# Online De-duplication in a Log-Structured File System for Primary Storage

Technical Report UCSC-SSRC-11-03
May 2011

Stephanie N. Jones
snjones@cs.ucsc.edu

Storage Systems Research Center
Baskin School of Engineering
University of California, Santa Cruz
Santa Cruz, CA 95064
`http://www.ssrc.ucsc.edu/`

Filed as a Master's Thesis in the Computer Science Department of the University of California Santa Cruz in December 2010.

UNIVERSITY OF CALIFORNIA
SANTA CRUZ

# ONLINE DE-DUPLICATION IN A LOG-STRUCTURED FILE SYSTEM FOR PRIMARY STORAGE

A thesis submitted in partial satisfaction of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

**Stephanie N. Jones**

December 2010

The Thesis of Stephanie N. Jones
is approved:

Professor Darrell D. E. Long, Chair

Professor Ethan L. Miller

Dr. Kaladhar Voruganti

Tyrus Miller
Vice Provost and Dean of Graduate Studies

# Table of Contents

# List of Figures

# List of Tables

For Mom and Dad.

Thank you for supporting me even when I wasn't sure I could support myself.

## Acknowledgments

I would like to thank my advisor, Darrell Long, and the other members of my reading committee, Ethan Miller and Kaladhar Voruganti, for taking the time to review my thesis. I would also like to thank Christina Strong who took the time to read multiple versions of this thesis before it was sent to my reading committee. I want to thank many of the folks at NetApp who provided me with both filer crawls and CIFS traces, and took the time to help me with my implementation—even when it wasn't their job. I would not have all of the results in this thesis if it weren't for the tireless efforts of Garth Goodson, Kiran Srinivasan, Timothy Bisson, James Pitcairn-Hill, John Edwards, Blake Lewis, and Brian Pawlowski.

## Abstract

Online De-duplication in a Log-Structured File System for Primary Storage

by

Stephanie N. Jones

Data de-duplication is a term used to describe an algorithm or technique that eliminates duplicate copies of data from a storage system. Data de-duplication is commonly performed on secondary storage systems such as archival and backup storage. De-duplication techniques fall into two major categories based on when they de-duplicate data: *offline* and *online*. In an offline de-duplication scenario, file data is written to disk first and de-duplication happens at a later time. In an online de-duplication scenario, duplicate file data is eliminated before being written to disk. While data de-duplication can maximize storage utilization, the benefit comes at a cost. After data de-duplication is performed, a file written to disk sequentially could appear to be written randomly. This fragmentation of file data can result in decreased read performance due to increased disk seeks to read back file data.

Additional time delays in a storage system's write path and poor read performance prevent online de-duplication from being commonly applied to primary storage systems. The goal of this work is to maximize the amount of data read per seek with the smallest impact to de-duplication possible. In order to achieve this goal, I propose the use of *sequences*. A *sequence* is defined as a group of consecutive file data blocks in an incoming file system write request. A sequence is considered a duplicate if a group of consecutive data blocks are found to be in the same consecutive order on disk. By using sequences, de-duplicated file data will not be fragmented over the disk. This will allow a de-duplicated storage system to have disk read performance similar to a system without de-duplication. I offer the design and analysis of three algorithms used to perform sequence-based de-duplication. I show through the use of different data sets that it is possible to perform sequence-based de-duplication on archival data, static primary data and dynamic primary data. Finally, I present a full scale implementation using one of the three algorithms and report the algorithm's impact on de-duplication and disk seek activity.

# Chapter 1

# Introduction

Data de-duplication is a term used to describe an algorithm or technique that eliminates duplicate copies of data from a storage system. Data de-duplication is commonly performed on secondary storage systems such as archival and backup storage. De-duplication techniques fall into two major categories based on when they de-duplicate data: *offline* and *online*. In an offline de-duplication scenario, file data is written to disk first and de-duplication happens at a later time. In an online de-duplication scenario, duplicate file data is eliminated before being written to disk. In either scenario, duplicate data is removed from a storage system which frees up more space for new data. This maximization of storage space allows companies or organizations to go longer periods of time without needing to purchase new storage. Less storage hardware means lower costs in hardware and in operation. While data de-duplication can maximize storage utilization and reduce costs, the benefit comes at a price.

When a file is written to disk, the file system will find the optimal placement for the file. After data de-duplication is performed, the on-disk layout of a storage system can be dramatically different. The file system places file data in ways that optimize for reading files and/or writing files to disk. Data de-duplication eliminates duplicate data without regard to file placement. A sequentially written file can have the on-disk layout of a randomly written file after data de-duplication completes. This fragmentation of file data can result in decreased read performance due to increased disk seeks to read back file data. While any system that performs de-duplication can suffer from file fragmentation, performing online de-duplication on a primary storage system has an extra challenge. Because online de-duplication is performed before file data is written to disk, this puts extra computational overhead into a system's write path. It is important that the file system not block while de-duplication is being performed.

The goal of this work is to maximize the amount of data read per seek with the smallest impact to de-duplication possible. In order to achieve this goal, I propose the use of *sequences*. A *sequence* is a group of consecutive incoming data blocks. A duplicate sequence occurs when a group of consecutive data blocks are found to be in the same consecutive order on disk. By using sequences, de-duplicated file data will not be fragmented over the disk. This will allow a de-duplicated storage system to have disk read performance similar to a system without de-duplication.

I implemented sequences in a log-structured file system where every data block of a file is written as a 4KB block. For simplicity, I perform block-based chunking on these 4KB file data blocks and define sequences as spatially sequential file data blocks. This means that a sequence is, and can only be, expressed as a continuous range of file data blocks. The continuous range of file data blocks used to de-duplicate a sequence are not required to be from the same file. Although a sequence is required to be a sequential range of file data blocks, the de-duplication of a sequence is still done on the block level. De-duplication of sequences is done on a per-block basis because it reduces the metadata overhead caused by de-duplication techniques. When a data block is de-duplicated, the file allocation table is modified to point to the on-disk copy of the data block. This way there are no extra levels of indirection for reading back a file or for handling file overwrites.

De-duplicating using sequences is a double-edged sword. On one hand, a sequence is not de-duplicated unless there is an exact match found on disk for the consecutive group of blocks in the sequence. Longer sequences are more difficult to match than shorter ones. First, all of the data needs to be already stored on disk and a longer sequence requires more duplicate data. Second, not only does the data need to exist on disk, it needs to exist on disk in the same consecutive order as the sequence in question. However, on the other hand, the sequence length is a guarantee. If a sequence is de-duplicated, then if the file is read there is a known minimum number of blocks that will be co-located. This prevents a file read of de-duplicated data resulting in one disk seek per 4KB block in the file. If sequences are of sufficient length, it may be possible to amortize the cost of a disk seek by reading enough data.

In this thesis I give an investigation of the viability of a sequence-based de-duplication algorithm compared to a fixed-size block-based de-duplication algorithm. I show that duplicate file data does exist in consecutive groups and a sequence-based de-duplication algorithm is a viable option for de-duplication. I offer the design and analysis of three algorithms used to perform sequence-based de-duplication. I show through the use of different data sets that it is possible to perform sequence-based de-duplication on archival data, static primary data and dynamic primary data. Finally, I present a full scale implementation using one of the three algorithms and report the algorithm's impact on de-duplication and disk seek activity.

This thesis is organized as follows: Chapter 2 defines many of the terms used in this thesis and discusses other related work. Chapter 3 describes the data sets used for analysis and the algorithms developed to evaluate segment-based de-duplication. Chapter 4 explains the implementation of the best algorithm in a primary storage system. Section 5 discusses the experimental setup, workloads, and results from the implementation. Chapter 6 lists all of the compromises that were made in my implementation, summarizes the work presented in this thesis and concludes.

# Chapter 2

# Background

There is a delicate balance between maximizing the amount of de-duplication possible while minimizing the amount of extra metadata generated to keep track of de-duplicated data and data stored on disk. In this section, I define common terms and techniques and elaborate on their impacts on a system that implements them.

## 2.1 Definition of Terms

**Primary Storage**

Primary storage refers to an active storage system. Common examples would be mail servers or user home directories—any storage system that is accessed daily.

**Secondary Storage**

In contrast to primary storage, secondary storage refers to a storage system that is infrequently accessed. Common examples of secondary storage are storage archives and backup storage. Accesses to these systems only occur for the purpose of retrieving old data or making a backup. Accesses do not usually exceed once a day for a file.

De-duplication timing refers to the point in time a de-duplication algorithm is applied. The timing of the algorithm places restrictions on how much time it has to perform data de-duplication and the amount of knowledge the algorithm has about new file data. The two most common timing categories are: offline de-duplication and online de-duplication.

**Offline De-duplication**

If data de-duplication is performed offline, all data is written to the storage system first and de-duplication occurs at a later time. The largest benefit of this approach is that when de-duplication occurs the system has a static view of the entire file system and knows about all the data it has access to and can maximize de-duplication. Offline de-duplication performance can be slow because it may compare file data to all data stored on disk. Also, data written to the storage system must be batched until the next scheduled de-duplication time. This creates a delay between when the data is written and when space is reclaimed by eliminating duplicate data.

**Online De-duplication**

Online data de-duplication is performed as the data is being written to disk. The major upside to this approach is that this allows for immediate space reclamation. However, by performing de-duplication in the file system's write path there will be an increase in write latency—especially if a write is blocked until all duplicate file data is removed. The approach presented in this thesis is an online de-duplication technique.

Once the timing of data de-duplication has been decided, there are a number of existing techniques that have already been created. The three most frequently used are, as coined by Mandagere [21]: whole file hashing (WFH), sub file hashing (SFH), and delta encoding (DE).

**Whole File Hashing**

In a whole file hashing (WFH) approach, an entire file is sent to a hashing function. The hashing function is almost always cryptographic, typically MD5 or SHA-1. The cryptographic hash is used to find entire duplicate files. This approach is fast with low computation and low additional metadata overhead. It works well for full system backups when whole duplicate files are more common. However, the larger granularity of duplicate matching prevents it from matching two files that only differ by a byte of data.

**Sub File Hashing**

Sub file hashing (SFH) is aptly named. Whenever SFH is being applied, it means the file is broken into a number of smaller pieces before data de-duplication. The number of pieces depends on the type of SFH that is being applied. The two most common types of SFH are fixed-size chunking and variable-length chunking. In a fixed-size chunking scenario, a file is divided up into a number of fixed-size pieces called "chunks". In a variable-length chunking scheme, a file is broken up into "chunks" of varying length. Techniques such as Rabin fingerprinting [28] are used to determine "chunk boundaries". Each piece is passed to a cryptographic hash function (usually SHA-1 or MD5) to get the "chunk identifier". The chunk identifier is used to locate duplicate data. Both of these SFH techniques catch duplicate data at a finer granularity but at a price. In order to quickly find duplicate

| **System** | **Storage Type** | **Timing** | **WFH** | **SFH** | **DE** |
|---|---|---|---|---|---|
| DDE [13] | Primary | Offline | No | Fixed Block Size | No |
| IBM N Series [1] | Primary | Offline | No | Fixed Block Size | No |
| REBL [16] | Secondary | Offline | No | Variable Length | Yes |
| Deep Store [36] | Secondary | Online | Yes | Variable Length | Yes |
| TAPER [14] | Secondary | Online | Yes | Variable Length | Yes |
| Venti [27] | Secondary | Online | No | Fixed Block Size | No |
| DDFS [37] | Secondary | Online | No | Variable Length | No |
| Foundation [29] | Secondary | Online | No | Variable Length | No |
| Extreme Binning [5] | Secondary | Online | No | Variable Length | No |
| HYDRAstor [11] | Secondary | Online | No | Variable Length | No |
| Sparse Indexing [19] | Secondary | Online | No | Variable Length | No |
| dedupv1 [23] | Secondary | Online | No | Variable Length | No |

Table 2.1: Taxonomy of Various De-duplication Systems. Characteristics listed are Storage Type (Primary or Secondary storage), De-duplication Timing (Online or Offline), Whole File Hashing (WFH), Sub-File Hashing (SFH) and Delta Encoding (DE).

chunks, chunk identifiers must be also be kept readily accessible. The additional amount of additional metadata is dependent on the technique, but is unavoidable.

**Delta Encoding**

The term delta encoding (DE) is derived from the mathematical and scientific use of the delta symbol. In math and science, delta is used to measure the "change" or "rate of change" in an object. Delta encoding is used to express the difference between a target object and a source object. If block A is the source and block B is the target, the DE of B is the difference between A and B that is unique to B. The expression and storage of the difference depends on how delta encoding is applied. Normally it is used when SFH does not produce results but there is a strong enough similarity between two items/blocks/chunks that storing the difference would take less space than storing the non-duplicate block.

## 2.2 Related Work

Many secondary storage systems that use de-duplication are used as backup storage systems. These systems are usually written to on a daily basis in the form of a nightly backup. The benefit of a offline de-duplication is that the de-duplication algorithm has full knowledge of what is on the storage system [13, 16]. This static view of the entire file system makes performing de-duplication very easy. However, the amount of time to calculate and perform the optimal case of de-duplication must be done in the amount of time left before the next backup begins. So the more data that is written to the storage system, the less time the system will have to optimize the de-duplication before the next backup begins. Also, while de-duplication is being performed, the storage system is either completely inaccessible or allows read-only access.

On the other hand, online de-duplication can attempt to de-duplicate the incoming data before it is written to either primary or secondary storage [3, 5, 11, 14, 19, 23, 27, 29, 36, 37]. In the backup scenario, an online de-duplication approach can begin de-duplication right when the backup starts and has until the next one begins to finish. However, there are even more challenges with an online approach. Regardless of the technique being used, writing to disk is blocked until some specified unit of data has been de-duplicated. This act of blocking in the write path will destroy the write throughput of the backup. An online approach has to be able to perform WFH or SFH techniques, look up and compare hash values to discover duplicate data in real time. There is an unavoidable trade-off between the amount de-duplication possible and the amount of time allowed. The more thorough a data de-duplication technique is, the more it costs in time and space. Even simple schemes can be costly in space. Assume a 4TB storage system is being backed up for the first time. If a de-duplication technique divides the backup data into fixed-size 4KB chunks, there are $2^{30}$ chunks to be considered. Systems that perform fixed-size chunking, like Venti [27] or DDFS [37], use the 160-bit SHA-1 cryptographic hash function. If only half of the SHA-1 hashes are unique, any fixed-size de-duplication technique will generate 10GB of additional metadata to store *only* the cryptographic hashes for each 4KB chunk. While it is true that 10GB is a small fraction of 4TB, consider that the algorithm will need to store the on-disk location for each hash. All of this metadata is too much to fit into main memory along with the incoming write data which means that the information needs to be paged to and from disk.

Whole file hashing (WFH) techniques compute a cryptographic hash (such as MD5 or SHA-1) for every file in the storage system [14, 24, 36]. The hash of every unique cryptographic hash is stored in an additional data structure, typically a hash table. Duplicate matching is done at the granularity of whole files. The benefit of whole file hashing is quick matching with low overhead for additional metadata, since there is only one hash entry per file. The major detractor from this approach is the large granularity of duplicate detection. If two files differ by only a single byte of data, their cryptographic hashes are radically different, resulting in no duplicate elimination. This can cause a large amount of duplicate data between subsets of file data.

To combat the large granularity of whole file hashing, de-duplication systems can perform sub-file hashing (SFH). Instead of finding duplicates on the granularity of entire files, sub-file hashing tries to identify duplicate subsets of file data. There are two common methods for determining the subsets of file data to be matched: fixed-size chunking [3, 7, 9, 13, 15, 27] and variable-length chunking [5, 11, 14, 16, 19, 23, 25, 29, 36, 37]. Both techniques use "chunking" to help identify duplicate data. With respect to this thesis, chunking is a technique where a subset of file data is used to compute a content based hash. This hash, called a "chunk identifier" is used to identify the subset of file data and find duplicate data. The way chunking is performed varies between fixed-size chunking and variable-length chunking.

Fixed-size chunking is a straightforward technique. First, each file is broken into fixed-size chunks. The size of each fixed-size chunk can be the size of a file system data block or some size determined by the de-duplication technique. If a match is found, the entire fixed-size chunk of file data is not written to disk, and updates are made to the file's allocation table. Blocks that do not match any on-disk data are written to disk and their chunk identifiers are added to an additional metadata hash table. The benefit of using fixed-size chunking is that it identifies more duplicate data than whole file hashing. However, it can suffer from the same types of problems as whole file hashing. If two fixed-size data blocks differ by only a single byte, their hashes differ and no de-duplication takes place for that block. Also, for every unique fixed-size data block stored on disk, there is an entry in the metadata hash table. This means that the size required to store this additional metadata increases as the amount of unique data stored on disk increases. This problem is magnified in Opendedup [3] which implements a file system that performs online de-duplication in user space. The underlying file system sees each unique fixed-size block as a separate file creating even more metadata and, depending on the underlying file system, may cause file fragmentation even when no de-duplication can be performed.

