A Fast Asymmetric Extremum Content Defined Chunking Algorithm for Data Deduplication in Backup Storage Systems

Yucheng Zhang, Dan Feng, *Member, IEEE*, Hong Jiang, *Fellow, IEEE*, Wen Xia, *Member, IEEE*, Min Fu, Fangting Huang, and Yukun Zhou

Abstract—Chunk-level deduplication plays an important role in backup storage systems. Existing Content-Defined Chunking (CDC) algorithms, while robust in finding suitable chunk boundaries, face the key challenges of (1) low chunking throughput that renders the chunking stage a serious deduplication performance bottleneck, (2) large chunk size variance that decreases deduplication efficiency, and (3) being unable to find proper chunk boundaries in low-entropy strings and thus failing to deduplicate these strings. To address these challenges, this paper proposes a new CDC algorithm called the Asymmetric Extremum (AE) algorithm. The main idea behind AE is based on the observation that the extreme value in an asymmetric local range is not likely to be replaced by a new extreme value in dealing with the boundaries-shifting problem. As a result, AE has higher chunking throughput, smaller chunk size variance than the existing CDC algorithms, and is able to find proper chunk boundaries in low-entropy strings. The experimental results based on real-world datasets show that AE improves the throughput performance of the state-of-the-art CDC algorithms by more than $2.3\times$, which is fast enough to remove the chunking-throughput performance bottleneck of deduplication, and accelerates the system throughput by more than 50 percent, while achieving comparable deduplication efficiency.

Index Terms—Storage systems, data deduplication, content-defined chunking algorithm, performance evaluation

1 Introduction

According to a study of International Data Corporation (IDC), the amount of digital information generated in the whole world is about 4.4 ZB in 2013, and that amount will reach 44 ZB by 2020 [1]. How to efficiently store and transfer such large volumes of digital data is a challenging problem. However, recent work reveals the existence of a large amount of duplicate data in storage systems [2]. As a result, data deduplication, a space- and bandwidth-efficient lossless compression technology that prevents redundant data from being stored in storage devices and transmitted over the networks, is one of the most important methods to tackle this challenge. Due to its significant data reduction efficiency, chunk-level deduplication is used in various fields, such as storage systems [2], Redundancy Elimination (RE) in networks [3], file-transfer systems (rsync [4]) and remote-file systems (LBFS [5]).

Chunk-level deduplication schemes split the data stream into multiple chunks, hash each chunk to generate a digest, called a fingerprint, with a secure hash function (such as SHA-1 or MD5). Two chunks with identical fingerprints are considered duplicates. By checking fingerprints, duplicate chunks can be identified and removed, only unique chunks whose fingerprints do not find a match are stored. Therefore, deduplication is considered to consist of four stages, namely chunking, fingerprinting, indexing and writing. As the first and key stage in the deduplication workflow, the chunking stage is responsible for dividing the data stream into chunks of either fixed size or variable size, which is decided by the chunking algorithm applied. Fixed-Size Chunking (FSC) algorithm marks chunks' boundaries by their positions and generates fixed-size chunks. This method is simple and extremely fast, but it suffers from low deduplication efficiency that stems from the boundary-shifting problem. For example, if one byte is inserted at the beginning of an data stream, all current chunk boundaries declared by FSC will be shifted and no duplicate chunks will be identified and eliminated. Content-Defined Chunking (CDC) algorithm divides the data stream into variable-size chunks. It solves the boundary-shifting problem by declaring chunk boundaries depending on local content of the data stream. If the local content is not changed, the chunks' boundaries will not be shifted. As a result, the CDC algorithm outperforms the FSC algorithm in terms of deduplication efficiency and has been widely used in various fields including storage systems. To provide the necessary basis to facilitate the discussion of and comparison among different CDC algorithms, we list below some key properties that a desirable CDC algorithm should have [6].

1) *Content defined*. To avoid the loss of deduplication efficiency due to the boundaries-shifting problem,

- *Y. Zhang, D. Feng, W. Xia, M. Fu, F. Huang, and Y. Zhou are with Wuhan National Laboratory for Optoelectronics, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, Hubei 430074, China. E-mail: hust.yczhang@gmail.com, {dfeng, xia, fumin, huangfangting, ykzhou}@hust.edu.cn.*
- *H. Jiang is with the Department of Computer Science and Engineering, University of Texas at Arlington, 640 ERB, 500 UTA Blvd, Arlington, TX 76019. E-mail: hong.jiang@uta.edu.*

Manuscript received 21 Jan. 2016; revised 13 June 2016; accepted 6 July 2016. Date of publication 27 July 2016; date of current version 20 Jan. 2017. Recommended for acceptance by Z. Shao.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TC.2016.2595565

the algorithm should declare the chunk boundaries based on local content of the data stream.

2) *Low computational overhead.* CDC algorithms need to check almost every byte in an data stream to find the chunk boundaries. This means that the algorithm execution time is approximately proportional to the number of bytes of the data stream, which can take up significant CPU resources. Hence, in order to achieve higher deduplication throughput, the chunking algorithm should be simple and devoid of time-consuming operations.
3) *Small chunk size variance.* The variance of chunk size has a significant impact on the deduplication efficiency. The smaller the variance of the chunk size is, the higher the deduplication efficiency will be achieved [7].
4) *Efficient for low-entropy strings.* The content of real data may sometimes include low-entropy strings [8]. These strings include very few distinct characters but a large amount of repetitive bytes. In order to achieve higher deduplication efficiency, it is desirable for the algorithm to be capable of finding proper chunk boundaries in such strings and eliminating them as many as possible.
5) *Less artificial thresholds on chunk size.* Minimum and maximum thresholds are often imposed on chunk size to avoid chunks being too short or too long. These measures reduce chunk size variance, but also make the chunk boundaries position-dependent and thus not truly content-defined, which also reduces the deduplication efficiency [9].

The Rabin fingerprint [10] based CDC algorithm (Rabin) is widely employed in deduplication systems [11], [12]. The main problems of the Rabin algorithm are its low chunking throughput, which renders the chunking stage the performance bottleneck of the deduplication workflow [13], [14], and large chunk size variance that lowers the deduplication efficiency [7]. MAXP [15] is a CDC approach that addresses the chunk-size variance problem of Rabin by treating the local extreme values as cut-points. Owing to its smaller chunk size variance and lower memory overhead than Rabin, MAXP was recommended to be used in redundancy elimination in networks [16], [17]. MAXP slides a fix-sized symmetric window over the byte stream on a byte-by-byte basis, and checks whether the value of the byte at the center of the current window is the extreme value in the window. The byte found to be the extreme value is declared a cut-point (chunk boundary). This strategy of finding the local extreme values dictates that the MAXP algorithm recheck some previously compared bytes in the reverse direction of the stream, which significantly lowers its chunking throughput.

We propose the Asymmetric Extremum chunking algorithm (AE), a new CDC algorithm that significantly improves the chunking throughput of the above existing algorithms while providing comparable deduplication efficiency by using the local extreme value in a variable-sized asymmetric window to overcome the aforementioned boundary-shifting problem. With a variable-sized asymmetric window, instead of a fix-sized symmetric window as in MAXP, the AE algorithm finds the extreme value in the window without having to backtrack and thus requiring only one comparison and two conditional branch operations per byte scanned. Therefore, AE's simplicity makes it very fast. It also has smaller chunk size variance than existing CDC algorithms and imposes no limitation on chunk size. Moreover, AE is able to eliminate more low-entropy strings than the other algorithms. To further the advantages of AE, we propose two optimizations to AE, with one optimization providing higher chunking throughput while the other offering higher deduplication efficiency by eliminating more low-entropy strings. Our experimental evaluations based on three real-world datasets show that AE improves the throughput performance of the state-of-the-art CDC algorithms by at least $2.3\times$ and accelerates the overall deduplication system throughput performance by more than 50 percent, while attaining comparable deduplication efficiency. More specifically, the optimization for higher chunking throughput speedups the performance by 56.9 and 73.1 percent on the i7-930 and i7-4770 CPUs respectively. The optimization to detect more low-entropy strings eliminates on average $8.8\times$ and $2.09\times$ more low-entropy strings than the Rabin and the original AE algorithms respectively, with only slight and no reduction in system throughput compared with the system using AE on the i7-930 and i7-4770 CPUs respectively.

The rest of paper is organized as follows. In Section 2, we present the background and motivation. We describe the detailed design and implementation of our AE and analyze some of its key properties in Section 3. We present the experimental setup and evaluation results in Section 4 and conclude the paper in Section 5.

2 Background and Motivation

In this section, we first provide the necessary background for the AE research by introducing the challenges facing the existing CDC algorithms, and then motivate our research by analyzing our key observations.

2.1 Background

The Rabin fingerprint [10] based CDC algorithm (Rabin) was first used to eliminate redundant network traffic [18]. It runs a sliding-window hash along the byte stream, declaring a chunk boundary whenever the k-lowest-order bits of the hash are equal to a pre-determined value. Though being efficient in dealing with boundaries-shifting problem, Rabin suffers from two major drawbacks, namely, its time-consuming fingerprint computation that results in low chunking throughput and its large chunk size variance that reduces deduplication efficiency. Moreover, Rabin is not capable of finding proper chunk boundaries in low-entropy strings (either all or none of the positions in the low-entropy strings will be considered as chunk boundaries). This, combined with the drawback of large chunk size variance, makes Rabin having to impose a minimum and a maximum threshold on chunk size to avoid chunks being too short or too long, since very short chunks imply more fingerprints to be stored and processed by deduplication systems and thus not cost-effective, while very long chunks reduce the deduplication efficiency.

