Future Generation Computer Systems 71 (2017) 145–156

Contents lists available at ScienceDirect

Future Generation Computer Systems

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

A new content-defined chunking algorithm for data deduplication in cloud storage

Ryan N.S. Widodo[a], Hyotaek Lim[b,*], Mohammed Atiquzzaman[c]

[a] Department of Ubiquitous IT, Dongseo University, 617-716 Busan, South Korea
[b] Division of Computer Engineering, Dongseo University, 617-716 Busan, South Korea
[c] School of Computer Science, University of Oklahoma, Norman, OK 73019, United States

HIGHLIGHTS

- Studies on multiple chunking algorithms.
- Proposes RAM, a fast and hash-less CDC algorithm for low performance devices.
- Analyzes the properties of RAM and other CDC algorithms.
- Carried out performance evaluation and compares it with other CDC algorithms.

ARTICLE INFO

Article history:
Received 14 November 2016
Received in revised form
9 January 2017
Accepted 8 February 2017
Available online 10 February 2017

Keywords:
Data deduplication
Cloud storage
Content-defined chunking
Hash-less chunking
Asymmetric window

ABSTRACT

Chunking is a process to split a file into smaller files called chunks. In some applications, such as remote data compression, data synchronization, and data deduplication, chunking is important because it determines the duplicate detection performance of the system. Content-defined chunking (CDC) is a method to split files into variable length chunks, where the cut points are defined by some internal features of the files. Unlike fixed-length chunks, variable-length chunks are more resistant to byte shifting. Thus, it increases the probability of finding duplicate chunks within a file and between files. However, CDC algorithms require additional computation to find the cut points which might be computationally expensive for some applications. In our previous work (Widodo et al., 2016), the hash-based CDC algorithm used in the system took more process time than other processes in the deduplication system. This paper proposes a high throughput hash-less chunking method called Rapid Asymmetric Maximum (RAM). Instead of using hashes, RAM uses bytes value to declare the cut points. The algorithm utilizes a fix-sized window and a variable-sized window to find a maximum-valued byte which is the cut point. The maximum-valued byte is included in the chunk and located at the boundary of the chunk. This configuration allows RAM to do fewer comparisons while retaining the CDC property. We compared RAM with existing hash-based and hash-less deduplication systems. The experimental results show that our proposed algorithm has higher throughput and bytes saved per second compared to other chunking algorithms.

1. Introduction

The Internet is one of the important technology advancements of recent times. Knowledge and media sharing are easier than before because the Internet provides data access everywhere and anywhere, and interconnects devices and machines around the world. However, the bandwidth requirement and the amount of data to be stored increase as the number of internet users increases. This has led to challenges researchers to develop high bandwidth networks and techniques to reduce the amount of data to be transferred.

A technique to reduce the amount of data transfer is to use compression. Data compression algorithms work by eliminating redundant data within a file or multiple files. In particular, a data compression technique called differential compression eliminates the amount of data to be transferred by sending only the differences between files. Differential compression works by

* Corresponding author.
E-mail addresses: ryannswidodo@gmail.com (R.N.S. Widodo), htlim@dongseo.ac.kr (H. Lim), atiq@ou.edu (M. Atiquzzaman).

http://dx.doi.org/10.1016/j.future.2017.02.013
0167-739X/© 2017 Elsevier B.V. All rights reserved.

comparing two files and only sends the differences. For example, when a user wants to update a file, the server compares and finds the difference between the old version and the new version. The server sends the difference, and the client is required to process the differences and the old file to get the new file. In this scenario, the differential compression technique is called local differential compression.

Remote differential compression (RDC) refers to another situation where the client has a file $F1$ and the server has a similar file $F2$. In this situation, the server and the client work together to discover the differences between the two files, and only the differences are sent. RSYNC by Tridgell et al. [1], which is an RDC protocol, does remote differential compression by splitting a file into multiple fixed length chunks. Checksums or hash values are calculated from the chunks and sent to the server. The server does the same thing to the file that it believes to be similar and compares the hashes. Only chunks with different hashes are sent to the client, thus reducing the amount of data to be transferred.

The fix-sized chunk used in RSYNC has a problem. Bjørner et al. [2] stated it cannot identify two identical files, F1 and F2, where F2 is F1 with a byte inserted at the beginning of the file. When F1 and F2 are processed using the RSYNC protocol described in the previous paragraph, no checksums of F1 matches the checksums of F2. This is called the byte shifting problem. Content defined chunking (CDC) algorithms fixed this problem by chunking the file based on the internal features of a file. Similar to RDC, CDC algorithms are also used in data deduplication to split a file into chunks for detecting duplicate data.

In the cloud storage ecosystem, differential compression is known as data deduplication. Data deduplication is important because it saves storage spaces by eliminating redundant data within files and between files. Data deduplication involves three main components: chunking, hashing, and comparing hashes to detect redundancy and it works by chunking the files into chunks and comparing the chunks by using a mathematical hash function. In a deduplication system, chunking defines the characteristic of the system because chunking algorithms used in the system can make the system more effective and faster.

One of the most known chunking algorithms is Rabin based chunking algorithm. Rabin based chunking algorithm uses Rabin rolling hash by Michael O. Rabin [3] to find the cut-point. The rolling hash works by using a sliding window, and a hash value is calculated when the sliding window is moving. Rabin rolling hash consumes a considerable amount of time for calculating the hashes. Unlike Rabin based chunking, Local Maximum Chunking (LMC), which is proposed by Bjørner et al. [2], uses byte values of a file and sliding window to determine the cut point. In this approach, no hash calculation is required but it uses comparisons for each byte processed. Another variant of the local maximum chunking is Asymmetric Extremum (AE) [4], which uses a fixed length window and a variable-sized window to determine a cut point. The extreme value is located between the two windows. Unlike local maximum, AE does not use sliding window to find the cut points. Thus, it requires fewer comparisons because the local maximum needs to find the extreme value in the sliding window each time the window slides. Although the number of comparisons is significantly reduced for AE compared to LMC, AE still does a lot of comparisons which might be expensive for performance limited devices. The *aim* of this work is to reduce the computational overhead of the existing CDC algorithm. We achieved the lower computational overhead by harnessing the statistical properties of the algorithm to minimize the number of comparisons.

This paper proposes a new Rapid Asymmetric Maximum (RAM) chunking algorithm which is based on the AE algorithm. Similar to AE, the algorithm uses two windows: a fixed-length window and a variable-sized window. Unlike AE, RAM uses different positions for the windows. The windows position is started with the fixed-length window and followed by the variable-sized window and the maximum sized byte. The cut point is located at the end of the chunk, which puts the maximum-valued byte at the end of the chunk. This configuration has different statistical properties than AE's configurations, which reduces the number of comparisons. Thanks to the reduced number of comparisons, RAM performs better than AE in terms of deduplication throughput. In addition to the reduced number of comparisons, the algorithm produces chunks with sizes distribution similar to AE's. The *contributions* of this paper are as follows:

(i) Studies on multiple chunking algorithms;
(ii) Proposes RAM, a fast and hash-less CDC algorithm;
(iii) Analyzes the properties of RAM and compares it with other CDC algorithms;
(iv) Carried out performance evaluation of RAM and compares it with other CDC algorithms to learn the performance of RAM compared to other CDC algorithms when used in a deduplication system.

The rest of this paper is organized as follows. Section 2 explains CDC algorithms, its problem, and our motivation. Section 3 discusses the design and analysis of our proposed RAM algorithm. In Section 4, we compare RAM with other CDC algorithms, followed by discussion of the results in Section 5. Lastly, we conclude our paper in Section 6.

2. Background and motivation

This section discusses the background and motivation of CDC algorithms, related work, their limitations, and motivation of our work.

2.1. Background

Chunking is used in many data compression applications. For example, it is used in data deduplication and remote differential compression. Data deduplication works by eliminating duplicate data within the files and between files. In data deduplication, a chunking algorithm is one of the vital parts to achieve high duplicate elimination. By choosing the correct chunking method, we can save time and space [5]. Data deduplication can be applied on cloud storage [6], virtual disk images [7], memory [8,9], and network traffic [10].