To prevent the problem of fixed-size chunking missing almost duplicate data, variable-length chunking can be used. Instead of dividing a file up into fixed-size chunks, variable-length chunking uses an algorithm to establish the size of each chunk by calculating "chunk boundaries". The most common algorithm used to establish chunk boundaries is known as Rabin fingerprinting [28]. Using Rabin fingerprinting, the size of a chunk is determined by the content of the file. Parameters such as minimum and maximum chunk sizes are given to limit chunk sizes. After each chunk boundary is determined, the Rabin fingerprint generated for the chunk is used as the chunk identifier. After the whole file has been chunked, chunks are matched to on-disk data in the same manner as whole file hashing and fixed-size chunking. Just like in fixed-size chunking, the chunks that are known to be stored on disk are de-duplicated and metadata is updated. All chunks not found on disk are written to disk and their fingerprints are added to a metadata hash table. The benefit of variable length chunking is that it matches more content than fixed-size chunking. It can even avoid the problem that arises when data blocks differ by a single byte. However, this improved matching comes at an even higher cost than fixed-size chunking. While fixed-size chunking hash a predetermined fixed size for each chunk, variable-length chunking has to apply the Rabin fingerprinting algorithm in order to establish the chunk boundaries before de-duplication. Also, depending on how the minimum and maximum chunk sizes are set in the Rabin fingerprinting algorithm, variable length chunking can easily produce more chunks per file than a fixed-size chunking approach. With more chunks generated per file, the required overhead to store the fingerprints for unique data increases.

In all of the chunking and hashing approaches, there is the case where one file or block differs from another file or block by a single byte. In all of the previous approaches, this would result in failed matching. But instead of writing redundant data along with the unique data, why not just write the data that differs? This is the purpose behind delta encoding. Delta encoding will record the change between a source file and a target file [10, 14, 16, 36]. This way when there is a difference of a single byte, only the byte that differs is stored on disk. The benefit of this approach is that it can be used to store no changes when files are identical, or just the small differences between almost identical files. The detractors to delta encoding are that there needs to be some quantification of "almost identical", this approach can only be performed on a pair of files, and there needs to be a record of which file or chunk has been used to delta encode a file or chunk.

Sometimes none of the techniques discussed will find an exact duplicate or highly similar file block or entire file. If this occurs, some systems utilize compression techniques to achieve the best space savings possible for each piece of data in the system. In REBL [16], a file is broken into variable length chunks used for SFH. The duplicate chunks are de-duplicated and delta encoding is performed on the remaining chunks. The chunks that do not contain strong similarities to on-disk data are compressed. Deep Store [36] compressed blocks, using the *zlib* compression library, that had no on-disk duplicates nor a strong similarity to data already stored on-disk. In some systems, data compression was the only chosen method for eliminating duplicate data. Burrows, *et al.* [8] performed online intra-segment compression in the Sprite log-structured file system. They tested various compression techniques and found they could reduce the amount of space used by a factor of 2.

De-duplication techniques are not limited to eliminating duplicate data on persistent storage. They can be used to eliminate redundant copies of data in memory or reduce network bandwidth utilization. Virtual machines (VMs) allow multiple copies of operating systems, same or different, to run concurrently on a single physical machine. However, multiple VMs may page the same pieces of kernel code into main memory. This duplication of data in main memory reduces the amount left for data specific to each VM. Waldspurger [34] presents the idea of *content-based page sharing* between VMs. The contents of pages in memory are hashed and the hashes are stored in a hash table. When a new page is generated, if it matches the hash of a page already in memory, a byte comparison is performed. If the bytes match, the page is de-duplicated and marked copy-on-write.

LBFS [25] used variable length chunking to reduce network bandwidth utilization. When a user wanted to send file write data over the network, LBFS would use Rabin fingerprinting to break the file into variable length chunks. The file identifier and the fingerprints would be sent over the network to the destination node. The destination node would compare the list of fingerprints sent by the user to the ones generated by the on-disk copy at the destination node. Any fingerprints that did not match are returned to the client indicating which blocks need to be sent over the network. Shark [4] utilized the fundamental scheme from LBFS and applied it to distributed file systems. A client wanting to store a file would still use Rabin fingerprinting to break the file up into smaller chunks. However, in Shark, the client uses the SHA-1 hash of a chunk to identify the remote storage node. TAPER [14] uses de-duplication techniques to remove redundancies when synchronizing replicas. First, the replica node is identified and the source node and replica nodes compare hierarchical hash trees to eliminate as much of the namespace as possible. When a node in the hash tree doesn't match, the children of the source node are broken into content defined chunks by using Rabin fingerprinting. However, unlike LBFS, TAPER sends the SHA-1 hash of each content defined chunk to the replica node. Unique SHA-1 hashes are sent back to the source node. The unique chunks are broken into smaller sizes with their Rabin fingerprints and SHA-1 hashes sent for more duplicate matching. The smaller blocks that still don't match are delta encoded with highly similar files on the replica node. If there is any data left, it is compressed and sent to the replica node. This whole process ensures that no duplicate data is sent over the network. Mogul, *et al.* [24] use de-duplication techniques to eliminate duplicate data being sent by HTTP transfers. They generate the MD5 hash for each payload in a HTTP cache. When a client requests a payload from a server, the server first computes the MD5 hash for the payload and sends it to the client. If the MD5 hash matches a hash already in the HTTP cache, there is no need to send the payload. This technique is also helpful to reduce duplicate payload data in the HTTP cache for proxy nodes that store payloads before forwarding them to the clients or other proxies.

Both online and offline de-duplication techniques can suffer from the increase of metadata information. Waiting for part of a table to page into memory to look up a hash that may not be there destroys write throughput. There has been recent work that attempts to reduce the disk accesses required to retrieve information for matching duplicate data [5, 19, 37]. Some of these techniques admit that in order to increase disk throughput, some duplicate data will slip through the cracks. One of the popular techniques to immediately weed out non-matching hashes is via a Bloom filter. While Bloom filters can be used as a quick first pass to eliminate non-candidate hashes, they can also have false positives. This means a Bloom filter can come back saying that a hash may already exist in the system when it doesn't. Bloom filters work well so long as hashes aren't being deleted frequently which is ideal for secondary storage since the data is either immutable or very rarely changes. Disk defragmentation becomes more difficult after data has been de-duplicated. Before relocating a data block, the file system must make sure that all references to the block are updated as well. Macko, *et al.* [20] propose the use of *back references* to solve problems such as finding all file inodes in a de-duplicated system that reference a block being defragmented. These back references prevent the file system from having to iterate over the set of all files in the file system to check all block locations.

# Chapter 3

# Analysis

In this chapter, I show that de-duplication of files can be performed using sequences, and the same is true for file system request data. In Section 3.1, I present the data sets used to measure the impact of sequences on de-duplication when compared to a block-based approach. For each data set presented in Section 3.1, I give statistics such as the data set size, number of files in the data set, and the most common file types in each data set. In Section 3.2, I present each of the algorithms developed to de-duplicate using sequences. For each algorithm given in Section 3.2, I also present the impact of sequences on de-duplication for different sequence lengths. I also give statistics for the relative loss in de-duplication as the sequence lengths increase and the relative gain in de-duplication between algorithms. By the end of this cection I will have shown which of the algorithms presented is the best choice to be fully implemented in a real system.

## 3.1 Data Sets

To analyze each algorithm, I gathered a variety of data sets from previously published works. The data sets gathered fall into two major categories: crawls and traces. In the UCSC crawls, the entire files were copied to external storage or downloaded to a local directory. For the NetApp filer crawls, the files were read in, broken into 4KB fixed-size chunks and sent to a SHA-1 cryptographic hashing function. The SHA-1 hash of each block was written to an output file. The NetApp CIFS traces were gathered from CIFS traffic between clients and two storage servers over a 3 month period. The difference between the traces and crawls is that the crawls contain an entire snapshot of the file system state where the traces only contain file data for files accessed. Even then the traces only contain file data for the byte ranges accessed

| **Data Set** | **Size** | **Number of Files** | **Average File Size** |
|---|---|---|---|
| Internet Archive | 58 GB | 2,098,632 | 28 KB |
| Linux Kernels | 32 GB | 2,371,219 | 12 KB |
| Publication Archive | 15 GB | 70,914 | 227 KB |
| Sentinel Archive | 46 GB | 158,555 | 312 KB |

Table 3.1: The total size, number of files, and average file size for each Archival data set introduced.

| **Internet Archive** | **Linux Kernel** | **Publication Archive** | **Sentinel Archive** |
|---|---|---|---|
| .html | .h | .html | .ini |
| .jpg | .c | .pdf | .qxt |
| .gif | .s | .png | .doc |
| .asp | Makefile | gif | .jpg |
| .shtml | .txt | .cat | .xtg |
| .pdf | .in | .cfs | .bak |
| .php | README | .abt | .qxd |
| .cgi | .help | .DS_Store | .txt |
| .cfm | .kconfig | .pl | .tif |
| .pl | .defconfig | .ppd | .tmp |

Table 3.2: Top ten most frequently occurring file types in each archival data set. The file types are listed in descending order from most common to less common.

from a file. However, the traces do provide real read and write access patterns.

### 3.1.1 Archival Data

Before any full scale implementation of a sequential de-duplication algorithm, it is necessary to evaluate the algorithm's potential benefit. The most important first step is to determine if large collections of files contain not only duplicate data, but sequentially duplicate data. If breaking a file into sequences destroys duplicate data detection, there is no further gain in designing a more optimal algorithm. The first data sets used to test the potential feasibility of sequential de-duplication are comprised of archival data. There are four data sets used for preliminary testing: the Internet Archive, Linux Archive, Publication Archive, and the Sentinel Archive. The Internet Archive data set contains data from the Internet Archive's Wayback Machine. The Linux Archive contains multiple versions of the Linux kernel downloaded from the Linux Kernel Archives [2]. The Publication Archive contains copied data of compilations of papers from multiple conferences that was distributed via CDs. The Sentinel Archive contains archived data for previous issues of the Santa Cruz Sentinel newspaper. Statistics for each of these data sets are listed in Table 3.1. Table 3.2 lists the top ten file types of each data set.

| Data Set | Size | Number of Files | Average File Size |
|---|---|---|---|
| Engineering Scratch | 1.21 TB | 519,937 | 2.45 MB |
| Home Directories | 471.77 GB | 2,577,941 | 191 KB |
| Web Server | 673.44 GB | 4,929,606 | 143.24 KB |

Table 3.3: Static primary data set statistics. Given in this table are the sizes, number of files and the average file sizes for each data set.

| Engineering Scratch | Home Directories | Web Server |
|---|---|---|
| .c | .thpl | .html |
| .h | .c | .log |
| No Type | No Type | No Type |
| .thpl | .h | .dat |
| .cc | .o | .txt |
| .xml | .d | .rslt |
| .o | .pm | .cgi |
| .html | .t | .gif |
| .pm | .cdefs | .htm |
| .log | .pl | .thpl |

Table 3.4: Top ten most frequently occurring file types in the static primary data sets. The No Type entry encompasses all files in a data set that do not have a file extension. The .thpl file type is unique to NetApp storage and is used to test their filers. The file types are listed in descending order from most common to less common.

### 3.1.2 Static Primary Data

If sequentially duplicate file data exists in archival data, the next objective is to test static primary data. In this case, static primary data refers to a file system crawl of a primary storage system. The data sets used to test static primary data are: Engineering Scratch, Home Directories, and Web Server. All three data sets were collected from NetApp internal filers that serve as primary storage servers to the employees. The Engineering Scratch data set is used as a sort of free workspace for any NetApp employee, but is typically used by the software engineers. The Home Directories data set contains data from a subset of NetApp employee home directories. The Web Server data set contains data from one of the web servers that NetApp runs and maintains internally. Data set statistics can be found in Table 3.3 and listings of the top ten file types for each data set are in Table 3.4.

### 3.1.3 Dynamic Primary Data

If sequentially duplicate data exists in static primary data, then the last test needs to determine whether this pattern exists in dynamic primary data. Static primary data was in the form of a file system crawl with a static view of each file read. Dynamic primary data

| **Data Set** | **Write Request** | **Read Request** | **Combined Request** |
|---|---|---|---|
| Corporate CIFS Traces | 16.94 GB | 34.63 GB | 51.57 GB |
| Engineering CIFS Traces | 42.77 GB | 45.58 GB | 88.35 GB |

Table 3.5: Dynamic primary data set statistics. Given in this table is the amount of data sent by CIFS write requests, the amount of data received as a response from a CIFS read request and the combined amount of read and write data that exists in each trace.

| **Corporate CIFS Traces** | **Engineering CIFS Traces** |
|---|---|
| No Type | .thpl |
| .xls | .c |
| .html | .h |
| .doc | .gif |
| .zip | .jpg |
| .tmp | .dll |
| .htm | .htm |
| .jar | .tmp |
| .ppt | .dbo |
| .CNM | .o |

Table 3.6: Top ten most frequently occurring file types in the dynamic primary data sets. The No Type entry encompasses all files in a data set that do not have a file extension. The .thpl file type is unique to NetApp storage and is used to test their filers. The file types are listed in descending order from most common to less common.

comes in the form of Common Internet File System (CIFS) traces. A file system trace contains activity information for the period of time file system tracing was active. Activity information includes unique identifiers for each client accessing file data and file system request data. For the purposes of testing my algorithm, none of the client identification and only 100GB from the 1TB of CIFS file system requests were used. The choice to use only a subset of the requests was made to keep the test run times short and additional metadata generated by the algorithm at a manageable size. The two CIFS trace data sets used as dynamic primary data are: Corporate CIFS and Engineering CIFS. The Corporate CIFS trace contains information for all file system activity that involved a read, write, truncate or delete request to user home directories. The Engineering CIFS trace contains information for all file system activity involving read, write, truncate, or delete requests to scratch storage.

The read requests contain all of the data returned by the file system in response to the read request. The write requests contain the CIFS write request sent by a client to the file system. The maximum size of a CIFS write request is 64KB. The truncate requests contain the file offset to begin truncating. The delete requests specify the file to be deleted. For confidentiality, file names and file data were anonymized. All file names were anonymized via SHA-1 hashing. Read and write requests data were aligned on 4KB boundaries and broken

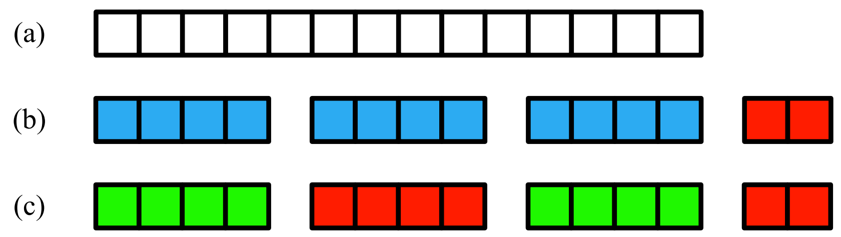


Figure 3.1: Example of the Naive algorithm de-duplicating a file. (a) The 56KB file is broken up into fourteen 4KB blocks. (b) The fourteen blocks are divided into groups of four. Each group in blue represents a sequence of blocks that will be looked up. The red group at the end is smaller than four blocks and is not de-duplicated. (c) The sequences in green have a duplicate copy on disk. The sequences in red are a unique grouping of blocks and are written to disk.

into 4KB block that were anonymized via SHA-1 hashing. All anonymized data was written to a text file for replaying the trace. The original CIFS requests were not used due to privacy concerns. Read and write request information for both traces is given in Table 3.5. The top ten file types for both traces is listed in Table 3.6.

## 3.2 Algorithms

### 3.2.1 Naive Algorithm

The Naive algorithm was developed to perform proof of concept tests using the data sets previously discussed. The sequence length was preset to some fixed number of blocks. The example shown in Figure 3.1 assumes the sequence length to be four 4KB blocks (or 16KB) per sequence. The Naive algorithm was applied to all three of the data sets: archival data, static primary data, and dynamic primary data.

Every file in each archival data set is sent to the Naive algorithm for de-duplication. Before sending a file to the Naive algorithm, the file is divided into 4KB-aligned blocks then hashed using SHA-1. A list of SHA-1 hash and block size pairs for each 4KB-aligned block in the file are sent to the Naive algorithm. The algorithm groups the SHA-1 hashes into sequences based on the specified sequence length as shown in Figure 3.1(b). In the case of the example figure, the last two red blocks do not meet the required sequence length and are automatically written to disk. The Naive algorithm looks up each sequence in a local hash table to attempt to find an on-disk duplicate. Sequences found on disk (green in Figure 3.1(c)) are de-duplicated and sequences not found on disk (red in Figure 3.1(c)) are written to disk and stored in the local hash table. If the Naive algorithm uses a sequence length of 1, it behaves the same as a

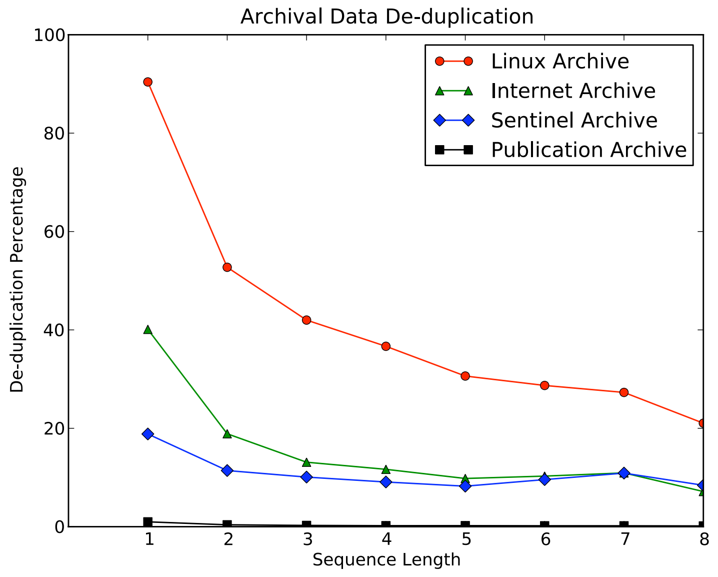


Figure 3.2: Impact of the Naive algorithm on the de-duplication of the archival data sets.

| | **Linux** | **Internet** | **Sentinel** | **Publication** |
|---|---|---|---|---|
| SL1 | 90.41% | 40.11% | 18.84% | 0.99% |
| SL2 | 52.73% | 18.89% | 11.41% | 0.39% |
| SL8 | 21.04% | 7.15% | 8.42% | 0.16% |

Table 3.7: Lists the de-duplication impact of the Naive algorithm for each of the archival data sets. The sequence length of 1 (SL1) is the best de-duplication possible using fixed-size blocks for each data set. The sequence length of 2 (SL2) is the de-duplication achieved when grouping 2 blocks together into a sequence. The last row (SL8) is the de-duplication achieved when 8 blocks are grouped into a sequence.

fixed-size de-duplication algorithm. However, if the algorithm uses any sequence length greater than 1, it is possible for a block to be duplicated on disk since the algorithm only de-duplicates blocks that make up a sequence. Any block not in a sequence is written to disk and the MD5 hash of its content is added to the hash table even if the block already exists on disk. The results of applying the Naive algorithm to the archival data sets are shown in Figure 3.2.

