IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received June 19, 2019, accepted June 25, 2019, date of publication July 1, 2019, date of current version July 17, 2019.

*Digital Object Identifier 10.1109/ACCESS.2019.2926195*

# MII: A Novel Content Defined Chunking Algorithm for Finding Incremental Data in Data Synchronization

**CHANGJIAN ZHANG[1], DEYU QI[1], ZHE CAI[1], WENHAO HUANG[1,2], XINYANG WANG[1], WENLIN LI[1], AND JING GUO[1]**

[1]Department of Computer Science and Engineering, South China University of Technology, Guangzhou 510000, China
[2]Guangzhou Mingsen Technology Company Ltd., Guangzhou 510000, China

Corresponding author: Changjian Zhang (zhang03091354@163.com)

This work was supported in part by the National Natural Science Foundation of China under Grant 61070015, in part by the Guangdong Provincial Frontier and Key Technology Innovation Special Funds Major Science under Grant 201802020035, and in part by the Technology Project under Grant 2014B010110004.

**ABSTRACT** In the data backup system, to reduce the bandwidth and processing time overhead caused by full backup technology during data synchronization between backups and source data, incremental backup technology is emerging as the focus of academic and industrial research. It is key but poorly-solved to find the incremental data between backups and source data for incremental backup technology. To find out the incremental data during the backup process, here, in this paper, we propose a novel content-defined chunking algorithm. The source data and backup data are chunked into some small chunks in the same way with the variable length. Then, by comparing whether a chunk of source data is different from any of the chunks in backup data, we can evaluate whether the chunk of source data is incremental data. By experiments, the chunking algorithm in this paper is compared to other ones which are the classical or state-of-the-art algorithms. The experimental results show that the incremental data found by this algorithm can be reduced by 13%–34% compared to the others with the same chunk throughput.

**INDEX TERMS** Data synchronization, chunking algorithm, data backup, increment.

## I. INTRODUCTION

Chunking algorithm can avoid dealing with a whole large file by chunking the large file into several small chunks and dealing with a small chunk each time, so as to achieve the desired purpose [1], [2].

### A. CHUNKING ALGORITHMS CLASSIFICATION

According to whether the length of the chunks is fixed, the chunking algorithm can be classified into fixed-length chunking algorithm and content-defined chunking algorithm(CDC) [3]. Fixed-length chunking, as the name implies, is to divide the file into chunks with fixed length. The advantage of fixed-length chunking algorithm is that the process is simple and easy to understand. The speed of chunking is very fast. It has higher chunk throughput. Its disadvantage is quite obvious. Because the length of all chunks is fixed in the process of chunking, if a byte in the file is shifted, it will cause all the subsequent chunks to be different from the original chunks, which means the ability to resist byte shifting is poor [4]. For content-defined chunking, the length of the chunks is not fixed, but based on the content of the file. While reading files sequentially in the form of binary stream, a cut point is formed only when a data interval read meets the preset conditions here. The resistance to byte shifting of content-defined chunking algorithms is improved greatly [5], but it also makes the process difficult to understand and the chunk throughput will decrease accordingly. Meanwhile, the big chunk variance will become a disadvantage in specific applications.

The associate editor coordinating the review of this manuscript and approving it for publication was Victor Sanchez.

### B. APPLICATION OF CHUNKING ALGORITHMS

Chunking algorithms have been widely used in many fields.

In the field of network transmission, data needs to be divided into several chunks by Chunking algorithms to avoid the situation that one single file is too big to transmit, and then deliver the chunks one by one [6]–[10].

In the data storage system, because of the large amount of duplicate data, the utilization of storage space is significantly reduced. Data deduplication technology, using chunking algorithms, reduces the storage of duplicate data and improves the utilization of storage space by dividing data into small chunks and storing the same chunks only once [11]–[14].

In the data synchronization system, the key and difficulty of incremental synchronization are how to find the incremental data between two files. With the help of chunking algorithms, two files are chunked into small chunks with the same method. Comparing the hash values of the chunks between the two files, the unequal chunks(i.e. incremental data) are found [15]–[17].

In the cache system, in order to get the highest cache hit rate, it is always necessary to save as much data as possible in the cache. Because only a small part of the big files is often used in most cases, it is also necessary to use chunking algorithms to chunk the files, and extract the chunks with high access rate and put them into the cache [18]–[22].

In the field of text recognition, when parsing natural language texts, a sentence needs to be chunked to extract the subject, predicate, object and other key phrases in it, and then get the true meaning of the sentence by grammar analysis [23]–[27].

### C. INCREMENTAL SYNCHRONIZATION

Data synchronization is the vital part in backup systems and multi-server platforms [28]. Incremental synchronization reduces a lot of bandwidth overhead by finding incremental data and only synchronizing these data, which has attracted much attention in academia and industry. The application of incremental synchronization mainly includes relational database system and cloud disk system.

#### 1) RELATIONAL DATABASE SYSTEM

When synchronizing incremental data are stored in database, the methods of discovering incremental data include database triggers [29], log-based document parsing [30], programming control and so on.

In database triggers mode, the use of triggers will greatly increase the complexity of database structure. In the case of huge amount of data tables, in order to ensure the discovery of all the incremental data, many triggers have to be added, which makes the structure of the database extremely complex and poses a great challenge to database maintenance. Once some triggers are omitted in maintenance, the whole incremental synchronization system will not work.

The method of log-based document parsing finds the incremental data by parsing the log document with the instructions given by the database company. Since some databases(e.g.,oracle), do not publish the format of log documents, this method is not universal.

For programming control, the incremental synchronization is realized by completely recording all the operations on the database in the code, and then applying all the operations to the target server that needs to be synchronized. This approach is very laborious and time-consuming for programmers, and a small programming error can make the whole system broken.

#### 2) CLOUD DISK SYSTEM

In the Cloud Disk System, the incremental synchronization of files is always used in distributed systems [28], [31]–[33]. By monitoring the changed directories and files, only the changed files and directories are dealt with when synchronizing. Suppose the source file is named as *fileSrc* and the target file is named as *fileDes*. First of all, *fileDes* is divided into chunks according to the fixed length; secondly, both the strength and weakness hash values are calculated for every chunk; thirdly the checksum, which is composed of the hash values, is transmitted to the source file server; then the source file server calculates both the strength and weakness hash values of the data in a sliding window with the same fixed length; then compare the hash values with the ones in the checksum, find out the changed chunks, transfer the changed chunks to the target file server, save the changed chunks in the target file server and finally realize incremental synchronization [16].

## II. BACKGROUND AND MOTIVATION

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

### A. BACKGROUND

The research of data chunking algorithms is mainly focused on content-defined chunking algorithm in academia, but anyway there are some researches on improving the performance of fixed length chunking algorithm such as DSFSC [34], which implements the chunking process from dual side of data at the same time.

As to CDC, the most classical algorithm is based on Rabin fingerprint chunking algorithm, which was proposed by MO Rabin [35]. Since the match rate of Rabin fingerprint is probably low, which may affect the stability of chunk size, Raju et al. proposed a algorithm to reduce Rabin's chunk-size variance with two divisors instead of one [36]. To speed the chunking process up, Won et al. developed a multithreaded variable size chunking method, which exploits the multicore architecture of the modern microprocessors [37]. Another multithread content based file chunking system is given by Tang et al. in CPU-GPUPU heterogeneous architecture [38]. To improve the parallel performance, a two-stage parallel CDC is proposed by Ni et al. to spit up the parallel chunking process into two stages [39]. To locate the changed data precisely, Romanski et al. proposed a algorithm which de-duplicates with big chunks as well as with their subchunks using a de-duplication context containing subchunk-to-container-chunk mappings [40].

Currently, the latest data chunking algorithms in academic research mainly include LMC(Local Maximum Chunking), AE(Asymmetric Extremum) and RAM(Rapid Asymmetric

Maximum) algorithms. The following contents give a brief description of these four algorithms respectively.

#### 1) RABIN CHUNKING ALGORITHM

The Rabin Chunking algorithm reads the data as a byte stream and calculates the Rabin fingerprint value of the data falling into a fixed-length sliding window. By comparing the fingerprint value to the preset fingerprint value, we can form a cut point here if these two values are the same. Otherwise, the sliding window is moved one byte forward to continue the calculation and comparison until a cut point is found.