One of the applications of data deduplication is remote differential compression. Remote differential compression does not save space but it saves network bandwidth and time by sending only the parts that are not available to the receiver as stated by Teodosio et al. [11]. Additionally, Ruppin et al. [12] proposed a data synchronization system that uses chunking for data synchronization across multiple devices.

Chunking algorithms can be categorized into two categories: (i) whole file chunking and (ii) block chunking. Whole file chunking means the whole file is treated as one chunk, while block chunking means the file is split into multiple chunks. When chunking a file into blocks or chunks, the chunk size can be fixed-sized or variable-sized. Fixed-sized chunking is fast and not resistant to byte insertion or shifting. When the file is shifted by a byte insertion or deletion, the chunks will become completely different chunks and undetectable by the chunk duplicate search. Content Defined Chunking (CDC) solves this problem by chunking the file into variable-sized chunks. CDC algorithms find the cut point by using internal features of the file. Therefore, when the file is shifted, only several chunks are affected. CDC has a higher probability of eliminating duplicates within the files and between files compared to fixed-sized chunking.

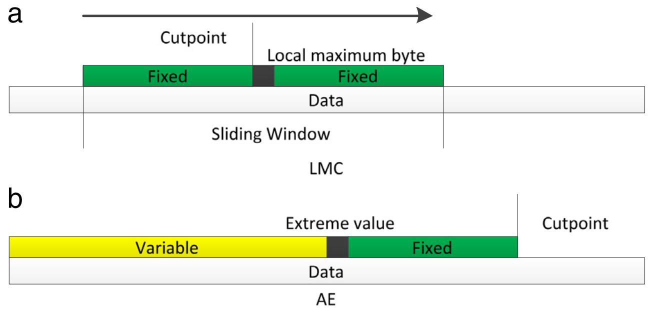


Fig. 1. Illustrations on LMC and AE.

Content defined chunking's definition allows the use of any method to find the cut-point under the condition that the cut-point is found by using the internal feature of the file. In case of hash-based chunking algorithm, a sliding window and a hash function are used to find a cut-point. There are also CDC algorithms that uses byte value to find the cut-point such as CDC algorithms proposed by Bjørner et al. [2] and Zhang et al. [4]. Although the method to find a cut point is not limited to hashes and bytes, hashes and bytes are commonly used to find the cut-point because of the computational overhead. Neural networks or other machine learning algorithms can also be used to find the cut-point. However, the computational overhead can be higher than using hashes and bytes to find a cut-point.

One of the oldest CDC algorithms is Rabin [3] based CDC algorithm. It finds the cut-point by using Rabin rolling hash. Rabin rolling hash uses sliding window and every time the window is moving, a hash value is calculated. When the hash value matches a predefined value, it uses the window position for the hash value as a cut-point. Since the checksum is calculated based on polynomials over a finite field, the old checksum can be used to calculate the new checksum when the window slides.

Another CDC algorithm proposed by Bjørner et al. [2] is Local Maximum Chunking method (LMC). LMC finds the maximum valued byte or local maximum byte by using a sliding window. The sliding window has three components: (i) the local maximum byte, (ii) left window, and (iii) right window. The local maximum byte is located in between the two fixed-sized windows as illustrated in Fig. 1(a). LMC defines a byte as the cut-point when the local maximum byte is larger than all of the bytes in the fixed windows. When this condition is fulfilled, a cut-point is found. The left fixed window will be included in the chunk which makes the minimum chunk size w, where w is the size of the window.

To address the slow chunking performance of LMC, Zhang et al. [4] proposed Asymmetric Extremum (AE) algorithm. The algorithm is similar to LMC in terms treating a byte as digits. As illustrated in Fig. 1(b), AE uses two windows, variable-sized window on the left and fix-sized window on the right. The extreme-valued byte is located in the middle of the two windows. A cut-point is found when the extreme-valued byte is bigger or smaller compared to all the values in the two windows. The minimum chunk size is $w + 1$ where w is the size of the window. Zhang et al. in [4] explains the performance difference between maximum and minimum for the extreme-valued byte is negligible. Therefore, we used only maximum for the extreme-valued byte in our analysis and performance evaluation.

Chunks management is also important because when the amount chunk is big, finding chunks can be a time-consuming task. There are multiple active researches on chunk management especially on techniques to reduce the time to search duplicate chunks. Some of the challenges in this area are reducing chunk search time by utilizing cache or bloom filter [13] and scalable chunk management system [14]. Even though chunk management is not our main focus, we implemented a chunk management system based on a bloom filter and a hash table for chunk duplicate search.

2.2. Challenges and motivation

CDC offers more benefits than fix-sized chunking. However, CDC process is slightly more time-consuming which limits the use of CDC algorithms on latency-critical applications and on devices with limited processing capability such as mobile devices and Internet of Things (IoT) devices. In our previous work [5], we used Rabin based chunking algorithm for the deduplication system to eliminate duplicate data. We found out that the main drawback of using CDC algorithms in the mobile application is its large processing time.

Before we discuss and compare various CDC algorithms, we would like to state the following criteria that can be used to compare CDC algorithms, which is used by Zhang et al. in [4]:

1. *Content dependent*. The algorithm should define the cut point based on the internal features of the file, which makes it resistant against byte shifting and allows the algorithm to find duplicate chunks between two or more files.
2. *Low chunk sizes variance*. The chunks produced by the algorithm should have low chunk variance because it might affect the deduplication efficiency [15]. To limit the chunk variance, we can add a limit on the maximum or minimum size of the chunks. However, this will affect the content dependent properties of the algorithm and make the algorithm vulnerable against byte shifting.
3. *Ability to eliminate low entropy strings*. Low entropy strings are strings which consist of repetitive bytes or patterns. When it encounters strings with low entropy or low variance, it is preferable for the algorithm to be able to eliminate the redundancy within the string.
4. *High throughput and duplicate detection*. The algorithm should have a good balance between deduplication performance and computational overhead.

Rabin based CDC algorithm uses polynomial over a finite field and a sliding window to calculate the hash [3]. Calculating the hash is fast because it only needs to consider the new byte and the most left byte in the window. Rabin-based CDC algorithms have a few disadvantages due to the use of the hash. It is time-consuming because of the hash calculation, and changing a byte in the chunk has a high probability of changing the cut-point as it might create a different hash value. It also has a large chunk variance because of the higher probability of having a long chunk [2,4]. In order to limit the chunk variances, we can use a limit on the chunk size. However, this will reduce the resistance of the algorithm against byte shifting.

Local Maximum Chunking (LMC) [2] is a CDC algorithm that compares bytes with bytes as a number to find the cut point. LMC has a resistance against byte changing and byte shifting. When there is a change in the chunk and the change has a value less than the maximum, it will only affect that chunk. The main drawback of this method is the requirement of rechecking all the bytes within the window when the window slides. This drawback makes Rabin-based CDC algorithms faster than LMC method because when the sliding window of Rabin slides, it only needs to subtract the most left byte and add the new byte into the hash. However, LMC needs all of the bytes in the window every time it slides the window.

AE is similar to the local maximum method because it treats a byte as a number. Treating the chunk as the windows allows AE to have a lower computational overhead than the LMC method. However, unlike the LMC method, AE puts the extreme-valued byte in the middle of the chunk. This makes AE less resistant to byte shifting. When there is a byte inserted at the fixed window, it will affect the chunk and the next chunk and might affect subsequent chunks. If we put the extreme-valued byte at the boundary of the chunk, inserting a byte will not affect the next chunk. Thus,

```
Algorithm 1: Algorithm for AE chunking
Input: input string, Str, left length of the input string, L;
Output: cut-point I;
Predefined values: window size, w;
function AEChunkning (Str, L)
        i=1;
        while (i<L)
                If Str[i].value<=max.value then
                        if i=max.position+w then
                                return i
                        end if

                else
                        max.value=Str[i].value
                        max.position = i
                end if
                i=i+1
        end while
end function
```

Fig. 2. The pseudo code for AE chunking.

