A HIGHLY RECOVERABLE FILESYSTEM FOR SOLID STATE DRIVES

by

Mohammed Alhussein
A Dissertation
Submitted to the
Graduate Faculty
of
George Mason University
in Partial Fulfillment of
The Requirements for the Degree
of
Doctor of Philosophy
Information Technology

Committee:

_____	Dr. Duminda Wijesekera, Dissertation Director
_____	Dr. Bernd-Peter Paris, Committee Member
_____	Dr. Edward Wegman, Committee Member
_____	Dr. Paulo Costa, Committee Member
_____	Dr. Stephen Nash, Senior Associate Dean
_____	Dr. Kenneth S. Ball, Dean, Volgenau School of Engineering

Date:_____ Spring Semester 2014
George Mason University
Fairfax, VA

A Highly Recoverable Filesystem for Solid State Drives

A Dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy at George Mason University

by

Mohammed Alhussein
Master of Science
Royal Holloway University of London, 2004
Bachelor of Computer Science
King Saud University, 2002

Director: Duminda Wijesekera, Professor
Department of Computer Science

Summer Semester 2014
George Mason University
Fairfax, VA

DEDICATION

To my parents Abdulaziz and Sarah, who made me the person I am today. To my loving wife Oroub, whose help and support made everything possible. To my daughter Sarah, this is for you.

ACKNOWLEDGEMENTS

First of all, I would like to express my deepest gratitude and thanks to my advisor Prof. Duminda Wijesekera for all his help, encouragements and support throughout the PhD process. Without his guidance, this dissertation would not have been possible. I would also like to extend my deepest thanks to my PhD committee members, Prof. Bernd-Peter Paris, Prof. Edward Wegman, and Prof. Paulo Costa for giving me their valuable time for discussions and feedback.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF EQUATIONS

LIST OF ABBREVIATIONS

Solid State Drive ..... SSD
Recovery Filesystem ..... RFS
Recovery Garbage Collection Scheme ..... RGCS
Filesystem In Userspace ..... FUSE

ABSTRACT

A HIGHLY RECOVERABLE FILESYSTEM FOR SOLID STATE DRIVES

Mohammed Alhussein

George Mason University, 2014

Dissertation Director: Dr. Duminda Wijesekera

Recovering deleted information from storage drives is a long-standing problem. Prior research has approached information recovery by developing file-carving techniques. However, two issues present significant challenges to on-going efforts. 1) Prior knowledge of file types is required to construct file carvers, including file headers and footers, and 2) fragmentation prevents file carvers from achieving successful recovery. More recently, solid state drives or "SSDs" have become more popular. SSDs provide several advantages over traditional mechanical hard drives. They have smaller sizes, are constructed without moving parts, and provide better performance. However, due to problems such as wear leveling and write amplification in SSDs, files are severely fragmented and thus exacerbate the data recovery problem. In addition, SSDs use TRIM and garbage collection schemes to enhance their performance, which can permanently delete data immediately after a delete operation.

In this dissertation, I developed a framework for recovering deleted files without knowing the file types and despite significant fragmentation. I developed the Recovery Filesystem by modifying an existing implementation of the exFat filesystem running on top of FUSE. The central idea underlying the Recovery Filesystem is a special identifier embedded in each data block. The identifier monitors each block by mapping the data block to a single file regardless of the file status, existing or deleted. The block sequence number and creation timestamp are also maintained to facilitate the recovery process. In addition, I developed a garbage collection scheme for SSDs that maximizes data retention without sacrificing SSD performance.

The experiments conducted in this dissertation demonstrate that the Recovery Filesystem yields acceptable read/write performance results. In addition, file recovery experiments used to compare the Recovery Filesystem with open source recovery techniques demonstrate that the Recovery Filesystem provides significant advantages in the case of fragmented data.

CHAPTER ONE: INTRODUCTION

1.1 Motivation and Problem Statement

The demand for digital data storage has rapidly increased. Currently, people rely heavily on computer systems to store personal, medical, financial, and other types of important information [1]. Corporations and governments also rely on computer-automated processes; thus, digital data storage is critical [2, 21]. To accommodate this increased demand, data storage solution providers have, in turn, produced ever-increasing storage capacities [3].

With this dependence on digital storage devices, the consequences of lost data are grave. A survey conducted by EMC in 2013 included 2300 organizations from around the world, and 29% reported a data loss incident within a year from the time that the survey was conducted. The average loss in revenue reported due to data loss was $585,892 per organization [4]. In a study by Carnegie Mellon University, hard drives used by data centers and internet service providers exhibited annual failure rates of up to 13% [5].

There are multiple causes for data loss. Human error is, by far, the most frequent reason data are lost. People often accidently delete files, and in certain cases, recovery attempts are difficult [6]. Further, computer users occasionally mismanage applications

by upgrading or configuring the applications incorrectly, which results in data loss or damage [7]. Hardware failure is another reason for data loss, especially where hard drives that contain mechanical parts are used [8]. In addition, software corruption can also cause data loss or damage, either because the software behavior deviates from expectations or due to compatibility issues [9]. Computer viruses and deliberate attacks on the computer system can also lead to data loss. In fact, corporations and government organizations are often attacked with the primary goal of deleting data [10].

To mitigate the data loss problem, the data recovery process was gradually created and has been enhanced [11]. Multiple techniques can be used to recover data from storage devices, depending on the nature of the deleted content and the associated file system. When a user deletes a file, the file system information linked to the deleted file remains intact, and recovery of the deleted file is a straightforward operation in most cases. For example, in the file allocation table, aka the "FAT" file system, deleting a file marks the corresponding cluster entries as empty (i.e., 0) in the FAT table. However, the information indicating this file remains until the corresponding entry in the FAT table is associated with another file. Where the indicative information is no longer available in the FAT table or the file system is corrupted, data recovery is more challenging and requires more sophisticated techniques to compensate for the absence of meta-data leading to the location of the file in the storage unit. Such techniques are referred to as file-carving techniques.

Pal et al. defined file carving as "a forensics technique that recovers files based merely on file structure and content and without any matching file system meta-data"

[12]. Certain file carvers use a file structure, such as the file header written by the user application, to recover data. More advanced file carvers use the file content to recover data by employing statistical and/or artificial intelligence techniques. Although file carving is a powerful technique for data recovery, I highlight two issues that present challenges for data recovery efforts:

- Prior knowledge of file types and file content
- File fragmentation

To construct a file carver, the file types and content must be known. Although, many file carvers can currently handle numerous file types, this problem remains for new file types. The second and more important problem is fragmentation. Recovering fragmented files presents significant challenges to file carving, as demonstrated in [13]. As the number of fragments increases, the problem becomes more difficult.

More recently, NAND-based flash solid state drives "SSDs" have become more popular [14]. SSDs provide several advantages over traditional mechanical hard drives. They are smaller, are constructed without moving parts, and provide better performance [15]. In addition to these advantages, the cost of SSDs has significantly decreased since they were first introduced, which has contributed to their widespread use [16]. Unfortunately, SSD use exacerbates the data recovery problem by rendering the recovery process much more difficult. One area in which SSDs are problematic from a data recovery perspective is fragmentation. Due to the design of NAND-based flash memory technology, files stored on SSDs are often severely fragmented [17]. SSDs require that

individual memory cells be erased before data can be written to the memory blocks [18]. However, each block can only be erased a limited number of times before the block becomes unreliable [19]. Thus, to prolong the life of an SSD, manufacturers utilize wear-leveling techniques that attempt to distribute memory block use across the entire drive. Thus, data are not written to the SSD contiguously but are fragmented depending on the erase cycle count of each block. In addition, because data cannot be overwritten in the same block, when the files are updated, more fragmentation occurs because the updated data are moved to different blocks.

To compound the problem, SSDs use a technique referred to as garbage collection to improve the data-writing performance. As mentioned earlier, when the filesystem requests that data be written to a particular "dirty" block that holds previously deleted data, the dirty block must be erased before the SSD can write the new data, which results in an additional erase operation. When the number of dirty blocks increases significantly, the block selected for the "write" operation is more likely to be dirty, which results in additional erase operations and, in turn, decreases performance.

The SSD community addressed this problem by introducing garbage collection and the TRIM command [20]. TRIM enables the filesystem to inform the SSD of which blocks are deleted and can, therefore, be erased. Erasing the deleted blocks can enhance write operation performance. Because these blocks are empty, they do not require an additional erase operation.

A side effect of TRIM and garbage collection that is relevant to our discussion is that deleted data can be permanently removed. Where TRIM commands are executed in real-time immediately following delete operations, data cannot be recovered [43].

In this dissertation, I introduce a filesystem termed the Recovery Filesystem, which mitigates the difficulties of data recovery discussed in this section. Specifically, the Recovery Filesystem enables deleted files to be recovered without knowing the filetypes and despite fragmentation. The central idea of the Recovery Filesystem is that file-identifying information is stored within the actual data blocks. The identifying meta-data for a cluster are stored as a file identifier, cluster sequence number, and write operation timestamp. When a filesystem is constructed this manner, recovering deleted files is a process of grouping and ordering disk clusters that belong to different files. Additionally, I introduce a garbage collection methodology that maximizes data retention in SSDs to facilitate higher levels of data recovery without sacrificing the performance benefits of using the TRIM command.

The remainder of the dissertation is organized as follows. In section 2 of this chapter, I discuss the privacy implications of my work. In chapter 2, I review the background information and related studies. Chapter 3 introduces the Recovery Filesystem. In chapter 4, I examine SSD data recovery and the Recovery Garbage Collection Scheme. Finally, in chapter 5, I discuss my conclusions and future plans.

1.2 Thesis Statement

In this dissertation, I make the following thesis statements:

1. It is possible to recover deleted files without prior knowledge of file types, and with the existence of fragmentation.
2. It is possible to increase data retention periods in solid state drives (SSDs) without sacrificing performance.

1.3 Privacy Discussion

As I discussed above, people rely on computer systems to store many types of data, including personal and sensitive information. In certain situations, it might be desirable for users that data be unrecoverable. Where a computer system is lost or stolen, for example, we do not want our search history accessed or old deleted documents recovered. Therefore, it is fair to ask why a computer users would want to adopt a filesystem that might harm or incriminate them in certain cases. To answer this question, we must first analyze what mechanisms are available to computer users to address data recovery and how the above problems can be mitigated.

Data sanitization techniques have been developed to assist computer users and ensure that unwanted data are permanently deleted [22]. Users can "shred" files, which immediately renders the files unrecoverable. In addition, available tools can enable computer users to constantly sanitize unallocated space (storage space that is not associated with existing files), which frustrates all file recovery efforts [23].

Therefore, computer users are always presented with the option of selectively or completely permanently deleting files. However, as discussed above, the ability to recover data is an increasingly difficult problem.

CHAPTER TWO: BACKGROUND AND PREVIOUS RESEARCH

2.1 Basic Data Recovery

Early data recovery efforts concentrated on recovering deleted files from information available in filesystems. When a file is created, file-identifying information, such as the location of the data blocks that belong to the file, are stored in the filesystem, typically in the file allocation table ("FAT") [24]. Figure 1 shows a typical FAT structure. When the file is deleted, it is marked for deletion; however, the data and FAT entries that correspond to the deleted file remain intact. Recovering such deleted files is a

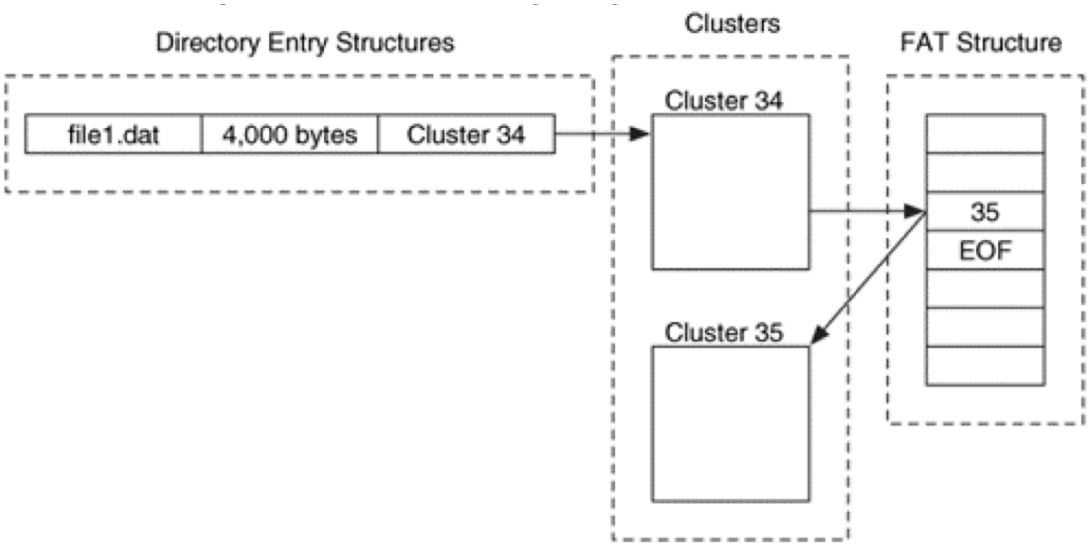

Figure 1. File Allocation Table. Carrier, B. (2005). *File system forensic analysis* (Vol. 3). Reading: Addison-Wesley

straightforward process. We simply traverse the available information in the FAT to recover the deleted file. Notably, once a file is deleted, the data blocks that belong to the file become unallocated; therefore, the data stored on these blocks can be overwritten. If the blocks are overwritten, file recovery may be partially or completely impossible, depending on how many blocks were overwritten.