The x-axis of Figure 3.2 is the sequence length used by the Naive algorithm for de-duplicating each data set. The sequence length was tested over single increments from a sequence length of 1 block (4KB) to a sequence length of 8 blocks (32KB). The y-axis of Figure 3.2 is the de-duplication percentage achieved by the Naive algorithm for each sequence length used. With respect to this graph, the higher the point is on the y-axis the better the

|  | **Linux** | **Internet** | **Sentinel** | **Publication** |
|---|---|---|---|---|
| SL2/SL1 | 58.33% | 47.10% | 60.57% | 39.20% |
| SL8/SL1 | 23.27% | 17.82% | 44.67% | 15.71% |

Table 3.8: Lists the normalized de-duplication from Table 3.7. All normalization in this table is with respect to the Naive algorithm with a sequence length of 1 (SL1). The format for the Naive algorithm with a sequence length of 2 normalized with respect to a sequence length of 1 is: SL2/SL1.

result. For example, the Linux Archive has a de-duplication percentage of 90.41% for a sequence length of 1. This means that if the Naive algorithm uses a sequence length of one block, it can de-duplicate 90.41% of the data set requiring only 9.59% of the original data set to be written to disk. Figure 3.2 shows that duplicate data does exist in sequences. An astute eye will notice that there is a peak in the graph at a sequence length of 7. This is caused by files with trailing blocks that are not de-duplicated with a sequence length of 7, but are caught when using a sequence length of 8.

Table 3.7 has the de-duplication percentages for each of the data sets using sequence lengths of 1 (SL1), 2 (SL2) and 8 (SL8). Table 3.8 has normalized percentages for each of the data sets. The normalization is represented as a fraction where the numerator is the sequence length (SL) being listed with respect to the sequence length given in the denominator. So, in Table 3.8 the SL2/SL1 row lists the de-duplication percentage of the sequence length of 2 with respect to the de-duplication percentage of the sequence length of 1 for all 4 data sets. The 60.57% in the Sentinel column means that 60.57% of the data de-duplicated using a sequence length of 1 can also be de-duplicated using a sequence length of 2. There is a trivial amount of de-duplication in the Publication Archive, shown in Table 3.7, due to the compressed PDF files. Table 3.8 shows the loss in de-duplication for the Linux Archive when using a sequence of 2 with respect to the de-duplication achieved using a sequence of 1. This dramatic drop is caused by the small average file size in the Linux kernel. Figure 3.2 demonstrates the viability of a sequence-based de-duplication algorithm for data sets that have sufficiently duplicate data. As is shown in the graph, the average file size and the file types have an impact on de-duplication using sequences.

The application of the Naive algorithm to the static primary data sets is exactly the same as the archival data sets. Each file is broken into 4KB-aligned blocks and the block data is hashed using SHA-1. The list of SHA-1 hash and block size pairs is sent to the algorithm. It de-duplicates each file as shown in Figure 3.1. The results of applying the Naive algorithm to the primary static data sets is shown in Figure 3.3. The x-axis of Figure 3.3 is the sequence lengths used by the Naive algorithm for de-duplicating each data set. Each tick on the x-axis

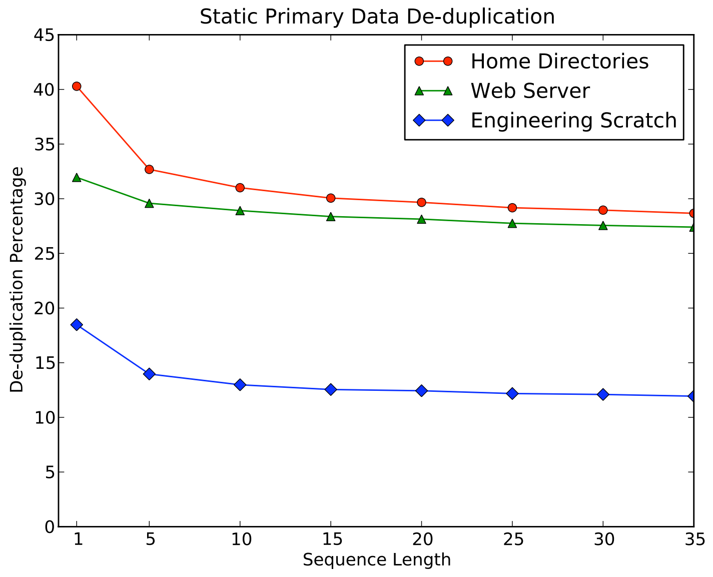


Figure 3.3: Impact of the Naive algorithm on the de-duplication of the static primary data sets.

| | **Home Directories** | **Web Server** | **Engineering Scratch** |
|---|---|---|---|
| SL1 | 40.29% | 31.95% | 18.47% |
| SL5 | 32.68% | 29.59% | 13.97% |
| SL35 | 28.67% | 27.40% | 11.94% |

Table 3.9: Lists the de-duplication for each of the static primary data sets. The sequence length of 1 (SL1) is the best de-duplication possible for each data set. The sequence length of 5 (SL5) is the de-duplication achieved when grouping 5 blocks together into a sequence. The last row (SL35) is the de-duplication achieved when 35 blocks are grouped into a sequence.

indicates a sequence length tested. The first length is a sequence length of 1 block (4KB). The following sequences were in multiples of 5 from a sequence length of 5 blocks (20KB) to a sequence length of 35 blocks (140KB). De-duplication was tested in 5 block increments to see if a large enough sequence length would result in no de-duplication for a data set. The y-axis of Figure 3.3 is the de-duplication percentage achieved by the Naive algorithm each sequence length used. With respect to this graph, the higher the point is on the y-axis the better the result. For example, the Home Directories data set has a de-duplication percentage of 40.29%. This means that if the Naive algorithm uses a sequence length of one block, it can de-duplicate 40.29% of the data set, requiring only 59.71% of the original data set to be written to disk.

Figure 3.3 shows that duplicate data exists in sequences in primary data sets. While

|  | **Home Directories** | **Web Server** | **Engineering Scratch** |
|---|---|---|---|
| SL5/SL1 | 81.12% | 92.59% | 75.61% |
| SL35/SL1 | 71.15% | 85.76% | 64.63% |

Table 3.10: Lists the normalized de-duplication from Table 3.9. All normalization in this table is with respect to the Naive algorithm with a sequence length of 1 (SL1). The format for the Naive algorithm with a sequence length of 5 normalized with respect to a sequence length of 1 is: SL5/SL1.

there is an early rapid drop for all data sets from a sequence length of 1 to a sequence length of 5, the remainder of the graph plateaus. Unfortunately, there is no data available to suggest if there is a sharp drop from a sequence length of 1 to a sequence length of 2 (similar to the drop in Figure 3.2), or if the decline is uniform from a sequence length of 1 to 2, 2 to 3, 3 to 4 and 4 to 5. The importance of Figure 3.3 is the plateau after the sequence length of 5. This tells us two significant things about primary storage file data. First, primary storage file data has significant duplicate data. Second, the duplicate data can be found using a sequence-based de-duplication algorithm.

Table 3.9 has the de-duplication percentages for each of the data sets using sequence lengths of 1 (SL1), 5 (SL5) and 35 (SL35). Table 3.10 has normalized percentages for each of the data sets. The normalization is represented as a fraction where the numerator is the sequence length (SL) being listed with respect to the sequence length given in the denominator. So, in Table 3.10 the SL5/SL1 row lists the de-duplication percentage of the sequence length of 5 with respect to the de-duplication percentage of the sequence length of 1 for all 3 data sets. The 81.12% in the Home Directories column means that 81.12% of the data de-duplicated using a sequence length of 1 can also be de-duplicated using a sequence length of 5. So now that it's known that primary storage files contain non-trivial amounts of sequentially duplicate data, what about the file system requests that create the files?

To measure the impact of sequences on file system requests, the Naive algorithm is applied to the dynamic primary data set. Because this data set is made up of CIFS trace data the Naive algorithm has to have added functionality. The Naive algorithm was modified to handle file truncates and deletes. Also, the amount of write data in both trace files, shown in Table 3.5, is at best only half of the available file data conveyed in each trace. The file data sent to the Naive algorithm is comprised of data from both read and write requests. As was explained in Section 3.1.3, the request data has already been broken into 4KB-aligned blocks which are replaced by the SHA-1 hash of the block's data. The Naive algorithm groups SHA-1 hashes into sequences based on the preset sequence length and de-duplicates the request in the same manner as shown in Figure 3.1. The result of applying the Naive algorithm to the

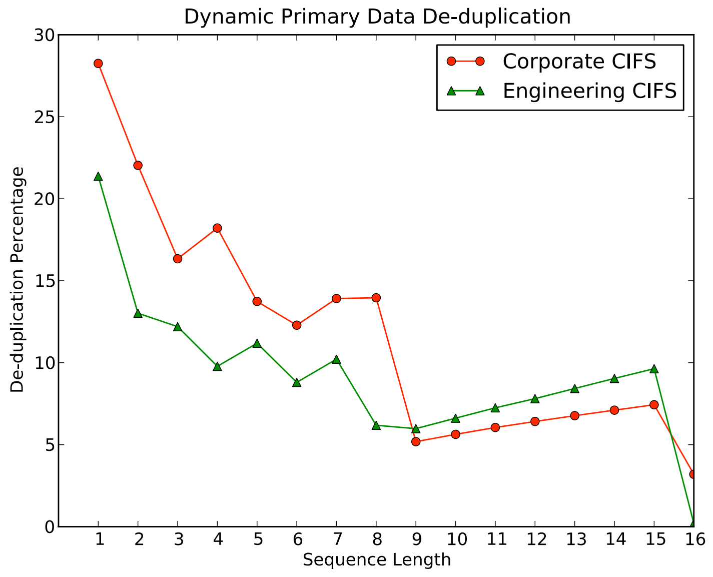


Figure 3.4: Impact of the Naive algorithm on the de-duplication of the dynamic primary data sets.

dynamic primary data sets is shown in Figure 3.4.

The x-axis of Figure 3.4 is the sequence lengths used by the Naive algorithm for de-duplicating the dynamic primary data sets. The sequence length was tested over single increments from a sequence length of 1 block (4KB) to a sequence length of 16 blocks (64KB or the maximum amount of data in a CIFS request). The y-axis of Figure 3.4 is the de-duplication percentage achieved by the Naive algorithm for each sequence length used. With respect to this graph, the higher a point is on the y-axis, the better the result. For example, the Corporate CIFS trace has a de-duplication percentage of 28.25% for a sequence length of 1. This means that if the Naive algorithm uses a sequence length of one block, it can de-duplicate 28.25% of the data set at the request level, requiring only 71.75% of the request data to be written to disk.

Figure 3.4 shows that duplicate data exists in groups even at the request level of a primary storage system. Unlike Figure 3.2, the peaks in this graph are obvious. They are artifacts caused by the Naive algorithm not de-duplicating files that are smaller than the sequence length or the remaining blocks of a file that are insufficient to make up a new sequence. Figure 3.5 shows how the Naive algorithm with a sequence length of 5 blocks will split a 10 block (40KB) request into two perfectly sized sequences before attempting de-duplication. Figure 3.6 shows how the Naive algorithm with a sequence length of 6 blocks will split that same request

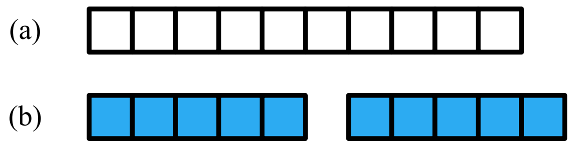


Figure 3.5: An example of how the Naive algorithm, using a sequence length of 5, breaks a 10 4KB block file into two sequences of 5 blocks.

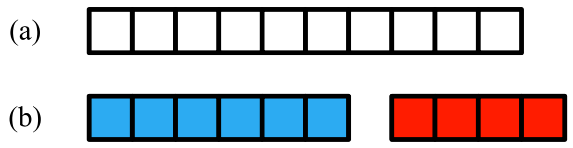


Figure 3.6: An example of how the Naive algorithm, using a sequence length of 6, breaks the same 10 4KB block file from Figure 3.5 into one sequence and writes the last 4 blocks to disk.

into one sequence of 6 blocks, but not de-duplicate the remaining 4 blocks. If the entire 10 block request could be de-duplicated, Figure 3.6 loses out on 20KB of duplicate data. However, if the Naive algorithm had a sequence length of 7, the request would be broken into one sequence length of 7 blocks and the remaining 3 blocks would be written to disk. While this is not better than the scenario in Figure 3.5, it does de-duplicate one more block than the scenario in Figure 3.6. It is this kind of situation that causes the peaks. Figure 3.4 shows that de-duplication can be performed on the request level. The peaks in the graph also indicate that if a more intelligent algorithm was applied to the dynamic primary data sets, the de-duplication percentage would be higher for longer sequence lengths.

Table 3.11 lists the de-duplication percentage for each of the dynamic primary data sets for sequence lengths of 1 (SL1), 2 (SL2), 7 (SL7), 8 (SL8), 9 (SL9), 15 (SL15) and 16 (SL16). Each of the values in the table is the same as its point on the graph in Figure 3.4. Table 3.12 has normalized percentages for each of the data sets. The normalization is done in the same way as Table 3.10. The row of SL2/SL1 is the de-duplication percentage of the sequence length of 2 with respect to the de-duplication achieved when using a sequence length of 1. The 78.00% in the Corporate CIFS column means that 78.00% of the data de-duplicated when using a sequence length of 1 is also de-duplicated when using a sequence length of 2.

Even though there is a drop from sequence length of 1 to sequence length of 2, applying the Naive algorithm shows good results up until a sequence length of 9 for the Corporate CIFS trace and a sequence length of 8 for the Engineering CIFS trace. Figure 3.6 also shows the reason for the sharp decline in both traces around the sequence length of 8. If the sequence length is larger than half of the request size one only sequence can be created. So if the largest

| | **Corporate CIFS** | **Engineering CIFS** |
|---|---|---|
| SL1 | 28.25% | 21.38% |
| SL2 | 22.03% | 13.02% |
| SL7 | 13.91% | 10.22% |
| SL8 | 13.96% | 6.18% |
| SL9 | 5.19% | 5.98% |
| SL15 | 7.44% | 9.63% |
| SL16 | 3.20% | 0.20% |

Table 3.11: Lists some of the achieved de-duplication for the dynamic primary data sets using the Naive algorithm with different sequence lengths. The best de-duplication possible for the Naive algorithm is achieved by using a sequence length of 1 (SL1).

| | **Corporate CIFS** | **Engineering CIFS** |
|---|---|---|
| SL2/SL1 | 78.00% | 60.91% |
| SL7/SL1 | 49.25% | 47.79% |
| SL8/SL1 | 49.42% | 28.93% |

Table 3.12: Lists the normalized de-duplication achieved from Table 3.11. All de-duplication in this table is normalized with respect to the de-duplication achieved using the Naive algorithm with a sequence length of 1 (SL1).

CIFS request has 16 4KB blocks, using a sequence length of 9 can result in de-duplicating 9 blocks and flushing 7 blocks to disk. A sequence length of 10 can result in de-duplicating 10 blocks and flushing 6 to disk. The Engineering CIFS has almost no CIFS requests containing 16 4KB blocks, which is why the de-duplication savings decreases noticeably at 7 instead of 8. In the Naive algorithm, the sequence is created using the first number of blocks in the request defined by the sequence length. In Figure 3.6, if blocks $1-6$ cannot be de-duplicated, the entire request is written to disk, even if blocks $2-7$ already exist on-disk sequentially. It is this situation that led to the Sliding Window algorithm.