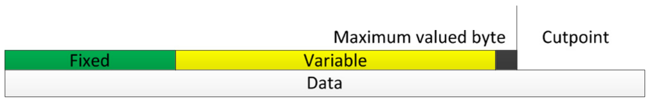


Fig. 3. RAM algorithm's windows configuration.

it minimizes the number of affected bytes. AE is capable of eliminating low entropy strings because AE has maximum chunk size. AE reach its maximum chunk size when it processes a long increasing sequence. The maximum chunk size is $256 \times (h - 1)$ where h is the length of the fixed window. The pseudo code for AE chunking is illustrated in Fig. 2.

In this paper, we proposed a new algorithm called Rapid Asymmetric Maximum (RAM) which improves the chunking throughput of AE by putting the extreme value at the boundary of the chunk. It has a low computational overhead which makes the algorithm faster than existing CDC algorithms. The low computation overhead of RAM reduces the cost of chunking process which makes chunking more attractive over AE for low performance devices such as mobile devices and IoT. Based on the ideal CDC algorithm criteria, we compared the existing CDC algorithms and our proposed algorithm in Table 1.

3. Rapid Asymmetric Maximum Algorithm (RAM)

In this section, we describe our proposed Rapid Asymmetric Maximum Algorithm (RAM) and analyze the chunking properties of the algorithm.

3.1. RAM algorithm design

With a goal of achieving low computational overhead and byte shift-resistant algorithm, we proposed a boundary version of AE, called RAM. RAM is similar to AE because it also uses two windows: fixed and variable-sized windows. The placement of the windows is, however different from AE. In RAM, the fixed-sized window is located at the beginning of the chunk and followed by the variable-sized window and the maximum-valued byte. As we can observe in Fig. 3, the maximum-valued byte is included in the chunk at the end of the chunk in the case of RAM.

The algorithm works by searching a byte with the maximum value in the fixed-sized window. If the byte next to the fixed-sized window has larger value than the one in the fixed-sized window, the byte is used as the maximum-valued byte and the cut-point is found. Otherwise, the algorithm moves to the next byte until it

```
Algorithm 2: Algorithm for RAM chunking
Input: input string, Str, length of the string, L;
Output: cut-point I;
Predefined values: window size, w;
function NAEChunkning (Str, L)
        i=1;
        while (i<L)
                if Str[i].value>=max.value then
                        if i>w then
                                return i
                        end if

                        max.value=Str[i].value
                        max.position = i
                end if
                i=i+1
        end while
end function
```

Fig. 4. The pseudo code for RAM chunking.

finds the larger byte as illustrated in the pseudo code of RAM in Fig. 4. Thus the algorithm's minimum chunk size is $w + 1$, where w is the size of the fixed-sized window.

Algorithm 2 in Fig. 4 shows the chunking used in our proposed RAM scheme. As can be seen in Algorithm 2, RAM reduces the computation time by searching the byte that is equal or larger than the current maximum value, while AE process all the bytes smaller or equal than the maximum-valued bytes. Since the probability that the next byte is smaller than the current maximum value is higher than the probability that the next byte is larger than the current maximum value, RAM enters the first condition less frequently than AE. This lowers RAM's overhead.

Fig. 5 illustrates an example for all chunking algorithms discussed in this paper with a window size of 5 bytes and 14 bytes of data. In the example, we used the same string of bytes to show how each algorithm works. Rabin-based chunking algorithm works by using a sliding window the sliding window starts at the beginning of the chunk as illustrated in Fig. 5(a). In the beginning of the example, the content of the sliding window is $0 \times 89594EA10D$. Since the hash of the window does not match the pattern, it slides to the next byte by removing the most significant byte, left shifting the window, and finally adding the new byte to the window. It keeps sliding and calculates the hash of the window until a hash value matched the pattern. In the example, the hash value of $0 \times 0D0A1AEA48$ matches the pattern. The sliding window can be included or excluded in the chunk. In the example, the sliding window is included inside the chunk.

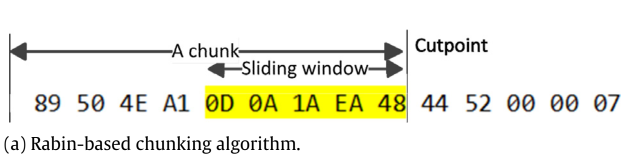
As seen in Fig. 5(b) LMC is similar to Rabin-based chunking algorithm because of the sliding windows. The sliding windows are composed of two fixed-sized windows and a byte located between the two fixed-sized windows. In the example, the cutpoint is found because the byte between the two fixed-sized windows is larger than the byte in both fixed-sized windows.

In Fig. 5(c), we can see an example for AE. Unlike LMC, AE scans each byte to find the cut-point. For each byte, AE compares the value of the scanned-byte with all bytes in the fixed-sized window. The fixed-sized window is always located at the right side of the scanned-byte. When the size of the byte is bigger than all bytes in the fixed-sized window, it found a cut point. Therefore, the value of the variable-sized window can be 0 byte when the first byte fulfills the condition. In the example, the algorithm scanned from 0×89 to $0 \times EA$. It found a cut-point when it scanned $0 \times EA$ because $0 \times EA$ is bigger than any byte in the fixed-sized window. The cut-point is located at the right side of the fixed-sized window.

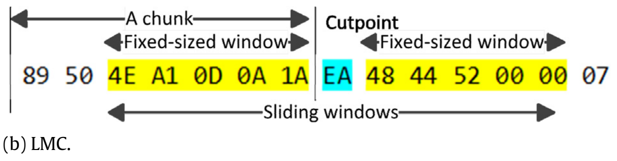
Similar to AE in terms of byte scanning, RAM scans each byte to compare the value with the maximum value of the window. RAM starts by finding the value in the fixed-sized window. In the

Table 1
A comparison between our proposed and existing CDC algorithms.

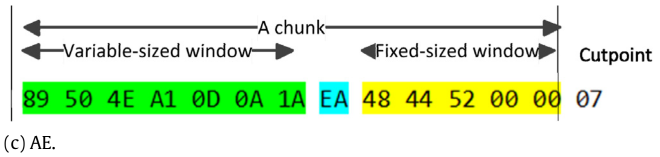
Algorithms	Content defined	Chunk variance	Low entropy string elimination	Throughput	Minimum chunk size	Maximum chunk size
Rabin	Yes	High	Yes, with limit	Low	Depending on limit	Depending on limit
LMC	Yes	High	No	High	w	$256 \times (h - 1)$
AE	Yes	Low	Yes	Low	$w + 1$	No limit
RAM	Yes	High, low with limit	Yes, with limit	Very high	$w + 1$	No limit (can be configured with limit)



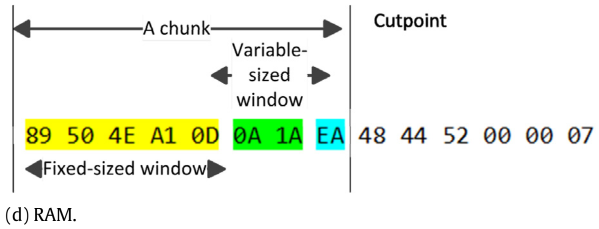
(a) Rabin-based chunking algorithm.



(b) LMC.



(c) AE.



(d) RAM.

Fig. 5. An example of how the algorithms work with a string of 14 bytes.

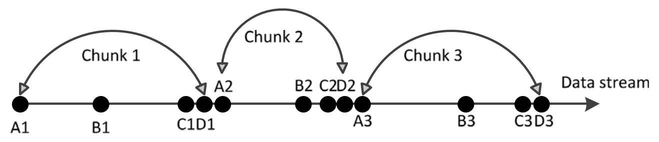
example, the maximum is 0 × A1. After it finished scanning the fixed-sized window, it compares each scanned-byte value with the maximum value. When the scanned-byte value is bigger than the maximum value, a cut-point is found. As can be seen in Fig. 5(d), 0 × EA is the cut-point because it is bigger than 0 × A1. The last scanned-byte is included in the chunk.