Directory Entry

File A	Cluster 150
File B	Cluster 152

File Allocation Table

150	151
151	EOF
152	153
153	154
154	EOF
155	0

Figure 2. Files stored contiguously in disk

When file information stored in the FAT is lost, recovering the deleted data is more problematic. We no longer know exactly where the blocks that belong to a file are stored within the hard disk. To recover files in such instances, the data recovery community developed file carving [25]. File carving includes a set of techniques and tools used to attempt the recovery of deleted files without relying on file information stored in the filesystem. The first stages in file carving are centered on recovering deleted files based on the file structure. The file types are analyzed, and "magic numbers", which are unique signatures that are typically at the beginning and end of files, are utilized to recover the deleted files [26]. Foremost is a prominent early and successful file carver that relies on file structure [27].

2.2 Fragmentation

The problem with first stage file carvers is their requirement that all data clusters belonging to the file be stored contiguously in the hard drive. Figure 2 shows the FAT entries for two files A and B that are stored contiguously in the hard drive. Because the magic numbers are typically in the first and last blocks, it is critical that all remaining blocks be available contiguously between the first and last blocks and in the correct order. Unfortunately, files can become fragmented, in which case locating the first and last blocks of a file is insufficient for file recovery. Files typically become fragmented when contiguous space is unavailable for storing the complete file. Files can also become

fragmented when they become larger, which requires additional blocks for storage if no blocks are available contiguously. For example, consider figure 2, assuming that File A was updated, and an additional block was required to store the new data. Figure 3 shows the FAT after updating File A. As shown, File A is divided into two fragments; the first

Directory Entry

File A	Cluster 150
File B	Cluster 152

File Allocation Table

150	151
151	155
152	153
153	154
154	EOF
155	EOF

Figure 3. File A fragmented in disk

fragment is composed of blocks (150,152), and the second fragment is block 155. Relying on a magic number is not sufficient in this case, as the recovered file will contain data blocks that belong to File B.

Several studies have attempted to mitigate the fragmentation problem. The fast object validation technique [13] attempts to recover files composed of two fragments. This technique requires that be are created through an encoding process because, for data blocks, the encoding/decoding process is used to determine a block's eligibility as part of the recovered file. In [13], Garfinkle presents the challenges of recovering files that are fragmented into many fragments, and he shows how important files are often fragmented. In [25], Pal et al. demonstrate a technique for recovering text and image files from multiple fragments. Although in [28] the authors showed experimental results from files recovered using fewer than 300 fragments, the fragments were part of complete files in an experimental data set. The authors demonstrated their technique using real world data sets [29, 30] and successfully recovered files occupying up to 4 fragments. Other file-carving techniques have also been introduced to recover fragmented files. However, many techniques are designed to target specific files types, such as image and video files [31, 32, 33, 34].

2.3 Solid State Drive

2.3.1 Organization and Memory Cells

As stated in chapter one, SSDs have recently become more popular. Factors such as a smaller physical size, a lack of moving parts, enhanced read/write performance, and

the continuous decrease in SSD prices contribute to their increased use. Further, as we discussed earlier, SSDs present an additional set of challenges to data recovery, and understanding the internal operation of SSDs is necessary to mitigate these difficulties. In this section, I first provide an overview of SSDs and highlight the implications for data recovery. I then discuss the related research on SSD data recovery.

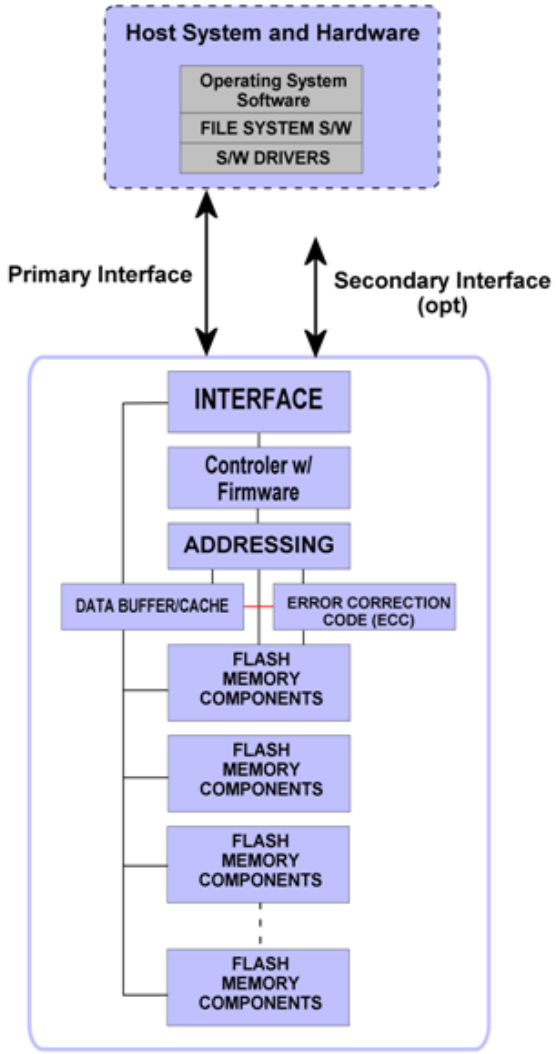

Figure 4. SSD Device Organization (source: storagereview.com)

Figure 4 shows the typical organization of an SSD. A host system is typically connected to an SSD through a SATA interface, which is similar to a traditional hard drive connection. In addition to the flash memory banks that are used to store data, the SSD contains a processor and a flash controller that maps block addresses from the logical block address "LBA" used by the host system to the physical block address "PBA" scheme used by the SSD. Figure 5 shows the block address management controller.

Flash Controller
Lookup Table

LBA	PBA
100	6545
101	3225
102	1249
103	8245
104	2478

Figure 5. The Flash Controller Lookup Table

As shown in figure 5, the sequential logical blocks are often mapped to physically fragmented blocks throughout the SSD storage space.

The basic unit of storage in an SSD is a memory cell, which is responsible for storing bits [35]. To read, write and erase data, voltage is applied to the floating gate transistors, which are the basic building blocks of these memory cells. A central concept in these memory cells, which are utilized in SSDs, is that they can only be programed to write bits from 1 to 0 [36]. Therefore, to program a particular sequence of bits to represent data, the bits must first be erased, where all bits are reset to 1; certain bits are then selectively set to 0 using the write operation.

Table 1. Maximum erase/program cycle counts

	Single Level Cell	Multiple Level Cell	Triple Level Cell
Program Erase Limit	100K	10K	5K
Bits Per Cell	1	2	3

One important property of SSDs is that the NAND memory cells have a limited program/erase cycle. After reaching a certain limit, depending on the type of memory cell utilized, the cells become unreliable, and the drive indicates that it is failed [37]. Table 1 shows the maximum erase/program cycle counts for the different cell types [38]. After reaching a certain percentage of failed blocks, the SSD will wear out and become unusable..

2.3.2 Wear Leveling

Because certain files are updated more often than others, the corresponding blocks in the SSD are subject to frequent erase/program cycles, and those blocks will wear out well before less frequently used data blocks. To overcome this problem, SSDs utilize a technique referred to as wear leveling to spread the cell wear throughout the SSD [39]. The early stages in wear leveling focus on dynamic wear leveling. The idea underlying dynamic wear leveling is that every time a write operation is requested, the empty block with the lowest erase count is selected upon which the data are written, which ensures that the wear is always spread throughout the available data blocks. As discussed in [40], dynamic wear leveling can extend the life of an SSD up to 4 years in certain cases. In contrast, not using wear leveling can reduce that number to less than one year.

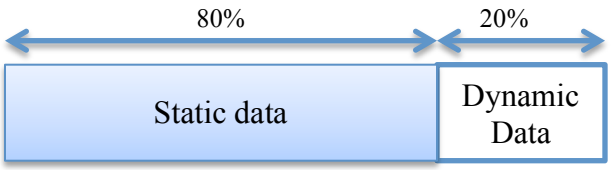

Figure 6. An SSD with 80% static data and 20% dynamic data

The problem with dynamic wear leveling is that certain files are static and are updated less frequently, if at all. For example, once read-only files are written to the

storage unit, they can be stored in specific blocks for the life of the SSD. However, they will not be available for the dynamic wear leveling process, which reduces the pool of available blocks and thus accelerates the wear of the drive. For example, where an SSD is used for multimedia storage, the static/dynamic block ratio resembles the ratio in figure 6. As shown, only a small portion of the blocks is available to the wear leveling algorithm, which dramatically reduces the life of the SSD. To combat this problem, static wear leveling was introduced to further prolong the life of the SSD. In static wear leveling, the blocks with the lowest erase counts are used regardless of whether they contain data [40]. Currently, when a write operation is requested, data blocks that hold static data will be moved to locations with higher erase counts, leading to an even distribution of block wear across the entire SSD. As shown in [40], through static wear leveling in specific use scenarios, the SSD life can be prolonged to over 15 years, whereas the life of the SSD is approximately 4 years when dynamic leveling is used under the same conditions, as indicated above.

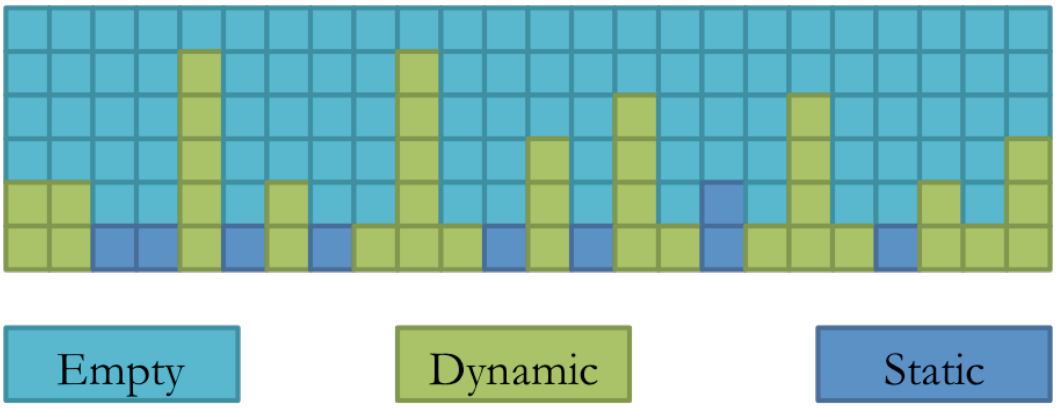

Figure 7. Wear Leveling

Although static wear leveling is crucial to prolonging the life of an SSD, the write operation performance is negatively impacted due to the overhead involved in moving data from static blocks and erasing those blocks before writing the data.

2.3.3 TRIM and Garbage Collection

Because data blocks must be either empty or erased before data can be written, it is advantageous to have as many empty blocks as possible, which leads to a higher probability that the selected block with the lowest erase count is empty. Because static wear leveling reduces the write operation performance, as stated above, it is paramount to further increase the number of empty blocks.

A method that can be used to increase the number of empty blocks is targeting dirty blocks. When a file is deleted, the blocks that belong to the file are marked as deleted in the FAT and as unallocated in the allocation bitmap. However, the SSD

indicates that they hold data, and the delete operation is not communicated to the SSD. The main reason for this observation is that, traditionally, modern filesystems were design to primarily interact with mechanical hard drives, where erasing the data from the data blocks does not benefit hard drive performance. Therefore, the TRIM command was introduced, which enables the filesystem to inform the SSD of blocks that are no longer needed and can therefore be erased. In turn, the SSD can erase these blocks, increasing the number of empty blocks through garbage collection algorithms.

Part of this dissertation is dedicated to examining the effects of the aforementioned SSD properties on data recovery and introducing solutions that facilitate data recovery by SSDs.

2.3.4 Research in SSD recovery

Researchers have identified SSDs as a new challenging issue in data recovery. In [41], Bell et al. showed that traditional data recovery techniques, which are suitable for mechanical hard drives, cannot be relied on for SSD recovery. Bonetti et al. presented comprehensive block box testing on SSD drives [42]. Their testing shows a nearly 0% recoverability rate when TRIM garbage collection is enabled. King et al. only recovered 27% from SSDs using TRIM [43]. Nisbet et al. compared SSD recovery across three filesystems, namely Windows, Linux and OS X. Their results illustrate the problem introduced by the TRIM command, from which little data remains for recovery. In addition to data recovery complications introduced by the TRIM command, researchers

focused on how fragmentation is more prevalent in SSDs. In [44], the author suggests that SSDs incur more file fragmentation due to wear leveling.

CHAPTER THREE: RECOVERY FILESYSTEM

3.1 Architecture of the Recovery Filesystem

3.1.1 Introduction

A cluster is the smallest unit of storage on disk that is used by the operating system to read and write data. Instead of reading and writing sectors in 512-byte increments, it is more efficient for operating systems to use larger blocks of data ranging, for example, from 512 bytes to 64 KB in an NTFS filesystem. Application data are written by the operating system in blocks (block and cluster are used interchangeably throughout the dissertation) equal to the cluster size; therefore, it is reasonable to consider file recovery in terms of identifying the blocks that comprise the file. As described, traditional file recovery techniques, or file carving, require knowing the file types for recovery processes to function. The file cluster contents must be analyzed to perform file recovery operations; such operations are also difficult to apply when a file is fragmented into even a few fragments, and they become almost unusable when the number of fragments significantly increases. The goal of the proposed Recovery Filesystem is to recover deleted files regardless of the number of fragments within a file and without knowing the file types.