### 3.2.2 Sliding Window

In order to solve one of the main problems of the Naive algorithm, the Sliding Window algorithm will shift a sequence over by one block if the first sequence doesn't exist on disk. Only the Corporate CIFS trace from the dynamic primary data sets was used to test this algorithm. The Corporate CIFS trace is comprised of real file system requests, which is the long-term target for this work, and it consistently demonstrated higher levels of de-duplication compared to the Engineering CIFS trace. Figure 3.7 shows how the Sliding Window algorithm is applied to the Corporate CIFS trace read and write requests using a sequence length of 4. The Sliding Window algorithm creates the first sequence of 4 blocks and finds a duplicate copy

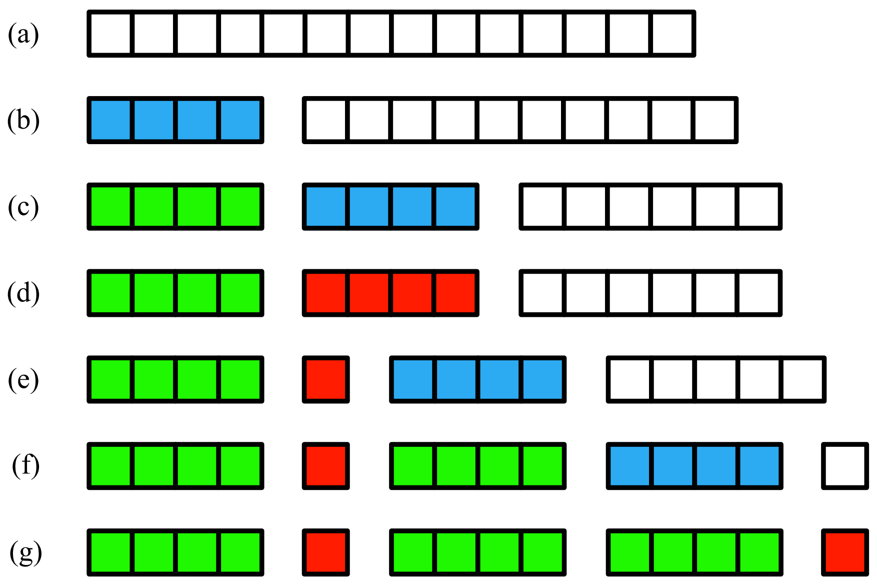


Figure 3.7: Example of the Sliding Window algorithm de-duplicating a file using sequences of four 4KB blocks. (a) The 56KB file is broken up into fourteen 4KB blocks. (b) The algorithm looks up the first sequence of four blocks. (c) The algorithm found the sequence on disk, de-duplicates the sequence, and looks up the next sequence. (d) The sequence is unique and not already on disk. (e) The algorithm marks the first block of the unique sequence to be written to disk. It checks looks up the next 4 blocks. (f) See (c). (g) There is only one 4KB block left in the file. The single block, unable to form a sequence, is written to disk.

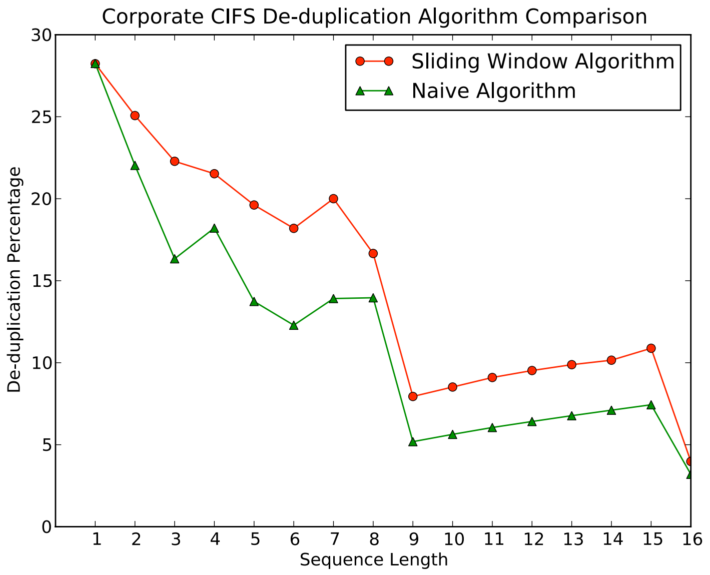


Figure 3.8: Impact of the Sliding Window algorithm on the Corporate CIFS trace. The de-duplication of the Corporate CIFS trace using the Naive Algorithm is included as a second line on the graph for comparison.

of the sequence on disk. The algorithm groups the next 4 blocks into a sequence but cannot find a duplicate sequence on disk. Instead of writing the entire sequence to disk, like the Naive algorithm, the Sliding Window algorithm marks the first block of the failed sequence to be written to disk. It then creates a new sequence and finds an on-disk copy of the new sequence. At the end, there is only one block left. The single block is insufficient to create a new sequence and is marked to be flushed to disk. Once the algorithm reaches the end of the request, all blocks that have been marked are written to disk. If the Sliding Window algorithm uses a sequence length of 1, it behaves the same as a fixed-size de-duplication algorithm. However, if the algorithm uses any sequence length greater than 1, it is possible for a block to be duplicated on disk since the algorithm only de-duplicates blocks that make up a sequence. Any block not in a sequence is written to disk and the MD5 hash of its content is added to the hash table even if the block already exists on disk.

Figure 3.8 compares the de-duplication achieved between the Naive algorithm and Sliding Window algorithm when applied to the Corporate CIFS trace. The peak at the sequence length of 7 and the slow rise starting at the sequence length of 9 is still explained by Figures 3.5 and 3.6. Table 3.13 lists the de-duplication percentage of the Corporate CIFS trace for both the Naive and Sliding Window algorithms using the sequence lengths of 1 to 8. Because the

|  | **Sliding Window** | **Naive** |
|---|---|---|
| SL1 | 28.25% | 28.25% |
| SL2 | 25.07% | 22.03% |
| SL3 | 22.28% | 16.34% |
| SL4 | 21.53% | 18.21% |
| SL5 | 19.62% | 13.74% |
| SL6 | 18.19% | 12.29% |
| SL7 | 20.00% | 13.91% |
| SL8 | 16.66% | 13.96% |

Table 3.13: Lists some of the achieved de-duplication for Corporate CIFS using the Naive algorithm and the Sliding Window algorithm.

|  | **Sliding Window** | **Naive** |
|---|---|---|
| SL2/SL1 | 88.74% | 77.98% |
| SL3/SL1 | 78.87% | 57.84% |
| SL4/SL1 | 76.21% | 64.46% |
| SL5/SL1 | 69.45% | 48.53% |
| SL6/SL1 | 64.39% | 43.50% |
| SL7/SL1 | 70.80% | 49.24% |
| SL8/SL1 | 58.97% | 49.42% |

Table 3.14: Lists the normalized de-duplication achieved in Table 3.13 for both the Naive and Sliding Window algorithms.

sequence length of 1 is the same as a 4KB fixed-size chunking technique, the de-duplication percentage is the same. The values in the table match to their respective points on the graph in Figure 3.8. Table 3.14 has the normalized percentages for both algorithms listed in Table 3.13. The normalization is done in the same way as Table 3.12. The row of SL2/SL1 is the de-duplication percentage of sequence length 2 with respect to the de-duplication achieved when using a sequence length of 1. The 88.74% in the Sliding Window column means that 88.74% of the data de-duplicated by a sequence length of 1 is also de-duplicated when using a sequence length of 2. This is a 10.74% increase in de-duplication over the Naive algorithm.

Table 3.15 also contains normalized percentages but instead of normalizing with respect to a sequence length, the data has been normalized with respect to the Naive algorithm at each sequence length listed. The row of SL2 is the de-duplication percentage of the Sliding Window algorithm using a sequence length of 2 with respect to the de-duplication percentage of the Naive algorithm using a sequence length of 2. The SL2 entry of 113.80% means that the Sliding Window algorithm can de-duplication 113.80% of the data de-duplicated by the Naive algorithm when they both use a sequence length of 2. Another way to say this is that the Naive algorithm can de-duplicate 87.87% of the data de-duplicated by the Sliding Window if they

|  | **SW/N** |
|---|---|
| SL2 | 113.80% |
| SL3 | 136.36% |
| SL4 | 118.23% |
| SL5 | 143.11% |
| SL6 | 148.02% |
| SL7 | 143.78% |
| SL8 | 119.34% |

Table 3.15: Lists the Sliding Window (SW) algorithm's achieved de-duplication from Table 3.14 when normalized to the Naive (N) algorithm's de-duplication.

both use a sequence length of 2. The improvement of the Sliding Window algorithm over the Naive algorithm is notable. Another point of interest in Figure 3.8 is the upward trend starting at the sequence length of 9. This upward trend indicates that there is a non-trivial amount of requests that can be de-duplicated entirely if the algorithm could use the extra leftover blocks instead of immediately flushing them to disk. This was the motivation behind the Minimum Threshold algorithm.

### 3.2.3 Minimum Threshold

The Minimum Threshold algorithm uses the sequence length as the smallest sequence size allowed. In this case the sequence length is referred to as the *threshold*. A threshold of 4 means there must be at least 4 sequential blocks in the sequence for it be eligible for de-duplication. Figure 3.9 shows how the Minimum Threshold is applied to the Corporate CIFS trace read and write requests using a threshold of 4. The Minimum Threshold algorithm groups the first 4 blocks into a sequence, looks the sequence up in a local hash table, and discovers that there is an on-disk copy of the sequence. The algorithm appends the next block to the sequence and looks up the new 5 block sequence, which also happens to exist on disk. The algorithm will continue to add one block to the sequence and look up the new sequence until the algorithm either uses all of the blocks left in the request, or the last block appended created a sequence that does not already exist on disk as shown in Figure 3.9(g). If the latter event happens, the last appended block is removed from the sequence reverting it to a sequence known to have an on-disk duplicate. The algorithm groups the next 4 blocks into a sequence starting from the block previously removed. If the newest sequence doesn't have a duplicate copy on disk, shown in Figure 3.9(i), the algorithm marks the first block of the sequence to be written to disk and groups the next 4 blocks into a sequence starting from the second block in unique sequence. If the newest sequence is found to have a duplicate copy stored on disk, as in Figure 3.9(k), the algorithm continues to add one block to the sequence until there are no blocks remaining in the

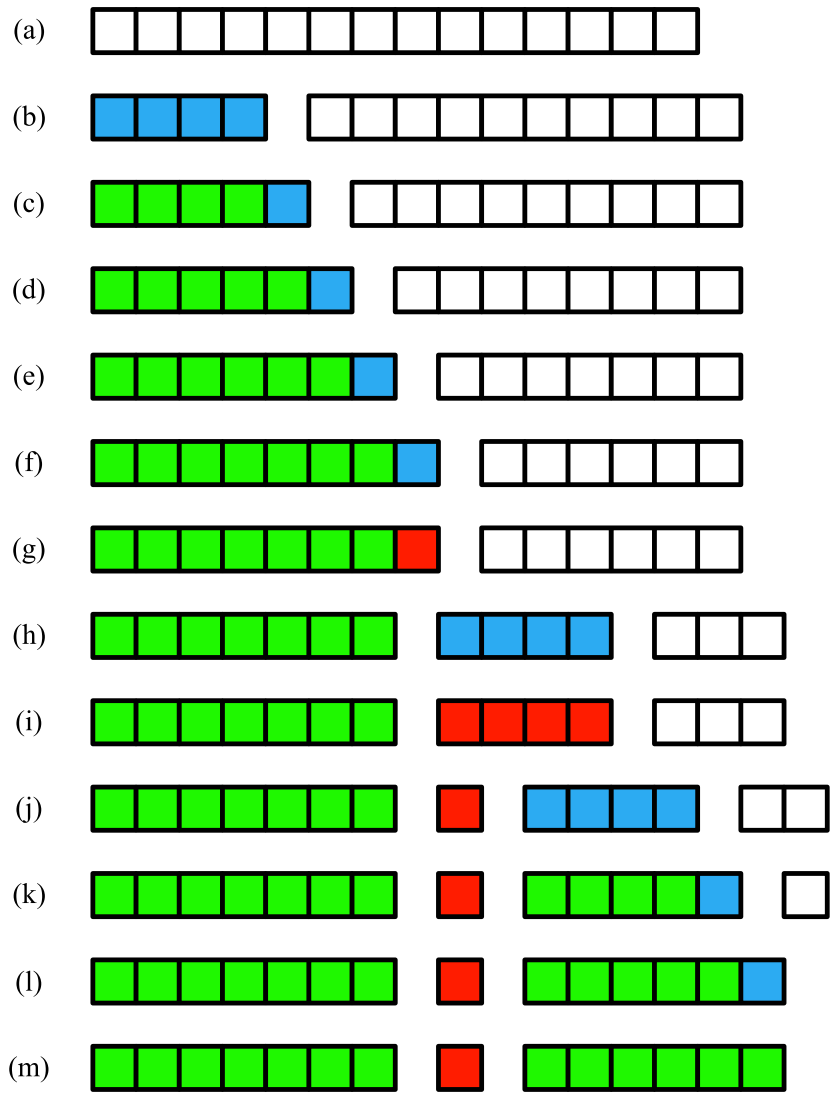


Figure 3.9: Example of the Minimum Threshold algorithm de-duplicating an incoming CIFS request using a minimum threshold of 4. (a, b) Same as the Sliding Window algorithm. (c) There is an on-disk copy of the sequence. Add the next block to the sequence and look for an on-disk duplicate. (d) There is an on-disk copy of the new sequence. Add the next block to the sequence and try again. (e, f) See (d). (g) There is no on-disk copy of any sequence that contains the last block added. Remove the block added in (f) and de-duplicate the sequence before the block was added. (h) The algorithm creates a new sequence starting at the block previously removed in (g). (i) There is no duplicate sequence on disk. (j) Same as the Sliding Window algorithm. (k) See (c). (l) See (d). (m) There are no more blocks left in the CIFS request. All blocks in green are de-duplicated and the block in red is written to disk.

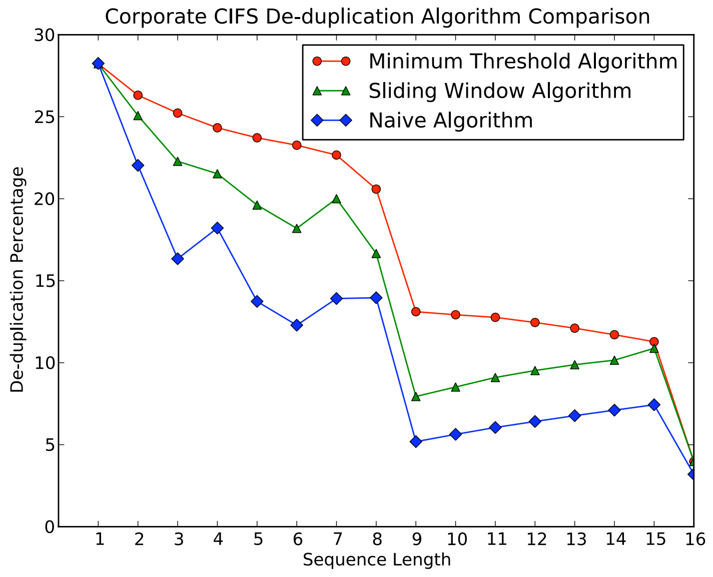


Figure 3.10: Impact of the Minimum Threshold algorithm on the Corporate CIFS trace. The de-duplication information of the Corporate CIFS trace using the Naive and Sliding Window algorithms have been included in the graph for comparison.

request, shown in Figure 3.9(m), or the latest block addition results in a sequence that is not found on disk as was shown in Figure 3.9(g). Just as in the Sliding Window algorithm, once the Minimum Threshold reaches the end of the request, all blocks that were marked as not being in a sequence are written to disk. If the Minimum threshold algorithm uses a threshold of 1, it behaves the same as a fixed-size de-duplication algorithm if data cannot be de-duplicated using sequences. However, if the algorithm uses any threshold greater than 1, it is possible for a block to be duplicated on disk since the algorithm only de-duplicates blocks that make up a sequence. Any block not in a sequence is written to disk and the MD5 hash of its content is added to the hash table even if the block already exists on disk.

Figure 3.10 compares the achieved de-duplication between the Minimum Threshold, Sliding Window and Naive algorithms. The x-axis of Figure 3.10 is the sequence lengths used by the Minimum Threshold, Sliding Window and Naive algorithms for de-duplicating the Corporate CIFS trace data. The sequence length was tested over single increments from a sequence length of 1 block (4KB) to a sequence length of 16 blocks (64KB or the maximum amount of data in a CIFS request). The y-axis of Figure 3.10 is the de-duplication percentage achieved by the Minimum Threshold, Sliding Window and Naive algorithms for each sequence length used. With respect to this graph, the higher a point is on the y-axis, the better the

| | Minimum Threshold | Sliding Window | Naive Algorithm |
|---|---|---|---|
| SL1 | 28.25% | 28.25% | 28.25% |
| SL2 | 26.31% | 25.07% | 22.03% |
| SL3 | 25.22% | 22.28% | 16.34% |
| SL4 | 24.33% | 21.53% | 18.21% |
| SL5 | 23.72% | 19.62% | 13.74% |
| SL6 | 23.26% | 18.19% | 12.29% |
| SL7 | 22.66% | 20.00% | 13.91% |
| SL8 | 20.58% | 16.66% | 13.96% |

Table 3.16: Lists the de-duplication achieved by all 3 algorithms: Naive, Sliding Window, and Minimum Threshold.

| | Minimum Threshold | Sliding Window | Naive Algorithm |
|---|---|---|---|
| SL2/SL1 | 93.13% | 88.74% | 77.98% |
| SL3/SL1 | 89.27% | 78.87% | 57.84% |
| SL4/SL1 | 86.12% | 76.21% | 64.46% |
| SL5/SL1 | 83.96% | 69.45% | 48.53% |
| SL6/SL1 | 82.34% | 64.39% | 43.50% |
| SL7/SL1 | 80.21% | 70.80% | 49.24% |
| SL8/SL1 | 72.85% | 58.97% | 49.42% |

Table 3.17: Lists the normalized de-duplication for some of the rows in Table 3.16. All de-duplication results have been normalized using their respective algorithms.

result. For example, the Corporate CIFS trace has a de-duplication percentage of 26.31% for a threshold of 2 when using the Minimum Threshold. This is a 1.24% improvement over the Sliding Window's de-duplication percentage using a sequence length of 2 and a 4.28% improvement over the Naive Algorithm's de-duplication percentage also using a sequence length of 2.

