A NEW TECHNIQUE: REPLACE ALGORITHM TO RETRIEVE A VERSION FROM A REPOSITORY INSTEAD OF DELTA APPLICATION

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
THE MIDDLE EAST TECHNICAL UNIVERSITY

BY

SÜLEYMAN ONUR OTLU

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
IN
THE DEPARTMENT OF COMPUTER ENGINEERING

APRIL 2004

Approval of the Graduate School of Natural and Applied Sciences.

Prof. Dr. Canan Özgen
Director

I certified that this thesis satisfies all the requirements as a thesis for the degree of Master of Science.

Prof. Dr. Ayse Kiper
Head of Department

This is to certify that we have read this thesis and that in our opinion it is fully adequate, in scope and quality, as a thesis for the degree of Master of Science.

Prof. Dr. Adnan Yazici Co-Supervisor	Assoc. Prof. Dr. Ahmet Cosar Supervisor

Examining Committee Members

Prof. Dr. Adnan Yazici	(Ceng)	
Assoc. Prof. Dr. Ahmet Cosar	(Ceng)	
Assoc. Prof. Dr. Ismail Hakki Toroslu	(Ceng)	
Assoc. Prof. Dr. Nihan Kesim Çiçekli	(Ceng)	
M. Sc. Abdullah Fisne	(Hacettepe Univ.)	

Name, Last name	:Süleyman Onur, OTLU
Signature	:

ABSTRACT

A NEW TECHNIQUE: REPLACE ALGORITHM TO RETRIEVE A VERSION FROM A REPOSITORY INSTEAD OF DELTA APPLICATION

Otlu, Süleyman Onur

M. S., Department of Computer Engineering

Supervisor: Assoc. Prof. Dr. Ahmet Cosar

Co-Supervisor: Prof. Dr. Adnan Yazici

April 2004, 51 Pages

The thesis introduces a new technique to retrieve a version from a repository as an alternative method to applying deltas to literal file sequentially. To my best knowledge; this is the first investigation about delta combination for copy/insert instruction type with many experimental results and conclusions. The thesis proves that the delta combination eliminates unnecessary I/O process for intermediate versions when delta application is considered, therefore reduces I/O time. Deltas are applied to literal sequentially to generate the required version in the classical way. *Replace algorithm* combines delta files which would be applied in delta application as combined delta, and applies it to literal to generate the required one. *Apply* runs in $O(size(D))$ time where D is the destination file and $size(D)$ is its size. To retrieve n^{th} version in a chain where 1st version is literal, it requires $n-1$ time *apply*. *Replace algorithm* runs in $O(i + c * log_2 n)$ time where i is the total length of all inserts, c is the total length of all copies in destination delta, and n is the number of instructions in source delta. To retrieve the same n^{th} version, it requires $n-2$ time *replace* and one *apply*.

Keywords: delta algorithm, delta application, delta combination, replace algorithm.

ÖZ

YENI BIR TEKNIK: VERI HAVUZUNDAN BIR VERSIYONU ÜRETMEK IÇIN FARK UYGULAMASI YERINE DEGISTIRME ALGORITMASI

Otlu, Süleyman Onur

Yüksek Lisans, Bilgisayar Mühendisligi Bölümü

Tez Danismani: Doç. Dr. Ahmet Cosar

Ortak Tez Danismani: Prof. Dr. Adnan Yazici

Nisan 2004, 51 sayfa

Bu tez veri havuzundan bir versiyonu üretmek için fark dosyalarini sabit dosyaya sirayla uygulamak yerine alternatif yöntem olarak yeni bir teknigi tanitmaktadir. Bilgim dahilinde, bu tez bir çok deneysel sonuç veren ve yargilara varan kopya/ekle komut tipi kullanan fark birlestirme konusunda yapilmis ilk arastirmadir. Bu tez, fark uygulama metodunu düsündügümüzde fark birlestirmenin ara versiyonlar için yapilan girdi çikti islemlerini ortadan kaldirdigini ve girdi çikti islem süresinin azaldigini göstermektedir. Klasik mantikta gerekli versiyonu üretmek için fark dosyalari sirasiyla sabit dosyaya uygulanir. *Degistirme algoritmasi* fark uygulamasinda kullanilan fark dosyalarini birlesik fark dosyayi olarak birlestirir ve bu birlesik fark dosyasini sabit dosyaya uygulayarak gerekli versiyonu üretir. *Uygulama O (uzunluk(D))* süresinde çalismaktadir, *D* hedef dosyasidir ve *uzunluk(D)* hedef dosyasinin uzunlugudur. Birinci versiyonu sabit dosya olan bir versiyon zincirinden *n*. versiyonu üretmek *n – 1* defa *uygulamayi* gerektirir. *Degistirme algoritmasi* $O(i + c * \log_2 n)$ süresinde çalismaktadir, *i* hedef fark dosyasindaki ekle komut tiplerinin uzunluklari toplamidir, *c* hedef fark dosyasindaki kopya komut tiplerinin uzunluklari toplamidir ve n kaynak fark dosyasindaki komutlarin sayisidir. Bu yöntemle ayni n. versiyonu üretmek için $n - 2$ defa *degistirme* ve bir defa *uygulama* gerekmektedir.

Anahtar Kelimeler: fark algoritmasi, fark uygulamasi, fark birlestirmesi, degistirme algoritmasi.

To My Family

ACKNOWLEDGEMENTS

I express sincere appreciation to Prof. Dr. Adnan Yazici and Assoc. Prof. Dr. Ahmet Cosar, for their guidance and encouragement throughout the research. I offer sincere thanks to my family for their emotional support.

TABLE OF CONTENTS

ABSTRACT... iv

ÖZ vi

ACKNOWLEDGEMENTS ix

TABLE OF CONTENTS... x

LIST OF TABLES xiii

LIST OF FIGURES.. xiv

CHAPTERS

1. INTRODUCTION... 1

    1.1. Motivation 1

    1.2. Thesis Goals... 3

    1.3. Organization of the Thesis 4

2. BACKGROUND... 5

    2.1. Delta Encoding... 6

    2.2. Encoding Metrics 6

    2.3. Delta Algorithm Concept 7

    2.4. General Greedy Algorithm and Linear Time Delta Algorithms 8

    2.5. Delta Storage Techniques.. 11

    2.6. Delta Application 14

    2.7. Related Work.. 16

    2.8. Implementation Notes.. 18

    2.8.1. Implementation Notes for the Delta Algorithm 18

    2.8.2. Implementation Notes for Apply Algorithm............................... 19

    2.8.3. Implementation Notes for Replace Algorithm 19

3. REPLACE ALGORITHM...20

3.1. Replace Algorithm..20

3.2. An Example to Make Clear How Replace Algorithm works 24

4. COMPLEXITY ANALYSIS...29

4.1. Retrieve Operation..29

4.1.1. Delta Application...29

4.1.2. Replace Algorithm 29

4.2. Delete Operation 31

4.2.1. Delta Application...31

4.2.2. Replace Algorithm 31

5. EXPERIMENTAL RESULTS...33

5.1. Retrieve Operation..34

5.1.1. How Delta Application works for Retrieve Operation.................34

5.1.2. How Delta Combination works for Retrieve Operation 34

5.2. Run Results for Retrieve Operation...35

5.2.1. The performance of Apply algorithm when insert stream is on disk or in memory...35

5.2.2. The performance of Apply algorithm when destination file is constructed on disk completely or with a fixed-size buffer in memory....40

5.2.3. The performance of Replace algorithm using binary search algorithm or hash data structure...41

5.2.4. The performance of Apply and Replace algorithms with a fair comparison..43

5.3. Delete Operation 47

5.3.1. How Delta Application works for Delete Operation....................47

5.3.2. How Delta Combination works for Delete Operation..................47

5.4. Run Results for Delete Operation..48

6. CONCLUSIONS...49

6.1. Future Work...50

REFERENCES 52

APPENDICES

A: THE VERSIONS AND THEIR FILE SIZE OF GNU PACKAGES USED IN EXPERIMENTAL RESULTS 54

B: THE INSTRUCTION STATISTICS OF THE GENERATED VERSION CHAINS AND GNU PACKAGES USED IN EXPERIMENTAL RESULTS..... 56

LIST OF TABLES

Table 1: The statistics of instructions that are applied by apply and replace algorithms for package *gawk* 40
Table 2: The statistics of instructions that are applied by apply and replace algorithms for package *chicken* 40

LIST OF FIGURES

Figure 1: Storing four versions of a file with forward delta technique, version F1 is stored as literal and other versions are stored as delta files in the chain. **2**

Figure 2: An example of encoding a delta file using copy/insert delta encoding............ **5**

Figure 3: The storage mechanism of *forward delta* technique............ 11

Figure 4: Clustered *forward delta* technique............ 11

Figure 5: The storage mechanism of *jumping delta* technique............ 12

Figure 6: The storage mechanism of *reverse delta* technique............ 13

Figure 7: The storage mechanism of branch in RCS............ 13

Figure 8: Pseudo code of the *apply algorithm* for copy/insert delta encoding............ 15

Figure 9: An example of encoding delta files in *replace* algorithm............ 21

Figure 10: Pseudo-code for *replace* algorithm............ 22

Figure 11: Pseudo-code for construction of hash table 23

Figure 12: Pseudo-code of *getFromHashTable* method for hash table 24

Figure 13: The state of hash table for given example 26

Figure 14: The figure of our generated version chain for experimental results 33

Figure 15: The performance of apply algorithm when insert stream is on disk or in memory, and intermediate versions are constructed on disk for package *gawk*............ 36

Figure 16: The performance of apply algorithm when insert stream is on disk or in memory, and intermediate versions are constructed with a buffer in memory for package *gawk* 37

Figure 17: The performance of apply algorithm when insert stream is on disk or in memory, and intermediate versions are constructed on disk for package *chicken*............ 38

Figure 18: The performance of apply algorithm when insert stream is on disk or in memory, and intermediate versions are constructed with a buffer in memory for package *chicken* 39

Figure 19: The comparison of apply algorithm when intermediate versions are constructed on disk fully and apply when they are constructed with a buffer in memory while insert stream is on disk for package *mailman*............ 41

Figure 20: The comparison of apply algorithm when intermediate versions are constructed on disk fully and apply when they are constructed with a buffer in memory while insert stream is in memory for package *mailman* 41

Figure 21: User Time of replace algorithm when binary search or hash data structure is used, and intermediate versions are constructed with a buffer in memory for 5 MB Chains...... 42

Figure 22: User Time of replace algorithm when binary search or hash data structure is used, and intermediate versions are constructed with a buffer in memory for package *nano*..... 42

Figure 23: The system time comparison of apply and replace algorithm when intermediate versions are constructed with a buffer in memory for 10 MB Chains 43

Figure 24: The system time comparison of apply and replace algorithm when intermediate versions are constructed with a buffer in memory for package *metahtml* 44

Figure 25: The system time comparison of apply and replace algorithm when intermediate versions are constructed with a buffer in memory for package *marst*.................................. 44

Figure 26: The total time comparison of apply and replace algorithm when intermediate versions are constructed with a buffer in memory for 10 MB Chains 44

Figure 27: The total time comparison of apply and replace algorithm when intermediate versions are constructed with a buffer in memory for package *mailman*............................ 45

Figure 28: The total time comparison of apply and replace algorithm when intermediate versions are constructed with a buffer in memory for package *gawk* 45

Figure 29: The comparison of apply when intermediate versions are constructed in memory completely and replace algorithm using hash data structure for 1 MB Chains and execution times are calculated by the program itself 46

Figure 30: The comparison of apply when intermediate versions are constructed in memory completely and replace algorithm using hash data structure to delete version 2 from 3 MB Chains and execution times are calculated by the program itself 48

CHAPTER 1

INTRODUCTION

1.1. Motivation

Today, many computer systems store and present users all over the world many files. Such files could be frequently changed and as changes occur, people subscribing to those changes will need to transfer the new file over network so that they have the most recent version. This process can be done more quickly by transferring only the changes (called a *delta*) that were performed on the previous version and subscribers locally applying updates on their local copy to create the most recent version of the file.

Another possible use of *delta* files is saving them so that an earlier version of a file can be recovered if it is needed. This would be the case when software is being developed, and earlier version is found to be better than the current file (such as a bug is discovered). Another reason for maintaining several versions of the same program could have to support multiple customers possibly using different earlier versions. By storing only delta files, we also conserve disk space since otherwise we would have to store multiple versions of the same file, as a whole, thus taking up much more space.

Storing versions of a file like a directory seems to be a good solution at first glance however the disk space occupied by versions is not used efficiently in this way. Another problem occurs when a version is accessed over network, because transmission of a version from one location to another is dependent on the file size and network capacity.

A *delta algorithm* encodes the difference between two versions of a file, *source* and *destination* (or *target*), and it stores the *encoding* in a *delta file*. A delta file includes only the changes between two files and its size is expected to be quite small when the similarity between versions is assumed to be high. It is sufficient to store source and delta file instead of storing two versions as intact files, which provides an efficient storage, especially when number of stored versions increases. When destination file is requested, applying delta file *encoding* to source file constructs the destination one. In a repository, storing one file fully and other versions as deltas is the simplest way to save space on disk (Figure 1). Delta file is also suitable for network transmission. If the client has source file and needs the destination one, it request the delta file instead of destination itself, which reduces the network traffic. Also, a program at client may include bugs and they should be fixed. Patch of the program fixes the bugs and it requires the erroneous version to construct the correct one, therefore deploying delta file instead of version file eliminates unauthorized use of software as well.

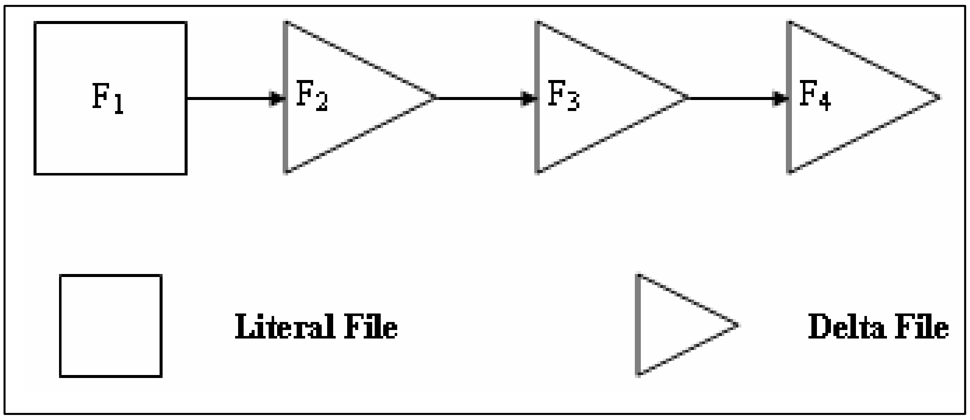


Figure 1: Storing four versions of a file with forward delta technique, version F1 is stored as literal and other versions are stored as delta files in the chain.

The storage of version files with deltas should be handled by more complicated solutions like a manager -*version control system*- (RCS, SCCS, etc). For example; many versions of a file will be produced when a program is under development. A developer may want to be able to access and modify previous versions (such as after discovering a buggy modification, or having to maintain old versions for customers still using them), possibly over a network connection, and to view their contents. There have been many investigations and version control systems produced to minimize the storage size, speed up the retrieve and insert time of a version, provide concurrent access to the same version, merge the versions, handle the branch problem, store the binary files as well as text files, and so on.

1.2. Thesis Goals

This thesis offers a new technique to generate a version from a repository instead of applying deltas to a literal file in sequential order, one delta at a time. *Replace algorithm* first combines deltas between a literal file and its required version to produce a combined delta which can be applied to the literal file to generate the required version. The most important goal of the *replace algorithm* is to reduce I/O operations by eliminating the construction of intermediate versions (which is to be stored on disk temporarily thus causing extra I/O overhead) when generating a required version of a literal file. The algorithm is to read the insert stream of both adjacent delta files before starting delta combination. This operation may be recognized as a disadvantage, however it is proven by experimental studies that taking insert stream into memory has no effect on the total execution time of the algorithm. Delta application is implemented with different memory usages and destination file construction. Apply algorithm can construct required version of a literal file using three different ways of creating intermediate ones. The algorithm can construct the intermediate versions and the final required version on disk completely or use a fixed-size buffer in memory to reduce I/O operations or use the memory completely. The benefit of using a buffer is shown explicitly in experimental studies. Another optional add-on is to take insert stream into memory before delta application.

Replace algorithm searches the beginning offset of each instruction in destination delta file over instructions in the source delta file. There are two ways that replace algorithm handles the search operation; hash data structure and binary search algorithm. Replace algorithm constructs a hash table to find the instruction set in source delta defining the range of the instruction in destination delta or it uses binary search algorithm to find the instruction in already sorted instruction list. The replace algorithm using binary search, puts a better execution time than the one using hash data structure. Because, hash table construction causes extra CPU time although the algorithm searches over narrowed range. Each algorithm is compared with its versions and also a fair comparison is made between both algorithms in Experimental Results.

1.3. Organization of the Thesis

Chapter 2 gives introductory information about delta encoding, delta storage techniques, delta algorithms, delta application, version control systems, and related work. Chapter 3 presents the design and implementation of *Replace Algorithm*, and a reader can find the benefits, properties, pseudo code, and an example of the algorithm. Chapter 4 demonstrates complexity of delta application and delta combination. Chapter 5 presents experimental results for our generated versions and some packages at Gnu Web Site. Chapter 6 concludes the subject and presents future work discussions. Appendix A includes names of versions and their sizes of some real-life packages used in Experimental Results. Appendix B gives instruction statistics of the generated version chains and packages occurred in Appendix A.

