2009

# A Running Time Improvement for Two Thresholds Two Divisors Algorithm

BingChun Chang

*San Jose State University*

# A RUNNING TIME IMPROVEMENT FOR TWO THRESHOLDS TWO DIVISORS ALGORITHM

A Project Report
Presented to
The Faculty of the Department of Computer Science
San Jose State University

In Partial Fulfillment
Of the Requirements for the Degree
Master of Computer Science

By
BingChun Chang
December 2009

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

Dr. Teng Moh Department of Computer Science Date

Dr. Agustin Araya Department of Computer Science Date

Dr. Sami Khuri Department of Computer Science Date

APPROVED FOR THE UNIVERSITY

ABSTRACT

# A RUNNING TIME IMPROVEMENT FOR TWO THRESHOLDS TWO DIVISORS ALGORITHM

by BingChun Chang

Chunking algorithms play an important role in data de-duplication systems. The Basic Sliding Window (BSW) algorithm is the first prototype of the content-based chunking algorithm which can handle most types of data. The Two Thresholds Two Divisors (TTTD) algorithm was proposed to improve the BSW algorithm in terms of controlling the variations of the chunk-size. In this project, we investigate and compare the BSW algorithm and TTTD algorithm from different factors by a series of systematic experiments. Up to now, no paper conducts these experimental evaluations for these two algorithms. This is the first value of this paper. According to our analyses and the results of experiments, we provide a running time improvement for the TTTD algorithm. Our new solution reduces about 7 % of the total running time and also reduces about 50 % of the large-sized chunks while comparing with the original TTTD algorithm and make average chunk-size closer to the expected chunk-size. These significant results are the second important value of this project.

## Table of Contents

## List of Figures

## List of Tables

## 1. Background

In this section, we provide related information as the fundamental knowledge in order to obtain the basic understanding about our project.

### *1.1 Data De-Duplication*

A research [1] estimated the 13 % of mid-sized enterprises used more than 10 terabytes of data storage in 2004, but this percentage had increased to 42 % in 2008. This increasing percentage indicates that the enterprises need to deploy more storage systems to handle and manage the information. Most of the information, however, is redundant. For example, performing system backup creates identical copies, same e-mail attachments are sent to multiple receivers, or different users duplicate same documents for individual working. In brief, deploying more data storage systems requires more energy to process information and more network resources to transmit information.

To reduce the costs, more and more enterprises are using data de-duplication technology [2]. The concept of data de-duplication technology is to identify redundant data, store only one copy for all duplicate data, and create logical reference to the copy so that users can access the data when needed. Decreasing the amount of data equals reducing a lot of costs for storage requirement, power consumption, and equipment maintenance. Moreover, de-duplication technology also makes data replication and data recovery more efficient and effective. We introduce two different schemes, the hash-based and the content-aware schemes, which are used for most popular data de-duplication systems.

### *1.2 Hash-Based Approach*

According the research [3], a hash-based de-duplication system consists of three components - file chunking, hash value generation, and redundancy detection. The Figure 1 shows these components and illustrates the concept of the hash-based approach. Simply speaking (see Figure 1), when a new file arrives, the de-duplication system breaks entire file into many small blocks as know as chunks. Then the system uses hash algorithm, like Secure Hash Algorithm - 1 (SHA-1) or Message Digest Algorithm - 5 (MD-5), to generate the unique signatures to present these chunks. Now, the de-duplication system can more easily and accurately compare these signatures with its database or lookup table to identify these small chunks are new data or redundant data. During the comparisons, if the system cannot find any signature matching in its lookup table or database, then the system assumes these chunks are new data and stores them. If the system detects a signature matching, then it creates a logical reference to the duplicate data whose copy was already stored in database.

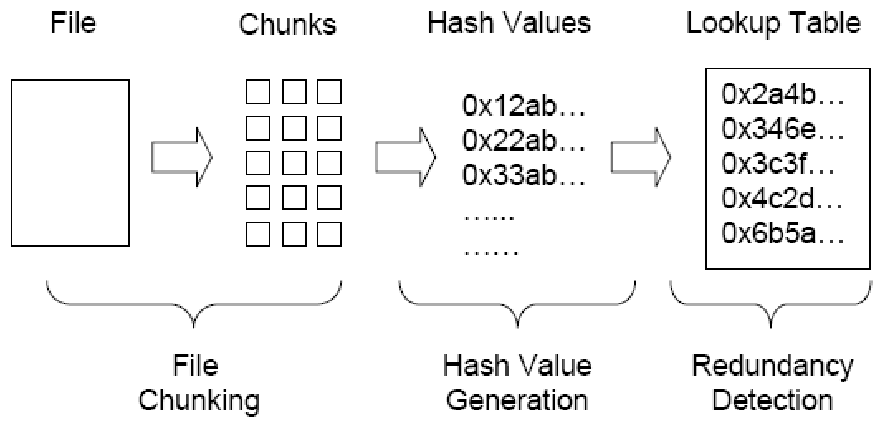

**Figure 1: Three Main Components in Hash-Based De-Duplication System.**

## *1.3 Content-Aware Approach*

In terms of the content-aware scheme [4], this approach firstly compares the new incoming file with its databases to identify the similarities and relationships (e.g., html files to html files or pdf documents to pdf documents). Once the system identified what specific the incoming file is, it then chooses a similar file as a reference file depending on file's name, path, or other related information. After that, the content-aware data de-duplication system performs a byte-to-byte or block-to-block comparisons after chunking. During the comparisons, the de-duplication system computes the *delta* [5] to present the duplicates and the differences between the new incoming file and its reference file. Finally, the system only stores the *delta* and a pointer which points to the reference file rather than entire incoming file.

We provide an example to illustrate the byte-to-byte comparisons shown in Figure 2. In

Figure 2, (a) presents the content of a new incoming file - *file A*. And (b) shows that the system chooses the *file B* as its reference file after identifying the similarities and relationships. During the byte-to-byte comparisons, the de-duplication system detects there are duplicate data from position 0 to 21 and from position 23 to 41. Then the system computes the *delta*, shown in (c), and only stores the *delta* as well as some related information for *file A* rather than entire *file A*.

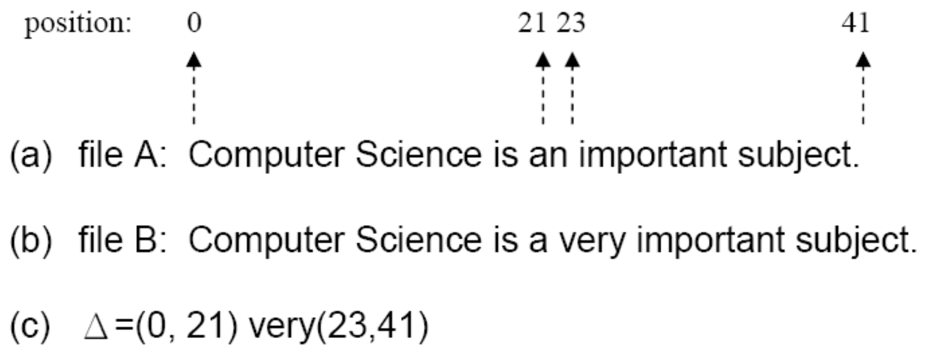
position: 0 21 23 41
(a) file A: Computer Science is an important subject.
(b) file B: Computer Science is a very important subject.
(c) Δ=(0, 21) very(23,41)

**Figure 2: A Byte-To-Byte Comparisons Example for the Content-Aware Scheme. (a) The New Coming File. (b) The Reference File. (c) The *delta.***

These two schemes have been implemented and developed by current industries for different purposes especially in data backup systems. For content-aware scheme, there are products from Sepaton [6, 7] and ExaGrid [8] (byte-level), NetApp [9] (block-level), and Hewlett-Packard [10, 11] (object-level) in current market. In addition, Quantum is the leading company in the hash-based horizon [12]. There are other popular leading companies, like IBM, EMC/Data Domain and Symantec, use their own specific techniques for data de-duplication technology [1].

