Analysis and Design of Native File System Enhancements for Storage Class Memory

by

Raymond Robles

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Approved April 2016 by the
Graduate Supervisory Committee:

Violet Syrotiuk, Chair
Sohum Sohoni
Carole-Jean Wu

ARIZONA STATE UNIVERSITY

May 2016

ABSTRACT

As persistent non-volatile memory solutions become integrated in the computing ecosystem and landscape, traditional commodity file systems architected and developed for traditional block I/O based memory solutions must be reevaluated. A majority of commodity file systems have been architected and designed with the goal of managing data on non-volatile storage devices such as hard disk drives (HDDs) and solid state drives (SSDs). HDDs and SSDs are attached to a computing system via a controller or I/O hub, often referred to as the southbridge. The point of HDD and SSD attachment creates multiple levels of translation for any data managed by the CPU that must be stored in non-volatile memory (NVM) on an HDD or SSD. Storage Class Memory (SCM) devices provide the ability to store data at the CPU and DRAM level of a computing system. A novel set of modifications to the ext2 and ext4 commodity file systems to address the needs of SCM will be presented and discussed. An in-depth analysis of many existing file systems, from multiple sources, will be presented along with an analysis to identify key modifications and extensions that would be necessary to execute file system on SCM devices. From this analysis, modifications and extensions have been applied to the FAT commodity file system for key functional tests that will be presented to demonstrate the operation and execution of the file system extensions.

DEDICATION

To Christine, Gemma, and Auggie.

ACKNOWLEDGMENTS

The first person I would like to thank is my thesis advisor and committee chair, Dr. Violet Syrotiuk. Dr. Syrotiuk took a big risk taking me on as a master's thesis student. In spite of my unique situation and thesis subject, Dr. Syrotiuk agreed to work with me and became a source of inspiration with her experience and passion for research. She taught and mentored me in my research methodology and well as in my thesis writing. I am also immensely grateful for her patience with the multiple delays in my thesis. This thesis would not have been possible without her relentless support and mentoring.

I am grateful to my committee members Dr. Sohum Sohoni and Dr. Carole-Jean Wu for their support and willingness to be a part of my thesis committee.

I would like to formally acknowledge the original authors, and Intel work colleagues, of the DAX ext2/4 code, Matthew Wilcox and Ross Zwisler. Without their vision and insight into the ext2/4 file systems, there would be no DAX.

I am grateful to my co-worker, and Linux guru, Keith Busch for answering all my silly Linux questions; Claes Olsson, my manager, who allowed me the flexibility to complete this thesis while working full-time; Carolyn Foster, my colleague, who covered for me so many times at work that I lost count.

Most importantly, I am extremely grateful to my wife Christine and her tireless, unconditional, and unrelenting love and support that provided me the motivation and strength to do that which I thought could not be done. I dedicate this thesis to you and our children.

TABLE OF CONTENTS

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

CHAPTER 1

INTRODUCTION

Problem Definition

Persistent memory solutions are becoming more prevalent in computer systems, from desktops to high-end enterprise data center solutions. As this type of non-volatile memory (NVM) penetrates the ecosystem, existing operating system software must evolve to meet the ever changing performance enhancements that accompany these persistent memory solutions. In particular, file systems have traditionally been architected and designed with the premise that all persistent storage is accessed through a traditional block I/O storage interface. However, in conjunction with new platform architectures, persistent memory solutions can be attached directly to the memory bus and central processing unit (CPU), creating a new and unique challenge for existing file systems.

At a high level, persistent memory solutions that can be directly attached to the memory bus or CPU are referred to as Storage Class Memory (SCM). SCM is a form of memory that has capacity and economics that are similar to storage but with performance that is similar to memory [4]. Essentially, SCM is memory that runs at dynamic random access memory (DRAM) speeds but is non-volatile (i.e., persistent across power cycles). To ease the transition and integration of SCM, changes are necessary to commodity file systems native to existing operating systems. To address the necessity of commodity file system modifications, emerging memory technologies must be examined. SCM solutions can be directly attached to the memory bus, bypassing any need to go through a traditional block I/O storage path, thus reducing traditional block storage I/O latency. This reduction in block storage I/O latency is the key to the required file system modifications.

The latency of an I/O can be defined as the time it takes a user space application to submit an I/O request, from start to finish. This latency includes the time spent in the operating system software, on the physical bus, at the storage device, and then back up the full stack again. SCM devices enable processors, and their running code, to access persistent storage through memory load/store instructions. Using load/store instructions on memory is orders of magnitude faster than accessing data on a traditional storage block device attached to a platform controller hub (PCH). This enables simpler and faster techniques for storing persistent data. However, access to SCM devices attached through the memory bus means the SCM device itself must be memory mapped to the system's address space. This presents an interesting problem with respect to many file system semantics such as file creation, file deletion, file security, and modifying file size due to file writes.

Another interesting aspect of this research is identifying and understanding the modifications to existing file systems that are necessary because these existing file system architectures were implemented with the premise that all persistent data storage would require block I/O requests through some type of I/O controller, as seen in Figure 1. In this diagram, normal block I/O requests would go through an I/O controller or PCH (sometimes referred to as a "southbridge") in order to access (read or write) persistent data on media such as spinning hard disk drives (HDDs) or solid state drives (SSDs). As shown in Figure 1, there are multiple "layers" necessary for accessing this data.

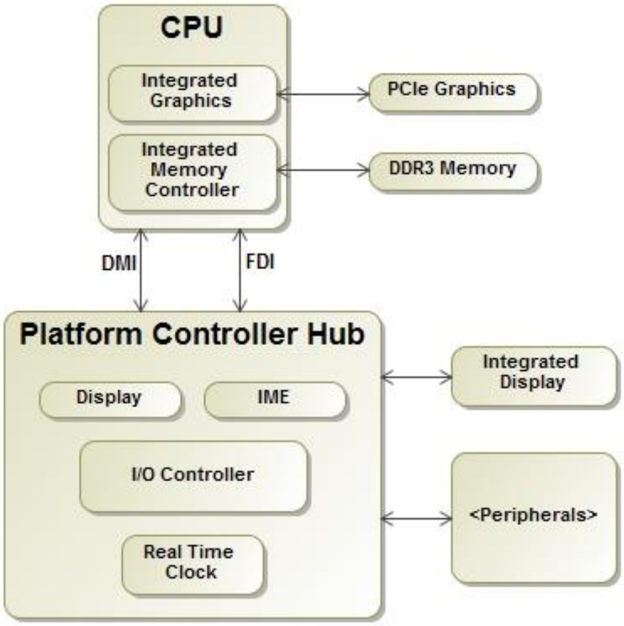


Figure 1 – General System Architecture for Storage Access

First, a CPU initiated I/O request must be sent to the southbridge or PCH. This usually means translating the CPU I/O request across a system bus, such as the Direct Media Interface (DMI) or Flexible Display Interface (FDI). This represents the first level of translation needed. Second, the storage controller in the southbridge chipset must handle the incoming I/O request and translate the I/O request from a DMI/FDI command to a storage-specific protocol request. Common storage controllers within a PCH include the Advanced Hardware Controller Interface (AHCI) which is used for Serial Advanced-Technology Attachment (SATA), Serial Attached SCSI (SAS), and Universal Serial Hub (USB). In Figure 1, the storage devices are represented by the “Peripherals” component.

These common storage protocols are bus protocols and require a second level of translation from the DMI/FDI request received to a storage protocol command. This is done by the storage controller attached to the PCH (in Figure 1, this would be represented by the controller communicating with the SATA, SAS, or USB ports and attached devices). Finally, the translated storage data request must be sent out on the physical storage link (SATA, SAS, or USB bus) to the attached device [17]. These multiple layers of translation add to the overall I/O latency, which is in addition to the actual storage access time. The CPU cycles to process the I/O becomes large and any running code/process on the CPU must give up context and handle any responses asynchronously (i.e., interrupts) in order to prevent large delays or stalls in executing code (although this is not a hard requirement as there certain scenarios where synchronous I/O is desirable). In Figure 2 below, an Intel Z87 chipset is presented to show the relationship between CPU (Intel Core Processor), Chipset (Intel Z87), and 6 x SATA ports (SATA HDDs/SSDs) [23].

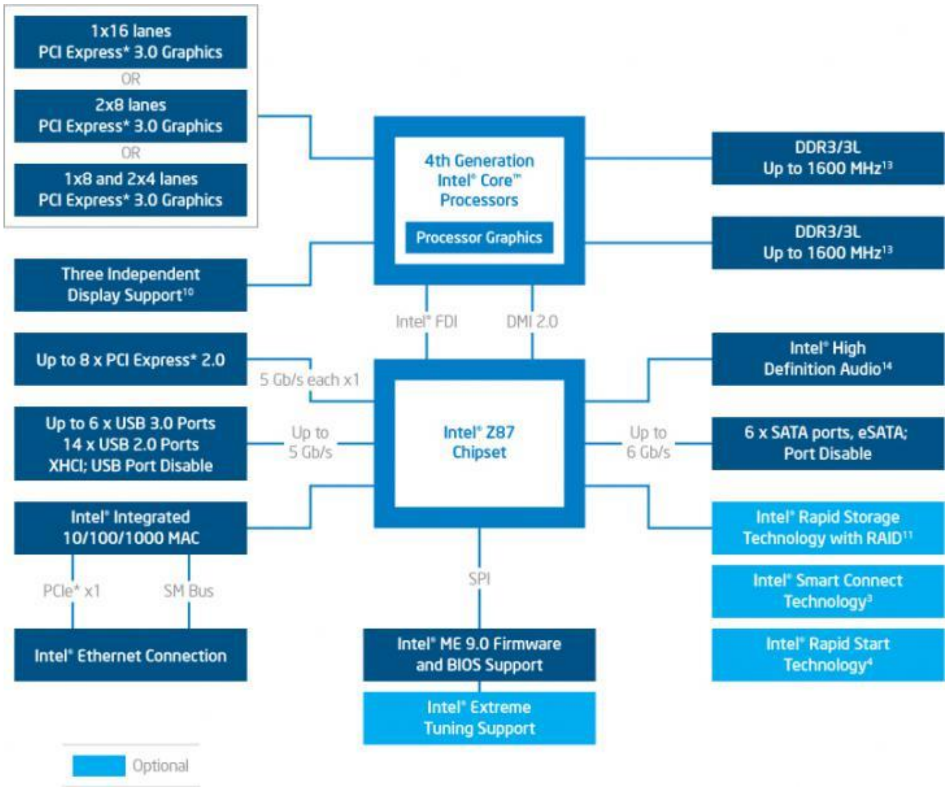


Figure 2 – Intel Z87 Chipset and Peripherals

Additional Considerations

The use of SCM directly attached to the memory bus or CPU alleviates the need to go through the southbridge to reach the persistent memory storage devices. Therefore, accessing SCM becomes similar to accessing DRAM memory, both in access latency and bandwidth. This is the key advantage to SCM; access time is drastically reduced compared to traditional block storage access. It should also be noted that SCM technology can be attached directly to the southbridge or to the storage controller on the southbridge itself. Many new SSDs are designed with a Peripheral Component Interface (PCI or PCIe, where "e" is for Express), and the internal topology of chipsets are implemented with a PCI or PCIe domain of buses. While attaching a SCM device to the southbridge does contradict the truest definition of what SCM is, there are architectures where SCM can be attached at any of these points and still meet the intent of the original definition.

In many cases, file systems are even optimized for spinning media such as HDDs by accounting for sector and head locations of previous I/O transactions. Because SCM devices will be implemented with solid state semiconductor memory, there is no need to architect or design file systems to keep track of recently accessed locations on media. Additionally, many file systems rely heavily on a buffer cache or page cache, provided by the native operating system (OS) kernel, for faster access to recent data [9]. With SCM devices, the need to use any caching disappears due to the low latency access of using loads and stores to access data. Using a page/buffer caching solution would actually add unnecessary CPU cycles to the already short I/O latency. Removing access to a page/buffer cache is also advantageous in that it reduces time spent in the file system code and resolving common cache hit/miss problems.

Additionally, when storage devices are attached directly to the memory bus they share system resources such as the bandwidth of the memory bus, the CPU cache, and the Translation Look-aside Buffer (TLB) [1]. Without modifications, the overhead of an existing commodity file system could have negative performance effects. Any file system modifications must consider these performance factors to help eliminate any unnecessary overhead. Also, when storage devices are attached to the memory bus, the overhead associated with interfacing through an emulated or generic block layer can have a negative performance impact. Modifications to the file system should take into account these overheads. SCM devices will require the need to address traditional file system operations (i.e., open, close, read, write) via memory mapped addresses (i.e., loads/stores), which is the axiom of utilizing SCM devices effectively and efficiently. Essentially, persistent memory must be treated like a memory mapped device, but at the same time with traditional functions such as opening and closing. In addition, memory mapped on a SCM device must be changed when a file is deleted, or grows from its initial size. Current native commodity file system architectures do not address these additional considerations.

In this thesis, background and relevant work on current commodity file systems, and how they may interact with SCM technologies, will be presented in Chapter 2. Additionally, in Chapter 3, a novel set of modifications from an existing prototype will be ported to an existing commodity file system to address the needs of SCM. The commodity file system used for analysis in this thesis is the File Allocation Table (FAT) file system. SCM extensions for the FAT file system for handling file creations, deletions, and I/O operations will also be presented in Chapter 3. An analysis of the research will be presented in Chapter 4. Conclusions and future work for further research in transitioning current file systems to SCM technologies will be discussed in Chapter 5.

CHAPTER 2

BACKGROUND

Motivation

Emerging breakthroughs in persistent memory, or non-volatile memory solutions, have created a need to revisit what is known and understood about current file system architectures and how they access data. Most file system architectures are based on the concept that block I/O storage requests must be used to access persistent data. As discussed, this is no longer a constraint for storage class memory. There are a current set of research efforts and prototypes addressing issue such as this research topic. Some of these efforts include hybrid flash/DRAM file systems [24], object based storage class memory file systems [1, 2], and novel file system prototypes that have been architected from the ground up [6], as well as adding extensions to existing commodity and native file systems. One particular prototype extension for the ext2 and ext4 file systems will discussed in great detail and also be used as the basis for the FAT file system extension modifications.

To best address the background and motivation of this research topic, an understanding of the current challenges is needed. The current challenges with existing file systems can be broken down in to four main categories [2]. The first category is performance overhead associated with sharing system resources that a file system would not normally access (i.e., memory management unit). The second category is contiguous file addresses for very large files. Normally, this is not an issue for block I/O storage, but because SCM requires memory to be memory mapped during run-time, consideration must be given to files that are larger than normal. The third category is run time memory mapping for file allocation. This category addresses the issues of having to create and delete files through the process of having to memory map memory at any time the system is up and running. Finally, the fourth category is unsure write ordering which describes the issue of how an SCM-aware file system should be aware of writes that complete out of order.

Performance Overhead

As discussed above, when storage devices are attached directly to the memory bus, they share system resources such as the bandwidth of the memory bus, the CPU cache, and the TLB. Without any modifications, the overhead of an existing file system architecture could prove to be too cumbersome and actually slow down overall system performance due to the sharing of critical resources. Additionally, interacting through a traditional block I/O interface will have a negative performance impact due to the overhead associated with preparing read and write requests as well as interacting with a page/buffer cache that will never be used for data stored on SCM devices. Some of the current existing solutions for this problem involve using existing resources in a new and novel manner. For example, SCMFS uses the existing memory management unit (MMU) to help speed up the process of address translation [2, 3]. Reusing the MMU provides additional support for data access from the TLB and MMU caches; this helps speed up translation operations of virtual addresses to physical addresses and provides protection mechanisms for users. However, assumptions are made in this approach to using the existing MMU. The assumption is that there is a way for firmware and/or software to distinguish persistent non-volatile memory from volatile memory attached to the same memory bus. The assumption allows for the existing MMU to be used to manage space on these SCM devices. Mappings need to be persistent across power cycles (clean and dirty), therefore address mappings must be hardened on persistent memory whenever space is allocated (i.e., file creation).

Contiguous Address Space for Large Files

Many existing file systems manage large files by using direct and indirect blocks to track data stored at different physical locations on storage media. The ext2 and ext4 file systems take this exact approach [21]. This approach makes reads and writes to files on SCM devices difficult to execute since now many levels of indirection must be followed in order to traverse the large array of fragmented data blocks. Also, this approach requires many different data structures to manage the locations, and the creation and deletion of these memory mapped blocks. The ext4 file system uses the same type of direct and indirect blocks for large files, however it also uses journaling (the concept of writing to a "journal" or "log" that is stored in volatile memory and written to persistent storage at intervals) to help offset some of performance overhead of having many indirect blocks for large files [6, 21]. Some existing prototype designs, such as SCMFS, each have files with contiguous logical address space inside them. Then SCMFS will build the file system on virtual address space and use page mapping to keep all blocks logically contiguous (NOTE: not physically contiguous). Therefore, there is no longer a need for complicated data structures, but rather just the starting logical address and size. Physical locations of data could be available through page mapping data structures, leveraging the existing OS infrastructure provided by the MMU. It should be noted that in this prototype solution, creating the file system address space on top of logically contiguous memory provides no real measure of value as physical mappings must still be managed and translated via the MMU.

Dynamic File Space Allocation

In traditional file systems, data blocks are allocated “on-demand” for file creation and growth (writes, appends, etc.). Space can also be freed immediately upon file deletion, although many times only file metadata is deleted and actual data is kept latent on the storage media. That latent data is usually later cleared or zeroed via a process called garbage collection [21]. However, on SCM devices, because storage must be memory mapped into the kernel address space, frequent allocation and de-allocation can produce lower performance through the MMU and create a new set of problems for memory mapped address space that is normally only mapped once at boot time. Any additional instructions added to the data path are considered extremely detrimental to overall I/O bandwidth, throughput, and most importantly latency. When the OS loads at boot time, memory that is memory mapped can consist of PCI configuration space for devices attached to the PCI bus, devices attached to the I/O controller hub of a system for the purposes of performing block or disk storage I/O and interfacing with the specific controller’s register set, or special volatile memory attached to the memory bus. The concept of memory mapping these controllers and devices at boot time is ideal because all system resources are “unused” and free memory is “physically contiguous” at this point allowing for large amounts of memory to be mapped.

One possible solution is to pre-allocate blocks, like null blocks/files, that can be used dynamically for creating or growing files. However, the downside to this is fragmentation and needing a background process to handle the de-allocation of null, or recently freed, data blocks. If all memory mapping is done by pre-allocating blocks at boot time, there is no accurate method to adjust the mapping size and regions based on workload. This essentially creates a secondary issue of having to deal with statically allocated, or boot time memory mapped memory, when there was no indication of how much memory would actually be needed as the system "up" time ran longer and longer. Another solution is to provide an abstraction to dynamically memory map SCM regions at run time, so that no additional overhead is needed to pre-allocate data blocks at run time or to have to clean up pre-allocated blocks that have recently been de-allocated, or deleted. Along with the need for runtime memory allocation/de-allocation, there is a new requirement that gives the file system the ability to perform these dynamic memory mappings. Fortunately, most operating system kernels provide a framework by which modules such as file systems can perform a mapping between logical sectors on a HDD/SSD to a kernel space address and page frame number. There would also be a need for memory system functions to grow or shrink files, and their affiliated memory mapped locations.

Write Sequencing

Writes are often cached, but not always flushed from cache in the order in which they were received from user space. There is no guarantee that writes are occurring on persistent storage in the same way they were issued from the processor, and its running code. Traditionally, file systems will issue block I/O write requests to a device attached on the southbridge of a system. These requests will go out sequentially, and may be cached by the cache manager. The method of how data is evicted from cache for write requests is not consistent across all machines.

Therefore there is no guarantee of write ordering. This problem is exacerbated by the fact SCM devices attached to the memory bus no longer need to traverse the traditional block I/O storage block (which is serialized). Additionally, out of order (OOO) pipelined processors can rearrange reads and writes for performance policies enforced at the processor level. To address the potential data integrity issues from reordering of reads and writes, some level of fencing and serialization must be implemented within the file system to ensure that data written is consistent with respect to expectations from a system level. Providing some type of fencing will guarantee atomicity for certain sets of commands to finish, and be written to persistent storage, before completing the next batch of commands that depend on the consistency of that data. Serialization can then be guaranteed within each of the fenced operations. Many file systems do address this issue via locking combined with the removal of the page/buffer cache from the direct I/O path.

File System Architecture

One definition of a file system is a mechanism or methodology for how data is stored on a physical storage medium. Without a file system, data that must be written to storage would just be stored in a raw format with no real method of how the data was stored or how it would be retrieved. A file system provides a method by which data can be grouped together into referable objects, called files, as well as contain information about the data being written (i.e., size of data written, where data is located on the disk, etc.) [21]. This method also allows the data to be retrieved at a later point in time without concern for integrity or correctness of the data [22]. Almost all file systems share similar concepts and structures. In the Linux kernel, there is a generic framework for file system usage and several structures that can, and should, be used by file system developers. In Linux, this file system framework is referred to as the virtual file system switch (VFS). The VFS provides a set of standard interfaces for upper layer applications to perform file I/O over a diverse set of file systems. The VFS provides the abstraction layer, separating the POSIX user space interface functions (i.e., open) from the details of how a particular file system implements that actual behavior [20]. The following sections address some of the key common file system concepts and components that needed to be addressed to modified for the research within this thesis.

Inodes

A file system manages two things for every file: the file data and the data that describes that file data. The latter is often referred to as metadata (the data that describes the file data) and is one of the key components of any file system [18]. Metadata describing file data usually consists of information such as the name of the file, the date the file was created, the date the file was last modified, the owner, its file permissions, etc. Almost all file system use inodes to represent both a file and a directory. Most of the metadata described is stored within the inode, and this inode is identified by an integer [18]. How and when inodes are created is file system specific. Some file systems may choose to allocate all inodes supported upon being first mounted resulting in a fixed number of files that can be created. Some file systems choose to only allocate inodes when they are needed upon file or directory creation. While other file systems that use a fixed number of inodes use a methodology called extents that extends memory of a file by using pre-allocated blocks. Regardless of the allocation method, inodes generally all store the same type of information. Any file system that is Portable Operating System Interface (POSIX) compliant requires certain attributes to be stored about the file. These attributes are described in Table 1 below. Note that the file name itself is not a required field in the inode to be POSIX compliant. This is because files can potentially have multiple names and if multiple names hard link to the same inode, then the names are all equivalent. The operating system will immediately convert a file name to an inode number then discards the file name. However, the file system will still manage and keep the file name, just not in the inode (this will be described the next section).