Figure 3.10 illustrates all of the duplicate data missed by both the Sliding Window and Naive algorithm. The Minimum Threshold algorithm outperforms the previous two algorithms except when using a threshold of 1 or a threshold of 16. The greedy nature of the Minimum Threshold algorithm allows it to outperform both the Sliding Window and Naive algorithms. By continuing to match blocks past the minimum threshold, it has evened out the peaks in the de-duplication graph and more than doubled the de-duplication at the threshold of 9 over the Naive algorithm.

Table 3.16 lists the de-duplication percentages of the Corporate CIFS trace for all of the algorithms (Naive, Sliding Window and Minimum Threshold) using sequence lengths, or thresholds, of 1 to 8. Again, because the sequence length of 1 is the same as any 4KB fixed-size chunking technique, the percentages for all 3 algorithms is the same. The values in the

| | **MT/SW** | **MT/N** |
|---|---|---|
| SL2 | 104.95% | 119.43% |
| SL3 | 113.19% | 154.34% |
| SL4 | 113.00% | 133.60% |
| SL5 | 120.89% | 173.01% |
| SL6 | 127.88% | 189.29% |
| SL7 | 113.30% | 162.90% |
| SL8 | 123.53% | 147.42% |

Table 3.18: Lists the de-duplication of Minimum Threshold algorithm when normalized with respect to the Sliding Window algorithm (MT/SW) and the Naive algorithm (MT/N).

table match their respective points on the graph in Figure 3.10. Table 3.17 has the normalized percentages for all three algorithms in Table 3.16. The normalization is done the same way as in Table 3.14. The row of SL2/SL1 is the de-duplication percentage of the sequence length, or threshold, of 2 for each algorithm with respect to the de-duplication percentage of the sequence length, or threshold, of 1 for the same algorithm. The 93.13% in the Minimum Threshold column means that 93.13% of the data de-duplicated by a threshold of 1 can be de-duplicated when using a threshold of 2.

Table 3.18 also contains normalized percentages but instead of normalizing with respect to a sequence length or threshold value, the data has been normalized with respect to either the Sliding Window algorithm (MT/SW) or the Naive algorithm (MT/N). The row of SL2 is the de-duplication percentage of the Minimum Threshold algorithm using a threshold of 2 with respect to the de-duplication percentage of the Sliding Window algorithm and the Naive algorithm each using a sequence length of 2. The 104.95% in the MT/SW column means that the Minimum Threshold algorithm using a threshold of 2 can de-duplicate 104.95% of the data de-duplicated by the Sliding Window algorithm using a sequence length of 2. Another way to say this is that the Sliding Window using a sequence length of 2 can de-duplicate 95.29% of the data de-duplicated by the Minimum Threshold algorithm using a threshold of 2. It is clear that each iteration of the algorithm is better than its predecessor. Not only that, but each successive algorithm was almost always a noticeable improvement for each sequence length. Due to the Minimum Threshold algorithm's clear dominance, it was the algorithm chosen for the full-scale implementation.

# Chapter 4

# Implementation

This chapter outlines the design and implementation of the Minimum Threshold in a Log-Structured File System. In Section 4.1, I give a high level overview of how incoming write requests are de-duplicated. In Section 4.2, I list and briefly describe the data structures that were required to store the additional metadata generated by the Minimum Threshold algorithm. In Section 4.3, I give detailed explanations about how the algorithm and file system handle write requests and delete requests. I also briefly discuss the impact of adding de-duplication to the file system when handling read requests.

## 4.1 Design

The algorithm for de-duplicating sequences is implemented in a log-structured file system that allows de-staged writes. As writes come in from users, the file system groups them into a segment. When the segment becomes full or a preset amount of time has elapsed, the segment data is flushed to disk. My algorithm is applied to the incoming write data when the file system decides to flush the segment data to disk. The algorithm de-duplicates sequentially duplicate data and writes the remaining data to disk.

The file system reorganizes and all incoming write data by grouping the data based on filenames. My algorithm checks for sequential write data in a file write request that exists on disk in the exact same order. If a group of spatially local data blocks are found to occur sequentially on disk, the spatially local blocks are grouped together in a sequence and de-duplicated.

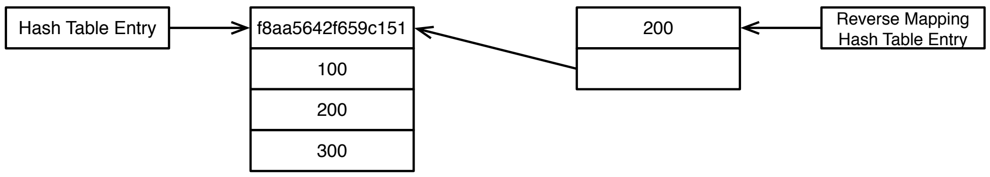


Figure 4.1: Example of Hash Table and Reverse Mapping Hash Table entries.

## 4.2 Data Structures

To keep track of the additional metadata required for the Minimum Threshold algorithm, I had to add new data structures to the file system. In this section I have broken the data structures into two categories: static and dynamic. The static data structures persist so long as the file system is up and running. The dynamic data structures are allocated and freed for each file write request sent to the algorithm.

### 4.2.1 Static Data Structures

I created two static data structures and recycled one existing data structure to store the additional metadata generated by the Minimum Threshold algorithm. The first two data structures contain mapping information. The Hash Table maps an MD5 hash to a list of on-disk locations with content that creates the MD5 hash. The second data structure contains a mapping of a single on-disk location to the MD5 hash of the content stored at the location. Lastly, the Reference Count File has a reference counter for every active on-disk location. It keeps track of the number of times a block is de-duplicated.

#### 4.2.1.1 Hash Table

The Hash Table data structure keeps track of the on-disk locations for all the MD5 hashes of data blocks that have been previously written to disk. On the left in Figure 4.1 is a simple Hash Table entry. The first item in the entry is an MD5 hash. The following 3 number entries are the on-disk locations of data blocks whose content generated the MD5 hash. To search for a duplicate block, the system accepts a MD5 hash and uses it to index into a bucket in the hash table. The system compares the MD5 hash to all of the hashes in the bucket. If a match is found, a list of on-disk locations for that hash are returned. If the hash is not found, the system returns nothing and the algorithm knows a block is unique.

#### 4.2.1.2 Reverse Mapping Hash Table

The Reverse Mapping Hash Table is a one-to-one mapping of on-disk locations to a MD5 hash. Instead of storing the hash value again, each bucket in the reverse mapped hash table points to its mapped MD5 hash entry in the main hash table. On the right in Figure 4.1 is a simple Reverse Mapping Hash Table entry. The first item in the entry is an on-disk location for a block. The second item in the entry is a pointer to the Hash Table entry for the on-disk location. The Reverse Mapping Hash Table helps with look ups when the algorithm has the location of a block but not the MD5 hash of the data that coincides with the on-disk location. The aforementioned situation is most frequently encountered when deleting file data blocks and checking data when performing file overwrites.

#### 4.2.1.3 Reference Count File

This file provides a mapping between every data block on disk and its reference count. Every time a file data block is written to disk, an entry is created in this file. Every time a block is de-duplicated against a block on disk, the counter for the block on disk is incremented. If a file is deleted, the reference counts of all de-duplicated blocks are decremented by one. When a file is deleted, any blocks that have a reference count of zero are removed from the Reverse Mapping Hash Table and Hash Table data structures before the block is marked as free.

### 4.2.2 Dynamic Data Structures

The dynamic data structures in this section only last while the file write request has not been written to disk. Each request has its own set of the three dynamic data structures listed below. When the request has completed, the dynamic data structures are freed. The Index List and De-duplication List are used to determine if a block should be de-duplicated or written to disk. The Algorithm List is used to find sequences and mark blocks for de-duplication.

#### 4.2.2.1 Algorithm List

The Algorithm List is the master control of the data structures. It is responsible for storing all of the on-disk locations for blocks that match file data. Once the list is populated with all possible locations, the algorithm scans the list and determines the sequence boundaries for de-duplication. As sequences are matched, the entries are inserted into the De-duplication List and removed from the Algorithm List and Index List. Figure 4.2 is an example state of the Algorithm List data structure after it has been populated with all of the on-disk locations for each block in a write request. The first number in each list element is between 0 and 5.

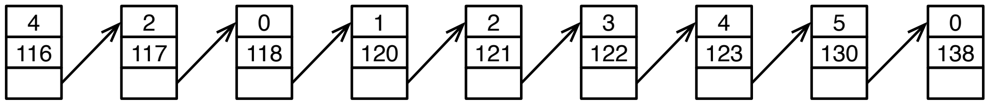


Figure 4.2: Example state of the Algorithm List data structure.

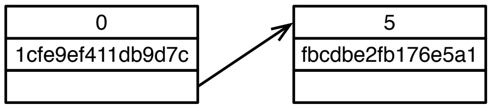


Figure 4.3: Example state of the Index List data structure based on the Algorithm List data structure given in Figure 4.2.

These are the file block numbers associated with each block in the write request. The second number in each list element is one of the on-disk locations that maps to the same MD5 hash as the file block it's paired with. For example, file block number 4 has two on-disk locations: 116 and 123. Therefore, there are two list elements for file block number 4 in Figure 4.2. As was previously stated, the elements in the Algorithm List shown in Figure 4.2 are sorted by on-disk location.

#### 4.2.2.2 Index List

When the algorithm is invoked, the Index List first contains an entry for every 4KB block write for the file being de-duplicated. Each entry contains two pieces of information: the block's file block number (FBN) and the MD5 hash. When the algorithm has finished determining the sequence boundaries of a file, the Index List only contains entries for the blocks that are not being de-duplicated. The entries for the blocks being de-duplicated are in the De-duplication List. Figure 4.3 is an example state of the Index List given the Algorithm List state in Figure 4.2 and assuming de-duplication was performed using a threshold of 3. The Algorithm List contains a sequence of four blocks, FBNs 1 through 4, that will be de-duplicated. FBNs 0 and 5 are not in a sequence and will be written to disk. The Index List in Figure 4.3 has two elements in it. The first entry in both elements is the FBN of the blocks being written to disk. The second entry is both elements is the MD5 hash of the content at each FBN.

#### 4.2.2.3 De-duplication List

The De-duplication List is empty when the de-duplication algorithm is invoked. When the algorithm completes, the De-duplication List contains entries for all blocks that are to be de-duplicated for a file write. Each entry in the list contains the following: the block's FBN and the

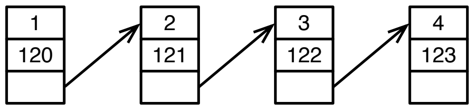


Figure 4.4: Example state of the De-duplication List data structure based on the Algorithm List data structure given in Figure 4.2.

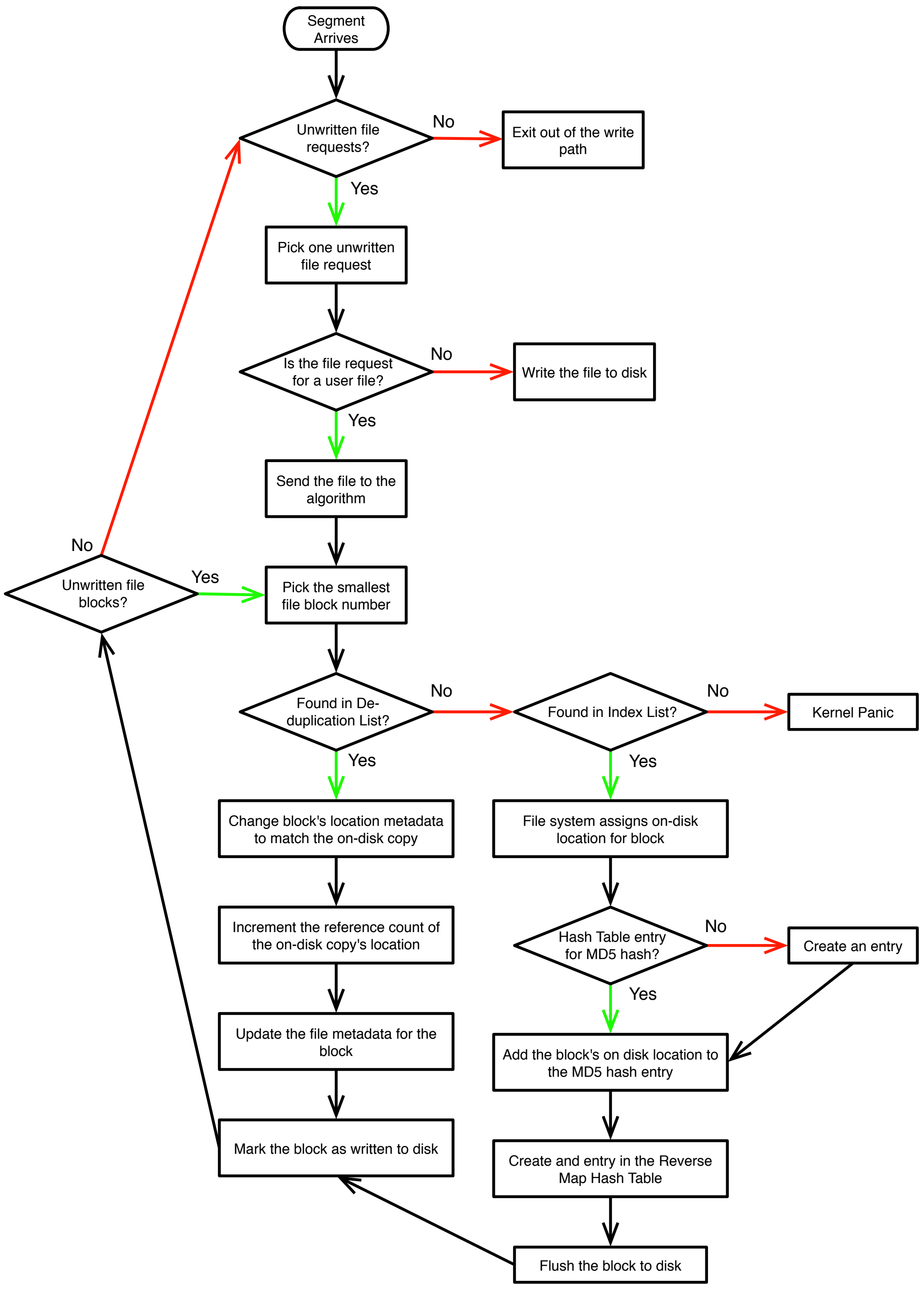
location of the on-disk data block that matched the duplicate file block. Figure 4.4 is an example state of the De-duplication List given the Algorithm List state in Figure 4.2 and assuming de-duplication was performed using a threshold of 3. The De-duplication List in Figure 4.4 has list elements for the FBNs found in sequence from the Algorithm List in Figure 4.2. The first entry in the list elements are the FBNs of the blocks to be de-duplicated. The second entry in the list elements are the on-disk locations being referenced when de-duplicating the block. The reference counts for these on-disk locations will each be incremented by one when the blocks are de-duplicated.

## 4.3 Implementation

In this section I discuss the three major types of file system requests that are received by the file system: write, read and delete. Section 4.3.1 details how file system write request goes through the write path. It covers how the dynamic data structures are populated and used to determine sequences for de-duplication, and how the file system was changed to handle both de-duplicated blocks and blocks that still need to be written to disk. Section 4.3.2 discusses how read requests are done in the file system and the impact using sequences for de-duplication has on handling read requests. Lastly, Section 4.3.3 describes how the delete path for a file has been changed to reflect the addition of de-duplication in the system.

### 4.3.1 Write Requests

Because this implementation sits on top of a log-structured file system, when a write is issued to disk there may be write data for more than one file. Before writing file data to disk, the file system batches all of the write data for each file together. After batching the write data, the file system writes the data blocks to disk one file at a time. This write allocation is ideal for de-duplicating files as they come in using sequences. For each file, there is an initial check to make sure that it is de-duplicatable. In this implementation only user file data is de-duplicated. If the incoming file belongs to the system or is a metadata or directory entry, the file is simply dismissed and no de-duplication is attempted. For file writes that belong to



Segment Arrives
Unwritten file requests?
No
Exit out of the write path
Yes
Pick one unwritten file request
Is the file request for a user file?
No
Write the file to disk
Yes
Send the file to the algorithm
No
Unwritten file blocks?
Yes
Pick the smallest file block number
Found in De-duplication List?
No
Found in Index List?
No
Kernel Panic
Yes
Yes
Change block's location metadata to match the on-disk copy
File system assigns on-disk location for block
Increment the reference count of the on-disk copy's location
Hash Table entry for MD5 hash?
No
Create an entry
Yes
Update the file metadata for the block
Add the block's on disk location to the MD5 hash entry
Mark the block as written to disk
Create and entry in the Reverse Map Hash Table
Flush the block to disk

Figure 4.5: Flow chart illustrating the write request path.

user files, the algorithm will set a flag in the file's inode to tell the file system that this file may contain de-duplicated data. The rest of this section describes, and is illustrated in Figure 4.5, how an incoming write request for a single file is processed by the algorithm and written to disk.