Among all processes of de-duplication, the chunking algorithm plays an important role no matter which scheme is used for data de-duplication. In brief, the chunking algorithm has many significant influences on the performance of the data de-duplication systems. In our project, our topic focuses on two different chunking algorithms. We provide the fundamental concepts for the chunking algorithms in next section.

# 2. Chunking

Chunking is a process to partition entire file into small pieces of chunks. For any data de-duplication system, chunking is the most time consuming processes since it has to traverse entire file without any exception. The process time of chunking totally depends on how the chunking algorithms break a file. Moreover, the smaller the size of a chunk has, the better result a de-duplication system has. Increasing the number of chunks, however, results in increasing the processing time for both schemes which we presented in previous section. For the hash-based de-duplication systems, increasing the number of chunk also means increasing the size of lookup table, and then the systems need to spend more time to perform the comparisons. The worst case is that the systems cannot load entire lookup table into memory when the size of the lookup table becomes very huge. In this case, the systems have to pay most expensive costs in disk I/O. The content-aware systems face the similar tradeoffs while performing the block-to-block comparisons. In other words, a good chunking algorithm has to satisfy certain conditions such as minimizing the processing time, balancing the scalability and de-duplication ratio, and controlling the variations of chunk-size.

## *2.1 Chunking Level*

According to how to break a file, there are three different chunking categories as shown in Figure 3.

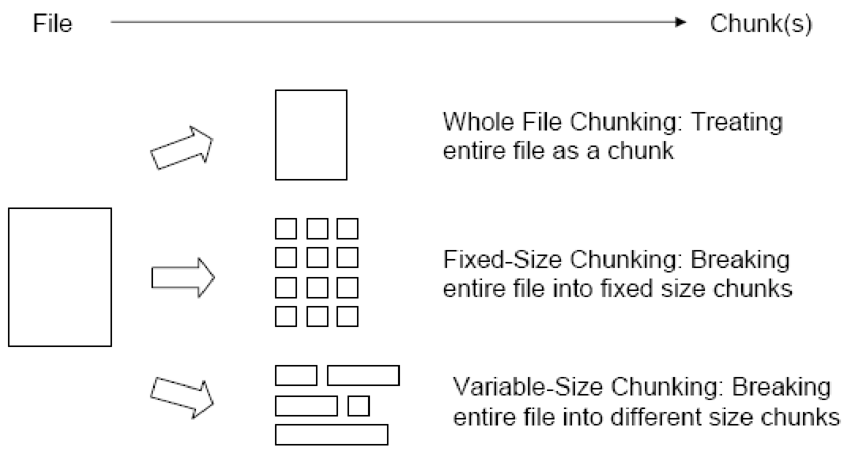
File
Chunk(s)
Whole File Chunking: Treating entire file as a chunk
Fixed-Size Chunking: Breaking entire file into fixed size chunks
Variable-Size Chunking: Breaking entire file into different size chunks

**Figure 3: Three Different Chunking Categories.**

These categories have been studied and researched [13, 14]. Generally speaking, whole-file chunking is the simplest and fastest, but it has the worst results regarding de-duplication ratio. The de-duplication ratio of the fixed-size chunking is totally depending on what the fixed-size is. The smaller the fixed size is, the better de-duplication ratio has. Again, the fixed-size chunking faces the tradeoffs in balancing the capacity scalability and the de-duplication ratio. Moreover, the most important issue of these two chunking algorithms is the boundary shifting problem [15]. We introduce this issue and then discuss the variable size chunking in next section.

## *2.2 Boundary Shifting Problem*

Both whole-file chunking and fixed-size chunking face the boundary shifting problem due to the data modifications. When users only insert or delete one byte, the whole file chunking will result in two different hash values between the modified file and the original file, even if most of the data remain unchanged. In same situation, after one-byte modification happens, the fixed-size chunking will generate totally different results for all the subsequent chunks even though most of the data in the file are unchanged. This problem is called as the boundary shifting problem. We show an example in Figure 4. In Figure 4, (a) shows the case when we break entire file into fixed-size chunks, and (b) shows after inserting 2 bytes string - "*12*" in the *c2* position of the original file, we obtain the totally different results, even other data remain the same.

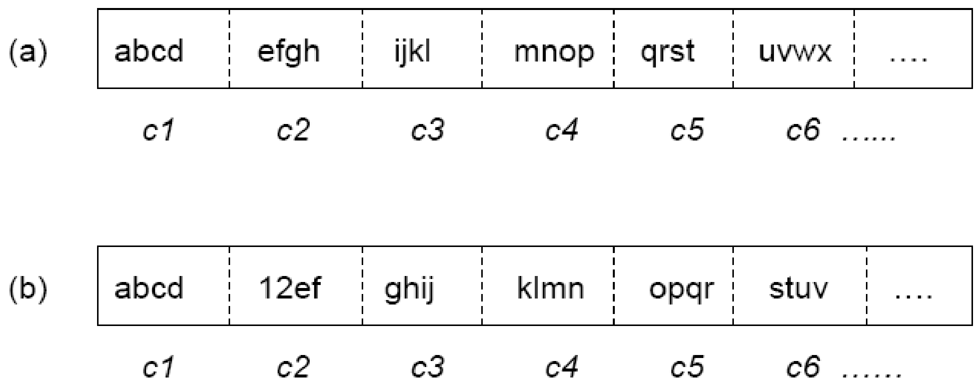

**Figure 4: A Boundary Shifting Problem Example. (a) The Original File. (b) The File after Insertion.**

Variable-size chunking is also called content-based chunking [15]. In other words, chunking algorithms determine chunk boundaries depending on the content of the file. For example, content-based chunking algorithm may determine chunk boundaries by punctuation, each line, or every paragraph. In contrast with fixed-size chunking algorithms which determine chunk boundaries by the distance from the beginning of the file. Therefore when data modifications happen, most of the chunks remain unchanged. This is why variable-size chunking can avoid the boundary shifting problem [3, 16].

## *2.3 Related Work*

Regarding the variable-size chunking algorithm, the paper [17] provided a detailed analyses about variable-size algorithm such as *k-gram* and *0 mod p* algorithms and also proposed a new variable-size algorithm - *Winnowing*. The paper [18] deeply categorized these variable-size chunking algorithms into overlap method such as *k-gram*, *0 mod p*, and *Winnowing* algorithms, and non-overlap method such as *hash-breaking*. Moreover, it also proved that the overlap methods need to use a good chunk selection technique to handle the huge number of chunks. The Basic Sliding Window (BSW) algorithm [16] is the first prototype of the hash-breaking chunking algorithm (non-overlap) and already been proven [16] to obtain the best performance in balancing capacity scalability and the de-duplication ratio. The Two Thresholds, Two Divisors (TTTD) algorithm [15] is the adaptation of the BSW algorithm to improve the problems in the BSW algorithm. The BSW algorithm and the TTTD algorithm are the important cores in our project. We introduce these two algorithms in following sections.

