15] D.A. Huffman, A method for the construction of minimum-redundancy codes, Proc. Inst. Radio Eng. 40 (9) (1952) 1098–1101.
[16] J. Ziv, A. Lempel, A universal algorithm for sequential data compression, IEEE Trans. Inform. Theory 23 (3) (1977) 337–343.
[17] J. Ziv, A. Lempel, Compression of individual sequences via variable-rate coding, IEEE Trans. Inform. Theory 24 (5) (1978) 530–536.
[18] A. Broder, Some applications of rabin's fingerprinting method, in: Sequences II: Methods in Communications, Security, and Computer Science, 1993.
[19] B. Zhu, K. Li, H. Patterson, Avoiding the disk bottleneck in the data domain deduplication file system, in: Proceedings of USENIX Conference on File and Storage Technologies, USENIX Association, 2008.
[20] G. Wallace, F. Douglis, H. Qian, P. Shilane, S. Smaldone, M. Chamness, W. Hsu, Characteristics of backup workloads in production systems, in: Proceedings of USENIX Conference on File and Storage Technologies, USENIX Association, 2012.
[21] I. Drago, E. Bocchi, M. Mellia, H. Slatman, A. Pras, Benchmarking personal cloud storage, in: Proceedings of the 13th ACM Internet Measurement Conference, ser. IMC'13, 2013.
[22] H. Slatman, Opening up the sky: a comparison of performance-enhancing features in skydrive and dropbox, in: Proceedings of the 18th Twente Student Conference on IT, 2013.
[23] E. Kruus, C. Ungureanu, C. Dubnicki, Bimodal content defined chunking for backup streams, in: Proceedings of USENIX Conference on File and Storage Technologies, USENIX Association, 2010.
[24] W. Xia, H. Jiang, D. Feng, Y. Hua, SiLo: A Similarity-Locality based Near-Exact Deduplication Scheme with Low RAM Overhead and High Throughput, in: Proceedings of the 2011 conference on USENIX Annual technical conference, USENIX Association, 2011.
[25] J.-l. Gailly, M. Adler, Zlib compression library, 2004.
[26] C.E. Leiserson, R.L. Rivest, C. Stein, T.H. Cormen, Introduction to Algorithms, The MIT press, 2001.
[27] E.K.C.U. Cezary Dubnicki, Krzysztof Lichota, Methods and systems for data management using multiple selection criteria, 2008, US Patent 7,844,581.
[28] W. Xia, H. Jiang, D. Feng, L. Tian, M. Fu, Z. Wang, P-dedupe: exploiting parallelism in data deduplication system, in: Networking, Architecture and Storage (NAS), 2012 IEEE 7th International Conference on, IEEE, 2012, pp. 338–347.
[29] W. Xia, H. Jiang, D. Feng, L. Tian, Combining deduplication and delta compression to achieve low-overhead data reduction on backup datasets, in: Data Compression Conference, DCC, IEEE, 2014, pp. 203–212.
[30] J. Bob, Spookyhash: a 128-bit noncryptographic hash, 2010.
[31] A. Appleby, Murmurhash 2.0, 2008.
[32] Y. Collet, xxhash - fast hash algorithm, 2012.
[33] C. Bienia, S. Kumar, J.P. Singh, K. Li, The parsec benchmark suite: characterization and architectural implications, in: Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques, ACM, 2008, pp. 72–81.
[34] Gnu archives, http://ftp.gnu.org/gnu/.
[35] Linux archives, ftp://ftp.kernel.org/.
[36] VMs archives, http://www.thoughtpolice.co.uk/vmware//.
[37] Redis key-value database, http://redis.io/.
[38] M. Lillibridge, K. Eshghi, D. Bhagwat, Improving restore speed for backup systems that use inline chunk-based deduplication, in: Proceedings of USENIX Conference on File and Storage Technologies, 2013.
[39] V. Tarasov, A. Mudrankit, W. Buik, P. Shilane, G. Kuenning, E. Zadok, Generating realistic datasets for deduplication analysis, in: Proceedings of the 2012 Conference on USENIX Annual Technical Conference, USENIX Association, 2012.
[40] Scilab archives, http://www.scilab.org/.



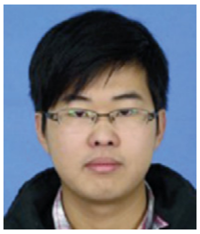
Wen Xia received his Ph.D. degree in Computer Science from Huazhong University of Science and Technology (HUST), Wuhan, China in June, 2014. His current research interests include data deduplication, delta compression, and backup storage system. He has more than 8 publications in major journals and conferences including IEEE Transactions on Computers (TC), USENIX ATC, IEEE DCC, IEEE NAS, ICA3PP, etc.

Hong Jiang received the B.Sc. degree in Computer Engineering in 1982 from Huazhong University of Science and Technology (HUST), Wuhan, China; the M.A.Sc. degree in Computer Engineering in 1987 from the University of Toronto, Toronto, Canada; and the Ph.D. degree in Computer Science in 1991 from the Texas A&M University, College Station, Texas, USA. Since August 1991 he has been at the University of Nebraska-Lincoln, Lincoln, Nebraska, USA, where he is Willa Cather Professor of Computer Science and Engineering. His present research interests include computer architecture, computer storage systems and parallel I/O, high-performance computing, big data computing, cloud computing, and performance evaluation. He serves as an Associate Editor of the IEEE Transactions on Parallel and Distributed Systems. He has over 200 publications in major journals and international Conferences in these areas, including IEEE-TC, IEEE-TPDS, ACM-TACO, JPDC, ISCA, MICRO, USENIX ATC, FAST, LISA, ICDCS, IPDPS, MIDDLEWARE, OOPLAS, ECOOP, SC, ICS, HPDC, ICPP. He is a senior member of IEEE, a member of ACM.

Dan Feng received the B.E., M.E., and Ph.D. degrees in Computer Science and Technology in 1991, 1994, and 1997, respectively, from Huazhong University of Science and Technology (HUST), China. She is a professor and vice dean of the School of Computer Science and Technology, HUST. Her research interests include computer architecture, massive storage systems, and parallel file systems. She has more than 100 publications in major journals and international conferences, including IEEE-TC, IEEE-TPDS, ACM-TOS, JCST, FAST, USENIX ATC, ICDCS, HPDC, SC, ICS, IPDPS, and ICPP. She servers as the program committees of multiple international conferences, including SC 2011, 2013 and MSST 2012. She is a member of IEEE and a member of ACM.

Lei Tian received his Ph.D. degree in Computer Engineering from Huazhong University of Science and Technology (HUST) in 2010. He is currently a Research Assistant Professor in the Department of Computer Science and Engineering at the University of Nebraska-Lincoln. His research interests mainly lie in storage systems, distributed systems, cloud computing and big data. He has over 30 publications in major journals and conferences including FAST, HOTSTORAGE, ICS, SC, HPDC, ICDCS, MSST, MASCOTS, ICPP, IPDPS, CLUSTER, IEEE TC, IEEE TPDS, ACM TOS, etc.

Min Fu is currently a Ph.D. student majoring in computer architecture at HuaZhong University of Science and Technology (HUST), China. His research interests include data deduplication, key–value store and emerging storage technologies, etc.



Yukun Zhou is currently a Ph.D. student majoring in computer architecture at HuaZhong University of Science and Technology (HUST), China. His research interests include data deduplication and storage security.