#### 2) LMC CHUNKING ALGORITHM

LMC also reads the data as a byte stream. The difference is that LMC algorithm sets up two fixed-length windows and there is a one-byte-length interval between the two windows. Two windows together with the byte of this interval form a large sliding window. By comparing the byte value of this interval to all byte values in the two fixed-length windows, a cut point is formed here if the byte value is bigger than or equal to all byte values in the two windows. Otherwise, the large sliding window will be moved one byte forward and continue the previous process until a cut point is found [41].

#### 3) AE CHUNKING ALGORITHM

AE algorithm also reads the data as a byte stream. This algorithm consists of a larger sliding window formed of a fixed-length window and the byte next to the window. By comparing the byte value of the last byte in the sliding window to the byte values of all bytes in the fixed-length window, a cut point is formed here if the byte value is bigger than or equal to the others. Otherwise, the sliding window is moved one byte forward to continue the previous process until a cut point is found [42].

#### 4) RAM CHUNKING ALGORITHM

RAM(Rapid Asymmetric Maximum) algorithm reads the data as a byte stream as well, but it does not set the sliding window at the end of the chunk. It sets a fixed window instead at the front of the chunk, and then sets a sliding window with only one byte, which is next to the fixed window. By comparing the byte value in the sliding window to all the byte values in the fixed window, a cut point is formed here if the byte value is bigger than or equal to the others. Otherwise, the sliding window is moved one byte forward to continue the previous process until a cut point is found [43].

### B. CHALLENGES AND MOTIVATION

CDC offers more benefits than fixed-length chunking. There are five challenges, which are also the performance indexes, of the CDC algorithms offered by Zhang et al. [42].

1) Content dependence. The chunking algorithm must be realized to decide the cut point based on the content, which makes it resistant against the byte shifting problem [44].
2) Low chunk size variance. Because the output chunks of the chunking algorithm will be stored in disk in the process of deduplication, the high size variance will affect the efficiency of storage and it also affects the performance of deduplication [5].
3) Ability to eliminate low entropy strings. Low entropy strings are strings which consist of repetitive bytes or patterns. This challenge means it is preferable for the algorithm to be able to eliminate the redundancy within this kind of string.
4) High throughput [45]. The algorithm is asked to receive a high throughput of chunks.
5) Performance. When you use a CDC algorithm, you always have a purpose. This challenge is to make sure you can accomplish the target of your purpose as well as possible.

Existing data chunking algorithms are mostly used in data deduplication. Since the chunks obtained by the chunking algorithm needs to be stored in the physical disk [46]–[48], the chunk length needs to be considered to improve the storage utilization of the disk as much as possible [49]. To achieve the stability of the chunk length, partial ability of resistance against the byte shifting needs to be sacrificed.

However, the ability of resistance against the byte shifting is more important than the stability of the chunk length when chunking algorithms are used to find incremental data. In the incremental backup system, the chunks obtained by the data chunking algorithm are only used to find the incremental data [50], [51] and will not be stored in the physical disk, so the requirement of the stability of the chunk length is not particularly necessary.

To improve the performance of finding less incremental data between two files, chunking algorithms must have a strong resistance against the byte shifting. Only in this way, the changed blocks will affect the same blocks minimally during the chunking.

By sacrificing the stability of the chunk length, we propose a novel chunking algorithm to improve the resistance against byte shifting, which will help to find less incremental data in incremental backup system.

## III. MINIMAL INCREMENTAL INTERVAL

We propose a novel content-defined chunking algorithm Minimal Incremental Interval(MII), which is applied in incremental synchronization between files.

### A. ALGORITHM PROCESS

In MII algorithm, the file, named *fileSrc*, to be chunked is read as a byte stream. During the reading process, the byte value named *valueCur* of the current read byte is compared with the byte value named *valuePre* of the previous read byte. If *valueCur* is bigger than *valuePre*, the tuple $< valueCur, valuePre >$ is recorded as satisfying relation $R$. If the tuple $< value2, value1 >$ consisting of any two adjacent byte values $value1$ (the previous byte value) and

*Value*2 (the latter byte value) in a data interval consisting of byte values continuously read from *fileSrc* satisfies the relation *R*, the data interval is marked as an incremental interval named *L*, and the number of byte values in the data interval is marked as the length of the incremental interval named *len*. If the length *len* of an incremental interval reaches the preset threshold, a cut point is formed here. Read the whole file as a byte stream and find all the cut points that satisfy the preset condition. The chunking process of the algorithm is given in Fig. 1. The pseudo code of the MII algorithm is given in Fig. 2.

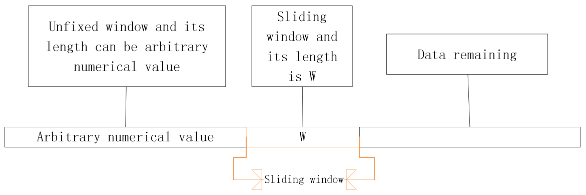

Unfixed window and its length can be arbitrary numerical value
Sliding window and its length is W
Data remaining
Arbitrary numerical value
W
Sliding window

**FIGURE 1.** **The algorithm process of MII.**

```
Algorithm 1: Algorithm for MII chunking
Input: input file,file; length of incremental interval,W;
Output: cut point,I;
function MIIChunking(file,w)
    i=1
    increment=0
    while(byte=readByte(file))
        if byte>previous.value then
            increment=increment+1
            if increment==W then
                return i
            end if
        else
            increment=0
        end if
        previous.value=byte
        i=i+1
    end while
end function
```

**FIGURE 2.** **The pseudo code of the MII algorithm.**

## B. HOW TO PRESET THE THRESHOLD

The only threshold that needs to be preset is the length of the incremental interval.

When a file is read as a byte stream, it is random whether the binary group $< value1, value2 >$ consisting of the byte values of two adjacent bytes satisfies the relation *R* mentioned in the previous section. If the threshold is set to *valueLen*, the probability that the length *len* of incremental interval is greater than or equal to *valueLen* is as follow.

$$p = C_{256}^{valueLen} / A_{256}^{valueLen}$$

In order to find an incremental interval that satisfies the condition, the number of bytes that need to be read continuously obeys the geometric distribution $G(p)$ approximately.

The difference of the threshold will affect the size of data chunks, so the threshold can be determined according to the actual demand for the size of chunks. When the threshold is big, the average size of the obtained chunks will also be big. When the threshold is small, the average size of the resulting chunks will be small as well. After a lot of experiments, for most documents, the threshold value can be set as 5, which can make the size of data chunks in an acceptable range and the average size of data chunks is about 700.

## C. CHUNK SIZES VARIANCE

As described above, the length of data chunks obeys the geometric distribution $G(p)$ approximately, which is to say that the variance of the size of chunks is kind of large and the number of small chunks is obviously much bigger than the number of large ones. Although the variance of chunk length will directly affect the efficiency of storage when the chunking algorithm is used in data deduplication, it will not reduce the applicability of the chunking algorithm when the chunking algorithm is only used to find incremental data and the chunks obtained are not stored directly. At the same time, the algorithm can avoid the generation of overly long chunks by adding an extra condition on the cut point conditions.

The comparison of the size variance of chunks between MII algorithm and other algorithms will be listed in the following experimental section.

## D. RESISTANCE TO BYTE SHIFTING

Suppose there is a segment of data in the byte stream as shown in the Fig. 3, where intervals $L1$, $L2$ and $L3$ are incremental intervals satisfying the preset conditions of the MII algorithm. When a byte shifting occurs outside the incremental interval satisfying the preset conditions, presuming it occurs in the interval $L4$, then interval $L1$ and interval $L2$ still satisfy the preset conditions of the algorithm. If the byte shifting causes a new interval $L6$ in $L4$, which satisfies the preset conditions of the algorithm, as shown in the Fig. 4, number of chunks will changes from one to two between $L1$ and $L2$, and the chunks before $L1$ and after $L2$ will not be affected. If the byte shifting does not produce a new interval that meets the preset conditions, then $L4$ and $L2$ can still form a chunk, and the rest of the chunks will not be affected. That is, when a byte shifting occurs outside a incremental interval, which satisfies the preset conditions, it will only affect the chunk where the byte shifting is located, and other chunks will not be affected. When a byte shifting occurs inside a incremental interval,