Field	Description
Size	File size in bytes
Device Id	The device containing the file
User Id	User id of file owner
Group Id	Group id of the file
File Mode	File type and how it can be accessed
User Flags	Flags used for file protection
Timestamps	Used for inode and file access times
Link Count	Number of hard links to this inode
Data Pointers	Pointers to disk blocks storing file contents

Table 1 – POSIX File Attributes Stored in an Inode

Inodes are managed by the file system to keep track of the metadata for a given file. However, the inode also stores pointers to the list of blocks where the data is stored on the block storage device [18]. This inode pointer structure is used in the inode to provide access to the list of data blocks associated with a given file. Most modern file systems use 15 pointers for data. The first 12 are used to point to blocks containing actual file data and are referred to as direct pointers. Figure 3 below illustrates this design. One pointer points to a singly indirect block of pointers, each of which refers to file data blocks. Another pointer points to a doubly indirect block, which is the same as a singly indirect block, but with one extra layer of block pointers. And the third is triply indirect block, which is the same as a doubly indirect pointer, except with a third block pointer indirection table.

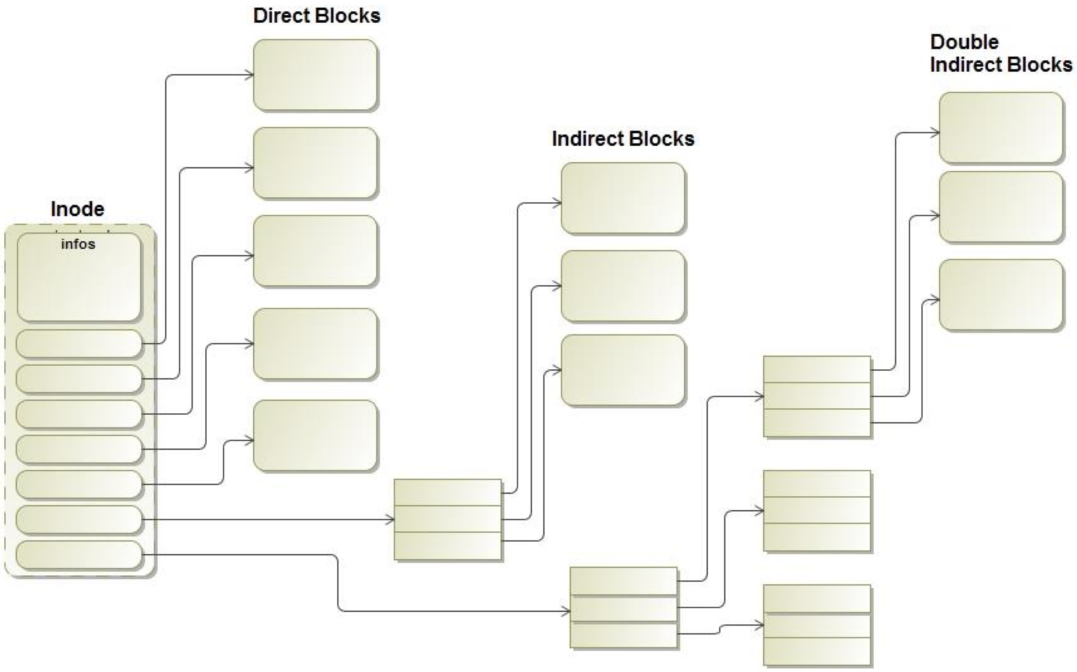


Figure 3 – Inode Pointer Structure

Directory Entry

It is important to note that inodes represent the underlying file data and because files can have multiple names (note that the path is included in the file name), an inode cannot store all the associated file names. Therefore, directory entries, or more commonly referred to as dentries, contain the name of a file in relation to that file’s location in the directory tree of a file system. In short, the inodes represent and describe the file data whereas dentries are the glue that hold the inodes and files together by relating inode numbers to file names. Dentries also enable directory caching which keeps the most recently used files in DRAM for quicker access (versus having to look up inodes stored on disk). Another use for dentries is the file system traversal between directories and the files stored. In essence, the dentry represents the relationship between an inode and the parent directory to which the inode belongs. Figure 4 below shows a very simple directory structure with four inodes (bar2, bar1, foo, and ‘/’) and three dentries (“bar1”, “bar2”, and “foo”). The first dentry will have a name “bar1”, an inode pointer to the bar1 inode, and a parent dentry pointer that points to the dentry for foo. All other dentries follow this model, one inode pointer for the underlying data and one parent pointer for the parent dentry. The dentry will always contain the file name and a function pointer to look up the inode associated with that dentry (or file name).

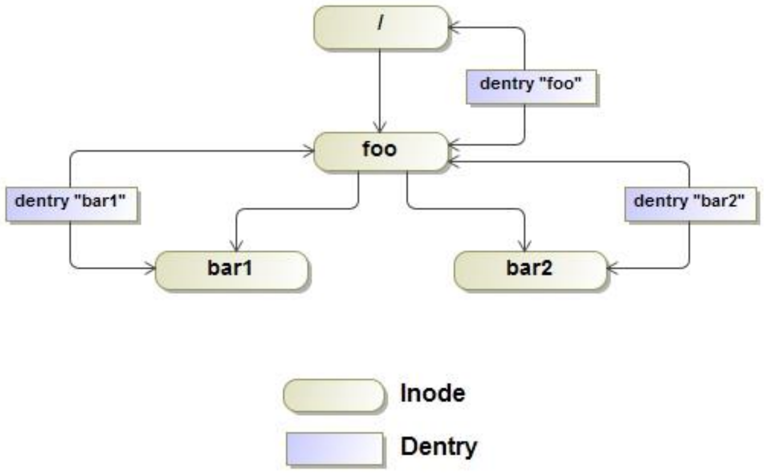


Figure 4 – Inode and Dentry Relationship

A good analogy of the relationship between inodes and dentries is that of data that is stored in DRAM versus data that is stored on persistent storage media. The dentry list that is created and maintained by the file system is nothing but a volatile list that is created at mount time. The dentry list contains only names and a subset of important metadata information. The dentry list is used for quick lookup of files and directories so that the file system may search for a file or directory without having to send performance expensive reads and write to persistent media, the inode. Simply put, it is the file name cache of the file system. The inode is a more permanent construct and is part of the metadata that is written to persistent storage (via the update mechanism for that particular file system – i.e., log structured). The inode contains all the metadata associated with the object as described above. The inode is only updated on permanent storage when necessary, but often the updates may sit in a journal or log depending on the file system architecture. This is because updating the inode on physical permanent storage is a costly operation for the file system just to maintain metadata.

File System Operations

It is important to distinguish between operations performed by a file system and operations performed on a file system. Operations on a file system such as mount, unmount, init, statfs, sync, etc. all operate on the file system itself. As mentioned above, these operations are performed by the VFS. The VFS also provides the interfaces to perform some of the more common file operations, but note that these operations that are carried out by a file system on either a file itself or its inode. Only when file system specific semantics and operations are needed does the VFS invoke the specific file system module.

CHAPTER 3

METHODOLOGY

The FAT File System

The FAT file system is a legacy file system that is still used today for many reasons; it is robust and simple. While the FAT file system offers relatively good performance, it does not compete with many modern file systems designed for performance. However, one of the primary reasons that it has lasted so long (it was first introduced in 1977) is compatibility. Nearly every commodity operating system for personal computers, mobile devices and embedded applications supports the FAT file system, and thus it is the perfect solution for format in data exchange between computers and devices of any type [10, 11]. While the FAT file system is not the default file system for commodity operating systems, it is the file system most often used for portable storage devices like universal serial bus (USB) storage devices, SSD memory cards, and flash memory cards. Common default file systems for operating systems include the extended fourth (ext4) file system for Linux, the New Technology File System (NTFS) for Windows, and the Hierarchical File System (HFS) Plus for Mac OS X. However, due to the usage of FAT, data can easily be exchanged between these operating systems without issue since FAT support is natively supported. It is also worth noting that FAT is also used in the boot stages of extensible firmware interface (EFI) compliant computers. Again, due to the non-specific nature of pre-boot environments (prior to an OS loading), there must still be a way to manage files and data on storage devices.

The FAT file system architecture within Linux is depicted in Figure 5 below. Note that like most file systems within Linux, there are components that use the native VFS framework. But there are also other functions that are FAT specific and must be handled by FAT file system code. It is work noting that the FAT file system will use the page/buffer cache for most I/O requests (for performance reasons) [8].

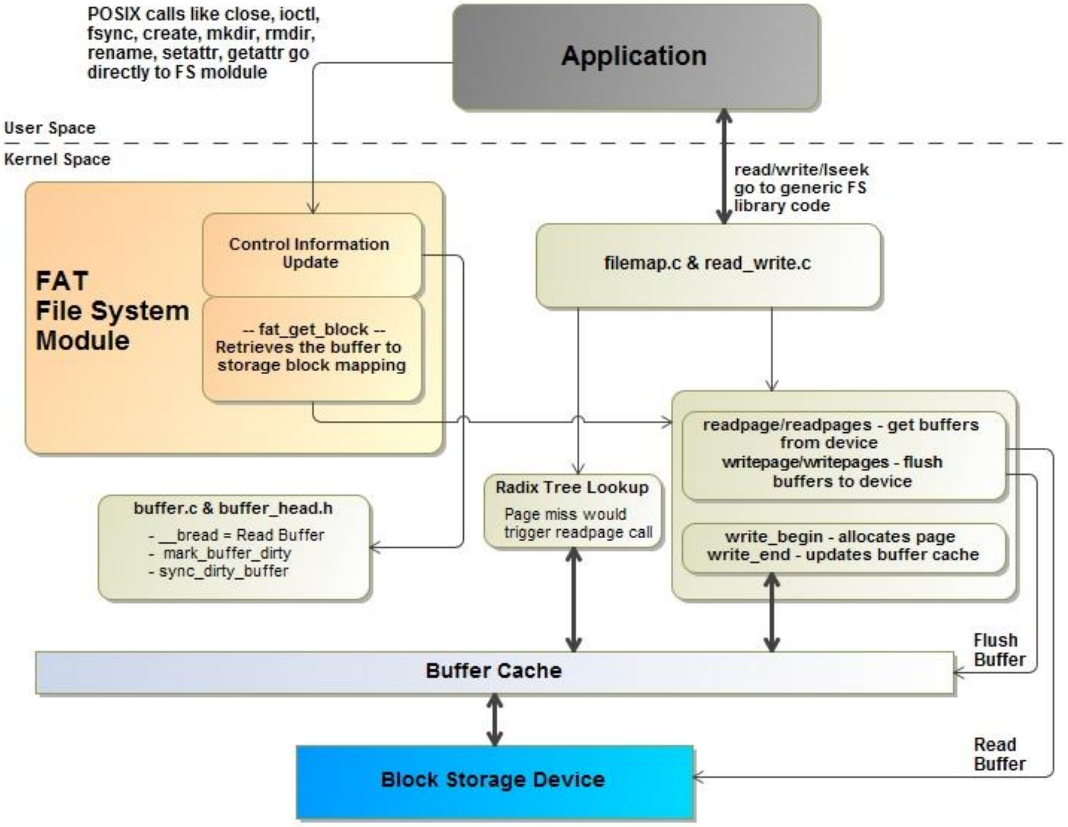

POSIX calls like close, ioctl, fsync, create, mkdir, rmdir, rename, setattr, getattr go directly to FS moldule
Application
User Space
Kernel Space
read/write/lseek go to generic FS library code
FAT File System Module
Control Information Update
-- fat_get_block -- Retrieves the buffer to storage block mapping
filemap.c & read_write.c
readpage/readpages - get buffers from device writepage/writepages - flush buffers to device
Radix Tree Lookup
Page miss would trigger readpage call
buffer.c & buffer_head.h
- __bread = Read Buffer
- mark_buffer_dirty
- sync_dirty_buffer
write_begin - allocates page write_end - updates buffer cache
Flush Buffer
Buffer Cache
Read Buffer
Block Storage Device

Figure 5 – FAT File System Architecture in Linux

The common POSIX interfaces such as open, close, ioctl, etc. are handled by the FAT file system module due to the specific nature of how it handles inode and file allocation. But once files are created, performing read and write operations to a file can be routed through the VFS provided by the Linux kernel. The name of the file system originates from the file system's prominent usage of an index table, the File Allocation Table, statically allocated at the time of formatting. The table contains entries for each cluster, a contiguous area of disk storage. Each entry contains either the number of the next cluster in the file, or else a marker indicating end of file, unused disk space, or special reserved areas of the disk. The root directory of the disk contains the number of the first cluster of each file in that directory; the operating system can then traverse the FAT table, looking up the cluster number of each successive part of the disk file as a cluster chain until the end of the file is reached. In much the same way, sub-directories are implemented as special files containing the directory entries of their respective files.

The FAT file system was chosen for the SCM modifications and analysis due to the compatibility across all operating systems environments. FAT is a well-documented file system with a long history and provides a viable platform on which to test the SCM modifications. However, as with all file systems, there are disadvantages. One such disadvantage is when using drives or partitions over a certain size (200 MiB – 2GiB), the FAT file system is not recommended. This is because as the size of the volume increases, performance with FAT will quickly degrade. This is counter-productive to the purpose of what SCM technology provides. FAT partitions are limited in size to a maximum of 4 Gigabytes (GB) under Windows NT and 2 GB in MS-DOS, but even before these sizes are reached, the overhead associated with updating the FAT partition table become too costly to be effective for performance. This is one of the primary reasons the FAT file system is not used as a default file system for commodity operating systems. Finally, the FAT file system does not support or implement any type of journaling or logging. While this help performance (at the right file system size), there is no protection for in flight data or data that is residing in the page/buffer cache on any type of error condition or dirty shutdown of a system.

Memory Page Management

To understand how the FAT file system allocates space for files, an understanding of how Linux manages memory is needed. Memory is typically partitioned and handled into 4KiB chunks, often referred to as pages (or memory pages). A page in memory can represent an equivalent set of blocks found within a file on a traditional block storage device. The distinction between a page and its component blocks is important. A 4KiB page in memory is likely represented by eight 512-byte logical sectors, or blocks, that reside on the block storage media. There are a few different Linux kernel data structures which contain information about this page as illustrated in Figure 6 below.

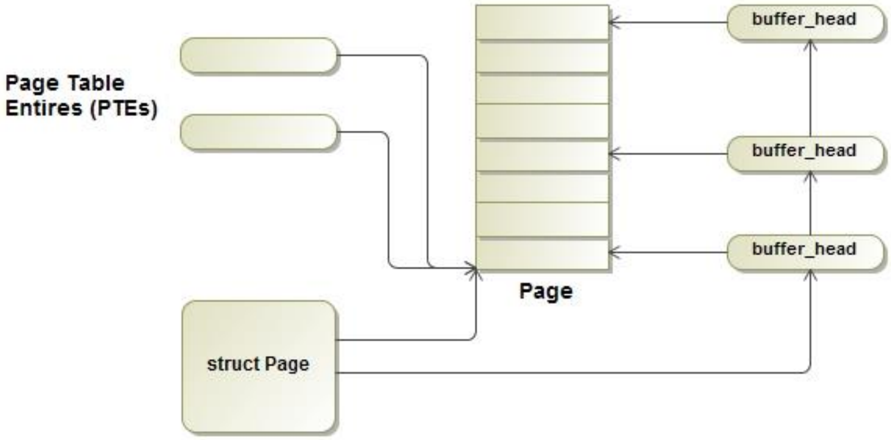


Figure 6 – Memory Page Structure

A memory page may be mapped into one or more processes' address spaces. For each such mapping, there will be a page table entry (PTE) which performs the translation between the user-space virtual memory address and the physical memory address where the page actually resides [13]. There is other information in the PTE, including a "dirty" bit. When an application modifies the page, the processor will set the dirty bit, allowing the operating system to respond by (for example) writing the page back to its backing store. Note that if there are multiple PTEs pointing to a single page, not all PTEs may have coherent data as to whether the page is dirty or not. The only way to know for sure is to scan all existing PTEs and see if any of them are marked dirty [13].

The Linux kernel maintains a separate data structure known as the system memory map. The system memory map contains one page structure for every physical page known to exist. This structure contains information about the page, including a pointer to the page's backing store (if any), a data structure allowing the associated PTEs to be found, and a set of page flags. One of those flags is a dirty bit, another flag which notes that the page is in need of writing to its backing store. Finally, there is another set of structures which may be associated with this page, the buffer head. The buffer head goes back to the earliest days of Linux. It can be thought of as a mapping between a logical sector or block on disk and its copy in kernel system memory. The buffer head is not central to Linux memory management in the way it once was, but a number of file systems (including the FAT file system) still use buffer heads to handle their disk I/O tracking. Note that there is not necessarily a buffer head structure for every block found within a page. If a file system has reason to believe that only some blocks need writing, it does not need to create a buffer head structure for the rest.

Direct Access (DAX) Extension

The DAX file system extensions for the second extended and fourth extended (ext2 and ext4, respectively) file systems were first introduced into the upstream

Linux kernel in version 4.0 [5]. The intent of the DAX extension was to reduce the overhead of unnecessary kernel buffer cache accesses, or page cache accesses. From the perspective of the OS kernel, SCM appears and behaves like any other storage device in the system. Therefore, the native file system will perform data access in a traditional manner with no regard that the storage media is of type SCM. A kernel page cache is used by an OS kernel to accelerate data accesses to files on traditional non-SCM like persistent storage (i.e., HDDs/SSDs). When a read or write occurs, a native file system will also "cache" the data access in the page cache. This allows faster access to the data at a later time without the increased overhead of having to access slower persistent storage. An OS kernel will utilize portions of DRAM memory for the page cache [9].

This traditional manner of accessing data to and from the kernel page cache for optimization purposes after the storage access completes can also happen in replacement of persistent storage access. However, because of the direct mapping of SCM to system memory, once the file system writes to the SCM memory, there is no longer a need for the file system to copy the data to and from the kernel page cache. At this point, the data is already stored in a location that is persistent, but also has the capability of being recalled with the access time equivalent to DRAM [9]. DAX removed this unnecessary and inefficient functionality for SCM device access. In order to modify a commodity file system to support SCM devices with maximum performance and efficiency, the 21-part DAX patch set was ported from ext2/4 to the FAT file system. In porting the DAX changes, a formal understanding of the modifications necessary at both a file system specific and agnostic layer were formalized.

SCM Extension for FAT (SEFT)

Much of the FAT file system implementation in Linux relies on an existing file system framework already in place provided by the kernel (i.e., VFS). The intent behind providing a file system framework for developers is for compatibility, ease of creation, commonality, and portability. Since the FAT and ext2/4 file systems have numerous differences (and similarities), not all components of DAX were necessary to port. However, there were components of FAT that needed to be modified that were not part of the original DAX extensions for ext2/4. This was due to the fact that there were several differences in the way FAT allocates memory and keeps track of these memory allocations. As discussed previously, FAT allocates memory in chunks called ‘clusters’. Clusters vary in size, but for the experiments in this thesis, FAT16 was used which means the FAT cluster size was equal to 2KiB. The following sections outline the modifications made for the porting of DAX to the FAT file system, with each section describing the change and identification of non-abstractable components.

Inode Modifications

SEFT introduced a new inode flag to indicate an I/O request is targeted for a file (or its backing store) that has been allocated on a SCM block storage device, and henceforth will be referred to as a SEFT I/O. The distinction of SEFT I/O and SEFT inodes is necessary in order to determine code paths to take when I/O requests are sent to a SCM storage block device. For example, when an I/O is detected to a SEFT inode, multiple steps are taken to ensure that buffered writes or page cache updates do not occur. The inode flag must be set during every inode allocation, and whether the flag is set or not is determined by where the inode is created. If the inode is being created in the FAT file system with SEFT enabled (via compilation flags which will be discussed later), then the inode flag is set. Mounting the FAT file system with the SEFT extension will insure that every inode (and corresponding) file created on that file system will have this new SEFT flag set. The new flag will help determine the correct SEFT functions to invoke as well as avoid unnecessary functions like calling buffered writes and performing page/buffer cache operations. Additional accessor methods and macros were added to the FAT file system to address the identification of inodes for files and directories on SEFT devices. Note that SEFT could be turned off and on via a file system mount parameter, but was enabled by default for ease of implementation.

File Mapping Modifications

Added checks for SEFT I/O in order to avoid page cache (buffered) reads and writes within the direct access functions native to the FAT file system. The direct access functions native to FAT could be invoked by user functions such as mmap or ioremap. In addition, there are also read and write iterator functions in the VFS portion of the Linux kernel that will invoke direct access function pointers. Upon return from these VFS read and write iterator functions if a read or write did not finish, or a hole was found, or there was an early EOF, then the read or write iterator functions will fall down a separate code path to perform a buffered read or write to the page cache so the rest of the I/O can be performed on actual storage at a later time. Meanwhile the page cache has the most updated copy of the read or write data (albeit the pages with the data or that would need to be paged in and the page cache would be dirty). This is a typical mechanism for handling I/O that may need to be buffered for processing in the future. However, with an SCM block storage device, buffered and delayed I/O must be avoided so as to not introduce additional I/O latency overhead or page/buffer cache thrash that is not necessary.

Block Device Direct Access

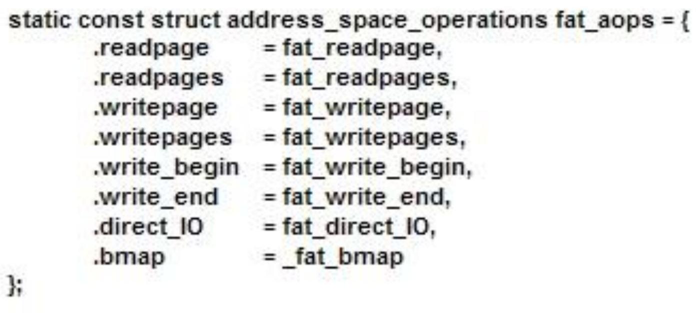
Functions were introduced to retrieve a system kernel address for directly addressable memory on block storage devices. File systems will translate mapping of a virtual address space buffer to logical block addresses (LBAs) on a block storage device. Because SCM devices must be byte addressable, the granularity of an LBA is not sufficient (typical granularity for a block storage device is 512 bytes). Therefore, SEFT must be able to take any LBA and convert it to a memory address that will later be used to perform reads and writes. This conversion is done in part by new functions introduced in SEFT and existing code in block device direct access functions. This change is one of the key pillars of the SEFT file system. Upon receiving a read or write request from the VFS, a user space virtual buffer has been passed as a source or destination buffer. This translates to a memory address for the I/O request. The file system then retrieves the first free or available LBA on the SCM storage device and creates a mapping of this user space virtual memory buffer to said LBA. Note that the physical memory address is also obtained for DMA purposes.