3.2. The properties of RAM

In this section, we address the properties of RAM subject to the criteria of an ideal CDC algorithm described in Section 2.2.

Content dependent. RAM chunks file based on the internal feature of the file, specifically based on the extreme values within the file. When it finds a byte larger than all of the bytes in the windows, it uses the byte position as a cut point. We use Fig. 6 as an example to show that RAM cut point is byte shift-resistant. First, we assume the change is smaller than the maximum value D. If there is a byte insertion or change at the fixed window [A1, B1], Chunk 1will be realigned and Chunk 2 will not be affected by the change as long as the maximum in [A1, B1] is not out of this window after the insertion. It also will not change the cut point if a byte is inserted in the variable-sized window [B1, C1] because Chunk 1 will realign, and Chunk 2 will not be affected. Second, we assume that the change is larger than the maximum value. If the change occurred in [A1, B1], the position of C1 will be changed until it finds a byte larger than the change. Thus, this may affect the following chunks. If the change occurred in [B1, C1], then D1 will be realigned to the changed location and thus may affect Chunk 2 and Chunk 3. The realignment will stop if the maximum byte in the fixed window region of the next chunk is bigger and still in the fixed window region. With this example, we can see the algorithm is able to realign the cut position while minimizing the number of chunks affected by the change.

Low chunk sizes variance and the ability to eliminate low entropy strings. RAM has a low probability of long chunk as explained in Section 3.3. However, low entropy string is a problem for RAM.



A = The start of a fixed sized window
B = The stop of a fixed sized window and the start of variable sized window
C = The stop of a variable sized window
D = Maximum valued byte
A~B = A fixed sized window
B~C = A variable sized window

Fig. 6. Byte insertion or shift and byte change example.

When the low entropy string starts at the beginning of the fixed-sized window, RAM is able to eliminate the low entropy string because the condition for a cut point is that the maximum-valued byte must be equal to or larger than the maximum in the fixed window. On the contrary, when there is a byte larger than any value in the low entropy string is in the fixed-sized window, the chunk size can become infinite because it cannot find a byte with larger value. To solve this, we can add a limitation on the maximum chunk size.

High throughput. From Algorithm 2 in Fig. 4, we can see that RAM has a low performance overhead. To prove that, we use the worst case of RAM, based on the number of comparisons. RAM uses while loop which takes $n + 1$ comparisons and two additional conditional branches which add $2n$ comparisons, where n is the length of the input data stream in bytes. Thus in total, it uses only $3n+1$ comparisons in the worst case scenario. Since the probability of finding a byte larger than the max is smaller than finding a smaller byte, on the average case it uses $2n + 1$ comparisons.

In the application, the probability that the value of the maximum byte in the fixed window being big is higher than the probability that the byte being small. We prove this by assuming that the data entries are random and splitting RAM into two parts: the fixed window part and the variable-sized part, where the fixed window part's length is w and the variable-sized part's minimum size is one.

Table 2
The 9 possible outcomes of 2 bytes with each byte $\in [0,\ 2]$.

00	10	20
01	11	21
02	12	22

Table 3
The probability of the maximum value being 251–255 with w of 100 with each byte $\in [0,\ 255]$.

Maximum	Probability
251	0.067926
252	0.100607
253	0.148782
254	0.219685
255	0.323884

Hypothesis 1. The probability of the maximum in the fixed window being small is lower than the probability of the maximum in the fixed window being larger.

Proof. We prove this using an example. Algorithm 2 explains that RAM will replace the current maximum if it finds a larger byte in the fixed window, where the value of a byte is $\in [0,\ 255]$. This will limit the number of elements in a byte to the number of current maximum because the elements larger than the maximum will replace the maximum. For example, we used a simpler condition with smaller space for the value of a byte. In the this example, a byte can have a value of 0, 1, or 2 ($\in [0,\ 2]$). For a window w with the size of 2 bytes, the probability of getting the maximum of 2 in the window is calculated as follows. The probability that the first byte being 2 is 1/3 and the complementary probability that the first byte is not 2 is 2/3. Thus, the probability of no 2 in 2 bytes is (2/3)^2 and the complementary probability that there is at least a 2 in the window is 1 − (2/3)^2 which is 5/9. As we can see in Table 2, there are 5 out of 9 events where there is at least a byte with a value of 2 in the window.

Next we calculate the probability of getting the maximum of 1 with the same configuration. Since RAM will replace the maximum with the larger byte if it finds one, we first consider 4 out of 9 events with no 2 in the window. Thus, the probability that the maximum in the window is 1 that is calculated as 1 − (1/2)^2 = 3/4. Adding the possibility that there is 2 in the element space, we get (3/4) × (2/3)^2 = 3/9 which is correct, as shown in Table 3. This example proves that for RAM, the probability of large maximum is higher than the probability of small maximum. To prove this, we compile the probability of 5 largest byte values in Table 3 with a configuration of $w = 100$ and each byte is $\in [0,\ 255]$. As we can see, the probability of being the maximum byte is smaller when the maximum byte is smaller. Because of the higher probability of getting a large maximum, RAM enters the second conditional branch less frequently than AE.

AE uses a comparison and two conditional branches to find a cut point. The conditional branches are used for to find the maximum byte position and the cut point. Thus, it uses $3n+1$ comparisons for each byte processed in worst case scenario. On the average case, it performs fewer than $3n+1$ comparisons because when it processes a byte larger than the current extreme-valued byte, it will do one fewer comparison. In reality, the probability that the next byte is smaller than the max is mostly higher than the probability of being smaller. Thus, it is still close to $3n + 1$.

LMC performs more comparisons than AE and RAM. For each byte, it needs to check all the bytes in the sliding window. If we use simple array filling and shifting, it needs to perform $2h$ comparisons for each byte in the data stream, where h is the size of the window. However, we can optimize the algorithm to work in a way similar to AE which reduces the number of comparison to 5 conditional branches. However, if the left window fails to meet the condition of being smaller than the local maximum, it needs to backtrack to $h - 1$ previous bytes in the worst case scenario. This makes the number of comparisons greater than 5 comparisons for each byte. In our tests, the simple array filling, shifting, and comparing method is more than 10 times slower than Rabin based chunking, and the optimized method only consumes slightly more time than Rabin based chunking.

Unlike AE, RAM, and LMC, Rabin-based chunking does more than byte comparisons. It needs to do hash calculations and array for the hash and the sliding window. For each byte, it needs to update the hash, do array shifting, and an array insertion. To update the hash, the previous hash value needs to be subtracted with the most left byte in the window multiplied by p^{h-1} where p is the prime and h is the length of the sliding window. Because of the need of subtracting the most left byte in the window and adding a new byte to the window, it needs to update the window for each byte. Additionally, it needs to compare the hash with the predefined pattern. These processes require more computation overhead than AE and RAM.

3.3. Probability of long chunk

It is desirable for the chunk variance to be as small as possible [15], and one of the variables to the chunk variance is the probability of a long chunk. A higher the probability of long chunks will cause higher chunk variance. When a long chunk is modified, the amount of new data to be stored will be increased more than when a small chunk is modified. Therefore, we want to minimize the amount of chunk's size variance and the probability of long chunk. In this section, we analyze the probability of long chunks of RAM, based on the equal probability of each byte being the maximum value.

Similar to Bjørner et al. [2], we let h be the horizon and the average chunk size is $2h + 1$. We estimate the probability of long chunk for RAM in the interval $[1,\ 2hM]$ and compare it with the other algorithms. M is the multiplier with a value larger than 0 but should not be too large.