CHAPTER 2

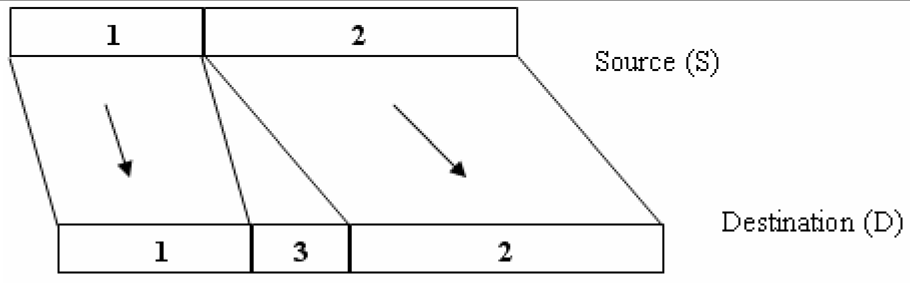
BACKGROUND

A delta algorithm computes the differences between two versions of a file. It takes two files - source (S) and destination (D) files - as input, generates a set of instructions and produces a delta file.

$$\text{Delta Algorithm (S, D)} \rightarrow ?_{s,\ d} \qquad (2.1)$$

Applying instruction set to S produces D, and there is an example in the chapter for copy/insert encoding how a delta file is applied to S.

$$\text{Apply (S, } ?_{s,\ d}) \rightarrow \text{D} \qquad (2.2)$$



(a)

copy	block 1
insert	block 2
copy	block 3

(b)

Figure 2: An example of encoding a delta file using copy/insert delta encoding

In Figure 2 (a) an example is given showing source and destination versions, with identical regions of versions marked as “1” and “2”, while newly inserted segment is marked with “3”. A corresponding encoding is given in Figure 2 (b).

2.1. Delta Encoding

There are two different delta encoding types used commonly; *copy/insert* and *insert/delete*. *Copy/insert* delta encoding has two different instructions; *copy(s, d, l)* and *insert(d, l, B)*. A *copy* instruction copies a block with length *l* from offset *s* in S to offset *d* in D, and an *insert* instruction adds the block *B* with length *l* in delta to offset *d* in D. Delta application of *copy/insert* delta encoding constructs D from an empty file and preserves S. It is essential to preserve S for version control systems. *Insert/delete* delta encoding has two different instructions; *delete(s, l)* and *insert(s, l, B)*. A *delete* instruction deletes *l* bytes from offset *s* in S and an *insert* instruction adds a block *B* with length *l* to offset *s* in S. Delta application of this kind of delta encoding operates each instruction on S and S -not preserved- is transformed to D, which is called in-place reconstruction. *Insert/delete* delta encoding is suitable for patch implementation. Burn, Stockmeyer, and Long [13] address the limited storage capacity and low-bandwidth networks and present algorithms that transform a delta file including copy/insert delta encoding to a delta file that can construct the target version in-place.

Literal file in version control system should be preserved and implementation of delta application of copy/insert delta encoding is straightforward therefore *replace algorithm* is designed to combine deltas consisting copy/insert encoding.

2.2. Encoding Metrics

Each delta algorithm produces an encoding to represent D with respect to S and the encoding is stored in a delta file. The encoding includes instructions and they can be stored in the delta file with different ways. MacDonald [8; 9] separates copy and insert instructions, and Burns [11] stores them in an order with *add*, *copy* and *end* codewords in the delta file. Therefore, delta size is not a suitable metric to compare delta algorithms. Hunt, Vo and Tichy [7] defines a metric in terms of LCS -Longest Common Subsequence-. LCS is the longest common block which appears between two files. However, repeated copy regions are not considered in the metric.

$$\text{difference} = \frac{\text{size(source)} + \text{size(destination)}}{2} - \text{size(LCS)} \qquad (2.3)$$

MacDonald [9] presents a metric *m* for upper bound on the optimal sequence of copy/insert delta encoding. Total delta size in bits is calculated by summation of metric values for each *copy* and *insert* evaluated. *XDelta* [8; 9] uses insert data of deltas as additional source as well as actual source file itself therefore k^d means a source position in source file or in one of insert data. The copy metric can be integrated for one source file and k^d is replaced with s. It is also assumed that a byte includes 8 bits.

$$m(\text{copy } s\ d\ l) = 1 + |\log l| + |\log k^d| + |\log d| \qquad (2.4)$$

$$m(\text{insert } l) = 1 + |\log l| + 8l \qquad (2.5)$$

2.3. Delta Algorithm Concept

The aim of a delta algorithm is to compute a delta encoding for D with respect to S. The delta algorithms vary by finding matches. A dynamic delta algorithm [6] encodes the difference based on LCS and greedy delta algorithm [10; 12] finds the common fragments between two versions.

UNIX "*diff*" command is a well known line-oriented delta algorithm. However, line-oriented algorithms encode whole line as *insert* if a line is changed, and this solution is not the optimum. Besides, line-oriented algorithms are applicable only for text files [15]. Myers [2] introduces a dynamic algorithm which requires $O(nm/w)$ time and computes the edit distance for particularly practical cases. Baker, Manber, and Muth [1] implement a delta algorithm with knowledge of the architecture in binary files, however, the delta algorithm is not suitable for generic solutions. Hunt, Vo, and Tichy [7] introduce a greedy delta algorithm *vdelta* that combines data compression and differencing. *vdelta* uses hash table instead of a suffix tree in Tichy's block-move algorithm [16]. The general greedy algorithm [12] runs in quadratic time, and it accepts the longest found match for searched position as best match, and optimum. Burns [12] proves that the general greedy algorithm produces an optimum encoding. Correcting one-pass algorithm runs in linear time and it produces a delta encoding which is quite comparable to the greedy algorithm's encoding. Therefore, in this thesis the general greedy algorithm and correcting one-pass delta algorithm are implemented and used.

2.4. General Greedy Algorithm and Linear Time Delta Algorithms

Burns [12] proves that general greedy algorithm generates an optimum difference for two versions of a file, however its execution time is quadratic and memory usage is proportional to size of *source*(S) file. He also introduces *one-pass, correcting one-pass* and *correcting 1.5-pass* algorithms that change data structure and search policies with some modifications in the general greedy algorithm. These algorithms run in linear time, improve memory usage utilization and produce good compression in terms of greedy one. MacDonald [9] also defines and uses a delta algorithm (*XDelta*) which is a fast, linear-time and linear-space approximation to the greedy algorithm.

The general greedy algorithm constructs a hash table on *S*, and searches a match for each offset of *Destination* (D) by using the hash table. The aim of hash table is to find candidate match offset from S. Burns [10] selects a *footprint* (Karp-Rabin) function for fixed-length byte streams to construct a hash table on S. The algorithm chooses a value *p*, calculates a footprint value for length p of byte streams in file S at all offsets until size(S) + 1 – p. Karp-Rabin method calculates footprint value of stream at 0th offset with length p. The next footprint values for other offset are calculated with incremental calculation instead of the same manner. If footprint value for an offset is calculated, then footprint value for the next offset can be calculated using previous value with a constant number of operations, and this reduces the creation time of the hash table. The greedy algorithm stores all offsets falling to the same entry with a linked list, and it requires a hash table of 4 times the size of S to be built in memory, assuming footprint value type is integer.

After hash table construction, general greedy algorithm scans the longest match for current search offset I of D. It calculates footprint value for D [I, I + p), and lookups hash table whether exists an entry for the value, or not. If exists, it generates a match for each offset in entries. Then, it chooses the longest match among the matches. Match search policy proceeds forward only. If the length of longest match is greater than the cost of optimum copy instruction, the instruction is concluded as copy and next search offset is set to i + l (length of the longest match), otherwise as insert. It is obvious that if no match exists, current search index i is incremented by 1.

The complexity of the general greedy algorithm is O (size(S)*size(D)), and size of hash table depends on size(S). The algorithm for large version files is not applicable because of quadratic time and memory usage. *One-pass*, *correcting one-pass* and *correcting 1.5-pass* algorithms are modification of general greedy in usage data structure, memory usage and search policy, they run in linear time. Hash table which each algorithm constructs do not has chain and also algorithms differ in usage of hash table. Correcting implies backward match besides forward one. The algorithm corrects the previous encoding with better matches if exists. Correcting can be tail correction, or general correction, or both of them.

The details of the algorithms except *correcting one-pass* algorithm will not be mentioned. *Correcting one-pass* algorithm creates two empty hash tables for S and D, HT_S and HT_D. It defines s_c, and d_c offsets for S and D respectively, and sets them to 0 initially. The algorithm calculates the footprint values of strings from s_c and d_c with length p. It puts the footprint values in HT_S and HT_D respectively. HTs do not have chains, therefore whether there exists an entry for a footprint value, new offset is added to HT. The algorithm does not remember the previous offsets for the corresponding entry. At this point, the algorithm tries to find candidate match. If footprint value calculated for S occurs in HT_D, there exists a candidate match at s_c and an offset at entry for that footprint value in HT_D. If the seeds are identical at offsets, match process starts and the rest of searching candidate match is skipped. If the seeds are not identical, the algorithm looks an entry footprint value calculated for D in HT_S in the same manner. If there does not exist a candidate match, the algorithm continues the process with incrementing both s_c and d_c by 1. Match occurs in both forward and backward directions. If the match overlaps only the non-encoded portion, the range between the end of encoded bytes and the start position of the match is concluded as insert and then match is encoded as a copy instruction. If the match overlaps the encoded and non-encoded portion, it requires tail correction on encoded substring. If the match overlaps only the encoded portion, it requires general correction on encoded substring. Tail or general correction means that previous instruction(s) falling completely into match range will be deleted. The algorithm stops when s_c + p > size (S) and d_c + p > size (D) and the rest of D is concluded as insert if there exists non-encoded bytes at the end.

As seen, the algorithm stores one offset for each footprint in hash table. This limitation reduces the performance of delta algorithm, however it improves the execution time drastically and corrections eliminate the bad encoding. As a result, linear algorithms still yield comparable solution according to general greedy algorithm especially for large version files.

XDelta selects a *fingerprint* (adler32) function for fixed-length byte streams as a hashing function. The algorithm selects a value s – a small power of 2 –, calculates a fingerprint value for length s of byte streams in file S at all offsets divisible by s. The algorithm includes one hash table keeping offsets. It also constructs an array corresponding fingerprint value for each offset to detect collision easily. 4 bytes are enough for each offset and fingerprint value, therefore the cost of data structure is 2 * 4 * size (S) / s. If s is 2^4, then the algorithm requires half of size of file S as memory space. It increments current search offset by 1 whether a match exists for current offset, or not. It takes a set of S instead of a single source file.

It is obvious that collision occurs in the hash table. Each byte stream with fixed-length is represented with an integer value, and hash method – footprint or fingerprint - can yield the same hash value for two different byte streams. Hash table has a mod value to insert an offset, and also this yields collision. *XDelta* solves the last collision problem by storing one offset for each entry and constructing an array to keep fingerprint value itself, however storing one offset for one entry affects the performance of the algorithm. In both hash table solutions, found offset in S for an offset in D is a candidate match, and it requires byte comparison between streams.

2.5. Delta Storage Techniques

Forward delta (FD) is the basic delta technique for storage (Figure 3). FD stores the first version as literal and subsequent versions as delta files in order. Deltas are calculated between adjacent versions; such as between first and second versions, second and third one, and so on. There are two main disadvantages of the storage technique. The first one is that retrieving *i*th version in a chain requires (*i*-1) times delta application and each delta is applied to literal file until reaching *i*th one. When F_3 is required, $?_{1,2}$ is applied to F_1, and F_2 is generated, then $?_{2,3}$ is applied to generated F_2, and F_3 is generated. The second one is that the storage is not suitable to insert a new version easily in a chain. The new delta is calculated between the most recent version and the new one, and each insert operation requires the generation of the most recent one. The triangle in Figure 3 implies the forward delta and square stands for the literal file.

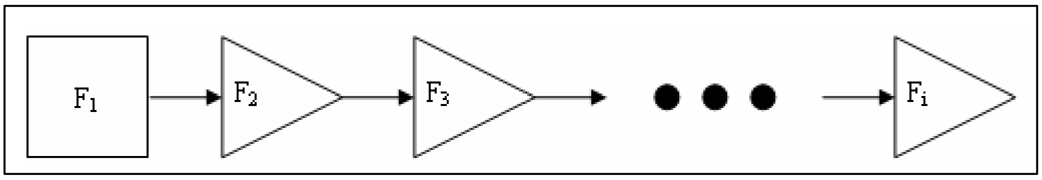


Figure 3: The storage mechanism of *forward delta* technique

Repository including many versions can be divided into *clusters*, and each cluster includes one literal file and deltas. It satisfies an upper bound to access a version in repository. Figure 4 shows a repository that has 2 clusters.

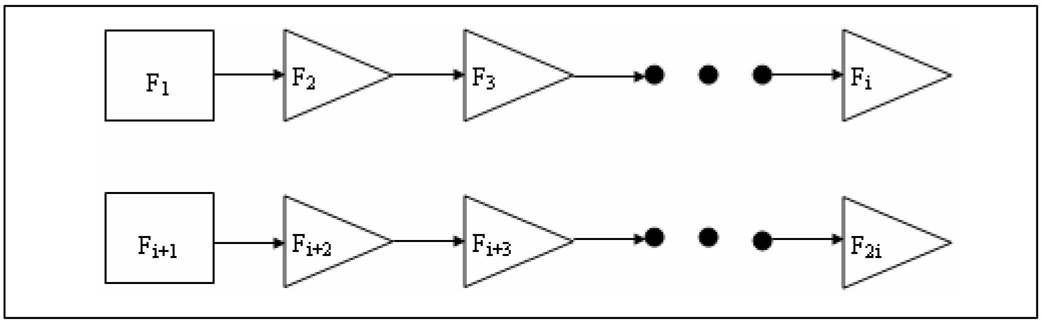


Figure 4: Clustered *forward delta* technique

Jumping delta (JD) -a different storage technique- improves insert and retrieve operations of a version in a chain considering FD. It stores the first version as a literal like FD. However when a new version is inserted, delta file is calculated between literal and newly introduced one instead of computing adjacent versions; such as between first and second versions, first and third ones, and so on. Figure 5 shows the storage mechanism of JD. The main benefit of the jump storage technique is to retrieve a version at most one delta application by using literal and the related delta. However, the similarity between literal and newly introduced versions decreases while the chain grows, therefore the storage consumes much disk space.

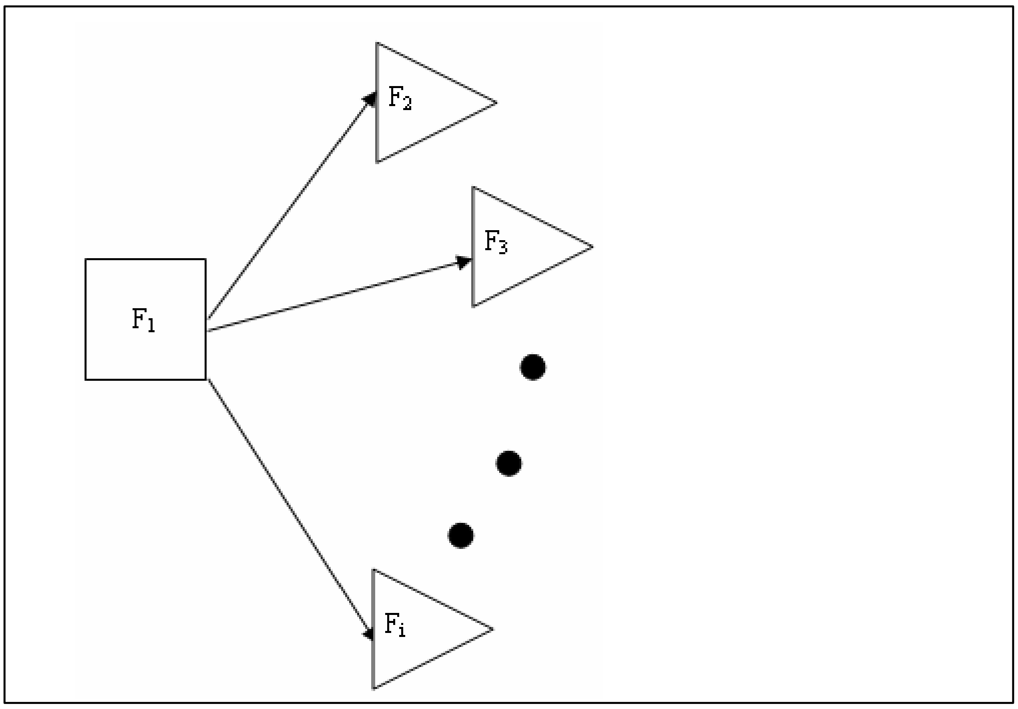


Figure 5: The storage mechanism of *jumping delta* technique

Burns and Long [11] improves storage and retrieve time of *AdStar Distributed Storage Manager (ADSM)* using jumping delta technique. They define a compressibility parameter between consecutive versions, and establish a worst-case formula choosing a low and high value for the parameter. They give an experimental result to compare the storage lost between the jumping and forward delta techniques and to determine the optimum number of versions which a cluster should include. The system includes server/client architecture where server stores the repository and client stores a copy of literal file in repository at the server side. When a version is requested from client, server sends the delta to client over network. Insertion is also easy because the client produces the delta between the new version and the literal one, and the server stores it in repository. When the cluster reaches the optimum number of versions in the repository, the client sends the new version itself.

Reverse delta (RD) is the most popular storage technique that is introduced by Tichy [15]. It stores the last version as literal and previous versions as deltas in reverse order. The most recent versions are accessed more frequently than older ones, and they can be constructed by applying several delta files to literal file. The same problem of accessing the recent versions in FD also occurs in accessing the older versions in RD. The triangle in Figure 6 specifies the reverse delta.