Once the I/O request is converted to an LBA on the SCM device, the new function called seft_get_addr will then take the LBA mapped and convert it to a page frame number (PFN) and memory address. This destination address on the SCM block storage device is obtained by allocating a two clusters of memory, which equal 4KiB (each FAT cluster is typically 2KiB). The allocation of two clusters represents one OS kernel system memory page, which is 4KiB as described earlier. These two clusters of memory are then mapped to the next free, available LBA ranges on the SCM device. From here, the 4KiB represents the two 2KiB clusters on the SCM block storage device and can be directly accessed because it has been memory mapped to the new seft_get_addr function in SEFT. The PFN and memory address returned are directly mapped to these two clusters. Therefore, reads or writes (i.e., loads and stores) to this memory address will translate in to offsets from the start of the 4KiB page boundary, which also happens to be the start of the two clusters on the SCM storage device.

This memory mapped address is the destination address of the FAT file system “memcpy” for the I/O request (note that Linux uses a structure called iov_iter to represent requests and uses these structures to actually do the memory copies). But this is the I/O in action and no additional communication with the SCM device is needed. Because the memory space is memory mapped to the kernel address space, once this memcpy is complete, the I/O is done. Therefore, no involvement with the buffer/page cache is needed. It is important to note that block storage devices traditionally format their physical media in sectors (or blocks). And these sectors are usually 512 bytes in length. Although it is worth noting that many storage devices today support sector sizes of 4KiB so that each sector can align with host memory page size thus allowing direct memory access (DMA) engines to copy data without having to translate between two different source and destination sizes.

SEFT I/O Functions

At the heart of the changes for SEFT are two new I/O functions (seft_do_io and seft_io). The primary purpose of these functions is to perform the actual write to the physical storage media on the SCM device, but through a kernel-level memcpy utility function that is specific to the underlying platform architecture (i.e., x86). While the purpose of this function is simple, it must perform multiple steps to setup up the read or write correctly. These new seft_do_io and seft_io functions are called directly from the function fat_direct_IO (note that seft_io is invoked directly from seft_do_io). The fat_direct_IO function existed previously to handle I/O in a FAT file system that was targeted for a file that was memory mapped into user space. This function is invoked because it is mapped to the .direct_IO function pointer in the FAT file system address space operations (aops) function pointer table. The function pointer table is shown in Figure 7 below.



```
static const struct address_space_operations fat_aops = {
        .readpage       = fat_readpage,
        .readpages      = fat_readpages,
        .writepage      = fat_writepage,
        .writepages     = fat_writepages,
        .write_begin    = fat_write_begin,
        .write_end      = fat_write_end,
        .direct_IO      = fat_direct_IO,
        .bmap           = _fat_bmap
};
```

Figure 7 – Fat Address Space Operations Function Pointer Table

Traditionally, the function fat_direct_IO would invoke the direct I/O function of the underlying storage device. For example, FAT currently invokes the function blockdev_direct_IO because I/O is typically routed to block devices (i.e., HDDs/SSDs). The function blockdev_direct_IO is a generic library function for all file systems to use for doing mmap I/O that first attempts to access the page cache. However, to implement the direct memcpy functionality that avoids this page cache access, a new I/O function was needed. This new functionality can be kept abstracted and portable with the assumption that all I/O structures are generic to other native file systems.

At the center of the new seft_io function is a loop that will iterate on the number of bytes that need to be written or read. The condition on which the loop iterates is whether the current file position pointer is at the end of the amount of data that needs to be written or read. For example, if there is 1MiB of data that needs to be written, and the current file pointer position is at 0 (the beginning of the file), then the loop will iterate until all the data is written and the file pointer is at 1MiB. Each iteration of the loop starts with a check to see if the current file pointer position is equal to the end of the allocated space. Simply put, the loop checks to see if space must to be allocated for the read or write (i.e., an empty file or appending at the end of a file). If the current file pointer position is at the end of the file, then a new size is calculated that is specific to the underlying method that the FAT file system uses to allocate blocks.

For example, if a file is empty and 1MiB of data needs to written, then the seft_io function will calculate how much space needs to be allocated and mapped on the SCM device by calling the native fat_get_block function of the FAT file system. The fat_get_block function performs multiple tasks, but the essence of its role is to retrieve the sectors (or LBAs) on the physical media storage that are mapped to the memory buffer used for the read or write request. In the Linux kernel, there are multiple structures that are used to represent the mapping of sectors/LBAs to memory addresses. One structure is the buffer_head structure (as described above). Historically, a buffer_head structure was used to map a single sector, or block, of typical size 512 bytes, within a memory page (i.e., 4KiB), and as the unit of I/O through the file system and block layers [12]. Today, the basic I/O unit has changed (to a block I/O structure), and buffer_heads are used for extracting sector/block mappings (via calls like fat_get_block) [12].

Because memory page size in Linux is 4KiB and typical sectors sizes on block storage devices are 512 bytes, data read or written to a particular sector/LBA on physical block storage media must also be tracked as an offset into a memory page. This is one of the primary purposes of the buffer_head structure. Every page of system kernel memory is 4K, therefore, each page has eight sectors/LBAs mapped to it (4KiB = 8 * 512). The key buffer_head structure members are defined in Table 2 below.

Field	Description
b_page	Memory page this buffer_head is mapped to
b_blocknr	Starting block (sector) number this buffer_head represents
b_size	Size of the mapping
b_data	Pointer to data within the page
b_bdev	Pointer to physical block storage media device

Table 2 – Buffer Head Structure Key Member Fields

Upon the first iteration of the seft_io loop, when an empty or new file is encountered, and space must be allocated, b_size is set to zero since there is no size associated with the file and no memory mappings have been created. The buffer_head structure is passed as a pointer reference to fat_get_block to be filled out by the FAT file system specific method. As shown in Figure 8 below, fat_get_block will call fat_bmap [7]. The function fat_bmap will attempt to map any sectors/LBAs, that contain valid data (via analyzing file pointer position and offset), and map those sectors/LBAs to FAT clusters and kernel memory. The mapping would be represented by the buffer_head structure. However, in the example used in this thesis, when this file is first accessed, it is empty. Therefore, no mapping exists for this file. In short, since this is the first time this file has been accessed since the system powered up, it has not been memory mapped. This is one of the critical axioms of modifying native commodity file systems to work with SCM devices. Files are constantly memory mapped, either for the first time, or because the file size is changing. If there is a downside to SCM devices and how modern file systems must interact with them, this is it. However, it is arguable as to whether the overhead associated with the constant memory mapping outweighs the overhead associated with passing I/O requests through two layers of translation down to a traditional block storage device attached to a southbridge.

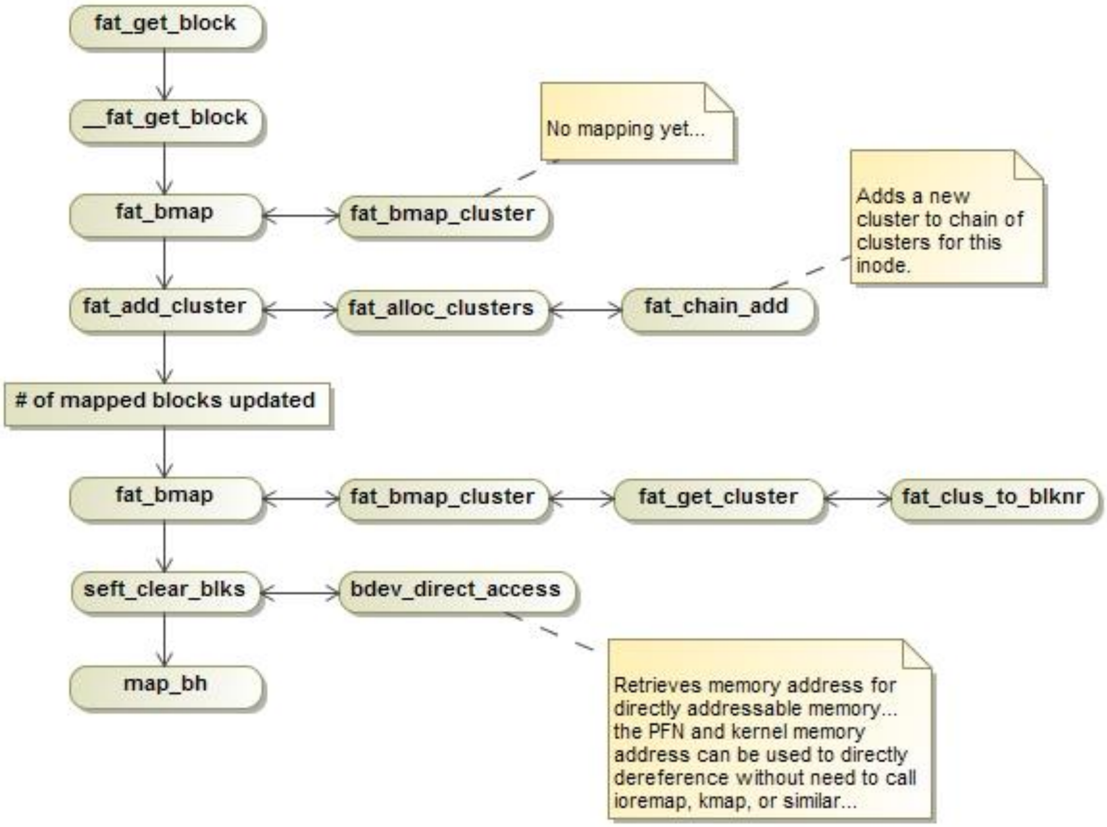

fat_get_block
__fat_get_block
No mapping yet...
fat_bmap
fat_bmap_cluster
Adds a new cluster to chain of clusters for this inode.
fat_add_cluster
fat_alloc_clusters
fat_chain_add
# of mapped blocks updated
fat_bmap
fat_bmap_cluster
fat_get_cluster
fat_clus_to_blknr
seft_clear_blks
bdev_direct_access
map_bh
Retrieves memory address for directly addressable memory... the PFN and kernel memory address can be used to directly dereference without need to call ioremap, kmap, or similar...

Figure 8 – Diagram of fat_get_block Call Tree

The first call to the fat_bmap function is expected to fail if the file is being created for the first time (or is being accessed for the first time since power on). Therefore, when the fat_bmap function returns a status indicating no block mapping occurred, the fat_get_block function will determine that the FAT file system must allocate and map new memory. The first step in doing this is by calling the function fat_add_cluster as shown in Figure 8 above. The function fat_add_cluster is native to the FAT file system and will perform two major steps: allocate clusters and then add them to a cluster chain associated with the current inode. As described earlier, the FAT file system represents physical memory space allocation with clusters. Each cluster represents a certain size, or chunk, of contiguous data in a file. Clusters can be chained together via pointers until a cluster points to an end of file (EOF) delimiter (instead of another cluster entry). From within fat_add_cluster, the function fat_alloc_cluster will be invoked, which performs the actual allocation of a cluster by finding the next available cluster in a pool of free clusters associated with the current kernel process. Once the next free cluster is discovered, a pointer to the next free cluster entry is returned. The pointer returned by the function fat_add_cluster is then used in a call to the function fat_chain_add. The function fat_chain_add takes the cluster that was just discovered and adds it to the cluster chain associated with the inode for this read/write request.

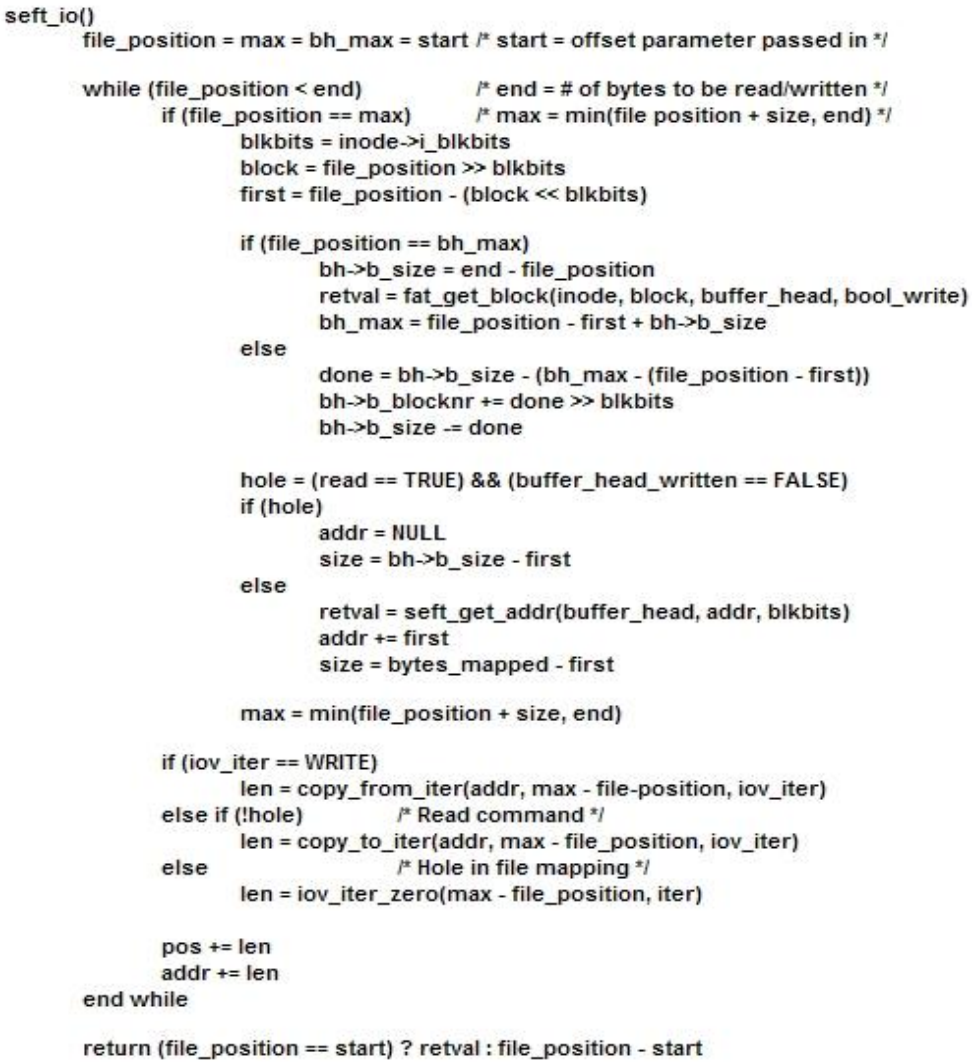
After the above sequence has occurred, the space for the write request has been allocated by the FAT file system, and therefore space can now be successfully memory mapped. As shown in Figure 8, the function fat_bmap is called for a second time. However, this time fat_bmap is called after updating parameters and variables indicating how much space was just allocated. From fat_bmap, the function fat_bmap_cluster is invoked with the cluster that was just allocated to get the first data cluster associated within the file system. Finally, the function fat_clus_to_blknr is called to translate the cluster allocated to a physical sector/LBA on the block storage device. The final outcome of calling the function fat_bmap is to get the physical sector/LBA address and the number of sectors/blocks that were mapped. These two pieces of information are then passed to the seft_clear_blocks, which then calls bdev_direct_access. These two functions serve the purpose of mapping the physical sector/LBA to a kernel memory address (as well as a PFN) and then performing a memsets to zero (to clear out any latent data). At this point, the buffer head is updated via the function call map_bh. The function map_bh will associate the physical sector/LBA address to the b_blocknr field, as well as set the b_size field to the size of the data that needs to be read or written. This mapping of the buffer head structure is the product of the new seft_io function calling fat_get_block, and will enable the loop to continue processing the direct I/O request.

After the seft_io loop gets the blocks associated with this read/write request, a condition is checked to see if there is a hole in the allocation of clusters for this file. In the context of the FAT file system, a hole is defined as a non-mapped region of a file. This condition could be due to a failure to map a part of the file or the buffer head was not updated successfully. In most cases, a hole in the file is not common. After the check for the hole in the file, the loop then calls the function seft_get_addr. As mentioned above, the purpose of this function is to take in the buffer head information (i.e., physical sector/LBA address and size) and convert it to a memory mapped address and PFN, which can be directly accessed by the file system (i.e., memory load/store). Once this address is retrieved, the loop will then add the current position of the first write to the memory address just received. In the example used in this thesis, the current position at this point is zero since the file has not been written yet. Also, the size is updated to reflect the total size of the read or write request minus the value of the first write position, which is zero in the example. In subsequent loop iterations, the update of the address and size will happen through common address calculations based on how much data was successfully written on the current loop iteration.

Once the memory address, size, and current position are updated, the seft_io function will perform the memcpy function from the user space buffer to the memory mapped location on the SCM device. The method by which the memory copy is performed is specific to the underlying architecture (32 or 64 bit, x86, SPARC, etc.). Therefore, a common memory library function that uses a special I/O structure is used to perform the memcpy. The common I/O structure used is the iov_iter structure and is provided natively by the Linux kernel [15]. At a high level, the iov_iter structure is used to describe a buffer (in this case, a user space buffer) that may be scattered in both physical and virtual memory [14]. The use of the iov_iter structure helps the portability and sustainability of SEFT as well as reuses a common structure in many other file systems.

The iov_iter structure is similar to how a scatter gather list (SGL) works in that it contains an array, or vector, of addresses that point to the scattered pieces of physical memory that make up the user space buffer. The iov_iter structure has numerous helper functions defined in the Linux kernel to assist in the copying of data to and from these buffers. For writes from the iov_iter user buffer to the memory address of the SCM block storage device, the copy_from_iter function is used. For reads in the other direction, the copy_to_iter function is used. There is a third case where a read request is attempting to read a hole in a file mapping, in which case the iov_iter_zero function is called. This function will copy zeros to the iov_iter mapped buffer to reflect the hole in the file mapping. Finally, as the loop nears its first iteration, the memory address being written to is updated along with the size to reflect the amount of data that was just read or written with the iov_iter functions.

Figure 9 below contains the pseudo code for the seft_io function.



```
seft_io()
        file_position = max = bh_max = start /* start = offset parameter passed in */

        while (file_position < end)             /* end = # of bytes to be read/written */
                if (file_position == max)       /* max = min(file position + size, end) */
                        blkbits = inode->i_blkbits
                        block = file_position >> blkbits
                        first = file_position - (block << blkbits)

                        if (file_position == bh_max)
                                bh->b_size = end - file_position
                                retval = fat_get_block(inode, block, buffer_head, bool_write)
                                bh_max = file_position - first + bh->b_size
                        else
                                done = bh->b_size - (bh_max - (file_position - first))
                                bh->b_blocknr += done >> blkbits
                                bh->b_size -= done

                        hole = (read == TRUE) && (buffer_head_written == FALSE)
                        if (hole)
                                addr = NULL
                                size = bh->b_size - first
                        else
                                retval = seft_get_addr(buffer_head, addr, blkbits)
                                addr += first
                                size = bytes_mapped - first

                        max = min(file_position + size, end)

                if (iov_iter == WRITE)
                        len = copy_from_iter(addr, max - file-position, iov_iter)
                else if (!hole)         /* Read command */
                        len = copy_to_iter(addr, max - file_position, iov_iter)
                else                    /* Hole in file mapping */
                        len = iov_iter_zero(max - file_position, iter)

                pos += len
                addr += len
        end while

        return (file_position == start) ? retval : file_position - start
```

Figure 9 – Pseudo Code for seft_io Function

Cluster Allocation

As described earlier, the FAT file system uses memory clusters to represent files. The primary function responsible for cluster allocation is fat_add_cluster. As described earlier, the function fat_add_cluster will perform two tasks: search and find an available cluster to be used and then add that free cluster to a cluster chain for the inode representing the file. The function fat_add_cluster is called from the function fat_get_block which is used to retrieve any existing sector/LBA allocations for a given file. However, because a file may or may not have been allocated clusters previously, the function fat_get_block may fail to find any previously allocated clusters. For files that are initially created, the first call to the function fat_get_block will always fail to find any existing clusters. Therefore, the fat_get_block function will call fat_add_cluster for empty files. One of the inherited shortcomings of the FAT file system is this process by which files are allocated clusters.

In order to add clusters to an inode for a file, the function fat_alloc_clusters must be called. However, when it is called from within the function fat_get_block, it is called with the purpose of only allocating one cluster. A single cluster in a traditional FAT16 file system of relatively small size (<100Mib) is equal to 2KiB. Unfortunately, this cluster size does not align well with system memory page size in the Linux kernel, which is 4KiB. Since all memory mappings are performed on a system memory page boundary, allocating only one cluster at a time is insufficient and creates data integrity issues. This is one of the other critical axioms of modifying commodity and native file system for SCM block storage device support. In the initial porting modifications from DAX, there were no modifications made to the native ext2 or ext4 file systems with respect to file block allocations. This is because both the ext2 and ext4 file systems allocate memory for files in 4KiB chunks. Therefore, when initial experiments were run on SEFT, numerous file corruption issues were observed since the cluster allocation was not modified to match the needed 4KiB allocation boundaries.

After extensive triage and debugging, the FAT file system contained a shortcoming in that it only allocates a single cluster for every call of fat_add_cluster. Interestingly, there exists a comment in the code that reads "TODO: multiple cluster allocations would be desirable", which confirms other usages for the FAT file system also possibly requiring allocation of more than one file cluster at a time. The modification to the fat_add_cluster algorithm introduced a check for a SEFT inode, which implied that two clusters (equaling 4KiB) needed to be allocated. Upon detection of a SEFT inode, a loop iterates twice over the calls to the functions fat_alloc_clusters and fat_chain_add. This modification translates back up to the seft_io function and its call to fat_get_block, which returns 4KiB of allocated memory, instead of 2KiB.

This modification to the fat cluster allocation mechanism is a prime example of how the framework of the native file system on which SCM support is being added cannot be ignored or overlooked. Every file system has its own method of allocating and tracking memory chunks used for files. And while many native file systems allocate memory chunks on 4KiB boundaries, others do not. Examples of the latter include specialized or legacy file systems that are architected and designed to work across multiple system platform architectures, operating systems, or embedded devices. No assumptions can be made on the reliability of the file memory allocation method without heavy investigation in to how the needs of the SCM modifications must be met. In short, the file memory allocation methodology cannot be easily ported from one file system to another, but instead is file system specific.