Recognizing the impact of the chunk-size variance on deduplication efficiency, Eshghi et al. [7] proposed the TTTD algorithm to reduce Rabin's chunk-size variance. The

TABLE 1
Properties of the State-of-the-Art CDC Algorithms

Properties	SampleByte	Rabin	MAXP	AE
Content Defined	Yes	Yes	Yes	Yes
Computational overheads	Low	High	High	Low
Chunk size variance	High	High	Low	Low
Efficient for low-entropy strings	No	No	No	Yes
Artificial thresholds on chunk size	Yes	Yes	No	No

TTTD algorithm introduces an additional backup divisor that has a higher probability of finding cut-points. When it fails to find a cut-point using the main divisor within a maximum threshold, it returns the cut-point found by the backup divisor, if any. If no cut-point is found by either of the divisors, it returns the maximum threshold. In fact, TTTD is not a new chunking algorithm but an optimization strategy of Rabin to reduce the chunk-size variance. This strategy can also be applied to the other chunking algorithms. There are some other optimization strategies that are similar to TTTD, for example, Regression chunking algorithm [9] uses multiple divisors to reduce forced cut-points declaration at maximum chunk size.

MAXP [15], [16] is a state-of-the-art CDC algorithm, which is first used in remote differential compression of files. Unlike Rabin that must compute a hash first, the MAXP algorithm treats the bytes directly as digits, which helps reduce the computational overhead. MAXP attempts to find the strict local extreme values in a fixed-size symmetric window, and then uses these points as chunk boundaries to divide the input stream. The main disadvantages of this strategy is that when declaring an extreme value, the algorithm must move backwards by a fixed-size region to check if there is any value greater (if the extreme value is the maximum value) than the value of the current position being examined. This backtracking process requires many extra conditional branch operations and increases the number of comparison operations for each byte examined. Since MAXP needs to check every byte in the data stream, any additional conditional branch operations result in a decreased chunking throughput.

EndRE [17] proposes an adaptive SampleByte algorithm for declaring fingerprints. The SampleByte algorithm combines the CDC algorithm's robustness to small changes in content with the efficiency of the FSC algorithm. It uses one byte to declare a fingerprint and stores $1/p$ representative fingerprints for content matching, where p is the sampling period. To avoid over-sampling, it skips $p/2$ bytes when a fingerprint has been found. SampleByte is fast since it (1) only needs one conditional branch per byte to judge the chunk boundaries and (2) skips about one third of bytes on the input data. However, the design principle of SampleByte dictates that the sampling period p be smaller than 256 Bytes, which means that the expected average chunk size must be smaller than 256 Bytes when used in the chunk-level deduplication. Unfortunately, a chunk granularity of 256 bytes or smaller is too fine to be cost efficient or practical, which makes the SampleByte algorithm inappropriate for coarse-grained chunk-level deduplication in backup systems. Moreover, it requires a lookup table that needs to be generated before deduplication according to the workload characteristics, which further restricts its applications.

Some other Rabin-based CDC variants are also proposed to improve the deduplication efficiency but introduce more computation overhead for chunking. For example, FBC [19] re-chunks the chunks which contains subchunks with high frequency to detect more finer-grained redundancy. Bimodal chunking [11] divides the data stream into large chunks, and then re-chunks the non-duplicate but duplicate-adjacent chunks into smaller chunks to eliminate more redundancy. Subchunk [20] is similar to Bimodal chunking, but it re-chunks all of the non-duplicate chunks for higher deduplication efficiency. MHD [21] dynamically merges consecutively duplicate chunks into a big chunk to reduce the chunk fingerprints that need to be stored. Fingerdiff [22] divides the data stream into small chunks and merges consecutively duplicate or unique chunks into big chunks to reduce the metadata overhead.

2.2 Challenges and Motivation

Chunk-level deduplication is faster than the traditional compression technology, such as GZip [23]. Further, because of its coarse-grained compression scheme, it achieves a higher compression ratio than the traditional compression technology in backup systems [24]. For certain backup datasets, the ratio of deduplication efficiency of chunk-level deduplication to that of GZip-only compression is 15:2 [19], [25]. Therefore, chunk-level deduplication is more commonly used in backup storage systems. Moreover, because of their higher deduplication efficiency than FSC [26], CDC algorithms are preferred in chunk-level deduplication. However, the low chunking throughput of the existing CDC algorithms hinders their wider applications because of the deduplication performance bottleneck [14], [27]. To alleviate the performance bottleneck and increase the throughput of the deduplication system, P-Dedupe [14] harnesses the idle CPU resources to pipeline and parallelize the compute-intensive processes. StoreGPU [27] and Shredder [13] exploit underutilized GPU resources to improve the chunking throughput. However, these schemes achieve their performance improvement from either additional resources or parallelization of the deduplication processes, but not from improving the chunking algorithm itself.

Table 1 compares the state-of-the-art CDC algorithms by summarizing their key properties. The SampleByte algorithm is confined to the fine-grained deduplication (the expected average chunk size should be smaller than or equal to 256 Bytes), and thus is not applicable to backup systems. The Rabin algorithm has the problems of high computational overheads and high chunk size variance. The MAXP algorithm is computationally expensive and cannot eliminate low-entropy strings. Obviously, the Rabin and MAXP algorithms share the common problem of high computational overhead that results in low chunking throughput.

As indicated by P-dedupe [14], the workflow of CDC-based deduplication consists of four independent stages of chunking, fingerprinting, indexing, and writing. Therefore, to make full use of the computation resources in multicore- or manycore-based computer systems and accelerate the throughput of deduplication system, some deduplication systems, such as ZFS [28], THCAS [29], Data Domain's file system [30], and P-dedupe, pipeline the deduplication workflow. As a result, the stage that has the lowest throughput performance will be the performance bottleneck of the

TABLE 2
Traditional Chunking and Fingerprinting Throughput on the i7-930 and i7-4770 CPUs

i7-930		i7-4770	
Rabin_0.25	sha-1	Rabin_0.25	sha-1
354 MB/s	526 MB/s	463 MB/s	888 MB/s

whole deduplication workflow. In the four stages of the workflow, the throughput of the writing stage is decided by two factor: deduplication efficiency and storage device. The amount of data needs to be stored is decided by the deduplication efficiency. The higher the deduplication efficiency is, the less the amount of data to be written.

Another factor is the storage devices, different storage devices have different sequential writing throughput, including some enterprise class storage devices that can easily achieve a throughput of several GB per second. As for the indexing stage, its throughput is much higher than the chunking and fingerprinting stages when chunk size is a few KB [31], [32]. As a result, the remaining two stages, namely, the chunking and fingerprinting stages, have the lowest throughput. We evaluate the average throughput of the chunking and fingerprinting stages in the i7-930 and i7-4770 CPUs, with evaluation results summarized in Table 2 where Rabin_0.25 and SHA-1 are the most common methods used in the chunking and fingerprinting stages respectively of today's deduplication systems. Without consideration of writing performance, we can draw the following observations from this table. First, chunking is the main performance bottleneck of the deduplication workflow since its throughput is much lower than the fingerprinting throughput. Second, the large difference between the chunking throughput and the fingerprinting throughput means that the improvement of system throughput will be significant if the chunking-throughput performance bottleneck is removed.

The various problems facing the state-of-the-art CDC algorithms summarized in Table 1, particularly that of the low throughput, stem from the high computation overheads and motivate us to propose a new chunking algorithm with low computation overhead to overcome these problems. In fact, our experimental observation finds that detecting local extreme values in an asymmetric window can not only deal with the boundaries-shift problem for Content-Defined Chunking, but also increase the chunking throughput and detect more low-entropy strings. As a result, our proposed Asymmetric Extremum chunking algorithm, by using an asymmetric window to find the local extreme value for chunking as elaborated in the next section, is able to better satisfy the key desirable properties of CDC algorithm to achieve high deduplication efficiency, and remove the chunking-throughput performance bottleneck of deduplication and improve performance of the whole deduplication system.

3 Asymmetric Extremum Chunking Algorithm

In this section we describe the design of the AE chunking algorithm and analyze its key properties.

3.1 The AE Algorithm Design

In AE, a byte has two attributes: position and value. Each byte in the data stream has a position number, and the position of the nth byte ($1 \leqslant n \leqslant stream\ length$) in the stream is n. Each interval of S consecutive characters/bytes in the data stream is treated as a value. For example, eight consecutive characters/bytes are converted to a value that is a 64-bit integer. The value of every such interval in the data stream is associated with the position of the first byte of the S consecutive characters/bytes that constitute this value. Therefore, each byte in the stream, except for the very last $S-1$ bytes, has a value associated with it. For convenience of description, we assume that data stream start from the leftmost byte. If a byte A is on the left of byte B, A is said to be $before\ B$, and B appears $after\ A$. Given a byte P in the data stream, the w consecutive bytes immediately after P are defined to be the *right window* of P, and w is referred to as the *window size*.

The extreme value in the AE algorithm can be either the maximum value or the minimum value. For convenience of discussion, in what follows in this section, we assume that the extreme value is the maximum value. Starting from the very first byte of the stream or the first byte after the last cut-point (chunk boundary), AE attempts to find the first byte of the data stream that satisfies the following two conditions.