# 3. The Basic Sliding Window (BSW) Algorithm

The BSW algorithm was proposed for a low bandwidth network file (LBFS) system [16]. The main purpose of this algorithm is to reduce the network bandwidth requirements by avoiding the boundary shifting problem. In following sections, we discuss the concepts and problems of BSW algorithm.

## *3.1 Concept of the BSW Algorithm*

In the BSW algorithm, there are three main parameters needed to be pre-configured, a fixed size of window *W*, an integer divisor - *D*, and an integer remainder - *R*, where $R < D$. We describe how the BSW algorithm works as follows (see Figure 5 [15, 19]):

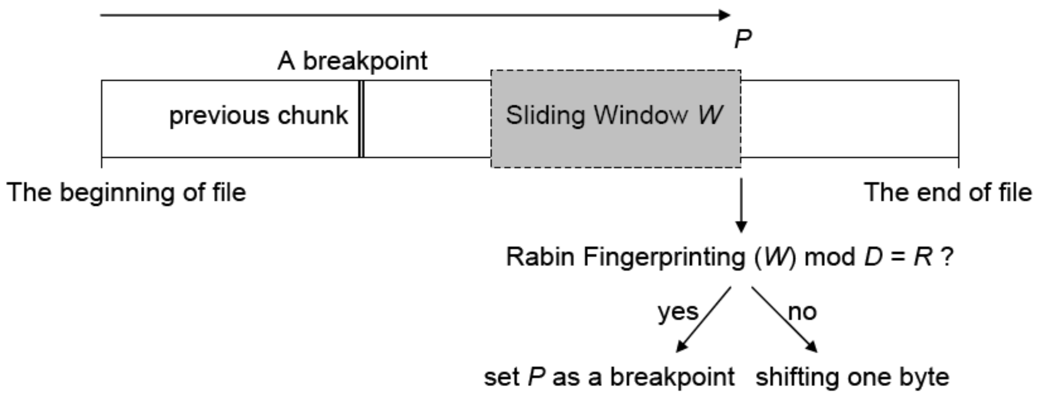

**Figure 5: Concept of the BSW Algorithm.**

(1) A fixed-size window *W* is shifting one byte at one time from the beginning of the file to end of the file.

(2) At every position *p*, uses Rabin Fingerprinting algorithm to compute a hash value *h* for the content of current window.

(3) If $h \bmod D = R$, the position *P* is a breakpoint for chunk boundary. Then the sliding window *W* starts at the position *P*. And repeats the computation and comparison.

(4) If $h \bmod D \neq R$, the sliding window *W* keeps shifting one byte. And repeats the computation and comparison.

Since the BSW algorithm determines the chunk boundaries by the content of file, this approach has been proved to be success to avoid the boundary shifting problem.

## *3.2 The Expected Chunk-Size*

In practice, the parameter *D* plays the most important role in the BSW algorithm because it can be configured to make the chunk-size close to our expectancy. Since any integer divided by *D*, the remainder is between 0 and $D - 1$. The window shifts one byte at one time. In each shifting, the probability of $h \bmod D = R$ is $1/D$. In other words, we expect to find a breakpoint for chunk boundary at every *D* bytes. For example, if we expect the size of every chunk equals to1000 bytes, then we set the value of *D* as 1000 and the value of *R* is any integer where $0 \leqq R \leqq 999$. In the best case, we can always expect to find a matching for $h \bmod D = R$ in every 1000 shifts, that is, 1000 bytes.

## *3.3 Problems of the BSW Algorithm*

There are two main problems in the BSW algorithm. The fist problem is that the sliding window may determine the breakpoint in each shifting in the worst case if the file contains a lot of continuous repeating string such as *aaaaaaaaaa* or *1111111111*. This worst case [10] causes that the metadata has same size as the original file, even has larger size than original file if metadata include extra information for the chunks. The second issue is that if the sliding window cannot find any breakpoint after traversing entire file. In this case, the BSW algorithm may treat entire file as one chunk, then we face the boundary shifting problem again. In other words, the BSW algorithm has very poor controls on the variations of chunk-size and the chunk-size may vary from very small to very large. It is not efficient and effective to transmit very small data or large data due to only one-byte modification.

These two worst cases are very unusual, but it does not mean these problems will never happen. They only waste network resources while these situations happen. In next section, we introduce Two Thresholds and Two Divisors (TTTD) algorithm [15] can help us to solve these problems.

# 4. The Two Thresholds Two Divisors (TTTD) Algorithm

The TTTD algorithm was proposed by HP laboratory [15] at Palo Alto, California. This algorithm use same idea as the BSW algorithm does. In addition, the TTTD algorithm uses four parameters, the maximum threshold, the minimum threshold, the main divisor, and the second divisor, to avoid the problems of the BSW algorithm which we presented in Section 3.3. We discuss the concept in following section.

## *4.1 Concept of the TTTD Algorithm*

The maximum and minimum thresholds are used to eliminate very large-sized and very small-sized chunks in order to control the variations of chunk-size. The main divisor plays the same role as the BSW algorithm and can be used to make the chunk-size close to our expected chunk-size. In usual, the value of the second divisor is half of the main divisor. Due to its higher probability, second divisor assists algorithm to determine a backup breakpoint for chunks in case the algorithm cannot find any breakpoint by main divisor. According to the research [15], we list these four parameters and their optimal values as shown in Table 1 when considering the expected chunk-size is 1000 bytes.

**Table 1: Purposes and Optimal Value of Four Parameters in the TTTD Algorithm.**

| Parameter | Purpose | Optimal Value |
|---|---|---|
| Maximum Threshold | To reduce very large chunks | 2800 (bytes) |
| Minimum Threshold | To reduce very small chunks | 460 (bytes) |
| Main Divisor | To determine breakpoint same as the BSW | 540 |
| Second Divisor | To determined a backup breakpoint | 270 |

Since the TTTD Algorithm uses the same concept as the BSW algorithm does. We describe how it works as follows and provide the pseudo code [15] in Figure 6.

(1) The algorithm shifts one byte at one time and computes the hash value.

(2) If the size from last breakpoint to current position is larger than minimum

threshold, it starts to determine the breakpoint by second and main divisors.

(3) Before the algorithm reaches the maximum threshold, if it can find a breakpoint by main divisor, then uses it as the chunk boundary. The sliding window starts at this position and repeats the computation and comparison until the end of file.

(4) When the algorithm reaches the maximum threshold, it uses the backup breakpoint if it found any one, otherwise use the maximum threshold as a breakpoint.

```
int currP = 0, lastP = 0, backupBreak = 0 ;

for ( ; ! endOfFile( input ) ; currP++ ) {
   unsigned char c = getNextByte( input ) ;
   unsigned int hash = updateHash( c ) ;

   if ( currP – lastP < minT ) {
      continue ;
   }
   if (( hash % secondD ) = = secondD – 1 ) {
      backupBreak = currP ;
   }
   if (( hash % mainD ) = = mainD – 1 ) {
      addBreakpoint( currP ) ;
      backupBreak = 0 ;
      lastP = currP ;
      continue ;
   }
   if ( currP – lastP < maxT ) {
      continue ;
   }
   if ( backupBreak ! = 0 ) {
      addBreakpoint( backupBreak ) ;
      lastP = backupBreak ;
      backupBreak = 0 ;
   }
   else {
      addBreakpoint( currP ) ;
      lastP = currP ;
      backupBreak = 0
   }
}
```

**Figure 6: The TTTD Algorithm Pseudo Code.**

## *4.2 Problems of the TTTD Algorithm*

The first problem is the tradeoffs. In order to control the variations of chunk-size, the TTTD algorithm uses the minimum and maximum thresholds to eliminate very large-sized and very small-sized chunks. Obviously, these eliminations cost the TTTD algorithm to increase the total number of chunks. Increasing the total number of chunks also increases the amount of metadata, the size of lookup table, and the lookup table searching time. The second issue is the role of the second divisor. In following discussion, we assume that the algorithm cannot find the breakpoint by main divisor and we use the optimal configuration in Table 1 (in *page 12*) to discuss this issue.