The central concept behind my Recovery Filesystem is to embed file-identifying information within data clusters. We reserve a number of bytes at the end of each cluster to record the file ID associated with the cluster and the sequence number of the cluster within the group of clusters that comprise the file. By recording the file ID and the cluster position in the file, we can precisely and efficiently determine where the cluster belongs and its order.

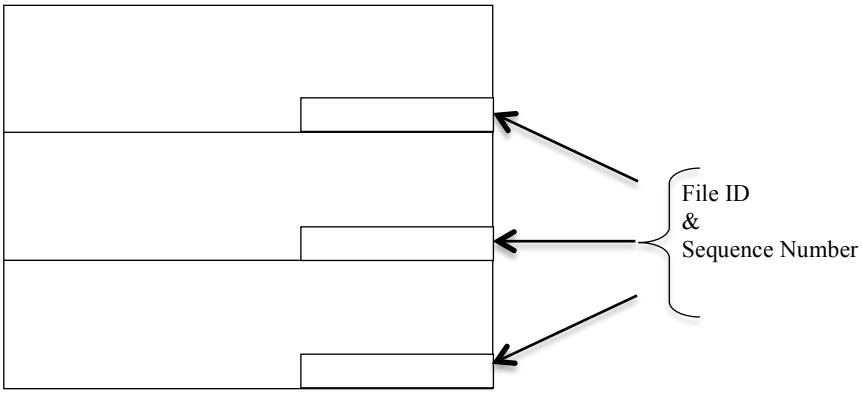

Figure 8. Three clusters showing the identifier

3.1.2 Recovery Filesystem Identifier

The identifier stores the cluster file ID and sequence number. To determine the size required to store the identifier, we must calculate the maximum number of files that can

be stored and the maximum number of clusters that can be in a file. Figure 8 shows the identifier.

Based on an implementable version of NTFS, the maximum number of files in an NTFS volume = $2^{32} - 1$, which requires 4 bytes to store the file ID. Further, for a 4 KB cluster system, the maximum number of clusters per NTFS volume = 2^{32} clusters, which also requires 4 bytes to store the cluster number. Therefore, we require 8 bytes to store the entire identifier.

3.1.3 Timestamps

Standard file operations, such as write, append, delete, and truncate, can affect a file differently. More specifically, in the Recovery Filesystem, updating a file can yield multiple clusters that share the same file identifier and sequence number. When a file is truncated to a smaller size, certain clusters are deleted. However, when we append certain additional data to the file, the file size increases and will therefore be allocated into new clusters with the same file identifier and sequence number as the deleted clusters. This process can lead to an interesting situation when we attempt recovery of a deleted file. We have multiple clusters with the same file ID and sequence number. Table 2 illustrates how this situation can arise.

Table 2. Effect of Truncating and Appending.

201	File A	ID=1, SQ=1	201	File A	ID=1, SQ=1
202	File B	ID=2, SQ =1	202	File B	ID=2, SQ=1
203	File A	ID=1, SQ=2	203	**Unallocated**	ID=1, SQ=2
(1) FAT			(2) FAT after deleting a cluster from File A		
201	File A	ID=1, SQ=1	201	File A	ID=1, SQ=1
202	**Unallocated**	ID=2, SQ=1	202	**File A**	ID=1, SQ=2
203	Unallocated	ID=1, SQ=2	203	Unallocated	ID=1, SQ=2
(3) FAT after deleting File B			(4) FAT after appending a cluster to File A		
201	Unallocated	ID=1,SQ =1			
202	Unallocated	ID=1,SQ =2			
203	Unallocated	ID=1,SQ =2			
(5) FAT after deleting File A					

Clearly, we require additional information to render the recovery processes deterministic in this situation. Here, I introduce a timestamp to the file identifier that records the time the cluster was written. The most recent timestamp can be used to select the correct cluster. Moreover, using multiple clusters with different timestamps can also facilitate recovering multiple versions of a file based on the timestamp value.

3.2 Environment and Consideration

3.2.1 Introduction

To embed clusters with file identifying information, changes to the reading/writing mechanism of the operating system must be implemented. Operating systems must ensure

that the Recovery Filesystem identifier is embedded in the cluster while performing write operations and must correctly remove the Recovery Filesystem identifier when reading data to maintain transparent operations to the applications served.

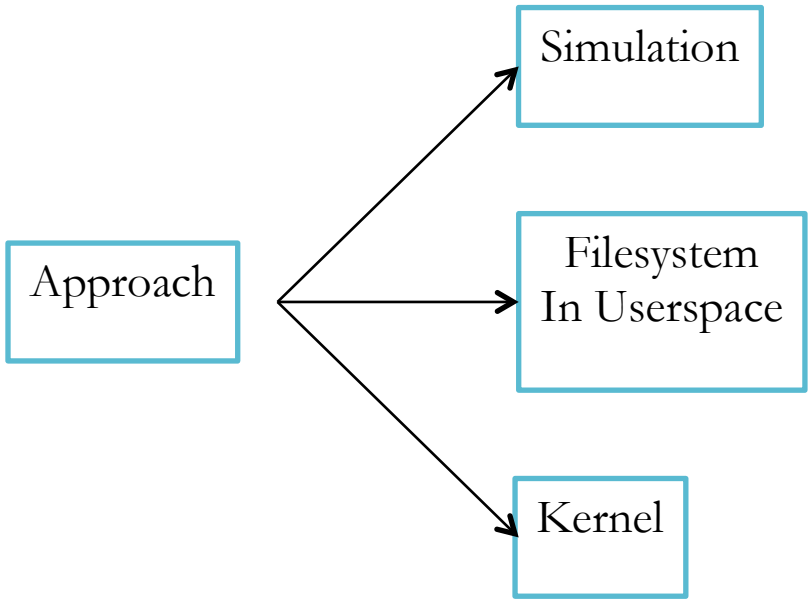

Figure 9. Recovery Filesystem Approach

3.2.2 Approach

Three approaches are available to implement the proposed system. The first approach is to utilize a filesystem simulation environment, where different modules simulate the different functions of a filesystem. While considering this approach, I evaluated DiskSim [59] and MOSS [60] filesystem simulators as potential candidates. Both simulation environments provide detailed file system functionality. Moreover, it also appeared that

customization would be possible to accommodate the necessary changes required for the Recovery Filesystem. Two issues however deterred me from selecting this approach. First, since I will be introducing overhead when embedding the file identifying information and the relevant operations to support this process, it will be vital to evaluate the performance penalty from such overhead in an actual full-fledged filesystem. This is especially true since the changes will affect both the different filesystem and the operating system modules. The second issue relates to the data recovery process, where the effects of the operating system, filesystem and storage device all contribute to the state of the data and the how recoverable it is. Therefore, it would be ideal to utilize an actual implementation of a filesystem instead of relying on simulation.

Two options are available for an actual implementation of the Recovery Filesystem. The first is to re-implement the kernel to embed the identifier in each cluster by changing the reading and writing mechanisms. The second option is to utilize a kernel module, such as FUSE "Filesystem in Userspace", that enables running customized filesystems without modifying the kernel [45]. The first approach is advantageous because it creates an efficient implementation; however, development is more complex and requires considerably more time. Moreover, because recompiling the kernel source code is necessary, only file systems supported by open source operating systems can be considered when selecting the file system of choice. I also believe that others in the data recovery community can easily adopt a Recovery Filesystem in user space as an alternative to kernel-modified operating systems. Therefore, I used the second approach and constructed the proposed Recovery Filesystem on top of FUSE.

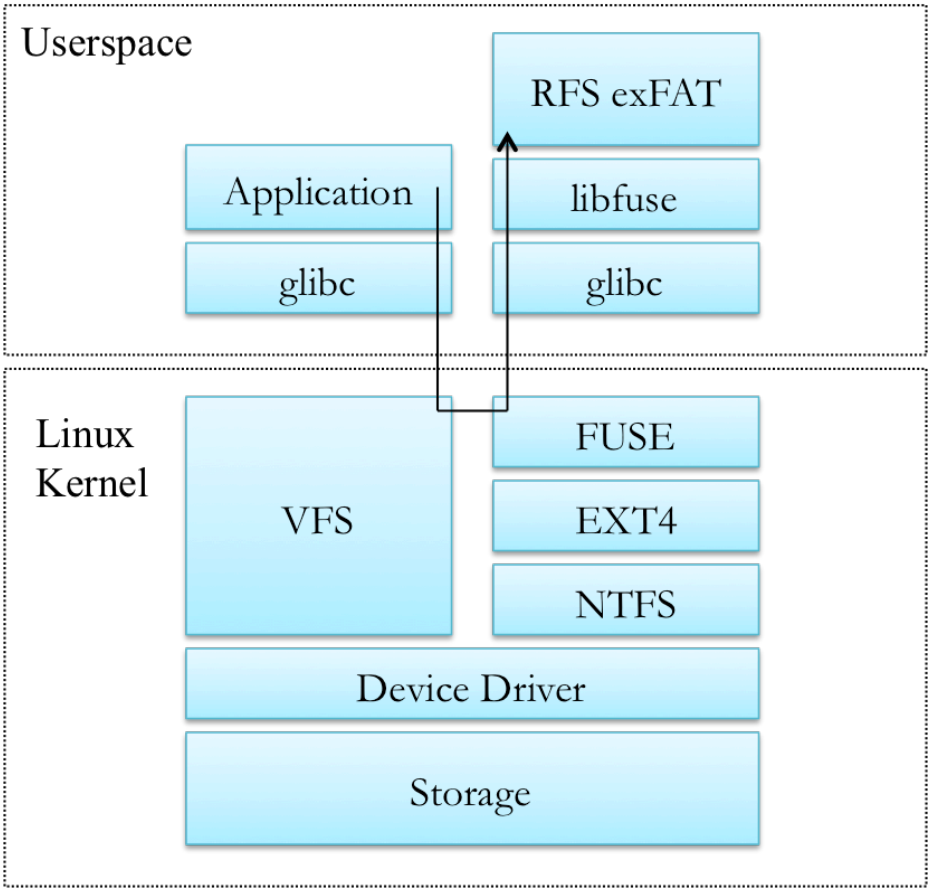

Figure 10. System Architecture

3.2.3 Filesystem in Userspace FUSE

When an application performs a filesystem operation, the request is passed to the kernel, which determines whether the file is in the FUSE volume, and is then passed to the FUSE kernel module. FUSE then forwards the request to the exFAT Recovery Filesystem, "FFS_exFAT," where the operation is implemented. Figure 10 describes how applications, kernel, FUSE and FFS_exFAT operate collectively.

3.2.4 Filesystem

The criteria that I adopted for customized filesystems in the proposed Recovery Filesystem are as follows:

- Widely adopted filesystem
- Supported by FUSE
- Ease of customization
- Maximum volume size
- Maximum file size

I considered a number of filesystems supported by FUSE, such as NTFS [46], ext3 [47], FAT [48] and exFAT [49]. One particularly important issue was the viability of customizing a filesystem without rewriting most of it. For example, NTFS-3g [50] provided an NTFS implementation on top of FUSE, but it could not interact with the clusters through the supplied APIs. ExFat was ideal for this consideration because the read and write APIs interacted directly with the data clusters.

To illustrate the changes in exFAT necessary to implement the proposed Recovery Filesystem, I will first explain how exFAT manipulates clusters to read and write. When an application requests to write to a file, FUSE passes the data in 4 KB blocks to exFAT. The data are passed through a buffer with offset from the start of the file and file information. Then, exFAT writes the information to the appropriate cluster by advancing the cluster pointer in the cluster linked list in the exfat_node structure. When exFAT

receives a read operation, exFAT begins reading the data in cluster-size increments while advancing the cluster pointer until all the data are read.

3.3 Design and Implementation

3.3.1 Introduction

The open source exFat FUSE implementation provides a filesystem implementation according to Microsoft's exFat specification [51]. I introduced a number of changes to the basic structure of the exFat filesystem to enable the support of embedding file identifying information. The standard file operations as well as directory operations were also modified to incorporate signature related functionality. Moreover, a number of new operations were also introduced to support the necessary changes to the filesystem structure.

In this section, I will introduce the Recovery Filesystem exFat structure, explaining the rational behind the introduced changes. In addition, I will also introduce the filesystem operations and how they are affected by the design principles of the Recovery Filesystem.

3.3.2 RFS Node Structure

The basic unit in the exFat filesystem that provides file information necessary for the different file operations is the exFat node structure. Cluster list, file size, and the offset of the start of the file are all contained within this structure. However, additional variables are introduced to support the functionality of the Recovery Filesystem.

The cornerstone of the Recovery Filesystem is the ability to embed a file ID in each cluster. Unlike certain other filesystems, such as ext4, which maintain a file ID as a consistent unique inode number, exFat does not maintain such information. Instead, a runtime file ID is created facilitating the compatibility of the exFat filesystem with the underlying operating system –in this case Linux- where inode numbers are required. Since runtime generated file ids are not persistent and do change every time the file is opened, they cannot be utilized reliably as the Recovery Filesystem file ID. The Recovery Filesystem node structure was modified to accommodate for a 32-bit file identifier *file_id*.

```
1. struct RFS_exfat_node
2. {
3.          struct RFS_exfat_node* parent;
4.          struct RFS_exfat_node* child;
5.          struct RFS_exfat_node* next;
6.          struct RFS_exfat_node* prev;
7.
8.          int file_id;
9.          int sequence_number;
10.         int references;
11.         uint32_t fptr_index;
12.         cluster_t fptr_cluster;
13.         cluster_t entry_cluster;
14.         off_t entry_offset;
15.         cluster_t start_cluster;
16.         int flags;
17.         uint64_t physical_size;
18.          uint64_t logical_size;
20.         time_t mtime, atime;
21.         le16_t name[EXFAT_NAME_MAX + 1];
22.        int mf;
23. };
```

Figure 11. Recovery Filesystem Node Structure

In addition to the file id, the exFat node structure also lacks a mechanism to track the sequence number of data blocks. Instead, exFat utilizes a linked list representation to navigate a file's data blocks. A 32-bit sequence identifier was introduced to enable the Recovery Filesystem to store the blocks sequence number.