Each file write has a list of all the data blocks that are to be written to disk. The algorithm iterates through the list of data blocks, computes the MD5 hash of each block and stores the FBN and the MD5 hash information in the Index List data structure. Once all the blocks have been hashed, the algorithm can start searching for sequence boundaries. The algorithm iterates over the Index List to detect duplicate blocks and find sequences. First, the algorithm sends the Hash Table the MD5 hash value associated with the first file block. If the MD5 hash exists in the table, the algorithm receives a pointer to the list of on-disk locations that contain data that matches the MD5 hash. Those on-disk locations are added as entries in the Algorithm List. Each entry in the Algorithm List contains both the on-disk location of the duplicate data block and the FBN that belongs to the incoming write. The FBN is included to make sure that the sequences de-duplicate blocks in the same order as they appear in the incoming file write. The algorithm then sends the second MD5 hash value to the hash table. If the hash table returns nothing to the algorithm, the second file block is unique. Currently, the algorithm does nothing as this will be noticed in due time. If the Hash Table did have matching blocks for the MD5 hash, the algorithm would iterate over the locations and add them into the Algorithm List in ascending sorted order based on the on-disk location.

After all entries from the Index List have been processed and added to the Algorithm List, the algorithm searches the list for sequence boundaries. This search is done in linear time due to the sorted nature of the list. The algorithm starts at the beginning of the Algorithm List and compares the entries in the list two elements at a time. Two entries are considered to be in sequential order if both the FBN and the on-disk location of the second entry are one larger than the first entry. If the two entries are in sequence, the algorithm compares the second and third entry in the Algorithm List. This comparison continues on until the algorithm finds two entries that aren't in sequence. Once a sequence is broken, the algorithm calculates how many blocks are in sequential order. If the number of blocks in sequential order is at least the minimum threshold length, the algorithm marks the blocks as being in a sequence. First, the FBNs are removed from the Index List and added to the De-duplication List. After, the algorithm removes all entries in the Algorithm List that contain the FBNs that are already marked as being in a sequence. This process continues on until the algorithm reaches the end of the Algorithm List and returns control back to the file system.

When the file system regains control, it iterates through all of the data blocks that need to be written to disk for the file. There are two paths a write can go down depending on if it can be de-duplicated. For each file block, if the FBN is in the Index List, it was unique or did not fit into a sequence and goes down the "no duplicate" path. Any block writes sent down this path are written to disk and their on-disk location and MD5 hash are added to the Hash Table and the Reverse Mapping Hash Table for future use and an entry is created in the Reference Count File for the block. If the FBN is in the De-duplication List, the block is in a sequence with an already known on-disk location. Instead of allowing the file system to write the block to disk, the block write is sent down the "de-duplicate" path. Any block writes sent down this path have their location modified to the known on-disk location and a reference count is incremented for that same location in the Reference Count File. The third path a write can down is the original write path. It's used for any writes that are not user file data.

After these writes complete, the process repeats for the next file in the write request. Because all de-duplication happens on the granularity of file data blocks, a file read doesn't require any extra levels of indirection. The file system sees the file's block allocation as if it chose to write the de-duplicated blocks there in the first place.

### 4.3.2 Read Requests

Because of the way de-duplication is done, the read request path did not need to be modified. When a write block is de-duplicated, the algorithm assigns the on-disk location of the data block and tells the file system to update the metadata as if the block has already been written to disk. Because the algorithm changes the file metadata to use already written data, any read request issued will be processed as if the data was written there in the first place.

### 4.3.3 File Deletion

The required changes to the file deletion path were minor because a lot of the checks for de-duplicated data in the file system already existed in the code base. When the file system gets a delete request for a data block, it first checks to see if the block's inode is flagged for potential de-duplicated blocks. If the inode is not flagged, the block is simply freed without any extra steps. If the inode has been flagged as potentially having de-duplicated data blocks, the data block is looked up in the Reference Count File. There are three potential outcomes after lookup is performed: 1) the block is not in the file, 2) the block is in the file and its reference count is greater than zero, or 3) the block is in the file and its reference count is zero. The first outcome has two possible origins and is beyond the scope of this section. The second and third outcomes are discussed below and are illustrated in Figure 4.6.

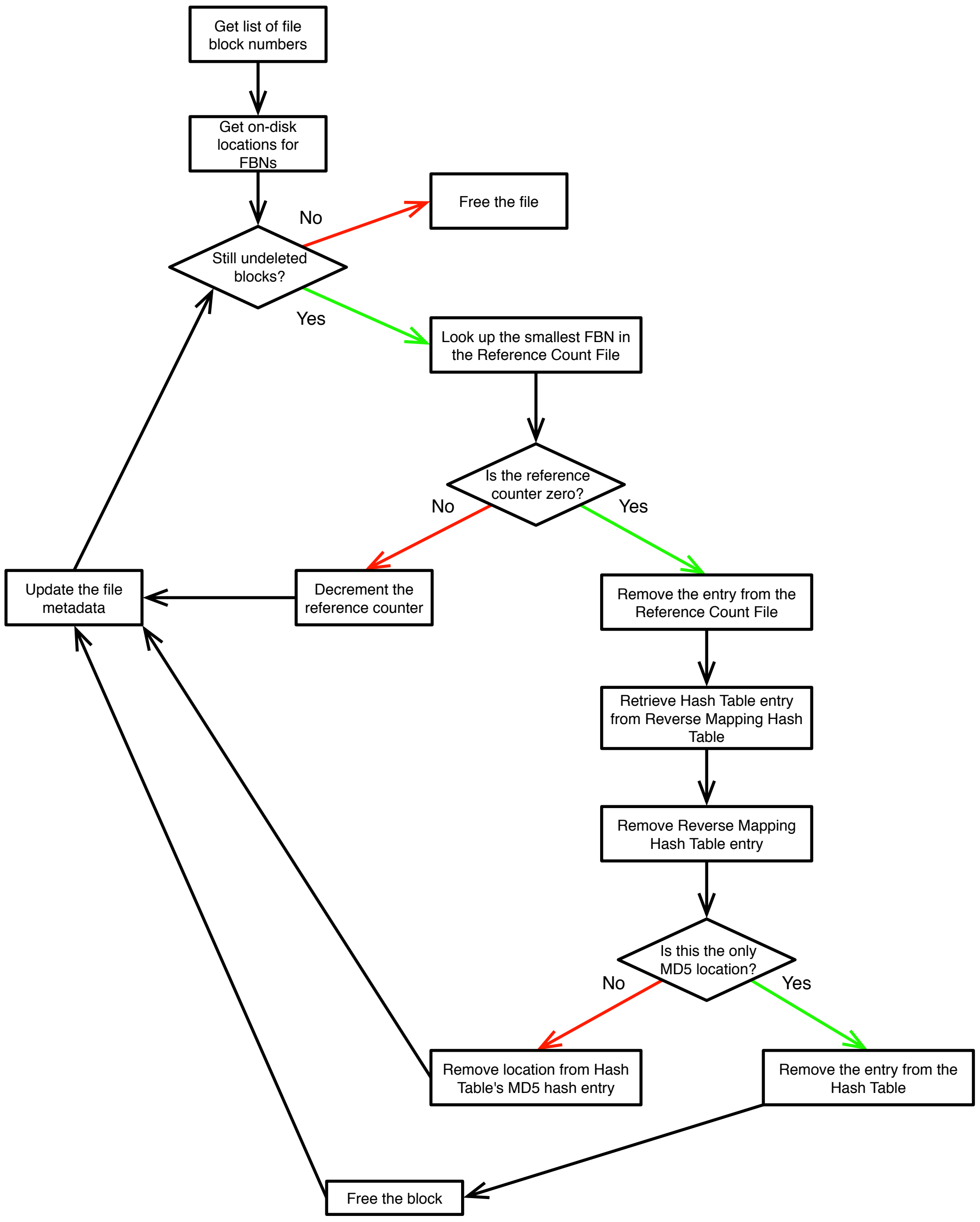

Get list of file block numbers
Get on-disk locations for FBNs
Still undeleted blocks?
No
Free the file
Yes
Look up the smallest FBN in the Reference Count File
Is the reference counter zero?
No
Yes
Update the file metadata
Decrement the reference counter
Remove the entry from the Reference Count File
Retrieve Hash Table entry from Reverse Mapping Hash Table
Remove Reverse Mapping Hash Table entry
Is this the only MD5 location?
No
Yes
Remove location from Hash Table's MD5 hash entry
Remove the entry from the Hash Table
Free the block

Figure 4.6: Flow chart illustrating the file deletion path.

If the block is in the Reference Count File with a reference count greater than zero, the reference count for the block is simply decremented by one. The data block is then marked as free and deletion continues. If the block is in the Reference Count File with a count of zero, that means the block is unique in the file system and has no other blocks referencing it. First, the block is looked up in the Reverse Mapping Hash Table. The MD5 hash value is saved and the file system follows the link to the Hash Table entry for the MD5 hash value. The file system checks to make sure that there is more than one on-disk location for the matching MD5 value. If there is, it simply removes the on-disk location for the block to be deleted. If there is only one entry for the MD5 value, the entire MD5 entry in the hash table is removed. Once the cleanup is done for these two data structures, the file system returns to the Reference Count File and removes the entry for the data block. After all of this has been completed, the file system can mark the data block as free.

# Chapter 5

# Experiments

To test my implementation I chose to use three workloads: Linux tars, the Corporate CIFS trace and the Engineering CIFS trace. The Linux workload was chosen because it is a vastly different workload than the Corporate and Engineering CIFS traces. The Linux workload consists of 109 versions of the Linux kernel from version 2.5.0 to 2.6.32. The Corporate and Engineering CIFS traces are the same as discussed in Chapter 3. Because they are CIFS requests gathered from an actual primary storage system, they are an ideal workload to test my implementation.

All workload data was kept on a client machine locally. The remote storage system running the implemented Minimum Threshold algorithm was mounted on the client machines via NFS. Each workload was sent to the remote storage system via NFS. By sending the workload data via NFS there is no cross-contamination of disk seeks or data de-duplication by storing the workload data on the same storage system being used for testing. NFS was chosen instead of CIFS because, as mentioned in 3.1.3, the Corporate and Engineering CIFS traces are two text files, not the actual file system requests themselves.

Each workload was used to test de-duplication and disk seeks for read requests. Before the de-duplication algorithm was tested, each workload was run without performing de-duplication to get baseline disk seek statistics. The disk seeks gathered report the starting address of a request, the size of a request and if it is a read or write request. Write request data was also gathered to help calculate the read seek distances. A read seek distance is the absolute value of the difference between the last location accessed and the starting location of the read request. All seek distances are on the granularity of 4KB blocks. This is because the file system writes data at the 4KB block granularity and reads it back in the same fashion. A seek distance of 1 means to access the starting location for a read request, the disk had to seek past one 4KB block of data. De-duplication using different thresholds will show the impact of threshold length for each workload. The disk seek logs will convey the impact of threshold length on read requests for each workload.

In this chapter I discuss the workloads and present their de-duplication and disk seek information for varying threshold lengths. In Section 5.1, I present the first workload: Linux kernel source code. I briefly describe how the workload is run and how the statistics are gathered. In Section 5.2, I present the second and third workloads: the Corporate and Engineering CIFS traces. I briefly describe how the traces are replayed and how the statistics are gathered for both traces. While the explanation given applies to both, they are tested independently. In Section 5.3, I present the de-duplication and disk seek results for all workloads.

## 5.1 Linux Workload

The Linux workload used to test my implementation is comprised of 109 versions of the Linux kernel from version 2.5.0 to 2.6.32. All versions were downloaded from the Linux Kernel Archives [2] in tar.gz format, gunzipped, and left in tar format. Each tar file is extracted to a directory on the storage server one file at a time. After all the file writes complete, the directory is tarred and stored locally on the client machine.

### 5.1.1 Writing tars

Before extracting any tar files, the testing code mounts an NFS exported directory from the storage server running my modified file system to a local client-side directory. All tar files are extracted to the mounted directory one tar file at a time. When all Linux kernel tar files have been extracted, I collect all de-duplication statistics from the storage server for all writes to the mounted directory.

### 5.1.2 Reading tars

After gathering all of the de-duplication data associated with the file writes from the extracted Linux kernels, I need to gather read data. The mounted directory that contains all of the extracted Linux kernels is tarred to the client's local /tmp directory. The disk seeks required to read every file in the mounted directory are logged. The log is crawled to calculate the number of disk seeks, the amount of data read per disk seek, and the distance between disk seeks.

## 5.2 CIFS Workload

The CIFS traces used to test my implementation are the same traces used and discussed in Chapter 3. The two trace data sets, Corporate CIFS and Engineering, are handled separately but follow the same testing protocol. As was mentioned in Chapter 3, the trace data only contains file data that was accessed during the trace period. If only the last half of a file was accessed during the trace period, I will not know anything about the data contained in the first half of the file. The traces do not provide a full view of every file in the file system, let alone provide any kind of guarantee that even one file will have all its data represented in the trace.

### 5.2.1 Warming Data Structures

While the traces do not give a full picture of the file system state, there is some data that is known to be on disk before the trace was created. This data comes from "unmodified read data". Unmodified read data is defined as file data that has been returned by a read request that does not have a matching write request earlier in the trace. By collecting all of the unmodified read data, a small subset of the file system state that existed before the traces were gathered can be recreated.

Before collecting unmodified read data, the testing code mounts an NFS exported directory from the storage server on the client. The trace is then replayed and all of the unmodified read data is sent as write requests to the file system. After the trace replay completes, the data structures contain information about file blocks that are known to be on disk before the trace gathering started. All de-duplication counters are reset, but the contents of the hash table data structures are left unmodified.

### 5.2.2 Gathering Statistics

After the counters have been reset, the trace is replayed a second time. All data is read from and written to the same NFS mounted directory. After the second replay of the trace has completed, all of the statistics are collected. First, the de-duplication numbers of the write traffic are extracted from the storage server. Then the disk seek log is copied to the local client machine. Just like the Linux kernel tests, the disk seek log is crawled to calculate the number of disk seeks, the amount of data read per seek and the distances between disk seeks.

## 5.3 Graphs

In this section I present the results for each of the data sets in turn. For each workload I report the de-duplication percentage as the threshold increases in powers of 2, and the disk seek performance for each threshold. The disk seek performance is given in the form of a histogram with tables highlighting important statistical information. After presenting the results for each workload, I state and explain the best threshold option for the data set being discussed.

### 5.3.1 Linux Workload

Figure 5.1 shows the amount of de-duplication achieved by the Minimum Threshold implementation when applied to the Linux workload. Each bar on the x-axis in Figure 5.1 is the de-duplication percentage for a threshold. The thresholds were tested from a threshold of 1 to a threshold of 8 in increments of powers of two. The y-axis of Figure 5.1 is the de-duplication achieved by the Minimum Threshold implementation for the Linux workload for each threshold tested. The number on top of each bar indicates the de-duplication percentage achieved by each threshold tested for the Linux workload. With respect to this graph, the higher the bar is on the y-axis, the better the result. For example, the Linux workload has a de-duplication percentage of 55.34% when using a threshold of 1. This means that 55.34% of the Linux workload data can be de-duplicated at the write request level, requiring only 44.66% of the write request data to be written to disk. The drop from a threshold of 2 to a threshold of 4 is due to the many small files that exist in the Linux workload. If a file is smaller than the threshold length, the file is not considered for de-duplication. Figure 5.1 shows that setting the threshold length to be larger than the average file size of a workload can have a significant impact on the de-duplication achieved. However, the impact of sequence-based de-duplication on disk seeks for read requests needs to be investigated before any conclusions can be made for this workload.

Figure 5.2 shows the impact of the Minimum Threshold implementation on read request disk seeks for the Linux workload. It presents the read request disk seek distances in a histogram based on the seek distance incurred by each seek. Each tick on the x-axis marks a bin. The first bin is for *zero-distance seeks* which indicates a read request that was satisfied without needing to perform a disk seek, and the remaining bins group disk seeks based on the seek distance's order of magnitude. Each bin has been shortened using interval notations. For example, the bin $[10^2, 10^3)$ contains the disk seeks with distances from 100 to 999 blocks inclusive. The last bin $[10^6, \infty)$ is all disk seeks that have a distance of 1,000,000 blocks or more. The y-axis of Figure 5.2 is the number of disk seeks for each bin. The higher the bar is

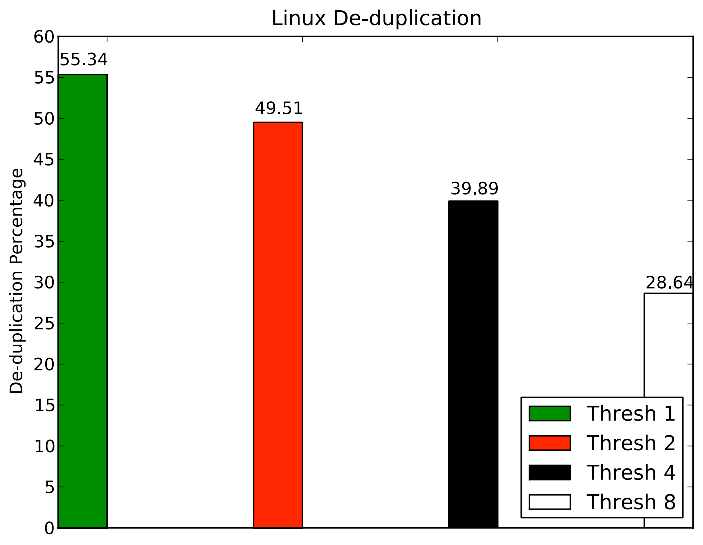


Figure 5.1: Impact of the Minimum Threshold implementation on the de-duplication of the Linux workload.