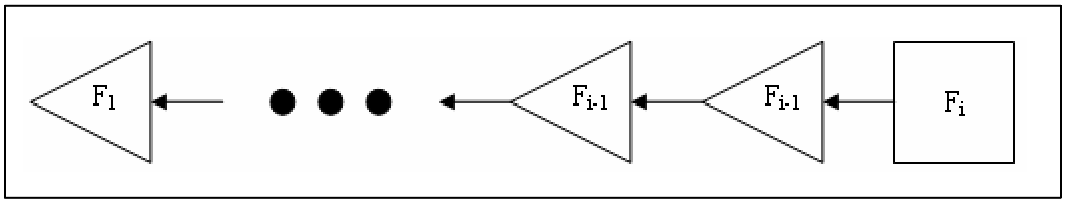


Figure 6: The storage mechanism of *reverse delta* technique

Eventually changes occur on previous versions because of some reasons; such as a request to fix a bug in an intermediate version used by a customer [15]. New form of the previous version is also to be stored and it cannot be introduced like a new version in result of changing some parts of the latest version. *Branch* handles the development of previous version by creating a new chain connected to it. Figure 7 shows how RCS stores branch versions.

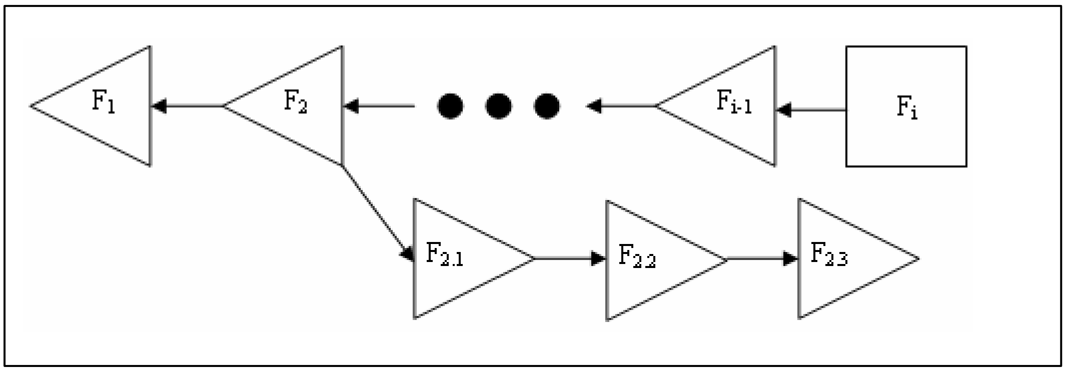


Figure 7: The storage mechanism of branch in RCS

SCCS (Source Code Control System) [14], one of the oldest tools, uses FD technique. RCS (Revision Control System) [15] stores the most recent version on the trunk as literal and uses RD technique to store the previous versions on the trunk. It handles branch using FD technique. XDFS [8] - The XDelta File System - offers two storage techniques; XDFS-f and XDFS-r. Although suffix of XDFS-f implies forward delta, it uses JD technique and XDFS-r uses RD.

RCS uses an ancestral tree to stores versions preserving the hierarchy in a repository. RCS expects a revision number for a new version. If it is not specified, then RCS tries to determine the number. XDFS stores versions of a repository in a single trunk, and gives a number sequentially to each inserted version to identify them. XDFS has a different approach to add a new version in a repository to handle the branch problem and reduce the disk size consumed by the repository. When a new version is introduced, XDFS concatenates the new one and inserts data of current deltas in the cluster, and computes the delta between the concatenated file and the literal one. Therefore, when two or more branches become dissimilar, delta computation can conclude a copy instruction from taking the source a delta file instead of storing the duplicate change in the new delta.

SCCS and RCS use UNIX *diff* command to compute the delta between adjacent versions, therefore they are applicable for text-oriented revisions. XDFS is applicable for binary files as well as text ones and it uses XDelta as delta algorithm.

2.6. Delta Application

Apply Algorithm is a simple implementation of delta application, and applies the delta files to literal version files consecutively until the required version is generated. Apply algorithm (Figure 8) applies each instruction in a delta to the source file and generates the destination one.

If 4th version is required in a forward chain, firstly *apply algorithm* applies $?_{1,\ 2}$ to literal F_1, and it generates 2nd version. Secondly, it applies $?_{2,\ 3}$ to 2nd one, and it generates 3rd one. Finally, it applies $?_{3,\ 4}$ to 3rd one, and it generates required 4th version. There is an example to make clear how the algorithm runs.

```
Apply (src, dst, ? )
1.  for i ← 0 to size[?] – 1 do
2.      if ?[i].type == "COPY"
3.          then copy (src, ?[i].frompos, dst, ?[i].topos, ?[i].length)
4.      else copy(?[i].buffer, 0, dst, ?[i].topos, ?[i].buffer.length)
```

Figure 8: Pseudo code of the *apply algorithm* for copy/insert delta encoding

Version Files

```
      0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29
1st:  a b c d e f g h i j k  l  m  n  1  2  3  4  5  6  7  8  9  4  3
```

```
      0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29
2nd:  a b c d e f g h i j k  l  m  b  0  1  2  3  4  5  6  7  8  9  1  4  3  5
```

Delta Files

? $_{1, 2}$

Index	type	frompos	topos	length	buffer
1.	copy	0	0	13	null
2.	insert	0	13	2	"b0"
3.	copy	14	15	9	null
4.	insert	0	24	4	"1435"

The algorithm applies each instruction in ? $_{1,\ 2}$ sequentially. 1st instruction is a copy instruction, and the algorithm copies 13 bytes from 0th byte position of 1st version to 0th position of generated 2nd version. Now, new file looks like below.

```
      0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29
2nd:  a b c d e f g h i j k  l  m
```

Then, it applies 2nd instruction which is insert one. It inserts “b0” byte sequence to 13th position of the file, and it becomes

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
2nd:	*a*	*b*	*c*	*d*	*e*	*f*	*g*	*h*	*i*	*j*	*k*	*l*	*m*	*b*	*0*															

After instructions are applied, new 2nd version is generated.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
2nd:	*a*	*b*	*c*	*d*	*e*	*f*	*g*	*h*	*i*	*j*	*k*	*l*	*m*	*b*	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*1*	*4*	*3*	*5*		

The complexity of the delta application is O (size (D)), because size (D) bytes are copied from one location to another.

2.7. Related Work

SCCS [14] stores versions of a repository in a single file and it uses *interleaved* deltas. The file is divided into fragments and each fragment includes a set of line(s) and a header. The header consists of versions where the fragment exists or not. SCCS traverses the fragments sequentially and combines the fragments together which belong to the required version. The interleaved storage provides an efficient reconstruction because reconstruction needs to traverse whole file to retrieve any versions in a repository. However, the performance of reconstruction reduces while the repository tends to grow and number of lines increases.

Tichy [15] introduces an algorithm to eliminate unnecessary copies for unchanged lines while delta application in RCS. It constructs a piece table - one-dimensional array -, which includes the address of each line in literal instead of line itself. It applies the delta to the piece table by deleting unnecessary entries from the piece table and inserting new entries required in next version. Adding a new entry into the piece table requires shifting the entries below further down however it would be wise to select a more sophisticated data structure instead of a one-dimensional array. The resultant piece table includes the addresses of lines in next version, and version is constructed by gathering the lines. He states that RCS reconstructs a version faster than SCCS if the number of deltas applied is not greater than or equal to 10. Deltas are line-oriented in RCS; therefore the solution is not applicable for binary-files. Hunt, Yo, and Tichy [7] states "A simple technique is to map the binary code into text and then applying *diff*. While this works reliably and is widely used in practice, the deltas produced are typically larger than the originals! ".

MacDonald [8; 9] mentions a new technique *reconstruct* operation to retrieve a version instead of using delta application. The *reconstruct* creates a balance interval tree to map byte ranges in required version to literal and insert data of deltas. It processes the each delta, and inserts the ranges in a delta to interval tree. MacDonald states that reconstruction algorithm runs in *O (n*z log z)* time, where *n* is the number of delta applied and *z* is the maximum number of ranges in a delta. To construct the version, ranges in the resultant tree are copied from literal or insert data of deltas and it requires *O (size (D))* steps. He states "Reconstruct can be considerably more efficient than simply applying each delta in sequence".

Subversion [4] project is a replacement of CVS, and it completes the lack of CVS besides offering the most of its features. *Subversion* uses *vdelta* delta algorithm to compute the delta between two versions therefore deltas also include target copies as well as source copies and new data -inserts-. Branko [5] designed an algorithm - delta composition - combines the deltas including target copies for retrieval function of *Subversion*. He establishes a relation T = AB(S) = B (A(S)); where A and B are deltas, S is source file, and T is target file. The algorithm uses a splay tree to map the ranges like MacDonald's balanced interval tree. The algorithm has to change each target copy of A with corresponding source copies and new data, because A should not depend on intermediate version between S and T before delta composition. The target copies and new data in B are not related to intermediate version; therefore the algorithm adds them to the resultant delta directly. However, it adds the equivalent instruction(s) in A defining the range of source copies in B. Meanwhile, the algorithm does opposite transformation in resultant delta to reduce cache trashing by increasing locality of reference. It puts the equivalent target copy when encouraged a source copy in B previously defined in T. This process keeps the history of source and target of each copy from A to do opposite transformation, and reduces the number of instructions in resultant delta.

Zeller [3] introduces a new technique for fast reconstruction problem for block -copy/insert delta encoding- algorithms in his thesis. The algorithm converts each copy block in target delta with blocks in source delta and it uses binary search algorithm to find start offset of first block in source delta covering the copy one. *Replace* algorithm is similar to Zeller's, however *replace* is designed and implemented without knowledge of Zeller's thesis. The thesis compares the algorithm against delta application. He uses versions of his thesis text as test data, and he constructs a reverse and a forward chain -70 deltas- generated from CVS. The experimental result shows that the delta combiner for reverse deltas yields a better execution time than delta application but it is not valid for forward deltas. He states, by further investigation, the resultant forward delta mainly includes very small fragments -instructions- because of truncation. The resultant combined delta of reverse chain because of being less fragmented, shows better performance in terms of execution time. He finally offers to divide the deltas into groups and combine each delta group separately due to the worst performance of small fragmentations. However, intermediate versions at intersection of adjacent groups are generated.

2.8. Implementation Notes

Apply and replace algorithms are implemented with different memory usage and data structure type. Two different delta algorithms are implemented, one of which runs like the general greedy algorithm, constructs one hash table and the other correcting one-pass constructs two hash tables.

2.8.1. Implementation Notes for the Delta Algorithm

The delta files, used by apply algorithm when destination and intermediate versions are constructed in memory completely and replace algorithm using that apply algorithm and constructing hash table, are produced by a delta algorithm that is the combination of the general greedy algorithm and XDelta. The algorithm constructs a hash table like XDelta does, however it does not store the fingerprint value itself in an array. All offsets corresponding to the same entry are stored with a chain. Match occurs in both directions, and each match is added to delta file with the rules of tail correction. If a match is found, next search offset is set to current search offset plus the length of the match. Although it almost produces optimum result, its execution time becomes worst like the general greedy algorithm for large version files. Then, correcting one-pass delta algorithm is implemented and it generates the delta files which are used by apply algorithm when destination and intermediate versions are constructed on disk or with a buffer in memory and replace algorithm that is optional to use binary search or hash data structure.

2.8.2. Implementation Notes for Apply Algorithm

The pseudo code in Figure 8 is used as underlying code in the apply algorithm for the thesis, however it is implemented with some improvements in view of memory usage and construction of destination file. The code is designed to calculate IO and CPU times separately. Apply algorithm constructs the (intermediate) versions with three different options; on disk completely, with a fixed-size buffer in memory, or in memory completely. If versions are constructed in memory completely, S and insert stream in $?_{S,\ d}$ are read into memory fully, and a memory block with length D is reserved for the construction of D before delta application. Then, constructio n of D occurs in memory. If more than one delta application is necessary, literal file is firstly read into memory and then each D becomes S of next delta application. While delta application, intermediate versions (Ds) are not stored except the last one because it is the required version. If versions are constructed on disk completely or in memory with a fixed-size memory buffer, S is not taken into memory. The size of buffer is not enough to construct the versions wholly in memory and buffer is flushed to disk when it becomes full. These two versions of apply are optional to take the insert stream into memory or not.

2.8.3. Implementation Notes for Replace Algorithm

Replace algorithm search the instructions in the source delta file using binary search algorithm and constructing a hash table, and the performance of the algorithm for each search option can be seen in the experimental results.

CHAPTER 3

REPLACE ALGORITHM

3.1. Replace Algorithm

When a version is requested in a chain and its generation necessitates applying more than one delta file to literal file, *replace algorithm* can combine the intermediate delta files between literal and required version as a single delta in the run time. This solution prevents unnecessary IO operations which delta application does, because intermediate versions are not generated and are not stored on disk temporarily in replace algorithm. The algorithm is applicable for copy/insert delta encoding therefore it is applicable for binary files.

Figure 9 shows a simple case among three versions to make clear how replace algorithm works. The two different blocks -block 1 and 2- occur in versions V_k, V_{k+1} and V_{k+2}. The delta algorithm concludes two copy instructions for these blocks and one insert instruction for block 3 in $?_{k,\ k+1}$. These three blocks occur in continuous sequence between V_{k+1} and V_{k+2} and it is concluded as a single copy instruction in $?_{k+1,\ k+2}$. If delta application generates V_{k+2}, then these three blocks are copied from one location to another location for two times. Replace algorithm converts the each instruction in $?_{k+1,\ k+2}$ with the corresponding instruction set in $?_{k,\ k+1}$. The copy instruction defining the continuous sequence between V_{k+1} and V_{k+2} can be converted using copy, insert, and copy instructions set in $?_{k,\ k+1}$. Because these three blocks defines a byte range in V_{k+1} where the single copy instruction in $?_{k+1,\ k+2}$ uses the same byte range as source to define a different byte range in V_{k+2}. However, the byte range defined by a copy instruction in $?_{k+1,\ k+2}$ can be a subset of the byte range defined by instruction(s) in $?_{k,\ k+1}$. This problem can be handled with re-calculation of edge instruction(s), whose length shortened or source position changed. Insert instructions in ? $_{k+1,\ k+2}$ do not require any calculations, and become stable except changing their destination position. The resulting delta is ? $_{k,\ k+2}$ and it is constructed from a new delta. Now, V_{k+2} can be constructed with applying ? $_{k,\ k+2}$ to V_k. The algorithm generates the combined delta in run time and does not store it on disk.

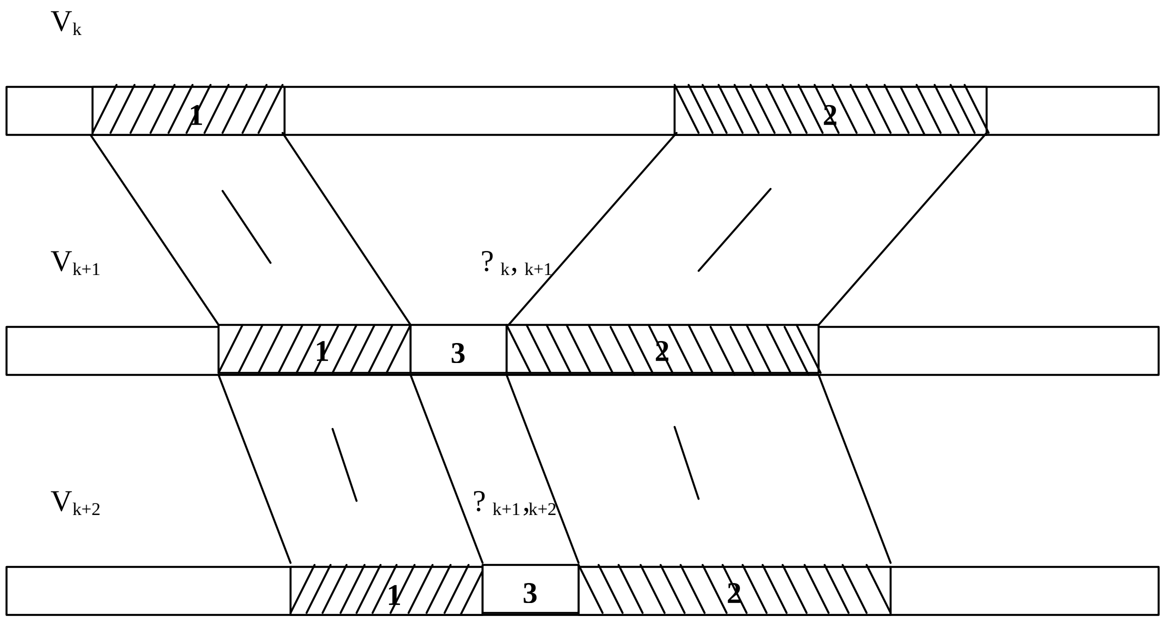


Figure 9: An example of encoding delta files in *replace* algorithm

Figure 10 includes the pseudo-code of replace algorithm in detail. The algorithm takes two consecutive delta files - $?_{k,\ k+1}$ and $?_{k+1,\ k+2}$ - as input and produces the combined delta - $?_{k,\ k+2}$ - as output. It yields the same result using binary search algorithm on source delta file which is already sorted in destination position or creating a hash table on source delta file and using binary search on narrow range to find the index of instruction in $?_{k,\ k+1}$ which defines for current copy instruction in $?_{k+1,\ k+2}$.

```
Replace ($?_{k,k+1}$, $?_{k+1,k+2}$, , mode)
1.Create an empty $?_{k,k+2}$ list
2.if mode == HASH_TABLE_MODE
3.  then hashTable ← createHashTable($?_{k,k+1}$)

4.for i ← 0 to size[ $?_{k+1,k+2}$ ] – 1 do
5.       if $?_{k+1,k+2}$ [ i ].type == "INSERT"
6.     then add( $?_{k,k+2}$, CInstruction($?_{k+1,k+2}$ [ i ], topos ) )
7.       continue
8.       if mode == BINARY_SEARCH_MODE
9.     then index ← binarySearch($?_{k,k+1}$, 0, size[$?_{k,k+1}$], $?_{k+1,k+2}$ [ i ].frompos)
10.        else index ← getFromHashTable( hashTable, $?_{k+1,k+2}$ [ i ].frompos)

11.found ← $?_{k,k+1}$ [ index ]
12.diff   ← $?_{k+1,k+2}$ [ i ].frompos – found.topos
13.toPos ← $?_{k+1,k+2}$ [ i ].topos
14.clone  ← CInstruction(found, topos, diff)
15.index  ← index + 1
16.length ← $?_{k+1,k+2}$ [ i ].length

17.while length > 0 do
18.     add( $?_{k,k+2}$, clone)
19.     length ← length – clone.length
20.     if  length < 0
21.        then shortened(clone, -1*length)
22.               break
23.     if  length == 0 OR index >= size[$?_{k,k+1}$ ]
24.        then break
25.     toPos ← toPos + clone.length
26.     clone ← CInstruction($?_{k,k+1}$[ index ] , topos)
27.     index ← index + 1
28.return $?_{k,k+2}$
```