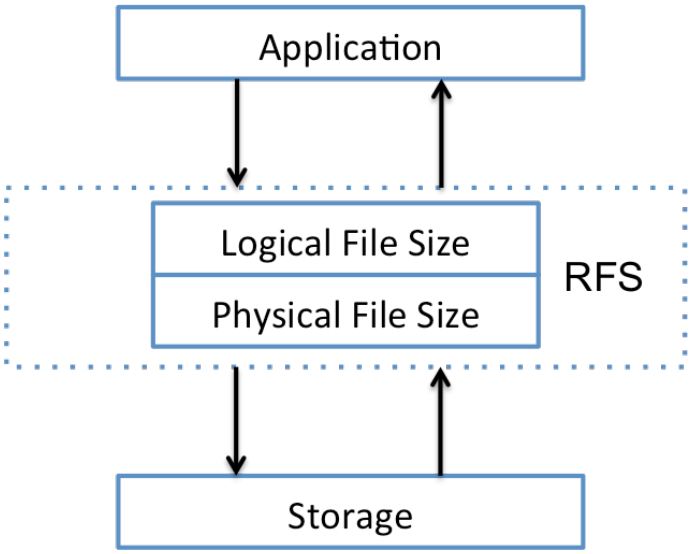

Figure 12. Logical vs. Physical File Size

In addition to the file id and sequence number, an important area that had to be addressed at the early stages of the design of the Recovery Filesystem was the issue of file size. Since the Recovery Filesystem was introducing additional data in the form of the 12-byte per block signature consisting of file id, sequence number and write timestamp, care must be taken to handle two different file size states:

- *Logical File Size:* The file size from the perspective of the applications
- *Physical File Size:* The file size from the perspective of the operating system

Maintaining the physical file size is critical to enable the filesystem to communicate correctly with the operating system's Application Programming Interface, "API". However, since applications are not aware of the embedded data, it is necessary to insure that read and write operations are transparent by utilizing a logical file size. Moreover, directory operations such as querying file size should also return the logical file size as opposed to the actual physical file size. Figure 12 shows call direction for both file size states.

To incorporate the physical and logical file size states, the exFat node structure was modified. Moreover, the Recovery Filesystem operations were also modified to communicate the appropriate file size state according to the call sequence as will be explained later in this section.

```
1. struct file_state_element
2. {
3.      char carry_buffer;
4.      int carry_sizel
5.      int RFS_offset
6.  }
7.  struct file_state
8.   {
9.        file_state_element;
10.        int index;
12.  }
```

Figure 13. File State Structure

3.3.3 File State Structure

File operations in the FUSE exFat filesystem are performed at the block level. When an application executes a write operation for example, FUSE communicates the write requests to the exFat filesystem a single block at a time, by providing the filename, application data, and the offset within the file where the data should be placed. For a 4KB block size filesystem, when an application executes a write operation of size 12KB for example, FUSE will issue three write calls to the exFat filesystem.

In the Recovery Filesystem data blocks are embedded with file identifying information. In order to do so, data equal to the size of the Recovery Filesystem signature must be extracted and temporarily stored in memory so that it can be written to the subsequent block. It is therefore important to create a file state so that data belonging to a file is written correctly and at accurate offsets. Figure 13 shows the file state structure.

To accomplish this, the File State Structure was created to keep track of the following information:

- *Carry Buffer:* The buffer where data from previous blocks are placed to make space for embedding the Recovery Filesystem signature. The size of the carry buffer is equal to the size of the block size, since a de novo write operation will be issued from the Recovery Filesystem to write the contents of the carry buffer once it is full. For the Recovery Filesystem's 4KB block size, a de novo write operation is issued once for every 1.4 MB of application data. This is

because every time 4 KB is writing, 12-bytes are stored in the carry buffer requiring 341 write operation before the carry buffer reaches maximum capacity

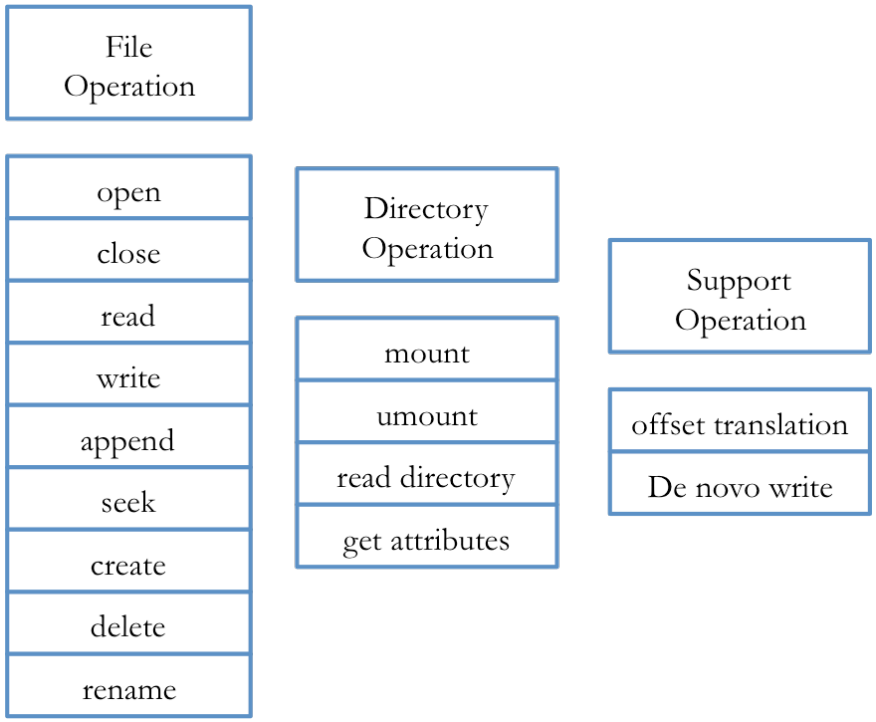

Figure 14. Recovery Filesystem Operations

- *Offset*: As previously discussed, FUSE does provide the offset for the different filesystem operations. However, since the Recovery Filesystem introduces additional data in the form of the signature, an additional offset is necessary to reflect data position with respect to the embedded data. To illustrate this concept, in the case of write operations for example, the *RFS_offset* is

required to map the logical offset received from the FUSE call to the physical addresses in storage.

3.3.4 Recovery Filesystem Operations

As described earlier, the exFat filesystem operations were modified to incorporate the functionality of the Recovery Filesystem. In this section, I will describe the different operations explaining the design and implementation choices that were necessary to enable data recovery.

- Write Operation:

  The central concept enabling the write operation to support the Recovery Filesystem is manipulating the carry buffer as described earlier. When data is received from FUSE, the RFS write buffer that is used to execute a 4 KB block write is first prepended with the contents of the carry buffer as described in figure 15. Data is then copied from the FUSE block to the RFS write buffer, leaving space for the 12-byte signature. A 32-bit write timestamp is generated and embedded along with the file id and sequence number. Additional data exceeding the 4 KB block size is transferred to the carry buffer to be written in subsequent calls. Once the carry buffer reaches maximum capacity, a de novo write operation originating from the Recovery Filesystem is executed, clearing the contents of the buffer. At this time, FUSE does not support "Big Write" operations, where data larger than the block size –but no necessarily equal to the file size- is sent

to the file system. Such support will have an impact on the performance of the Recovery Filesystem, since more data is processed at a time, requiring less memory swaps. Figure 15 illustrates the necessary steps for carrying out the write operation.

```
1.  // Make space for data in carry buffer
2.  memcpy(buffer, fuse_buffer, size);
3.  memcpy(buffer, carry_buffer, carry_size);
4.  if (size + buffer_size > block_size)
5.      memcpy(carry_buffer, fuse_buffer, (extra data))
6.   sequence_number ++;
7.      write_block(buffer,  offset,  size,  file  id,
sequence_number, timestamp);
```

Figure. 15. Write operation pseudocode

- Read Operation:

  The read operation differs from the write operation in that the input data can vary in size per FUSE operation. When data is passed to the Recovery Filesystem from FUSE, the file-identifying signature is extracted first before placing the data in a read-buffer as shown in figure 16. The signature is then verified, by comparing the file id from the data block with the file id information from the RFS node structure, where a failed verification results in a logging the file id and block information. A critical step in the read operation is the correct calculation of the offset

within a file. Since the incoming FUSE call originating from the application will request data at the logical offsets that are understood by the application, logical offsets must be translated to the physical offsets to enable reading data at the correct positions. Figure 18 illustrates the steps required for offset translation.

```
1. translate_offset(offset)
2. read_block(buffer, offset)
3. verify(id, sequence_number)
4. memcpy(fuse_buffer, buffer, size – 12)
```

Figure. 16. Read operation pseudocode

- Rename Operation:

  The performance of the rename operation contributed heavily on one of the design choices in the Recovery Filesystem, namely the construction of the file id. As discussed earlier, the exFat filesystem lacks intrinsic support for file identifiers. The first choice was to utilize the filename already available in the exFat node structure in conjunction with the file path as a unique identifier of files, since this combination is unique within the filesystem according to the exFat specification. This was to be accomplished by hashing then embedding the hash value of concatenating

the file name and the file's complete path. As shown in figure 17, a 32-bit hash function was utilized to return file ids capable of representing the maximum number of files allowed within the exFat specification. Such representation was sufficient to implement the Recovery Filesystem. However, once the rename operation was implemented, a major performance issue was encountered. When renaming a file, instead of just altering the name value within the RFS node structure, all file ids must now be renamed requiring performing write operations on the complete file. To overcome this issue, unique file ids were created and stored within the RFS node structure as explained earlier.

```
1. uint32_t create_path_hash(const char *path, size_t len)
2. {
3.   uint32_t hash, i;
4.    for(hash = i = 0; i < len; ++i)
5.   {
6.        hash += path[i];
7.       hash += (hash << 10);
8.       hash ^= (hash >> 6);
9.   }
10.   hash += (hash << 3);
11.  hash ^= (hash >> 11);
12.   hash += (hash << 15);
13.  return hash;
14. }
```

Figure 17. Hash Based File ID

- Open operation

  When a file is opened, the File State Structure for that file is created. This step is necessary to enable the establishment of a file state, linking the different operations for the file together. The File State Structure is dynamically allocated when an open operation is requested to reduce memory requirement for an initialized structure at the time of mounting the filesystem

```
1. off_t translate_offset(off_t offset)
2. {
3.     off_t loffset;
4.     loffset = offset + ((offset / 4096) * 12);
5.     if ((offset % 4096) >= 4084)
6.           loffset = loffset + 12;
7.   return loffset;
8. }
```

Figure 18. Offset Translation

- Close Operation

  In addition to filesystem maintenance issues such as releasing the file descriptor and freeing memory structures, Recovery Filesystem operations must also be performed at this stage. One important issue is dealing with

the contents of the carry buffer. A de novo write operation must be executed to transfer data remaining in the carry buffer to storage.

3.4 Performance

3.4.1 Introduction

Because the Recovery Filesystem introduces overhead into the original exFat implementation, it is important to understand the effects of the modifications on the system performance and storage requirements.

3.4.2 System Performance

To evaluate the Recovery Filesystem, I created a performance evaluation script that compares the Recovery Filesystem with the original exFat implementation. The read and write operations were used for the evaluation because they are the two file IO operations most effected by my modifications.

To ensure the best precision, we mounted the filesystem before each operation and un-mounted the filesystem after the operation to avoid syncing and caching

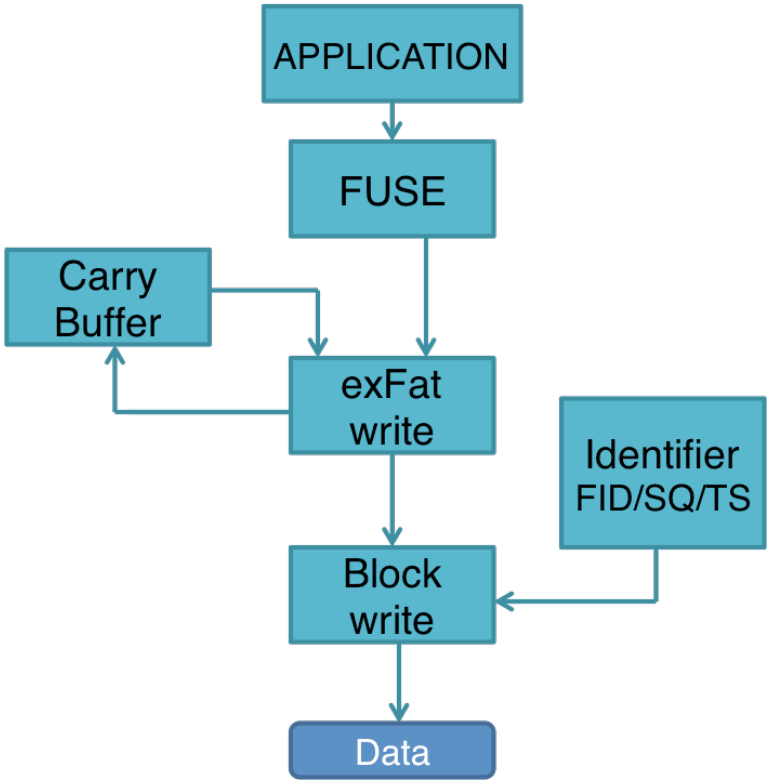

Figure 19. Read operation call sequence

problems. I also selected five file sizes and conducted the experiment one thousand times per file. Figure 22 shows the performance evaluation script.