In this estimation, we assume that each byte in the chunk is probabilistically independent. RAM uses one fixed-sized window of size a on the left. Thus, the minimum size of a chunk is $a+1$. Since each byte is probabilistically independent, each byte in the chunk has an equal probability of being the maximum point. Therefore, for a byte at position i, the probability for that byte in the range $[1,\ a+i]$ to be the maximum is $1/(a+i)$ and the complementary probability, that there is no maximum point in the range is $1 - 1/(a+i)$. Based on the probability that there is no maximum point in the range, we can compute the probability of no maximum in the interval of $[ch+a+1,\ ch+a+h]$ where c is a constant which is given by telescoping product:

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

The complementary probability that there is at least one maximum in the interval is $h/(h+a)$. Then we break the interval from $[1,\ 2hM]$ into $2M$ blocks with length h. Considering the jth subinterval $[(j-1)\,h+1,\ jh]$, the probability of no maximum

Table 4
Probability of having no cut point in long intervals.

M	AE	RAM	LMC	Rabin
	$\frac{1}{[(e-1)M]!}$	$\frac{1}{[2M]!}$	$\frac{2^{2M}}{(2M)!}$	e^{-M}
2	0.166667	0.041667	0.666667	0.135335
3	0.008333	0.001389	0.088889	0.049787
4	0.001389	2.48E−05	0.006349	0.018316
5	2.48E−05	2.76E−07	0.000282	0.006738
6	2.76E−07	2.09E−09	8.55E−06	0.002479
7	2.51E−08	1.15E−11	1.88E−07	0.000912
8	1.61E−10	4.78E−14	3.13E−09	0.000335

point is $h/(h+(j-1)\,h) = 1/j$. Because of the independence, we multiply the probability in the $2M$ subintervals. Thus, the probability of no maximum in the interval of $[1, 2hM]$ is:

$$\prod_{j=1}^{2M} \frac{1}{j} = \frac{1}{2M!}.$$

In the respective paper of the algorithms ([4] for AE and [2] for Rabin based chunking and LMC), the authors also analyzed the algorithms in a similar manner as in this section. In Table 4, we compiled various algorithm's probability of having long chunks in multiple intervals M.

4. Performance evaluation

In this section, we compare the performance of the chunking algorithms we discussed in Sections 2 and 3. We analyzed the chunking algorithms based on the properties discussed in Section 2.2. The properties of the resulting chunks from each algorithm are discussed in Section 4.2. In Section 4.3. we present our test results related to the chunking throughput and duplicate data found in the datasets.

The performances of the following algorithms have been evaluated:

1. Asymmetric extremum (AE) [4].
2. Our proposed algorithm (RAM).
3. RAM with limit on maximum chunk size (RAML).
4. Local maximum chunking (LMC) [2].
5. Rabin based chunking algorithm (Rabin) [13].

As can be seen in the above list, we added RAM with a limit in the test. The purpose of adding RAM with the limit is to show the performance of RAM when a limit is applied to the maximum size of a chunk. Additionally, it also shows the improvement of RAM when the chunk variance is reduced.

The performance comparison consists of three datasets. The datasets used in the tests are chosen to represent the use cases of the chunking algorithm. The first dataset is the compilation of multiple Linux distributions which have a lot of duplicate data in different locations in each file. The second dataset consists of 10 H.264 encoded videos of length 23 min each to simulate deduplication of media files in cloud storage. Lastly, the third dataset contains TCP dump files from [16] to represent deduplication network traffic. The dumps [16] contains 15 GB of data. However, we only used 9 GB of the data because of the limitation of our test system. The chunks metadata consumes a lot of memory and causes the program to stop working when the total number of chunks went over 10 million of chunks. We did not optimize the chunks management because our focus in this performance evaluation is the chunking performance. Table 5 summarizes the content of the three datasets.

4.1. Configurations

To make a fair comparison between the chunking algorithms, we configure each chunking algorithm with the same average chunk size. The average chunk size has a direct relation to the chunking performance. For example for network traffic, we need to make the average chunk size similar to the size of the average TCP packet because if the chunk size is too large, the algorithm might miss duplicate data. On the other hand, if the chunk size is too small, the metadata of the chunks will be bigger than the size of the duplicate data.

We used SHA1, a bloom filter, and a hash table for hashing and comparing chunks. A SHA1 hash value will be calculated for each chunk. The bloom filter and the hash table are used for duplicate finding process. Each hash value will be checked by using the bloom filter. Because of the nature of bloom filter, false positive is possible. Therefore, when the bloom filter says that the hash existed, the hash is also checked in the hash table. If the hash is not in the hash table, it means the chunk corresponding to the hash is not a duplicate chunk and the hash is added to the hash table. If bloom filter or hash table says the hash is new or not available in memory, and then the hash will be added to the bloom filter and the hash table. By using bloom filter, we can speed up the lookup process by not checking the hash table when bloom filter says the hash is new because bloom filter lookup time is shorter than a hash table lookup. In this paper, we focused on the chunking algorithm performance. Thus, this paper does not include the chunk management and storage processes. The data deduplication process in the performance evaluation is only done until duplicate finding process.

For AE, RAM, and LMC, we configured the windows size to get the same average chunk size. Since it is harder for Rabin based chunking to adjust the average chunk size, we ran Rabin chunking algorithm first to get the average chunk size. Then, we adjust the other three algorithms to match the average chunk size. From our experiment, changing the datasets does affect the average chunk size. This is due to the fact that some datasets may contain low entropy strings. Table 6 shows our configuration for the algorithms in our test. Additionally, we set maximum chunk size limitation for RAML to 4 times the window size. As for Rabin, there is no limitation to the minimum and maximum chunk sizes.

As for our test system, the specification are as follows: Intel i7 6700 3.6 GHz with 4 cores 8 threads, 16 GB DDR4, 2133 MHz, and Samsung 850 Evo 120 GB SSD. The SSD is connected through SATA3 interface.

4.2. Chunks properties

In this section, we compare the chunking algorithms based on the chunk properties. The chunks produced by each chunking algorithm are analyzed based on the chunk size distribution and the chunk variance. The chunk size distribution will be displayed in a histogram while the chunk variance is calculated using $\sigma^2 = \frac{1}{N}\sum_{i=1}^{N}(x_i - \bar{x})$. As discussed in Section 2.2, we want the chunk variance to be as low as possible.

4.2.1. Dataset 1: Linux distribution

Dataset 1 contains Linux distribution operating system installation images which have a lot of similar contents and low entropy strings. This dataset is used to show the performance of a chunking algorithm in finding duplicates between files that have similar contents. The results for this dataset are compiled in Table 7 and displayed in Fig. 7.

There are three interesting things that we can observe: chunk variance, chunk size distribution, and the duplicates found by the algorithms. The chunk variance data shows that AE has the lowest

Table 5
The datasets used in the tests.

Dataset	Content	Size (bytes)
Dataset 1: Operating system installation image	Arch Linux 2016.08.01 dual [17] Chromium OS (2016.08.09) [18] Debian 8.5.0 AMD64 XFCE [19] ElementaryOS 0.3.2 StableAMD64 [20] Linux Mint 18 Cinnamon 64 bit [21] Lubuntu 14.04.5 Desktop AMD64 [22] Solus 1.2 [23] Ubuntu16.04.1 Desktop AMD64 [24]	7,601,234,473
Dataset 2: Media files	10 × 23 min H.264 videos	5,672,289,025
Dataset 3: Network traffic	Data Capture from National Security Agency (NSA) CDX 2009 [16]	9,744,031,059

Table 6
The configuration for each chunking algorithm.