Figure 10: Pseudo-code for *replace* algorithm

The algorithm produces a *hash table* to address the instructions in $?_{k,\ k+1}$ (line 3) if HASH_TABLE_MODE is selected. *for loop* processes each instruction of $?_{k+1,\ k+2}$ (line 4). If the type of current instruction is *insert* one, instruction is inserted into $?_{k,\ k+2}$ without any calculations (line 6). Otherwise, it is *copy*, and the function *getFromHashTable* or *binarySearch* finds which instruction in the *hash table* defines the *from position* of the current instruction. Then, it returns the index of the instruction in $?_{k,\ k+1}$ (line 9 or 10). The statement (line 14) clones the found instruction and truncates unnecessary byte(s) from beginning of it. The difference between the *source position* of the searched instruction and *destination position* of found instruction is the length of unnecessary bytes. The current copy instruction can be defined a subset of an instruction, an instruction, or a group of consecutive instructions in $?_{k,\ k+1}$. In the similar way, the last instruction may have unnecessary byte(s) and are eliminated with cutting byte(s) from the end of instruction by using method *shortened* (line 21). *While loop* replaces the current copy instruction with above three possib le ways in the statements until *length* is equal to zero (line 17-27).

```
HashTable createHashTable(List ? )
1.  Create an empty hashList
2.  increment ← filesize / size[? ]
        /* filesize is the size of D for current ?, and it can be calculated with
        sum of topos and length of last instruction in the list */

3.  for i ← 0 to size [? ] – 1 do
4.      length ← ? [i].length
5.      pos ← ? [i].topos + ? [i].length
6.      idx ← ? [i].topos / increment
7.  last ← idx * increment

8.      while pos > last do
9.          innerList ← hashList [idx]
10.         if innerList == null
11.             then innerList ← new List
12.               hashList [idx] ← innerList

13.         idx ← idx + 1
14.         add( innerList, i)
15.         last ← last + increment
16. return hashList
```

Figure 11: Pseudo-code for construction of hash table

Before the example to make clear how the algorithm works, how the *hash table* to be constructed (Figure 11), how *getFromHashTable* (Figure 12) and *shortened* method work are described with pseudo code. *createHashTable* constructs a two dimensional array as a *hash table* suitable for *replace* procedure. The first dimension of the *hash table* defines ranges from 0 to *increment* – 1 as first index, from *increment* to 2* *increment* – 1 as second index, from 2* *increment* to 3* *increment* – 1 as third index, and go on. Each index of the *hash table* keeps an array to store instructions falling into the range. *getFromHashTable* method takes a *from position* as an input, and finds which instruction defines *from position* in $?_{k,\ k+1}$. First of all, the method finds inner array, calculates begin and end index of it and then calls *binarySearch*.

```
getFromHashTable(int frompos)
1. innerList ← hashList [frompos / increment]
2. beginIndex ← innerList [0]
3. endIndex ← innerList [0] + size[innerList] - 1

4. return binarySearch(? k, k+1, beginIndex, endIndex, frompos)
```

Figure 12: Pseudo-code of *getFromHashTable* method for hash table

3.2. An Example to Make Clear How Replace Algorithm works

Version Files

```
       0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29
1st:   a b c d e f g h i j k  l  m  n  1  2  3  4  5  6  7  8  9  4  3

       0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29
2nd:   a b c d e f g h i j k  l  m  b  0  1  2  3  4  5  6  7  8  9  1  4  3  5

       0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29
3rd:   z a b c d e f g h i j  k  l  m  b  0  0  2  3  4  5  6  7  8  9  1  4  c  3  5
```

Delta Files

$?_{1,2}$

Index	type	frompos	topos	length	buffer
1.	copy	0	0	13	null
2.	insert	0	13	2	“b0”
3.	copy	14	15	9	null
4.	insert	0	24	4	“1435”

? $_{2,3}$

Index	type	frompos	topos	length	buffer
1.	insert	0	0	1	“z”
2.	copy	0	1	15	null
3.	insert	0	16	1	“0”
4.	copy	16	17	10	null
5.	insert	0	27	3	“c35”

There are three version files and their delta files are generated with general greedy delta algorithm. If ?$_{1, 2}$ and ?$_{2, 3}$ delta files are passed to replace algorithm respectively and search mode is HASH_TABLE_MODE, then *hash table* are constructed on ?$_{1, 2}$. If BINARY_SEARCH_MODE, there is no need to construct any data structures. Finally, delta ?$_{1,3}$ is produced as an output. The example below is prepared for replace algorithm using hash data structure because binary search solution is obvious.

When *createHashTable* runs for delta ?$_{1, 2}$, it yields the *hash table* in Figure 13. The *file size* of 2nd version is calculated by adding *to position* and *length* of the last instruction of ?$_{1, 2}$, and it is 28. The number of instruction in ?$_{1, 2}$ is 4. According the 4th line, *increment* value is set to 7. The first dimension of *hash table* addresses the each byte of 2nd version. 0th index addresses the bytes between 0 and 6, 1st index addresses the bytes between 7 and 13, and so on. 2nd instruction defines 13th and 14th byte positions in 2nd version. 13th position is defined by 1st index and 14th position is defined by 2nd index, therefore two inner lists keep the index of that instruction.

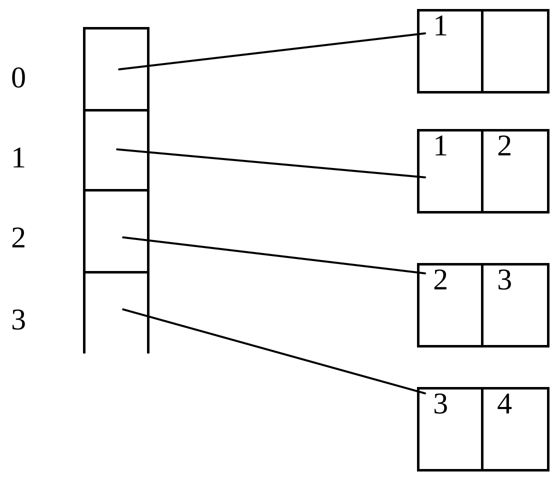


Figure 13: The state of hash table for given example

The algorithm processes the each instruction in $?_{2, 3}$ sequentially.

1st instruction in $?_{2, 3}$ is “insert 0, 0, 1, z”, and it is added into $?_{1, 3}$ without any calculations.

$?_{1, 3}$ *(by replace algorithm)*

1. insert 0, 0, 1, “z”

2nd instruction in $?_{2, 3}$ is “copy 0, 1, 15”, and it copies 15 bytes from 0th position of 2nd version to 1st position of 3rd version. *from position* of the instruction is 0. *Get* method called at line 8 of *replace* method finds the index of instruction where *from position* occurs. 0th position falls into 0th index in *hash table*. The method finds the index of instruction in inner list with sequantial search. The found instruction in $?_{1, 2}$ is “copy 0, 0, 13”. *From position* of searched instruction and *to position* of found instruction are equal, and the *length* of search instruction is greater than the *length* of found instruction. Therefore, there is no need any cut at beginning or end of found instruction. The algorithm clones the found instruction, and changes the *to position* with the *to position* of searched instruction. New cloned instruction is added to $?_{1, 3}$.

?$_{1,3}$ *(by replace algorithm)*

```
1.  insert  0,  0,  1,  “z”
2.  copy   0,  1, 13, null
```

The length of 2nd instruction in ?$_{2,3}$ is 15, and first 13 bytes is replaced with new copy instruction. 15-13=2 bytes are left. The next (3rd) instruction in ?$_{1,2}$ is an insert, and length of it is 2. It is added into ?$_{1,3}$.

?$_{1,3}$ *(by replace algorithm)*

```
1.  insert  0,  0,  1,  “z”
2.  copy   0,  1, 13, null
3.  insert  0, 14, 2, “b0”
```

The searched instruction is replaced with founded instructions completely. The next instruction (3rd) instruction in ?$_{2,3}$ is an insert, and it is added into ?$_{1,3}$ without any calculations.

?$_{1,3}$ *(by replace algorithm)*

```
1.  insert  0,  0,  1,  “z”
3.  copy   0,  1, 13, null
4.  insert  0, 14, 2, “b0”
5.  insert  0, 16, 1, “0”
```

4th instruction in ?$_{2,3}$ is “copy 16, 17, 10”, and *from position* 16 is defined by 3rd instruction “copy 14, 15, 9” in ?$_{1,2}$. However, 3rd instruction defines the bytes block between 15 th and 23 th. The clone of the found instruction becomes “copy 15, 17, 8”. Because the first byte is unnecessary. *From Position* and *length* of clone instruction is incremented by one, *to position* is set to searched one. It is added into ?$_{1,3}$.

?$_{1,3}$ *(by replace algorithm)*

```
1.  insert  0,    0,    1,  “z”
2.  copy   0,    1,  13,  null
3.  insert  0,  14,    2,  “b0”
4.  insert  0,  16,    1,  “0”
5.  copy  15, 17,    8,  null
```

Then, there are two bytes left to replace with instruction(s). The first two bytes of 4th instruction in $?_{1,2}$ is cloned as a new instruction, and it is added into $?_{1,3}$.

$?_{1,3}$ *(by replace algorithm)*

1.	insert	0,	0,	1,	“z”
2.	copy	0,	1,	13,	null
3.	insert	0,	14,	2,	“b0”
4.	insert	0,	16,	1,	“0”
5.	copy	15,	17,	8,	null
6.	insert	0,	25,	2,	“14”

The type of the last instruction $?_{2,3}$ in is insert, it is added into $?_{1,3}$ and combined delta file $?_{1,3}$ is generated.

$?_{1,3}$ *(by replace algorithm)*

1.	insert	0,	0,	1,	“z”
2.	copy	0,	1,	13,	null
3.	insert	0,	14,	2,	“b0”
4.	insert	0,	16,	1,	“0”
5.	copy	15,	17,	8,	null
6.	insert	0,	25,	2,	“14”
7.	insert	0,	27,	3,	“c35”

The delta file below is generated by *greedy delta algorithm.*

$?_{1,3}$ *(by greedy delta algorithm)*

Index	type	frompos	topos	length	buffer
1.	insert	0	0	1	“z”
2.	copy	0	1	13	null
3.	insert	0	14	3	“b00”
4.	copy	15	17	8	null
5.	insert	0	25	5	“14c35”

CHAPTER 4

COMPLEXITY ANALYSIS

Let V_1, V_2, V_3, V_4 be four versions of a file and $?_{1,2}$, $?_{2,3}$, $?_{3,4}$ be delta files respectively. The chain stores V_1 as literal, and other versions as delta files. *i* is the sum of lengths of each insert, and *c* is the sum of lengths of each copy in $?_{2,3}$.

4.1. Retrieve Operation

Delta application and delta combination can be compared when more than one delta application is required. Generation of V_1 and V_2 are simple. V_1 is literal version and a copy of it is created. If V_2 is required, it requires $?_{1,2}$ to be applied to V_1.

4.1.1. Delta Application

It applies $?_{1,2}$ to V_1, and generates V_2. Then, it applies $?_{2,3}$ to V_2, and generates the desired V_3. The complexity of the algorithm is size (V_2) + size (V_3) bytes are read from one location and written to another.

$$O\ (\text{size}\ (V_2)) + O\ (\text{size}\ (V_3)) \tag{4.1}$$

4.1.2. Replace Algorithm

Replace Algorithm replaces instructions in $?_{2,3}$ by using instructions in $?_{1,2}$. Insert instructions are added into $?_{1,3}$ without any calculations, and each insert instruction takes O(1) time. Copy instructions are replaced with a subset of a single or a group of instructions defining its source range in $?_{1,\ 2}$. Then, the cost of replacing each copy instruction becomes vital in the algorithm.

Instructions in $?_{1,2}$ are in ascending order releative to *To Position*. Therefore, instructions are already sorted, and then complexity of finding an instruction is O ($\log_2 n$) over sorted instruction list using binary search algorithm, and n stands for the number of instructions in $?_{1,2}$. The complexity of the algorithm thus becomes

$$O\ (i + c * \log_2 n) \tag{4.2}$$

Lemma *The algorithm is bounded by:*

$$O\ (\text{size}\ (?_{2,\ 3})) <= O\ (i + c * \log_2 n) <= O\ (\text{size}\ (?_{2,\ 3}) * \log_2 n) \tag{4.3}$$

Proof If all instructions are insert in $?_{2,3}$, then lower bound becomes O (size ($?_{2,3}$)) because instructions are added into $?_{1,3}$ without any calculations. If all instructions are copy, then upper bound becomes O (size ($?_{2,\ 3}$) * $\log_2 n$). If the cost of delta application is added, the complexity finally becomes

$$O\ (i + c * \log_2 n) + O\ (\text{size}\ (V_3)) \tag{4.4}$$

By the way, hash table data structure can also be used besides binary search algorithm to find the *From Position* of each copy instruction in $?_{2,3}$. Entries in the HT define a consecutive fixed-size range and keep the index of instructions defining the corresponding byte range. The range is calculated by r = size (V_2) / size ($?_{1,2}$). r is the average length of an instruction. If each instruction had the same and equal size, then r would be 1. Therefore on average, it can be concluded that a few instructions fall into the range, assuming that the number of instructions in a range is x. O (size (V_2)) stands for construction of HT.

$$O\ (i + c * \log_2 x) + O\ (\text{size}\ (V_2)) \tag{4.5}$$

If the cost of delta application of generated delta is added, the complexity becomes

$$O\ (i + c * \log_2 x) + O\ (\text{size}\ (V_2)) + O\ (\text{size}\ (V_3)) \tag{4.6}$$

If more than 2 delta files are combined, HT construction except the first one can be done while creating combined delta. Transformed instructions can be put into hash table, while they are inserted in combined delta list. However, binary search algorithm always shows better performance than HT data structure in experimental results. The reader can see the performance results in Chapter Experimental Results.

When there is a memory usage limitation or working on large version files; such as S cannot be read into memory at once, D file cannot be constructed completely in memory without disk I/O, replace algorithm can be considerably efficient than delta application. Delta application has to write generated intermediate versions to disk, and read them as S from disk for the next delta application.

4.2. Delete Operation

When a version at the edges of a chain is deleted, the solution is clear. The version can be a literal one or the last delta in the chain. If it is the last delta, it is deleted from disk. If it is literal, then the consecutive delta file is applied to literal one and old literal is deleted from disk. Deletion of intermediate version in the chain requires more operations.

4.2.1. Delta Application

The deletion of V_3 requires generation of V_2 and V_4, and computation of $?_{2,4}$ by using them. This application necessitates 3 time delta applications, and 1 time delta computation.

$$O\ (\text{size}\ (V_2)) + O\ (\text{size}\ (V_3)) + O\ (\text{size}\ (V_4)) + DA\ (\text{size}\ (V_2),\ \text{size}\ (V_4)) \qquad (4.7)$$

DA is the delta application. If the delta algorithm runs in quadratic time, it becomes dominant in the complexity. Generated intermediate versions besides $?_{2,3}$ and $?_{3,4}$ are deleted from the disk.

4.2.2. Replace Algorithm

The replace algorithm takes delta files $?_{2,3}$, $?_{3,4}$ as input, and produces the $?_{2,4}$, then $?_{2,3}$ and $?_{3,4}$ are deleted from the disk. The deletion of delta files is the same with delta application except intermediate versions. The produced delta file is not optimum, however size of combined delta file can be quite efficient when considering the execution time of the algorithm or working with large version files.

CHAPTER 5

EXPERIMENTAL RESULTS

A real database table is used to produce version chains with variable size and percentage in terms of file size for experimental results. Each version chain (Figure 14) has 5 version files, and there is a difference ratio between two adjacent versions in terms of size. Chain in the figure is used as a standard in our experimental work. The file size of first version (literal) in a chain approximately can be 50 KB, 100 KB, 300 KB, 500 KB, 1 MB, 3 MB, 5 MB or 10 MB. The file size difference ratio for each chain can be 1, 3, 5, 10, 20, 30, or 50 %. For example; the file size of first version for a chain is 1 MB, and the difference ratio is 10. Then, file sizes in chain are 1 MB, 1.1 MB, 1.21 MB, 1.331 MB, and 1.4641 MB respectively. There are 8 different file sizes and 7 difference ratio options; therefore 56 version chains are created and used in the experimental results. Also some real-life packages at http://www.gnu.org/directory/all/ are used to observe the characteristics of replace algorithm.