- It is the first byte or its value is greater than the values of all bytes $before$ it.
- Its value is *not less than* the values of all bytes in its $right\ window$.

The first byte found to meet these conditions is referred to as a *maximum point*. These two conditions make sure that the maximum point has the maximum value in the region from the very first byte of the stream or the first byte after the last cut-point to the rightmost byte of the maximum point's $right\ window$. There are two further implications. First, this first byte can be a maximum point. Second, AE allows for ties between the byte being examined and bytes in its $right\ window$. If a maximum point has been found, AE returns the rightmost byte in its $right\ window$, which is also the byte being processed, as a cut-point (chunk boundary). AE does not need to backtrack, since the process after the returned cut-point is independent of the content before the cut-point. Moreover, the position and the value of the bytes processed (except for the byte having the temporary local maximum value) need not be kept in memory. The workflow of AE is described in Fig. 1. Algorithm 1 below provides a more detailed implementation of the AE chunking algorithm.

From the algorithm description above, we know that the minimum chunk size of AE is $w+1$. In what follows we discuss the expected chunk size of AE.

Theorem 1. *Consider a byte in position p in the current data stream, starting from the first byte after the last cut point (i.e., excluding the bytes in the data stream that have already been chunked), the probability of this byte being a maximum point is $1/(w+p)$, where w is the window size.*

Proof. We assume that the content of real data is random, an assumption that is reasonably justified by our experimental evaluation and previous work such as MAXP [15].

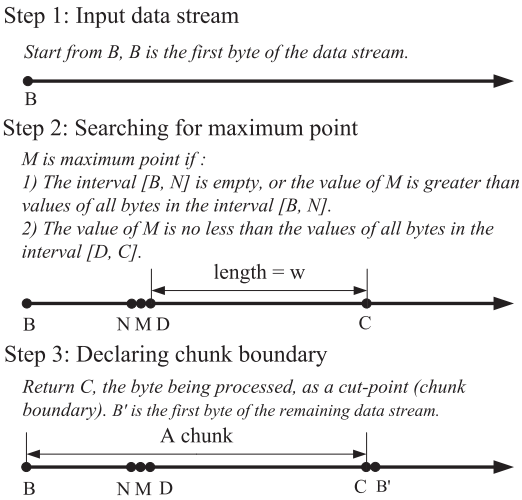


Fig. 1. The workflow of the AE chunking algorithm, where N, M, D in the figure are neighboring positions, and so are the positions of C and B′.

According to the conditions set for a maximum point, if byte p is a maximum point, it should have the maximum value in the interval $[1, p+w]$. In this interval, each byte is equally likely to be of the maximum value. Thus, the probability of byte p being the maximum point is $1/(w+p)$. Note that if position p is the maximum point, the chunk size will be $p+w$. □

Now we determine the range of possible position of the maximum point, namely, the position x that makes the cumulative probability equal to 1. According to Theorem 1, we can compute the position x using the following equation:

$$\frac{1}{w+1}+\frac{1}{w+2}+\frac{1}{w+3}+\cdots+\frac{1}{w+x}=1.$$

The left side of this equation is approximately equal to $ln(w+x)-lnw$. Thus, the value of x is approximately $(e-1)\times w$. For each possible position, the expected chunk size is equal to the probability of being the maximum point multiplying by the chunk size if it is the maximum point. Finally, we compute the expected chunk size by adding the expected chunk size of all possible positions, and the result, namely, the expected chunk size, is $(e-1)\times w$.

3.2 Properties of the AE Algorithm

In this section, we analyze the AE algorithm in regards to the desirable properties of CDC algorithms listed in Section 1.

Content Defined. The MAXP algorithm considers a byte with the local maximum value a chunk boundary. Therefore, any modifications within a chunk, as long as they do not replace the local maximum value, will not affect the adjacent chunks, since the chunk boundaries will simply be re-aligned. Unlike MAXP, the AE algorithm returns the wth position after the maximum point as the chunk boundary. It puts the maximum points inside the chunks instead of considering them as chunk boundaries. This strategy may slightly decrease the deduplication efficiency, but AE is still content defined since the maximum points inside the chunks can also re-align the chunk boundaries.

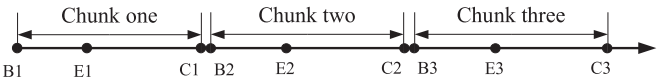


Fig. 2. An example of efficiency loss of AE.

Take Fig. 2 for example, E1, E2, E3 are the three maximum points, C1, C2, C3 are the cut-points of the three corresponding chunks. Assume that all modifications in the example will not replace the local maximum value. If there is an insertion (or deletion) in the interval [B1, E1) in Chunk 1, Chunk 2 will not be affected since the chunk boundary will be re-aligned by the maximum point E1. If the insertion is in the interval (E1, C1], the starting point of Chunk 2 will be changed, and E2 will re-align the boundary to keep Chunk 3 from being affected. If a sequence of consecutive chunks has been modified, the loss of efficiency is determined by the position of the modification in the last modified chunk. If the modification is *before* the maximum point, there is no efficiency loss. Otherwise, only one duplicate chunk that is immediately after this modified region will be affected. In addition, the deduplication efficiency is also determined by many other factors, such as chunk-size variance and the ability to eliminate low-entropy strings. As we will see shortly, AE's ability to eliminate low-entropy strings and reduce chunk-size variance has more than compensated for this relatively small loss of deduplication efficiency.

Algorithm 1. Algorithm of AE Chunking

Input: input string, *Str*; left length of the input string, L;
Output: chunked position (cut-point), i;

```
1:  Predefined values: window size w;
2:  function AECHUNKING (Str, L)
3:      i ← 1
4:      max.value ← Str[i].value
5:      max.position ← i
6:      i ← i + 1
7:      while i < L do
8:          if Str[i].value ≤ max.value then
9:              if i = max.position + w then
10:                 return i
11:             end if
12:         else
13:             max.value ← Str[i].value
14:             max.position ← i
15:         end if
16:         i ← i + 1
17:     end while
18:     return L
19: end function
```

Computational Overheads. Table 3 shows the computational overheads of the three algorithms, AE, MAXP and Rabin. As shown in the table, the Rabin algorithm needs 1 OR, 2 XORs, 2 SHIFTs and 2 ARRAY LOOKUPs per byte examined to compute the fingerprints and one conditional branch to judge the chunk boundaries. While both the MAXP and AE algorithms use comparison operations to find the local maximum values, their strategies are quite different and it is this difference that makes AE much faster than MAXP. Fig. 3 shows the difference between the two

TABLE 3
Computational Overheads of the Three Algorithms

Algorithm	Computational overhead per byte scanned
Rabin	1 or, 2 xors, 2 shifts, 2 array lookups, 1 conditional branch
MAXP	2 mod, $2-\frac{1}{p}$ comparisons, $5+\frac{1}{p}$ conditional branches
AE	1 comparison, 2 conditional branches

p is the expected chunk size.

algorithms. As shown in the figure, MAXP finds the maximum values in a fixed-size window [A, D]. If the byte M that is in the center of this window has the maximum value in the window, its value must be strictly greater than that of any byte in both regions of [A, B] and [C, D]. Assuming that all bytes in the window [A, D] have been scanned and M has the maximum value and has been returned as a cut-point, some of the bytes in region [C, D] must be scanned again when MAXP processes the byte E. This means that MAXP needs an array to store the information of the bytes in the fixed-size region immediately before the current byte. Therefore, it requires two modular operations to update the array, and $2-\frac{1}{p}$ comparison and $5+\frac{1}{p}$ conditional branch operations to find the local maximum value.

In contrast, AE only needs to find the maximum value in an asymmetric window [F, I], which includes a fixed-size region [H, I] and a variable-size region [F, G], whose size is determined by the content of the data stream. As a result, we only need to store a candidate maximum point and the position of the candidate maximum point, and do not need to backtrack to declare the local maximum value. Therefore, AE only needs one comparison and two conditional branch operations. Clearly, AE requires much fewer operations, particularly the time-consuming conditional branch and table lookup operations, than the other two algorithms.

Chunk Size Variance. Here we analyze the chunk size variance of the AE algorithm. We use the probability of a long region not having any cut-point to estimate the chunk size variance.

Theorem 2. *AE has no maximum point in a given range, if and only if in each interval of w consecutive bytes in this range, there exists at least one byte that satisfies the first condition of the maximum point, namely, it is the first byte or its value is greater than the values of all bytes before it.*

Proof. In this range, if there exists one byte in each interval of w consecutive bytes whose value is greater than the values of all bytes before it, then the second condition of the maximum point, namely, its value is *not less than* the values of all bytes in its right window, will never be satisfied. In other words, there is no maximum point in the range. □

Given an interval $[cw+a+1, cw+a+w]$, where c is a constant, the probability of no byte satisfying the first condition of maximum point is

$$\prod_{i=1}^{w}\left(1-\frac{1}{a+i}\right)=\frac{a}{a+w}.$$

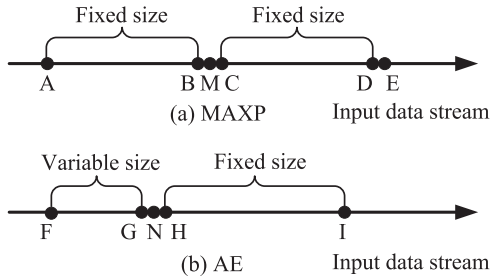


Fig. 3. Illustration of the key difference between the MAXP and AE algorithms, where B, M, C are neighboring positions, so are the positions of D and E and positions of G, N, and H.