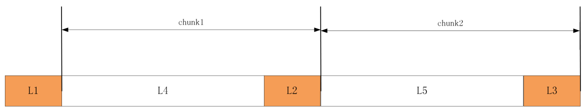


**FIGURE 3.** **Step1 for the resistance to byte shifting of MII.**

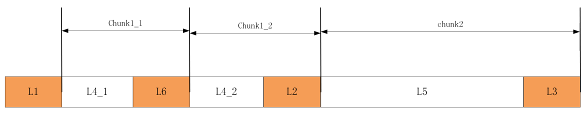


**FIGURE 4.** Step2 for the resistance to byte shifting of MII.

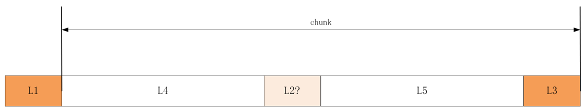


**FIGURE 5.** Step3 for the resistance to byte shifting of MII.

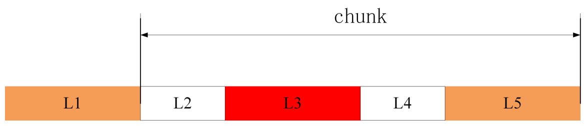


**FIGURE 6.** Situation of low entropy string.

which satisfies the preset conditions, let us assume that the byte shifting occurs in $L2$, as shown in the Fig. 5. Regardless of whether the changed $L2$ meets the preset conditions of the algorithm or not because of the byte shifting, $L1$ and $L3$ do not change and they are still the boundaries of chunks, which means the chunks before $L1$ and after $L3$ are not affected. That is, when a byte shifting occurs in a incremental interval, which satisfies the preset conditions, it will only affect the chunk where the byte shifting is located and the next chunk adjacent to it, and other chunks will not be affected.

### *E. ABILITY TO ELIMINATE LOW ENTROPY STRINGS*

In MII algorithm, a chunk is found by finding the incremental interval whose length reaches the preset value, so even if there is a low entropy string, there will be numerical differences in it. As long as there are numerical differences, there will be an incremental interval, which can be used to reach the conditions of MII algorithm and implement MII algorithm as normal. In a more extreme case, it is consumed that there is a string with equal byte values in the low entropy string as shown in the Fig. 6. $L1$ and $L5$ are the chunks satisfying the preset conditions of the MII algorithm. $L4$ is the string with equal byte values and $L2$, $L4$ are normal intervals. In this case, the chunk will never be changed unless one or more intervals of L1, L2, L3, L4 and L5 are updated. According to the steady of the chunk, L3 will always be eliminated with the total chunk.

## IV. TIME AND SPACE COMPLEXITY

In this section, we discuss the time and space complexity of the MII algorithm and other algorithms mentioned in our paper.

### *A. PSEUDO CODE AND CHUNKING PROCESS OF ALGORITHM*

Since the pseudo code and chunking process of MII have been illustrated in section III, this section shows the pseudo codes and chunking process diagrams of other algorithms. The approximate processes of these algorithms can be found in section II.

```
Algorithm 5: Algorithm for Rabin chunking
Input: input file,file; default value,Value;length of sliding window,W;
Output: cut point,I;
function RabinChunking(file,Value,W)
    i=1
    index=0
    while(byte=readByte(file))
        array[index%W+1]=byte
        if array.length>=W then
            if hashValue(array,index,W)==Value then
                return i
            end if
        else
            continue
        end if
        i=i+1
    end while
end function
```

**FIGURE 7.** The pseudo code of the Rabin algorithm.

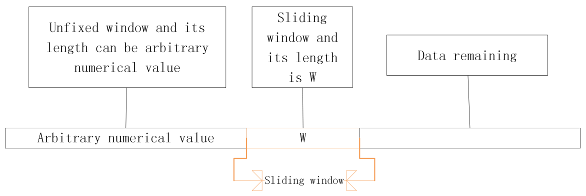

Unfixed window and its length can be arbitrary numerical value
Sliding window and its length is W
Data remaining
Arbitrary numerical value
W
Sliding window

**FIGURE 8.** The algorithm process of Rabin.

```
Algorithm 4: Algorithm for LMC chunking
Input: input file,file; size of fixed window,W;
Output: cut point,I;
function LMCChunking(file,W)
    i=1
    start=1
    while(byte=readByte(file))
        if byte<=max.value then
            if i==max.position+w and max.position>=start+w then
                start=max.position+1
                return max.position
            end if
        else
            max.value=byte
            max.position=i
        end if
        i=i+1
    end while
end function
```

**FIGURE 9.** The pseudo code of the LMC algorithm.

The pseudo code and chunking process of Rabin algorithm are shown in the Fig. 7 and the Fig. 8.

The pseudo code and chunking process of LMC algorithm are shown in the Fig. 9 and the Fig. 10.

The pseudo code and chunking process of AE algorithm are shown in the Fig. 11 and the Fig. 12.

The pseudo code and chunking process of RAM algorithm are shown in the Fig. 13 and the Fig. 14.

### *B. TIME COMPLEXITY*

In Rabin algorithm, there is only one loop in the function as shown in the Fig. 7, so the time complexity of Rabin

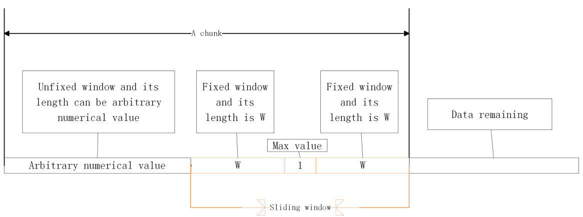

A chunk
Unfixed window and its length can be arbitrary numerical value
Fixed window and its length is W
Fixed window and its length is W
Data remaining
Max value
Arbitrary numerical value
W
1
W
Sliding window

**FIGURE 10.** The algorithm process of LMC.

```
Algorithm 3: Algorithm for AE chunking
Input: input file,file; size of fixed window,W;
Output: cut point,I;
function AEChunking(file,W)
    i=1
    while(byte=readByte(file))
        if byte<=max.value then
            if i==max.position+w then
                return i
            end if
        else
            max.value=byte
            max.position=i
        end if
        i=i+1
    end while
end function
```

**FIGURE 11.** The pseudo code of the AE algorithm.

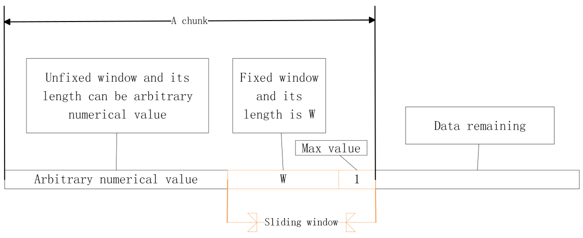

A chunk
Unfixed window and its length can be arbitrary numerical value
Fixed window and its length is W
Data remaining
Max value
Arbitrary numerical value
W
1
Sliding window

**FIGURE 12.** The algorithm process of AE.

algorithm is $O(n)$. Since the operations include two additions, one modular reduction, two assignments, one comparison, and one hash in the loop, the time of each loop can be defined as $1ADD + 2M + 2ASS + 1C + 1H$.

In LMC algorithm, there is also only one loop in the function as we can see in the Fig. 9, so the time complexity of LMC algorithm is $O(n)$. Since the operations include four additions, two assignments and three comparisons in the loop at most, the time of each loop can be defined as $4ADD + 2ASS + 3C$.

In AE algorithm, there is only one loop as well in the function as we can see in the Fig. 11, so the time complexity of AE algorithm is $O(n)$. Since the operations include two additions, one assignment and two comparisons in the loop at most, the time of each loop can be defined as $2ADD + 1ASS + 2C$.

```
Algorithm 2: Algorithm for RAM chunking
Input: input file,file; size of fixed window,W;
Output: cut point,I;
function RAMChunking(file,W)
    i=1
    while(byte=readByte(file))
        if byte>=max.value then
            if i>w then
                return i
            end if
            max.value=byte
            max.position=i
        end if
        i=i+1
    end while
end function
```

**FIGURE 13.** The pseudo code of the RAM algorithm.

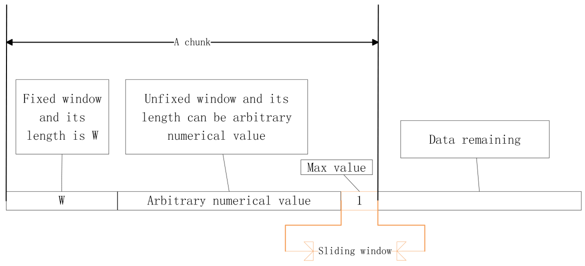

A chunk
Fixed window and its length is W
Unfixed window and its length can be arbitrary numerical value
Data remaining
Max value
W
Arbitrary numerical value
1
Sliding window

**FIGURE 14.** The algorithm process of RAM.

**TABLE 1.** Time complexity contrast between algorithms.

| Algorithm | Time complexity | each time(operation) |
|---|---|---|
| Rabin | $O(n)$ | $1ADD + 2M + 2ASS + 1C + 1H$ |
| LMC | $O(n)$ | $4ADD + 2ASS + 3C$ |
| AE | $O(n)$ | $2ADD + 1ASS + 2C$ |
| RAM | $O(n)$ | $1ADD + 3ASS + 2C$ |
| MII | $O(n)$ | $2ADD + 3ASS + 2C$ |

In RAM algorithm, there is only one loop as well in the function as we can see in the Fig. 13, so the time complexity of RAM algorithm is $O(n)$. Since the operations include one addition, three assignments and two comparisons in the loop at most, the time of each loop can be defined as $1ADD + 3ASS + 2C$.