For smaller write operations, the Recovery Filesystem requires approximately 5% overhead to complete the operations; however, this value decreases to under 2% when writing larger files. I believe that the initial overhead associated with the write operation in constructing the data structures and memory management related to the carry buffer contributes to the initially higher overhead. Table 3 shows the results of comparing the Recovery Filesystem's write operations to an original exFat implementation.

The results of the read operation evaluation are shown in Table 4 and indicate highly comparable performance with the original exFat implementation; Recovery Filesystem outperformed the original implementation for the 25 MB file size. I believe that these results are due to the read operation, which does not introduce noticeable overhead because the file-identifying information is simply verified and skipped before moving to the next read operation. Moreover, because the precision is not better, the values from the /usr/bin/time command that measures the read operation in seconds were most likely rounded; otherwise, the results would have been even more consistent.

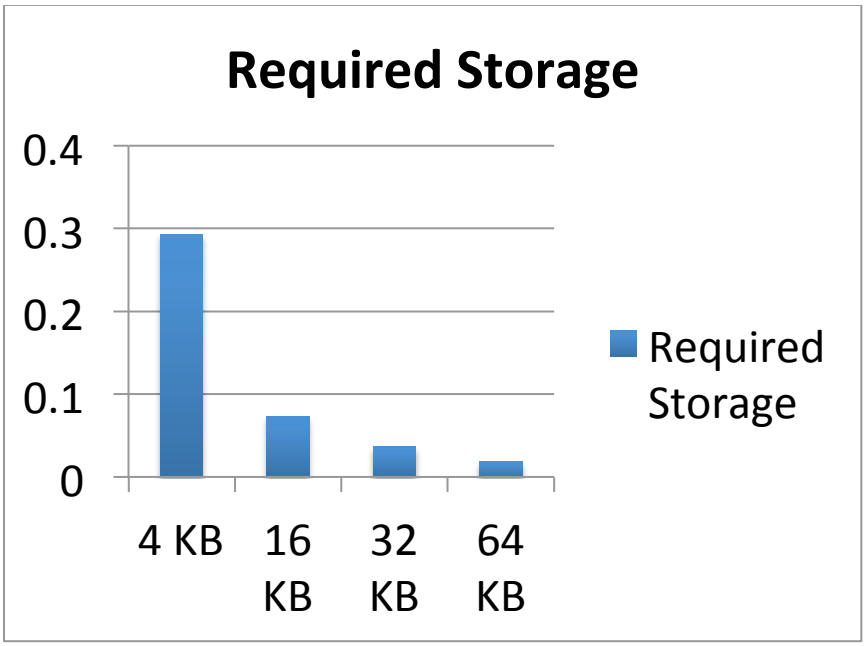

Figure 20. Storage Requirements vs. Block Size

3.4.3 Storage

The identifier stores the file ID, sequence number and write timestamp of a cluster. To determine the size required to store the identifier, in addition to the 4 bytes required for the timestamp, we must calculate the maximum number of files that can be stored in the filesystem and the maximum number of clusters in a file.

Table 3. RFS vs original exFat write operations running time in seconds

File Size	Original exFat	Recovery exFat	Performance Hit
10 MB	0.47	0.49	4.25%
25 MB	1.82	1.90	4.39%
50 MB	4.31	4.39	1.85%
100 MB	8.89	9.10	2.36%
500 MB	47.26	47.93	1.41%

Table 4. RFS vs original exFat read operations running time in seconds

File Size	Original exFat	Recovery exFat	Performance Hit
10 MB	.070	.070	0%
25 MB	0.20	.019	-5%
50 MB	0.41	0.42	2.43%
100 MB	1.64	1.69	3.04%
500 MB	7.32	7.37	0.68%

The maximum number of files in an exFat volume = 232 – 1; 4 bytes is required to store the file ID; further, for a 4 kb block filesystem, the maximum number of clusters

per volume = 232 clusters, for which 4 bytes is also required to store the cluster number. Therefore, 12 bytes of storage is necessary to store the complete identifier. To calculate the additional required storage space, we divided 12 by 4096 = 0.29%. We conclude that the Recovery Filesystem sacrifices less than 0.3% of disk space.

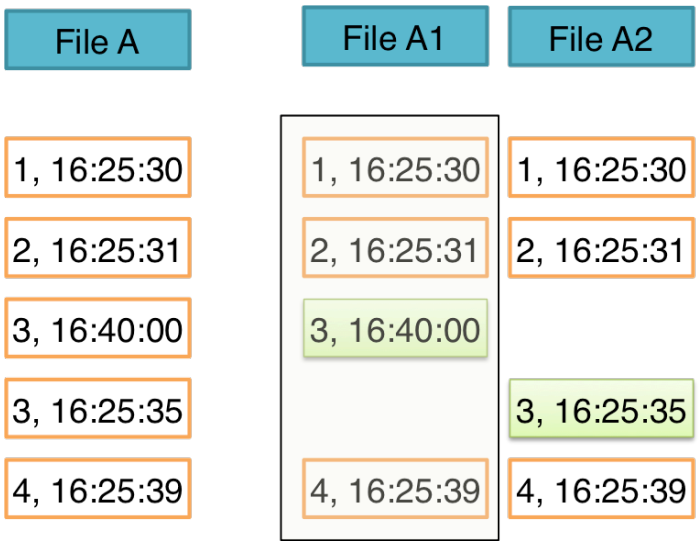

Figure 21. File Extraction Based on Creation Timestamp

3.5 File recovery experiments

3.5.1 File extractor

I created a file recovery tool that examines Recovery Filesystem exFat images to recover deleted files. The file extractor can operate using exFat images with or without the filesystem information intact. Where file information is available, the file extractor

```
1.  Initialize time = 0;
2. for file = 1 to number of files
3.     for i = 1 to 1k
4.         mount exFat filesystem,;
5.         /usr/bin/time –p dd if=file of=/mnt/exFat;
6.         time += /usr/bin/time output;
7.         unmount exFat filesystem
8.     end;
9.     output time/1k;
10. end;
```

Figure 22. Performance evaluation script pseudocode

```
1. for (i = 1; i <= total_blocks; step 1)
2.     cp random.bock, random.block.i
3. for (i = 1; i <= total_blocks; step 2)
4.     rm random.block.i
```

Figure 23. Interleaving Fragmentation Pseudocode

will leverage the information to recover the deleted files without reassembling the files using the corresponding information in the File Allocation Table. However, without this information, the file extractor will recover and reassemble all files in the analyzed image.

The file extractor recovers files by first creating a file map. A file map is a linked list of all clusters associated with a particular file ID, which are also sorted by sequence number. When the file map is constructed, the files are then recovered by transferring the

corresponding clusters to the memory for reassembly. The files are then written to a different storage device.

The file extractor is also capable of recovering multiple versions of the same file when different blocks belonging to the same file have the same sequence number. This is accomplished by sorting the blocks by sequence number followed by timestamp. As illustrated in figure 21, when two blocks share the same sequence number, two versions of the file are recovered. The multiple versions are differentiated by having a unique serial number appended to the file name, where the most recent version has the smallest number.

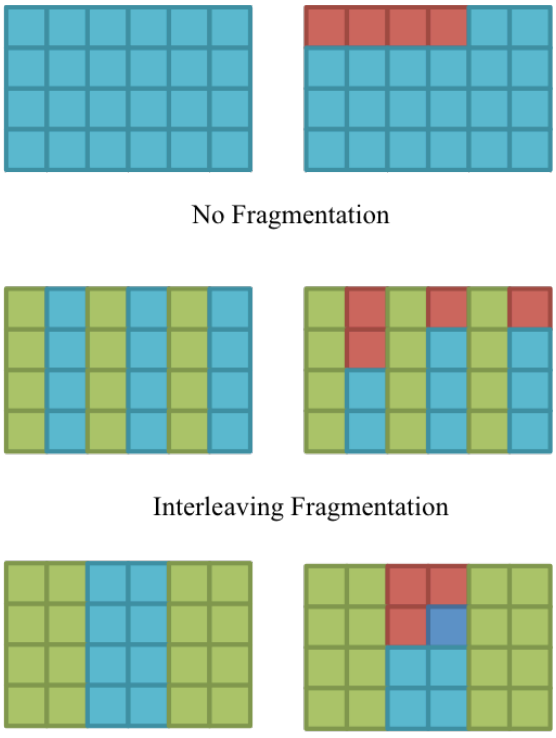

Figure 24. Fragmentation Scheme

3.5.2 File extractor vs. file carves

I selected two file carvers, photorec and scalpel, for inclusion in the file recovery experiments [52, 53]. I created an image and files of random data at 4 KB to occupy the complete image, ensuring that no clusters were entirely empty. All files were subsequently deleted to create room to perform the experiments. I then created a second image; however, only files in odd cluster numbers were deleted, which yielded an entirely fragmented image as shown in figure 23. I then created .png photo files as test files and copied the files into both fragmented and un-fragmented images. Figure 24 illustrates the different fragmentation schemes created for this experiment.

As shown in table 5, file carvers were unable to successfully recover deleted files during fragmentation. Even where the files were fragmented into only two fragments, the file carvers were unable to recover the deleted files fully. The file extractor recovered all of the deleted files that were not overwritten between the deletion and recovery times.

Table 5. Recovery rate for file extractor vs. file carvers

File Recovery Method	FS information: Yes Fragmentation: No	FS information: Yes Fragmentation: Yes	FS information: No Fragmentation: No	FS information: No Fragmentation: Yes
Scalpel	1	0	0	0
Photorec	0.92	0.15	0	0
File Extractor	1.0	1.0	1.0	1.0

CHAPTER FOUR: RECOVERY FILESYSTEM FOR SOLID STATE DRIVE TECHNOLOGY

4.1 Introduction

As discussed in chapter one, using SSDs for storage can affect the capacity for data recovery. Problems such as wear leveling, data update, and garbage collection are critical to the state of the data once deleted.

In this chapter, I will examine the challenges of data recovery in SSDs. Specifically, data fragmentation and the effects of garbage collection are investigated in detail. In section 2, I discuss the effects of SSD flash technology on fragmentation and evaluate the Recovery Filesystem that I developed against open source recovery techniques. In section 3, I examine the garbage collection problem and how it can lead to difficulties in data recovery and SSD performance. Finally, in section 4, I introduce a garbage collection methodology that maximizes data retention without sacrificing performance.

4.2 Data Recovery in SSDs

As discussed in the introduction, SSD technology has introduced significant challenges to the data recovery field. In addition to garbage collection, which is discussed in detail in the next section, fragmentation is currently considered a central obstacle to attempts at data recovery. In this section, I will investigate the different elements in SSDs that contribute to data fragmentation. I will also examine how the Recovery Filesystem that I developed can be used in SSDs to solve the fragmentation problem. Moreover, I will compare my approach with other available data recovery tools and techniques.

4.2.1 SSD Fragmentation

Several factors play a role in SSD fragmentation. Wear leveling is an important concept necessary to prolong the life of an SSD, as explained in chapter one, and it generates severe fragmentation within the physical blocks in the SSD. This type of fragmentation is minimal during the early stages of the SSD lifecycle. However, with increasing use, fragmentation dramatically increases.

Let us recall the different types of wear leveling from chapter one. Specifically static wear leveling, which selects blocks based on the cycle count, yields the best performance and is, thus, widely adopted. Because data blocks are moved in lower cycle counts, fragmentation is expected to increase with use.

Another feature of SSD that exacerbates the problem of fragmentation is updating. When data are updated, the updated data blocks are relocated into available blocks because data cannot be overwritten in a SSD.

Importantly, fragmentation can occur in both the logical block address "LBA", which is considered from the perspective of the operating system, and the physical block address ("PBA"), which corresponds to the physical layout of the SSD memory cells.

4.2.2 Recovery Filesystem for SSDs

As demonstrated in the experiments described in chapter two, the Recovery Filesystem was successful at recovering fragmented files in traditional hard drives. To test the effects of wear leveling and the SSD update problem, I conducted an experiment using an SSD.

Figure 13 shows the steps necessary to conduct the experiment. Ten photo files were created, each at approximately 10 MB. Next, after mounting the Recovery Filesystem, the files were copied. It was then important to un-mount the filesystem to ensure that all files were actually been written to the SSD and then remount it. These un-mount and mount processes were initiated each time data were written to the SSD. The files were then updated to trigger the SSD update problem before being deleted, and the file recovery technique was executed to attempt to recover the data.

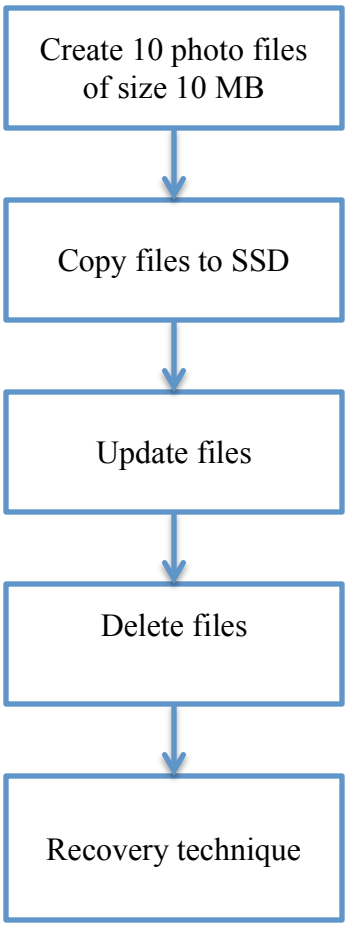

Figure 25. SSD Data Recovery Experiment

This experiment was conducted without garbage collection. I tested two open source photo recovery tools in addition to the Recovery Filesystem. As shown in table 6, both tools were unable to recover files, while the File Extractor was able to recover all of the deleted files.