The values of the main divisor and second divisor are 540 and 270 respectively. In other words, the probability which breakpoints are determined by second divisor is twice larger than the first divisor. From 460 to 2800, these are about 4 chances to obtain the breakpoint by the main divisor (2800 – 460 = 2340 and 2340/540 ≈ 4.33). Also, there are about 8 chances to obtain the backup breakpoint by the second divisor (2340/270 ≈ 8.66).

However, if we carefully review the algorithm from line 22 to line 31 in Figure 6 (in *page 13*), the TTTD algorithm decides whether it uses a backup breakpoint or maximum threshold as the chunk boundary only until the algorithm reaches the maximum threshold. In other words, the algorithm wastes time in calculations and comparison even if it can find a backup breakpoint earlier during the total 8 chances. This situation also implies the algorithm will always pick up the last backup breakpoint it found as the chunk boundary and the chunk-size which is determined by second divisor will be closer to the maximum threshold. In other words, the second divisor only prevents the algorithm from using the maximum threshold as the breakpoint and plays a trivial role in the TTTD algorithm.

The above analyses point out that the TTTD algorithm obviously increases both total running time and total number of chunks in order to obtain better controls on the variations of chunk-size while comparing with the BSW algorithm. If these increasing ratios are very large, said 1:2 or 1:3, then the TTTD algorithm needed to be re-considered. In the next section, we perform a series of experiments in order to compare and evaluate the BSW and the TTTD algorithms in terms of these factors.

## 5. Experimental Comparisons and Evaluations

In this section, we perform the complete experiments to compare and evaluate the BSW and the TTTD algorithms. Up to now, no report conducts these experimental analyses for BSW and TTTD algorithms. This is the first important value of this paper. The goals of our experiments are to compare the performances for BSW and TTTD algorithms and also prove our observations and assumptions about the tradeoffs and the issues of the second divisor for the TTTD algorithm.

### *5.1 Experimental Configurations*

We perform our experiments on a machine with following hardware equipments:

- Intel Core2 T7200 2.00GHz processor.
- 2GBytes Physical Memory.
- 120GBytes SATA-2 hard disk drive, 5400 RPM, Toshiba.

We implement BSW and TTTD algorithms by C++ and use Visual Studio 6.0 as the compiler. In our experiments, the expected chunk-size is 1000 bytes and we use the same parameter configurations as the original paper [15] in order to perform fair and accurate evaluations. We use *main divisor* – 1 as the remainder *R* for two algorithms and *second divisor* –1 as the remainder to determine the backup breakpoint in the TTTD algorithm. We list the parameter configurations for two algorithms in Table 2.

**Table 2: Parameters Configuration for the BSW and the TTTD Algorithms**

| **Algorithm** / **Parameter** | **BSW** | **TTTD** |
|---|---|---|
| Window Size (bytes) | 48 | 48 |
| Main Divisor | 1000 | 540 |
| Second Divisor | N/A | 270 |
| Maximum Threshold | N/A | 2800 |
| Minimum Threshold | N/A | 460 |

## *5.2 Experimental Datasets*

We use open resources as our test data and download all test data from GNU website [20]. Table 3 provides the summary about our test data sets. Please refer to Appendix – A and Appendix – B for the detailed information.

**Table 3: Test Data Sets of the Experiments.**

| **Data Set** | **#1** | **#2** | **#3** | **#4** |
|---|---|---|---|---|
| Data Name | Emacs | Emacs | GNU Manuals | GNU Manuals |
| Data Type | tar | source code | html | txt |
| No. of Files | 5 | 16994 | 40 | 40 |
| Total Size (MB) | 171.3 | 607.2 | 36 | 22.3 |

Data Set #1 contains five versions of Emacs from version 21.4 to version 23.1. They are all compressed tar files. We uncompress all tar files from Data Set#1 as our Data Set #2. Most of the files in Data Set #2 are *.c* and *.h* source code. We use the software manuals from GNU Manual Online [20] for our Data Set #3 and #4. The contents of Data Set #3 and #4 are same as each other except that there are different format – html and txt. We carefully choose a variety of data sets depending on their types, size, formats, and number of files. We believe these data can help us to perform the comprehensive evaluations. Finally, we would like to point out three special cases that chunks cannot be determined by algorithm itself.

(1) The size of file is smaller than window size or minimum threshold.

(2) The fragment, from last break point to end of file, whose size is smaller than window size or minimum threshold.

(3) The algorithm cannot find any breakpoint from last break point to end of file, even if the size of this fragment is larger than window size or minimum threshold.

Of course, these three situations are unusual. We carefully choose our test data to avoid the first case. For second and third case, we take this fragment as one chunk regardless what the size is.

## *5.3 Results of Experiments*

We present the results of our experiments and take deeper analyses for the BSW and TTTD algorithms in this section.

### 5.3.1 Running Time, Total Chunks, and Average Chunk-Size

We discussed the importance for running time, total number of chunks, and average chunk-size in Section 3.2 and Section 4.2. The first thing we are interested in is how the performances of the BSW and the TTTD algorithms are in term of these factors. We present the results in Table 4.

**Table 4: Running Time, Total Chunks and Average Chunk-Size Comparisons for the BSW and the TTTD Algorithms.**

| | **Total Running Time (sec)** | | **Total number of Chunks** | | **Average Chunk-Size (bytes)** | |
|---|---|---|---|---|---|---|
| **Data Set** | **BSW** | **TTTD** | **BSW** | **TTTD** | **BSW** | **TTTD** |
| #1 | 2910 | 2885 | 172874 | 182582 | 1040 | 985 |
| #2 | 10568 | 11011 | 391036 | 481963 | 1629 | 1321 |
| #3 | 617 | 639 | 24692 | 32364 | 1532 | 1169 |
| #4 | 381 | 398 | 17803 | 19590 | 1316 | 1196 |
| Average | 3619 | 3733 | 151601 | 179125 | 1379 | 1168 |

In Table 4, we clearly understand the TTTD algorithm increase the total running time and total number of chunks. In average case, the ratio of the BSW algorithm to the TTTD algorithm is about 1:1.03 for running time, and is about 1:1.18 for the total number of chunks. These two ratios are close to 1:1 and show the tradeoffs for the TTTD algorithm are very small.

Regarding the average chunk-size, the results of the TTTD algorithm is much closer to our expected chunk-size (1000 bytes) than the BSW algorithm for all data sets since the TTTD algorithm has a better control on the variations of chunk-size. Comparing with the BSW algorithm, the TTTD algorithm obtains much better performance without too many concessions.

### 5.3.2 Maximum and Minimum Chunk-Size

The second interesting we would like to present is how better the TTTD algorithm controls the variations of chunk-size. We compare the maximum and minimum chink-size for the BSW and TTTD algorithms. The result is presented on Table 5.