In MII algorithm, there is only one loop as well in the function as we can see in the Fig. 2, so the time complexity of MII algorithm is $O(n)$. Since the operations include two additions, three assignments and two comparisons in the loop at most, the time of each loop can be defined as $2ADD + 3ASS + 2C$.

The Contrast between time complexities of algorithms is shown in Table 1.

### C. SPACE COMPLEXITY

In Rabin algorithm shown in the Fig. 7, when Rabin fingerprint matching in the sliding window fails, we need to slide one byte forward to calculate the Rabin fingerprint of the next

**TABLE 2.** Space complexity contrast between algorithms.

| Algorithm | Time complexity | exact space used |
|---|---|---|
| Rabin | $O(1)$ | $Array[W]$,$W$ is the length of the sliding window |
| LMC | $O(1)$ | $Array[2W+1]$,$W$ is the length of the fixed window |
| AE | $O(1)$ | one integer variable |
| RAM | $O(1)$ | one integer variable |
| MII | $O(1)$ | two integer variables |

sliding window, which means it is necessary to record all the byte values of the sliding window. Therefore, the space complexity of the algorithm is $O(1)$, specifically an array of length $W$ ($W$ is the length of the sliding window).

In LMC algorithm shown in the Fig. 9, to reduce the time overhead, it is necessary to store all byte values in the whole sliding window with a circular queue because the maximum value is located in the middle of two fixed windows and make a calculation of hash every time the latest data is read. If the maximum value is in the middle of the circular queue, the cut point is found. Therefore, the space complexity of the algorithm is $O(1)$, specifically an array of length $2W + 1$ ($W$ is the length of the fixed window).

In AE algorithm shown in the Fig. 11, an integer variable is required to store the maximum byte value in the data reading process when searching for the cut point, and the value in the sliding window does not need to be recorded during the process because only the byte value of the latest read byte is compared with the saved maximum byte value. Therefore, the space complexity of the algorithm is $O(1)$, specifically one integer variable.

In RAM algorithm, the space complexity is the same as in AE algorithm, which is $O(1)$,specifically one integer variable, because only the maximum value is needed to be stored as we can tell from the Fig. 13.

In MII algorithm shown in the Fig. 2, an integer variable is needed to store the current byte value in the data reading process for comparison with the next byte value read. Whereas, another integer variable is needed to store the length of the current incremental interval. Therefore, the space complexity of the algorithm is $O(1)$, specifically two integer variables.

The Contrast between space complexities of algorithms is shown in Table 2.

## V. EXPERIMENTS

In this section, we present the experimental evaluations of our MII algorithm in terms of multiple performance metrics. To characterize the benefits of MII, we also compare it with four existing CDC algorithms including Rabin, LMC, AE and RAM.

### A. EXPERIMENTAL ENVIRONMENT CONSTRUCTION

In this section, the experimental environment construction is given in table 3.

**TABLE 3.** experimental environment construction.

| item | detail |
|---|---|
| CPU | Inter(R) Core(TM) i5-7500, 3.40GHz |
| MEMORY | DDR4, 16GB |
| OS | Windows 7, x64 |
| Programming language | Java, jdk1.8 |
| disk | mechanical hard disk, 1000GB |

```
Algorithm 6: Algorithm for Generating Files
Input: number of files,n;
Output: File,file;
function GenerateFile(n)
    for i 0 to n by 1 do
        file=openFile(i)
        for j 0 to 2000000000 by 1 do
            byte=Random(0,255)
            file.write(byte)
        end for
    end for
end function
```

**FIGURE 15.** The pseudo code of algorithm to generate the datasets.

### B. DATASETS

The experimental datasets include 9 files, each of which is $2 * 10^9$ bytes in size. All the bytes of files with byte values in the interval [0,255] are generated by the Mersenne Twister Pseudo-Random Number Generator [52], and then bytes with a number of $2 * 10^9$ are added to the file one by one to form an experimental dataset. Nine files are generated in the same way, which are the experimental datasets of this paper. The pseudo code for algorithm generating experimental datasets is given in Fig. 15.

Besides, there are six more datasets, which are generated in the same way, only used to analysis incremental data discovery rate. The numbers of byte values in three of them are $1 * 10^9$ and the other three are $1.5 * 10^9$.

The reasons why the experimental datasets are randomly generated instead of the real data are shown as follow:

1) There are so many types of real data, it is not representative to select only several of them and almost impossible to choose all of them.
2) Random data can reflect the randomness and applicability of the experimental datasets.
3) Since the algorithm to generate random data is given above, it is easy to reappear the experiments.

The total number of datasets is 15. The reason why it is 15 is that several files are enough to test the performance of chunking algorithms considering of the randomness of the datasets. Perhaps the more files, the better, but 15 files are fair enough.

All the datasets are shown in Table 4.

**TABLE 4. the datasets used in this paper.**

| name | size(bytes) | generating algorithm |
|---|---|---|
| data1 | $2 * 10^9$ | Mersenne Twister Pseudo-Random Number Generator |
| data2 | $2 * 10^9$ | Mersenne Twister Pseudo-Random Number Generator |
| data3 | $2 * 10^9$ | Mersenne Twister Pseudo-Random Number Generator |
| data4 | $2 * 10^9$ | Mersenne Twister Pseudo-Random Number Generator |
| data5 | $2 * 10^9$ | Mersenne Twister Pseudo-Random Number Generator |
| data6 | $2 * 10^9$ | Mersenne Twister Pseudo-Random Number Generator |
| data7 | $2 * 10^9$ | Mersenne Twister Pseudo-Random Number Generator |
| data8 | $2 * 10^9$ | Mersenne Twister Pseudo-Random Number Generator |
| data9 | $2 * 10^9$ | Mersenne Twister Pseudo-Random Number Generator |
| data10 | $1 * 10^9$ | Mersenne Twister Pseudo-Random Number Generator |
| data11 | $1 * 10^9$ | Mersenne Twister Pseudo-Random Number Generator |
| data12 | $1 * 10^9$ | Mersenne Twister Pseudo-Random Number Generator |
| data13 | $1.5 * 10^9$ | Mersenne Twister Pseudo-Random Number Generator |
| data14 | $1.5 * 10^9$ | Mersenne Twister Pseudo-Random Number Generator |
| data15 | $1.5 * 10^9$ | Mersenne Twister Pseudo-Random Number Generator |

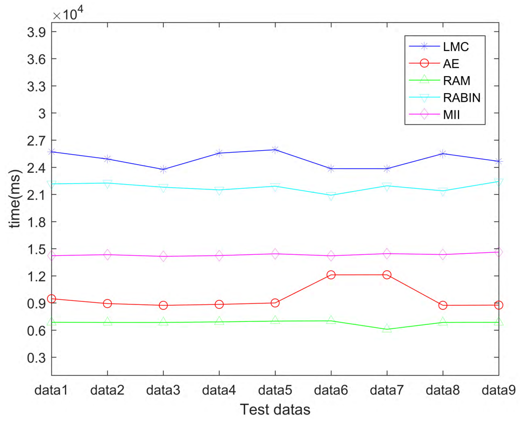


**FIGURE 16. The running time of chunking algorithms in different datasets.**

## C. PARAMETER SETTING

The preset condition of MII algorithm is that the incremental interval length is 5, the fixed window length of RAM, AE and LMC is 700, and the sliding window size of Rabin is 7. The principle of parameter selection is that the total number of chunks in the results of each algorithm is approximately equal, which means the bandwidth cost is almost the same when the chunks converted into fingerprints traverse the network.