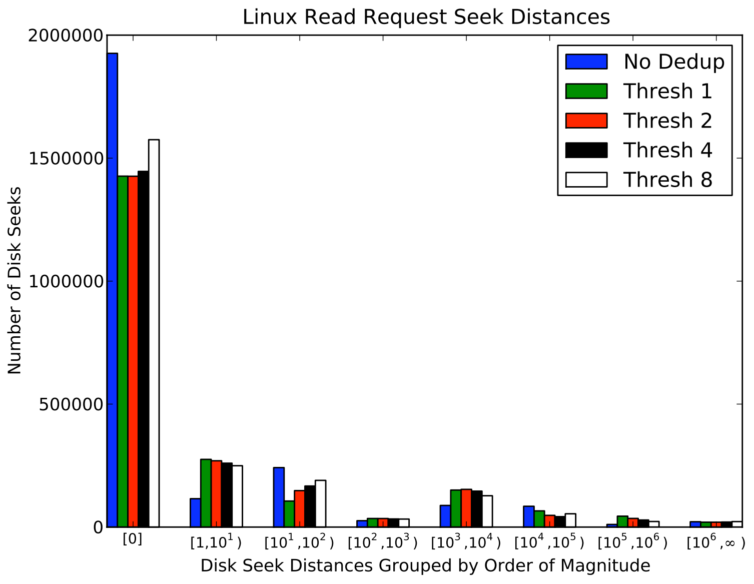


Figure 5.2: Impact of the Minimum Threshold implementation on disk seeks that respond to read requests in the Linux workload. The information is presented as a histogram where each bin represents an order of magnitude for disk seek distances.

| | **Total Disk Seeks** | **Zero Distance** | **Non-Zero Distance** |
|---|---|---|---|
| No De-dup | 2,513,716 | 1,925,956 | 587,760 |
| Thresh 1 | 2,123,195 | 1,426,662 | 696,533 |
| Thresh 2 | 2,134,909 | 1,426,322 | 708,587 |
| Thresh 4 | 2,143,405 | 1,446,025 | 697,380 |
| Thresh 8 | 2,273,086 | 1,574,923 | 698,163 |

Table 5.1: Lists the total disk seeks required to satisfy all the read requests generated by each test of the Linux workload. Total disk seeks are also split into two categories: disk seeks of zero distance and disk seeks of non-zero distance.

| | **Mean Seek Distance** | **Average Blocks Read per Seek** |
|---|---|---|
| No De-dup | 25,366.5 | 2.33 |
| Thresh 1 | 24,411.5 | 1.45 |
| Thresh 2 | 23,952.2 | 1.60 |
| Thresh 4 | 24,720.6 | 1.84 |
| Thresh 8 | 25,959.9 | 2.00 |

Table 5.2: Lists the mean seek distance and the average blocks read per seek for the disk seeks listed in Table 5.1.

on the y-axis, the more disk seeks traveled the distances represented by the bin to read data for a read request. There is a drop in the number of zero-distance seeks for all tests with de-duplication turned on. There is also a spike in the read request disk seeks with lengths between 1 to 9 blocks and 1,000 to 9,999 blocks. Based on this graph alone, it would suggest that there is no improvement in read request seek distances when using a sequence-based de-duplication scheme. However, when the information in Figure 5.2 is paired with the information from Tables 5.1 and 5.2, the best choice for the Linux workload would be a threshold of 2.

Tables 5.1 and 5.2 give read request disk seek statistics for the Linux workload. Each row in both tables corresponds to a test of the workload shown in Figure 5.2. The first rows in both tables report read request disk seek statistics for the Linux workload without de-duplication. The remaining four rows in both tables give the same statistics with each row representing a threshold length. The first column in Table 5.1 is the total number of disk seeks required to read all of the data for the read requests in the workload. The second column is for the zero distance seeks that were discussed previously. It gives the number of disk seeks that required a zero distance seek to read the requested file data. The third column is for non-zero distance seeks—any disk seek that did not have a seek distance of zero falls into this category. The first column in Table 5.2 is the mean seek distance for each test of the Linux workload. The mean seek distance is the the sum of all the zero and non-zero seek distances divided by the total number of disk seeks. The second column in Table 5.2 is the average number of blocks read per disk seek.

The Linux workload without de-duplication has the most overall disk seeks, the most zero distance seeks, the fewest non-zero distance seeks but also has the second highest mean seek distance. This indicates that while no de-duplication gives the most zero distance seeks, the non-zero distance seeks are of non-trivial distances. The Linux workload without de-duplication does have the highest average blocks read per seek with a threshold of 8 coming in second. However, the threshold of 8 has the largest mean seek distance. Using a threshold of 2 is the best choice for Linux workload. The threshold of 4 has more zero distance seeks and, on average, more blocks read per seek than a threshold of 2. However, the threshold of 2 has fewer total disk seeks, a shorter mean seek distance and a de-duplication percentage of 49.51% of the workload where the threshold of 4 only de-duplicates 39.89%.

### 5.3.2 Corporate CIFS Trace Workload

Figure 5.3 shows the amount of de-duplication achieved by the Minimum Threshold implementation when applied to the Corporate CIFS trace workload. Each bar on the x-axis in Figure 5.3 is the de-duplication percentage for a threshold. The thresholds were tested from a threshold of 1 to a threshold of 8 in increments of powers of two. The y-axis of Figure 5.3 is the de-duplication achieved by the Minimum Threshold implementation for the Corporate CIFS trace workload for each threshold tested. The number on top of each bar indicates the de-duplication percentage achieved by the Minimum Threshold implementation when running the Corporate CIFS trace workload with different threshold lengths. With respect to this graph, the higher the bar is on the y-axis, the better the result.

For example, the Corporate CIFS trace workload has a de-duplication percentage of 32.25% when using a threshold of 1 and a de-duplication percentage of 29.36% when using a threshold of 8. This is a loss of 2.84% in overall de-duplication by increasing the threshold from 1 to 8. The overall loss is also an 8.96% relative loss in the amount of data de-duplicated using a threshold of 8 when compared to using a threshold of 1. The small loss in overall de-duplication for the Corporate CIFS trace workload means the best choice for this workload is dependent on the impact of threshold lengths on the read request disk seeks.

Figure 5.4 shows the impact of the Minimum Threshold implementation on read request disk seeks for the Corporate CIFS trace workload. It presents the read request disk seeks in the same format as Figure 5.2 and should be read the same way. Figure 5.4 also shows that by using a sequence-based de-duplication scheme, there is a decrease in the number of zero-distance seeks. More of the disk seeks also seem to have a distance between 100 and 999 blocks. Based on Figure 5.4 alone, it appears that performing sequence-based de-duplication

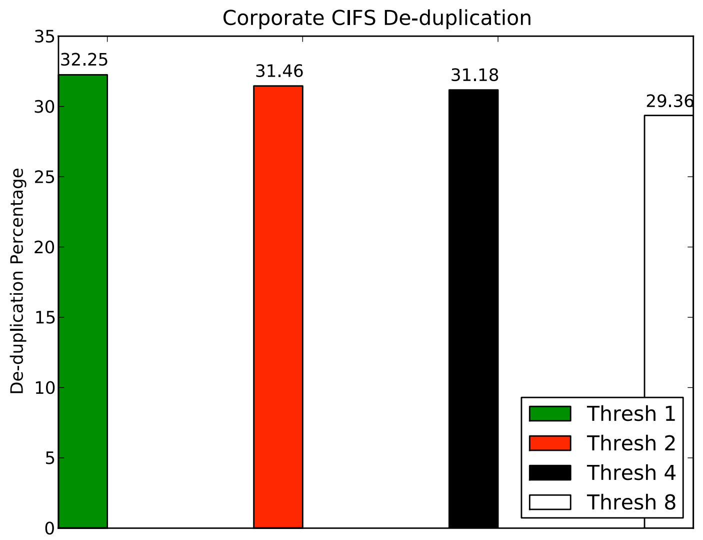


Figure 5.3: Impact of the Minimum Threshold implementation on the Corporate CIFS trace workload.

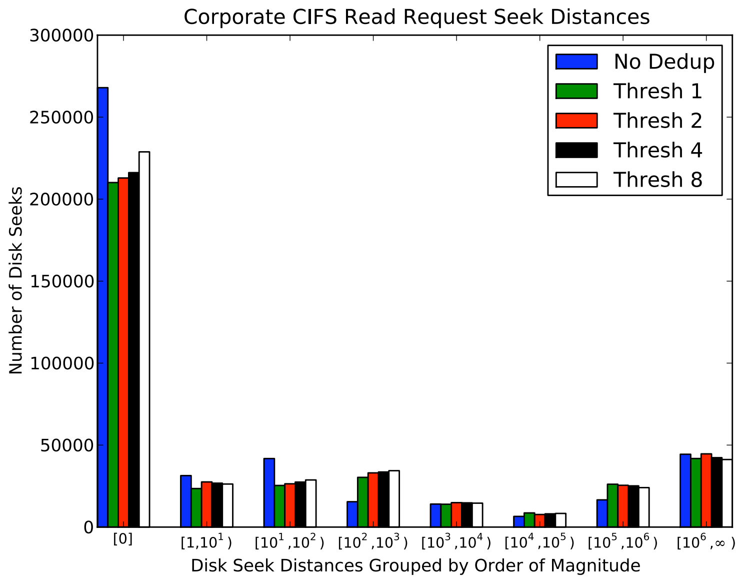


Figure 5.4: Impact of the Minimum Threshold implementation on disk seeks that respond to read requests in the Corporate CIFS trace workload. The information is presented as a histogram where each bin represents an order of magnitude for disk seek distances.

| | **Total Disk Seeks** | **Zero Distance** | **Non-Zero Distance** |
|---|---|---|---|
| No De-dup | 438,142 | 267,956 | 170,186 |
| Thresh 1 | 379,805 | 210,094 | 169,711 |
| Thresh 2 | 392,553 | 212,882 | 179,671 |
| Thresh 4 | 394,298 | 216,150 | 178,148 |
| Thresh 8 | 406,337 | 228,813 | 177,524 |

Table 5.3: Lists the total disk seeks required to satisfy all the read requests generated by each test of the Corporate CIFS trace workload. Total disk seeks are also split into two categories: disk seeks of zero distance and disk seeks of non-zero distance.

| | **Mean Seek Distance** | **Blocks Read per Seek** |
|---|---|---|
| No De-dup | 332,181 | 11.86 |
| Thresh 1 | 284,092 | 11.92 |
| Thresh 2 | 301,777 | 11.66 |
| Thresh 4 | 283,044 | 11.67 |
| Thresh 8 | 275,500 | 11.49 |

Table 5.4: Lists the mean seek distance and the average blocks read per seek for the disk seeks listed in Table 5.3.

incurs longer disk seeks for read requests. But when the information from Figure 5.4 is paired with the information from Tables 5.3 and 5.4, it is shown that the best choice for the Corporate CIFS trace workload is a threshold of 8.

Tables 5.3 and 5.4 give read request disk seek statistics for the Corporate CIFS trace workload. Each row in both tables corresponds to a test of the workload shown in Figure 5.4. The layout of data in both tables is in the same format as Tables 5.1 and 5.2 and should be read the same way. The Corporate CIFS trace workload without de-duplication has the largest number of overall disk seeks and the most zero distance seeks, but it also has the highest mean seek distance. This indicates, similar to the Linux workload, that while no de-duplication gives the most zero distance seeks, the non-zero distance seeks are of non-trivial distances. The Corporate CIFS trace workload with a threshold of 1 reads the most blocks per disk seek on average with 11.92 4KB blocks read per disk seek. Based on the numbers presented in Tables 5.3 and 5.4, using a threshold of 1 appears to be the best choice for the Corporate CIFS trace workload. The threshold of 1 has the fewest total disk seeks, fewer non-zero distance seeks, more blocks read per seek and the best possible de-duplication out of all the thresholds. However, I propose using a threshold of 8 instead of 1 for the Corporate CIFS trace workload. Because the minimum threshold expresses the smallest sequence that can be de-duplicated, it is possible that the Corporate CIFS trace workload will degrade into a block-based de-duplication scheme when using a threshold of 1. All rows in Table 5.4 read at least 11 blocks per seek on

| | **Total Disk Seeks** | **Zero Distance** | **Non-Zero Distance** |
|---|---|---|---|
| No De-dup | 693,034 | 373,115 | 319,919 |
| Thresh 1 | 722,602 | 349,938 | 372,664 |
| Thresh 2 | 704,153 | 368,870 | 335,283 |
| Thresh 4 | 699,325 | 365,154 | 334,171 |
| Thresh 8 | 714,294 | 370,176 | 344,118 |

Table 5.5: Lists the total disk seeks required to satisfy all the read requests generated by each test of the Engineering CIFS trace workload. Total disk seeks are also split into two categories: disk seeks of zero distance and disk seeks of non-zero distance.

average. In order to preserve that average, a longer sequence length would serve the workload better than a shorter one.

### 5.3.3 Engineering CIFS Trace Workload

Figure 5.5 shows the amount of de-duplication achieved by the Minimum Threshold implementation when applied to the Engineering CIFS trace workload. Each bar on the x-axis in Figure 5.5 is the de-duplication percentage for a threshold. The thresholds were tested from a threshold of 1 to a threshold of 8 in increments of powers of two. The y-axis of Figure 5.5 is the de-duplication achieved by the Minimum Threshold implementation for the Engineering CIFS trace workload for each threshold tested. The number on top of each bar indicates the de-duplication percentage achieved by the Minimum Threshold implementation when running the Engineering CIFS trace workload with different threshold lengths. With respect to this graph, the higher the bar is on the y-axis, the better the result.

For example, the Engineering CIFS trace workload has a de-duplication percentage of 17.90% when using a threshold of 1 and a de-duplication percentage of 16.61% when using a threshold of 8. This is a loss of 2.84% in overall de-duplication by increasing the threshold from 1 to 8. The overall loss is also an 7.21% relative loss in the amount of data de-duplicated using a threshold of 8 when compared to using a threshold of 1. The small loss in overall de-duplication for the Engineering CIFS trace workload means the best choice for this workload is dependent on the impact of threshold lengths on the read request disk seeks.

Figure 5.6 shows the impact of the Minimum Threshold implementation on read request disk seeks for the Engineering CIFS trace workload. It presents the read request disk seeks in the same format as Figure 5.2 and should be read the same way. Figure 5.6 also shows that by using a sequence-based de-duplication scheme, there is a decrease in the number of zero-distance seeks—most notably for a threshold of 1. The remaining disk seeks in Figure 5.6 are close to the test of the Engineering CIFS trace workload without de-duplication. Because

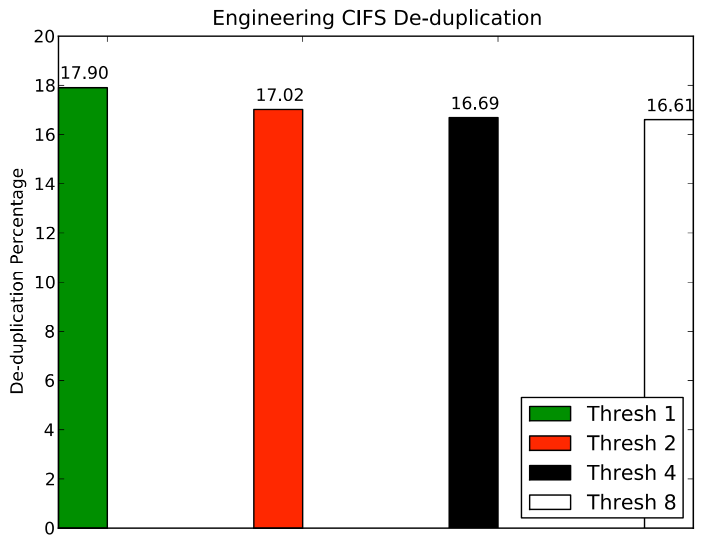


Figure 5.5: Impact of the Minimum Threshold implementation on the Engineering CIFS trace workload.