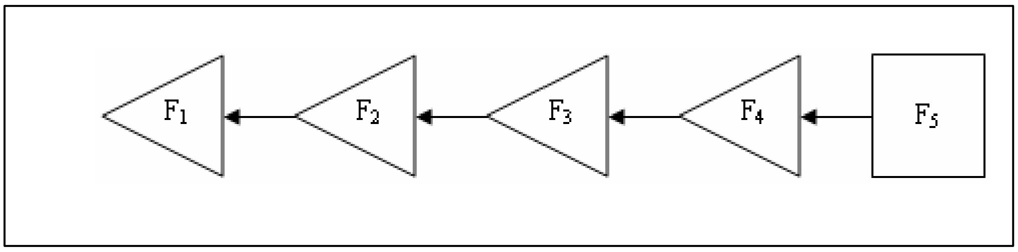


Figure 14: The figure of our generated version chain for experimental results

Each operation (retrieve and delete) is executed twenty times, and their average value is taken in consideration. The UNIX “diff” command is used to calculate the execution time of apply algorithm when intermediate versions are constructed on disk completely or with a fixed-size buffer in memory and replace algorithm which uses one of the above versions of apply. The execution time is split into user and system time. User time indicates the time spent for the CPU process in a program, and system time gives the execution time in kernel and it mainly includes I/O process time. Total time is the sum of both user and system times. The execution time of apply algorithm generating intermediate versions in memory completely and replace algorithm using this version of apply is calculated by the program itself. The computer used for experimental results has Intel Pentium 4 CPU 2.4 GHz, and 512 MB RAM. The operating system is Mandrake 9.1.

5.1. Retrieve Operation

Delta application and delta combination can be compared when more than one delta application is necessary. All versions except literal file and previous version of it in a chain can be generated by both algorithms and their execution results can be compared meaningfully. For example; each generated chain for the thesis has 5 version files and version 5 is literal file in Figure 14, therefore delta application and delta combination can be compared for version 1, 2 and 3.

5.1.1. How Delta Application works for Retrieve Operation

It simply applies intermediate deltas to literal F_5 to produce the required one. If version 2 is required, apply is called three times and the path of execution can be described as

$$\text{Apply}(F_5, ?_{5,4}) + \text{Apply}(F_4, ?_{4,3}) + \text{Apply}(F_3, ?_{3,2})$$

The algorithm requires (n – i) times of delta application to generate version i from the chain, and version n is the literal one.

5.1.2. How Delta Combination works for Retrieve Operation

It combines intermediate delta files between literal and the required one as a single combined delta, and it applies the combined delta to literal F_5. If version 2 is required, then the path of execution can be described as

$$\text{Replace}(?_{5,4}, ?_{4,3}) + \text{Replace}(?_{5,3}, ?_{3,2}) + \text{Apply}(F_5, ?_{5,2}).$$

The algorithm calls replace (n – i - 1) times to generate combined delta file ? $_{n, i}$ and applies it to version n.

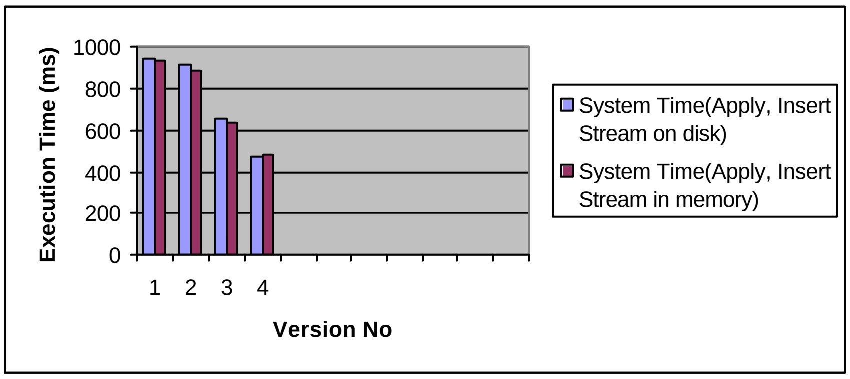
5.2. Run Results for Retrieve Operation

The below cases are studied to observe the retrieve operation of apply and replace algorithms individually and to compare both algorithms in the thesis.

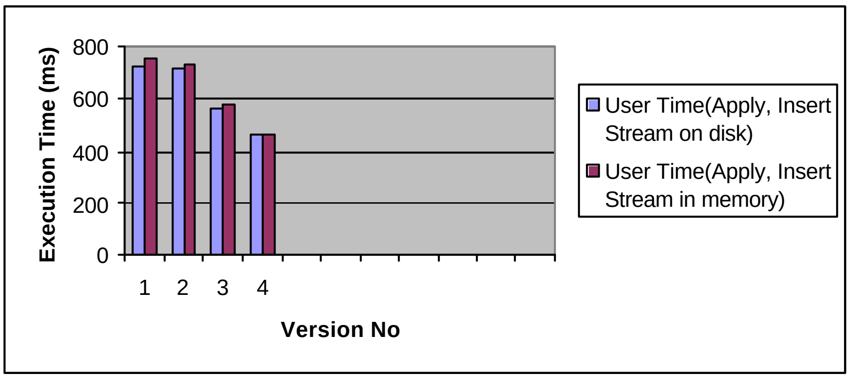
- The performance of apply algorithm when insert stream is on disk or in memory.
- The performance of apply algorithm when destination file is constructed on disk completely, with a fixed-size buffer in memory, or in memory completely. Buffer size is not enough to construct the destination file in memory completely.
- The performance of replace algorithm when it uses binary search algorithm or when it constructs a hash table to find an instruction in an already sorted array.
- The performance considering I/O and CPU time of apply and replace algorithms with a fair comparison

5.2.1. The performance of Apply algorithm when insert stream is on disk or in memory

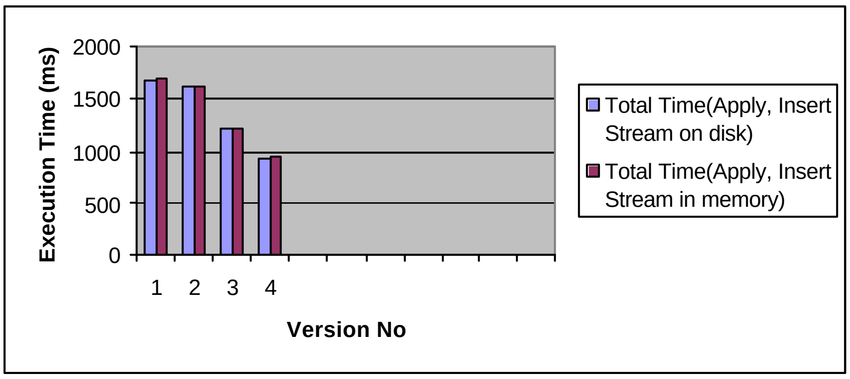
The Figures (15, 16, 17 and 18) observe the performance of apply algorithm when insert stream is on disk or in memory during delta application. Although insert stream length is sufficiently large for both packages (Table 1 and 2), taking the insert stream in memory does not change the total execution time of apply at all. A small improvement is achieved in system time at some figures when insert stream is taken into memory. However, this improvement brings an overhead to user time. The figures imply that insert stream being on disk or in memory has no effect on the total execution time of apply. Therefore, it can be stated that taking the insert stream into memory does not cause any loss on the performance of the replace algorithm in terms of total execution time.



(A)

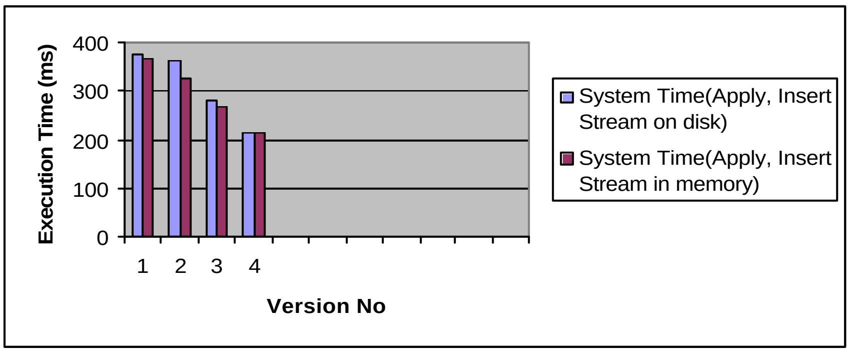


(B)

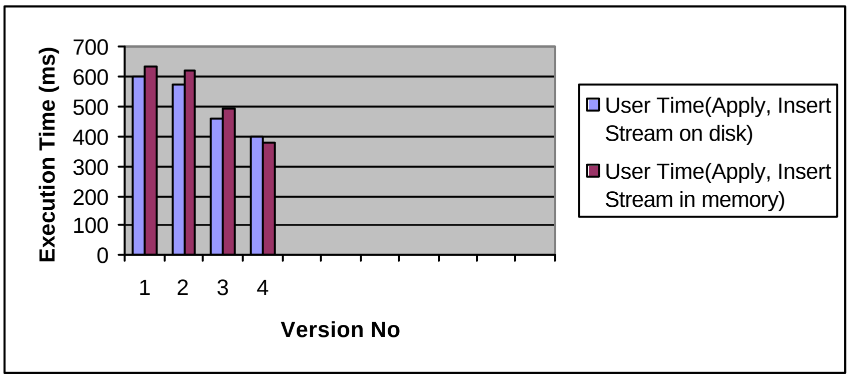


(C)

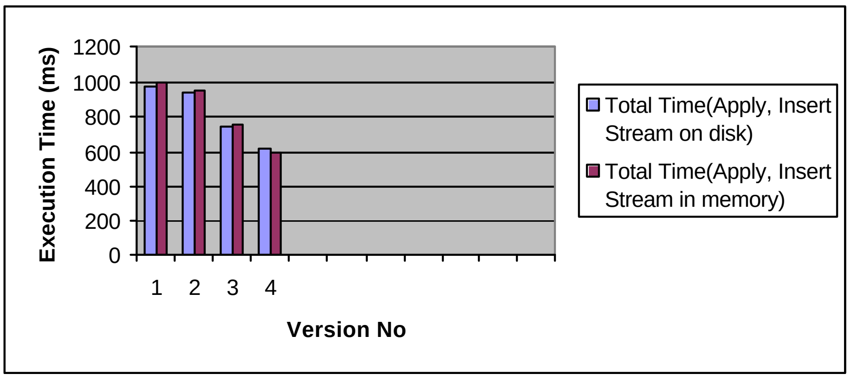
Figure 15: The performance of apply algorithm when insert stream is on disk or in memory, and intermediate versions are constructed on disk for package *gawk*



(A)

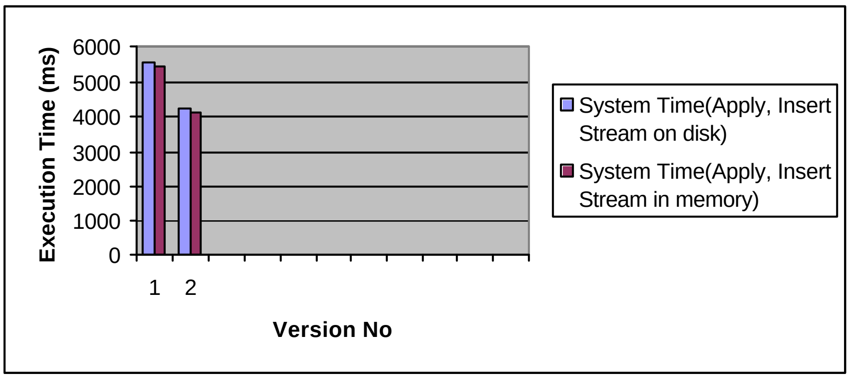


(B)

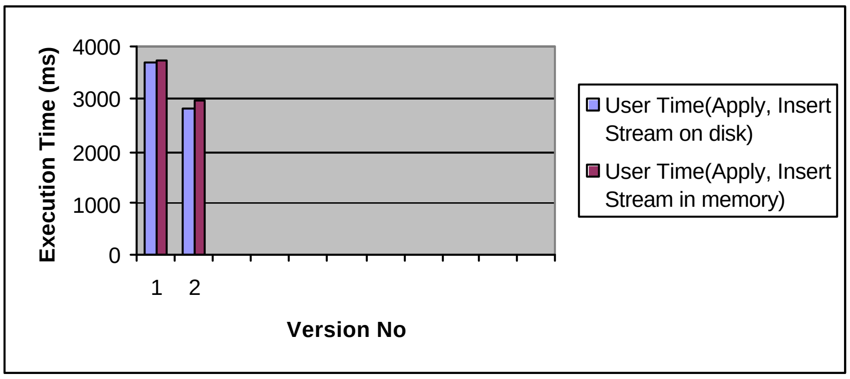


(C)

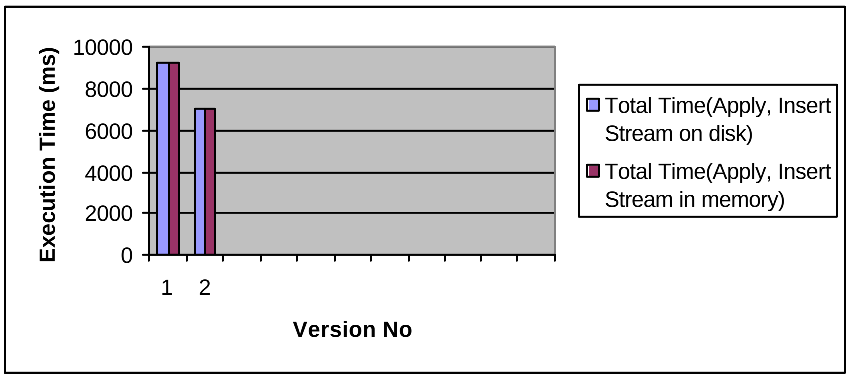
Figure 16: The performance of apply algorithm when insert stream is on disk or in memory, and intermediate versions are constructed with a buffer in memory for package *gawk*



(A)

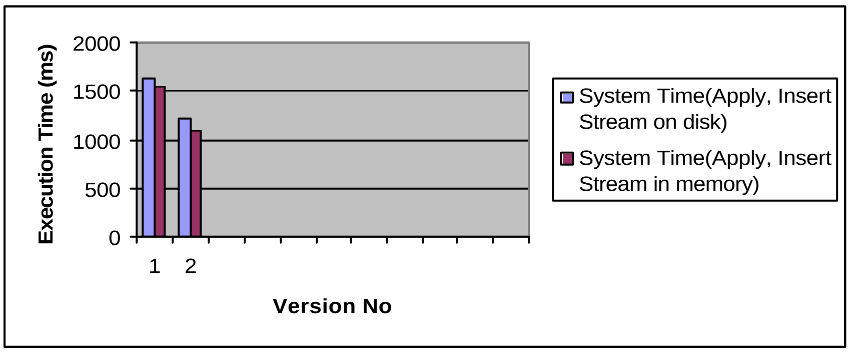


(B)

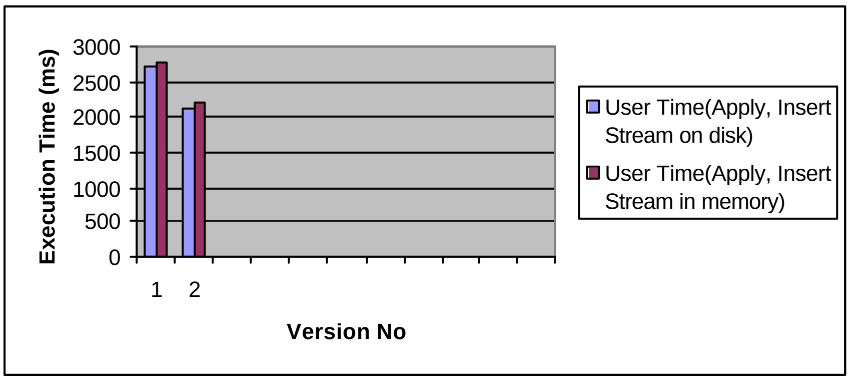


(C)

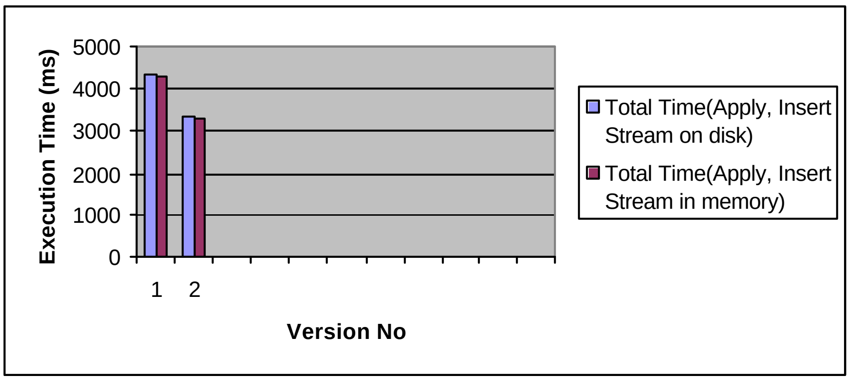
Figure 17: The performance of apply algorithm when insert stream is on disk or in memory, and intermediate versions are constructed on disk for package *chicken*



(A)



(B)



(C)

Figure 18: The performance of apply algorithm when insert stream is on disk or in memory, and intermediate versions are constructed with a buffer in memory for package *chicken*

Table 1: The statistics of instructions that are applied by apply and replace algorithms for package *gawk*

Algorithm	Version No	Apply Copy #	Apply Copy Length	Average Copy Length	Apply Insert #	Apply Insert Length	Average Insert Length
Apply	1	109,812	24,373,007	222	75,652	5,353,713	71
Apply	2	108,118	20,485,349	189	74,485	5,319,451	71
Apply	3	77,129	18,770,449	243	48,065	3,091,951	64
Apply	4	56,288	14,229,416	253	33,300	1,591,384	48
Replace	1	26,480	966,994	37	46,284	2,954,926	64
Replace	2	26,459	974,364	37	44,566	2,968,036	67
Replace	3	32,196	4,013,366	125	29,363	2,028,234	69
Replace	4	46,375	6,145,114	133	29,105	1,412,006	49

Table 2: The statistics of instructions that are applied by apply and replace algorithms for package *chicken*

Algorithm	Version No	Apply Copy #	Apply Copy Length	Average Copy Length	Apply Insert #	Apply Insert Length	Average Insert Length
Apply	1	853,081	57,716,083	68	397,974	9,611,917	24
Apply	2	649,933	38,286,711	59	304,471	7,506,569	25
Replace	1	500,928	15,785,214	32	335,905	5,749,506	17
Replace	2	465,800	16,295,555	35	271,701	5,597,565	21

5.2.2. The performance of Apply algorithm when destination file is constructed on disk completely or with a fixed-size buffer in memory

Figures 19 and 20 show that using memory buffer to construct intermediate versions in apply algorithm improves the system time. When intermediate versions are constructed on disk completely; each instruction, even if negligible in length, copies a byte block from one location to another on disk. However, a buffer in memory eliminates disk seek as long as it is not full.