## D. CHUNK THROUGHPUT

In this section, the running time of each chunking algorithm is obtained by implementing every chunking algorithm on the same experimental dataset, which is used to compare the chunk throughput among these chunking algorithms. The running time of the chunking algorithms corresponding to each experimental dataset is shown in the Fig. 16.

The experimental results show that RAM algorithm is the fastest, the chunking time of MII and AE algorithm are almost at the same level at the second place, and the chunking time of Rabin and LMC algorithm are relatively long. That is to say, MII chunking algorithm is only behind the RAM algorithm in speed and keeps the same level with AE algorithm. The chunk throughput can be calculated with a formula *number of chunks/chunking time*. When the number of chunks are approximately the same as mentioned in last section, the less chunking time cost, the better chunk throughput we got. So the chunk throughput of MII is bigger than the average value.

## E. CHUNK SIZE DISTRIBUTION

In this section, the chunk size distribution is obtained after each chunking algorithm is implemented on the same experimental dataset, which is used to compare the chunk size stability of these chunking algorithms. The chunk size

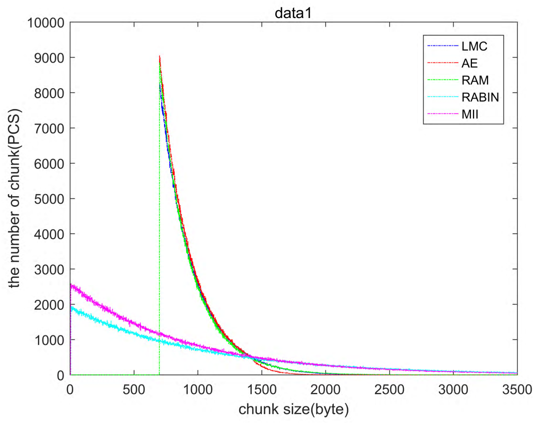


**FIGURE 17. The chunk size distributions of chunking algorithms in data1.**

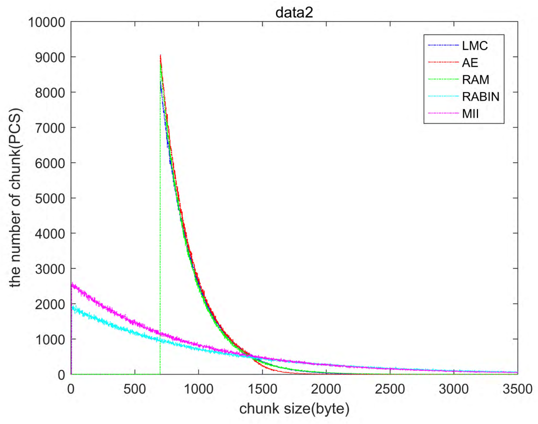


**FIGURE 18. The chunk size distributions of chunking algorithms in data2.**

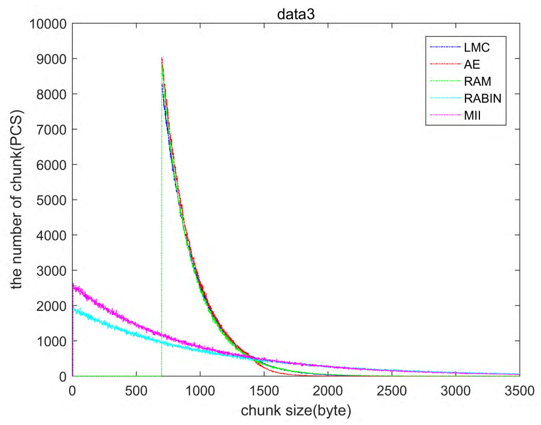


**FIGURE 19.** The chunk size distributions of chunking algorithms in data3.

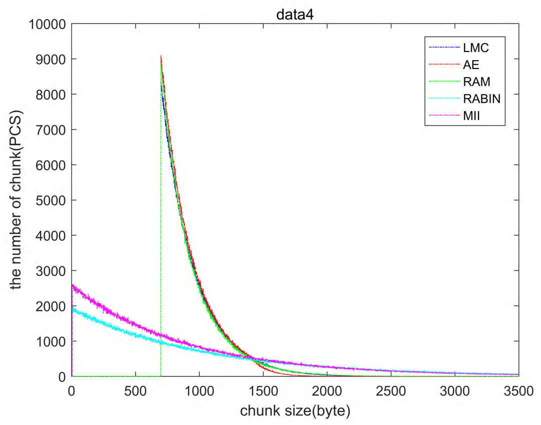


**FIGURE 20.** The chunk size distributions of chunking algorithms in data4.

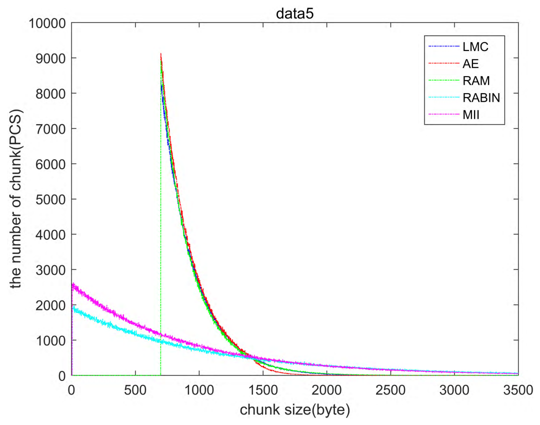


**FIGURE 21.** The chunk size distributions of chunking algorithms in data5.

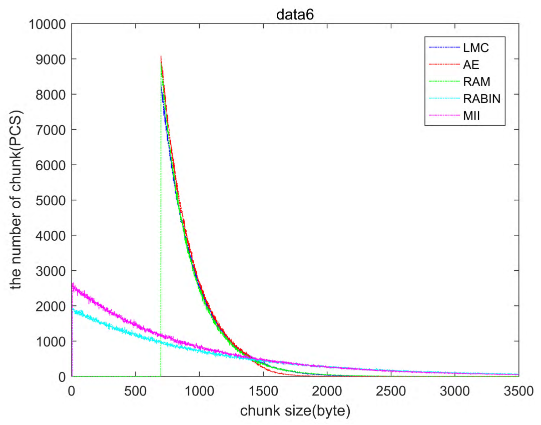


**FIGURE 22.** The chunk size distributions of chunking algorithms in data6.

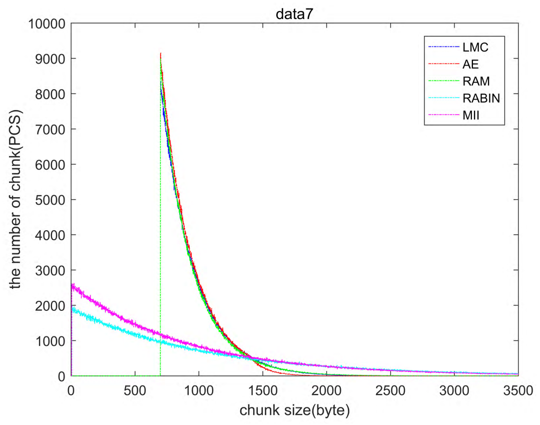


**FIGURE 23.** The chunk size distributions of chunking algorithms in data7.

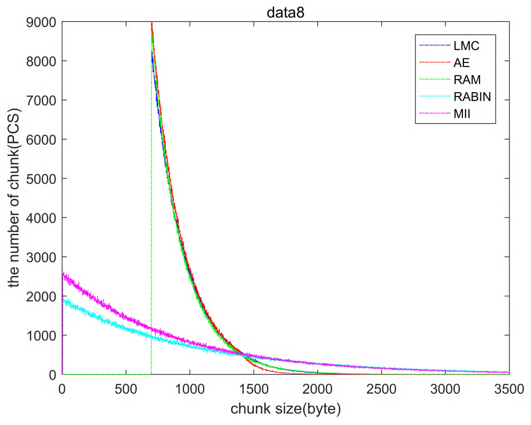


**FIGURE 24.** The chunk size distributions of chunking algorithms in data8.

distributions of the chunking algorithms corresponding to each dataset are shown in the following figures.