Table 6. Recovery rate for file extractor vs. file carvers in SSD

File Recovery Method	FS information: Yes	FS information: No
Scalpel	0	0
Photorec	0	0
File Extractor	1.0	1.0

4.3 TRIM and Garbage Collection

4.3.1 Introduction

As discussed in chapter 1, SSDs use NAND-based flash technology, where data can be written only to empty blocks. Therefore, when the filesystem requests that data be written to a particular "dirty" block that holds previously deleted data, the dirty block must be erased before the SSD can write the new data, requiring an additional erase operation. When the number of dirty blocks is significantly increased, the block selected for the "write" operation is more likely to be dirty, which yields additional erase operations and, in turn, decreases performance.

This problem was addressed by introducing the TRIM command. Through TRIM, the filesystem informs the SSD of which blocks are deleted and can therefore be erased. Erasing deleted blocks can enhance the write operation performance. Because such blocks would be empty when a write operation is requested, they would not require the additional erase operation.

One critical side effect of TRIM involves data recovery. Once a TRIM command is issued for a particular SSD block, its contents are permanently lost. Therefore, depending on how the garbage collection schemes that trigger the TRIM commands are designed and implemented, data recovery is significantly affected.

I have identified three different existing schemes that filesystems use to implement garbage collection. In a later section, I will evaluate the effects of the different garbage collection schemes on recoverability.

```
1. struct TRIM Range {
2.     uint64_t start;
3.     uint64_t len;
4.     uint64_t minlen;
5. };
6. #define TRIM  _IOWR('X', 121, struct TRIM_Range)
```

Figure 26. TRIM system call definition

4.3.2 TRIM Support

Because TRIM was recently adopted by SSD device manufacturers, not all commercial operating systems and filesystems support TRIM. In fact, Microsoft does not support TRIM in the exFat filesystem. Because I based my Recovery Filesystem on an open source implementation of exFat, I was unable to utilize an existing TRIM implementation.

The only other option was to implement TRIM and support it using my exFat Recovery Filesystem. The greatest difficulty I faced was a lack of documentation on the subject because, as stated earlier, TRIM support is a recent addition to both operating systems and filesystems.

After investigating how the Linux operating system provides TRIM support to the ext4 filesystem through the fstrim script [54], I developed a TRIM system call for the exFat filesystem running on top of FUSE.

The basic principle of the TRIM system call design is using the input/output control system call ("ioctl"), which is used by device specific drivers [55]. Special codes have been established in technology communities for use in these calls. Figure 26 shows the definition of the TRIM ioctl used in the exFAT filesystem.

4.3.3 Evaluation of Garbage Collection Schemes

- Real-Time Garbage Collection

  In real-time garbage collection, TRIM commands are sent to the SSD immediately after a file is deleted. Recovering deleted files is impossible because data blocks are erased from the SSD immediately following a file's deletion. However, the SSD performance is improved because the deleted blocks are empty when the "write" operations are requested. In Linux operating systems, a file

system is mounted using the "mount-o discard" option implemented in real-time TRIM [56].

- Batch Garbage Collection

  Instead of sending TRIM commands immediately after a delete operation, batch garbage collection schemes group the TRIM commands so that they are sent together at certain intervals. FSTRIM exemplifies this scheme.

- Non-TRIM Filesystems

  Certain filesystems either do not support garbage collection or do not enable garbage collection by default. Without TRIM enabled, performance will decrease as a function of the increase in dirty blocks. However, data recovery is significantly improved because data will be available for recovery until the filesystem requests that a block be overwritten.

4.3.4 Analyses of Generic Garbage Collection Schemes

Enabling TRIM is critical to enhancing write operation performance. Therefore, it is reasonable to assume that using a non-TRIM scheme is disadvantageous. The two alternative garbage collection schemes offer enhanced performance over non-TRIM schemes; however, more investigations are necessary to understand the effects of TRIM and garbage collection on data recoverability.

Table 7. Data recoverability for the different garbage collection schemes.

GC scheme	Time = 0	Time = 24H
Real-Time	0%	0%
Batch-24	100%	0%
No TREIM	100%	100%

To investigate the effects of all three schemes on data recovery, I designed an experiment where data are recovered from an SSD using all three schemes. I constructed the experiment to use the Recovery Filesystem and to recover data immediately after deletion (Time = 0) and 24 hours after the data are deleted (Time = 24). Recovery was attempted a second time to allow the batch garbage collection schemes to issue TRIM commands for the deleted files. One hundred 10 MB files were written to the SSD and then subsequently deleted at time = 0. As shown in Table 7, for the batch garbage collection scheme, before time = 24 h, the deleted data were recoverable, in contrast to the real-time scheme.

Clearly, the batch garbage collection schemes provide desirable data recovery effects over real-time schemes. Moreover, larger batch intervals retain more deleted data, which therefore yields better data recovery. However, as the batch interval increases, performance is negatively affected, from which we conclude that it would be interesting to examine how performance is impacted as the batch interval size increases.

In the next section, I introduce experiments designed to analyze the performance of different batch garbage collection intervals.

4.4 Recovery-Sensitive Garbage Collection Scheme

4.4.1 Introduction

My goal is to design a garbage collection scheme that will maximize the data retention period for deleted data without sacrificing the performance benefits from utilizing TRIM in a garbage collection scheme.

In this section, I first introduce the concept of the Recoverability Ratio, which is a critical building block in the Recovery-Sensitive Garbage Collection Scheme design. I further investigate the relationship between performance and recoverability to understand how longer intervals between TRIM commands affect performance. I then introduce my Recovery-Sensitive Garbage Collection Scheme. Finally, I illustrate the effectiveness of the Recovery-Sensitive Garbage Collection Scheme through a set of experiments.

4.4.2 Recoverability Ratio

Central to my methodology, to maintain maximum data retention, it is unnecessary to execute TRIM operations unless they enhance SSD performance. As discussed in chapter two, in SSDs, if a block of data is written to, that block of data must not contain data; otherwise an erase operation is required before the write operation, which decreases performance. Therefore, it is important to understand what filesystem parameters can be used to track the filesystem use over time to study the effects of delayed TRIM commands on performance.

One option inherently available in all filesystems is to use filesystem utilization as such a parameter. Filesystem utilization measures the ratio of allocated storage space to the overall storage space as illustrated in (1).

utilization ratio = allocated blocks / total blocks *(1)*

In the literature, filesystem utilization is used as a measure in SSD TRIM-related research. However, filesystem utilization is inadequate to determine the state of the SSD in terms of recoverability because it does not consider the status of the unallocated blocks. As explained earlier, unallocated blocks can either be clean (a block that does not contain data) or dirty. The dirty blocks are important because they are the data blocks that require erasure before the write operation, which decreases performance.

Therefore, I use the ratio of dirty blocks to the total number of blocks in an SSD as a parameter to monitor the SSD usage, from which we can determine when the blocks must be erased using TRIM, as illustrated in (2).

recoverability ratio = dirty blocks / total blocks *(2)*

4.4.3 Optimal Recoverability Ratio

Understanding the relationship between the recoverability ratio and the SSD write operation performance is important in determining whether the interval between TRIM operations can be increased without significantly decreasing SSD performance.

To examine this relationship, I designed a set of experiments to measure SSD write performance at certain recoverability ratios. However, before the experiments could be implemented, the exFAT Recovery Filesystem had to be modified to produce the recoverability ratio.

```
1.  // Make space for data in carry buffer
2.  fuse_unlink(path, cluster)
3. {
4.      ….
5.      free_cluster(cluster);
6.      dirty_clusters++;
7.      ….
8. }
```

Figure 27 Tracking dirty blocks pseudocode

A variable that monitors the number of dirty blocks was introduced to incorporate the recoverability ratio. Figure 27 shows the pseudocode for the steps necessary to monitor the number of dirty blocks.

To measure SSD performance at different recoverability ratios, I used a 64 GB SSD drive to which data were written in 20% intervals. Figure 28 illustrates the different steps necessary to conduct the experiment.

The goal of the experiment was to set the dirty blocks ratio (recoverability ratio) to different intervals (stages) and then measure the write operation performance at those intervals. The first step required to construct the experiment at each different stage was to erase all blocks in the SSD and reset their cycle count, which was necessary because, in addition to erasing all blocks, the cycle count must be the same for all blocks at the beginning of each stage. ATA Secure Erase is a command that is supported by most SSD manufacturers that resets the SSD to the factory defaults based on the ATA specification; therefore, all blocks are erased and cycle counts are reset.

Further, to incorporate the effects of wear leveling into the experimental design, a disk usage simulation had to be performed to yield varying block cycle counts. This step

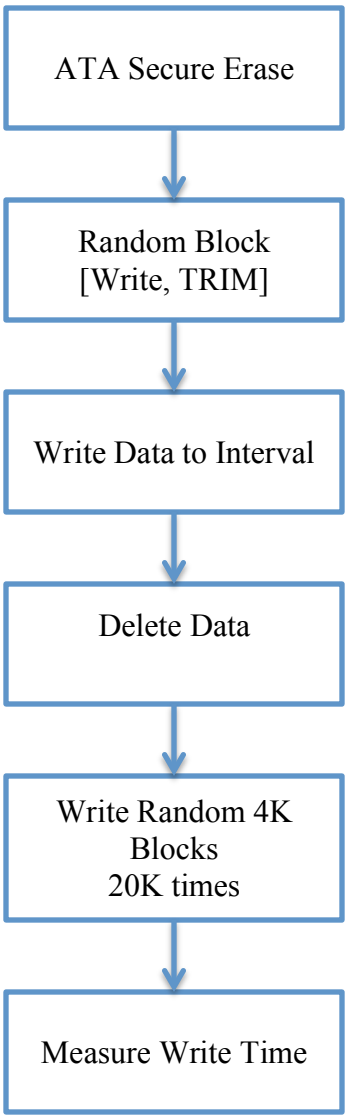

Figure 28. Optimal Recovery Ratio Experiment Steps

was important to avoid only considering the special case of a new SSD, where all blocks have the same cycle count. Therefore, after the ATA Secure Erase command, I randomly execute write operations based on the block size followed immediately by TRIM

commands to the same blocks. The number of times each block is written to is randomly selected.

In the next step, data were written to the SSD drive until the desired interval was reached, whereupon the data were then deleted. Importantly, at this time, no TRIM commands were to be executed, which would clean the dirty blocks and therefore change the status of the recoverability ratio at the given interval. Next, 4 K blocks were randomly written 20,000 times. Therefore, 80 MB of data were written, and the time required for the write operations was recorded. Each experiment was executed 10 times, and the resulting write times were the average of the 10 experiments. For this experiment, I began with a zero recoverability ratio to measure the baseline, wherein no blocks are dirty. I then performed the experiment at 20% intervals with the last stage at a 100% ratio, when all blocks are dirty, which is the worst-case scenario.

As shown in figure 29, the write operation performance did not demonstrably decrease until the recoverability ratio exceeded 40%. In addition, no significant performance hit was generated until the recoverability ratio exceeded 60%.

From the results shown in figure 29, we can better analyze the different garbage collection schemes discussed earlier.

- Non-TRIM Filesystems: Clearly, non-TRIM filesystems provide the best data retention of the three garbage collection schemes. However, although performance might initially be acceptable, it is clear herein that performance is severely impacted over extended SSD use.

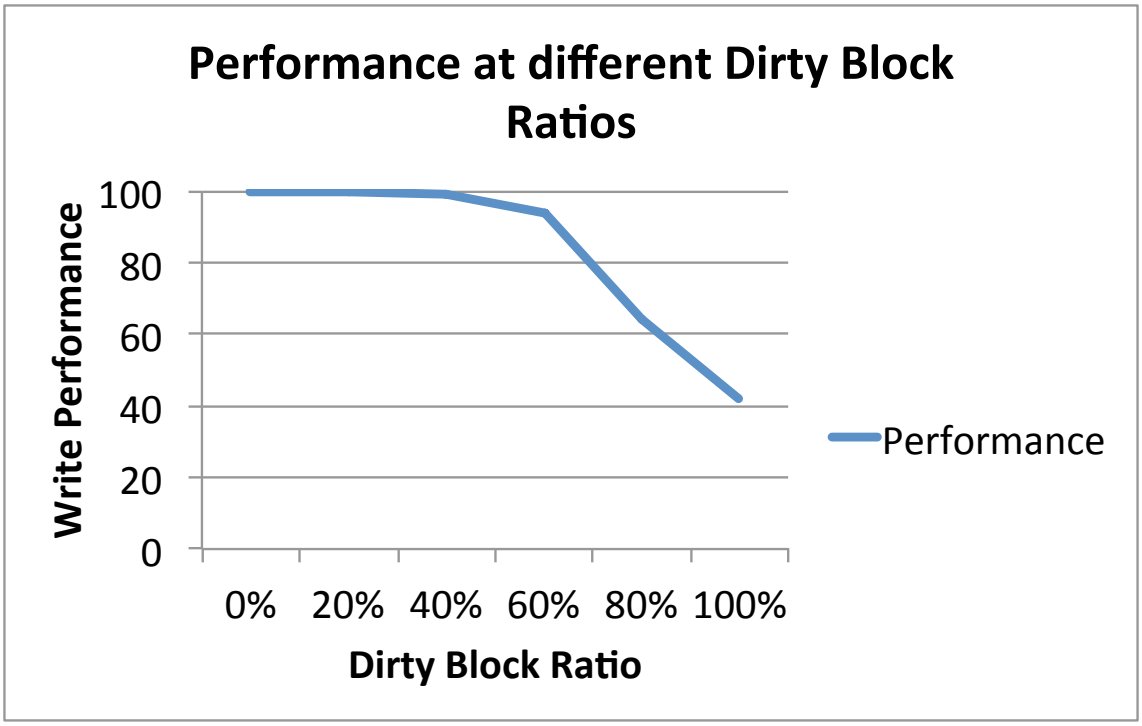

Figure 29. Write Performance for multiple Recovery Ratios

- Real-Time Garbage Collection: Real-time TRIM schemes will maintain optimal SSD write performance throughout usage. However, as shown in figure 29, for many SSD recoverability ratios, the same performance can be realized without TRIM garbage collection, which, increases data retention. It is also important to note that TRIM operations also require overhead. Therefore, for recoverability ratios under 40%, performance is not enhanced, but TRIM overhead is incurred.
- Batch Garbage Collection: It is clear that batch garbage collection provides certain advantages over the two alternative schemes. There is a certain level of data retention for deleted files, TRIM commands are not

executed in real time. In addition, it is likely to be possible to recover from degraded SSD write operation performance because batched TRIM commands are executed at certain times.