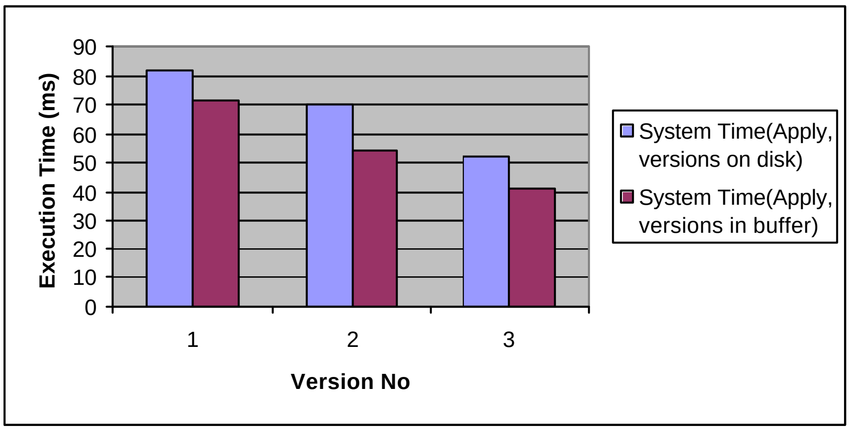


Figure 19: The comparison of apply algorithm when intermediate versions are constructed on disk fully and apply when they are constructed with a buffer in memory while insert stream is on disk for package *mailman*

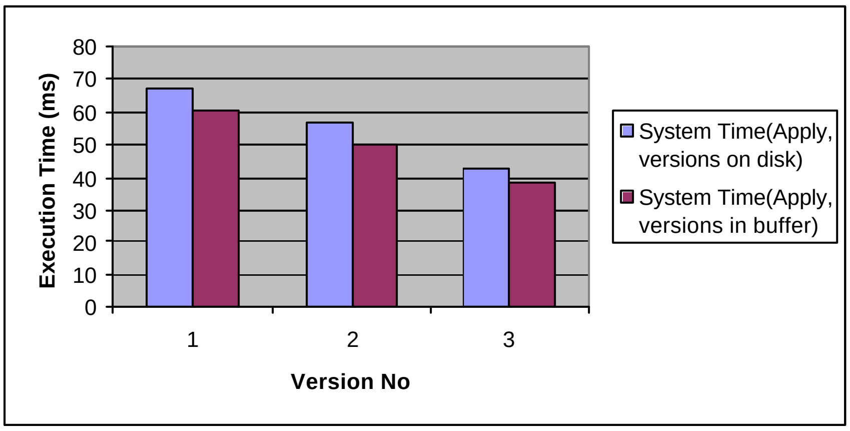


Figure 20: The comparison of apply algorithm when intermediate versions are constructed on disk fully and apply when they are constructed with a buffer in memory while insert stream is in memory for package *mailman*

5.2.3. The performance of Replace algorithm using binary search algorithm or hash data structure

Figures 21 and 22 show that binary search algorithm reduces the user execution time of replace algorithm when compared with hash data structure, because hash data structure consumes an extra time to construct a hash table although it searches over a narrow range. While the difference ratio of the two adjacent versions with respect to file size increases, the difference between the user times also increases in Figure 21. It is also concluded that replace algorithm using binary search produces the required version in less time between 500 KB and 10 MB chains from experimental results. It means that when the number of instructions increases, binary search becomes applicable instead of hash data structure. Figure 22 shows that the number of delta files that are combined and the difference between the user times increases.

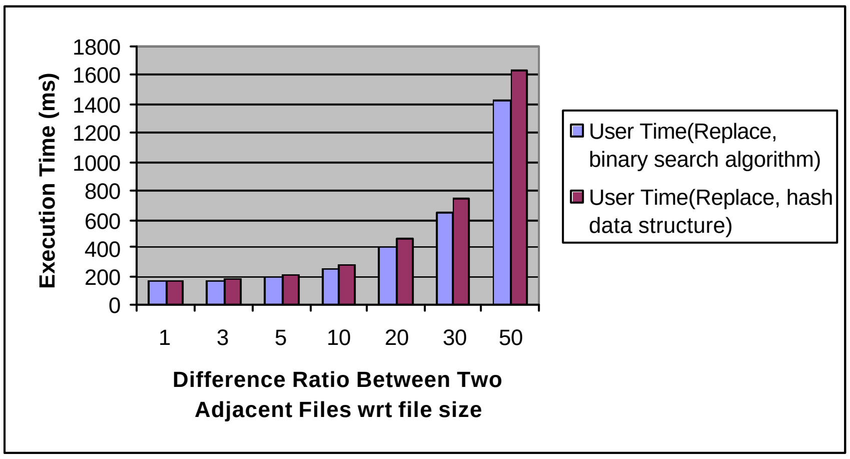


Figure 21: User Time of replace algorithm when binary search or hash data structure is used, and intermediate versions are constructed with a buffer in memory for 5 MB Chains

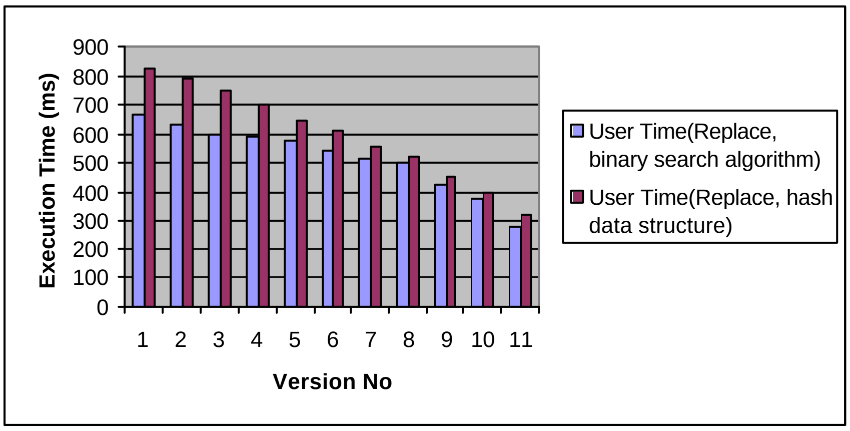


Figure 22: User Time of replace algorithm when binary search or hash data structure is used, and intermediate versions are constructed with a buffer in memory for package *nano*

5.2.4. The performance of Apply and Replace algorithms with a fair comparison

Section 5.2.2 and 5.2.3 show that best total execution time is achieved in apply algorithm generating intermediate versions with a fixed-size buffer in memory and replace algorithm using binary search compared to their alternatives. Section 5.2.1 proves that when insert stream is on disk or in memory has no effect on the total execution time of apply algorithm during delta application. Therefore, a fair comparison can be made between replace algorithm using binary search, and apply algorithm taking insert stream into memory while intermediate versions are constructed with a buffer in memory.

Figures (23, 24 and 25) show that replace algorithm provides great reduction in system time when compared to delta application and the conclusion is valid for all our experiments. As previously mentioned, delta combination does not generate intermediate versions and the difference between system times is a waste consumed by delta application. Replace algorithm generates the version in less time than apply algorithm in Figures (26, 27 and 28).

The improvement in system time brings an overhead to user time of replace algorithm. The reduction in the system time is greater than increment in the user time for our generated chains. As a result, it is concluded that replace algorithm yields a better execution time when considering our generated chains especially while difference ratio increases for the series between 300 KB and 10 MB.

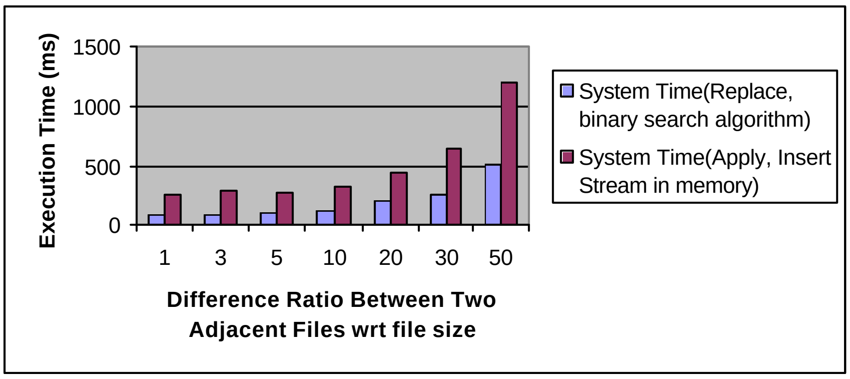


Figure 23: The system time comparison of apply and replace algorithm when intermediate versions are constructed with a buffer in memory for 10 MB Chains

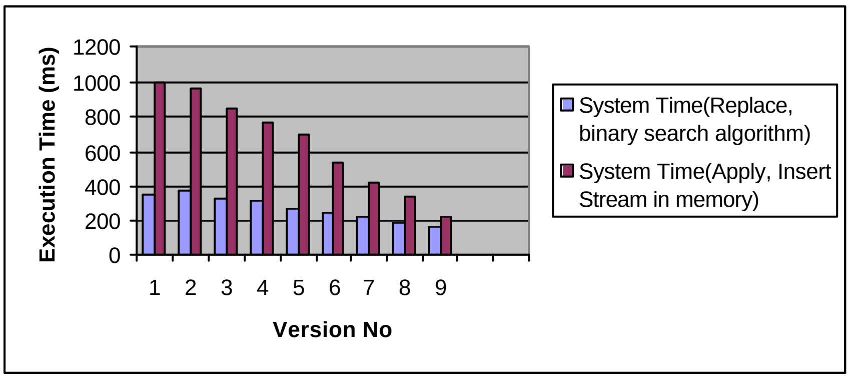


Figure 24: The system time comparison of apply and replace algorithm when intermediate versions are constructed with a buffer in memory for package *metahtml*

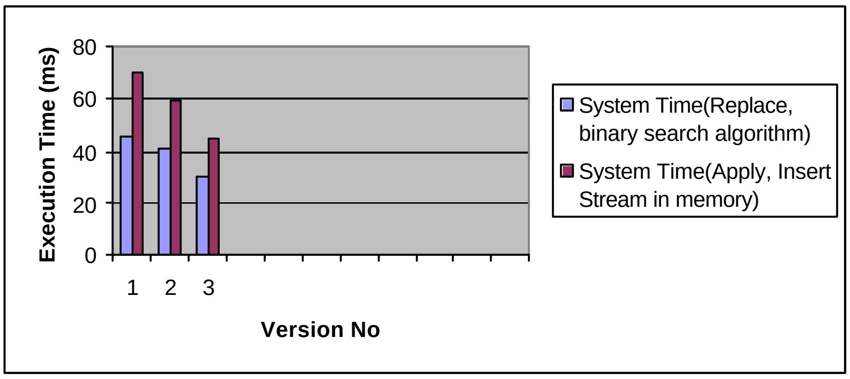


Figure 25: The system time comparison of apply and replace algorithm when intermediate versions are constructed with a buffer in memory for package *marst*

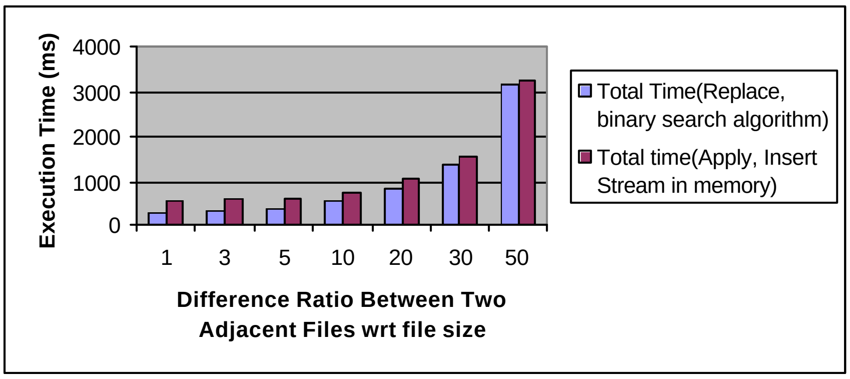


Figure 26: The total time comparison of apply and replace algorithm when intermediate versions are constructed with a buffer in memory for 10 MB Chains

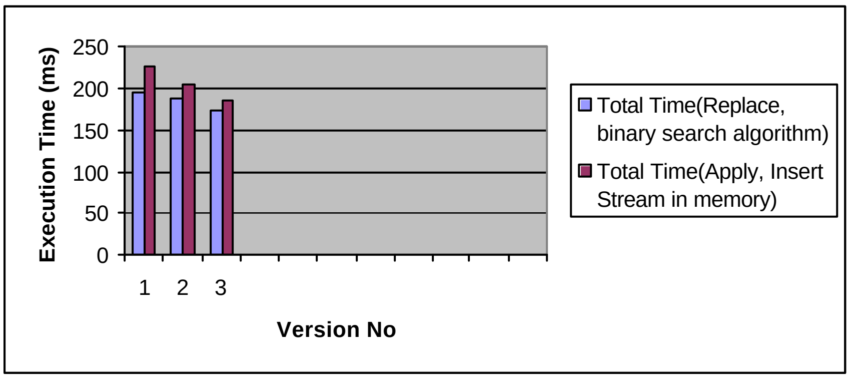


Figure 27: The total time comparison of apply and replace algorithm when intermediate versions are constructed with a buffer in memory for package *mailman*

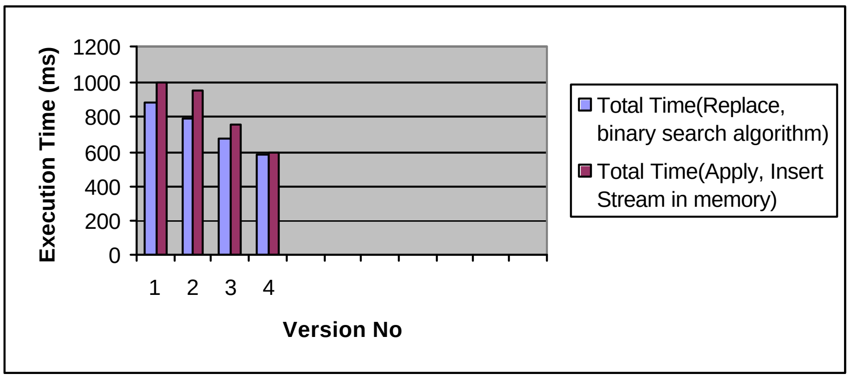
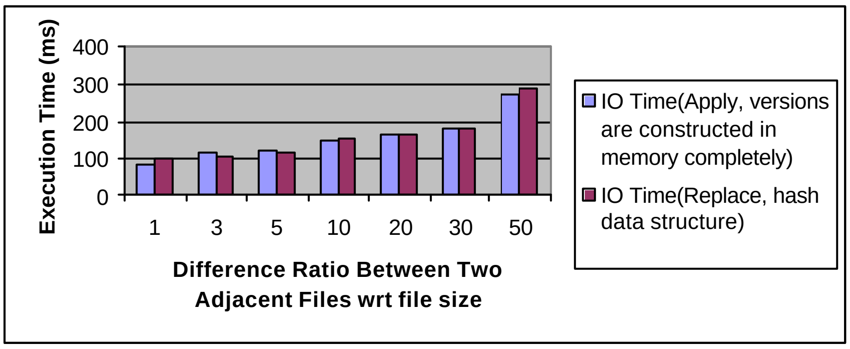
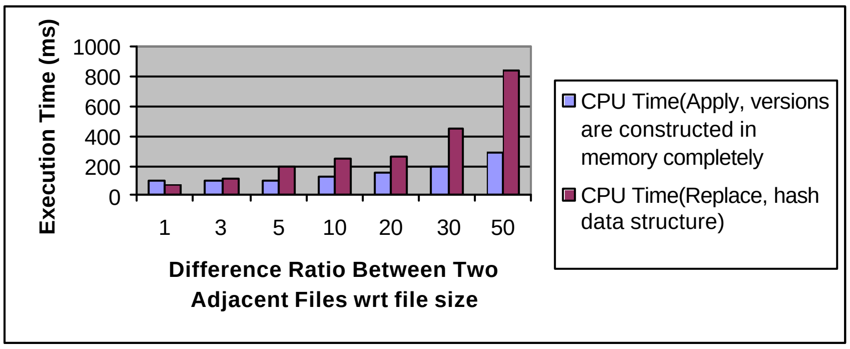


Figure 28: The total time comparison of apply and replace algorithm when intermediate versions are constructed with a buffer in memory for package *gawk*

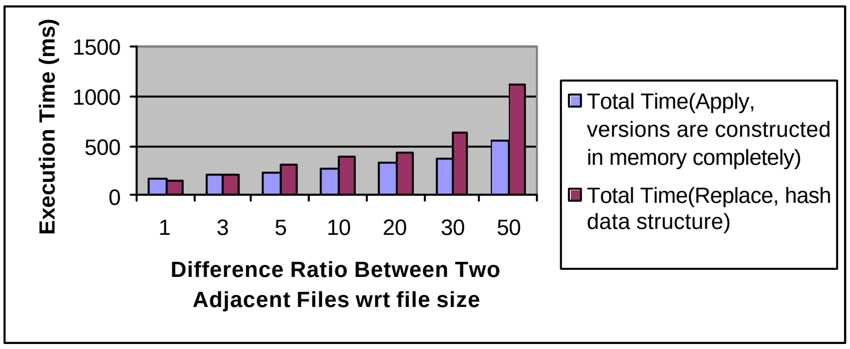
The delta application almost beats delta combination for each of our generated version chains when apply algorithm generates the intermediate versions in memory completely. CPU performance of replace algorithm is dominant in execution time. While delta application generates the required version, it also produces intermediate versions temporarily. These intermediate files are not stored on disk while execution, because memory size is enough to keep an intermediate version. That becomes the main advantage of delta algorithm; because it does not require any IO operations for intermediate files.