The results on data1 are shown in the Fig. 17. The results on data2 are shown in the Fig. 18. The results on data3 are shown in the Fig. 19. The results on data4 are shown in the Fig. 20. The results on data5 are shown in the Fig. 21. The results on data6 are shown in the Fig. 22. The results on data7 are shown in the Fig. 23. The results on data8 are shown in the Fig. 24. The results on data9 are shown in the Fig. 25.

The experimental results show that the chunk size distributions of LMC, RAM and AE algorithms are relatively stable, because these three algorithms set the fixed interval, so the size of chunk cannot be smaller than the length of the fixed interval. The chunk size stabilities of Rabin and MII

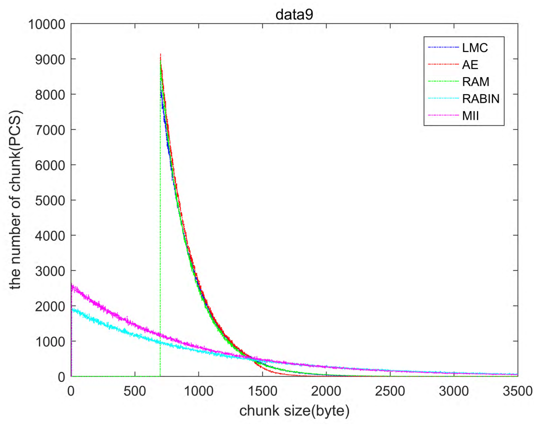


**FIGURE 25.** The chunk size distributions of chunking algorithms in data9.

```
Algorithm 7: Algorithm for Inserting
Input: Input file,fileIn;
Output: Output file,fileOut;
function Insert(fileIn)
    i=0
    while(byte=readByte(fileIn))
        i=i+1
        fileOut.write(byte)
        if i%10000==0 then
            for j 0 to 100 by 1 do
                fileOut.write(Random(0,255))
            end for
        end if
    end while
end function
```

**FIGURE 26.** The algorithm for random bytes insertion.

algorithms are poor, because the probability of finding a cut point in a random string by Rabin and MII algorithms can be roughly calculated, which makes finding the cut point similar to implementing a Bernoulli experiment and the chunk size distributions obtained similar to the geometric distribution.

## F. INCREMENTAL DATA DISCOVERY RATE

In this section, the experimental data files are modified. For an original experimental data file named *file*, three new files named *file_insert*, *file_delete* and *file_add* are generated by random bytes insertion, random bytes deletion and bytes addition. By doing the same operation on the nine original experimental data files, we get 45 new experimental data files.

The algorithm for random bytes insertion is shown in the Fig. 26. The algorithm for random bytes deletion is shown in the Fig. 27. The algorithm for random bytes addition is shown in the Fig. 28.

The experimental procedure of implementing each chunking algorithm to find the incremental data on the datasets is shown as follow:

```
Algorithm 8: Algorithm for Deleting
Input: Input file,fileIn;
Output: Output file,fileOut;
function Delete(fileIn)
    i=0
    j=0
    while(byte=readByte(fileIn))
        i=i+1
        if j==0 then
            fileOut.write(byte)
        else
            j=j-1
        end if
        if i%10000==0 then
            j=100
        end if
    end while
end function
```

**FIGURE 27.** The algorithm for random bytes deletion.

```
Algorithm 9: Algorithm for Adding
Input: Input file,fileIn;
Output: Output file,fileOut;
function Add(fileIn)
    i=0
    j=0
    while(byte=readByte(fileIn))
        fileOut.write(byte)
    end while
    for j 0 to 20000 by 1 do
        fileOut.write(Random(0,255))
    end for
end function
```

**FIGURE 28.** The algorithm for random bytes addition.

### 1) STEP 1

Implement the chunking algorithm on the original dataset *file* and save all the chunks obtained.

### 2) STEP 2

Implement the chunking algorithm on the modified file *file_insert* and compare each chunk *chunk* in the results with the chunks saved in Step 1. If a same chunk cannot be found, the chunk *chunk* is an incremental chunk. Collect all the incremental chunks to get the incremental data *increment_insert* in random insertion case.

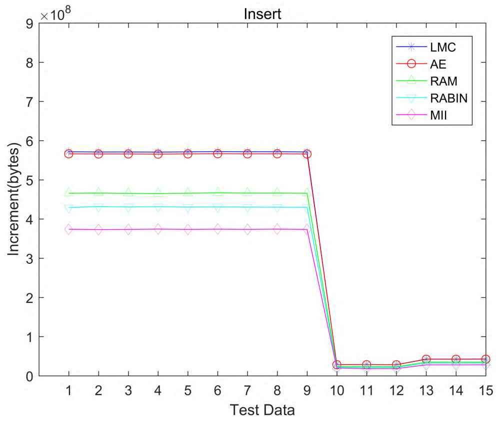


**FIGURE 29.** **The incremental data discovery of the chunking algorithms in random bytes insertion case.**

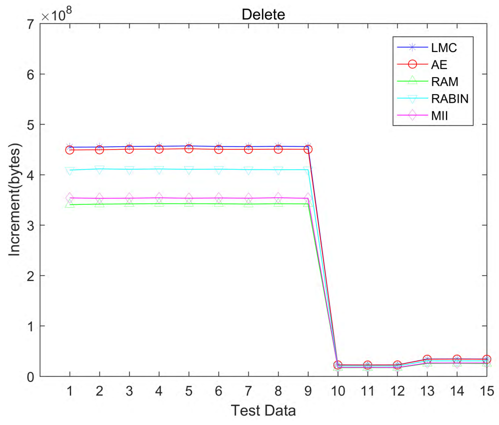


**FIGURE 30.** **The incremental data discovery of the chunking algorithms in random bytes deletion case.**

### 3) STEP 3

Implement the chunking algorithm on the modified file *file_delete* and compare each chunk *chunk* in the results with the chunks saved in Step 1. If a same chunk cannot be found, the chunk *chunk* is an incremental chunk. Collect all the incremental chunks to get the incremental data *increment_delete* in random deletion case.

### 4) STEP 4

Implement the chunking algorithm on the modified file *file_add* and compare each chunk *chunk* in the results with the chunks saved in Step 1. If a same chunk cannot be found, the chunk *chunk* is an incremental chunk. Collect all the incremental chunks to get the incremental data *increment_add* in random deletion case.

### 5) STEP 5

Change the original dataset *file* to another one and repeat the procedure from Step1 to Step 4 until all the datasets are used.

Perform the same experimental procedure on every chunking algorithm and get the *increment_insert*s, *increment_deletes* and *increment_add*s of the chunking algorithms implemented on all the datasets.

The experimental results are as follows. In random bytes insertion case, the incremental data discovered by the chunking algorithms on each dataset is shown in Fig. 29. In random bytes deletion case, the incremental data discovered by the chunking algorithms on each dataset are shown in Fig. 30. In random bytes addition case, the incremental data discovered by the chunking algorithms on each dataset is shown in Fig. 31.

The experimental results show that the incremental data found by each algorithm are almost the same when searching for the incremental data of the modified files formed by random bytes addition at the end of the files. MII algorithm and RAM algorithm have obvious advantages over other algorithms in searching for the incremental data of modified files formed by random bytes deletion. In searching for the

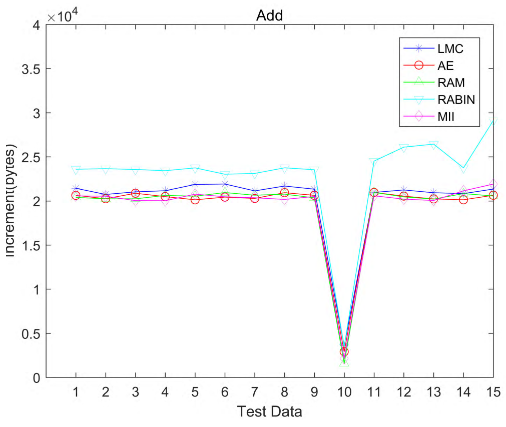


**FIGURE 31.** **The incremental data discovery of the chunking algorithms in random bytes addition case.**

incremental data of modified files formed by random insertion, which is the most frequent operation in reality, MII algorithm has greater advantages than other algorithms and reduces the incremental data by 13%∼34%, because MII algorithm has more superior performance in resistance to byte shifting.