Block Device Direct Access

The function responsible for creating the mapping between the physical sectors/LBAs on the SCM storage block device and a directly accessible kernel memory address is bdev_direct_access. The function bdev_direct_access is called from within two locations in the SEFT modifications: seft_clear_blocks and seft_get_addr. Both of these functions are utility functions within the SEFT extension code to obtain the memory addresses directly mapped from physical sectors/LBAs for the purposes of clearing the storage space or for performing a memcpy. By definition, the function bdev_direct_access is tasked with obtaining the address for directly-accessible memory. Essentially, the function will return the PFN and address of the memory that needs to be read or written on the SCM storage block device without the need to call ioremap, kmap, or a similar function. The first issue encountered with the function bdev_direct_access was a check for support of direct access operations for block devices.

In order to test the SEFT extension code, or any other direct access type of storage, a simulated SCM device must be used because no commercial or non-commercial SCM devices exist today. For this thesis, a block RAM (random access memory) device was chosen to represent the SCM storage block device. The Linux kernel provides up to 16 RAM disk options via the files that can be located in the directory /dev. The block RAM disk works as a typical file (everything in Linux is represented as a file). The RAM disk file can be assigned memory so that any component of the OS can read or write to the RAM disk file as if it were writing to a traditional block storage media device. RAM disks are common simulation devices on which modifications can be tested. Once the RAM disk (file) has been assigned backing storage memory, a partition can be created on top of the file as well as a file system created and mounted. This will allow any user or kernel layer component to access the file as if it were a real HDD, SSD, or SCM block storage device. However, because the SEFT (and also the DAX) extension code is treating this RAM disk as memory addressable, the configuration of RAM disks had to be modified in the compile time environment. These modifications required adding a new menuconfig item (i.e., compiler #define) for the Linux kernel compile process. The new config parameter is CONFIG_BLK_DEV_RAM_SEFT and its purpose is to support the SEFT extension in order to allow direct access to RAM block devices. This addition avoids double buffering data in the page cache before copying to the RAM block device. This configuration also prevents RAM block devices from being allocated from highmem.

Within the Linux kernel itself, a compile time switch was added to the file /drivers/block/brd.c (this file represents the block RAM disk functionality) to wrap a new function brd_direct_access. The function brd_direct_access was part of the DAX extension and was ported to SEFT without the need of modification. However, the ease of portability was due to the fact that DAX also used a RAM block disk to test functionality. With another type of storage medium (such as true SCM block storage devices), modifications to this function would be necessary to fit the underlying mapping of sectors/LBAs to physical memory. The primary purpose of the brd_direct_access function is to take in a sector for a particular RAM block device and convert it to a kernel memory address and PFN. This function, brd_direct_access, is called from the bdev_direct_access function describe above, via a function pointer in the function pointer table named block_device_operations. This allows the lower layer of the RAM block device method for mapping the address and PFN to be abstracted away from the upper layer of the FAT file system when calling bdev_direct_access.

The function bdev_direct_access does not, and should not, know that the backing storage media is a true SCM device or not. It just needs to know that it's a block device represented by a block device (bdev) structure pointer with direct access capability. Within the bdev structure is the function pointer table to handle all requests that are specific to that type of block storage (in the case of this thesis, a RAM block disk). The block_device_operations function pointer table contains a function pointer entry called .direct_access. It is this function pointer entry that is populated with the brd_direct_access function. Therefore, any calls to the .direct_access function pointer for a RAM block device come to brd_direct_access. This is another one of the key axioms of SEFT, and the portability of such a modification that could be done for other file systems. This lower layer direct access function for RAM block devices must be changed for true SCM devices or any other type of SCM simulation devices.

The second issue discovered with bdev_direct_access was a secondary check in the function to ensure that any sector passed in was aligned on a 4KiB memory page boundary. The reason for this discrepancy is because FAT does not natively allocate memory on 4KiB memory page boundaries. The DAX extension code relied on the underlying ext2 and ext4 framework for allocating memory for files. The memory allocated for ext2 and ext4 were always aligned to 4KiB memory page boundaries. However, because the FAT file system is a legacy file system around 30 years old, its file allocation method (clusters) does not always allocate on 4KiB memory page boundaries. As described above, the FAT file system will allocate clusters of relative size depending on the implementation of FAT (FAT12, FAT16, FAT32) and the size of the overall disk/device on which FAT was mounted. In the example for this thesis, FAT natively allocates 2KiB clusters and it was described above how this had to be rectified in order to work with the current Linux kernel and

SEFT. Because FAT allocates on non-4KiB memory page boundaries, the secondary check in bdev_direct_access consistently failed. Although the modification was made to always allocate 4KiB of memory per FAT cluster allocation, this secondary check for 4KiB alignment had to be removed because the offset at which user data could be stored could potentially start at a cluster that maps to a sector that is not on a 4KiB memory page boundary. Upon removal of the 4KiB memory page boundary alignment check, all SEFT I/O succeeded in getting passed to the .direct_access function pointer in the block device operations table for the associated RAM disk device.

Filemap for Direct I/O

The filemap functionality lives in the memory management (MM) component of the Linux kernel. The relevant modifications here are related to the overall architecture of how all file systems integrate with the Linux kernel. As described earlier, there are common components of all file systems that utilize common file system code provided by the Linux kernel. One example of such common code is the generic read and write file functions. This group of functions live in the /mm/filemap.c file and serve the purpose of providing a generic interface to read or write file data. The function itself does all the work needed for actually writing data to a file. More importantly, it will call all the proper sub-functions based on whether a direct I/O or a standard buffered read/write is needed. However, all I/O to a SCM storage block device will be direct I/O. Therefore, the direct I/O code path within the generic_file_direct_write and generic_file_direct_read functions must always go through the .direct_IO function pointer defined in the address_space_operations structure used in the file structure defined by the Linux kernel. The file structure defined by Linux contains an address_space structure. This address_space structure describes the memory and page mappings to the current file. Within this address_space structure, used by the file structure, is the address_space_operations (aops) function pointer table. It is within this aops table that the .direct_IO function pointer will be invoked when a direct I/O request is detected. The .direct_IO function pointer is set to fat_direct_IO in the /fs/fat/inode.c file. Therefore, any file created in a FAT file system, including SEFT I/O, will get routed to the function fat_direct_IO. As explained earlier, the fat_direct_IO function will invoke the new seft_do_io and seft_io functions.

Compiler Modifications

All of the changes described above for the SEFT extension required modifications to the Linux kernel build environment. First, because new files were introduced, the Makefile under the /fs directory had to be modified to compile the new SEFT module (two new files were seft.c and seft.h). Second, the Kconfig file was modified to include a new compiler define (i.e., #define) for SEFT called CONFIG_FS_SEFT. The Kconfig files provide a way to configure a Linux kernel build. The Linux kernel build process has two parts: configuring the kernel options and building the kernel source with those options [16]. To configure the kernel, a user launches a text or graphical based kernel configuration tool. The kernel configuration tool prompts users to select kernel options to be included, or excluded, from the Linux kernel compilation process. Because the Kconfig file in the /fs directly was modified to add SEFT as an option, when a user launches the kernel configuration tool, there is an option presented under the File Systems sub-menu to turn on, or off, the SEFT extension. Finally, as mentioned previously, a new kernel define was added for the block RAM device direct access configuration, which was needed to support SEFT on top of RAM block devices. An updated block architecture with all SEFT extension changes is shown in Figure 10 below.

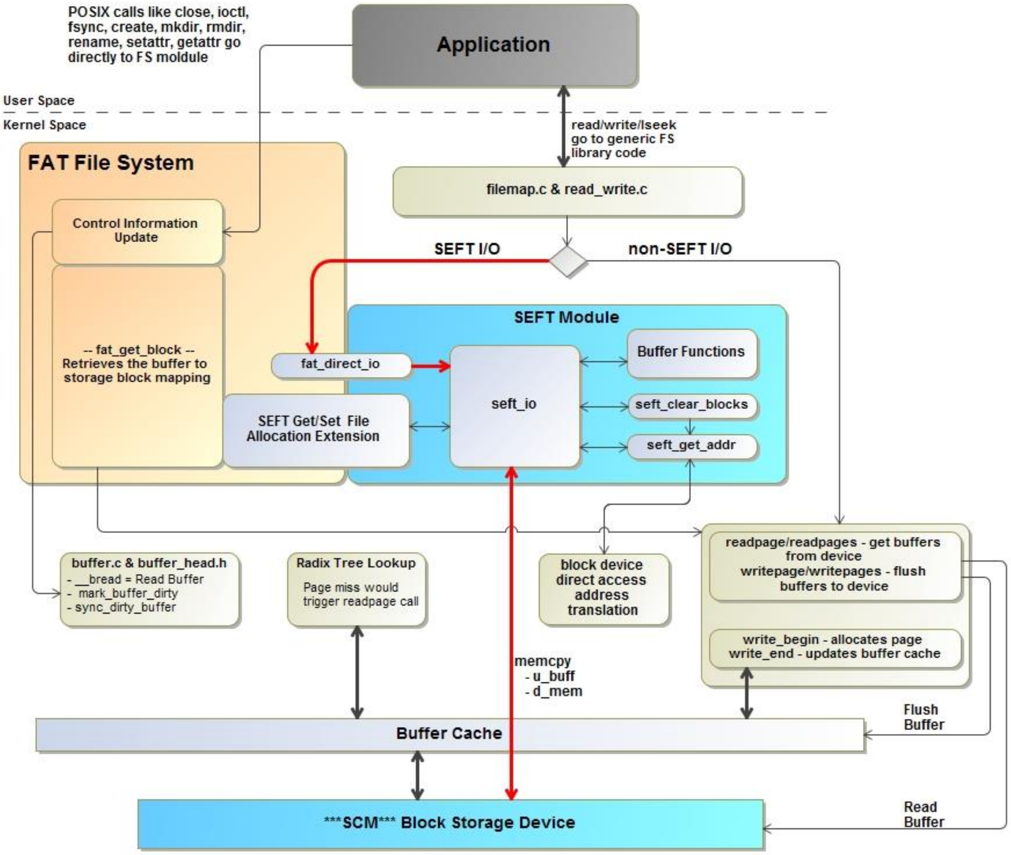

POSIX calls like close, ioctl, fsync, create, mkdir, rmdir, rename, setattr, getattr go directly to FS moldule
Application
User Space
Kernel Space
read/write/lseek go to generic FS library code
FAT File System
filemap.c & read_write.c
Control Information Update
SEFT I/O
non-SEFT I/O
SEFT Module
-- fat_get_block -- Retrieves the buffer to storage block mapping
fat_direct_io
Buffer Functions
seft_io
seft_clear_blocks
SEFT Get/Set File Allocation Extension
seft_get_addr
buffer.c & buffer_head.h
- __bread = Read Buffer
- mark_buffer_dirty
- sync_dirty_buffer
Radix Tree Lookup
Page miss would trigger readpage call
block device direct access address translation
readpage/readpages - get buffers from device
writepage/writepages - flush buffers to device
write_begin - allocates page
write_end - updates buffer cache
memcpy
- u_buff
- d_mem
Flush Buffer
Buffer Cache
SCM Block Storage Device
Read Buffer

Figure 10 – SEFT Block Architecture in the FAT File System

CHAPTER 4

ANALYSIS

Directory Operations

The initial testing and analysis of the SEFT extension was broken down into four components for typical VFS operations: mount, initialize, file creation, and file deletion. However, it should be noted that other functions were tested as well, which will be listed later in this section. In order to analyze the testing, a formalized method for testing was first designed. As described in several publications [19, 20], a typical flow for testing VFS functionality on a file system is as follows:

1. Create the temporary backing store (RAM disk)
2. Perform a mkfs on the backing store
3. Mount the RAM disk (and file system) to a directory

Once the RAM disk is mounted to a directory, any operations performed on that directory or sub-directories will invoke the FAT file system with SEFT extension. As stated earlier, for this thesis, a RAM disk of size 100 MiB was used for test. Upon successful creation of the RAM disk backing store, the file system was created and mounted with the following commands below. Note that /dev/ram7 is the backing store, or RAM disk, used for test and /mnt/fat is a temporary directory setup for test purposes.

- mkfs –t fat /dev/ram7
- mount /dev/ram7 /mnt/fat

The use of print statements and the kernel log (accessed via the dmesg command) allowed the execution of the file system to be verified at differing points in time. Initially, the file system mount action was the first to be tested. Since the SEFT modifications centered on I/O, there were no changes to how the FAT file system was mounted, other than to enable the SEFT code by default. In addition to mounting the file system, the initialization sequence was also verified. After the successful mount, the following sequence of tests were successfully performed.

1. Create a single, top level directory
2. Create multiple, top level directories
3. Create a secondary sub-directory under first level directory
4. Create multiple sub-directories under first level directory
5. Create additional sub-directories at multiple levels
6. Delete directory, with no sub-directories
7. Delete directory, with sub-directories and parent directories
8. Delete entire directory tree under root folder in file system
9. Rename directories, with and without sub-directories

With the above tests, all inode mappings were confirmed using the tree command with the --inode parameter to display inodes associated with each directory. In analyzing the results of the directly testing, it was clear that the SEFT modifications had little effect on directory operations. While inodes and dentries are created and managed for all directories, because there is no backing store associated with a directory, there was no authentic need to invoke the new SEFT extension code introduced.

File Operations

After successful directory testing, the initial file creation and deletion operations tests were performed. There were two options available for testing methodology: POSIX open/close and native file editors. In an attempt to more closely match how an end user would create files, the native file editor method was used for test purposes. Most Linux operating system vendors (OSVs) provide the vi or vim tool by default as part of the operating system image. The file editor tool vi/vim is a common command line editor tool used for a variety of reasons, hence its selection for testing. It is worth nothing how vi/vim creates files before proceeding with the analysis of file creations and deletions. First, vi/vim will create what is called a swap file for every file that is created. This is a backup file that can be used to recover file data should anything bad occur while a system is running. The swap files are hidden from the user using vi/vim, and are often times stored in special directories in order to ensure that the backup file is not lost with the original file in case the entire directory suddenly becomes deleted or corrupted. The practice of creating a backup file that cannot be seen by an end user is common for other file editors such as Microsoft Word, GNOME gEdit, and Emacs (just to name a few).

In testing the SEFT extension modifications, a file was opened with the command line: vi testfile.txt. This one command triggered multiple actions with the FAT file system. It should be noted that directory location for file testing was determined to be irrelevant, as all issues encountered during file testing occurred regardless of where the file was in a directory tree. The multiple actions that occurred all revolved around the allocation of the inode and clusters for the file being created with vi/vim. As described earlier, the allocation of blocks for files by the FAT file system is different than ext2/ext4. The ext2/4 file system always allocates blocks on 4KiB boundaries. Linux natively supports 4KiB memory allocation, which is inherited by the ext2/4 file system (ext4 is the default file system for the Linux kernel). However, because FAT was developed so long ago, its file allocation is unique and also varies depending on the overall file system size. Therefore, the initial port of DAX to FAT did not work because of the file allocation issues. The SEFT I/O functions were designed to loop through the size of an I/O request. However, every call to the SEFT I/O function had a theoretical maximum (called the bh_max) set when getting the blocks for a particular file/inode.

The call to retrieve the blocks would discover only 2KiB of memory could be allocated at a time and that became the upper limit. Therefore, any I/O coming down (for a 4KiB request, because all access are done on 4KiB page boundaries) would return only after reading or writing 2KiB of data. This caused the secondary I/O to come down to finish reading or writing the other 2KiB of data, but because the memory page associated with the I/O request contained the first 2KiB (or 4 sector/LBAs), the same chunk of data was retrieved. This led to the issue of reading and writing to the same sectors/LBAs, and eventual data corruption. Once the allocation issue was fixed (to allocate 4KiB of memory for every I/O), the data integrity issues were resolved. However, this confirms that not every file system can support the DAX or SEFT extensions. The determination is how the underlying file memory is allocated. It is also worth noting that because 2KiB clusters were allocated for files by the FAT file system, file chunks didn’t always start on 4KiB boundaries. The file chunks could, and often would, start on 2KiB boundaries, thus creating a problem for the mapping of LBAs to kernel address space (via bdev_direct_access). Multiple checks for 4KiB memory page alignment had to be removed in order to successfully perform reads and writes.

Performance

Analysis of performance with the SEFT modifications is limited due to the fact that no SCM devices exist in the market. All testing, for this research and others, is reliant on emulation. The most common form of emulation is RAM disk. However, using RAM disk provides a common denominator for testing. Because the primary purpose of SCM extensions is to reduce overall I/O latency, testing performed had the following criteria.

- Queue depth of one
- Single threaded
- Synchronous I/O completions

The queue depth of one insured that latency measurements were showing statistics for one I/O request at a time. If a queue depth larger than one was used, then results would become mixed because the CPU time (cycles) spent processing an I/O to a SEFT device would also be accounting for other outstanding I/O requests. Using a queue depth of one insures that CPU cycles, and latency measurements, would be for a single I/O. The same can be said for using single threaded tests. If more than one worker thread were to be used, then the CPU cycles, and therefore latency measurements, would not accurately represent a single I/O request. Multiple worker threads would cause multiple I/O requests to the file system. This is also true for asynchronous I/O (AIO) completions. Hence the need for synchronous I/O completions. AIO completions would cause an overlap in I/O requests being processed by the file system, which is contradicts the measurement of a single I/O.

As for other parameters, such as transfer size, the value is less relevant as long as the value remains the same for both performance measurements with and without SEFT. For the purposes of this research, 4KiB random reads and random writes were used as this is an often used benchmark for storage performance measurements. It should be noted that the performance tool used was FIO. FIO stands for flexible I/O tester and was created for the purpose testing “flexible” workloads to a storage device [25]. The FIO test script used contains the following commands below.

```
[global]
ioengine=sync
rw=randwrite
size=2M
directory=/mnt/fat/fio
thread=1
iodepth=1
direct=1
bs=4k

[ray-randwrite-2M]
```

This script will test random writes, changing the “rw” field to randread will test random reads. The random reads and random writes test will test the 4KiB size I/O requests to a file in the FAT file extension with and without the SEFT extension. The results are shown in Table 3 and Figures 10 and 11 below.

Workload	Bandwidth (KiB/s)	Throughput (IOPS)	Latency (us)
4K RR (SEFT)	512000	128000	5.36
4K RR (non-SEFT)	1000	256000	1.93
4K RW (SEFT)	512000	128000	6.68
4K RW (non-SEFT)	1000	256000	1.98

Table 3 – Performance Measurements (QD = 1, # Threads = 1, AIO)

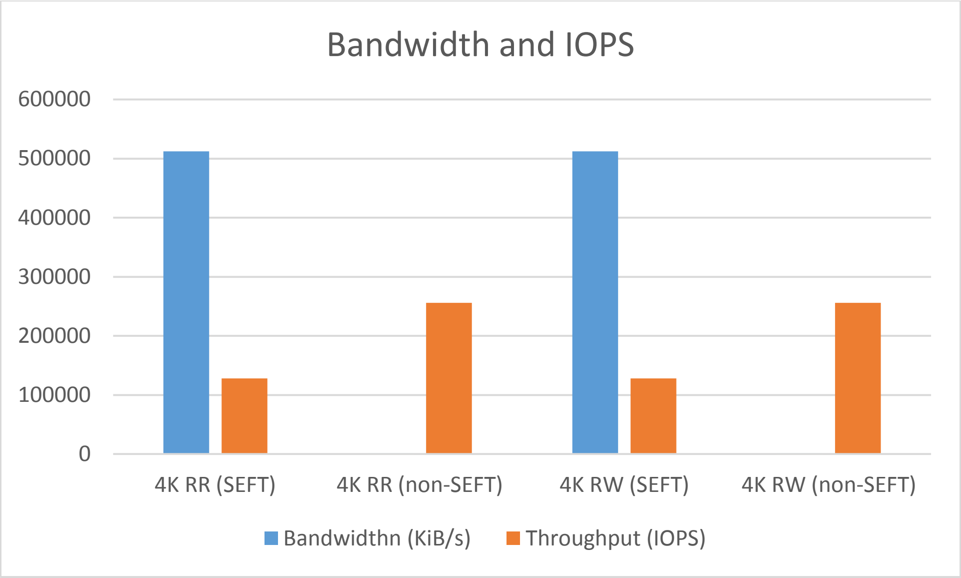


Figure 10 – Bandwidth and IOPS Measurements

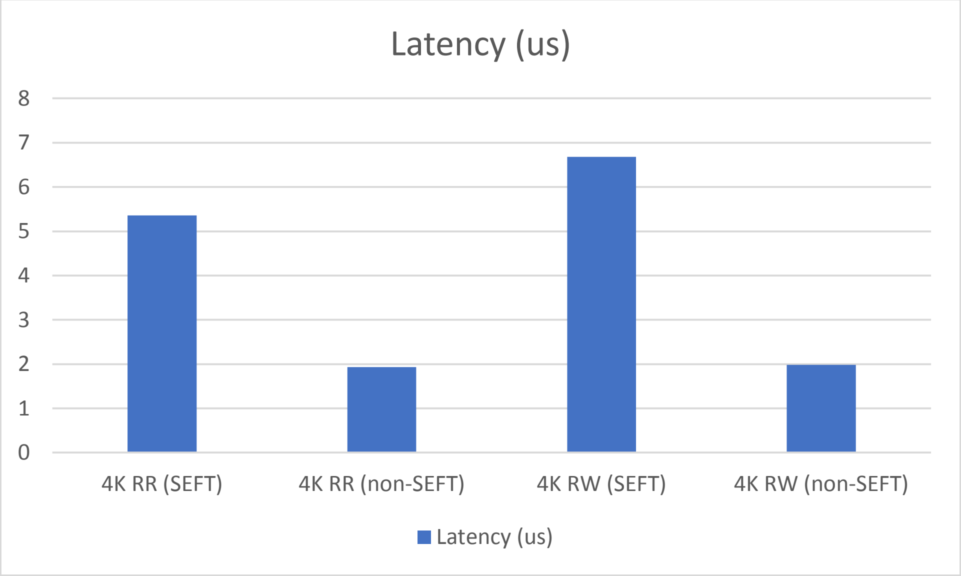


Figure 11 – I/O Latency Measurements

The bandwidth and IOPS (I/Os per second) showed mixed results. The bandwidth was orders of magnitude higher with the SEFT extension code (averaging 512 MiB/s). But the IOPS were exactly half with the SEFT extension code (averaging

128KiB IOPS). However, this lower IOPS can be attributed to the most surprising measurement, the increase in latency. IOPS are a directly result of latency. If latency is higher per I/O, then IOPS will be slower. Because the I/O latency was just below 2 microseconds on the non-SEFT FAT file system, the IOPS were significantly lower. The bandwidth increase seen with the SEFT extension is attributed to the fact that more data per second was transferred (versus the non-SEFT FAT file system). It can be concluded that while throughput increased, there is still a significant CPU overhead being added by the SEFT extension code executing. The numbers measured in this research are somewhat contradictory, but also make sense considering a new code path was added. The new SEFT code path was not originally designed for the FAT file system, especially with respect to file allocation, so there are improvements that could be made to SEFT that could allocate memory more efficiently.