Dataset	AE (window size)	RAM (window size)	RAML (window size)	LMC (fixed window size)	Rabin
1	770	764	764	198	1k chunk size
2	770	770	770	203	1k chunk size
3	770	770	770	197	1k chunk size

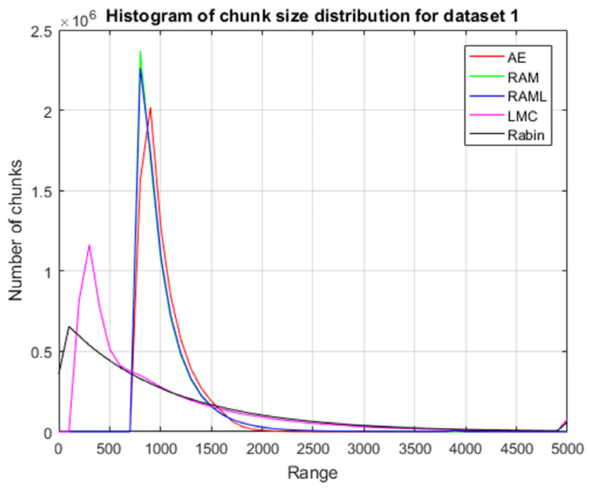


Fig. 7. Histogram of chunk size distribution properties for dataset 1.

chunk variance compared to others because of the ability of AE to eliminate low entropy strings. As for RAM, the chunk variance is worse than other algorithms. This is caused by the lack of the maximum chunk size of RAM which affect the low entropy string elimination performance. The limit on RAML reduces RAM chunk variance and improves its low entropy string elimination. This can be seen in the 1.8% improvement in duplicates found for RAML over RAM. LMC and Rabin have high chunk variance because of high counts of long chunks.

The chunk size distribution of RAM is similar to RAML and AE. The chunks are mostly narrowly distributed near the mean as illustrated in Fig. 7. We can also see that the chunk sizes for RAM and RAML are more concentrated at below the mean because of the higher probability of finding the cut points, while the chunk size distributions for LMC and Rabin are wider than others.

The duplicates found by the algorithms for dataset 1 are not much different. We observed a difference of only 2.1% between the maximum and minimum. All of the tested algorithms were able to find 17%–18% duplicate between files. The low entropy string elimination capability of AE and RAML does help to increase the duplicate detection.

4.2.2. *Dataset 2: media files*

With the compressed H.264 video files in dataset 2, we saw different results. Dataset 2 contents are compressed video files encoded in H.264. Therefore, the number of low entropy strings is lower in this dataset compared to the other two datasets. The results are compiled in Table 8 and chunk size distribution is illustrated in a histogram in Fig. 8.

In this dataset, RAM has lower variance compared to the dataset 1 because the dataset contains fewer low entropy strings. Another interesting point in this dataset is RAML's result. The limit on RAML affects the content defined property of RAM and reduces the number of duplicates found in this dataset. Thus, adding the limit to RAM is only encouraged for a dataset with a high amount of low entropy strings.

Similar to the previous dataset, the value of chunk size variances for LMC and Rabin are larger than AE, and RAML. On the contrary, we find that LMC which has the high chunk variance detected more duplicates compared to the other algorithms despite the larger chunk size variance. As we can see in Fig. 8, the chunk size distribution of LMC peaked at 300 bytes, which is not close to the minimum chunk size of LMC which is 203 bytes. Although the chunk distribution of LMC is not better than AE, RAM, and RAML, LMC eliminated more duplicates than the other algorithms and netting 12.8% more duplicates found compared to Rabin.

4.2.3. *Dataset 3: network dump files*

The test on dataset 3 represents the performance of a chunking algorithm when processing network traffic data. The content of dataset 3 is not compressed and it is possible to find low entropy strings in this dataset. The size of the packet is not defined. However, from our knowledge the default MTU size for most machines with traditional Ethernet is 1500 bytes. Similar to the other two datasets, we compiled the results in Table 9 for chunk size information, and Fig. 9 for the histogram of the chunk size distribution.

The chunk size variance for dataset 3 shows a result similar to that using dataset 2. RAM is still behind AE and ahead of LMC and Rabin in terms of chunk size variance. Although not significant, adding a limit to RAML improves the number of duplicates found by RAM. The main cause of the improvement due to limit is the existence of low entropy string in the dataset.

Table 7
Chunks properties for each algorithm for dataset 1.

Range	AE	RAM	RAML	LMC	Rabin
[0,1000]	4,278,622	4,643,773	4,595,328	4,865,576	4,606,661
[1000,2000]	3,101,223	2,660,591	2,718,236	1,645,625	1,733,733
[2000,3000]	13,790	85,032	89,629	550,893	655,020
[3000,4000]	121	7,203	16,046	198,763	246,085
[4000, 5000]	19	2,008	0	77,425	93,209
[5000, 6000]	2	829	0	33,040	35,183
[6000, 7000]	1	396	0	16,094	13,376
[7000, 8000]	0	235	0	8,188	5,141
[8000, 9000]	0	130	0	5,000	2,170
[9000, 10000]	0	115	0	2,979	823
[10000, 11000]	0	0	0	4	1
Total number of chunks	7,393,778	7,400,312	7,419,239	7,403,587	7,391,402
Mean	1,030	1,030	1,020	1,025	1,030
Variance	48,700	15,400,000	88,300	12,200,000	10,800,000
Duplicates found (bytes)	1,024,444,745	1,006,199,523	1,024,635,083	1,003,515,719	1,011,368,274

Table 8
Chunks properties for each algorithm for dataset 2.

Range	AE	RAM	RAML	LMC	Rabin
[0, 1000]	3,117,878	3,403,698	3,403,746	3,495,518	3,448,464
[1000, 2000]	2,392,131	2,138,311	2,138,316	1,314,110	1,301,558
[2000, 3000]	4,142	40,001	40,021	460,609	489,890
[3000, 4000]	1	951	1,042	164,826	184,840
[4000, 5000]	0	73	0	59,608	69,392
[5000, 6000]	0	5	0	21,271	26,165
[6000, 7000]	0	0	0	7,815	9,970
[7000, 8000]	0	0	0	2,759	3,610
[8000, 9000]	0	11	0	1,032	1,416
[9000, 10000]	0	1	0	372	559
[10000, 11000]	0	0	0	0	0
Total number of chunks	5,514,152	5,583,051	5,583,125	5,527,920	5,535,864
Mean	1,029	1,016	1,016	1,026	1,025
Variance	43,466	62,075	61,930	921,860	1,049,000
Duplicates found (bytes)	2,128,338	2,030,863	2,021,033	2,259,134	2,002,697

Table 9
Chunks properties for each algorithm for dataset 3.

Range	AE	RAM	RAML	LMC	Rabin
[0, 1000]	5,435,437	5,768,110	5,783,201	6,280,039	5,752,016
[1000, 2000]	3,961,755	3,438,323	3,467,617	2,145,609	2,170,580
[2000, 3000]	25,264	120,281	123,282	652,904	836,368
[3000, 4000]	177	15,750	35,747	238,053	327,584
[4000, 5000]	13	5,479	0	99,870	127,361
[5000, 6000]	1	3,155	0	42,968	50,628
[6000, 7000]	0	1,795	0	23,496	20,390
[7000, 8000]	0	971	0	13,456	8,399
[8000, 9000]	0	793	0	7,391	3,931
[9000, 10000]	0	896	0	5,058	1,973
[10000, 11000]	0	2	0	8	1
Total number of chunks	9,422,647	9,355,555	9,409,847	9,508,852	9,299,231
Mean	1,034	1,041	1,036	1,023	1,048
Variance	52,992	306,650	104,550	5,386,300	1,185,100
Duplicates found (bytes)	328,492,971	276,568,141	276,827,741	257,244,341	339,501,545

In terms of duplicates found, Rabin chunking algorithm found more duplicate data compared to other algorithms in dataset 3, while RAM is behind AE in terms of duplicates found. We suspect that no limit on Rabin helps Rabin to detect duplicate on the network traffic because of the packet size randomness. The chunk size distribution in Fig. 8 shows all algorithms produced similar chunk size distributions to the previous datasets.

Overall, RAM performance is between AE, LMC, and Rabin in terms of chunk qualities. Additionally, we found that, depending on the dataset, some chunking algorithms are better than others. For example, LMC is better for compressed file and Rabin is more suitable for network traffic files. This fact is similar to what has been reported in [25].

4.3. Chunking throughput

This section compares CDC algorithms based on the chunking throughput and Bytes Saved per Second (BSPS). The purpose of this section is to find the algorithm has the best balance between chunking throughput and the number duplicates found. The chunking throughput is calculated by dividing the amount of data processed and the amount of time consumed for chunking the files. BSPS was first used by Yinjin et al. [26] as a performance metric

Table 10
Chunking throughput and byte saved per second.