So the complementary probability, that there exists at least one byte satisfying the first condition of the maximum point, is $w/(w+a)$. Divide the interval into subintervals with the length of w and then number them from 1 to m. Consider the pth subinterval $[(p-1)w+1, pw]$. The probability of no maximum point in it is

$$\frac{w}{(p-1)\times w+w}=\frac{1}{p}.$$

Multiplying the probabilities of the continuous m subintervals, we have $\frac{1}{m!}$. Given that the average chunk size of AE is $(e-1)\times w$, the probability of no maximum point in m consecutive chunks of average chunk size becomes

$$P(AE)=\frac{1}{[(e-1)\times m]!},$$

here m should be more than 1.

Next we compare the probabilities of very long chunks among the AE, MAXP and Rabin algorithms. Table 4 shows formulas to calculate the theoretical probability of no cut-points in a region of length $m\times average\text{-}chunk\text{-}size$ [15] and lists such probabilities when $m=2,3,\ldots,8$ for the three algorithms, where Rabin_0 represents the Rabin algorithm without minimum threshold, and Rabin_0.25 represents Rabin with a minimum threshold on chunk size, and the ratio of the minimum threshold to the expected chunk size is 0.25. As can be seen from the table, the probability of generating exceptionally long chunks by AE is much lower than the other two algorithms, which also means that AE has smaller chunk-size variance.

Dealing with Low-Entropy Strings. Ties between the byte being examined and the bytes in the *right window* may

TABLE 4
Probability of No Cut-Points in a Region of Length $m\times Average\text{-}Chunk\text{-}Size$

m	AE	Rabin_0	Rabin_0.25	MAXP
	$\frac{1}{[(e-1)\times m]!}$	e^{-m}	$e^{-1.2m}$	$\frac{2^{2m}}{(2m)!}$
2	0.0938	0.1353	0.0907	0.6667
3	0.0064	0.0498	0.0273	0.0889
4	2.56×10^{-4}	0.0183	0.0082	0.0063
5	6.85×10^{-6}	0.0067	0.0025	2.82×10^{-4}
6	1.32×10^{-7}	0.0025	7.47×10^{-4}	8.55×10^{-6}
7	1.94×10^{-9}	9.12×10^{-4}	2.25×10^{-4}	1.88×10^{-7}
8	2.25×10^{-11}	3.35×10^{-4}	6.77×10^{-5}	3.13×10^{-9}

appear in the data stream. If a tie happens to be between two local maximum values, we can break the tie by one of the following two strategies: (1) selecting the first maximum value or (2) going beyond the right window to search for a strictly maximum value. Strategy (1) can help identify and eliminate low-entropy strings. AE allows for ties in its right window and the maximum point can be the first byte, so that it can divide the low-entropy strings into fixed-size chunks whose size is $w+1$. On the other hand, Strategy (2), which is used in the MAXP algorithm, will lead the algorithm to miss detecting and eliminating low-entropy strings. Note that the AE algorithm cannot detect all low-entropy strings. If the length of a low-entropy string is greater than $2w+2$, then AE can identify a part of it. Furthermore, Strategy (2) requires more conditional branch operations in finding the maximum points. For these reasons, we chose Strategy (1) for AE.

Artificial Thresholds on Chunk Size. AE's strategy of finding the maximum values implies that the length of its chunk will be greater than or equal to $w+1$, so that an artificial minimum threshold on chunk size is unnecessary. In addition, according to Table 4, the probability of AE generating exceptionally long chunks is extremely small. This, combined with the fact that AE is able to find cut-points in low-entropy strings, makes it unnecessary for AE to impose the maximum threshold on chunk size, a point that is amply demonstrated in the detailed sensitivity study of the AE algorithm in the next section.

3.3 Optimizations

The First Optimization. In fact, AE uses the comparison operation to find the extreme point, which provides us with an opportunity to optimize the algorithm. As shown in Algorithm 1, every time the algorithm moves forward one byte, the value of the new byte $Str[i].value$ must be compared with the temporary maximum value $max.value$. However, $max.value$, which is selected out of the values of all bytes between the starting point and the current position i, is expected to be greater than $Str[i].value$. In other words, lines 8 through 10 in the Algorithm 1 are expected to be executed much more frequently than lines 12 through 14, which means that optimizing lines 8 through 10 in the algorithm will significantly improves the chunking throughput. We found that declaring boundary by Algorithm 1 (line 9) is unnecessary in the region between $max.position$ and $max.position+w$ due to AE's strategy of declaring boundaries. Therefore, we optimize the algorithm by removing as many of the boundary declaring operations as possible in the region mentioned above as described in Algorithm 2. For simplicity, we use $max.v$ and $max.pos$ instead of $max.value$ and $max.position$ in Algorithm 2. This optimization helps increase the chunking throughput since it only needs one comparison and one conditional branch in the region between $max.position$ and $max.position+w$ when $max.value$ is greater than $Str[i].value$, rather than one comparison and two conditional branches required by the original AE algorithm. Note that, while the chunking stage is not the performance bottleneck in the deduplication workflow, higher chunking throughput translates into less computational overheads and thus frees up more CPU resources for the other tasks.

Algorithm 2. The First Optimization to AE

Input: input string, Str; left length of the input string, L;
Output: chunked position (cut-point), i;

```
1: Predefined values: window size w;
2: function AECHUNKING-OPT1(Str, L)
3:     i ← 1
4:     max.v ← Str[i].v
5:     max.pos ← i
6:     i ← i + 1
7:     while i < L do
8:         if L < max.pos + w then
9:             endPos = L
10:        else
11:            endPos = max.pos + w
12:        end if
13:        while Str[i].v < max.v and i < endPos do
14:            i ← i + 1
15:        end while
16:        if Str[i].v > max.v then
17:            max.v ← Str[i].v
18:            max.pos ← i
19:        else
20:            return max.pos + w
21:        end if
22:        i ← i + 1
23:    end while
24:    return L
25: end function
```

The Second Optimization. Low-entropy strings may exist in two patterns [8]. One pattern is of a string that contains only one unique character, such as

$$0000000000000000\ldots.$$

Another pattern is of a string that contains repetitive substrings, such as

$$abcabcabcabcabcabc\ldots.$$

Recall that AE declares the cut-point (chunk boundary) by finding the local maximum values, which provides us with another opportunity to further optimize the algorithm to detect more low-entropy strings of the first pattern. We know that the maximum value and the minimum value in a region being equal means that all values in the region are equal. As a result, we can define a low-entropy-string-threshold $LEST$ that is smaller than the expected chunk size and optimize the algorithm using the following strategy:

- *AE also finds the minimum value when finding the maximum value and the cut-point. When the algorithm is processing the $LEST$th byte of the input stream, it checks if the maximum value is equal to the minimum value. If the maximum value and the minimum value are equal, the algorithm outputs the $LEST^{th}$ byte as a cut-point. Otherwise, it continues processing the subsequent bytes of the stream.*

This optimization makes AE capable of detecting more low-entropy strings of the first pattern and outputting them with a shorter chunk size $LEST$. A smaller $LEST$ is helpful since it only reduces the average size of the low-entropy

chunks without affecting the other chunks. This optimized algorithm is described in Algorithm 3. Compared with Algorithm 1, the optimized algorithm requires more conditional branches, leading to decreased throughput performance. Further, the strategy used by the first optimization to speedup the chunking throughput is not applicable to this optimization. Therefore, AE can use either of the two optimizations described above, but can not use both of them simultaneously. We suggest the use of the second optimization if the dataset includes many low-entropy string, such as source code. Otherwise, the use of the first optimization is recommended.

Algorithm 3. The Second Optimization to AE

Input: input string, Str; left length of the input string, L;
Output: chunked position (cut-point), i;

```
1: Predefined values: window size w; threshold LEST;
2: function AECHUNKING-OPT2(Str, L)
3:     i ← 1
4:     max.v ← Str[i].v
5:     min.v ← Str[i].v
6:     max.pos ← i
7:     i ← i + 1
8:     while i < L do
9:         if Str[i].v ≥ min.v and Str[i].v ≤ max.v then
10:            if min.v = max.v and i = LEST then
11:                return i
12:            end if
13:            if i = max.pos + w then
14:                return i
15:            end if
16:        else if Str[i].v > max.v then
17:            max.v ← Str[i].v
18:            max.pos ← i
19:        else
20:            min.v ← Str[i].v
21:            if i = max.pos + w then
22:                return i
23:            end if
24:        end if
25:        i ← i + 1
26:    end while
27:    return L
28: end function
```

4 Performance Evaluation

In this section, we present the experimental evaluation of our AE and its two optimization chunking algorithms in terms of multiple performance metrics. To characterize the benefits of AE, we also compare it with two existing CDC algorithms, namely, Rabin and MAXP.