CHAPTER 5

CONCLUSION

Discussion

The analysis of the port of the DAX extension to the FAT file system revealed several key data points around the viability and portability of a common file extension for supporting SCM storage block devices. Several key issues were identified prior to the investigation and research into the SEFT modifications. Those key issues were performance overhead, contiguous address space for large files, dynamic (run-time) memory mapping of files, and write order. However, the issues identified during the research and implementation of SEFT were different from those issues identified prior. The key issues identified during the implementation more accurately reflected the portability and compatibility of a SCM file system extension. The key issues identified during implementation were file space allocation, mapping of logical sectors/blocks on a SCM storage block device to kernel memory, and configuring the underlying storage medium to be able to align to system memory page size. And while important, the initial four key issues really defined the potential architectural limitations of such a file system extension.

To address the initial four key issues, an examination of the SEFT implementation provides answers to all four. First, the performance overhead of sharing system resources with a file system is a challenging task to identify or measure. A system would require a heavy I/O workload to any SCM block storage device in addition to performing complex, CPU and MMU intensive operations in order to fully realize the bottleneck that would, or could, be encountered. However, even with such workloads, the variability of system resources and configurations would drastically affect the results of any such measurements. In short, the performance overhead of shared resources, such as the MMU, is not a reliable source for analysis and measurement as every system differs in resource capabilities. The SEFT modifications use several native Linux kernel memory libraries, structures, and operations. It would be difficult to identify and measure the resource utilization of operations originated by SEFT.

The second issue of contiguous address space for large files was a testament of the unknown. This original issue/concern was based on the concept that all memory mapped regions must be physically contiguous. However, this assumption turned out to be false. This was due in large part to how OS kernels memory map PCI configuration space (which is one of the most common scenarios to memory map an external device to the system's memory map). However, what was not understood at the time of the initial investigation was that not all memory mapped space need be physically contiguous. And this is true for any type of storage block device (SCM, RAM disk, or otherwise). The physically contiguous requirement of memory that is memory mapped is specific to PCI configuration space due to the nature of how PCI devices must be compatible across all systems. PCI devices present a set of common and uncommon registers sets. The PCI specifications explicitly state that in order to maintain compatibility with all systems, the address space that is being mapped must be physically contiguous for direct memory addressing (DMA) purposes. However, with block storage devices, because all data is accessed via LBAs, there is no need to ensure the memory mapping is physically contiguous because the LBAs are described by buffer head structures that are linked in the memory page structure. This was a critical finding in the implementation of SEFT.

For the third initial issue, run time memory allocation, some file systems actually prevent this from happening, and instead force all allocation of resources to be done at run time. At the same time, some file systems do allocate resources upon mount time. Therefore, providing a common run time solution would not be effective as resource management, in particular memory mapping of address space, because resource management is file system specific. The initial concern here revolved around the time taken to memory map files. However, during the implementation of SEFT, it became clear that memory mapping at run time is not an issue due to the nature of how easily the Linux kernel maps memory. As files are created and deleted, the data block storage to kernel memory translation must happen regardless of whether memory has been mapped ahead of time. The act of doing the memory mapping does not place any additional overhead or resource contention on the file creation and deletion process.

The fourth issue of write ordering turned out to be a non-issue due to the removal of the page/buffer cache from the SEFT I/O path. Since there is no buffering above the file system layer in any kernel, SEFT I/O requests that come down will be processed in the order in which they were received. The file system will bypass any buffered reads or writes and perform all I/O in real time via loads and stores. This ensures that write ordering is preserved.

The key issues discovered during the implementation of SEFT are really the key issues that determine portability and compatibility. As identified above, all of the issues revolved around how individual file systems allocate memory for files and map the LBAs to the kernel memory address space. In Table 3 below, each of the key issues identified through the research have been categorized with respect to file system frameworks and portability. Note that for the SEFT I/O functions, in order to maintain portability (and remain file system agnostic), the SCM file system extension must use the OS native structures that represent I/O commands and requests, memory, user space buffers, and the arrangement of sectors on storage block devices.

Key SCM Extension Components	File System Agnostic	Operating System Agnostic
File Space Allocation	No	No
Logical Block to Memory Mapping	No	No
SCM Extension I/O Functions	Yes*	Yes
Inode Modification and Detection	Yes	No
SCM Test Infrastructure	Yes	No

Table 4 – Portability Matrix of SCM Extension Components

Future Work

There are many areas of file system extensions that can be investigated for future work with respect to SCM memory. The future potential work can be categorized into three categories: continuation of portability, performance enhancements, and functional enhancements. Each of these categories has its own challenges and assumptions. With the continuing study of portability, the emergence of commercial SCM devices will help accelerate this investigation. From this thesis, to other similar papers and file system prototypes, there has not been a completely file system agnostic solution (including SEFT). Because of the nature of file systems and memory management, there are few ways to abstract all the functionality of reading and writing data to a storage medium that do not include using native structures. Of course, abstraction and glue layers can always be introduced in order to provide buffering between the heart of a SCM extension and the file system on which it runs.

But, the concept of adding shim or glue layers goes against the very principle of emerging SCM technologies. SCM technologies reduce the device access latency times down to single millisecond numbers, and possibly even further (i.e., hundreds of nanoseconds). Introducing even the smallest piece of unnecessary code adds to overall I/O latency. The overall I/O latency was once driven by the storage media access time, but now the time spent in the OS kernel now become relevant, especially as we get closer to nanosecond storage media access times. There is a point of diminishing returns when trying to develop an overly portable or compatible piece of software, and with SCM devices, that point is an important factor for performance.

The potential performance and functional investigation work should center on what can be done to help users access their data faster and easier. Much of this may be determined by the type of SCM technologies that are released, as well as which ones become viable market solutions. The storage ecosystem is constantly evolving and there are innovative solutions introduced every year to squeeze more and more performance and functionality out of our computers. But SCM is a game-changing technology, thus opening the doors to a wider array of possible solutions.

REFERENCES

[1] Y. Kang, J. Yang, E. L. Miller, "Object-based SCM: An Efficient Interface for Storage Class Memories," in Proceedings of 2011 IEEE 27th Symposium on Mass Storage Systems and Technologies (MSST).

[2] X. Wu and A. L. Narashimha Reddy, "SCMFS: A File System for Storage Class Memory," in Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, Article No. 39.

[3] H. Volos, S. Nalli, S. Panneerselvam, V. Varadarajan, P. Saxena, M. Swift, "Aerie: Flexible File-System Interfaces to Storage-Class Memory", in Proceedings of the 9th European Conference on Computer Systems, Article No. 14.

[4] Storage Class Memory, Towards a disruptively low-cost solid-state non-volatile memory, IBM Almaden Research Center. URL http://researcher.watson.ibm.com/researcher/files/us-gwburr/Almaden_SCM_overview_Jan2013.pdf

[5] M. Wilcox, DAX: Page cache bypass for file systems on memory storage. URL https://lwn.net/Articles/618064/

[6] PMFS: A file system for persistent memory. URL https://github.com/linux-pmfs/pmfs/blob/master/ Documentation/filesystems/pmfs.txt, 2013

[7] A. Brouwer, FAT under Linux, The FAT filesystem. URL http://www.win.tue.nl/~aeb/linux/fs/fat/fat-2.html

[8] Description of the FAT32 File System. URL https://support.microsoft.com/en-us/kb/154997

[9] T. Krenn, Linux Page Cache Basics. URL https://www.thomas-krenn.com/en/wiki/Linux_Page_Cache_Basics

[10] File Allocation Table. URL https://en.wikipedia.org/wiki/File_Allocation_Table

[11] Design of the FAT File System. URL https://en.wikipedia.org/wiki/Design_of_the_FAT_file_system

[12] L. Torvalds, Buffer Heads. URL http://yarchive.net/comp/linux/buffer_heads.html

[13] J. Corbet, A Nasty File Corruption Bug – Fixed. URL http://lwn.net/Articles/215235/

[14] J. Corbet, The iov_iter Interface. URL https://lwn.net/Articles/625077/

[15] J. Corbet, Asynchronous block loop I/O. URL https://lwn.net/Articles/535034/

[16] G. Kroah-Hartman, The Kernel Configuration and Build Process, The Linux Journal. URL http://www.linuxjournal.com/article/6568

[17] Platform Controller Hub. URL https://en.wikipedia.org/wiki/Platform_Controller_Hub

[18] J. Layton, What's an Inode? Linux Magazine. URL http://www.linux-mag.com/id/8658/

[19] B. Fields, Virtual File System Switch. URL http://www.fieldses.org/~bfields/kernel/vfs.txt

[20] M. Tim Jones, Anatomy of the Linux Virtual File System switch. URL http://www.ibm.com/developerworks/library/l-virtual-filesystem-switch/

[21] M. Tim Jones, Anatomy of the Linux File System. URL http://web.archive.org/web/20150505112327/http://www.ibm.com/developerworks/linux/library/l-linux-filesystem/

[22] Inode Pointer Structure. URL https://en.wikipedia.org/wiki/Inode_pointer_structure

[23] Intel Z87 Chipset Platform Diagram. URL http://www.intel.com/content/www/us/en/chipsets/performance-chipsets/z87-chipset-diagram.html

[24] S. Qiu and A. L. Narashimha Reddy, "NVMFS: A Hybrid File System for Improving Random Write in NAND-flash SSD," in Proceedings of 2013 IEEE 29th Symposium on Mass Storage Systems and Technologies (MSST).