(A)



(B)



(C)

Figure 29: The comparison of apply when intermediate versions are constructed in memory completely and replace algorithm using hash data structure for 1 MB Chains and execution times are calculated by the program itself

5.3. Delete Operation

The performance of apply and replace algorithms are compared also for delete operation of a version in a chain. Deleting edge versions of a chain, literal or first one, is straightforward, in which replace algorithm is not applicable. Deletion of version 1 requires only deleting ? $_{2,1}$ from disk, deletion of literal requires applying ? $_{5,4}$ to F_5 and deleting F_5 from disk.

5.3.1. How Delta Application works for Delete Operation

Deletion of version 2 from our chain requires generating version 1 and version 3 internally, then calculating delta file ?$_{3,1}$ of them, and deleting ?$_{2,1}$ and ?$_{3,2}$ from disk. Generation of versions in deletion operation requires apply algorithm 4 times for the case. If version n is literal and version i which is not at any edge of the chain will be deleted, the algorithm requires delta application (n + 1 – i) times. The execution path can be described as

Apply(F_5, ? $_{5,\ 4}$) + Apply(F_4, ? $_{4,\ 3}$) + Apply(F_3, ? $_{3,2}$) + Apply(F_2, ? $_{2,1}$) + DeltaAlgorithm(F_3, F_1)

Apply algorithm has a linear execution time therefore the execution time of the delta algorithm becomes vital in delete operation. If the general greedy algorithm is preferred, it runs in O (n^2) time and it becomes dominant on the total execution time (Figure 29). If a linear delta algorithm is selected, it runs in O (n) time.

5.3.2. How Delta Combination works for Delete Operation

It is straightforward to delete version 2 with replace algorithm, it combines ? $_{3,2}$ and ? $_{2,1}$ only, and the path of execution can be described as

Replace(? $_{3,2}$, ? $_{2,1}$)

5.4. Run Results for Delete Operation

Figure 30 compares apply algorithm when intermediate versions are constructed in memory completely and replace algorithm using hash data structure. The execution time of apply algorithm is divided into two bars to show the overhead of the general greedy delta algorithm. The second bar indicates the execution time of apply algorithm which does not include the time consumed by the delta algorithm and the longest bar indicates the total execution time of apply algorithm. As seen in the figure, replace algorithm runs in less time than apply algorithm even if the time consumed by delta algorithm is not included in the second bar. Its performance is better than delta application not including the performance of delta algorithm for larger than 500 KB chains. However, replace algorithm does not produce an optimum delta file as a delta algorithm does. It can be preferable when working on large version files and/or when the performance of delta algorithm is considered.

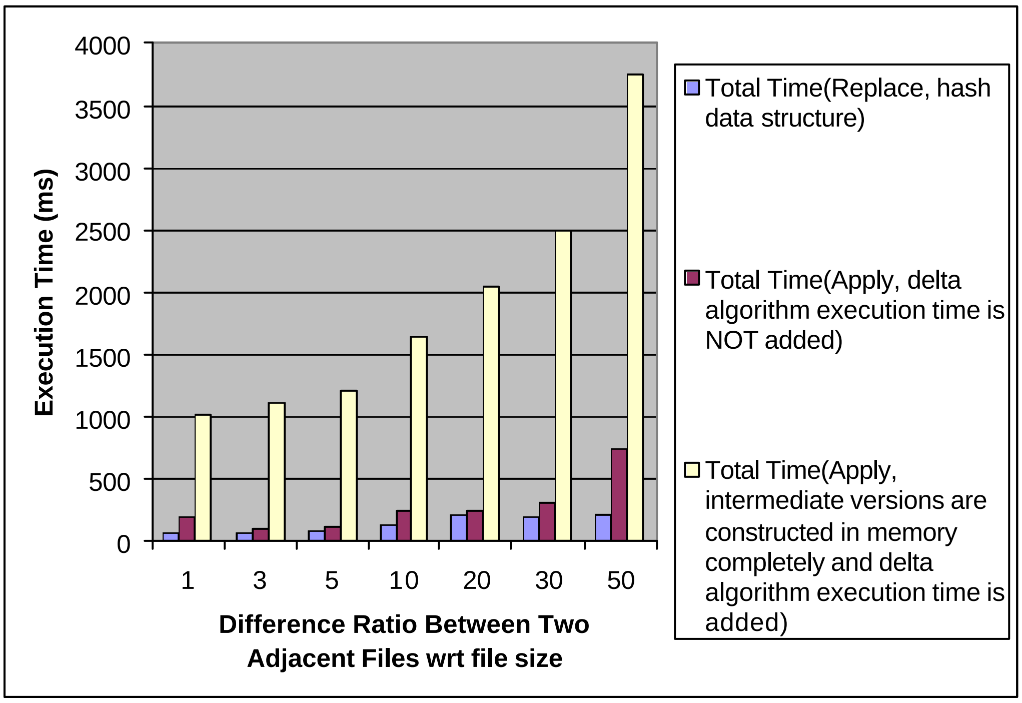


Figure 30: The comparison of apply when intermediate versions are constructed in memory completely and replace algorithm using hash data structure to delete version 2 from 3 MB Chains and execution times are calculated by the program itself

CHAPTER 6

CONCLUSIONS

The strategy of replace algorithm is to eliminate the I/O operations that are done for intermediate versions to retrieve a version in a chain while delta application. Replace algorithm combines the intermediate delta files and generates a single combined delta in the run time. It finally generates the required version with one delta application using the single combined delta.

Delta application is better than delta combination when apply algorithm generates the intermediate versions in memory completely instead of storing them on disk temporarily. However, memory capacity and server load may not allow intermediate versions to be constructed in memory completely especially while working on large version files. If an intermediate version of a chain cannot be stored in memory wholly during delta application, then replace algorithm can be applicable and yield a better solution.

Many cases are studied and experiments are performed to observe the performance of delta application and delta combination for retrieve operation, and below results are concluded.

- There is no significant improvement in total execution time of apply algorithm when insert stream is on disk or in memory. Therefore, it can be stated that although loading insert stream into memory during delta combination is possible, it does not affect the performance of replace algorithm.
- Generating intermediate versions with a buffer in memory improves the I/O time of apply algorithm.

- Using binary search in the replace algorithm reduces the CPU time when compared with hash table search, because hash table construction causes a significant overhead. Our experiments show that binary search becomes preferable when the number of instructions in a delta file increases.

- Replace algorithm reduces the I/O operations when it is compared with delta application because it does not generate intermediate versions, temporarily, on disk. Thus, it would be useful for reducing I/O load on a file server, while shifting the CPU load to the clients.

Delete operation for replace algorithm is simple; it combines two adjacent delta files of version which will be deleted. However, delta application produces adjacent versions, and computes difference of them. Replace algorithm for deleting intermediate versions in a chain is can be a solution when working on large files. Its performance is better than delta application without considering performance of delta algorithm for larger than 500 KB Chains.

6.1. Future Work

Our experimental results show that replace algorithm causes a reduction in the I/O time while it also causes CPU time to be increased. Thus, considering total execution time, the CPU overhead eliminates some of the reduction in I/O time. Our results also show that delta combination generates the same version in less time than delta application for our generated chains and some gnu software packages delivered over Internet. If the characteristics of replace algorithm can be defined for different data types, then it can be decided that replace algorithm or apply algorithm is preferable by checking the characteristics of the delta file.

A version chain generator will be implemented, and it will produce the versions in terms of the count, length and ratio of instructions and file size. The combined delta produced by replace algorithm will be compared with the one produced by delta application in terms of delta file size, number of instruction – insert and copy -, and sum of length of each instruction. Therefore, the characteristic of the algorithm can be defined with these data sets. Then, a version control system can decide to use apply or replace algorithm to generate a version from a chain.

REFERENCES

[1] Brenda S. Baker, Udi Manber, and Robert Muth, “Compressing Differences of Executable Code”, April 1999.

[2] G. Myers. *A fast bit-vector algorithm for approximate pattern matching based on dynamic programming*. In Proc. CPM'98, LNCS v. 1448, pp. 1-13, Springer-Verlag, 1998.

[3] Henner Zeller, "Design and Implementation of a Distributed Application Independent Versioning Object Repository and Investigation of its Usability as a Component of the System CAMPUS for Case-Based Training in Medicine". MS Thesis. Medizinische Informatik, Universität Heidelberg Fachhochschule Heilbronn, July 2001.

[4] http://subversion.tigris.org , Apr 26, 2004

[5] http://svn.collab.net/repos/svn-xml/trunk/notes/fs-improvements.txt , Apr 26, 2004

[6] J.W. HUNT, T.G. SZYMANSKI A fast algorithm for computing longest common subsequences. Communications of the ACM, 20(5):350–353, May 1977.

[7] J. J. Hunt, Kiem-Phong Vo, and W. F. Tichy. Delta algorithms: An empirical analysis. *ACM Transactions on Software Engineering and Methodology*, v. 7(2): pp. 192–214, 1998.

[8] Josh MacDonald. File System Support for Delta Compression. MS Thesis. Department of Electrical Engineering and Computer Science, University of California at Berkeley EECS, May 2000

[9] Josh MacDonald, "Versioned File Archiving, Compression and Distribution" UC Berkeley.

[10] Miklos Ajtai, Randal Burns, Ronald Fagin, Darrell D. E. Long, Larry Stockmeyer, "Compactly Encoding Unstructured Inputs with Differential Compression" v. 49(3): pp. 318–367, 2002.

[11] Randal C. Burns, Darrell D. E. Long, "Efficient Distributed Backup with Delta Compression," Proceedings of the Fifth Workshop on I/O in Parallel and Distributed Systems, ACM: San Jose, pp. 26-36, Nov 1997

[12] Randal C. Burns, "Differential Compression: A Generalized Solution for Binary Files". MS Thesis. Department of Computer Science, University of California at Santa Cruz, December 1996.

[13] Randal C. Burns, Larry Stockmeyer and Darrell Long. "Experimentally Evaluating In-Place Delta Reconstruction," Proceedings of the NASA and IEEE Mass Storage Conference, College Park: IEEE, pp. 137–151, April 2002.

[14] Rochkind, Marc J., "The Source Code Control System" IEEE Transactions on Software Engineering, vol. SE-1, no. 4, pp. 364-370, Dec. 1975.

[15] W.F. Tichy, "RCS- A System for Version Control", Software-Practice and Experience, vol. 15, no. 7, pp. 637-654, July 1985

[16] W. F. Tichy. "*The string-to-string correction problem with block move*" ACM Transactions on Computer Systems, 2(4), November 1984.

APPENDIX A

THE VERSIONS AND THEIR FILE SIZE OF GNU PACKAGES USED IN EXPERIMENTAL RESULTS

Table A.1: The versions of package *mailman*

#	Name	Size (byte)
1	Mailman-2.0.1.tar	1,710,080
2	Mailman-2.0.2.tar	1,710,080
3	Mailman-2.0.3.tar	1,710,080
4	Mailman-2.0.4.tar	1,710,080
5	Mailman-2.0.5.tar	1,720,320

Table A.2: The versions of package *metahtm*

#	Name	Size (byte)
1	metahtml-5.00.tar	5,509,120
2	metahtml-5.01.tar	8,007,680
3	metahtml-5.02.tar	8,816,640
4	metahtml-5.03.tar	9,656,320
5	metahtml-5.04.tar	9,666,560
6	metahtml-5.05.tar	9,963,520
7	metahtml-5.06.tar	10,137,600
8	metahtml-5.07.tar	9,420,800
9	metahtml-5.08.tar	12,072,960
10	metahtml-5.09.tar	9,123,840
11	metahtml-5.091.tar	10,362,880

Table A.3: The versions of package *nano*

#	Name	Size (byte)
1	nano-1.0.0.tar	1,433,600
2	nano-1.0.1.tar	1,433,600
3	nano-1.0.2.tar	1,546,240
4	nano-1.0.3.tar	1,648,640
5	nano-1.0.4.tar	1,740,800
6	nano-1.0.5.tar	1,812,480
7	nano-1.0.6.tar	1,832,960
8	nano-1.0.7.tar	1,863,680
9	nano-1.0.8.tar	1,904,640
10	nano-1.0.9.tar	1,955,840
11	nano-1.2.0.tar	3,256,320
12	nano-1.2.1.tar	3,266,560
13	nano-1.2.3.tar	3,491,840

Table A.4: The versions of package *marst*

#	Name	Size (byte)
1	Marst-2.0.tar	716,800
2	Marst-2.1.tar	1,556,480
3	Marst-2.2.tar	1,566,720
4	Marst-2.3.tar	1,464,320
5	Marst-2.4.tar	1,423,360

APPENDIX B

THE INSTRUCTION STATISTICS OF THE GENERATED VERSION CHAINS AND GNU PACKAGES USED IN EXPERIMENTAL RESULTS

Package Marst

Table B.1: The statistics of instructions that are applied by Apply and Replace algorithms for package marst

Algorithm	Version No	Apply Copy #	Apply Copy Length	Average Copy Length	Apply Insert #	Apply Insert Length	Average Insert Length
Apply	1	12,457	4,101,344	329	9,994	1,202,976	120
Apply	2	7,281	3,871,092	532	5,472	716,428	131
Apply	3	6,663	2,320,810	348	5,118	710,230	139
Replace	1	7,444	142,278	19	7,556	574,522	76
Replace	2	6,682	847,609	127	5,618	708,871	126
Replace	3	6,473	859,055	133	5,157	707,665	137

Table B.2: The statistics of instructions that are replaced by Replace algorithm for package marst

Algorithm	Version No	Replace Copy #	Replace Copy Length	Average Copy Length	Replace Insert #	Replace Insert Length	Average Insert Length
Replace	1	11,926	2,643,300	222	9,727	1,196,700	123
Replace	2	6,750	2,413,048	357	5,205	710,152	136
Replace	3	6,132	862,766	141	4,851	703,954	145

Package Mailman

Table B.3: The statistics of instructions that are applied by Apply and Replace algorithms for package mailman

Algorithm	Version No	Apply Copy #	Apply Copy Length	Average Copy Length	Apply Insert #	Apply Insert Length	Average Insert Length
Apply	1	3,241	6,827,578	2,107	2,784	12,742	5
Apply	2	2,564	5,119,388	1,997	2,236	10,852	5
Apply	3	1,908	3,410,599	1,788	1,699	9,561	6
Replace	1	1,551	1,700,262	1,096	1,552	9,818	6
Replace	2	1,491	1,701,004	1,141	1,514	9,076	6
Replace	3	1,431	1,701,347	1,189	1,359	8,733	6

Table B.4: The statistics of instructions that are replaced by Replace algorithm for package mailman

Algorithm	Version No	Replace Copy #	Replace Copy Length	Average Copy Length	Replace Insert #	Replace Insert Length	Average Insert Length
Replace	1	2,524	5,121,020	2,029	2,218	9,220	4
Replace	2	1,847	3,412,830	1,848	1,670	7,330	4
Replace	3	1,191	1,704,041	1,431	1,133	6,039	5

Package MetaHtml

Table B.5: The statistics of instructions that are applied by Apply and Replace algorithms for package metahtml

Algorithm	Version No	Apply Copy #	Apply Copy Length	Average Copy Length	Apply Insert #	Apply Insert Length	Average Insert Length
Apply	1	279,506	72,272,273	259	218,609	21,341,807	98
Apply	2	261,387	68,282,516	261	206,046	19,822,444	96
Apply	3	221,641	61,971,804	280	174,647	18,125,476	104
Apply	4	175,875	57,266,082	326	137,597	14,014,558	102
Apply	5	168,716	47,639,425	282	132,262	13,984,895	106
Apply	6	126,177	40,464,550	321	99,653	11,493,210	115
Apply	7	100,311	32,925,487	328	79,952	9,068,753	113
Apply	8	81,936	24,740,365	302	65,792	7,116,275	108
Apply	9	28,043	9,040,691	322	23,944	3,032,269	127
Replace	1	37,450	1,094,001	29	89,280	4,415,119	49
Replace	2	46,026	1,446,868	31	144,880	6,560,812	45
Replace	3	48,428	1,752,288	36	120,361	7,064,352	59
Replace	4	56,528	4,203,944	74	109,108	5,452,376	50
Replace	5	55,752	4,212,220	76	106,587	5,454,340	51
Replace	6	48,702	4,912,543	101	76,565	5,050,977	66
Replace	7	46,680	5,382,940	115	61,907	4,754,660	77
Replace	8	43,248	6,066,925	140	44,478	3,353,875	75
Replace	9	28,043	9,040,691	322	23,944	3,032,269	127

Table B.6: The statistics of instructions that are replaced by Replace algorithm for package metahtml