As noted, batch garbage collection provides certain advantages over the two alternative schemes. However, batch garbage collection is executed in particular conditions, under which it is triggered at arbitrary intervals. For example, in Linux operating systems, the interval is often once a day or week [54, 57]. However, for daily interval batch garbage collection, for example, actual use might generate high recoverability ratios during the first few hours, resulting in a degraded performance. However, SSD use can be minimal; therefore, the data can be permanently deleted.

In the next section, I introduce the Recovery Garbage Collection Scheme, which addresses the limitations of the three available garbage collection schemes, to provide reliable performance and maximize data retention.

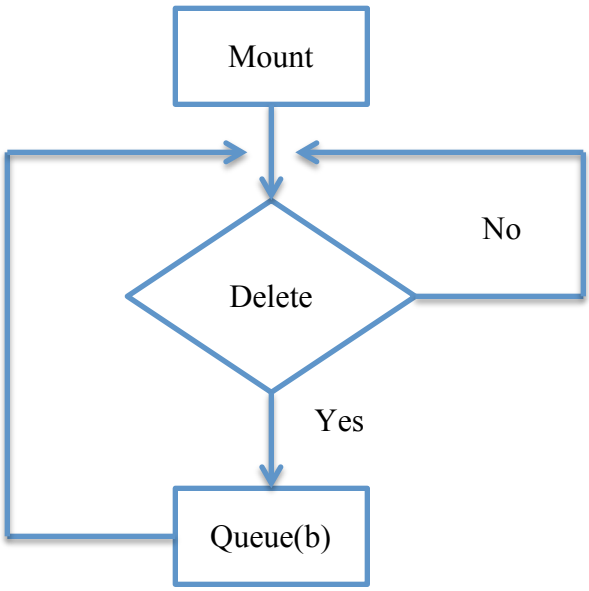

Figure 30. Recovery Garbage Collection Scheme – Queue Process

4.4.4 Recovery Garbage Collection Scheme

4.4.4.1 Motivation and Design

Central to the Recovery Garbage Collection Scheme is a methodology that will trigger TRIM commands at the precise interval points to maintain performance and maximize data retention. The first step is to analyze the optimal recovery ratio experimental results and utilize the optimal recoverability ratio as the threshold to trigger garbage collection. When we execute TRIM commands, we are presented with three different options.

- TRIM the blocks that are marked after deletion.

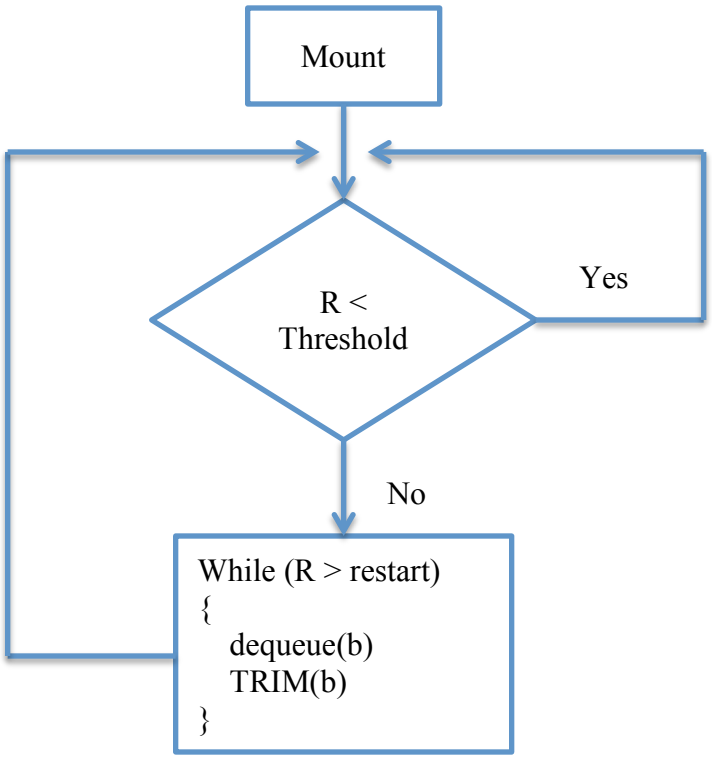

Figure 31. Recovery Garbage Collection Scheme – deQueue Process

- Switch to real-time TRIM status.
- TRIM a certain number of blocks such that the following holds.

$$Threshold > recoverability\ ratio > 0$$

The first option provides a data retention capacity. However, the drawback is that it is unnecessary to execute TRIM commands for all dirty blocks, which will permanently remove deleted data, although not immediately. Switching to real-time garbage collection at the threshold provides the maximum data retention. However, the overhead for executing TRIM commands in real-time negatively impacts performance. The third

option is an ideal solution. TRIM commands are executed for a small percentage of blocks, which I refer to as the retain ratio; therefore, real-time garbage collection is avoided, and most of the data stored in dirty blocks is retained Figure 32 illustrates a disk with 40% recovery ratio, and 35% restart ratio.

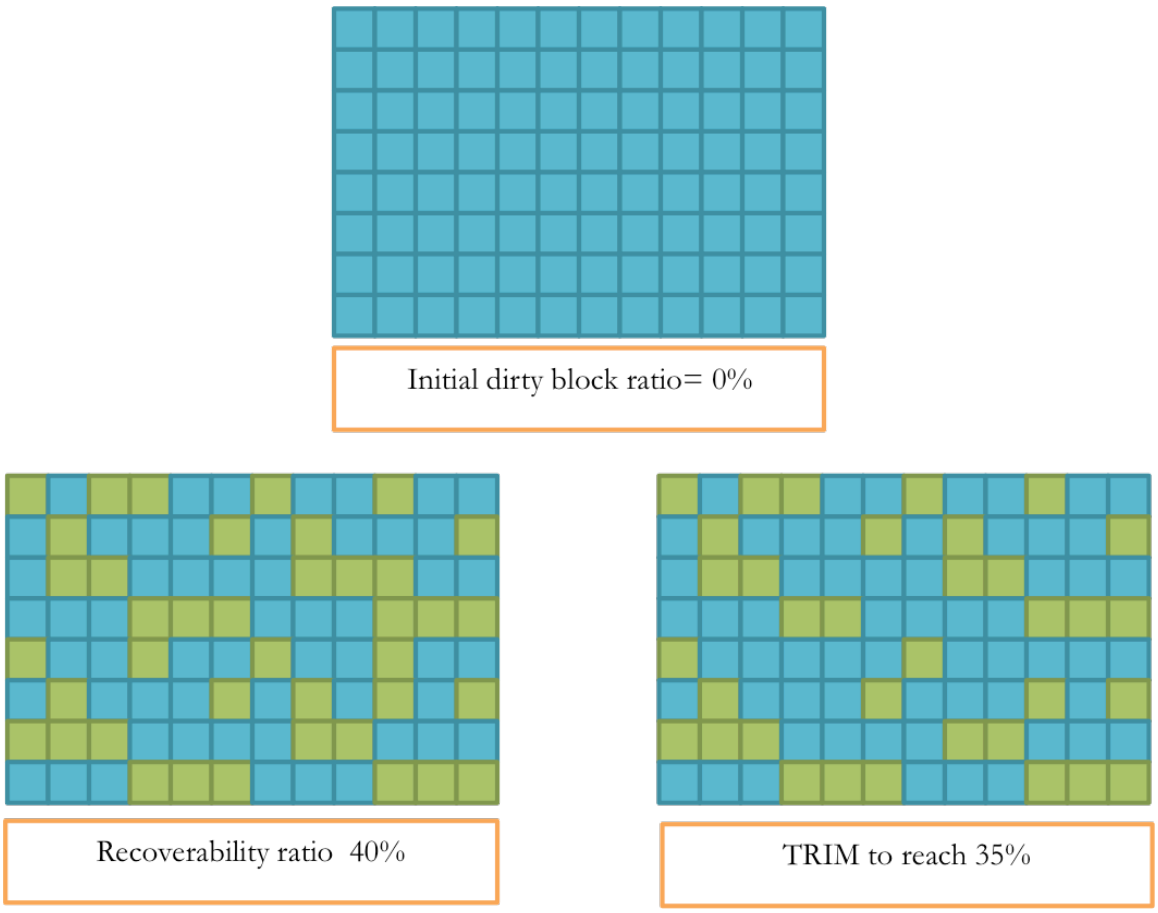

Figure 32 Recovery Garbage Collection Scheme

Another issue that can significantly impact data recovery is the order in which the dirty blocks are permanently deleted using TRIM. It is reasonable to assume that the most recently deleted data are more important, from a recovery perspective, than the least

recent data; therefore, it is advantageous to retain the most recently deleted data over the oldest data. This case holds, as explained in chapter one, where most recovery scenarios arise from events such as accidental data deletion, system and application misconfiguration, and malicious attacks that lead to data loss. Moreover, these scenarios are most susceptible to gaps in data backup solutions, whereas older data are more likely to have been backed up. Therefore, in the Recovery Garbage Collection Scheme, the older data are permanently deleted first, and the most recent data are recoverable. Figures 30 and 31 show the steps necessary for the queue and dequeue processes.

In addition to the scheme parameters discussed above, the quantity of data deleted on average are also important for demonstrating how the Recovery Garbage Collection Scheme can be used. Figure 33 demonstrates the example of a 128 GB SSD with average daily deletion levels at 100 MB, 1 GB, and 5 GB.

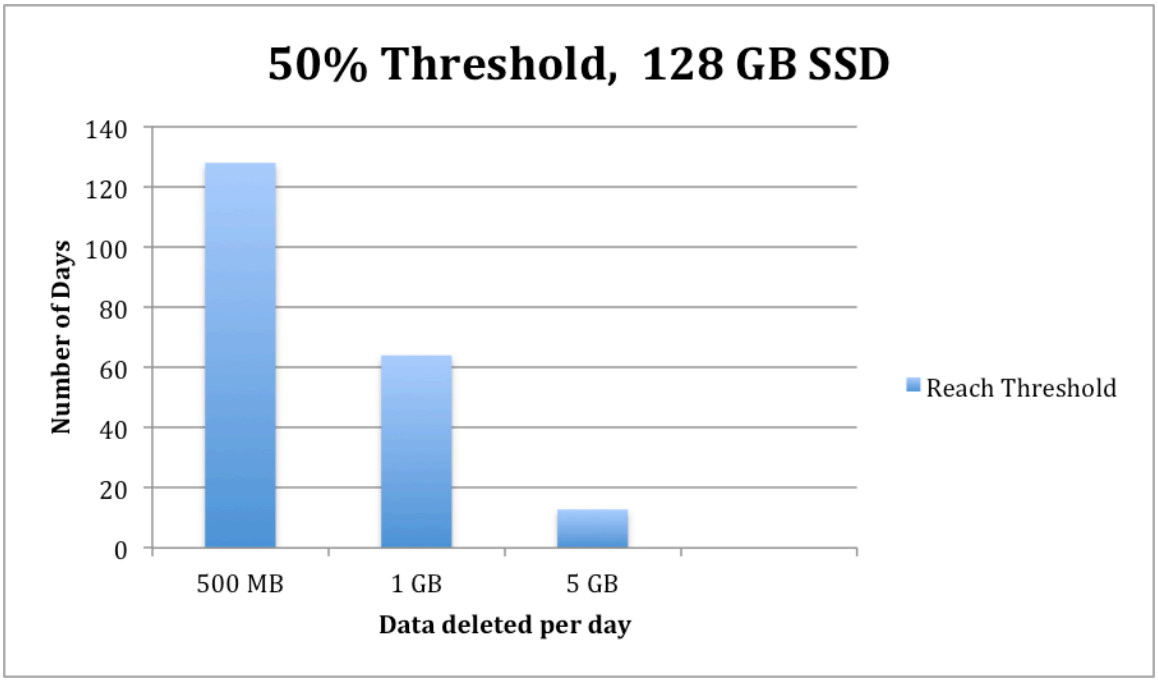

Figure 33. Number of Days required To Reach Threshold

As shown, on average, where more data are deleted, we reach the threshold more rapidly. However, data are permanently deleted until the threshold is reached, while only the oldest data that are necessary to restore the recoverability ratio to the restart factor are permanently deleted.