[25] FIO(1) – Linux Man Page. URL http://linux.die.net/man/1/fio

APPENDIX A

SEFT CODE KERNEL PATCH

GitHub Link to Repo – https://github.com/rcrobles/sxf-linux-4.3.5.git

```
diff --git a/drivers/block/Kconfig b/drivers/block/Kconfig
index 1b8094d..d564f18 100644
--- a/drivers/block/Kconfig
+++ b/drivers/block/Kconfig
@@ -404,6 +404,17 @@ config BLK_DEV_RAM_DAX
           and will prevent RAM block device backing store memory from being
           allocated from highmem (only a problem for highmem systems).

+config BLK_DEV_RAM_SEFT
+        bool "Support SCM Extension for FAT (SEFT) to RAM block devices"
+        depends on BLK_DEV_RAM && FS_SEFT
+        default n
+        help
+          Support filesystems using SEFT to access RAM block devices.  This
+          avoids double-buffering data in the page cache before copying it
+          to the block device.  Answering Y will slightly enlarge the kernel,
+          and will prevent RAM block device backing store memory from being
+          allocated from highmem (only a problem for highmem systems).
+
 config CDROM_PKTCDVD
         tristate "Packet writing on CD/DVD media"
         depends on !UML
diff --git a/drivers/block/brd.c b/drivers/block/brd.c
index b9794ae..334cdfc 100644
--- a/drivers/block/brd.c
+++ b/drivers/block/brd.c
@@ -103,7 +103,7 @@ static struct page *brd_insert_page(struct brd_device *brd,
sector_t sector)
          * restriction might be able to be lifted.
          */
         gfp_flags = GFP_NOIO | __GFP_ZERO;
-#ifndef CONFIG_BLK_DEV_RAM_DAX
+#if !defined(CONFIG_BLK_DEV_RAM_DAX) &&
!defined(CONFIG_BLK_DEV_RAM_SEFT)
         gfp_flags |= __GFP_HIGHMEM;
 #endif
         page = alloc_page(gfp_flags);
@@ -372,18 +372,22 @@ static int brd_rw_page(struct block_device *bdev,
sector_t sector,
         return err;
 }

-#ifdef CONFIG_BLK_DEV_RAM_DAX
+#if defined(CONFIG_BLK_DEV_RAM_DAX) ||
defined(CONFIG_BLK_DEV_RAM_SEFT)
 static long brd_direct_access(struct block_device *bdev, sector_t sector,
-                             void __pmem **kaddr, unsigned long *pfn)
+                              void __pmem **kaddr, unsigned long *pfn)
 {
        struct brd_device *brd = bdev->bd_disk->private_data;
        struct page *page;

-       if (!brd)
+       if (!brd) {
                return -ENODEV;
+       }
+
        page = brd_insert_page(brd, sector);
-       if (!page)
+       if (!page) {
                return -ENOSPC;
+       }
+
        *kaddr = (void __pmem *)page_address(page);
        *pfn = page_to_pfn(page);

diff --git a/fs/Kconfig b/fs/Kconfig
index da3f32f..0f78ef9 100644
--- a/fs/Kconfig
+++ b/fs/Kconfig
@@ -31,6 +31,13 @@ source "fs/btrfs/Kconfig"
 source "fs/nilfs2/Kconfig"
 source "fs/f2fs/Kconfig"

+config FS_SEFT
+       bool "SCM Extension for FAT (SEFT) Support"
+       depends on MMU
+       depends on !(ARM || MIPS || SPARC)
+       help
+         SEFT can be used on directly addressable persistent block devices
+
 config FS_DAX
        bool "Direct Access (DAX) support"
        depends on MMU
diff --git a/fs/Makefile b/fs/Makefile
index f79cf40..1a51e4e 100644
--- a/fs/Makefile
+++ b/fs/Makefile
@@ -30,6 +30,7 @@ obj-$(CONFIG_EVENTFD)          += eventfd.o
 obj-$(CONFIG_USERFAULTFD)      += userfaultfd.o
 obj-$(CONFIG_AIO)              += aio.o
 obj-$(CONFIG_FS_DAX)           += dax.o
+obj-$(CONFIG_FS_SEFT)          += seft.o
 obj-$(CONFIG_FILE_LOCKING)     += locks.o
 obj-$(CONFIG_COMPAT)           += compat.o compat_ioctl.o
 obj-$(CONFIG_BINFMT_AOUT)      += binfmt_aout.o
diff --git a/fs/block_dev.c b/fs/block_dev.c
index 073bb57..0838a90 100644
--- a/fs/block_dev.c
+++ b/fs/block_dev.c
@@ -29,6 +29,9 @@
```

```
 #include <linux/log2.h>
 #include <linux/cleancache.h>
 #include <linux/dax.h>
+
+#include <linux/seft.h>
+
 #include <asm/uaccess.h>
 #include "internal.h"

@@ -153,12 +156,18 @@ blkdev_direct_IO(struct kiocb *iocb, struct iov_iter *iter,
loff_t offset)
        struct file *file = iocb->ki_filp;
        struct inode *inode = file->f_mapping->host;

-       if (IS_DAX(inode))
-               return dax_do_io(iocb, inode, iter, offset, blkdev_get_block,
-                               NULL, DIO_SKIP_DIO_COUNT);
-       return __blockdev_direct_IO(iocb, inode, I_BDEV(inode), iter, offset,
-                                   blkdev_get_block, NULL, NULL,
-                                   DIO_SKIP_DIO_COUNT);
+       if (IS_DAX(inode)) {
+           return dax_do_io(iocb, inode, iter, offset, blkdev_get_block,
+                       NULL, DIO_SKIP_DIO_COUNT);
+       } else if (IS_SEFT(inode)) {
+           printk(KERN_NOTICE "SEFT: blkdev_direct_IO: calling seft_do_io");
+           return seft_do_io(iocb, inode, iter, offset, blkdev_get_block,
+                        NULL, DIO_SKIP_DIO_COUNT);
+       } else {
+           return __blockdev_direct_IO(iocb, inode, I_BDEV(inode), iter, offset,
+                                  blkdev_get_block, NULL, NULL,
+                                  DIO_SKIP_DIO_COUNT);
+       }
 }

 int __sync_blockdev(struct block_device *bdev, int wait)
@@ -381,8 +390,9 @@ int bdev_read_page(struct block_device *bdev, sector_t
sector,
                        struct page *page)
 {
        const struct block_device_operations *ops = bdev->bd_disk->fops;
-       if (!ops->rw_page || bdev_get_integrity(bdev))
+       if (!ops->rw_page || bdev_get_integrity(bdev)) {
               return -EOPNOTSUPP;
+       }
        return ops->rw_page(bdev, sector + get_start_sect(bdev), page, READ);
 }
 EXPORT_SYMBOL_GPL(bdev_read_page);
@@ -412,8 +422,9 @@ int bdev_write_page(struct block_device *bdev, sector_t
sector,
        int result;
        int rw = (wbc->sync_mode == WB_SYNC_ALL) ? WRITE_SYNC : WRITE;
        const struct block_device_operations *ops = bdev->bd_disk->fops;
```

```
-       if (!ops->rw_page || bdev_get_integrity(bdev))
+       if (!ops->rw_page || bdev_get_integrity(bdev)) {
                return -EOPNOTSUPP;
+       }
        set_page_writeback(page);
        result = ops->rw_page(bdev, sector + get_start_sect(bdev), page, rw);
        if (result)
@@ -455,17 +466,26 @@ long bdev_direct_access(struct block_device *bdev,
sector_t sector,

        if (size < 0)
                return size;
-       if (!ops->direct_access)
+       if (!ops->direct_access) {
                return -EOPNOTSUPP;
-       if ((sector + DIV_ROUND_UP(size, 512)) >
-                                       part_nr_sects_read(bdev->bd_part))
+       }
+
+       if ((sector + DIV_ROUND_UP(size, 512)) > part_nr_sects_read(bdev-
>bd_part)) {
                return -ERANGE;
+       }
+
        sector += get_start_sect(bdev);
-       if (sector % (PAGE_SIZE / 512))
-               return -EINVAL;
+
+       if (sector % (PAGE_SIZE / 512)) {
+              printk(KERN_NOTICE "SEFT: bdev_direct_access: sector = 0x%llx,
PAGE_SIZE = 0x%lx\n",
+                    (unsigned long long)sector, PAGE_SIZE);
+       }
+
        avail = ops->direct_access(bdev, sector, addr, pfn);
-       if (!avail)
+       if (!avail) {
                return -ERANGE;
+       }
+
        return min(avail, size);
 }
 EXPORT_SYMBOL_GPL(bdev_direct_access);
@@ -480,8 +500,10 @@ static struct kmem_cache * bdev_cachep __read_mostly;
 static struct inode *bdev_alloc_inode(struct super_block *sb)
 {
        struct bdev_inode *ei = kmem_cache_alloc(bdev_cachep, GFP_KERNEL);
+
        if (!ei)
                return NULL;
+
        return &ei->vfs_inode;
 }

@@ -1636,6 +1658,7 @@ ssize_t blkdev_write_iter(struct kiocb *iocb, struct
iov_iter *from)
                        ret = err;
        }
        blk_finish_plug(&plug);
+
        return ret;
 }
 EXPORT_SYMBOL_GPL(blkdev_write_iter);
diff --git a/fs/fat/cache.c b/fs/fat/cache.c
index 93fc622..e637e0d 100644
--- a/fs/fat/cache.c
+++ b/fs/fat/cache.c
@@ -127,6 +127,7 @@ static struct fat_cache *fat_cache_merge(struct inode
*inode,
                        return p;
                }
        }
+
        return NULL;
 }

@@ -233,8 +234,9 @@ int fat_get_cluster(struct inode *inode, int cluster, int
*fclus, int *dclus)

        *fclus = 0;
        *dclus = MSDOS_I(inode)->i_start;
-       if (cluster == 0)
-               return 0;
+       if (cluster == 0) {
+           return 0;
+       }

        if (fat_cache_lookup(inode, cluster, &cid, fclus, dclus) < 0) {
                /*
@@ -257,8 +259,9 @@ int fat_get_cluster(struct inode *inode, int cluster, int
*fclus, int *dclus)
                }

                nr = fat_ent_read(inode, &fatent, *dclus);
-               if (nr < 0)
-                       goto out;
+               if (nr < 0) {
+                   goto out;
+               }
                else if (nr == FAT_ENT_FREE) {
                        fat_fs_error_ratelimit(sb,
                                      "%s: invalid cluster chain (i_pos %lld)",
@@ -270,11 +273,14 @@ int fat_get_cluster(struct inode *inode, int cluster, int
*fclus, int *dclus)
```

```
                        fat_cache_add(inode, &cid);
                        goto out;
                }
+
                (*fclus)++;
                *dclus = nr;
+
                if (!cache_contiguous(&cid, *dclus))
                        cache_init(&cid, *fclus, *dclus);
        }
+
        nr = 0;
        fat_cache_add(inode, &cid);
 out:
@@ -287,17 +293,21 @@ static int fat_bmap_cluster(struct inode *inode, int
cluster)
        struct super_block *sb = inode->i_sb;
        int ret, fclus, dclus;

-       if (MSDOS_I(inode)->i_start == 0)
+       if (MSDOS_I(inode)->i_start == 0) {
                return 0;
+       }

        ret = fat_get_cluster(inode, cluster, &fclus, &dclus);
-       if (ret < 0)
+       if (ret < 0) {
                return ret;
+       }
+
        else if (ret == FAT_ENT_EOF) {
                fat_fs_error(sb, "%s: request beyond EOF (i_pos %lld)",
                                      __func__, MSDOS_I(inode)->i_pos);
                return -EIO;
        }
+
        return dclus;
 }

@@ -313,39 +323,50 @@ int fat_bmap(struct inode *inode, sector_t sector,
sector_t *phys,

        *phys = 0;
        *mapped_blocks = 0;
+
        if ((sbi->fat_bits != 32) && (inode->i_ino == MSDOS_ROOT_INO)) {
                if (sector < (sbi->dir_entries >> sbi->dir_per_block_bits)) {
                        *phys = sector + sbi->dir_start;
                        *mapped_blocks = 1;
                }
+
                return 0;
        }

        last_block = (i_size_read(inode) + (blocksize - 1)) >> blocksize_bits;
        if (sector >= last_block) {
-               if (!create)
+               if (!create) {
                        return 0;
+               }

                /*
                 * ->mmu_private can access on only allocation path.
                 * (caller must hold ->i_mutex)
                 */
-               last_block = (MSDOS_I(inode)->mmu_private + (blocksize - 1))
-                       >> blocksize_bits;
-               if (sector >= last_block)
+               last_block = (MSDOS_I(inode)->mmu_private + (blocksize - 1)) >>
blocksize_bits;
+               if (sector >= last_block) {
                        return 0;
+               }
        }

        cluster = sector >> (sbi->cluster_bits - sb->s_blocksize_bits);
        offset  = sector & (sbi->sec_per_clus - 1);
        cluster = fat_bmap_cluster(inode, cluster);
-       if (cluster < 0)
+
+       if (cluster < 0) {
                return cluster;
-       else if (cluster) {
+       } else if (cluster) {
                *phys = fat_clus_to_blknr(sbi, cluster) + offset;
-               *mapped_blocks = sbi->sec_per_clus - offset;
-               if (*mapped_blocks > last_block - sector)
+               //*mapped_blocks = sbi->sec_per_clus - offset;
+               *mapped_blocks = (sbi->sec_per_clus * 2) - offset;
+
+               fat_msg(sb, KERN_NOTICE, "SEFT: fat_bmap: *phys = 0x%llx",
(unsigned long long)*phys);
+               fat_msg(sb, KERN_NOTICE, "SEFT: fat_bmap: *mapped_blocks =
0x%lx", *mapped_blocks);
+
+               if (*mapped_blocks > last_block - sector) {
                        *mapped_blocks = last_block - sector;
+               }
        }
+
        return 0;
 }
diff --git a/fs/fat/dir.c b/fs/fat/dir.c
index 4afc4d9..bd9d679 100644
```

```
--- a/fs/fat/dir.c
+++ b/fs/fat/dir.c
@@ -1377,26 +1377,29 @@ found:
                        dir->i_size = (dir->i_size + sbi->cluster_size - 1)
                                & ~((loff_t)sbi->cluster_size - 1);
                }
+
                dir->i_size += nr_cluster << sbi->cluster_bits;
                MSDOS_I(dir)->mmu_private += nr_cluster << sbi->cluster_bits;
        }
+
        sinfo->slot_off = pos;
        sinfo->de = de;
        sinfo->bh = bh;
        sinfo->i_pos = fat_make_i_pos(sb, sinfo->bh, sinfo->de);

        return 0;
-
 error:
        brelse(bh);
        for (i = 0; i < nr_bhs; i++)
                brelse(bhs[i]);
+
        return err;

 error_remove:
        brelse(bh);
        if (free_slots)
                __fat_remove_entries(dir, pos, free_slots);
+
        return err;
 }
 EXPORT_SYMBOL_GPL(fat_add_entries);
diff --git a/fs/fat/fat.h b/fs/fat/fat.h
index be5e153..1d765cb 100644
--- a/fs/fat/fat.h
+++ b/fs/fat/fat.h
@@ -50,7 +50,10 @@ struct fat_mount_options {
                 tz_set:1,          /* Filesystem timestamps' offset set */
                 rodir:1,           /* allow ATTR_RO for directory */
                 discard:1,         /* Issue discard requests on deletions */
-                dos1xfloppy:1;          /* Assume default BPB for DOS 1.x floppies */
+                dos1xfloppy:1,          /* Assume default BPB for DOS 1.x floppies */
+//#ifdef CONFIG_FS_SEFT
+               seft:1;          /* 1 = SEFT enabled (SCM extensions), 0 = SEFT
disabled */
+//#endif
 };

 #define FAT_HASH_BITS   8
@@ -357,7 +360,11 @@ extern int fat_count_free_clusters(struct super_block *sb);
 /* fat/file.c */
 extern long fat_generic_ioctl(struct file *filp, unsigned int cmd,
                               unsigned long arg);
+
+
 extern const struct file_operations fat_file_operations;
+
+
 extern const struct inode_operations fat_file_inode_operations;
 extern int fat_setattr(struct dentry *dentry, struct iattr *attr);
 extern void fat_truncate_blocks(struct inode *inode, loff_t offset);
@@ -380,6 +387,9 @@ extern int fat_fill_inode(struct inode *inode, struct
msdos_dir_entry *de);

 extern int fat_flush_inodes(struct super_block *sb, struct inode *i1,
                             struct inode *i2);
+extern int fat_get_block(struct inode *inode, sector_t iblock,
+                       struct buffer_head *bh_result, int create);
+
 static inline unsigned long fat_dir_hash(int logstart)
 {
         return hash_32(logstart, FAT_HASH_BITS);
diff --git a/fs/fat/fatent.c b/fs/fat/fatent.c
index 8226557..1a5d716 100644
--- a/fs/fat/fatent.c
+++ b/fs/fat/fatent.c
@@ -527,7 +527,6 @@ int fat_alloc_clusters(struct inode *inode, int *cluster, int
nr_cluster)
         sbi->free_clusters = 0;
         sbi->free_clus_valid = 1;
         err = -ENOSPC;
-
 out:
         unlock_fat(sbi);
         mark_fsinfo_dirty(sb);
diff --git a/fs/fat/file.c b/fs/fat/file.c
index a08f103..f8883ad 100644
--- a/fs/fat/file.c
+++ b/fs/fat/file.c
@@ -6,6 +6,7 @@
  *  regular file handling primitives for fat-based filesystems
  */

+#include <linux/fs.h>
 #include <linux/capability.h>
 #include <linux/module.h>
 #include <linux/compat.h>
@@ -14,8 +15,48 @@
 #include <linux/backing-dev.h>
 #include <linux/fsnotify.h>
 #include <linux/security.h>
+#include <linux/seft.h>
 #include "fat.h"
```

```
+#ifdef CONFIG_FS_SEFT
+static int fat_seft_fault(struct vm_area_struct *vma, struct vm_fault *vmf)
+{
+	return seft_fault(vma, vmf, fat_get_block, NULL);
+}
+
+static int fat_seft_pmd_fault(struct vm_area_struct *vma, unsigned long addr,
+				pmd_t *pmd, unsigned int flags)
+{
+	return seft_pmd_fault(vma, addr, pmd, flags, fat_get_block, NULL);
+}
+
+static int fat_seft_mkwrite(struct vm_area_struct *vma, struct vm_fault *vmf)
+{
+	return seft_mkwrite(vma, vmf, fat_get_block, NULL);
+}
+
+static const struct vm_operations_struct fat_seft_vm_ops = {
+	.fault		= fat_seft_fault,
+	.pmd_fault	= fat_seft_pmd_fault,
+	.page_mkwrite	= fat_seft_mkwrite,
+	.pfn_mkwrite	= seft_pfn_mkwrite,
+};
+
+static int seft_file_mmap(struct file *file, struct vm_area_struct *vma)
+{
+	if (!IS_SEFT(file_inode(file)))
+		return generic_file_mmap(file, vma);
+
+	file_accessed(file);
+	vma->vm_ops = &fat_seft_vm_ops;
+	vma->vm_flags |= VM_MIXEDMAP | VM_HUGEPAGE;
+	//vma->vm_flags |= VM_MIXEDMAP;
+	return 0;
+}
+#else
+#define seft_file_mmap	generic_file_mmap
+#endif
+
 static int fat_ioctl_get_attributes(struct inode *inode, u32 __user *user_attr)
 {
 	u32 attr;
@@ -164,12 +205,11 @@ int fat_file_fsync(struct file *filp, loff_t start, loff_t end, int datasync)
 	return res ? res : err;
 }
 
-
 const struct file_operations fat_file_operations = {
 	.llseek		= generic_file_llseek,
-	.read_iter	= generic_file_read_iter,
-	.write_iter	= generic_file_write_iter,
-	.mmap		= generic_file_mmap,
+	.read_iter	= generic_file_read_iter,
+	.write_iter	= generic_file_write_iter,
+	.mmap		= seft_file_mmap,
 	.release	= fat_file_release,
 	.unlocked_ioctl		= fat_generic_ioctl,
 #ifdef CONFIG_COMPAT
diff --git a/fs/fat/inode.c b/fs/fat/inode.c
index 509411d..9fce733 100644
--- a/fs/fat/inode.c
+++ b/fs/fat/inode.c
@@ -19,6 +19,9 @@
 #include <linux/uio.h>
 #include <linux/blkdev.h>
 #include <linux/backing-dev.h>
+
+#include <linux/seft.h>
+
 #include <asm/unaligned.h>
 #include "fat.h"

@@ -95,16 +98,39 @@ static struct fat_floppy_defaults {

 static int fat_add_cluster(struct inode *inode)
 {
-	int err, cluster;
+	int err = 0;
+	int cluster = 0;
+	int index = 0;
+
+	/* For SEFT I/O, need to allocate 4K (2 clusters) */
+	if (IS_SEFT(inode)) {
+	    for (index = 0; index < 2; index++) {
+		err = fat_alloc_clusters(inode, &cluster, 1);
+		if (err) {
+			return err;
+		}
+
+		/* FIXME: this cluster should be added after data of this cluster is writed
*/
+		/* NOTE: After fat_alloc_clusters, the var clusters holds fat_ent.entry
value */
+		/* NOTE: After final call to fat_chain_add, inode->i_blocks = 8 */
+		err = fat_chain_add(inode, cluster, 1);
+		if (err) {
+			fat_free_clusters(inode, cluster);
+		}
+	    } /* end for loop*/
+	} else {
+		err = fat_alloc_clusters(inode, &cluster, 1);
+		if (err) {
```

```
+                       return err;
+               }
+
+               /* FIXME: this cluster should be added after data of this cluster is writed
*/
+               err = fat_chain_add(inode, cluster, 1);
+               if (err) {
+                       fat_free_clusters(inode, cluster);
+               }
+       }

-       err = fat_alloc_clusters(inode, &cluster, 1);
-       if (err)
-               return err;
-       /* FIXME: this cluster should be added after data of this
-        * cluster is writed */
-       err = fat_chain_add(inode, cluster, 1);
-       if (err)
-               fat_free_clusters(inode, cluster);
        return err;
 }

@@ -119,15 +145,19 @@ static inline int __fat_get_block(struct inode *inode,
sector_t iblock,
        int err, offset;

        err = fat_bmap(inode, iblock, &phys, &mapped_blocks, create);
-       if (err)
+       if (err) {
                return err;
+       }
+
        if (phys) {
                map_bh(bh_result, sb, phys);
                *max_blocks = min(mapped_blocks, *max_blocks);
                return 0;
        }
-       if (!create)
+
+       if (!create) {
                return 0;
+       }

        if (iblock != MSDOS_I(inode)->mmu_private >> sb->s_blocksize_bits) {
                fat_fs_error(sb, "corrupted file size (i_pos %lld, %lld)",
@@ -138,30 +168,58 @@ static inline int __fat_get_block(struct inode *inode,
sector_t iblock,
        offset = (unsigned long)iblock & (sbi->sec_per_clus - 1);
        if (!offset) {
                /* TODO: multiple cluster allocation would be desirable. */
-               err = fat_add_cluster(inode);
-               if (err)
+               err = fat_add_cluster(inode);
+               if (err) {
                        return err;
+             }
        }
-       /* available blocks on this cluster */
-       mapped_blocks = sbi->sec_per_clus - offset;
+
+       /* Available blocks on this cluster -- sbi->sec_per_clus = 4 */
+       /* For SEFT I/O, 8 blocks will be mapped... 2 clusters */
+       /* After call to fat_add_cluster, inode->i_blocks = 8 */
+       if (IS_SEFT(inode)) {
+             //mapped_blocks = (sbi->sec_per_clus * 2) - offset;  // original
+             //mapped_blocks = inode->i_blocks - offset;  // not sure we can rely on
i_blocks
+             mapped_blocks = (sbi->sec_per_clus * 2) - offset;
+       } else {
+             mapped_blocks = sbi->sec_per_clus - offset;
+       }

        *max_blocks = min(mapped_blocks, *max_blocks);
        MSDOS_I(inode)->mmu_private += *max_blocks << sb->s_blocksize_bits;

        err = fat_bmap(inode, iblock, &phys, &mapped_blocks, create);
-       if (err)
+       if (err) {
                return err;
+       }

        BUG_ON(!phys);
-       BUG_ON(*max_blocks != mapped_blocks);
+       //BUG_ON(*max_blocks != mapped_blocks);
+
+       /*
+        * Come back and add check for SEFT inode... if so, then
+        * must clear the blocks (initialized) before they are put in chain
+        * that it's not found by another thread before it's initialized.
+        */
+       if (IS_SEFT(inode)) {
+          /*
+           * Block must be initialized before we put it in the chain so that
+           * it's not found by another thread before it's initialized.
+           */
+          err = seft_clear_blocks(inode, iblock, 1 << inode->i_blkbits);
+          if (err) {
+             fat_msg(sb, KERN_NOTICE, "SEFT: __fat_get_block: seft_clear_blocks
failed = 0x%x", err);
+          }
+       }
+
        set_buffer_new(bh_result);
        map_bh(bh_result, sb, phys);
```

```
         return 0;
 }

-static int fat_get_block(struct inode *inode, sector_t iblock,
-                          struct buffer_head *bh_result, int create)
+int fat_get_block(struct inode *inode, sector_t iblock,
+                 struct buffer_head *bh_result, int create)
 {
         struct super_block *sb = inode->i_sb;
         unsigned long max_blocks = bh_result->b_size >> inode->i_blkbits;
@@ -211,13 +269,14 @@ static int fat_write_begin(struct file *file, struct
address_space *mapping,
                         struct page **pagep, void **fsdata)
 {
         int err;
-
         *pagep = NULL;
+
         err = cont_write_begin(file, mapping, pos, len, flags,
                                 pagep, fsdata, fat_get_block,
                                 &MSDOS_I(mapping->host)->mmu_private);
         if (err < 0)
                 fat_write_failed(mapping, pos + len);
+
         return err;
 }

@@ -227,6 +286,7 @@ static int fat_write_end(struct file *file, struct
address_space *mapping,
 {
         struct inode *inode = mapping->host;
         int err;
+
         err = generic_write_end(file, mapping, pos, len, copied, pagep, fsdata);
         if (err < len)
                 fat_write_failed(mapping, pos + len);
@@ -235,6 +295,7 @@ static int fat_write_end(struct file *file, struct
address_space *mapping,
                 MSDOS_I(inode)->i_attrs |= ATTR_ARCH;