**Table 5: The Maximum and Minimum Chunk-Size for the BSW and the TTTD Algorithms.**

| | **Max Chunk-Size (bytes)** | | **Min Chunk-Size (bytes)** | |
|---|---|---|---|---|
| **Data Set** | **BSW** | **TTTD** | **BSW** | **TTTD** |
| #1 | 16442 | 2800 | 48 | 412 |
| #2 | 154075 | 2800 | 8 | 8 |
| #3 | 97168 | 2800 | 48 | 68 |
| #4 | 68224 | 2800 | 48 | 62 |
| Average | 83977 | 2800 | 38 | 138 |

For the TTTD algorithm, all the maximum chunk-sizes are 2800 bytes in all data sets. This is because the TTTD algorithm uses maximum threshold to limit the maximum chunk-size. The BSW algorithm, however, has a huge variation in maximum chunk-size among all data sets. In average case, the difference of maximum chunk-size for two algorithms is 81177 bytes. This difference is large enough to affect how average chunk-size closes to the expected chunk-size. This difference also indicates that the BSW algorithm needs to consume more network resource to transmit large-sized chunk while data modifications happen. Theoretically speaking, the minimum chunk-size should be 460 bytes for TTTD algorithm. The results are little bit different from assumption. This is because the special cases, the fragments, which we presented in the Section 5.2.

The results illustrate why the maximum threshold plays an important role in the TTTD algorithm and how the TTTD algorithm controls the variations of chunk-size by eliminating the large size of chunks which the BSW algorithm generates.

### 5.3.3 Chunk-Size Distributions

In the last part of comparison, we consider the chunk-size distribution for two algorithms. According to the features of two algorithms, we take the chunk-size which is from 0 to 48 bytes (window size) as the first interval for the BSW algorithm, and from 0 to 460 bytes (minimum threshold) as the first interval for TTTD algorithm. After that, we simply increase 400 bytes for each interval to group our statistical data. We present the results of the BSW algorithm in Table 6 and the TTTD algorithm in Table 7.

**Table 6: Chunk-Size Distributions of the BSW Algorithm.**

| | **Data Set #** | | | | |
|---|---|---|---|---|---|
| **Interval (bytes)** | **#1** | **#2** | **#3** | **#4** | **Average** |
| < 48 (%) | 0 | 0.01 | 0 | 0 | 0.002 |
| 48 ~ 459 (%) | 34.14 | 42.64 | 41.04 | 43.28 | 40.28 |
| 460 ~ 799 (%) | 19.35 | 13.55 | 14.1 | 14.62 | 15.41 |
| 800 ~ 1199 (%) | 15.3 | 9.23 | 10.3 | 11.2 | 11.51 |
| 1200 ~ 1599 (%) | 10.29 | 6.35 | 7.24 | 6.94 | 7.71 |
| 1600 ~ 1999 (%) | 6.93 | 4.93 | 5.27 | 5.63 | 5.69 |
| 2000 ~ 2399 (%) | 4.51 | 3.79 | 4.09 | 3.65 | 4.01 |
| 2400 ~ 2799 (%) | 3.07 | 3.07 | 3.06 | 3 | 3.05 |
| >= 2800 (%) | 6.41 | 16.41 | 14.91 | 11.67 | 12.35 |

Firstly, we re-indicate the problems of the fragment which we presented in Section 4.2. In the average case of the first interval, the BSW algorithm is 0.002 % (in Table 6) and the TTTD algorithm is 0.28 % (in Table 7). The two very small numbers present the percentage of the fragments. These fragments are totally depending on how we conduct the experiments, that is, how many files the algorithm processes at one time. For example, data set #1 only has 5 files (in Table 3, *page 16*) needed to be processed at one time, so both Table 6 and Table 7 show the percentages are close to 0. Moreover, both Table 6 and Table 7 also show the data set #2 has the largest percentage because data set #2 has 16994 files (in Table 3, *page 16*) needed to be handled at one time. Even though each file has at most one fragment, the total percentage is too small to affect the results of experiments and we can simply ignore it.

**Table 7: Chunk-Size Distributions of the TTTD Algorithm.**

| | **Data Set #** | | | | |
|---|---|---|---|---|---|
| **Interval (bytes)** | **#1** | **#2** | **#3** | **#4** | **Average** |
| < 460 (%) | 0 | 1.03 | 0.05 | 0.05 | 0.28 |
| 460 ~ 799 (%) | 47.0 | 32.17 | 42.88 | 39.75 | 40.45 |
| 800 ~ 1199 (%) | 27.7 | 20.78 | 21.47 | 22.54 | 23.12 |
| 1200 ~ 1599 (%) | 13.4 | 13.63 | 11.15 | 12.28 | 12.62 |
| 1600 ~ 1999 (%) | 6.43 | 10.09 | 8.48 | 8.96 | 8.49 |
| 2000 ~ 2399 (%) | 3.34 | 8.91 | 6.19 | 6.88 | 6.33 |
| 2400 ~ 2799 (%) | 2.11 | 11.3 | 6.67 | 7.17 | 6.81 |
| = 2800 (%) | 0.03 | 2.09 | 3.11 | 2.36 | 1.9 |

Secondly, Table 6 shows that the most of the chunks, 40.28 % in average, are determined before 460 bytes in the BSW algorithm. Table 7 indicates that most of the chunks, 40.45 % in average, are determined from 460 to 799 bytes in the TTTD algorithm. In other words, the minimum threshold of the TTTD algorithm successfully reduces the very small-sized chunks. The percentages of the last interval in average case, 12.35 % (BSW algorithm) and 1.9 % (TTTD algorithm), also show the maximum threshold of the TTTD algorithm really plays a good role to limit the large-sized chunks.

According to the above comparisons and evaluations, the TTTD algorithm forces the average chunk-size close to expected chunk-size by a better control on the variations of chunk-size. In brief, the TTTD algorithm only increases a little cost in total running time and total number of chunks to obtain the better performance than the BSW algorithm.

## 6. New Improvement of the TTTD Algorithm

The second important value of this paper is that we proposal a new improvement of the TTTD and this new improvement really obtains some significant achievements. In this section, we review the problems of the TTTD algorithm and introduce our improvement. We also perform the experiments to evaluate our new solution.

### *6.1 Review for problems of the Second Divisor*

We discussed the problems of the TTTD algorithm in the Section 4.2. The major issue comes from the second divisor. Simply speaking, the TTTD algorithm decides whether using the backup breakpoint until the algorithm reaches the maximum threshold. We measure the total number of chunks and their percentages depending on how they are determined. We present the result in Table 8.

**Table 8: The Percentage of the Chunks Determined by Main Divisor, Second Divisor, and the Maximum Threshold in the TTTD Algorithm.**

| | **Data Set #** | | | | |
|---|---|---|---|---|---|
| **Chunk Determined by** | **#1** | **#2** | **#3** | **#4** | **Average** |
| Main Divisor | 180156 (98.67%) | 391106 (84.04%) | 28784 (89.04%) | 17470 (89.36%) | 90.3% |
| Second Divisor | 2374 (1.3%) | 64481 (13.86%) | 2545 (7.87%) | 1619 (8.28%) | 7.8 % |
| Max Threshold | 47 (0.03%) | 9795 (2.1%) | 997 (3.08%) | 461 (2.36%) | 1.9 % |

In average case, most of the chunks are determined by main divisor (90.3 %) and only 7.8 % of the total chunks are determined by second divisor. We re-present the data which are shown in Table 7 (in *page 20*) as the Figure 7 to compare the results with Table 8 and we can obtain a clearer view.