4.4.4.1 Implementation Details

To implement the Recovery Garbage Collection Scheme, a new data structure must be introduced. We need a way to monitor dirty blocks and to maintain the time of deletion information. I introduce a queue structure that serves as a mechanism to monitor

deleted blocks. When a block is deleted, it is added to the end of the Recovery Garbage Collection Scheme Queue ("RGCS Queue"). Further, when the TRIM commands are executed, the blocks are obtained from the head of the RGCS queue.

CHAPTER FIVE: CONCLUSIONS AND FUTURE WORK

5.1 Summary

Currently, several problems exist that render the data recovery process difficult. File carvers, which are currently the most effective file-recovery methods available, require knowledge of file encoding and structure. Moreover, file carvers do not perform well on fragmented files. In this dissertation, I introduce a recovery filesystem that mitigates the data recovery problem. Using the Recovery Filesystem, deleted files can be recovered without knowing the file types and despite fragmentation.

I utilized FUSE to modify an existing implementation of the exFat filesystem. Through experiments designed to measure the ability to recover fragmented data, I demonstrated how the Recovery Filesystem can recover 100% of the deleted data. In contrast, two open source file-carving tools were unable to recover the test files when the data were fragmented. When the Recovery Filesystem was tested against an unmodified exFat filesystem, the maximum overhead introduced was less than 5%, whereas the additional storage required for embedding file-identifying information was less than 0.5%.

In addition to the fragmentation problem, I examined the difficulties that SSDs introduce in data recovery. I illustrated how the Recovery Filesystem is particularly well-equipped to mitigate the challenges caused by the severe

fragmentation presented by SSDs. Moreover, I introduced the Recovery Garbage Collection Scheme, which maximizes data retention without sacrificing SSD performance.

5.2 Conclusions and Future Work

The problems with recovering fragmented data through file carving techniques are, in my opinion, a losing battle. With ever-increasing data storage capacities, new complex file types, and the shift to SSDs from traditional hard drives, files are becoming severely fragmented, and file carvers must become more complex. The Recovery Filesystem provides a mechanism by which such fragmentation need not impede recovery efforts. In addition, as I discussed in chapter two, the additional accessibility that enables data recovery can be controlled using data-sanitization techniques.

SSDs currently lack the standardization that previously benefited traditional hard drives. Therefore, joint efforts by the SSD community, including device manufactures as well as operating system and file system designers, are necessary to introduce greater standardization in SSDs. Such standardization will, in turn, enhance the SSD recovery process because the internal SSD operations that impact data recovery are generic, and the processes developed for data recovery can be applied to SSDs irrespective of the manufacturer.

Moving forward, the Recovery Filesystem can be applied to different areas, including computer forensics applications. Government organizations and corporations can use the Recovery Filesystem to enable more efficient investigations in computer-related or computer-assisted crimes. In such cases, more data will be reliably and efficiently recovered.

Another application for the Recovery Filesystem's file identification embedding mechanism is cloud computing. Determining jurisdiction for data that move through a cloud infrastructure is problematic. Embedding data blocks with location information will help to clarify the path of the data across the cloud infrastructure.

Finally, SSD technology is evolving. As changes are introduced into SSD controller and operating systems, continuous evaluation of such changes and their effects on SSD recovery will be necessary.

REFERENCES

1. Jones, W. Personal Information Management. *Annual Review of Information Science and Technology*, 453-504.

2. Davenport, T. H. (2013). *Process innovation: reengineering work through information technology*. Harvard Business Press.

3. Coughlin, T. (2012, September 5). Digital Storage Needs the Right Stuff. *Forbes*. Retrieved July 7, 2014, from http://www.forbes.com/sites/tomcoughlin/2012/09/05/digital-storage-needs-the-right-stuff/

4. EMC. (2013, January 1). EMC IT Trust Curve 2013 Survey Results. . Retrieved July 7, 2014, from http://www.emc.com/campaign/it-trust-curve/index.htm

5. Schroeder, B., & Gibson, G. Understanding disk failure rates: What does an MTTF of 1,000,000 hours mean to you?. *ACM Transactions on Storage (TOS)*, *Volume 3*, Article No. 8.

6. Taking Action to Protect Sensitive Data. (2007, July 1). . Retrieved July 7, 2014, from http://www.itpolicycompliance.com/research-reports/taking-action-to-protect-sensitive-data/

7. Wang, H. J., Platt, J. C., Chen, Y., Zhang, R., & Wang, Y. M. (2004, December). Automatic Misconfiguration Troubleshooting with PeerPressure. In *OSDI* (Vol. 4, pp. 245-257).

8. Ye, Z. S., Xie, M., & Tang, L. C. (2013). Reliability evaluation of hard disk drive failures based on counting processes. *Reliability Engineering & System Safety*, *109*, 110-118.

9. Smith, D. M. (2003). The cost of lost data. *Journal of Contemporary Business Practice*, *6*(3).

10. Zheng, A. K., Song, P., Han, B. X., & Zheng, M. J. (2013). Reflection of the Nation Cybersecurity's Evolution. *Applied Mechanics and Materials*, *347*, 2553-2558.

11. Casey, E. (2011). *Digital evidence and computer crime: forensic science, computers and the internet*. Academic press.

12. Pal, A., & Memon, N. (2009). The evolution of file carving. *Signal Processing Magazine, IEEE*, *26*(2), 59-71.

13. Garfinkel, S. L. (2007). Carving contiguous and fragmented files with fast object validation. *digital investigation*, *4*, 2-12.

14. IBM, (2011,September, 21). *IBM Survey: Nearly Half of IT Decision Makers Favor Solid-State Storage Technology*. Retrieved June 11 2014, from http://www-03.ibm.com/press/us/en/pressrelease/35475.wss

15. Chen, F., Koufaty, D. A., & Zhang, X. (2009, June). Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In *ACM SIGMETRICS Performance Evaluation Review* (Vol. 37, No. 1, pp. 181-192). ACM.

16. Gasior, G (2012, June, 21). *SSD prices in steady, substantial decline A look at the cost of the current generation*. Retrieved June 11 2014, from http://techreport.com/review/23149/ssd-prices-in-steady-substantial-decline

17. Roberts, D., Kgil, T., & Mudge, T. (2009). Integrating NAND flash devices onto servers. *Communications of the ACM*, *52*(4), 98-103.

18. aDam LeVenthaL, B. Y. (2008). Flash storage memory. *Communications of the ACM*, *51*(7).

19. Agrawal, N., Prabhakaran, V., Wobber, T., Davis, J. D., Manasse, M. S., & Panigrahy, R. (2008, June). Design Tradeoffs for SSD Performance. In *USENIX Annual Technical Conference* (pp. 57-70).

20. Intel, *Solid State Drives and Caching*. Retrieved June 11 2014, from http://www.intel.com/support/ssdc/hpssd/sb/CS-031846.htm?wapkw=(TRIM)

21. Mayer-Schönberger, V., & Lazer, D. (Eds.). (2007). *Governance and information technology: From electronic government to information government*. Mit Press.

22. Geiger, M. (2005, August). Evaluating Commercial Counter-Forensic Tools. In *DFRWS*.

23. Garfinkel, S. L., & Malan, D. J. (2006, January). One big file is not enough: a critical evaluation of the dominant free-space sanitization technique. In *Privacy Enhancing Technologies* (pp. 135-151). Springer Berlin Heidelberg.

24. *How FAT Works*. (2013 March 28). retrieved May 20 2014, from http://technet.microsoft.com/en-us/library/cc776720(v=ws.10).aspx

25. Pal, A., & Memon, N. (2009). The evolution of file carving. *Signal Processing Magazine, IEEE*, *26*(2), 59-71.

26. Richard III, G. G., & Roussev, V. (2005, August). Scalpel: A Frugal, High Performance File Carver. In *DFRWS*.

27. Foremost, from http://foremost.sourceforge.net

28. Memon, N., & Pal, A. (2006). Automated reassembly of file fragmented images using greedy algorithms. *Image Processing, IEEE Transactions on*, *15*(2), 385-393.

29. B. Carrier , V. Wietse , and C. Eoghan , F ile Carving Challenge 2006 [Online]. Available: http://www.dfrws.org/2006/challenge

30. B. Carrier , V. Wietse , and C. Eoghan , F ile Carving Challenge 2007 [Online]. Available: http://www.dfrws.org/2007/challenge

31. Poisel, R., Tjoa, S., & Tavolato, P. (2011). Advanced file carving approaches for multimedia files. *Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications*, *2*(4), 42-58.

32. Yoo, B., Park, J., Lim, S., Bang, J., & Lee, S. (2012). A study on multimedia file carving method. *Multimedia Tools and Applications*, *61*(1), 243-261.

33. Yannikos, Y., Ashraf, N., Steinebach, M., & Winter, C. (2013). Automating Video File Carving and Content Identification. In *Advances in Digital Forensics IX* (pp. 195-212). Springer Berlin Heidelberg.

34. Li, B., Wang, L., Sun, Y., & Wang, Q. (2012, November). Image Fragment Carving Algorithms Based on Pixel Similarity. In *Multimedia Information Networking and Security (MINES), 2012 Fourth International Conference on* (pp. 979-982). IEEE.

35. Samsung, *NAND Basics - Understanding The Technology Behind Your SSD*. retrieved May 20 2014, from http://www.samsung.com/global/business/semiconductor/minisite/SSD/global/html/about/whitepaper03.html

36. Micron, (2006). *NAND Flash 101: An Introduction to NAND Flash and How to Design It In to Your Next Product*.

37. Boboila, S., & Desnoyers, P. (2010, February). Write Endurance in Flash Drives: Measurements and Analysis. In *FAST* (Vol. 10, pp. 9-9).

38. Centon, *Flash Chip Type (TLC, MLC, SLC)*. Retrieved May 20 2014, from http://centon.com/flash-products/chiptype

39. Chang, L. P. (2007, March). On efficient wear leveling for large-scale flash-memory storage systems. In *Proceedings of the 2007 ACM symposium on Applied computing* (pp. 1126-1130). ACM.

40. Micron, (2008). *TN-29-42: Wear-Leveling Techniques in NAND Flash Devices Introduction*.

41. Bell, G. B., & Boddington, R. (2010). Solid state drives: the beginning of the end for current practice in digital forensic recovery?. *Journal of Digital Forensics, Security and Law*, *5*(3), 1-20.

42. Bonetti, G., Viglione, M., Frossi, A., Maggi, F., & Zanero, S. (2013, December). A comprehensive black-box methodology for testing the forensic characteristics of solid-state drives. In *Proceedings of the 29th Annual Computer Security Applications Conference* (pp. 269-278). ACM.

43. King, C., & Vidas, T. (2011). Empirical analysis of solid state disk data retention when used with contemporary operating systems. *digital investigation*, *8*, S111-S117.

44. Nisbet, A., Lawrence, S., & Ruff, M. (2013). A Forensic Analysis And Comparison Of Solid State Drive Data Retention With Trim Enabled File Systems.

45. Retrieved May 20 2014, from Filesystem In Userspace Web Site: http://fuse.sourceforge.net

46. *Overview of FAT, HPFS, and NTFS File Systems*. Retrieved May 20 2014, from http://http://support.microsoft.com/kb/100108

47. *Ext3 Filesystem*. Retrieved May 20 2014, from https://www.kernel.org/doc/Documentation/filesystems/ext3.txt

48. *FAT Filesystem*. Retrieved May 20 2014, from http://technet.microsoft.com/en-us/library/cc938438.aspx

49. *exFAT Filesystem*. Retrieved May 20 2014, from http://www.microsoft.com/en-us/legal/intellectualproperty/iplicensing/programs/exfatfilesystem.aspx

50. *NTFS-3G Manual*. Retrieved May 20 2014, from http://www.tuxera.com/community/ntfs-3g-manual/

51. *FUSE exFAT*. Retrieved May 20 2014, from https://code.google.com/p/exfat/

52. *Photorec*. Retrieved May 20 2014, from http://www.cgsecurity.org/wiki/PhotoRec

53. Richard III, G. G., & Roussev, V. (2005, August). Scalpel: A Frugal, High Performance File Carver. In *DFRWS*.

54. *FSTRIM*. Retrieved May 20 2014, from http://sourceforge.net/projects/fstrim/

55. Corsetti, L (2004 January 1). *Controlling Hardware with ioctls*. Retrieved May 20 2014, from http://www.linuxjournal.com/article/6908

56. *SDB:SSD discard (trim) support*. Retrieved May 20 2014, from https://en.opensuse.org/SDB:SSD_discard_(trim)_support

57. Andrew, (2013 January 15). *ENABLE TRIM ON SSD (SOLID-STATE DRIVES) IN UBUNTU FOR BETTER PERFORMANCE*. Retrieved May 20 2014, from http://www.webupd8.org/2013/01/enable-trim-on-ssd-solid-state-drives.html

58. Carrier, B. (2005). *File system forensic analysis* (Vol. 3). Reading: Addison-Wesley.

59. DiskSim. Retrieved July 23 2014, from http://www.pdl.cmu.edu/DiskSim/

60. MOSS. Retrieved July 23 2014, from http://www.ontko.com/moss/

BIOGRAPHY

Mohammed Alhussein obtained his BSc in Computer Science from King Saud University in 2002. He completed his master's degree in information Security from Royal Holloway University of London in 2004. Since then, he has worked as an information security consultant for a number of years before starting his PhD in George Mason University in 2007.