                 mark_inode_dirty(inode);
         }
+
         return err;
 }

@@ -247,7 +308,7 @@ static ssize_t fat_direct_IO(struct kiocb *iocb, struct iov_iter
*iter,
         size_t count = iov_iter_count(iter);
         ssize_t ret;

-        if (iov_iter_rw(iter) == WRITE) {
```

```
+       if (iov_iter_rw(iter) == WRITE) {
                /*
                 * FIXME: blockdev_direct_IO() doesn't use ->write_begin(),
                 * so we need to update the ->mmu_private to block boundary.
@@ -258,17 +319,30 @@ static ssize_t fat_direct_IO(struct kiocb *iocb, struct
iov_iter *iter,
                 * Return 0, and fallback to normal buffered write.
                 */
                loff_t size = offset + count;
-               if (MSDOS_I(inode)->mmu_private < size)
-                       return 0;
+             fat_msg(inode->i_sb, KERN_NOTICE, "SEFT: fat_direct_IO:
mmu_private = 0x%llx",
+                    (long long)MSDOS_I(inode)->mmu_private);
+
+             // Commenting out for test... should we return here if mmu_private is
0?????
+             // NOTE: This probably should not fail after cluster allocation fix...
+               //if (MSDOS_I(inode)->mmu_private < size) {
+               //      return 0;
+             //}
        }

        /*
         * FAT need to use the DIO_LOCKING for avoiding the race
         * condition of fat_get_block() and ->truncate().
         */
-       ret = blockdev_direct_IO(iocb, inode, iter, offset, fat_get_block);
-       if (ret < 0 && iov_iter_rw(iter) == WRITE)
+       if (IS_SEFT(inode)) {
+           fat_msg(inode->i_sb, KERN_NOTICE, "SEFT: fat_direct_IO: calling
seft_do_io");
+           ret = seft_do_io(iocb, inode, iter, offset, fat_get_block, NULL,
DIO_LOCKING);
+       } else {
+           ret = blockdev_direct_IO(iocb, inode, iter, offset, fat_get_block);
+       }
+
+       if (ret < 0 && iov_iter_rw(iter) == WRITE) {
                fat_write_failed(mapping, offset + count);
+       }

        return ret;
 }
@@ -294,7 +368,11 @@ static sector_t _fat_bmap(struct address_space *mapping,
sector_t block)
  */
 int fat_block_truncate_page(struct inode *inode, loff_t from)
 {
-       return block_truncate_page(inode->i_mapping, from, fat_get_block);
+    if (IS_SEFT(inode)) {
+       return seft_truncate_page(inode, from, fat_get_block);
+    } else {
+        return block_truncate_page(inode->i_mapping, from, fat_get_block);
+    }
 }

 static const struct address_space_operations fat_aops = {
@@ -461,6 +539,18 @@ int fat_fill_inode(struct inode *inode, struct
msdos_dir_entry *de)
        inode->i_version++;
        inode->i_generation = get_seconds();

+       fat_msg(inode->i_sb, KERN_NOTICE, "SEFT: fat_fill_inode: de->name[0] =
%d", de->name[0]);
+       fat_msg(inode->i_sb, KERN_NOTICE, "SEFT: fat_fill_inode: de->name[1] =
%d", de->name[1]);
+       fat_msg(inode->i_sb, KERN_NOTICE, "SEFT: fat_fill_inode: de->name[2] =
%d", de->name[2]);
+       fat_msg(inode->i_sb, KERN_NOTICE, "SEFT: fat_fill_inode: de->name[3] =
%d", de->name[3]);
+       fat_msg(inode->i_sb, KERN_NOTICE, "SEFT: fat_fill_inode: de->name[4] =
%d", de->name[4]);
+       fat_msg(inode->i_sb, KERN_NOTICE, "SEFT: fat_fill_inode: de->name[5] =
%d", de->name[5]);
+       fat_msg(inode->i_sb, KERN_NOTICE, "SEFT: fat_fill_inode: de->name[6] =
%d", de->name[6]);
+       fat_msg(inode->i_sb, KERN_NOTICE, "SEFT: fat_fill_inode: de->name[7] =
%d", de->name[7]);
+       fat_msg(inode->i_sb, KERN_NOTICE, "SEFT: fat_fill_inode: de->name[8] =
%d", de->name[8]);
+       fat_msg(inode->i_sb, KERN_NOTICE, "SEFT: fat_fill_inode: de->name[9] =
%d", de->name[9]);
+       fat_msg(inode->i_sb, KERN_NOTICE, "SEFT: fat_fill_inode: de->name[10] =
%d", de->name[10]);
+
        if ((de->attr & ATTR_DIR) && !IS_FREE(de->name)) {
                inode->i_generation &= ~1;
                inode->i_mode = fat_make_mode(sbi, de->attr, S_IRWXUGO);
@@ -473,7 +563,6 @@ int fat_fill_inode(struct inode *inode, struct
msdos_dir_entry *de)
                if (error < 0)
                        return error;
                MSDOS_I(inode)->mmu_private = inode->i_size;
-
                set_nlink(inode, fat_subdirs(inode));
        } else { /* not a directory */
                inode->i_generation |= 1;
@@ -488,7 +577,15 @@ int fat_fill_inode(struct inode *inode, struct
msdos_dir_entry *de)
                inode->i_fop = &fat_file_operations;
                inode->i_mapping->a_ops = &fat_aops;
                MSDOS_I(inode)->mmu_private = inode->i_size;
-       }
```

```
+       }
+
+       /* Set i_flgs in inode to include S_SEFT based on sb mount option */
+       if (sbi->options.seft) {
+           inode->i_flags |= S_SEFT;
+       } else {
+           inode->i_flags &= ~S_SEFT;
+       }
+
        if (de->attr & ATTR_SYS) {
                if (sbi->options.sys_immutable)
                        inode->i_flags |= S_IMMUTABLE;
@@ -538,6 +635,7 @@ struct inode *fat_build_inode(struct super_block *sb,
        }
        inode->i_ino = iunique(sb, MSDOS_ROOT_INO);
        inode->i_version = 1;
+
        err = fat_fill_inode(inode, de);
        if (err) {
                iput(inode);
@@ -640,6 +738,7 @@ static struct kmem_cache *fat_inode_cachep;
 static struct inode *fat_alloc_inode(struct super_block *sb)
 {
        struct msdos_inode_info *ei;
+       fat_msg(sb, KERN_NOTICE, "SEFT: fat_alloc_inode\n");
        ei = kmem_cache_alloc(fat_inode_cachep, GFP_NOFS);
        if (!ei)
                return NULL;
@@ -656,6 +755,7 @@ static void fat_i_callback(struct rcu_head *head)

 static void fat_destroy_inode(struct inode *inode)
 {
+       fat_msg(inode->i_sb, KERN_NOTICE, "SEFT: fat_destroy_inode\n");
        call_rcu(&inode->i_rcu, fat_i_callback);
 }

@@ -700,6 +800,8 @@ static int fat_remount(struct super_block *sb, int *flags,
char *data)
        struct msdos_sb_info *sbi = MSDOS_SB(sb);
        *flags |= MS_NODIRATIME | (sbi->options.isvfat ? 0 : MS_NOATIME);

+       fat_msg(sb, KERN_NOTICE, "SEFT: fat_remount\n");
+
        sync_filesystem(sb);

        /* make sure we update state on remount. */
@@ -710,6 +812,7 @@ static int fat_remount(struct super_block *sb, int *flags,
char *data)
                else
                        fat_set_state(sb, 1, 1);
        }
+
        return 0;
 }

@@ -793,21 +896,23 @@ retry:
        if (wait)
                err = sync_dirty_buffer(bh);
        brelse(bh);
+
        return err;
 }

 static int fat_write_inode(struct inode *inode, struct writeback_control *wbc)
 {
        int err;
+       fat_msg(inode->i_sb, KERN_NOTICE, "SEFT: fat_write_inode\n");

        if (inode->i_ino == MSDOS_FSINFO_INO) {
                struct super_block *sb = inode->i_sb;
-
                mutex_lock(&MSDOS_SB(sb)->s_lock);
                err = fat_clusters_flush(sb);
                mutex_unlock(&MSDOS_SB(sb)->s_lock);
-       } else
+       } else {
                err = __fat_write_inode(inode, wbc->sync_mode == WB_SYNC_ALL);
+       }

        return err;
 }
@@ -935,6 +1040,7 @@ enum {
        Opt_obsolete, Opt_flush, Opt_tz_utc, Opt_rodir, Opt_err_cont,
        Opt_err_panic, Opt_err_ro, Opt_discard, Opt_nfs, Opt_time_offset,
        Opt_nfs_stale_rw, Opt_nfs_nostale_ro, Opt_err, Opt_dos1xfloppy,
+       Opt_seftenable,
 };

 static const match_table_t fat_tokens = {
@@ -1050,6 +1156,10 @@ static int parse_options(struct super_block *sb, char
*options, int is_vfat,
        opts->tz_set = 0;
        opts->nfs = 0;
        opts->errors = FAT_ERRORS_RO;
+#ifdef CONFIG_FS_SEFT
+       /* Turn on SEFT by default when mounting */
+       opts->seft = 1;
+#endif
        *debug = 0;

        if (!options)
@@ -1232,6 +1342,12 @@ static int parse_options(struct super_block *sb, char
*options, int is_vfat,
                        opts->discard = 1;
                        break;

+               /* Uncomment to allow enabling SEFT via FAT mount options */
+               /*case Opt_seftenable:
+                       opts->seft = 1;
+                       break;
+               */
+
                /* obsolete mount options */
                case Opt_obsolete:
                        fat_msg(sb, KERN_INFO, "\"%s\" option is obsolete, "
@@ -1497,6 +1613,8 @@ int fat_fill_super(struct super_block *sb, void *data, int
silent, int isvfat,
        long error;
        char buf[50];

+       fat_msg(sb, KERN_NOTICE, "SEFT: fat_fill_super");
+
        /*
         * GFP_KERNEL is ok here, because while we do hold the
         * supeblock lock, memory pressure can't call back into
@@ -1755,8 +1873,7 @@ int fat_fill_super(struct super_block *sb, void *data, int
silent, int isvfat,

 out_invalid:
        error = -EINVAL;
-       if (!silent)
-               fat_msg(sb, KERN_INFO, "Can't find a valid FAT filesystem");
+       fat_msg(sb, KERN_INFO, "Can't find a valid FAT filesystem");

 out_fail:
        if (fsinfo_inode)
@@ -1792,6 +1909,7 @@ static int writeback_inode(struct inode *inode)
        ret = sync_inode_metadata(inode, 0);
        if (!ret)
                ret = filemap_fdatawrite(inode->i_mapping);
+
        return ret;
 }

@@ -1806,6 +1924,7 @@ static int writeback_inode(struct inode *inode)
 int fat_flush_inodes(struct super_block *sb, struct inode *i1, struct inode *i2)
 {
        int ret = 0;
+
        if (!MSDOS_SB(sb)->options.flush)
                return 0;
        if (i1)
@@ -1816,6 +1935,7 @@ int fat_flush_inodes(struct super_block *sb, struct inode
*i1, struct inode *i2)
                struct address_space *mapping = sb->s_bdev->bd_inode-
>i_mapping;
```

```
 		ret = filemap_flush(mapping);
 	}
+
 	return ret;
 }
 EXPORT_SYMBOL_GPL(fat_flush_inodes);
@@ -1824,6 +1944,8 @@ static int __init init_fat_fs(void)
 {
 	int err;
 
+	printk(KERN_NOTICE "SEFT: init_fat_fs\n");
+
 	err = fat_cache_init();
 	if (err)
 		return err;
@@ -1841,6 +1963,7 @@ failed:
 
 static void __exit exit_fat_fs(void)
 {
+	printk(KERN_NOTICE "SEFT: exit_fat_fs\n");
 	fat_cache_destroy();
 	fat_destroy_inodecache();
 }
diff --git a/fs/fat/misc.c b/fs/fat/misc.c
index c4589e9..77aa394 100644
--- a/fs/fat/misc.c
+++ b/fs/fat/misc.c
@@ -87,8 +87,8 @@ int fat_clusters_flush(struct super_block *sb)
 			fsinfo->next_cluster = cpu_to_le32(sbi->prev_free);
 		mark_buffer_dirty(bh);
 	}
-	brelse(bh);
 
+	brelse(bh);
 	return 0;
 }
 
@@ -111,8 +111,10 @@ int fat_chain_add(struct inode *inode, int new_dclus, int nr_cluster)
 		int fclus, dclus;
 
 		ret = fat_get_cluster(inode, FAT_ENT_EOF, &fclus, &dclus);
-		if (ret < 0)
+		if (ret < 0) {
 			return ret;
+		}
+
 		new_fclus = fclus + 1;
 		last = dclus;
 	}
@@ -128,8 +130,10 @@ int fat_chain_add(struct inode *inode, int new_dclus, int nr_cluster)
```

```
                        ret = fat_ent_write(inode, &fatent, new_dclus, wait);
                        fatent_brelse(&fatent);
                }
-               if (ret < 0)
+
+               if (ret < 0) {
                        return ret;
+               }
                /*
                 * FIXME:Although we can add this cache, fat_cache_add() is
                 * assuming to be called after linear search with fat_cache_id.
@@ -144,8 +148,9 @@ int fat_chain_add(struct inode *inode, int new_dclus, int
nr_cluster)
                 */
                if (S_ISDIR(inode->i_mode) && IS_DIRSYNC(inode)) {
                        ret = fat_sync_inode(inode);
-                       if (ret)
+                       if (ret) {
                                return ret;
+                       }
                } else
                        mark_inode_dirty(inode);
        }
@@ -155,6 +160,7 @@ int fat_chain_add(struct inode *inode, int new_dclus, int
nr_cluster)
                            (llu)(inode->i_blocks >> (sbi->cluster_bits - 9)));
                fat_cache_inval_inode(inode);
        }
+
        inode->i_blocks += nr_cluster << (sbi->cluster_bits - 9);

        return 0;
diff --git a/fs/fat/namei_msdos.c b/fs/fat/namei_msdos.c
index b7e2b33..f31e9c83 100644
--- a/fs/fat/namei_msdos.c
+++ b/fs/fat/namei_msdos.c
@@ -137,6 +137,7 @@ static int msdos_find(struct inode *dir, const unsigned char
*name, int len,
                if (err)
                        brelse(sinfo->bh);
        }
+
        return err;
 }

@@ -218,6 +219,7 @@ static struct dentry *msdos_lookup(struct inode *dir, struct
dentry *dentry,
                inode = ERR_PTR(err);
        }
        mutex_unlock(&MSDOS_SB(sb)->s_lock);
+
        return d_splice_alias(inode, dentry);
 }

@@ -250,10 +252,11 @@ static int msdos_add_entry(struct inode *dir, const
unsigned char *name,
                return err;

        dir->i_ctime = dir->i_mtime = *ts;
-       if (IS_DIRSYNC(dir))
+       if (IS_DIRSYNC(dir)) {
                (void)fat_sync_inode(dir);
-       else
-               mark_inode_dirty(dir);
+       } else {
+              mark_inode_dirty(dir);
+       }

        return 0;
 }
@@ -287,6 +290,7 @@ static int msdos_create(struct inode *dir, struct dentry
*dentry, umode_t mode,
        err = msdos_add_entry(dir, msdos_name, 0, is_hid, 0, &ts, &sinfo);
        if (err)
                goto out;
+
        inode = fat_build_inode(sb, sinfo.de, sinfo.i_pos);
        brelse(sinfo.bh);
        if (IS_ERR(inode)) {
@@ -299,8 +303,10 @@ static int msdos_create(struct inode *dir, struct dentry
*dentry, umode_t mode,
        d_instantiate(dentry, inode);
 out:
        mutex_unlock(&MSDOS_SB(sb)->s_lock);
+
        if (!err)
                err = fat_flush_inodes(sb, dir, inode);
+
        return err;
 }

@@ -647,6 +653,7 @@ static void setup(struct super_block *sb)

 static int msdos_fill_super(struct super_block *sb, void *data, int silent)
 {
+       // Super block not filled yet
        return fat_fill_super(sb, data, silent, 0, setup);
 }

@@ -654,6 +661,7 @@ static struct dentry *msdos_mount(struct file_system_type
*fs_type,
                        int flags, const char *dev_name,
                        void *data)
 {
```

```
+       // Don't have the suber block yet
        return mount_bdev(fs_type, flags, dev_name, data, msdos_fill_super);
 }

diff --git a/fs/fhandle.c b/fs/fhandle.c
index d59712d..ccb3c9f 100644
--- a/fs/fhandle.c
+++ b/fs/fhandle.c
@@ -26,8 +26,9 @@ static long do_sys_name_to_handle(struct path *path,
         * support decoding of the file handle
         */
        if (!path->dentry->d_sb->s_export_op ||
-           !path->dentry->d_sb->s_export_op->fh_to_dentry)
+           !path->dentry->d_sb->s_export_op->fh_to_dentry) {
                return -EOPNOTSUPP;
+       }

        if (copy_from_user(&f_handle, ufh, sizeof(struct file_handle)))
                return -EFAULT;
diff --git a/fs/inode.c b/fs/inode.c
index 78a17b8..ad973d3 100644
--- a/fs/inode.c
+++ b/fs/inode.c
@@ -207,6 +207,7 @@ static struct inode *alloc_inode(struct super_block *sb)
                        inode->i_sb->s_op->destroy_inode(inode);
                else
                        kmem_cache_free(inode_cachep, inode);
+
                return NULL;
        }

diff --git a/fs/namei.c b/fs/namei.c
index 33e9495..03dfcb2 100644
--- a/fs/namei.c
+++ b/fs/namei.c
@@ -2660,9 +2660,11 @@ int vfs_create(struct inode *dir, struct dentry *dentry,
umode_t mode,
        error = security_inode_create(dir, dentry, mode);
        if (error)
                return error;
+
        error = dir->i_op->create(dir, dentry, mode, want_excl);
        if (!error)
                fsnotify_create(dir, dentry);
+
        return error;
 }
 EXPORT_SYMBOL(vfs_create);
@@ -2999,6 +3001,7 @@ static int lookup_open(struct nameidata *nd, struct path
*path,
                error = security_path_mknod(&nd->path, dentry, mode, 0);
                if (error)
```

```
                        goto out_dput;
+
                error = vfs_create(dir->d_inode, dentry, mode,
                                nd->flags & LOOKUP_EXCL);
                if (error)
diff --git a/fs/seft.c b/fs/seft.c
new file mode 100644
index 0000000..7e8c0da
--- /dev/null
+++ b/fs/seft.c
@@ -0,0 +1,841 @@
+/*
+ * fs/seft.c - SCM Extension for FAT filesystem code
+ * Copyright (c) 2016
+ * Author: Ray Robles <ray.c.robles@gmail.com>
+ *   File contents taken from dax.c
+ *   - Author: Matthew Wilcox <matthew.r.wilcox@intel.com>
+ *   - Author: Ross Zwisler <ross.zwisler@linux.intel.com>
+ *
+ * This program is free software; you can redistribute it and/or modify it
+ * under the terms and conditions of the GNU General Public License,
+ * version 2, as published by the Free Software Foundation.
+ *
+ * This program is distributed in the hope it will be useful, but WITHOUT
+ * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
+ * FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License for
+ * more details.
+ */
+
+#include <linux/atomic.h>
+#include <linux/blkdev.h>
+#include <linux/buffer_head.h>
+#include <linux/seft.h>
+#include <linux/fs.h>
+#include <linux/genhd.h>
+#include <linux/highmem.h>
+#include <linux/memcontrol.h>
+#include <linux/mm.h>
+#include <linux/mutex.h>
+#include <linux/sched.h>
+#include <linux/uio.h>
+#include <linux/vmstat.h>
+
+int seft_clear_blocks(struct inode *inode, sector_t block, long size)
+{
+       struct block_device *bdev = inode->i_sb->s_bdev;
+       sector_t sector = block << (inode->i_blkbits - 9);
+
+       might_sleep();
+       do {
+               void *addr;
+               unsigned long pfn;
+               long count;
+
+               count = bdev_direct_access(bdev, sector, &addr, &pfn, size);
+
+               if (count < 0)
+                       return count;
+
+               BUG_ON(size < count);
+
+               while (count > 0) {
+                       unsigned pgsz = PAGE_SIZE - offset_in_page(addr);
+
+                       if (pgsz > count) {
+                               pgsz = count;
+                   }
+
+                   if (pgsz < PAGE_SIZE) {
+                          memset(addr, 0, pgsz);
+                   } else {
+                          clear_page(addr);
+                   }
+
+                       addr += pgsz;
+                       size -= pgsz;
+                       count -= pgsz;
+                       BUG_ON(pgsz & 511);
+                       sector += pgsz / 512;
+                       cond_resched();
+               }
+       } while (size);
+
+       return 0;
+}
+EXPORT_SYMBOL_GPL(seft_clear_blocks);
+
+static long seft_get_addr(struct buffer_head *bh, void **addr, unsigned blkbits)
+{
+       unsigned long pfn;
+       sector_t sector = bh->b_blocknr << (blkbits - 9);
+       return bdev_direct_access(bh->b_bdev, sector, addr, &pfn, bh->b_size);
+}
+
+static void seft_new_buf(void *addr, unsigned size, unsigned first, loff_t pos,
+                   loff_t end)
+{
+       loff_t final = end - pos + first; /* The final byte of the buffer */
+
+       if (first > 0)
+               memset(addr, 0, first);
+       if (final < size)
+               memset(addr + final, 0, size - final);
+}
```

```
+
+static bool buffer_written(struct buffer_head *bh)
+{
+       return buffer_mapped(bh) && !buffer_unwritten(bh);
+}
+
+/*
+ * When FAT encounters a hole, it returns without modifying the buffer_head
+ * which means that we can't trust b_size.  To cope with this, we set b_state
+ * to 0 before calling get_block and, if any bit is set, we know we can trust
+ * b_size.  Unfortunate, really, since FAT knows precisely how long a hole is
+ * and would save us time calling get_block repeatedly.
+ */
+static bool buffer_size_valid(struct buffer_head *bh)
+{
+       return bh->b_state != 0;
+}
+
+static ssize_t seft_io(struct inode *inode, struct iov_iter *iter,
+                   loff_t start, loff_t end, get_block_t get_block,
+                   struct buffer_head *bh)
+{
+       ssize_t retval = 0;
+       loff_t pos = start;
+       loff_t max = start;
+       loff_t bh_max = start;
+       void *addr;
+       bool hole = false;
+
+       if (iov_iter_rw(iter) != WRITE)
+               end = min(end, i_size_read(inode));
+
+       while (pos < end) {
+              size_t len;
+               if (pos == max) {
+                       unsigned blkbits = inode->i_blkbits;
+                       long page = pos >> PAGE_SHIFT;
+                       sector_t block = page << (PAGE_SHIFT - blkbits);
+                       unsigned first = pos - (block << blkbits);
+                       long size;
+
+                       if (pos == bh_max) {
+                               bh->b_size = PAGE_ALIGN(end - pos);
+                               bh->b_state = 0;
+
+                          /*
+                           * Calling get_block will call the following functions:
+                           *
+                           *   - fat_get_block()
+                           *   - __fat_get_block()
+                           *   - fat_bmap() ...first time with mmu_private = 0
+                           *   - fat_add_cluster()
+                        *   - fat_alloc_cluster()
+                        *   - fat_chain_add()
+                        *     - investigate how return values of fat_chain_add make
mmu_private = 0x800
+                        *     - for seft io, we need this value to always be 0x1000
(4k)
+                        *   - <then mmu_private gets updated to 0x800 (for the 2k
cluster... 4 blocks)...>
+                        *   - fat_bmap() ...second time with mmu_private = 0x800
+                        *   - fat_bmap_cluster()
+                        *   - fat_get_cluster()
+                        *   - fat_clus_to_blknr()
+                        *   - <*phys and *mapped blocks get updated...>
+                        *
+                        *   bh->b_size = (*mapped_blocks * 512) = 2K (1
cluster)... need to make this 2 clusters
+                        *   bh->b_blocknr = (*phys ) = 0x12x or 0x13x
+                        */
+                               retval = get_block(inode, block, bh, iov_iter_rw(iter) ==
WRITE);
+                               if (retval){
+                                       break;
+                          }
+
+                               if (!buffer_size_valid(bh)) {
+                                       bh->b_size = 1 << blkbits;
+                          }
+
+                               bh_max = pos - first + bh->b_size;
+                       } else {
+                               unsigned done = bh->b_size - (bh_max - (pos - first));
+                               bh->b_blocknr += done >> blkbits;
+                               bh->b_size -= done;
+                       }
+
+                       hole = iov_iter_rw(iter) != WRITE && !buffer_written(bh);
+                       if (hole) {
+                               addr = NULL;
+                               size = bh->b_size - first;
+                       } else {
+                               retval = seft_get_addr(bh, &addr, blkbits);
+                               if (retval < 0) {
+                                       break;
+                          }
+
+                               if (buffer_unwritten(bh) || buffer_new(bh)) {
+                                       seft_new_buf(addr, retval, first, pos, end);
+                               }
+
+                               addr += first;
+                               size = retval - first;
+                       }
```

```
+
+                       max = min(pos + size, end);