In Figure 7, we can see the first peak appears between 460 and 799 bytes. No doubt this peak presents the chunks which are determined by the main divisor. The second peak, between 2400 bytes and 2799 bytes, indicates that chunks which are determined by second divisor. Obviously, this peak is close to the maximum threshold. Combining Figure 7 with the Table 8 (in p*age 21*), the second peak implies the second divisor wants to prevent the TTTD algorithm from using the maximum threshold as the breakpoints, but the result only has slightly different. In other words, the TTTD algorithm wastes extra time in unnecessary calculations and comparisons due to the second divisor. It also has a poor control on the variations of chunk-size since these chunks which are determined by the second divisor are larger and close to the maximum threshold. From the results of our experiments, we prove our analyses in section 4.2.

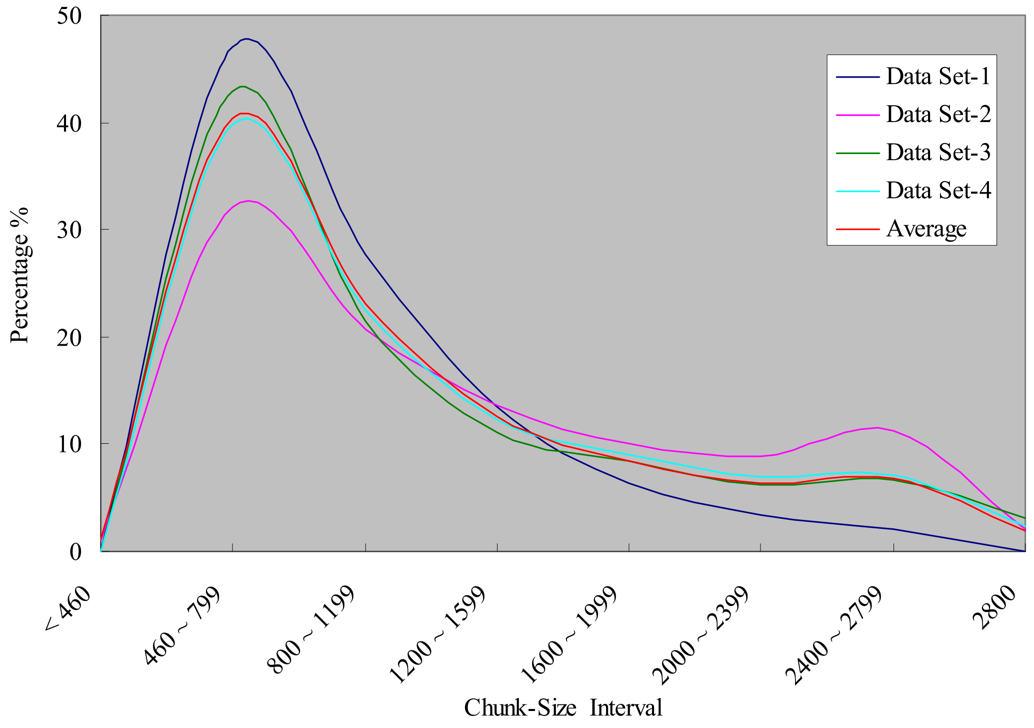

**Figure 7: Chunk-Size Distributions of the TTTD Algorithm.**

## *6.2 Concept of the New Improvement*

We cannot simply remove the second divisor from the algorithm in order to achieve our targets. If we remove the second divisor, then about 10 % of the total chunks will be determined by maximum threshold. In this case, the 10 % of the total chunks are chunked by fixed-size chunking. In other words, they may face the boundary shifting problem again. Moreover, we cannot adjust the value of the maximum threshold to be smaller either. According to the research [15], the value of the maximum threshold should be 2.8 times average chunk-size due to the simplifying system design for components such as buffer size or packet size.

However, if we can make second peak happen earlier, which means the chunks are determined earlier, then we can reduce the unnecessary calculations and comparisons to increase the running time. By this approach, we also obtain the better controls on the variations of chunk-size by reducing the large-size chunks. We use this concept to improve the TTTD algorithm. For convenience, we assume our expected chunk-size is 1000 bytes and use the same optimal parameters' configuration as the original TTTD algorithm to explain our concept.

According to the results of our experiments, we know that about 70 % of the total chunks which are determined before 1600 bytes are determined by main divisor. We also know that the second peak begins at 2400 bytes. The concept of our new improvement is that we use a new parameter, where 1500 < *new parameter* < 2400. When the size from the previous breakpoint to current position is larger than this parameter, then we use the value of the second divisor as the value of the main divisor and use 1/2 of the original value of second divisor as the new value of the second divisor. After we found a breakpoint, then we switch the values back to the original values for two divisors. By increasing the probability of the main divisor after the first peak, we expect the second peak will happen earlier.

Since we use the new parameter to switch the values of two divisors at the specific position, we call this new parameter as *switchP*; and hence, we call our improvement as TTTD-S in the following discussions. We provide the pseudo code for the TTTD-S algorithm in Figure 8. We perform the same experiments to evaluate our approach and the original TTTD algorithm. In Figure 8, the TTTD-S algorithm uses the function - *switchDivisor( )*, in line12, to reduce values of the main divisor and second divisor. Once the algorithm determines the breakpoints, it uses function - *resetDivisor( )*, in line21, 31, and 37, to set the main divisor and second divisor back to original values.

```
int currP = 0, lastP = 0, backupBreak = 0 ;

for ( ; ! endOfFile( input ) ; currP++ ) {
   unsigned char c = getNextByte( input ) ;
   unsigned int hash = updateHash( c ) ;

   if ( currP – lastP < minT ) {
      continue ;
   }
   if ( currP – lastP > switchP )
   {
      switchDivisor( ) ;
   }
   if (( hash % secondD ) = = secondD – 1 )
      backupBreak = currP ;
   }
   if (( hash % mainD ) = = mainD – 1 ) {
      addBreakpoint( currP ) ;
      backupBreak = 0 ;
      lastP = currP ;
      resetDivisor( ) ;
      continue ;
   }
   if ( currP – lastP < maxT ) {
      continue ;
   }
   if ( backupBreak != 0 ) {
      addBreakpoint( backupBreak ) ;
      lastP = backupBreak ;
      backupBreak = 0 ;
      resetDivisor( ) ;
   }
   else {
      addBreakpoint( currP ) ;
      lastP = currP ;
      backupBreak = 0
      resetDivisor( ) ;
   }
}
```

**Figure 8: The TTTD-S Algorithm Pseudo Code**

## *6.3 Experimental Evaluations for the TTTD-S Algorithm*

We use the same parameters' configuration and same data sets to evaluate our new improvement. We perform a series of systematic experiments and hill-climbing strategy to test different values for the parameter – *switchP*, from 1400 to 2200. We found the best value for the *switchP* is 1600. In practice, the value of *switchP* should be 1.6 times the expected chunk-size.

We perform the same comparisons for the TTTD algorithm and our new improvement – TTTT-S algorithm in terms of running time, total number of chunks, and the average chunk-size. We present the results in Table 9.