## VI. CONCLUSION

In this study, the application fields of data chunking algorithms, the algorithm processes of the classical or state of art chunking algorithms and the shortcomings of these algorithms in finding incremental data are discussed. We propose a novel data chunking algorithm called MII, which is specifically implemented to find incremental data in data synchronization system. We compare and analyze the time and space complexity of the mentioned algorithms. We discuss the chunk throughput, chunk size variance and incremental data discovery rate of the mentioned algorithms. The experimental results show that the chunk throughput of the MII is proved at the second place among all the chunking algorithms, it has obvious advantage in the incremental data discovery rate although the performance on chunk size variance is a little bad.

The main advantage of the MII algorithm is that a strong resistance against the byte shifting problem is obtained by sacrificing some stability of chunk size, which allows to locate incremental data more accurately in data synchronization system.

## ACKNOWLEDGMENT

*(Changjian Zhang and Deyu Qi contributed equally to this work.)*

## REFERENCES

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

[2] S. Luo and M. Hou, "A novel chunk coalescing algorithm for data deduplication in cloud storage," in *Proc. IEEE Jordan Conf. Appl. Elect. Eng. Comput. Technol. (AEECT)*, Dec. 2013, pp. 1–5.

[3] W. Xia, D. Feng, H. Jiang, Y. Zhang, V. Chang, and X. Zou, "Accelerating content-defined-chunking based data deduplication by exploiting parallelism," *Future Gener. Comput. Syst.*, vol. 98, pp. 406–418, Sep. 2019. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167739X18320053

[4] M. Yoon, "A constant-time chunking algorithm for packet-level deduplication," *ICT Express*, vol. 5, no. 2, pp. 131–135, Jun. 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S2405959518302157

[5] W. Tian, R. Li, Z. Xu, and W. Xiao, "Does the content defined chunking really solve the local boundary shift problem?" in *Proc. IEEE 36th Int. Perform. Comput. Commun. Conf. (IPCCC)*, Dec. 2017, pp. 1–8.

[6] Q. N. Nguyen, M. Arifuzzaman, K. Yu, and T. Sato, "A context-aware green information-centric networking model for future wireless communications," *IEEE Access*, vol. 6, pp. 22804–22816, 2018.

[7] S. Sanyal and P. Zhang, "Improving quality of data: IoT data aggregation using device to device communications," *IEEE Access*, vol. 6, pp. 67830–67840, 2018.

[8] J. Li, J. Wu, and L. Chen, "Block-secure: Blockchain based scheme for secure P2P cloud storage," *Inf. Sci.*, vol. 465, pp. 219–231, Oct. 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0020025518305012

[9] Z. Liu, N. Murray, B. Lee, E. Fallon, and Y. Qiao, "MVP2P: Layer-dependency-aware live MVC video streaming over peer-to-peer networks," *Signal Process. Image Commun.*, vol. 65, pp. 173–186, Jul. 2018.

[10] D. Guo, Y. K. Kwok, and J. Xin, "Valuation of information and the associated overpayment problem in peer-to-peer systems," *Comput. Commun.*, vol. 80, pp. 59–71, Apr. 2016.

[11] T.-Y. Youn, K.-Y. Chang, K.-H. Rhee, and S. U. Shin, "Efficient client-side deduplication of encrypted data with public auditing in cloud storage," *IEEE Access*, vol. 6, pp. 26578–26587, 2018.

[12] Y. Zhou, Y. Deng, L. T. Yang, R. Yang, and L. Si, "LDFS: A low latency in-line data deduplication file system," *IEEE Access*, vol. 6, pp. 15743–15753, 2018.

[13] Y. Zhou, D. Feng, Y. Hua, W. Xia, M. Fu, F. Huang, and Y. Zhang, "A similarity-aware encrypted deduplication scheme with flexible access control in the cloud," *Future Gener. Comput. Syst.*, vol. 84, pp. 177–189, Jul. 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167739X17309238

[14] A. Bhalerao and A. Pawar, "A survey: On data deduplication for efficiently utilizing cloud storage for big data backups," in *Proc. Int. Conf. Trends Electron. Informat. (ICEI)*, May 2017, pp. 933–938.

[15] D. Rasch and R. Burns, "In-place rsync: File synchronization for mobile and wireless devices," in *Proc. Annu. Conf. USENIX Annu. Tech. Conf. (ATEC)*. Berkeley, CA, USA: USENIX Association, 2003, p. 15. [Online]. Available: http://dl.acm.org/citation.cfm?id=1247340.1247355

[16] M. Constantinou, "Tuning of rsync algorithm for optimum cloud storage performance," Dept. Comput. Sci., Univ. Bath, Bath, U.K., Tech. Rep. CSBU-2013-10, Dec. 2013.

[17] J. Ma, C. Bi, Y. Bai, and L. Zhang, "UCDC: Unlimited content-defined chunking, a file-differing method apply to file-synchronization among multiple hosts," in *Proc. 12th Int. Conf. Semantics, Knowl. Grids (SKG)*, Aug. 2016, pp. 76–82.

[18] K. Thar, N. H. Tran, S. Ullah, T. Z. Oo, and C. S. Hong, "Online caching and cooperative forwarding in information centric networking," *IEEE Access*, vol. 6, pp. 59679–59694, 2018.

[19] H. Noh and H. Song, "Progressive caching system for video streaming services over content centric network," *IEEE Access*, vol. 7, pp. 47079–47089, 2019.

[20] X. Zhang, N. Wang, V. G. Vassilakis, and M. P. Howarth, "A distributed in-network caching scheme for P2P-like content chunk delivery," *Comput. Netw.*, vol. 91, pp. 577–592, Nov. 2015. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1389128615002959

[21] M. Zhu, L. Dan, F. Wang, A. Li, K. K. Ramakrishnan, L. Ying, J. Wu, Z. Nan, and L. Xue, "CCDN: Content-centric data center networks," *IEEE/ACM Trans. Netw.*, vol. 24, no. 6, pp. 3537–3550, Dec. 2016.

[22] C. Li, J. Tang, H. Tang, and Y. Luo, "Collaborative cache allocation and task scheduling for data-intensive applications in edge computing environment," *Future Gener. Comput. Syst.*, vol. 95, pp. 249–264, Jun. 2019. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167739X18309439

[23] X. Ren, Y. Zhou, Z. Huang, J. Sun, X. Yang, and K. Chen, "A novel text structure feature extractor for Chinese scene text detection and recognition," *IEEE Access*, vol. 5, pp. 3193–3204, 2017.

[24] W. Lu, H. Sun, J. Chu, X. Huang, and J. Yu, "A novel approach for video text detection and recognition based on a corner response feature map and transferred deep convolutional neural network," *IEEE Access*, vol. 6, pp. 40198–40211, 2018.

[25] M. Huang and R. M. Haralick, "A method for discovering knowledge in texts," *Pattern Recognit. Lett.*, vol. 124, pp. 21–30, Jun. 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167865518300801

[26] Y.-P. Ruan, Q. Chen, and Z. Ling, "A sequential neural encoder with latent structured description for modeling sentences," *IEEE/ACM Trans. Audio, Speech, Language Process.*, vol. 26, no. 2, pp. 231–242, Feb. 2018.

[27] Q. Fu, H. Sun, Z. Dienes, and X. Fu, "Implicit sequence learning of chunking and abstract structures," *Consciousness Cognition*, vol. 62, pp. 42–56, 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1053810017303951

[28] F. Xhafa, "Data replication and synchronization in P2P collaborative systems," in *Proc. IEEE 26th Int. Conf. Adv. Inf. Netw. Appl.*, Mar. 2012, p. 7.

[29] E. Bertino, G. Guerrini, and I. Merlo, "Trigger inheritance and overriding in an active object database system," *IEEE Trans. Knowl. Data Eng.*, vol. 12, no. 4, pp. 588–608, Jul./Aug. 2000.

[30] J. Lee, K. Kim, and S. K. Cha, "Differential logging: A commutative and associative logging scheme for highly parallel main memory database," in *Proc. 17th Int. Conf. Data Eng.*, Apr. 2001, pp. 173–182.

[31] J. Zhang, "A data synchronization method oriented to custom hierarchical multi-node system," in *Proc. IEEE Int. Conf. Comput. Intell. Commun. Technol.*, Feb. 2015, pp. 666–669.

[32] M. Akter, A. Gani, M. O. Rahman, M. M. Hassan, A. Almogren, and S. Ahmad, "Performance analysis of personal cloud storage services for mobile multimedia health record management," *IEEE Access*, vol. 6, pp. 52625–52638, 2018.