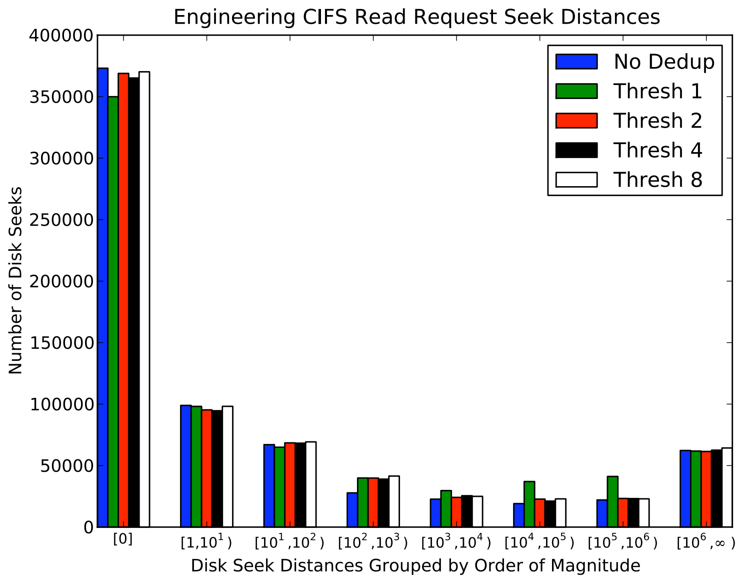


Figure 5.6: Impact of the Minimum Threshold implementation on disk seeks that respond to read requests in the Engineering CIFS trace workload. The information is presented as a histogram where each bin represents an order of magnitude for disk seek distances.

|  | **Mean Seek Distance** | **Blocks Read per Seek** |
|---|---|---|
| No De-dup | 399,197 | 8.89 |
| Thresh 1 | 375,863 | 8.36 |
| Thresh 2 | 339,250 | 8.64 |
| Thresh 4 | 344,859 | 8.62 |
| Thresh 8 | 350,142 | 8.57 |

Table 5.6: Lists the mean seek distance and the average blocks read per seek for the disk seeks listed in Table 5.5.

the disk seek distances are presented as a graph of the distribution of probabilities, Figure 5.6 shows how the Minimum Threshold implementation impacts read request disk seeks but does not give a perspective on the absolute disk seek statistics. Based on Figure 5.6 alone, it would seem that performing sequence-based de-duplication incurs longer disk seeks for read requests for a threshold of 1 and is almost on par for thresholds 2, 4 and 8. But when the information from Figure 5.6 is paired with the information from Tables 5.5 and 5.6, it is shown that the best choice for the Engineering CIFS trace workload is a threshold of 2.

Tables 5.5 and 5.6 give read request disk seek statistics for the Engineering CIFS trace workload. Each row in both tables corresponds to a test of the workload shown in Figure 5.6. The layout of data in both tables is in the same format as Tables 5.1 and 5.2 and should be read the same way. The Engineering CIFS trace workload without de-duplication has the fewest overall disk seeks, the most zero-distance seeks and the fewest non-zero distance seeks, but it also has a high mean seek distance. Just like the Linux and Corporate CIFS trace workloads, the non-zero distance seeks for the Engineering CIFS trace workload are of non-trivial distances. The Engineering CIFS trace workload without de-duplication reads the most blocks per disk seek on average with 8.89 4KB blocks read per disk seek. Based on the numbers presented in Tables 5.5 and 5.6, using a threshold of 1 appears to be the worst choice for the Engineering CIFS trace workload unlike the Corporate CIFS trace workload. The threshold of 1 has the most total disk seeks, the least non-zero distance seeks, the most non-zero distance seeks and the fewest blocks read per seek on average. The only benefit to using a threshold of 1 is the getting the best possible de-duplication out of all the thresholds. I propose using either a threshold of 2 or 4 for the Engineering CIFS trace workload. The threshold of 2 has a shorter mean seek distance, reads more blocks per seek on average and de-duplicates more data than a threshold of 4. However, there is an inherent danger of using such a small threshold value as I discussed in Section 5.3.2. I was unable to test a threshold of 3 on the Engineering CIFS trace workload due to time constraints.

# Chapter 6

# Conclusion

Data de-duplication is a technique commonly applied to secondary storage systems in order to maximize storage space utilization by eliminating duplicate file data. There are many different ways to perform data de-duplication, but a common trait among many approaches is to perform what Mandagere [21] refers to as “Sub File Hashing” (SFH). SFH techniques break a file into smaller pieces to increase the probability of finding duplicate file data.

However, removing duplicate file data using a subset of file data inherently causes file fragmentation. When a file is written out sequentially, the underlying file system attempts to optimize the on-disk placement for the file. After data de-duplication, the on-disk layout of a file can change drastically. Any optimal data placement done by the file system can be undone by de-duplication. A sequentially written file can have the on-disk layout of a randomly written file after data de-duplication completes.

This change in the on-disk representation of a file will result in an additional disk seeks to read the entire file. Depending on the de-duplication technique being used, there may be extra accesses to metadata structures to determine the on-disk locations for a file’s de-duplicated blocks. All of these extra seeks can hurt the read performance of a file system. While this read performance penalty typically goes unnoticed on secondary systems, it is one of the reasons de-duplication techniques are not performed on primary storage.

In Chapter 3, I have shown that a simple sequence-based de-duplication algorithm is able to detect and eliminate duplicate data from files stored on both secondary and primary storage systems. The same simple algorithm can also de-duplicate data from CIFS requests issued to a primary storage system. I presented two additional sequence-based de-duplication algorithms that were an improvement on the first algorithm. I have shown that each algorithm presented is an improvement over its predecessor by comparing the de-duplication achieved across all three algorithms for the same data set. I chose the Minimum Threshold algorithm, based on its de-duplication performance, to be used in a full-scale implementation.

In Chapter 4, I gave the major components involved in the full-scale implementation. I presented the static data structures added to the file system to accommodate the additional metadata generated by the Minimum Threshold algorithm, the transient data structures that are generated for each write and described the relationships between the data structures. I have outlined the code flow for a file write request and described, in detail, how a file write is handled from the time it enters the write path to determining which blocks are to be de-duplicated and which are written to disk.

In Chapter 5, I briefly described the workloads used to test the full-scale implementation and how they are run. I discussed how de-duplication is calculated and how disk seeks are gathered and measured. I then, for each workload, presented the impact of the Minimum Threshold on de-duplication and disk seeks for thresholds of 1, 2, 4 and 8. For comparison I also tested the workloads without de-duplication enabled. My results show that, for a properly selected threshold based on the contents of the workload, a sequence-based de-duplication algorithm can shorten the mean disk seek distance, reduce the total number of disk seeks and still read the same average number of blocks per seek when compared to running the same workload without de-duplication. Not only is the read performance comparable, my implementation also de-duplicates data.

## 6.1 Limitations

To get the Minimum Threshold implementation functioning in the file system, some compromises were required. The implementation only considered 3000 blocks at a time for each file. It will attempt to find duplicate blocks on disk for the first 3000 file blocks before looking at the next 3000 blocks. This means that there is actually a maximum segment size that is being enforced on the file system. Also, the implementation will not find the longest duplicate sequence, but the first duplicate sequence that is at least as long as the threshold value.

Currently the Hash Table only stores the first 32 on-disk locations for each unique MD5 hash. In the current implementation there is no algorithm to only keep locations that have been used for de-duplication. The first 32 locations a MD5 hash is written to are the only locations that are used in the table. This is partly because no code was written to handle paging the data structures in and out of memory and partly to enforce an upper bound on the number of items in the Algorithm List when determining segment boundaries.

None of the files in the segment know about each other. The current implementation attempts to de-duplicate each file write as quickly as possible, but it may have to block if de-duplicating a write takes too long. Unfortunately, the next thing the file system may do is choose to de-duplicate the next file write in the segment before finishing the first. This leads to two major problems. First, because the de-duplication of the first file write did not complete, none of the unique file data has been written to disk or added to the hash tables. The second file write loses out on any potential sequences from the first file and now any duplicate data could be written twice. The second problem is a race condition. For example, de-duplication of the first file write has begun and all sequences have been found, but before the write is flushed to disk the process is blocked and the next file is de-duplicated. The next write is found to be an overwrite and since none of the blocks have been referenced, the old data is deleted and new data is written. The race condition occurs if the first file write was using any of the deleted blocks in a sequence or sequences. Currently this race condition does not exist in any of the data sets tested, so the solution has yet to be decided.

One of the largest hindrances to the results of the implementation was the lack of a starting file system image. The data structures were pre-warmed with what little data was available in the traces, but a full file system crawl of the files would've yielded more significant results.

## 6.2 Future Work

The work presented in this thesis could benefit from the use of Flash memory. The number of on-disk locations was restricted because no code was written to page data structures in and out of memory. Writing the code will solve the limitation on storing on-disk locations problem, but it will also introduce a disk bottleneck problem. The larger the Hash Table and Reverse Mapping Hash Table data structures become, it becomes increasingly likely that there will be constant paging in and out of disk. Instead of writing the data structures to disk, a future task would be to write the data structures to Flash memory. A replacement strategy could be used to keep the most frequent common matches from the Hash Table and Reverse Mapping Hash Table in main memory while the remainder of the tables sit in Flash. This will remove the cap on the number of on-disk locations paired with a MD5 hash and the disk bottleneck problem. Because the additional metadata will be stored on Flash memory, there are no disk seeks to access the extra metadata generated by the de-duplication algorithm.

All of the experiments were run with just one client connected to the remote server. It is similar to an ideal situation where no one else was writing their data to disk and I had the best chance to perform sequential file writes. While the Corporate and Engineering CIFS traces contain the read and write requests from all users that accessed or wrote file data, the traces had no way to distinguish which files belonged to each user. So there was no real way to determine the impact of my implementation on each user. From my point of view, and the way that it was reported in my results, all seeks belonged to one user. In general however, multiple simultaneous workloads will increase the average seek distance length. The magnitude by which the seek distances would increase is dependent on the workloads. If there is a strong intersection between the workloads that would allow for significant de-duplication, the seek distances would be shorter on average. However, if the workloads have a small intersection or—even worse—are mutually exclusive, the distance between disk seeks for read requests for multiple users will increase rapidly.

Crawling a file system to create a static file system state before gathering 3 months of traces would be immensely beneficial to testing my implementation. The crawl would allow me to create the starting file system state and warm my data structures before running the traces. For each client in the trace, create a client that reads and writes the file data for that client in the trace. If possible, create a way to match read request disk seeks to each client to get a better feel for system performance as perceived by each client.

# Bibliography

[1] IBM System Storage N series Software Guide. `http://www.redbooks.ibm.com/abstracts/sg247129.html`, December 2010.

[2] The Linux Kernel Archives. `http://www.kernel.org/`, December 2010.

[3] Opendedup. `http://www.opendedup.org/`, December 2010.

[4] Siddhartha Annapureddy, Michael J. Freedman, and David Mazières. Shark: Scaling file servers via cooperative caching. In *Proceedings of the 2nd Symposium on Networked Systems Design and Implementation (NSDI)*, pages 129–142, 2005.

[5] Deepavali Bhagwat, Kave Eshghi, Darrell D. E. Long, and Mark Lillibridge. Extreme Binning: Scalable, parallel deduplication for chunk-based file backup. In *Proceedings of the 17th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS '09)*, London, UK, September 2009.

[6] Deepavali Bhagwat, Kristal Pollack, Darrell D. E. Long, Ethan L. Miller, Jehan-François Pâris, and Thomas Schwarz, S. J. Providing high reliability in a minimum redundancy archival storage system. In *Proceedings of the 14th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS '06)*, Monterey, CA, September 2006.

[7] Sergey Brin, James Davis, and Héctor García-Molina. Copy detection mechanisms for digital documents. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, pages 398–409, San Jose, CA, 1995.

[8] Michael Burrows, Charles Jerian, Butler Lampson, and Timothy Mann. On-line data compression in a log-structured file system. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 2–9, Boston, MA, October 1992.

[9] Austin T. Clements, Irfan Ahmad, Murali Vilayannur, and Jinyuan Li. Decentralized deduplication in SAN cluster file systems. In *Proceedings of the 2009 USENIX Annual Technical Conference*, San Diego, CA, June 2009.

[10] Fred Douglis and Arun Iyengar. Application-specific delta-encoding via resemblance detection. In *Proceedings of the 2003 USENIX Annual Technical Conference*, pages 113–126. USENIX, June 2003.

[11] Cezary Dubnicki, Leszek Gryz, Lukasz Heldt, Michal Kaczmarczyk, Wojciech Kilian, Przemyslaw Strzelczak, Jerzy Szczepkowski, Cristian Ungureanu, and Michal Welnicki. HYDRAstor: a scalable secondary storage. In *Proceedings of the 7th USENIX Conference on File and Storage Technologies (FAST)*, pages 197–210, San Francisco, CA, February 2009.

[12] Dave Hitz, James Lau, and Michael Malcom. File system design for an NFS file server appliance. In *Proceedings of the Winter 1994 USENIX Technical Conference*, pages 235–246, San Francisco, CA, January 1994.

[13] Bo Hong, Demyn Plantenberg, Darrell D. E. Long, and Miriam Sivan-Zimet. Duplicate data elimination in a SAN file system. In *Proceedings of the 21st IEEE / 12th NASA Goddard Conference on Mass Storage Systems and Technologies*, pages 301–314, College Park, MD, April 2004.

[14] Navendu Jain, Mike Dahlin, and Renu Tewari. TAPER: Tiered approach for eliminating redundancy in replica synchronization. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST)*, San Francisco, CA, December 2005.

[15] Ricardo Koller and Raju Rangaswami. I/O deduplication: Utilizing content similarity to improve I/O performance. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies (FAST)*, San Jose, CA, February 2010.

[16] Purushottam Kulkarni, Fred Douglis, Jason LaVoie, and John M. Tracey. Redundancy elimination within large collections of files. In *Proceedings of the 2004 USENIX Annual Technical Conference*, pages 59–72, Boston, MA, June 2004. USENIX.

[17] Andrew Leung, Minglong Shao, Timothy Bisson, Shankar Pasupathy, and Ethan L. Miller. Spyglass: Fast, scalable metadata search for large-scale storage systems. In *Proceedings of the 7th USENIX Conference on File and Storage Technologies (FAST)*, pages 153–166, February 2009.

[18] Andrew W. Leung, Shankar Pasupathy, Garth Goodson, and Ethan L. Miller. Measurement and analysis of large-scale network file system workloads. In *Proceedings of the 2008 USENIX Annual Technical Conference*, June 2008.

[19] Mark Lillibridge, Kave Eshghi, Deepavali Bhagwat, Vinay Deolalikar, Greg Trezise, and Peter Camble. Sparse indexing: Large scale, inline deduplication using sampling and locality. In *Proceedings of the 7th USENIX Conference on File and Storage Technologies (FAST)*, pages 111–123, San Francisco, CA, February 2009.

[20] Peter Macko, Margo Seltzer, and Keith A. Smith. Tracking back references in a write-anywhere file system. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies (FAST)*, February 2010.

[21] Nagapramod Mandagere, Pin Zhou, Mark A Smith, and Sandeep Uttamchandani. Demystifying data deduplication. In *Proceedings of the 9th International ACM/IFIP/USENIX Middleware Conference (MIDDLEWARE 2008)*, pages 12–17, Leuven, Belgium, 2008.

[22] Marshall Kirk McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, August 1984.

[23] Dirk Meister and André Brinkmann. dedupv1: Improving deduplication throughput using solid state drives (SSD). In *Proceedings of the 26th IEEE Conference on Mass Storage Systems and Technologies*, pages 1–6, Incline Village, NV, May 2010.

[24] Jeffrey C. Mogul, Yee Man Chan, and Terence Kelly. Design, implementation and evaluation of duplicate transfer detection in HTTP. In *Proceedings of the 1st Symposium on Networked Systems Design and Implementation (NSDI)*, San Francisco, CA, March 2004.

[25] Athicha Muthitacharoen, Benjie Chen, and David Mazières. A low-bandwidth network file system. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 174–187, October 2001.

[26] Calicrates Policroniades and Ian Pratt. Alternatives for detecting redundancy in storage systems data. In *Proceedings of the 2004 USENIX Annual Technical Conference*, pages 73–86, Boston, MA, June 2004. USENIX.

[27] Sean Quinlan and Sean Dorward. Venti: A new approach to archival storage. In *Proceedings of the 2002 Conference on File and Storage Technologies (FAST)*, pages 89–101, Monterey, California, USA, 2002. USENIX.

[28] Michael O. Rabin. Fingerprinting by random polynomials. Technical Report TR-15-81, Center for Research in Computing Technology, Harvard University, 1981.

[29] Sean Rhea, Russ Cox, and Alex Pesterev. Fast, inexpensive content-addressed storage in foundation. In *Proceedings of the 2008 USENIX Annual Technical Conference*, June 2008.

[30] Drew Roselli, Jay Lorch, and Tom Anderson. A comparison of file system workloads. In *Proceedings of the 2000 USENIX Annual Technical Conference*, pages 41–54, San Diego, CA, June 2000. USENIX Association.

[31] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1):26–52, February 1992.

[32] Mark W. Storer, Kevin M. Greenan, Darrell D. E. Long, and Ethan L. Miller. Secure data deduplication. In *Proceedings of the 2008 ACM Workshop on Storage Security and Survivability*, October 2008.

[33] Cristian Ungureanu, Benjamin Atkin, Akshat Aranya, Salil Gokhale, Stephen Rago, Grzegorz Calkowski, Cezary Dubnicki, and Aniruddha Bohra. HydraFS: a high-throughput file system for the HYDRAstor content-addressable storage system. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies (FAST)*, San Jose, CA, February 2010.

[34] Carl A. Waldspurger. Memory resource management in VMware ESX server. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*, Boston, MA, December 2002. USENIX.

[35] Lawrence L. You and Christos Karamanolis. Evaluation of efficient archival storage techniques. In *Proceedings of the 21st IEEE / 12th NASA Goddard Conference on Mass Storage Systems and Technologies*, College Park, MD, April 2004.

[36] Lawrence L. You, Kristal T. Pollack, and Darrell D. E. Long. Deep Store: An archival storage system architecture. In *Proceedings of the 21st International Conference on Data Engineering (ICDE '05)*, Tokyo, Japan, April 2005.

[37] Benjamin Zhu, Kai Li, and Hugo Patterson. Avoiding the disk bottleneck in the Data Domain deduplication file system. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST)*, February 2008.