**Table 9: The Maximum Chunk-Size and Minimum Chunk-Size Comparisons for the TTTD and the TTTD-S Algorithms.**

| | **Total Running Time (sec)** | | **Total number of Chunks** | | **Average Chunk-Size (bytes)** | |
|---|---|---|---|---|---|---|
| **Data Set** | **TTTD** | **TTTD - S** | **TTTD** | **TTTD - S** | **TTTD** | **TTTD - S** |
| #1 | 2885 | 2818 | 182582 | 186757 | 985 | 963 |
| #2 | 11011 | 10242 | 481963 | 513330 | 1321 | 1241 |
| #3 | 639 | 603 | 32364 | 33360 | 1169 | 1134 |
| #4 | 398 | 379 | 19590 | 20385 | 1196 | 1149 |
| Average | 3733 | 3510 | 179125 | 188458 | 1168 | 1121 |

Firstly, our improvement really reduces the total running time from 3777 seconds to 3510 seconds in average case. This number is even better than 3619 seconds in the BSW algorithm (in Table 4, *page 17*). For running time, the ratio of the TTTD-S algorithm to the TTTD algorithm is 1: 1.07. In other words, we speed up about 7 % running time for the original TTTD algorithm. This is a significant achievement in our project. The average size of all test data sets is only 207.58 MB. However, data de-duplication systems need to deal with the data from hundred GB to TB in reality. Moreover, our new improvement makes the average chunk-size closer to the expected chunk-size from 1168 bytes to 1121 bytes.

Secondly, we present the chunk-size distribution of the TTTD-S algorithm in Table 10 and Figure 9. We found that our new solution does not affect the chunk-size distribution before 1600 bytes. While comparing the percentage of the average case, the TTTD algorithm has 76.47 % before 1600 bytes (in Table 7, *page 20*), and the TTTD-S has 75.74 % before 1600 bytes (in Table 10). These two percentages are similar. In other words, the TTTD-S algorithm does not affect the original behaviors of the main divisor for the TTTD algorithm. Also, our new improvement reduces large-size chunks between 2400 bytes to 2800 bytes from 8.7 % of the TTTD algorithm to 4.4 % of the TTTD-S algorithm. The decreasing ratio is about 50 %.

**Table 10: Chunk-Size Distributions of the TTTD-S Algorithm.**

| | **Data Set #** | | | | |
|---|---|---|---|---|---|
| **Interval (bytes)** | **#1** | **#2** | **#3** | **#4** | **Average** |
| < 460 (%) | 0 | 1.13 | 0.05 | 0.06 | 0.31 |
| 460 ~ 799 (%) | 47.13 | 32.35 | 42.64 | 39.95 | 40.52 |
| 800 ~ 1199 (%) | 27.67 | 20.38 | 21.38 | 22.21 | 22.91 |
| 1200 ~ 1599 (%) | 13.31 | 12.64 | 10.49 | 11.55 | 12.0 |
| 1600 ~ 1999 (%) | 9.23 | 20.62 | 15.26 | 15.34 | 15.11 |
| 2000 ~ 2399 (%) | 2.09 | 7.24 | 5.34 | 5.71 | 5.1 |
| 2400 ~ 2799 (%) | 0.86 | 5.34 | 4.67 | 5.68 | 4.14 |
| = 2800 (%) | 0.0 | 0.3 | 0.25 | 0.53 | 0.27 |

We present the data of the Table 10 in Figure 9 to obtain a deeper view. Comparing with the original TTTD algorithm (in Figure 7, *page 22*), our improvement successfully make the second peak happen earlier. The tradeoff is the increasing total chunks. For the increasing chunks, the ratio of the TTTD algorithm to the TTTD- S algorithm is 1:1.05. This ratio is acceptable when comparing the ratio of the BSW algorithm to the TTTD algorithm - 1:1.18.

Simply speaking, our new improvement obtains the better performance in average only by adding a new parameter and without changing the structure of the original algorithm.

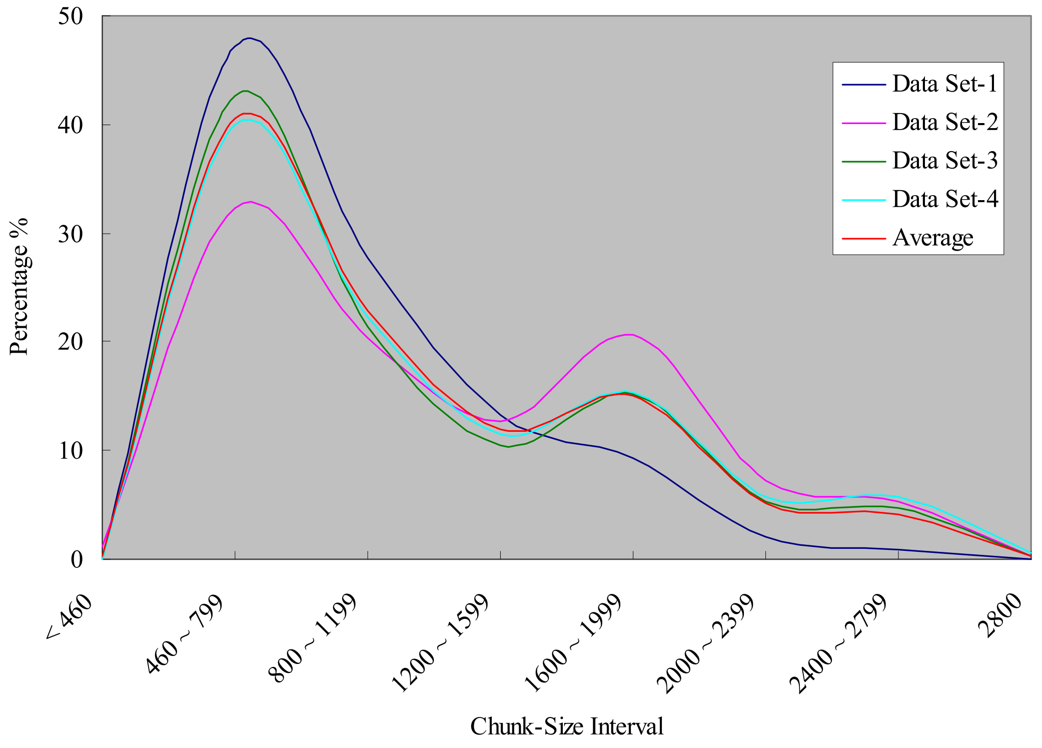

**Figure 9: Chunk-Size Distributions of the TTTD-S Algorithm.**

## 7. Conclusion and Future Work

In brief, our new approach, the TTTD-S algorithm, not only successfully achieves the significant improvements in running time and average chunk-size, but also obtains the better controls on the variations of chunk-size by reducing the large-sized chunks. Our project focuses on the chunking algorithm. However, there are still many related problems and issues needed to be studied for data de-duplication system. For example, the hashing-based data de-duplication systems have to face the natural limitation – the hash collision problem [1]. Moreover, how to index the metadata to speed up the lookup table searching is another interesting topic [12]. Finally, the security issue [1] of the hashing-based data de-duplication cannot be ignored either. These related topics are also worth working on in the future.

## Reference

[1] D. Geer. Reducing the Storage Burden via Data Deduplication. *Computer*, 41(12):15-17, December 2008.

[2] IDC., "White paper: Data Deduplication for Backup: Accelerating and Efficieeency Driving Down IT Costs," 2009. [Online]. Available: http://www.emc.com/collateral/analyst-reports/idc-20090519-data-deduplication.pdf.

[3] Y. W, R. Kim, J. Ban, J.Hur, S. Oh, and J. Lee. PRUN: Eliminating Information Redundancy for Large Scale Data Backup System. *In Computational Sciences and Its Applications*, *ICCSA 2008*, *International Conference on*, Pages 139-144, 2008.