4.1 Evaluation Setup

We implement the MAXP and AE algorithms, along with AE's two optimizations, in an open-source deduplicaiton prototype system called Destor [33], [34] on two servers. One server has an 8-core Intel i7-930 2.8 GHz, running the Ubuntu 12.04 operating system. Another and more powerful one has an 8-core Intel i7-4770 3.4 GHz system, running the Ubuntu 14.04 operating system. Note that Destor uses the SHA-1 hash function to generate chunk fingerprints for the detection and elimination of duplicate chunks. To evaluate the improvement on system throughput by using AE and its optimizations, we also implemented them in the P-dedupe deduplication system [14] on the two servers mentioned above. We index all fingerprints to evaluate the exact deduplication efficiency for both the Destor and P-dedupe systems, and put all fingerprint index in RAM for P-dedupe to maximally examine the impact of different chunking algorithms on the system throughput.

Datasets. To evaluate the three CDC algorithms, we use the following three real-world datasets.

Bench: This dataset is generated from the snapshots of a cloud storage benchmark. We simulate common operations of file systems, such as file create/delete/modify operations as suggested by [35] and [36] on the snapshots, and obtain a 108 GB dataset with 20 versions.

Open-source: This dataset includes 10 versions of Emacs, 15 versions of SciLab, 20 versions of GCC, 15 versions of GDB, 40 versions of Linux kernels, 23 versions of CentOS, 18 versions of Fedora, 24 versions of FreeBSD, 17 versions of Ubuntu. The total size of this dataset is 169.5 GB.

VMDK: This dataset is from a virtual machine installed Ubuntu 12.04 LTS, which is a common use-case in real-world [37]. We run a HTTP server on this virtual machine and backup it regularly. Finally we obtain a 1.9 TB dataset with 125 backups. Since all backups are full backup, there exists a large amount of duplicate content in this dataset.

Evaluation Methodology. Every dataset was tested several times by each chunking algorithm with different average chunk sizes. For the Rabin algorithm, we still use the typical configuration Rabin_0.25 (see Section 3). We also impose a maximum threshold on chunk size whose value is eight times the expected chunk size. Because of the minimum threshold, the real average chunk size of Rabin will be greater than the expected chunk size. It is approximately equal to the expected chunk size plus the minimum threshold. For the sake of fairness, for each test, we first processed using Rabin to get the real average chunk size, and then adjusted the real average chunk size to the same value when using other algorithms. For convenience of discussion, Rabin's expected chunk sizes are used as labels to distinguish different tests on each dataset.

Since the AE algorithm is extreme-value based, the extreme value can be either maximum value or minimum value. As such, it is necessary to find out how sensitive is the AE algorithm to the choice of the form of extreme value, maximum or minimum. Therefore, in Destor [33] we implemented both versions of AE, AE_MAX and AE_MIN, to carry out a sensitivity study. In addition, in order to experimentally verify our theoretic analysis and conclusion that it is not necessary to impose a maximum threshold on AE, as summarized in Table 4, we evaluated AE's sensitivity to the maximum chunk-size threshold by implementing the same threshold on AE as that imposed on Rabin, which leads to two more AE versions of AE_MAX_T and AE_MIN_T. To evaluate the two optimizations of AE, we include them in our test. We label the original AE as AE, its optimization for higher chunking throughput as AE-Opt1 and its optimization for low-entropy strings as AE-Opt2 to distinguish them.

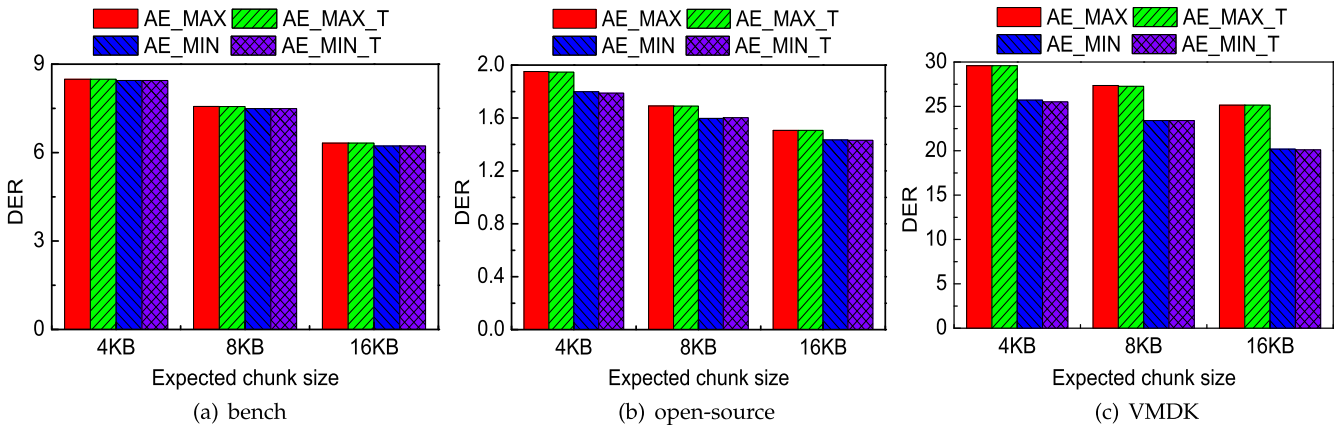


Fig. 4. Deduplication efficiency of AE. AE_MAX and AE_MIN respectively represent AE using maximum and minimum as extreme values. AE_MAX_T and AE_MIN_T denote the AE_MAX and AE_MIN with a maximum threshold.

4.2 Deduplication Efficiency

In this section, we evaluate the deduplication efficiency of our algorithm. We use deduplication elimination ratio (DER), which we define as the ratio of the size of input data to the size of data need to be actually stored, to measure the deduplication efficiency. Therefore, the greater the value of DER is, the higher the deduplication efficiency is.

Sensitivity of AE to Design Parameters. Fig. 4 shows AE's sensitivity to the key design parameters, i.e., the form of extreme value (maximum versus minimum) and the necessity of imposing a maximum chunk-size threshold, in terms of deduplication efficiency across the three datasets under AE_MAX, AE_MIN, AE_MAX_T and AE_MIN_T. Consistent with our theoretic analysis of Table 4, the experimental results show that the AE without a maximum threshold achieves nearly identical deduplication efficiency as the AE with it. Specifically, the gains in deduplication efficiency from adding the maximum threshold are negligibly small (i.e., on average 0.003 percent). But for some datasets, it actually reduces the deduplication efficiency, since the cut-points declared by AE with a maximum threshold is no longer strictly content-defined but position dependent.

Another key observation is that AE_MAX achieve higher DER than AE_MIN in three datasets. Recall that, just like the MAXP, the AE algorithm treats the bytes directly as digits. The deduplication efficiency of AE_MAX and AE_MIN depends on the frequency and the encoding of the characters in the data stream. We also tested other datasets including network traffic, video files, different types of compressed files, etc., and we found that AE_MAX can obtain the same or higher deduplication efficiency than AE_MIN on all datasets except for the network traffic dataset. This implies that we should use AE_MAX in backup storage systems and AE_MIN in Redundancy Elimination in networks. In this paper we only focus on backup storage systems. In what follows, based on the AE sensitivity study above, we used maximum value for AE algorithm. We also tested the MAXP algorithm using either the maximum or the minimum value on the three datasets, and the results are similar to and consistent with those of AE. Therefore, we used MAXP with maximum value.

Now we discuss the selection of the low-entropy-string-threshold $LEST$ (see Section 3.3) for AE-Opt2 and evaluate the benefits brought by AE and AE-Opt2 with their ability to detect the low-entropy strings. As mentioned in Section 3.3, low-entropy strings may exist in two patterns [8], here we only focus on the first pattern, namely, a string that contains only one unique character, since the second pattern of such strings is difficult to detect. The Rabin algorithm can also detect some low-entropy strings with the help of the maximum threshold when the low-entropy strings are long enough. However, MAXP does not have this capability. In this evaluation, we set the value of $LEST$ to $\frac{1}{2}$, $\frac{1}{4}$ and $\frac{1}{8}$ of the expected chunk size (ECS) respectively for AE-Opt2. Table 5 shows the ratio of the size of the low-entropy strings detected to the size of the input data for the open-source and VMDK datasets (bench dataset is not included in the table since it contains very few low-entropy strings). As shown in the table, AE and AE-Opt2 detect more low-entropy strings than Rabin and MAXP in both datasets. Specifically, AE detects up to $7.9\times$, with an average of $3.9\times$,

TABLE 5
Ratio of the Size of the Low-Entropy Strings Detected to the Size of Data Stream, Where ECS in the Table Means Expected Chunk Size

algorithm	open-source				vmdk			
	2 KB	4 KB	8 KB	16 KB	2 KB	4 KB	8 KB	16 KB
Rabin	0	0	0	0	0	0	0	0
Rabin_0.25	1.04%	0.59%	0.23%	0.14%	2.34%	1.78%	1.49%	1.38%
MAXP	0	0	0	0	0	0	0	0
AE	5.47%	1.98%	1.49%	1.11%	6.07%	3.27%	2.84%	2.43%
AE-opt2 ($LEST = \frac{1}{2} \times ECS$)	8.62%	5.04%	3.26%	1.49%	7.95%	5.41%	3.02%	2.67%
AE-opt2 ($LEST = \frac{1}{4} \times ECS$)	9.54%	5.58%	3.69%	2.32%	8.48%	5.78%	4.14%	2.84%
AE-opt2 ($LEST = \frac{1}{8} \times ECS$)	10.04%	6.27%	4.2%	2.61%	8.73%	6.24%	4.81%	3.68%