	AE	RAM	RAML	MAXP	Rabin
Dataset 1 processing time (s)	19.700	13.801	18.219	93.598	72.678
Dataset 2 processing time (s)	15.314	10.289	12.658	63.601	61.508
Dataset 3 processing time (s)	26.037	17.374	22.076	112.154	103.437
Throughput dataset 1 (MBps)	385.843	550.761	417.207	81.211	104.588
Throughput dataset 2 (MBps)	370.407	551.279	448.107	89.185	92.221
Throughput dataset 3 (MBps)	374.243	560.851	441.379	86.881	94.202
BSPS dataset 1 (MBps)	52.001	72.906	56.239	10.722	13.916
BSPS dataset 2 (MBps)	0.139	0.197	0.160	0.036	0.033
BSPS dataset 3 (MBps)	12.617	15.919	12.540	2.294	3.282

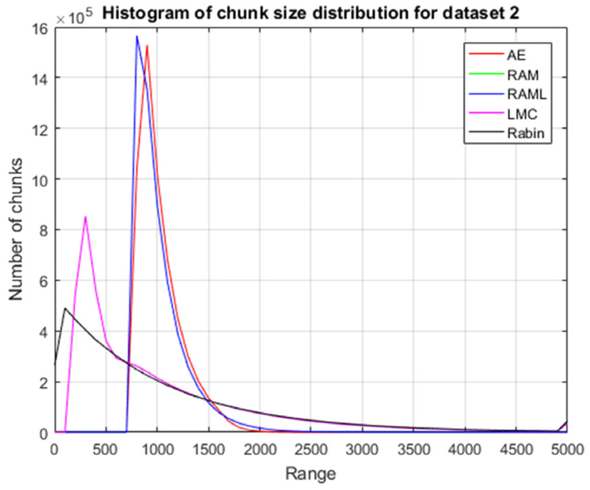


Fig. 8. Histogram of chunk size distribution properties for dataset 2.

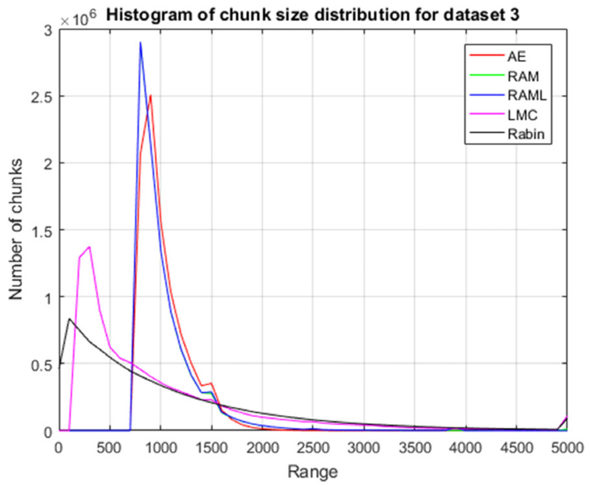


Fig. 9. Histogram of chunk size distribution properties for dataset 2.

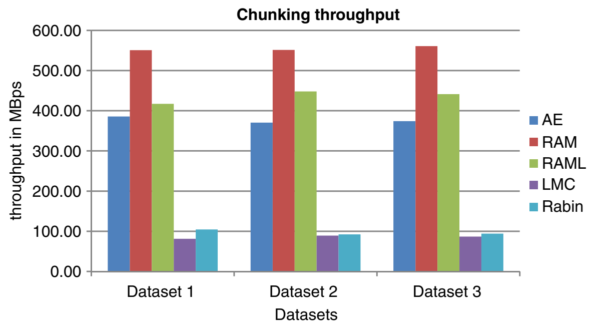


Fig. 10. Chunking throughput for the five algorithms.

for deduplication systems. The BSPS is calculated by dividing the amount of duplicates found by the amount of file processed and multiplying the result with the throughput.

$$Throughput = \frac{dataset\ size\ (bytes)}{time\ for\ chunking\,(s)}$$

$$Byte\ Saved\ per\ Second = \frac{duplicate\ found\ size\ (bytes)}{original\ dataset\ size\ (bytes)} \times throughput.$$

The results for dataset 1, dataset 2, and dataset 3 for the five chunking algorithms considered in this paper are compiled in Table 10, the chunking throughput for the five algorithms is illustrated in Fig. 10, and the byte saved per second is shown in Fig. 11. The chunking time excludes the read time from the drive because our main focus is the chunking performance.

Our results shows that RAM has higher throughput compared to other algorithms. Although it found fewer duplicate data compared to AE in all datasets, RAM has 42%–49% higher throughput compared to AE for all datasets. The high throughput of RAM helps it to achieve higher byte saved per second compared to AE by 26%–40% depending on the dataset. Overall, RAM performs better in terms of throughput compared to other algorithms.

5. Discussion

As a chunking algorithm, RAM performs well compared to other chunking algorithms in terms of produced chunk characteristics. The chunk size distribution produced by RAM is narrower than most of other algorithms. Although RAM has higher chunk size variance because of the lack of maximum chunk size, the chunks have similar chunk distributions to the chunks produced by AE. The loss on higher chunk size variance makes RAM perform better in terms of chunking throughputs. Generally, RAM improved the performance of AE on byte saved per second.

In Section 4.2, we found that lower chunk size variance does not mean that the algorithms will perform worse than the algorithm with lower chunk size variance, which is shown in Section 4.2.1 where RAML with worse chunk size variance compared to AE detected more duplicates, in Section 4.2.2 where LMC outperform AE in terms of duplicate detected, and in Section 4.2.3 where AE once again outperformed by other algorithms with worse chunk variance. This means the content dependent capability of a CDC algorithm is more important than chunk size variance. This is proven by fixed-sized chunking. Fixed-sized chunking has zero chunk size variance but it is not content dependent which makes fixed-sized chunking performs worse in terms of duplicate detection compared to CDC algorithms. Another proof that chunk variance does not significantly affect the duplicate detection is our test in Section 4 for RAML, which is another version of RAM with a limit on the maximum chunk size. For dataset 2, adding limits helped reducing the chunk variance of RAM. However, it also decreased the number of duplicates found because the limit makes it less content defined. Therefore, chunk variance is an appropriate metric to compare CDC algorithms when the algorithms have similar content defined characteristic.

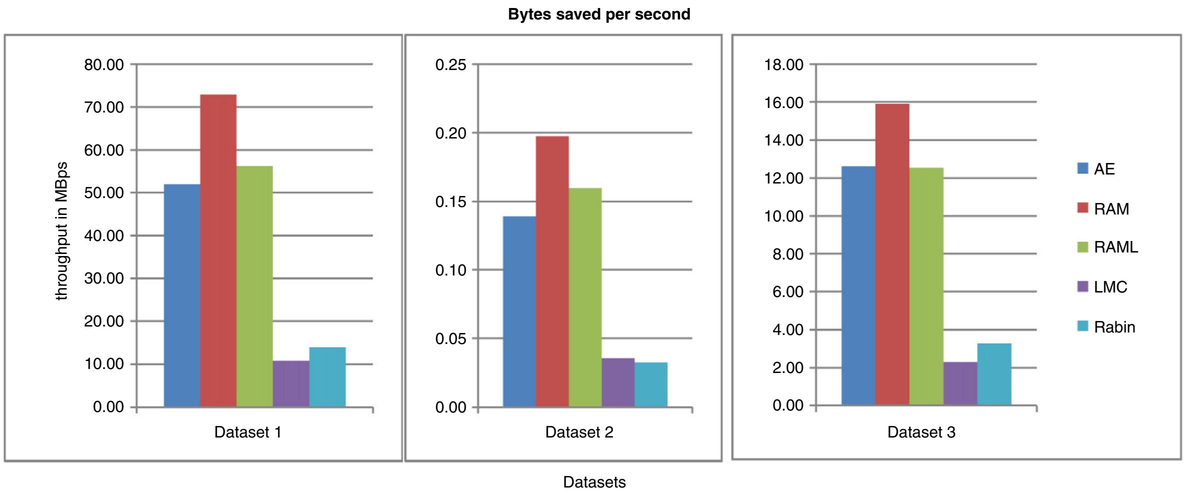


Fig. 11. Bytes saved per second for the five algorithms.