Algorithm	Version No	Replace Copy #	Replace Copy Length	Average Copy Length	Replace Insert #	Replace Insert Length	Average Insert Length
Replace	1	265,084	63,051,373	238	207,491	20,199,827	97
Replace	2	246,965	59,061,616	239	194,928	18,680,464	96
Replace	3	207,219	52,750,904	255	163,529	16,983,496	104
Replace	4	161,453	48,045,182	298	126,479	12,872,578	102
Replace	5	154,294	38,418,525	249	121,144	12,842,915	106
Replace	6	111,755	31,243,650	280	88,535	10,351,230	117
Replace	7	85,889	23,704,587	276	68,834	7,926,773	115
Replace	8	67,514	15,519,465	230	54,674	5,974,295	109
Replace	9	26,102	9,113,641	349	19,677	2,959,319	150

Package Nano

Table B.7: The statistics of instructions that are applied by Apply and Replace algorithms for package nano

Algorithm	Version No	Apply Copy #	Apply Copy Length	Average Copy Length	Apply Insert #	Apply Insert Length	Average Insert Length
Apply	1	53,482	19,403,421	363	42,859	4,291,939	100
Apply	2	52,243	17,974,635	344	41,853	4,287,125	102
Apply	3	48,644	16,636,063	342	38,857	4,192,097	108
Apply	4	47,269	15,142,186	320	37,749	4,139,734	110
Apply	5	45,525	13,537,888	297	36,215	4,095,392	113
Apply	6	44,276	11,863,024	268	35,229	4,029,456	114
Apply	7	42,315	10,064,339	238	33,703	4,015,661	119
Apply	8	40,901	8,273,680	202	32,624	3,973,360	122
Apply	9	38,294	6,460,897	169	30,605	3,922,463	128
Apply	10	32,807	5,471,995	167	27,720	3,006,725	108
Apply	11	17,565	4,854,489	276	14,240	1,668,391	117
Replace	1	2,223	93,563	42	23,074	1,340,037	58
Replace	2	2,128	89,907	42	22,826	1,343,693	59
Replace	3	2,421	102,120	42	22,686	1,444,120	64
Replace	4	2,709	111,156	41	26,322	1,537,484	58
Replace	5	2,968	119,852	40	29,080	1,620,948	56
Replace	6	3,133	120,581	38	31,537	1,691,899	54
Replace	7	3,247	126,927	39	30,662	1,706,033	56
Replace	8	3,205	128,288	40	30,667	1,735,392	57
Replace	9	3,351	139,736	42	30,117	1,764,904	59
Replace	10	4,842	266,306	55	28,257	1,689,534	60
Replace	11	10,896	1,642,947	151	20,975	1,613,373	77

Table B.8: The statistics of instructions that are replaced by Replace algorithm for package nano

Algorithm	Version No	Replace Copy #	Replace Copy Length	Average Copy Length	Replace Insert #	Replace Insert Length	Average Insert Length
Replace	1	43,193	17,726,567	410	34,753	2,702,233	78
Replace	2	41,954	16,297,781	388	33,747	2,697,419	80
Replace	3	38,355	14,959,209	390	30,751	2,602,391	85
Replace	4	36,980	13,465,332	364	29,643	2,550,028	86
Replace	5	35,236	11,861,034	337	28,109	2,505,686	89
Replace	6	33,987	10,186,170	300	27,123	2,439,750	90
Replace	7	32,026	8,387,485	262	25,597	2,425,955	95
Replace	8	30,612	6,596,826	215	24,518	2,383,654	97
Replace	9	28,005	4,784,043	171	22,499	2,332,757	104
Replace	10	22,518	3,795,141	169	19,614	1,417,019	72
Replace	11	7,276	3,177,635	437	6,134	78,685	13

50 KB Series

Table B.9: The statistics of instructions that are applied by apply and replace algorithms for 50 KB Chain

Algorithm	Ratio	Apply Copy #	Apply Copy Length	Average Copy Length	Apply Insert #	Apply Insert Length	Average Insert Length
Apply	1	26	201,880	7,765	0	0	0
Apply	3	47	213,248	4,537	0	0	0
Apply	5	119	212,920	1,789	3	34	11
Apply	10	231	230,463	998	32	327	10
Apply	20	562	267,005	475	128	1,907	15
Apply	30	1,087	303,657	279	449	5,631	13
Apply	50	2,861	360,183	126	1,790	27,799	16
Replace	1	21	49,784	2,371	0	0	0
Replace	3	44	51,744	1,176	0	0	0
Replace	5	106	48,868	461	3	34	11
Replace	10	190	50,515	266	26	249	10
Replace	20	389	49,070	126	98	1,400	14
Replace	30	560	43,921	78	290	3,707	13
Replace	50	890	36,674	41	759	11,052	15

Table B.10: The statistics of instructions that are replaced by replace algorithm for 50 KB Chain

Algorithm	Ratio	Replace Copy #	Replace Copy Length	Average Copy Length	Replace Insert #	Replace Insert Length	Average Insert Length
Replace	1	18	150,626	8,368	0	0	0
Replace	3	35	158,074	4,516	0	0	0
Replace	5	85	155,688	1,832	3	34	11
Replace	10	145	165,637	1,142	19	179	9
Replace	20	369	182,089	493	77	1,171	15
Replace	30	743	193,172	260	339	4,396	13
Replace	50	1,472	210,211	143	892	14,699	16

100 KB Series

Table B.11: The statistics of instructions that are applied by apply and replace algorithms for 100 KB Chain

Algorithm	Ratio	Apply Copy #	Apply Copy Length	Average Copy Length	Apply Insert #	Apply Insert Length	Average Insert Length
Apply	1	50	404,446	8,089	0	0	0
Apply	3	119	418,950	3,521	0	0	0
Apply	5	219	426,292	1,947	9	106	12
Apply	10	501	450,569	899	43	623	14
Apply	20	1,200	423,896	437	326	4,814	15
Apply	30	2,071	594,409	287	816	10,643	13
Apply	50	5,584	783,079	140	3,206	41,493	13
Replace	1	47	99,176	2,110	0	0	0
Replace	3	107	100,548	940	0	0	0
Replace	5	195	99,103	508	5	73	15
Replace	10	395	93,912	238	40	560	14
Replace	20	788	92,138	117	247	3,510	14
Replace	30	1,034	92,852	90	464	5,834	13
Replace	50	1,950	88,532	45	1,473	17,700	12

Table B.12: The statistics of instructions that are replaced by replace algorithm for 100 KB Chain

Algorithm	Ratio	Replace Copy #	Replace Copy Length	Average Copy Length	Replace Insert #	Replace Insert Length	Average Insert Length
Replace	1	42	301,350	7,175	0	0	0
Replace	3	81	310,170	3,829	0	0	0
Replace	5	157	311,557	1,984	7	83	12
Replace	10	358	318,476	890	30	154	17
Replace	20	825	352,722	428	228	3,410	15
Replace	30	1,279	384,071	300	491	6,165	13
Replace	50	3,172	459,240	145	1,864	24,880	13

300 KB Series

Table B.13: The statistics of instructions that are applied by apply and replace algorithms for 300 KB Chain

Algorithm	Ratio	Apply Copy #	Apply Copy Length	Average Copy Length	Apply Insert #	Apply Insert Length	Average Insert Length
Apply	1	133	1,215,494	9,139	0	0	0
Apply	3	343	1,262,991	3,682	6	33	5
Apply	5	683	1,282,326	1,877	52	690	13
Apply	10	1,567	1,351,727	863	196	2,339	12
Apply	20	3,589	1,590,164	443	976	11,058	11
Apply	30	7,200	1,787,257	248	3,051	32,015	10
Apply	50	18,787	2,301,810	123	10,833	122,612	11
Replace	1	129	299,194	2,319	0	0	0
Replace	3	318	303,685	955	3	17	6
Replace	5	617	296,976	481	48	650	14
Replace	10	1,230	285,731	232	158	1,899	12
Replace	20	2,372	290,738	123	715	7,672	11
Replace	30	3,699	269,644	73	1,856	18,280	10
Replace	50	6,066	243,300	40	4,830	50,798	11

Table B.14: The statistics of instructions that are replaced by replace algorithm for 300 KB Chain

Algorithm	Ratio	Replace Copy #	Replace Copy Length	Average Copy Length	Replace Insert #	Replace Insert Length	Average Insert Length
Replace	1	100	906,500	9,065	0	0	0
Replace	3	250	934,707	3,739	3	17	6
Replace	5	485	936,659	1,931	31	417	13
Replace	10	1,087	959,494	883	127	1,494	12
Replace	20	2,374	1,079,077	455	610	7,155	12
Replace	30	4,760	1,137,922	239	2,074	21,418	10
Replace	50	10,828	1,333,746	123	6,331	76,278	12

500 KB Series

Table B.15: The statistics of instructions that are applied by apply and replace algorithms for 500 KB Chain

Algorithm	Ratio	Apply Copy #	Apply Copy Length	Average Copy Length	Apply Insert #	Apply Insert Length	Average Insert Length
Apply	1	224	2,027,218	9,050	2	10	5
Apply	3	606	2,100,576	3,466	9	152	17
Apply	5	1,135	2,146,691	1,891	69	783	11
Apply	10	2,480	2,284,094	921	271	2,736	10
Apply	20	6,291	2,629,566	418	1,910	20,452	11
Apply	30	12,090	2,956,261	245	5,047	52,731	10
Apply	50	31,499	3,786,727	120	17,745	179,725	10
Replace	1	213	498,810	2,342	2	10	5
Replace	3	567	502,295	886	9	151	17
Replace	5	1,015	496,750	489	59	698	12
Replace	10	1,972	484,591	246	228	2,175	10
Replace	20	4,032	476,731	118	1,372	14,249	10
Replace	30	6,225	442,308	71	3,106	30,738	10
Replace	50	10,258	399,170	39	8,056	74,268	9

Table B.16: The statistics of instructions that are replaced by replace algorithm for 500 KB Chain

Algorithm	Ratio	Replace Copy #	Replace Copy Length	Average Copy Length	Replace Insert #	Replace Insert Length	Average Insert Length
Replace	1	172	1,511,934	8,790	2	10	5
Replace	3	450	1,553,357	3,452	7	139	20
Replace	5	840	1,567,660	1,866	44	536	12
Replace	10	1,768	1,623,939	919	200	2,077	10
Replace	20	4,150	1,782,786	430	1,223	12,966	11
Replace	30	7,615	1,888,728	248	3,150	34,228	11
Replace	50	18,547	2,180,567	118	10,735	113,123	11

1 MB Series

Table B.17: The statistics of instructions that are applied by apply and rplace algorithms for 1 MB Chain

Algorithm	Ratio	Apply Copy #	Apply Copy Length	Average Copy Length	Apply Insert #	Apply Insert Length	Average Insert Length
Apply	1	417	4,064,447	9,747	10	103	10
Apply	3	1,311	4,165,784	3,178	29	294	10
Apply	5	2,330	4,280,128	1,837	158	1,590	10
Apply	10	5,303	4,525,952	853	783	7,234	9
Apply	20	12,445	5,242,229	421	3,417	32,229	9
Apply	30	23,122	6,021,415	260	8,684	77,811	9
Apply	50	62,833	7,696,062	122	33,673	273,690	8
Replace	1	406	1,000,477	2,464	10	103	10
Replace	3	1,211	994,829	821	23	263	11
Replace	5	2,081	988,234	475	145	1,468	10
Replace	10	4,234	955,196	226	692	6,282	9
Replace	20	8,159	949,993	116	2,576	23,637	9
Replace	30	12,342	925,083	75	5,549	46,979	8
Replace	50	21,237	847,970	40	15,634	114,194	7

Table B.18: The statistics of instructions that are replaced by rplace algorithm for 1 MB Chain

Algorithm	Ratio	Replace Copy #	Replace Copy Length	Average Copy Length	Replace Insert #	Replace Insert Length	Average Insert Length
Replace	1	319	3,032,891	9,507	2	13	6
Replace	3	953	3,076,021	3,228	20	199	10
Replace	5	1,716	3,125,879	1,822	127	1,301	10
Replace	10	3,731	3,211,327	861	512	4,935	10
Replace	20	8,398	3,538,947	421	2,330	21,295	9
Replace	30	15,073	3,859,030	256	5,804	53,032	9
Replace	50	36,681	4,456,603	121	20,068	169,487	8

3 MB Series

Table B.19: The statistics of instructions that are applied by apply and replace algorithms for 3 MB Chain

Algorithm	Ratio	Apply Copy #	Apply Copy Length	Average Copy Length	Apply Insert #	Apply Insert Length	Average Insert Length
Apply	1	1,244	12,184,408	9,795	21	226	11
Apply	3	3,856	12,541,860	3,253	144	1,356	9
Apply	5	6,801	12,861,244	1,891	383	3,510	9
Apply	10	15,553	13,642,415	877	2,082	15,943	8
Apply	20	36,093	15,966,948	442	8,624	60,658	7
Apply	30	69,135	17,945,520	260	24,346	171,936	7
Apply	50	183,585	23,284,724	127	91,861	666,476	7
Replace	1	1,206	3,000,190	2,488	19	178	9
Replace	3	3,622	2,998,228	828	144	1,356	9
Replace	5	6,111	2,973,823	487	356	3,221	9
Replace	10	12,540	2,893,090	231	1,802	13,688	8
Replace	20	24,421	2,930,947	120	6,582	44,823	7
Replace	30	37,372	2,771,735	74	15,751	105,055	7
Replace	50	63,953	2,587,013	40	43,437	287,229	7

Table B.20: The statistics of instructions that are replaced by replace algorithm for 3 MB Chain

Algorithm	Ratio	Replace Copy #	Replace Copy Length	Average Copy Length	Replace Insert #	Replace Insert Length	Average Insert Length
Replace	1	923	9,094,150	9,853	13	152	12
Replace	3	2,841	9,264,733	3,261	100	971	10
Replace	5	4,910	9,401,448	1,915	272	2,436	9
Replace	10	11,104	9,681,580	872	1,498	11,600	8
Replace	20	24,365	10,823,816	444	5,841	41,934	7
Replace	30	44,572	11,487,396	258	15,998	116,294	7
Replace	50	108,950	13,473,689	124	55,854	421,633	8

5 MB Series

Table B.21: The statistics of instructions that are applied by apply and replace algorithms for 5 MB Chain

Algorithm	Ratio	Apply Copy #	Apply Copy Length	Average Copy Length	Apply Insert #	Apply Insert Length	Average Insert Length
Apply	1	2,056	20,301,307	9,874	21	177	8
Apply	3	6,517	20,872,192	3,203	194	1,416	7
Apply	5	11,705	21,349,276	1,824	668	5,022	8
Apply	10	26,230	22,636,825	863	3,348	23,715	7
Apply	20	60,021	26,494,425	441	13,524	90,427	7
Apply	30	115,813	29,857,622	258	39,513	272,772	7
Apply	50	299,928	38,779,139	129	145,214	988,379	7
Replace	1	2,020	5,001,843	2,476	20	175	9
Replace	3	6,129	4,984,469	813	189	1,379	7
Replace	5	10,529	4,928,296	468	621	4,534	7
Replace	10	21,116	4,781,796	226	2,909	20,106	7
Replace	20	40,218	4,841,923	120	10,217	66,309	6
Replace	30	62,251	4,595,777	74	25,408	166,827	7
Replace	50	104,753	4,378,282	42	68,431	417,348	6

Table B.22: The statistics of instructions that are replaced by replace algorithm for 5 MB Chain

Algorithm	Ratio	Replace Copy #	Replace Copy Length	Average Copy Length	Replace Insert #	Replace Insert Length	Average Insert Length
Replace	1	1,530	15,150,427	9,902	21	177	8
Replace	3	4,760	15,418,413	3,239	134	907	7
Replace	5	8,447	15,590,497	1,846	466	3,459	7
Replace	10	18,625	16,058,945	862	2,437	16,975	7
Replace	20	40,562	17,911,101	442	9,116	62,001	7
Replace	30	74,232	19,096,048	257	25,688	182,414	7
Replace	50	174,830	22,488,110	129	86,075	604,806	7

10 MB Series

Table B.23: The statistics of instructions that are applied by apply and replace algorithms for 10 MB Chain

Algorithm	Ratio	Apply Copy #	Apply Copy Length	Average Copy Length	Apply Insert #	Apply Insert Length	Average Insert Length
Apply	1	4,218	40,585,107	9,622	47	319	7
Apply	3	13,288	41,729,101	3,140	452	3,317	7
Apply	5	22,758	42,875,049	1,884	1,116	7,007	6
Apply	10	52,288	45,261,238	866	6,032	40,536	7
Apply	20	118,763	53,002,031	446	25,290	163,557	6
Apply	30	225,914	59,784,426	265	72,290	455,292	6
Apply	50	589,458	77,711,428	132	276,926	1,758,046	6
Replace	1	4,139	9,992,251	2,414	47	319	7
Replace	3	12,502	9,952,752	796	436	3,166	7
Replace	5	20,498	9,907,516	483	1,052	6,556	6
Replace	10	42,248	9,567,454	226	5,258	34,586	7
Replace	20	80,156	9,698,180	121	19,356	120,832	6
Replace	30	122,021	9,245,949	76	46,811	277,397	6
Replace	50	208,567	8,857,226	42	131,600	741,580	6

Table B.24: The statistics of instructions that are replaced by replace algorithm for 10 MB Chain

Algorithm	Ratio	Replace Copy #	Replace Copy Length	Average Copy Length	Replace Insert #	Replace Insert Length	Average Insert Length
Replace	1	3,139	30,285,046	9,648	30	188	6
Replace	3	9,784	30,815,120	3,150	311	2,254	7
Replace	5	16,669	31,322,793	1,879	821	5,063	6
Replace	10	37,255	32,096,014	862	4,478	30,640	7
Replace	20	80,654	35,840,087	444	17,417	113,369	7
Replace	30	144,576	38,230,227	264	46,748	301,707	6
Replace	50	342,605	45,043,997	131	161,206	1,041,189	6