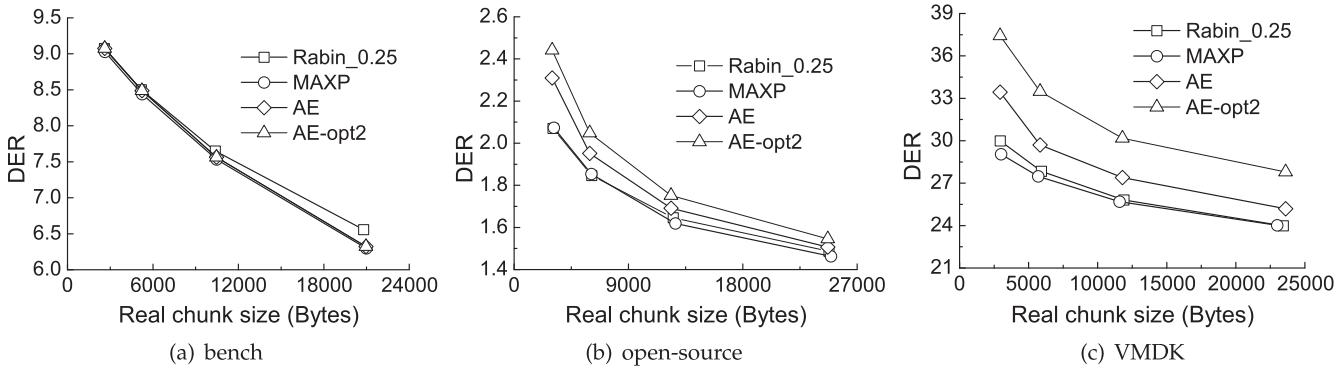


Fig. 5. Deduplication efficiency of the three chunking algorithms on the three real-world datasets.

more low-entropy strings than Rabin_0.25. AE-opt2 with an $LEST$ of $\frac{1}{8}$ of ECS detects up to $18.6\times$ ($8.8\times$ on average) more such strings than Rabin_0.25 and up to $3.16\times$ ($2.09\times$ on average) more than AE. Furthermore, for AE-Opt2, a smaller $LEST$ contributes to detecting more low-entropy string, but too small chunks cause higher metadata overhead. In the following evaluations, we set the value of $LEST$ to $\frac{1}{8}$ of ECS. Note that all detected low-entropy strings have the same length. For Rabin, the length is equal to Rabin's maximum chunk-size threshold. For AE and AE-Opt2, it is equal to $w+1$ and $LEST$ respectively. Therefore, all of these low-entropy strings can be eliminated directly by chunk-level deduplication.

Next we compare the deduplication efficiency of the four chunking algorithms, namely Rabin_0.25, MAXP, AE, AE-Opt2. AE-Opt1 is not included in this comparison since it is an optimization of AE focusing on improving the chunking throughput performance and thus has the same deduplication efficiency as AE. Fig. 5 shows the results of this comparison. As shown in the figure, on the bench dataset, four algorithms achieve almost the same DER, and Rabin_0.25 achieves slightly higher DER than the other three algorithms when real chunk size is greater than 10 KB. On the open-source and VMDK datasets, Rabin_0.25 achieves comparable or higher DER than MAXP, AE attains higher DER than Rabin_0.25 and MAXP, while AE-Opt2 achieves the highest DER among the four algorithms. Obviously the reason for AE-Opt2's superior DER performance to AE is its ability to detect and eliminate more low-entropy strings.

Chunk Size Variance. Chunk size variance also has a significant impact on deduplication efficiency [7]. We have proved that AE has smaller chunk size variance than the other algorithms in theory in Table 4, here we test it using the real-world datasets. We selected a gcc file in the open-source dataset, and processed it using the three algorithms with the expected chunk size of 4, 8, and 16 KB respectively. Fig. 6 depicts the distribution. In this figure we can see that AE achieve more uniform chunk-size distribution, which also means that AE has smaller chunk-size variance than the other two algorithms.

4.3 Chunking Throughput

Next we evaluate the chunking throughput. In this evaluation we use a Ramdisk-driven emulation that uses RAM exclusively, which is the same as that used in Shredder [13], to avoid the performance bottleneck caused by read and write operations on the disks. Fig. 7 compare the chunking throughput measure among the Rabin_0.25, MAXP, AE, AE-Opt1, and AE-Opt2 algorithms on the i7-4770 CPU across the three datasets. On each dataset, the expected chunk size is set at 2, 4, 8, and 16 KB respectively. To better present the throughput bottleneck of the deduplication workflow, we also present the throughput of SHA-1 in the figure.

From the figure, we can draw the following observations. First, AE outperforms the Rabin_0.25 and MAXP algorithms in terms of the chunking throughput. Specifically, on average, AE improves the throughput performance of Rabin_0.25 and MAXP by $3.21\times$ and $2.55\times$ respectively. Second, AE-Opt1 significantly improves the chunking throughput. On average, it accelerates the chunking speed of AE by 73.1 percent. Third, AE-Opt2 decreases the chunking throughput. However, its throughput is still significantly greater than that of SHA-1.

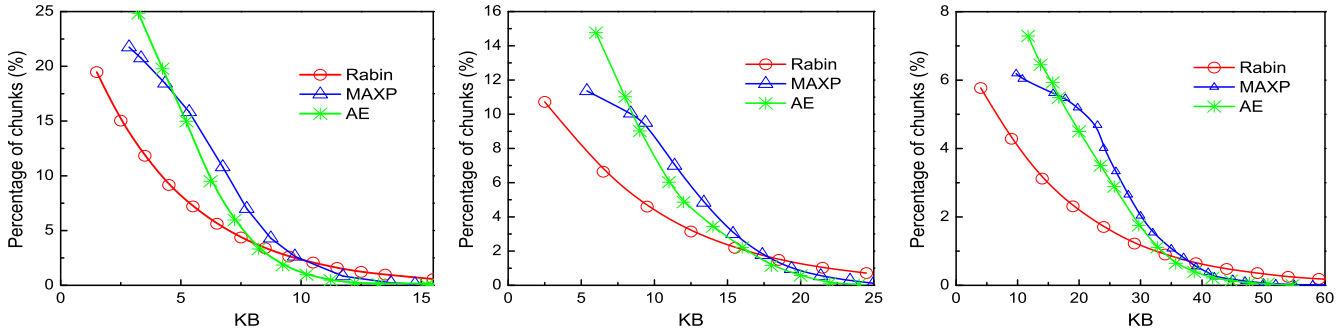


Fig. 6. Distribution of the chunk size for the three algorithms. Expected chunk sizes are 4, 8, and 16 KB respectively.

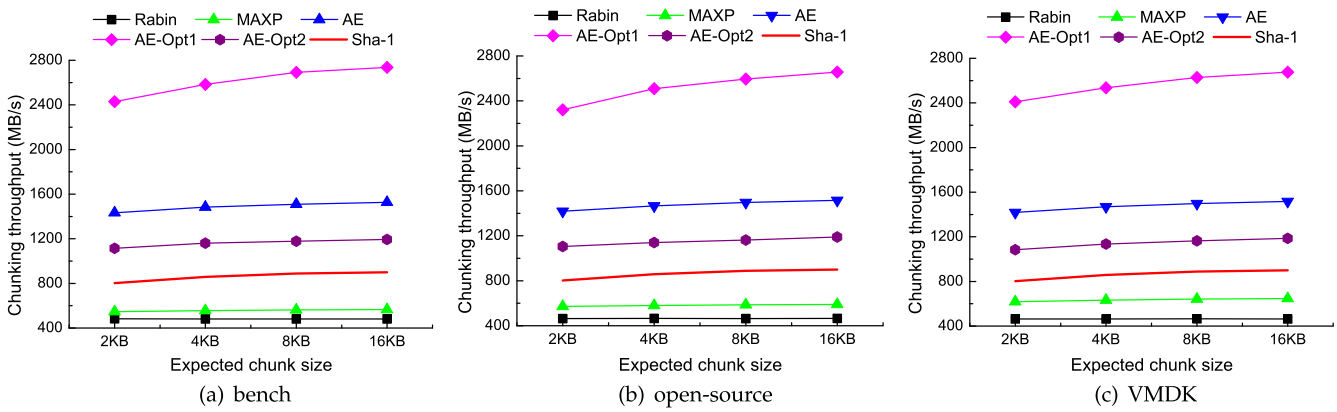


Fig. 7. Throughput of SHA-1 and the five algorithms on the i7-4770 CPU.

4.4 Impact of the Chunking Algorithms on the System Throughput

Now we discuss the impact of the chunking algorithms on the overall throughput of the deduplication system encompassing the entire deduplication workflow, or system throughput for brevity. As mentioned before, the chunking stage is the performance bottleneck of the deduplication workflow, which means that improvement on the chunking throughput can directly benefit the system throughput before the system meets another performance bottleneck. For the convenience of description, we refer to the P-dedupe systems with the Rabin_0.25, MAXP, AE, AE-Opt1 and AE-Opt2 algorithms as P-dedup-Rabin, P-dedup-MAXP, P-dedup-Rabin-AE, P-dedup-AE-Opt1 and P-dedup-AE-Opt2 respectively. In this evaluation we still use a Ramdisk-driven emulation to avoid the performance bottleneck caused by disk I/O. For each dataset, we only use 3 GB, a small part of its total size in the evaluation, while the VMDK dataset is not included since even a single file in this dataset is too large to fit in the RAM.