The results in Section 4.3 indicate that RAM is preferable for low performance devices such as mobile devices and IoT, or applications where computing time and storage is important. Additionally, RAM is also useful for client side data deduplication or remote differential compression because with lower computational overhead, RAM can reduce the stress on client hardware.

In the future, we will address the high chunk variance of RAM to improve the duplicate finding performance. We will also study how chunk size variance affects the duplicate finding performance.

6. Conclusion

In this paper, we have discussed the importance of content defined chunking for multiple applications and why it is better than fix-sized chunking. We proposed a new chunking algorithm, called Rapid Asymmetric Maximum (RAM) based on asymmetric chunking algorithm. We analyzed and compared RAM with other chunking algorithms. Our results show that RAM offers lower computational overhead compared to other CDC algorithms.

The main advantage of RAM is its low computation overhead which allows high chunking throughput. The high chunking throughput comes at the cost of higher chunk variance. The higher chunk variance produced by RAM is negligible compared to the performance gain over other chunking schemes based on local maximum chunking. In some cases, RAM offers 26%–40% higher byte saved per second compared to the other chunking algorithms.

Acknowledgment

This work was supported by the National Research Foundation of Korea under Grant 2016R1D1A1A09916932.

References

[1] A. Tridgell, P. Mackerras, The rsync algorithm, Jt. Comput. Sci. Tech. Rep. Ser., 1996.
[2] N. Bjørner, A. Blass, Y. Gurevich, Content-dependent chunking for differential compression, the local maximum approach, J. Comput. System Sci. 76 (3–4) (2010) 154–203.
[3] M.O. Rabin, Fingerprinting by random polynomials, no. TR-15-81, 1981, pp. 15–18. Center for Research in Computing Techn., Aiken Computation Laboratory, Univ..
[4] Y. Zhang, H. Jiang, D. Feng, W. Xia, M. Fu, F. Huang, Y. Zhou, AE□: An asymmetric extremum content defined chunking algorithm for fast and bandwidth-efficient data deduplication, in: 2015 IEEE Conference on Computer Communications, INFOCOM, 2015, pp. 1337–1345.
[5] R.N.S. Widodo, H. Lim, M. Atiquzzaman, SDM□: Smart deduplication for mobile cloud storage, Future Gener. Comput. Syst. (2016).
[6] C. Yang, J. Ren, J. Ma, Provable ownership of files in deduplication cloud storage, Secur. Commun. Netw. (July 2013) (2015) 2457–2468.
[7] K. Jin, E.L. Miller, The effectiveness of deduplication on virtual machine disk images, in: Proc. SYSTOR 2009 Isr. Exp. Syst. Conf., no. May, 2009, pp. 1–12.
[8] S.H. Kim, J. Jeong, J. Lee, Selective memory deduplication for cost efficiency in mobile smart devices, IEEE Trans. Consum. Electron. 60 (2) (2014) 276–284.
[9] J. Ahn, D. Shin, Optimizing power consumption of memory deduplication scheme, in: 2014 International Symposium on Consumer Electronics, ISCE, 2014, pp. 1–2.
[10] A. Bremler-barr, S.T. David, Y. Harchol, D. Hay, Leveraging traffic repetitions for high-speed deep packet inspection, in: 2015 IEEE Conference on Computer Communications, INFOCOM, 2015, pp. 2578–2586.
[11] D. Teodosiu, N. Bjørner, Y. Gurevich, M. Manasse, J. Porkka, Optimizing File Replication over Limited-Bandwidth Networks using Remote Differential Compression, Microsoft Res. TR-2006-157, 2003, pp. 1–16.
[12] A. Ruppin, D. Peri, Y. Ben-Natan, G.S. Shidlansik, M. Liram, O. Saporta, P. David, U. Yulevich, T. Choi, System and Method for Secure Synchronization of Data Across Multiple Computing Devices, US 8,984,582 B2, 2015.
[13] B.H. Bloom, Space / time trade-offs in hash coding with allowable errors, Commun. ACM 13.7 13 (7) (1970) 422–426.
[14] J. Wei, H. Jiang, K. Zhou, D. Feng, MAD2: A scalable high-throughput exact deduplication approach for network backup services, in: 2010 IEEE 26th Symp. Mass Storage Syst. Technol. MSST2010, 2010.
[15] H. Khuern, A Framework for Analyzing and Improving Content-Based Chunking Algorithms, Tech. Rep. TR, Hewlett-Packard Labs, 2005, pp. 1–10.
[16] Data Capture from National Security Agency (NSA), West Point United States Military Academy, 2009. [Online]. Available: http://www.westpoint.edu/crc/SitePages/DataSets.aspx [Accessed: 09.08.16].
[17] J. Vinet, A. Griffin, Arch Linux, 2016. [Online]. Available: https://www.archlinux.org/download/ [Accessed: 09.08.16].
[18] Google, Chromium OS, Google, 2016. [Online]. Available: https://www.chromium.org/chromium-os [Accessed: 09.08.16].
[19] SPI Inc., Debian, 2016. [Online]. Available: https://www.debian.org/CD/torrent-cd/ [Accessed: 09.08.16].
[20] D. Foré, Elementary OS, 2016. [Online]. Available: https://elementary.io/ [Accessed: 09.08.16].
[21] C. Lefebvre, Linux Mint, 2016. [Online]. Available: https://www.linuxmint.com/edition.php?id=217 [Accessed: 09.08.16].
[22] M. Behling, J. Lavergne, Lubuntu, 2016. [Online]. Available: http://lubuntu.net/ [Accessed: 09.08.16].
[23] Solus, Solus Project, Solus. [Online]. Available: https://solus-project.com/download/ [Accessed: 09.08.16].
[24] M. Shuttleworth, Ubuntu, Ubuntu, 2016. [Online]. Available: http://www.ubuntu.com/download/desktop [Accessed: 09.08.16].
[25] D. Meister, A. Brinkmann, Multi-level comparison of data deduplication in a backup scenario, in: Proc. SYSTOR 2009 Isr. Exp. Syst. Conf., 2009, p. 8.
[26] F.X. Yinjin, H.J.J, N. Xiao, L. Tian, F. Liu, AA-dedupe□: An application-aware source deduplication approach for cloud backup services in the personal computing environment, in: 2011 IEEE International Conference on Cluster Computing AA-Dedupe, 2011, pp. 112–120.

Ryan N.S. Widodo received his B.E. degree from the Department of Electrical Engineering, Petra Christian University in 2013. He is expected to complete his M.Sc. from Graduate School of Ubiquitous IT, Dongseo University in Feb. 2017. His research interests include mobile computing, cloud computing, data compression, computer networks.

Hyotaek Lim received his B.S. degree in Computer Science from Hongik University in 1988, the M.S. degree in computer science from POSTECH and the Ph.D. degree in computer science from Yonsei University in 1992 and 1997, respectively. From 1988 to 1994, he had worked for Electronics and Telecommunications Research Institute as a research staff. Since 1994, he has been with Dongseo University, Korea, where he is currently a professor in the Division of Computer Engineering. His research interests include ubiquitous and mobile networking, cloud computing, and storage area networks.

Mohammed Atiquzzaman obtained his M.S. and Ph.D. in Electrical Engineering and Electronics from the University of Manchester (UK). He currently holds the Edith Kinney Gaylord Presidential professorship in the School of Computer Science at the University of Oklahoma, and is a senior member of IEEE. Dr. Atiquzzaman is the Editor-in-Chief of Journal of Networks and Computer Applications and has served on the editorial boards of several journals and on the review panels of several funding agencies. His research interests include communications switching, transport protocols, wireless and mobile networks, ad hoc networks, satellite networks, Quality of Service, and optical communications.