[33] J. Liu, J. Wang, X. Tao, and J. Shen, "Secure similarity-based cloud data deduplication in ubiquitous city," *Pervas. Mobile Comput.*, vol. 41, pp. 231–242, Oct. 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1574119217301657

[34] P. K. Krishnaprasad and B. A. Narayamparambil, "A proposal for improving data deduplication with dual side fixed size chunking algorithm," in *Proc. 3rd Int. Conf. Adv. Comput. Commun.*, Aug. 2013, pp. 13–16.

[35] M. O. Rabin, "Fingerprinting by random polynomials," Tech. Rep. TR-15-81, 1981, pp. 15–18.

[36] R. Raju, M. Moh, and T.-S. Moh, "Compression of wearable body sensor network data using improved two-threshold-two-divisor data chunking algorithms," in *Proc. Int. Conf. High Perform. Comput. Simulation (HPCS)*, Jul. 2018, pp. 949–956.

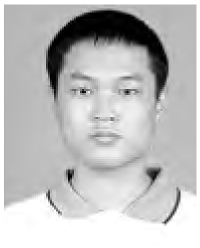
[37] Y. Won, K. Lim, and J. Min, "MUCH: Multithreaded content-based file chunking," *IEEE Trans. Comput.*, vol. 64, no. 5, pp. 1375–1388, May 2015.

[38] Z. Tang and Y. Won, "Multithread content based file chunking system in CPU-GPGPU heterogeneous architecture," in *Proc. 1st Int. Conf. Data Compress., Commun. Process.*, Jun. 2011, pp. 58–64.
[39] F. Ni, X. Lin, and S. Jiang, "SS-CDC: A two-stage parallel content-defined chunking for deduplicating backup storage," in *Proc. 12th ACM Int. Conf. Syst. Storage (SYSTOR)*, New York, NY, USA, 2019, pp. 86–96. doi: 10.1145/3319647.3325834.
[40] B. Romañski, L. Heldt, W. Kilian, K. Lichota, and C. Dubnicki, "Anchor-driven subchunk deduplication," in *Proc. 4th Annu. Int. Conf. Syst. Storage (SYSTOR)*, New York, NY, USA, 2011, pp. 16-1–16-13. doi: 10.1145/1987816.1987837.
[41] N. Bjørner, A. Blass, and Y. Gurevich, "Content-dependent chunking for differential compression, the local maximum approach," *J. Comput. Syst. Sci.*, vol. 76, nos. 3–4, pp. 154–203, May 2010. doi: 10.1016/j.jcss.2009.06.004.
[42] Y. Zhang F. Dan, J. Hong, X. Wen, F. Min, F. Huang, and Y. Zhou, "A fast asymmetric extremum content defined chunking algorithm for data deduplication in backup storage systems," *IEEE Trans. Comput.*, vol. 66, no. 2, pp. 199–211, Feb. 2017.
[43] R. N. S. Widodo, H. Lim, and M. Atiquzzaman, "A new content-defined chunking algorithm for data deduplication in cloud storage," *Future Gener. Comput. Syst.*, vol. 71, pp. 145–156, Jun. 2017.
[44] Y. Tan and Z. Yan, "Multi-objective metrics to evaluate deduplication approaches," *IEEE Access*, vol. 5, pp. 5366–5377, 2017.
[45] B. Chapuis, B. Garbinato, and P. Andritsos, "Throughput: A key performance measure of content-defined chunking algorithms," in *Proc. IEEE 36th Int. Conf. Distrib. Comput. Syst. Workshops (ICDCSW)*, Jun. 2016, pp. 7–12.
[46] Z. J. Sun, G. Kuenning, S. Mandal, P. Shilane, V. Tarasov, N. Xiao, and E. Zadok, "Cluster and single-node analysis of long-term deduplication patterns," *ACM Trans. Storage*, vol. 14, no. 2, pp. 13-1–13-27, May 2018. doi: 10.1145/3183890.
[47] X. Wen, J. Hong, F. Dan, T. Lei, F. Min, and Y. Zhou, "Ddelta: A deduplication-inspired fast delta compression approach," *Perform. Eval.*, vol. 79, no. 2, pp. 258–272, 2014.
[48] G. Wang, S. Chen, M. Lin, and X. Liu, "SBBS: A sliding blocking algorithm with backtracking sub-blocks for duplicate data detection," *Expert Syst. Appl.*, vol. 41, no. 5, pp. 2415–2423, 2014. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S095741741300794X
[49] M. Murugan, K. Kant, A. Raghavan, and D. H. Du, "Software defined deduplicated replica management in scale-out storage systems," *Future Gener. Comput. Syst.*, vol. 97, pp. 340–354, Aug. 2019. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167739X18323884
[50] L. Aronovich, R. Asher, D. Harnik, M. Hirsch, S. T. Klein, and Y. Toaff, "Similarity based deduplication with small data chunks," *Discrete Appl. Math.*, vol. 212, pp. 10–22, Oct. 2016. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0166218X15004795
[51] Z. Cao, H. Wen, X. Ge, J. Ma, J. Diehl, and D. H. C. Du, "TDDFS: A tier-aware data deduplication-based file system," *ACM Trans. Storage*, vol. 15, no. 1, pp. 4-1–4-26, Feb. 2019. doi: 10.1145/3295461.
[52] G. P. Wagner and J. Draghi, "Evolution of evolvability in a developmental model," *Evolution*, vol. 62, no. 2, pp. 301–315, 2008. doi: 10.1111/j.1558-5646.2007.00303.x.

**CHANGJIAN ZHANG** received the B.S. degree from the Department of Computer Science and Technology, Xi 'an Electronic and Engineering University, in 2013. He is currently pursuing the Ph.D. degree in computer science and technology with the South China University of Technology.

Since 2015, he has been studying on data synchronization for big data with the South China University of Technology. His research interests include data synchronization, data compression, data storage, and their applications in cloud computing and big data.

**DEYU QI** received the M.S. degree from the National University of Defense Technology and the Ph.D. degree from the South University of Technology, where he is currently a Full Professor and a Doctoral Supervisor with the School of Computer Science and Engineering, the academic Team Leader of the advanced computing architecture, and the Director of the Research Institute of Computer Systems. His research interests include software developing method and architecture, software developing environment and tools, distributed computing system, the new generation computer architecture, and computer system security. He has published over 200 journal papers, one monograph, and two educational materials. He also obtained many patent for invention and software copyright. He has also held the 863 project, and NSFC project. He proposed the VLSI dynamic analysis method fanalysis, object-oriented LOODS abstract model, the large granularity distributed application system inter-operation model XIOM, and multi-database middle-ware DoD.

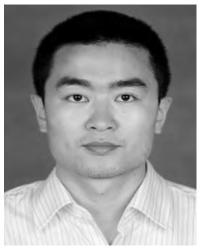


**ZHE CAI** was born in Shantou, Guangdong, China, in 1994. He received the bachelor's degree in computer science and technology from South China Normal University, in 2017, where he is currently pursuing the master's degree in software engineering. His research interests include Java web development, development under Unix environment systems, and C++ language. He has a long internship in a software company, so he is familiar with software development.

**WENHAO HUANG** received the B.S. degree from Yangtze University, the M.S. degree from the South China University of Technology, and the Ph.D. degree with the South China University of Technology. His research interests include intelligent manufacturing, the Internet of Things, deep learning, machine learning, and software engineering. He is currently taking his research for his Ph.D. degree with the Research Institute of Computer Systems, South China University of Technology.

**XINYANG WANG** received the Ph.D. degree from the South China University of Technology. He has authored and coauthored some papers in areas of parallel and distributed system architecture, parallel computing, and network topology properties. His research interests include on computer network topology, parallel computing, cloud computing, heterogeneous data integration, and network fault-tolerance.

**WENLIN LI** received the B.S. and M.S. degrees from Northeast Forestry University, in 2013 and 2016, respectively. She is currently pursuing the Ph.D. degree in computer science and technology with the South China University of Technology.



**JING GUO** received the B.Eng. degree from Sun Yat-sen University, the M.S. degree from Soochow University, and the Ph.D. degree with the South China University of Technology. He has authored and coauthored some papers in areas of numerical solution of differential equations. His research interests include numerical analysis, data analysis, machine learning, and software engineering. He is currently taking his research for his Ph.D. degree with the Research Institute of Computer Systems, South China University of Technology.