Fig. 8 shows the system throughput of the two systems with an expected chunk size of 8 KB. As shown in the figure, P-dedup-MAXP attains almost the same system throughput as P-dedup-Rabin on the i7-930 CPU and slightly higher system throughput than P-dedup-Rabin on the i7-4770 CPU. P-dedup-AE-Opt1 obtains almost the same system throughput as P-dedup-AE. In addition, P-dedup-AE outperforms P-dedup-Rabin significantly in terms of system throughput, by a speedup factor of up to $1.63\times$, with an average factor of $1.54\times$ on the i7-930 CPU. On the i7-4770 CPU, the improvement is even greater, with the system throughput achieved by P-dedup-AE reaching up to $2\times$ (on average $1.84\times$) higher than that of P-dedup-Rabin. As for P-dedup-AE-Opt2, its system throughput is slightly lower than P-dedup-AE on the i7-930 CPU. This is because the chunking stage is not fast enough to totally remove the chunking stage performance bottleneck. Despite of the lower system throughput compared with P-dedup-AE (with a decrease of 8.3 percent on average), it still accelerates the system throughput of P-dedup-Rabin by more than $1.4\times$. On the i7-4770 CPU, the system throughput of P-dedup-AE-Opt2 and P-dedup-AE are almost the same, which also means that the performance bottleneck of the deduplication workflow has been shifted from the chunking stage to the fingerprinting stage since the chunking throughput has no effect on the system throughput. This, combined with the results of Table 5, shows that AE-Opt2 is useful for the datasets that contain large amounts of

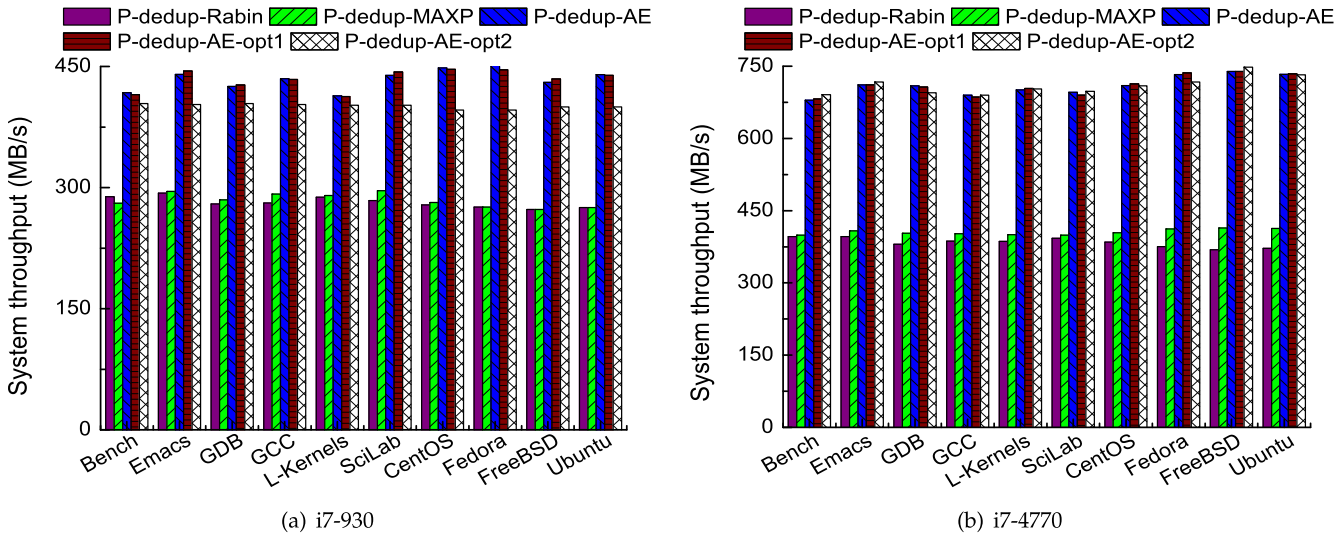


Fig. 8. System throughput of P-dedup-Rabin, P-dedup-MAXP, P-dedup-AE, P-dedup-AE-opt1 and P-dedup-AE-opt2 on the i7-930 and i7-4770 CPUs with Ramdisk.

low-entropy strings since it can detect and eliminate more low-entropy strings (by up to $3.16\times$), while has marginal or no affect on the system throughput performance.

5 Conclusion

High computational overheads of the existing CDC algorithms render the chunking stage the performance bottleneck of the deduplication workflow. We presented AE, a new CDC algorithm that effectively employs an asymmetric sliding window to find the local extreme value for fast content-defined chunking. As a result, AE is shown to have lower computational overheads and thus higher chunking throughput, smaller chunk size variance, and the ability to eliminate more low-entropy strings than the state-of-the-art algorithms. To strengthen the superiority of AE, we present two optimizations to AE, with one optimization providing higher chunking throughput while the other offering higher deduplication efficiency by eliminating more low-entropy strings. Our experimental evaluation based on three real-world datasets demonstrates the robustness of AE in terms of deduplication efficiency and superior chunking throughput performance that is able to shift the performance bottleneck of the deduplication workflow from the chunking stage to the fingerprinting stage and significantly improve the system throughput performance.

Acknowledgments

The work was partly supported by NSFC Nos. 61502190, 61502191, 6140050892, 61232004, and 61402061; 863 Project 2013AA013203, 2015AA016701, and 2015AA015301; Fundamental Research Funds for the Central Universities, HUST, under Grant No. 2015MS073; US National Science Foundation under Grants CNS-1116606 and CNS-1016609; the work was also supported by NSFC under Grant No. 61502191, CCF-Tencent Open Fund 2015, Key Laboratory of Information Storage System, Ministry of Education, China. An early version of this paper appeared in the proceedings of IEEE INFOCOM, 2015. Dan Feng is the corresponding author.

References

[1] The Digital Universe of Opportunities: Rich Data and the Increasing Value of the Internet of Things, Apr. 2014. [Online]. Available: http://www.emc.com/leadership/digital-universe/2014iview/executive-summary.htm, *EMC Digital Universe with Research & Analysis by IDC*.

[2] S. Quinlan and S. Dorward, "Venti: A new approach to archival storage," in *Proc. 1st USENIX Conf. File Storage*, 2002, Art. no. 7.

[3] S. Sanadhya, R. Sivakumar, K.-H. Kim, P. Congdon, S. Lakshmanan, and J. P. Singh, "Asymmetric caching: Improved network deduplication for mobile devices," in *Proc. 18th Annu. Int. Conf. Mobile Comput. Netw.*, 2012, pp. 161–172.

[4] A. Tridgell, *Efficient Algorithms for Sorting and Synchronization*. Canberra, Australia: Australian Nat. Univ., 1999.

[5] A. Muthitacharoen, B. Chen, and D. Mazieres, "A low-bandwidth network file system," in *Proc. 18th Symp. Operating Syst. Principles*, 2001, pp. 174–187.

[6] W. Xia, et al., "A comprehensive study of the past, present, and future of data deduplication," *Proc. IEEE*, 2016, Doi: 10.1109/JPROC.2016.2571298.

[7] K. Eshghi and H. K. Tang, "A framework for analyzing and improving content-based chunking algorithms," Hewlett-Packard Labs Palo Alto, CA, USA, vol. 30, Tech. Rep. HPL-2005–30R1, 2005.

[8] S. Schleimer, D. S. Wilkerson, and A. Aiken, "Winnowing: Local algorithms for document fingerprinting," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2003, pp. 76–85.

[9] A. El-Shimi, R. Kalach, A. Kumar, A. Ottean, J. Li, and S. Sengupta, "Primary data deduplication—large scale study and system design," in *Proc. USENIX Conf. Annu. Tech. Conf.*, 2012, pp. 26–26.

[10] M. O. Rabin, *Fingerprinting by Random Polynomials*. Cambridge, MA, USA: Center for Research in Computing Techn., Aiken Computation Laboratory, Harvard Univ., 1981.

[11] E. Kruus, C. Ungureanu, and C. Dubnicki, "Bimodal content defined chunking for backup streams," in *Proc. 8th USENIX Conf. File Storage Technologies*, 2010, pp. 18–18.

[12] W. Xia, H. Jiang, D. Feng, and L. Tian, "Combining deduplication and delta compression to achieve low-overhead data reduction on backup datasets," in *Proc. IEEE Data Compression Conf.*, 2014, pp. 203–212.

[13] P. Bhatotia, R. Rodrigues, and A. Verma, "Shredder: GPU-accelerated incremental storage and computation," in *Proc. 10th USENIX Conf. File Storage Technologies*, 2012, pp. 14–14.

[14] W. Xia, H. Jiang, D. Feng, L. Tian, M. Fu, and Z. Wang, "P-Dedupe: Exploiting parallelism in data deduplication system," in *Proc. IEEE 7th Int. Conf. Netw. Architecture Storage*, 2012, pp. 338–347.

[15] N. Bjørner, A. Blass, and Y. Gurevich, "Content-dependent chunking for differential compression, the local maximum approach," *J. Comput. Syst. Sci.*, vol. 76, no. 3, pp. 154–203, 2010.

[16] A. Anand, C. Muthukrishnan, A. Akella, and R. Ramjee, "Redundancy in network traffic: Findings and implications," in *Proc. ACM 11th Int. Joint Conf. Meas. Modeling Comput. Syst.*, 2009, pp. 37–48.