[4] IBM Corporation., "White paper: Redbooks - Guide to Data De-duplication," 2008. [Online]. Available: http://www.redbooks.ibm.com/redpieces/abstracts/sg247652.html?Open

[5] L. You, and C. Karamanolis, Evaluation of Efficient Archival Storage Techniques. in *12th NASA Goddard Conference on Mass Storage Systems and Technologies*. April 2004.

[6] SEPATON, Inc., "White paper: Comparing Deduplication Approaches: Technology Considerations for Enterprise Environments," 2008. [Online]. Available: http://www.sepaton.com/about_us/resourceCenter_whitepapers.php#

[7] SEPATON, Inc., "White paper: Reducing Costs in the Data Center: Comparing Costs and Benefits of Leading Data Protection Technologies," 2007. [Online]. Available: http://www.sepaton.com/about_us/resourceCenter_whitepapers.php#

[8] ExaGrid Systems, "White paper: Data De-duplication Methodologies: Comparing ExaGrid's Byte-Level Data Deduplication to Block Level Data Deduplication," [Online]. Available: http://www.exagrid.com/products/disk-based_backup_whitepapers.asp

[9] The Enterprise Strategy Group, Inc., "Lab Validation Report: NetApp deduplication for FAS - Doing More with Less," 2008. [Online]. Available: http://www.enterprisestrategygroup.com/ESGPublications/ReportDetail.asp?ReportID=1237

[10] Hewlett-Packard Development Company, L.P., "White paper: Understanding the HP Data Deduplication Strategy: Why one size doesn't fit everyone,"2008. [Online]. Available: http://h20195.www2.hp.com/V2/getdocument.aspx?docname=/4AA1-9796ENW.pdf

[11] Hewlett-Packard Development Company, L.P., "White paper: Integrating HP Data Protector software with HP Data Deduplication Solutions,"2009. [Online]. Available: http://h20338.www2.hp.com/ERC/downloads/4AA2-2654ENW.pdf

[12] Quantum Corp., "White paper: Data Deduplication Background: A Technical White Paper" 2009. [Online]. Available: http://salestools.quantum.com/querydocretriever_inc.cfm?ext=.pdf&mime=application/pdf&filename=283835.pdf

[13] N. Mandagere, P. Zhou, M. A. Smith, and S. Uttamchandani. Demystifying Data Deduplication. in *Companion '08: Proceedings of the ACM/IFIP/USENIX Middleware '08 Conference Companion*, pages 12-17, 2008.

[14] D. R. Bobbarjung, S. Jagannathan, and C. Dubnicki, Improving Duplicate Elimination in Storage Systems. *ACM Transactions on Stroage*, 2(4):424-448, November 2006.

[15] K. Eshghi and H.K. Tang . A Framework for Analyzing and Improving Content-Based Chunking Algorithms. Hewlett-Packard Labs Technical Report, TR 2005-30. URL: http://www.hpl.hp.com/techreports/2005/HPL-2005-30R1.html

[16] A. Muthitacharoen, B. Chen, and D. Mazieres, A low-bandwidth network file system. in *Symposium on Operating Systems Principles*, 2001, page 174-187, 2001.

[17] S. Schleimer, D. S. Wilkerson, and A. Aiken. Winnowing: Local Algorithms for Document Fingerprinting. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, page 76 - 85, June 2003.

[18] J. Seo, and W. B. Croft. Local Text Reuse Detection. In *Proceedings of the 31st annual international ACM SIGIR conference on Research and development in information retrieval*, page 105-112, July 2008.

[19] G. Forman, K. Eshghi, and S. Chiocchetti. Finding Similar Files in Large Document Repositories. In *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*, page 394 - 400, July 2008.

[20] GNU website http://www.gnu.org/

[21] D. Bhagwat, K. Eshghi and P. Mehra. Content-based Document Routing and Index Partitioning for Scalable Similarity-based Searches in a Large Corpus. In *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, page 105 – 112, August 2007.

## Appendix – A: Test Data Set #1 and #2 Detailed Information

**Table 11: The Detailed Information for Test Data Set #1 and Data Set #2**

| Data Set # | #1 (tar type) | | #2 (uncompressed type) | |
|---|---|---|---|---|
| File Name | No. of Files | File Size (MB) | No. of Files | File Size (MB) |
| emacs-21.4a | 1 | 19.4 | 2553 | 70.9 |
| emacs-22.1 | 1 | 36.4 | 3492 | 129.5 |
| emacs-22.2 | 1 | 36.9 | 3514 | 131.3 |
| emacs-22.3 | 1 | 37.7 | 3525 | 132.6 |
| emacs-23.1 | 1 | 40.9 | 3910 | 142.9 |
| Total | 5 | 171.3 | 16994 | 607.2 |

## Appendix – B: Test Data Set #3 and #4 Detailed Information

**Table 12: The Detailed Information for Test Data Set #3 and Data Set #4.**

| File Name | No. of Files | File Size (KB) | |
|---|---|---|---|
| | | Data Set #3 (html type) | Data Set #4 (txt type) |
| Autoconf | 1 | 1567 | 915 |
| Autogen | 1 | 1192 | 493 |
| Automake | 1 | 906 | 529 |
| Bash | 1 | 588 | 398 |
| Bison | 1 | 629 | 425 |
| Cflow | 1 | 128 | 81 |
| Coreutils | 1 | 1261 | 665 |
| Diffutils | 1 | 256 | 183 |
| Elisp | 1 | 4361 | 2892 |
| Emacs | 1 | 3266 | 2087 |
| Emacs Lisp Intro | 1 | 1003 | 684 |
| Epsilon | 1 | 242 | 198 |
| Gawk | 1 | 1818 | 1005 |
| Gawkinet | 1 | 268 | 200 |
| GMP | 1 | 512 | 355 |
| GNATS | 1 | 418 | 273 |
| Gnulib | 1 | 2530 | 1177 |
| GnuTLS | 1 | 1107 | 776 |
| Gperf | 1 | 102 | 72 |
| Grep | 1 | 155 | 85 |
| SASL | 1 | 367 | 292 |
| GSL | 1 | 2153 | 1241 |
| GSS | 1 | 262 | 228 |
| Guile | 1 | 2683 | 1697 |
| Guile-RPC | 1 | 144 | 98 |

# Appendix – B: Test Data Set #3 and #4 Detailed Information (Cont'd)

**Table 12: The Detailed Information for Test Data Set #3 and Data Set #4.**

| **File Name** | **No. of Files** | **File Size (KB)** | |
|---|---|---|---|
| | | **Data Set #3 (html type)** | **Data Set #4 (txt type)** |
| Hello | 1 | 45 | 35 |
| ID utils | 1 | 99 | 50 |
| indent | 1 | 136 | 65 |
| Info | 1 | 100 | 61 |
| info standalone | 1 | 166 | 96 |
| Libc | 1 | 4392 | 2772 |
| Libidn | 1 | 277 | 223 |
| Libtool | 1 | 433 | 274 |
| lightning | 1 | 151 | 122 |
| M4 | 1 | 1 | 334 |
| Make | 1 | 793 | 483 |
| Sed | 1 | 161 | 87 |
| Tar | 1 | 1066 | 526 |
| Texinfo | 1 | 1196 | 712 |
| Wdiff | 1 | 19 | 14 |
| total | 40 | 36.0 (MB) | 22.3 (MB) |