[17] B. Agarwal, et al., "EndRE: An end-system redundancy elimination service for enterprises," in *Proc. 7th USENIX Conf. Networked Syst. Des. Implementation*, 2010, pp. 28–28.

[18] N. T. Spring and D. Wetherall, "A protocol-independent technique for eliminating redundant network traffic," in *Proc. ACM Conf. Appl. Technologies Architectures Protocols Comput. Commun.*, 2000, pp. 87–95.

[19] G. Lu, Y. Jin, and D. H. Du, "Frequency based chunking for data de-duplication," in *Proc. IEEE Int. Symp. Modeling Anal. Simulation Comput. Telecommun. Syst.*, 2010, pp. 287–296.

[20] B. Romański, Ł. Heldt, W. Kilian, K. Lichota, and C. Dubnicki, "Anchor-driven subchunk deduplication," in *Proc. ACM 4th Annu. Int. Conf. Syst. Storage*, 2011, pp. 16:1–16:13.

[21] B. Zhou and J. Wen, "Hysteresis re-chunking based metadata harnessing deduplication of disk images," in *Proc. IEEE 42nd Int. Conf. Parallel Process.*, 2013, pp. 389–398.

[22] D. R. Bobbarjung, S. Jagannathan, and C. Dubnicki, "Improving duplicate elimination in storage systems," *ACM Trans. Storage*, vol. 2, no. 4, pp. 424–448, 2006.

[23] J.-l. Gailly, *Gzip: The Data Compression Program*. Bloomington, IN, USA: iUniverse, 2000.

[24] G. Wallace, et al., "Characteristics of backup workloads in production systems," in *Proc. 10th USENIX Conf. File Storage Technologies*, 2012, pp. 4–4.

[25] T. Summers, "Hardware compression in storage and network attached storage," *SNIA tutorial, Spring* 2007. [Online]. Available: http://www.snia.org/education/tutorials/2007/spring

[26] D. T. Meyer and W. J. Bolosky, "A study of practical deduplication," *Proc. 9th USENIX Conf. File Stroage Technologies*, 2011, pp. 1–1.

[27] S. Al-Kiswany, A. Gharaibeh, E. Santos-Neto, G. Yuan, and M. Ripeanu, "StoreGPU: Exploiting graphics processing units to accelerate distributed storage systems," in *Proc. 17th Int. Symp. High Performance Distrib. Comput.*, 2008, pp. 165–174.

[28] J. Bonwick, ZFS deduplication, Nov. 2009. [Online]. Available: https://blogs.oracle.com/bonwick/entry/zfs_dedup

[29] C. Liu, Y. Xue, D. Ju, and D. Wang, "A novel optimization method to improve de-duplication storage system performance," in *Proc. IEEE 15th Int. Conf. Parallel Distrib. Syst.*, 2009, pp. 228–235.

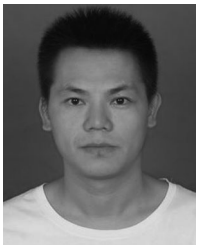
[30] B. Zhu, K. Li, and R. H. Patterson, "Avoiding the disk bottleneck in the data domain deduplication file system," in *Proc. 6th USENIX Conf. File Storage Technologies*, 2008, pp. 18:1–18:14.

[31] W. Xia, H. Jiang, D. Feng, and Y. Hua, "SiLo: A similarity-locality based near-exact deduplication scheme with low RAM overhead and high throughput," in *Proc. USENIX Conf. USENIX Annu. Tech. Conf.*, 2011, pp. 26–28.

[32] M. Lillibridge, K. Eshghi, D. Bhagwat, V. Deolalikar, G. Trezis, and P. Camble, "Sparse indexing: Large scale, inline deduplication using sampling and locality," in *Proc. 7th Conf. File Storage Technologies*, 2009, pp. 111–123.

[33] M. Fu, "Destor: An experimental platform for chunk-level data deduplication," 2014. [Online]. Available: https://github.com/fomy/destor

[34] M. Fu, D. Feng, Y. Hua, X. He, Z. Chen, and W. Xia, "Design tradeoffs for data deduplication performance in backup workloads," in *Proc. 13th USENIX Conf. File Storage Technologies*, 2015, pp. 331–344.
[35] V. Tarasov, A. Mudrankit, W. Buik, P. Shilane, G. Kuenning, and E. Zadok, "Generating realistic datasets for deduplication analysis," in *Proc. USENIX Conf. Annu. Tech. Conf.*, 2012, pp. 24–24.
[36] M. Lillibridge, K. Eshghi, and D. Bhagwat, "Improving restore speed for backup systems that use inline chunk-based deduplication," in *Proc. 11th USENIX Conf. File Storage Technologies*, 2013, pp. 183–198.
[37] F. Guo and P. Efstathopoulos, "Building a high-performance deduplication system," in *Proc. Conf. USENIX Annu. Tech. Conf.*, 2011, pp. 25–25.

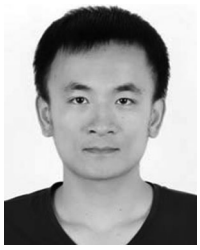


Yucheng Zhang is currently working toward the PhD degree majoring in computer architecture at Huazhong University of Science and Technology, China. His research interests include data deduplication, storage systems, etc.

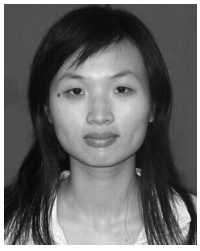
Dan Feng received the BE, ME, and PhD degrees in computer science and technology from Huazhong University of Science and Technology (HUST), China, in 1991, 1994, and 1997, respectively. She is a professor and the dean of the School of Computer Science and Technology, HUST. Her research interests include computer architecture, massive storage systems, and parallel file systems. She has more than 100 publications in major journals and international conferences, including the *IEEE Transactions on Computers*, the *IEEE Transactions on Parallel and Distributed Systems*, the *ACM Transactions on Storage*, the *Journal of Computer Science and Technology*, FAST, USENIX ATC, ICDCS, HPDC, SC, ICS, IPDPS, and ICPP. She has served as the program committees of multiple international conferences, including SC 2011, 2013 and MSST 2012, 2015. She is a member of the IEEE and the ACM.

Hong Jiang received the BSc degree in computer engineering from Huazhong University of Science and Technology, Wuhan, China, in 1982, the MASc degree in computer engineering from the University of Toronto, Toronto, Canada, in 1987, and the PhD degree in computer science from Texas A&M University, College Station, TX, USA, in 1991. He is currently a chair and Wendell H. Nedderman Endowed professor in Computer Science and Engineering Department, University of Texas at Arlington. Prior to joining UTA, he served as a program director at the National Science Foundation (Jan. 2013-Aug. 2015) and he was at the University of Nebraska-Lincoln since 1991, where he was Willa Cather professor of computer science and engineering. He has graduated 13 PhD students who upon their graduations either landed academic tenure-track positions in PhD-granting US institutions or were employed by major US IT corporations. His current research interests include computer architecture, computer storage systems and parallel I/O, high-performance computing, big data computing, cloud computing, and performance evaluation. He recently served as an associate editor of the *IEEE Transactions on Parallel and Distributed Systems*. He has more than 200 publications in major journals and international conferences in these areas, including the *IEEE Transactions on Parallel and Distributed Systems*, the *IEEE Transactions on Computers*, *Proceedings of the IEEE*, the *ACM Transactions on Architecture and Code Optimization*, the *Journal of Parallel and Distributed Computing*, the *International Science Congress Association*, MICRO, USENIX ATC, FAST, EUROSYS, LISA, SIGMETRICS, ICDCS, IPDPS, MIDDLEWARE, OOPLAS, ECOOP, SC, ICS, HPDC, INFOCOM, ICPP, etc., and his research has been supported by US National Science Foundation, DOD, the State of Texas and the State of Nebraska. He is a fellow of the IEEE, and a member of the ACM.

Wen Xia received the PhD degree in computer science from Huazhong University of Science and Technology (HUST), Wuhan, China, in 2014. He is currently an assistant professor in the School of Computer Science and Technology, HUST. His research interests include deduplication, data compression, storage systems, cloud storage, etc. He has published more than 20 papers in major journals and international conferences including the *IEEE Transactions on Computers*, the *IEEE Transactions on Parallel and Distributed Systems*, USENIX ATC, USENIX FAST, INFOCOM, IFIP Performance, IEEE DCC, MSST, IPDPS, HotStorage, etc. He is a member of the ACM, CCF, and the IEEE.



Min Fu is currently working toward the PhD degree majoring in computer architecture at Huazhong University of Science and Technology, Wuhan, China. His current research interests include data deduplication, storage systems, and reliability. He has several papers in major journals and conferences including the *IEEE Transactions on Parallel and Distributed Systems*, USENIX ATC, FAST, etc.



Fangting Huang received the BE degree in software engineering from Sun Yet-sen University, China, in 2010. She is currently working toward the PhD degree in computer architecture at Huazhong University of Science and Technology. Her research interest includes computer architecture and storage systems.

Yukun Zhou is currently working toward the PhD degree majoring in computer architecture at Huazhong University of Science and Technology, China. His research interests include data deduplication, storage security, etc. He has several papers in refereed journals and conferences including *Performance Evaluation*, MSST, etc.