+               } else {
+                 printk(KERN_NOTICE "SEFT: seft_io: (pos != max)... no handling -
while loop\n");
+               }
+
+               if (iov_iter_rw(iter) == WRITE) {
+                       printk(KERN_NOTICE "SEFT: seft_io: (iov_iter_rw(iter) == WRITE)
********** while loop\n");
+                       len = copy_from_iter(addr, max - pos, iter);
+               } else if (!hole) {
+                       printk(KERN_NOTICE "SEFT: seft_io: (!hole) 2 ********** while
loop\n");
+                       len = copy_to_iter(addr, max - pos, iter);
+               } else {
+                       printk(KERN_NOTICE "SEFT: seft_io: (READ and hole == TRUE)
********** while loop\n");
+                       len = iov_iter_zero(max - pos, iter);
+               }
+
+               if (!len) {
+                       break;
+               }
+
+               pos += len;
+               addr += len;
+       } /* end while (pos < end) */
+
+       return (pos == start) ? retval : pos - start;
+}
+
+/**
+ * seft_do_io - Perform I/O to a SEFT file
+ * @iocb: The control block for this I/O
+ * @inode: The file which the I/O is directed at
+ * @iter: The addresses to do I/O from or to
+ * @pos: The file offset where the I/O starts
+ * @get_block: The filesystem method used to translate file offsets to blocks
+ * @end_io: A filesystem callback for I/O completion
+ * @flags: See below
+ *
+ * SEFT
+ *
+ * This function uses the same locking scheme as do_blockdev_direct_IO:
+ * If @flags has DIO_LOCKING set, we assume that the i_mutex is held by the
+ * caller for writes.  For reads, we take and release the i_mutex ourselves.
+ * If DIO_LOCKING is not set, the filesystem takes care of its own locking.
+ * As with do_blockdev_direct_IO(), we increment i_dio_count while the I/O
+ * is in progress.
+ */
+ssize_t seft_do_io(struct kiocb *iocb, struct inode *inode,
+                 struct iov_iter *iter, loff_t pos, get_block_t get_block,
+                 dio_iodone_t end_io, int flags)
+{
+	struct buffer_head bh;
+	ssize_t retval = -EINVAL;
+	loff_t end = pos + iov_iter_count(iter);
+
+	memset(&bh, 0, sizeof(bh));
+
+	if ((flags & DIO_LOCKING) && iov_iter_rw(iter) == READ) {
+		struct address_space *mapping = inode->i_mapping;
+		mutex_lock(&inode->i_mutex);
+		retval = filemap_write_and_wait_range(mapping, pos, end - 1);
+		if (retval) {
+			mutex_unlock(&inode->i_mutex);
+			goto out;
+		}
+	}
+
+	/* Protects against truncate */
+	if (!(flags & DIO_SKIP_DIO_COUNT))
+		inode_dio_begin(inode);
+
+	retval = seft_io(inode, iter, pos, end, get_block, &bh);
+
+	if ((flags & DIO_LOCKING) && iov_iter_rw(iter) == READ)
+		mutex_unlock(&inode->i_mutex);
+
+	if ((retval > 0) && end_io) {
+		end_io(iocb, pos, retval, bh.b_private);
+	}
+
+	if (!(flags & DIO_SKIP_DIO_COUNT)) {
+		inode_dio_end(inode);
+	}
+ out:
+	return retval;
+}
+EXPORT_SYMBOL_GPL(seft_do_io);
+
+/*
+ * The user has performed a load from a hole in the file.  Allocating
+ * a new page in the file would cause excessive storage usage for
+ * workloads with sparse files.  We allocate a page cache page instead.
+ * We'll kick it out of the page cache if it's ever written to,
+ * otherwise it will simply fall out of the page cache under memory
+ * pressure without ever having been dirtied.
+ */
+static int seft_load_hole(struct address_space *mapping, struct page *page,
+			struct vm_fault *vmf)
+{
+	unsigned long size;
```

```
+        struct inode *inode = mapping->host;
+
+        if (!page)
+            page = find_or_create_page(mapping, vmf->pgoff,
+                                       GFP_KERNEL | __GFP_ZERO);
+
+        if (!page)
+            return VM_FAULT_OOM;
+
+        /* Recheck i_size under page lock to avoid truncate race */
+        size = (i_size_read(inode) + PAGE_SIZE - 1) >> PAGE_SHIFT;
+        if (vmf->pgoff >= size) {
+            unlock_page(page);
+            page_cache_release(page);
+            return VM_FAULT_SIGBUS;
+        }
+
+        vmf->page = page;
+        return VM_FAULT_LOCKED;
+}
+
+static int copy_user_bh(struct page *to, struct buffer_head *bh,
+                        unsigned blkbits, unsigned long vaddr)
+{
+        void *vfrom;
+        void *vto;
+
+        if (seft_get_addr(bh, &vfrom, blkbits) < 0)
+            return -EIO;
+
+        vto = kmap_atomic(to);
+        copy_user_page(vto, (void __force *)vfrom, vaddr, to);
+        kunmap_atomic(vto);
+        return 0;
+}
+
+static int seft_insert_mapping(struct inode *inode, struct buffer_head *bh,
+                          struct vm_area_struct *vma, struct vm_fault *vmf)
+{
+        struct address_space *mapping = inode->i_mapping;
+        sector_t sector = bh->b_blocknr << (inode->i_blkbits - 9);
+        unsigned long vaddr = (unsigned long)vmf->virtual_address;
+        //void __pmem *addr;
+        void *addr;
+        unsigned long pfn;
+        pgoff_t size;
+        int error;
+
+        i_mmap_lock_read(mapping);
+
+        /*
+         * Check truncate didn't happen while we were allocating a block.
+        * If it did, this block may or may not be still allocated to the
+        * file.  We can't tell the filesystem to free it because we can't
+        * take i_mutex here.  In the worst case, the file still has blocks
+        * allocated past the end of the file.
+        */
+       size = (i_size_read(inode) + PAGE_SIZE - 1) >> PAGE_SHIFT;
+       if (unlikely(vmf->pgoff >= size)) {
+               error = -EIO;
+               goto out;
+       }
+
+       error = bdev_direct_access(bh->b_bdev, sector, &addr, &pfn, bh->b_size);
+       if (error < 0)
+               goto out;
+       if (error < PAGE_SIZE) {
+               error = -EIO;
+               goto out;
+       }
+
+       if (buffer_unwritten(bh) || buffer_new(bh)) {
+               //clear_pmem(addr, PAGE_SIZE);
+               //wmb_pmem();
+             clear_page(addr);
+       }
+
+       error = vm_insert_mixed(vma, vaddr, pfn);
+
+ out:
+       i_mmap_unlock_read(mapping);
+
+       return error;
+}
+
+/**
+ * __seft_fault - handle a page fault on a SEFT file
+ * @vma: The virtual memory area where the fault occurred
+ * @vmf: The description of the fault
+ * @get_block: The filesystem method used to translate file offsets to blocks
+ * @complete_unwritten: The filesystem method used to convert unwritten blocks
+ *     to written so the data written to them is exposed. This is required for
+ *     required by write faults for filesystems that will return unwritten
+ *     extent mappings from @get_block, but it is optional for reads as
+ *      seft_insert_mapping() will always zero unwritten blocks.
+ *      If the fs does not support unwritten extents, then it
+ *      should pass NULL.
+ *
+ * When a page fault occurs, filesystems may call this helper in their
+ * fault handler for SEFT files. __seft_fault() assumes the
+ * caller has done all the necessary locking for the page fault
+ * to proceed successfully.
+ */
+int __seft_fault(struct vm_area_struct *vma, struct vm_fault *vmf,
+		get_block_t get_block, seft_iodone_t complete_unwritten)
+{
+	struct file *file = vma->vm_file;
+	struct address_space *mapping = file->f_mapping;
+	struct inode *inode = mapping->host;
+	struct page *page;
+	struct buffer_head bh;
+	unsigned long vaddr = (unsigned long)vmf->virtual_address;
+	unsigned blkbits = inode->i_blkbits;
+	sector_t block;
+	pgoff_t size;
+	int error;
+	int major = 0;
+
+	size = (i_size_read(inode) + PAGE_SIZE - 1) >> PAGE_SHIFT;
+	if (vmf->pgoff >= size)
+		return VM_FAULT_SIGBUS;
+
+	memset(&bh, 0, sizeof(bh));
+	block = (sector_t)vmf->pgoff << (PAGE_SHIFT - blkbits);
+	bh.b_size = PAGE_SIZE;
+
+ repeat:
+	page = find_get_page(mapping, vmf->pgoff);
+	if (page) {
+		if (!lock_page_or_retry(page, vma->vm_mm, vmf->flags)) {
+			page_cache_release(page);
+			return VM_FAULT_RETRY;
+		}
+		if (unlikely(page->mapping != mapping)) {
+			unlock_page(page);
+			page_cache_release(page);
+			goto repeat;
+		}
+		size = (i_size_read(inode) + PAGE_SIZE - 1) >> PAGE_SHIFT;
+		if (unlikely(vmf->pgoff >= size)) {
+			/*
+			 * We have a struct page covering a hole in the file
+			 * from a read fault and we've raced with a truncate
+			 */
+			error = -EIO;
+			goto unlock_page;
+		}
+	}
+
+	error = get_block(inode, block, &bh, 0);
+	if (!error && (bh.b_size < PAGE_SIZE))
+		error = -EIO;		/* fs corruption? */
+	if (error)
+		goto unlock_page;
+
+	if (!buffer_mapped(&bh) && !buffer_unwritten(&bh) && !vmf->cow_page) {
```

```
+		if (vmf->flags & FAULT_FLAG_WRITE) {
+			error = get_block(inode, block, &bh, 1);
+			count_vm_event(PGMAJFAULT);
+			mem_cgroup_count_vm_event(vma->vm_mm, PGMAJFAULT);
+			major = VM_FAULT_MAJOR;
+			if (!error && (bh.b_size < PAGE_SIZE))
+				error = -EIO;
+			if (error)
+				goto unlock_page;
+		} else {
+			return seft_load_hole(mapping, page, vmf);
+		}
+	}
+
+	if (vmf->cow_page) {
+		struct page *new_page = vmf->cow_page;
+		if (buffer_written(&bh))
+			error = copy_user_bh(new_page, &bh, blkbits, vaddr);
+		else
+			clear_user_highpage(new_page, vaddr);
+		if (error)
+			goto unlock_page;
+		vmf->page = page;
+		if (!page) {
+			i_mmap_lock_read(mapping);
+			/* Check we didn't race with truncate */
+			size = (i_size_read(inode) + PAGE_SIZE - 1) >>
+			PAGE_SHIFT;
+			if (vmf->pgoff >= size) {
+				i_mmap_unlock_read(mapping);
+				error = -EIO;
+				goto out;
+			}
+		}
+		return VM_FAULT_LOCKED;
+	}
+
+	/* Check we didn't race with a read fault installing a new page */
+	if (!page && major)
+		page = find_lock_page(mapping, vmf->pgoff);
+
+	if (page) {
+		unmap_mapping_range(mapping, vmf->pgoff << PAGE_SHIFT,
+					PAGE_CACHE_SIZE, 0);
+		delete_from_page_cache(page);
+		unlock_page(page);
+		page_cache_release(page);
+	}
+
+	/*
+	 * If we successfully insert the new mapping over an unwritten extent,
+	 * we need to ensure we convert the unwritten extent. If there is an
```

```
+	 * error inserting the mapping, the filesystem needs to leave it as
+	 * unwritten to prevent exposure of the stale underlying data to
+	 * userspace, but we still need to call the completion function so
+	 * the private resources on the mapping buffer can be released. We
+	 * indicate what the callback should do via the uptodate variable, same
+	 * as for normal BH based IO completions.
+	 */
+	error = seft_insert_mapping(inode, &bh, vma, vmf);
+	if (buffer_unwritten(&bh)) {
+		if (complete_unwritten)
+			complete_unwritten(&bh, !error);
+		else
+			WARN_ON_ONCE(!(vmf->flags & FAULT_FLAG_WRITE));
+	}
+
+ out:
+	if (error == -ENOMEM)
+		return VM_FAULT_OOM | major;
+	/* -EBUSY is fine, somebody else faulted on the same PTE */
+	if ((error < 0) && (error != -EBUSY))
+		return VM_FAULT_SIGBUS | major;
+	return VM_FAULT_NOPAGE | major;
+
+ unlock_page:
+	if (page) {
+		unlock_page(page);
+		page_cache_release(page);
+	}
+	goto out;
+}
+EXPORT_SYMBOL_GPL(__seft_fault);
+
+/**
+ * sef_fault - handle a page fault on a SEFT file
+ * @vma: The virtual memory area where the fault occurred
+ * @vmf: The description of the fault
+ * @get_block: The filesystem method used to translate file offsets to blocks
+ *
+ * When a page fault occurs, filesystems may call this helper in their
+ * fault handler for SEFT files.
+ */
+int seft_fault(struct vm_area_struct *vma, struct vm_fault *vmf,
+	      get_block_t get_block, seft_iodone_t complete_unwritten)
+{
+	int result;
+	struct super_block *sb = file_inode(vma->vm_file)->i_sb;
+
+	if (vmf->flags & FAULT_FLAG_WRITE) {
+		sb_start_pagefault(sb);
+		file_update_time(vma->vm_file);
+	}
+	result = __seft_fault(vma, vmf, get_block, complete_unwritten);
```

```
+	if (vmf->flags & FAULT_FLAG_WRITE)
+		sb_end_pagefault(sb);
+
+	return result;
+}
+EXPORT_SYMBOL_GPL(seft_fault);
+
+
+//#if 0
+#if 1
+
+#ifdef CONFIG_TRANSPARENT_HUGEPAGE
+/*
+ * The 'colour' (ie low bits) within a PMD of a page offset.  This comes up
+ * more often than one might expect in the below function.
+ */
+#define PG_PMD_COLOUR ((PMD_SIZE >> PAGE_SHIFT) - 1)
+
+int __seft_pmd_fault(struct vm_area_struct *vma, unsigned long address,
+			pmd_t *pmd, unsigned int flags, get_block_t get_block,
+			seft_iodone_t complete_unwritten)
+{
+	struct file *file = vma->vm_file;
+	struct address_space *mapping = file->f_mapping;
+	struct inode *inode = mapping->host;
+	struct buffer_head bh;
+	unsigned blkbits = inode->i_blkbits;
+	unsigned long pmd_addr = address & PMD_MASK;
+	bool write = flags & FAULT_FLAG_WRITE;
+	long length;
+	//void __pmem *kaddr;
+	void *kaddr;
+	pgoff_t size, pgoff;
+	sector_t block, sector;
+	unsigned long pfn;
+	int result = 0;
+
+	/* Fall back to PTEs if we're going to COW */
+	if (write && !(vma->vm_flags & VM_SHARED))
+		return VM_FAULT_FALLBACK;
+
+	/* If the PMD would extend outside the VMA */
+	if (pmd_addr < vma->vm_start)
+		return VM_FAULT_FALLBACK;
+
+	if ((pmd_addr + PMD_SIZE) > vma->vm_end)
+		return VM_FAULT_FALLBACK;
+
+	pgoff = linear_page_index(vma, pmd_addr);
+	size = (i_size_read(inode) + PAGE_SIZE - 1) >> PAGE_SHIFT;
+	if (pgoff >= size)
+		return VM_FAULT_SIGBUS;
```

```
+
+       /* If the PMD would cover blocks out of the file */
+       if ((pgoff | PG_PMD_COLOUR) >= size)
+               return VM_FAULT_FALLBACK;
+
+       memset(&bh, 0, sizeof(bh));
+       block = (sector_t)pgoff << (PAGE_SHIFT - blkbits);
+
+       bh.b_size = PMD_SIZE;
+       length = get_block(inode, block, &bh, write);
+       if (length)
+               return VM_FAULT_SIGBUS;
+
+       i_mmap_lock_read(mapping);
+
+       /*
+        * If the filesystem isn't willing to tell us the length of a hole,
+        * just fall back to PTEs.  Calling get_block 512 times in a loop
+        * would be silly.
+        */
+       if (!buffer_size_valid(&bh) || bh.b_size < PMD_SIZE)
+               goto fallback;
+
+       /*
+        * If we allocated new storage, make sure no process has any
+        * zero pages covering this hole
+        */
+       if (buffer_new(&bh)) {
+               i_mmap_unlock_read(mapping);
+               unmap_mapping_range(mapping, pgoff << PAGE_SHIFT, PMD_SIZE,
0);
+               i_mmap_lock_read(mapping);
+       }
+
+       /*
+        * If a truncate happened while we were allocating blocks, we may
+        * leave blocks allocated to the file that are beyond EOF.  We can't
+        * take i_mutex here, so just leave them hanging; they'll be freed
+        * when the file is deleted.
+        */
+       size = (i_size_read(inode) + PAGE_SIZE - 1) >> PAGE_SHIFT;
+       if (pgoff >= size) {
+               result = VM_FAULT_SIGBUS;
+               goto out;
+       }
+
+       if ((pgoff | PG_PMD_COLOUR) >= size)
+               goto fallback;
+
+       if (!write && !buffer_mapped(&bh) && buffer_uptodate(&bh)) {
+               spinlock_t *ptl;
+               pmd_t entry;
+		struct page *zero_page = get_huge_zero_page();
+
+		if (unlikely(!zero_page))
+			goto fallback;
+
+		ptl = pmd_lock(vma->vm_mm, pmd);
+		if (!pmd_none(*pmd)) {
+			spin_unlock(ptl);
+			goto fallback;
+		}
+
+		entry = mk_pmd(zero_page, vma->vm_page_prot);
+		entry = pmd_mkhuge(entry);
+		set_pmd_at(vma->vm_mm, pmd_addr, pmd, entry);
+		result = VM_FAULT_NOPAGE;
+		spin_unlock(ptl);
+	} else {
+		sector = bh.b_blocknr << (blkbits - 9);
+		length = bdev_direct_access(bh.b_bdev, sector, &kaddr, &pfn,
+						bh.b_size);
+		if (length < 0) {
+			result = VM_FAULT_SIGBUS;
+			goto out;
+		}
+
+		if ((length < PMD_SIZE) || (pfn & PG_PMD_COLOUR))
+			goto fallback;
+
+		if (buffer_unwritten(&bh) || buffer_new(&bh)) {
+			int i;
+			for (i = 0; i < PTRS_PER_PMD; i++) {
+			    //clear_pmem(kaddr + i * PAGE_SIZE, PAGE_SIZE);
+				//clear_pages(kaddr + i * PAGE_SIZE);
+				clear_page(kaddr + i * PAGE_SIZE);
+		     }
+
+			//wmb_pmem();
+			count_vm_event(PGMAJFAULT);
+			mem_cgroup_count_vm_event(vma->vm_mm, PGMAJFAULT);
+			result |= VM_FAULT_MAJOR;
+		}
+
+		result |= vmf_insert_pfn_pmd(vma, address, pmd, pfn, write);
+	}
+
+out:
+	i_mmap_unlock_read(mapping);
+
+	if (buffer_unwritten(&bh))
+		complete_unwritten(&bh, !(result & VM_FAULT_ERROR));
+
+	return result;
```

```
+
+fallback:
+       count_vm_event(THP_FAULT_FALLBACK);
+       result = VM_FAULT_FALLBACK;
+       goto out;
+}
+EXPORT_SYMBOL_GPL(__seft_pmd_fault);
+
+/**
+ * seft_pmd_fault - handle a PMD fault on a seft file
+ * @vma: The virtual memory area where the fault occurred
+ * @vmf: The description of the fault
+ * @get_block: The filesystem method used to translate file offsets to blocks
+ *
+ * When a page fault occurs, filesystems may call this helper in their
+ * pmd_fault handler for SEFT files.
+ */
+int seft_pmd_fault(struct vm_area_struct *vma, unsigned long address,
+                 pmd_t *pmd, unsigned int flags, get_block_t get_block,
+                 seft_iodone_t complete_unwritten)
+{
+       int result;
+       struct super_block *sb = file_inode(vma->vm_file)->i_sb;
+
+       if (flags & FAULT_FLAG_WRITE) {
+               sb_start_pagefault(sb);
+               file_update_time(vma->vm_file);
+       }
+
+       result = __seft_pmd_fault(vma, address, pmd, flags, get_block,
complete_unwritten);
+       if (flags & FAULT_FLAG_WRITE)
+               sb_end_pagefault(sb);
+
+       return result;
+}
+EXPORT_SYMBOL_GPL(seft_pmd_fault);
+#endif /* CONFIG_TRANSPARENT_HUGEPAGE */
+
+/**
+ * seft_pfn_mkwrite - handle first write to DAX page
+ * @vma: The virtual memory area where the fault occurred
+ * @vmf: The description of the fault
+ *
+ */
+int seft_pfn_mkwrite(struct vm_area_struct *vma, struct vm_fault *vmf)
+{
+       struct super_block *sb = file_inode(vma->vm_file)->i_sb;
+
+       sb_start_pagefault(sb);
+       file_update_time(vma->vm_file);
+       sb_end_pagefault(sb);
```

```
+
+	return VM_FAULT_NOPAGE;
+}
+EXPORT_SYMBOL_GPL(seft_pfn_mkwrite);
+#endif
+
+/**
+ * seft_zero_page_range - zero a range within a page of a SEFT
+ * file
+ * @inode: The file being truncated
+ * @from: The file offset that is being truncated to
+ * @length: The number of bytes to zero
+ * @get_block: The filesystem method used to translate file offsets to blocks
+ *
+ * This function can be called by a filesystem when it is zeroing part of a
+ * page in a SEFT file.  This is intended for hole-punch
+ * operations. If you are truncating a file, the helper function
+ * seft_truncate_page() may be more convenient.
+ *
+ * We work in terms of PAGE_CACHE_SIZE here for commonality with
+ * block_truncate_page(), but we could go down to PAGE_SIZE if the filesystem
+ * took care of disposing of the unnecessary blocks.  Even if the filesystem
+ * block size is smaller than PAGE_SIZE, we have to zero the rest of the page
+ * since the file might be mmapped.
+ */
+int seft_zero_page_range(struct inode *inode, loff_t from, unsigned length,
+							get_block_t get_block)
+{
+	struct buffer_head bh;
+	pgoff_t index = from >> PAGE_CACHE_SHIFT;
+	unsigned offset = from & (PAGE_CACHE_SIZE-1);
+	int err;
+
+	/* Block boundary? Nothing to do */
+	if (!length)
+		return 0;
+
+	BUG_ON((offset + length) > PAGE_CACHE_SIZE);
+
+	memset(&bh, 0, sizeof(bh));
+	bh.b_size = PAGE_CACHE_SIZE;
+	err = get_block(inode, index, &bh, 0);
+	if (err < 0)
+		return err;
+	if (buffer_written(&bh)) {
+		//void __pmem *addr;
+		void *addr;
+		err = seft_get_addr(&bh, &addr, inode->i_blkbits);
+		if (err < 0)
+			return err;
+
+		clear_page(addr + offset);
```

```
+                //clear_pmem(addr + offset, length);
+                //wmb_pmem();
+        }
+
+        return 0;
+}
+EXPORT_SYMBOL_GPL(seft_zero_page_range);
+
+/**
+ * seft_truncate_page - handle a partial page being truncated in
+ * a SEFT file
+ * @inode: The file being truncated
+ * @from: The file offset that is being truncated to
+ * @get_block: The filesystem method used to translate file offsets to blocks
+ *
+ * Similar to block_truncate_page(), this function can be called by a
+ * filesystem when it is truncating a SEFT file to handle the
+ * partial page.
+ *
+ * We work in terms of PAGE_CACHE_SIZE here for commonality with
+ * block_truncate_page(), but we could go down to PAGE_SIZE if the filesystem
+ * took care of disposing of the unnecessary blocks.  Even if the filesystem
+ * block size is smaller than PAGE_SIZE, we have to zero the rest of the page
+ * since the file might be mmapped.
+ */
+int seft_truncate_page(struct inode *inode, loff_t from, get_block_t get_block)
+{
+        unsigned length = PAGE_CACHE_ALIGN(from) - from;
+        return seft_zero_page_range(inode, from, length, get_block);
+}
+EXPORT_SYMBOL_GPL(seft_truncate_page);
diff --git a/fs/super.c b/fs/super.c
index 954aeb8..ae45449 100644
--- a/fs/super.c
+++ b/fs/super.c
@@ -970,6 +970,8 @@ struct dentry *mount_bdev(struct file_system_type
*fs_type,
         fmode_t mode = FMODE_READ | FMODE_EXCL;
         int error = 0;

+        printk("mount_bdev: entering");
+
         if (!(flags & MS_RDONLY))
                 mode |= FMODE_WRITE;

diff --git a/include/linux/fs.h b/include/linux/fs.h
index 72d8a84..6525c3b 100644
--- a/include/linux/fs.h
+++ b/include/linux/fs.h
@@ -72,6 +72,7 @@ typedef int (get_block_t)(struct inode *inode, sector_t iblock,
 typedef void (dio_iodone_t)(struct kiocb *iocb, loff_t offset,
                         ssize_t bytes, void *private);
 typedef void (dax_iodone_t)(struct buffer_head *bh_map, int uptodate);
+typedef void (seft_iodone_t)(struct buffer_head *bh_map, int uptodate);

 #define MAY_EXEC           0x00000001
 #define MAY_WRITE          0x00000002
@@ -1751,6 +1752,11 @@ struct super_operations {
 #else
 #define S_DAX              0       /* Make all the DAX code disappear */
 #endif
+#ifdef CONFIG_FS_SEFT
+#define S_SEFT          16384   /* SCM Extensions for FAT... no page cache */
+#else
+#define S_SEFT          0       /* Disable SEFT */
+#endif

 /*
  * Note that nosuid etc flags are inode-specific: setting some file-system
@@ -1789,6 +1795,7 @@ struct super_operations {
 #define IS_AUTOMOUNT(inode)    ((inode)->i_flags & S_AUTOMOUNT)
 #define IS_NOSEC(inode)        ((inode)->i_flags & S_NOSEC)
 #define IS_DAX(inode)          ((inode)->i_flags & S_DAX)
+#define IS_SEFT(inode)         ((inode)->i_flags & S_SEFT)

 #define IS_WHITEOUT(inode)     (S_ISCHR(inode->i_mode) && \
                               (inode)->i_rdev == WHITEOUT_DEV)
@@ -2867,7 +2874,10 @@ extern void replace_mount_options(struct super_block
*sb, char *options);

 static inline bool io_is_direct(struct file *filp)
 {
-       return (filp->f_flags & O_DIRECT) || IS_DAX(file_inode(filp));
+       //return (filp->f_flags & O_DIRECT) || IS_DAX(file_inode(filp));
+       return (filp->f_flags & O_DIRECT)   ||
+               IS_DAX(file_inode(filp))    ||
+               IS_SEFT(file_inode(filp));
 }

 static inline int iocb_flags(struct file *file)
diff --git a/include/linux/seft.h b/include/linux/seft.h
new file mode 100644
index 0000000..2eae65b
--- /dev/null
+++ b/include/linux/seft.h
@@ -0,0 +1,46 @@
+#ifndef _LINUX_SEFT_H
+#define _LINUX_SEFT_H
+
+#include <linux/fs.h>
+#include <linux/mm.h>
+#include <asm/pgtable.h>
+
+ssize_t seft_do_io(struct kiocb *iocb, struct inode *inode,
+                   struct iov_iter *iter, loff_t pos, get_block_t get_block,
+                   dio_iodone_t end_io, int flags);
+int seft_clear_blocks(struct inode *, sector_t block, long size);
+int seft_fault(struct vm_area_struct *vma, struct vm_fault *vmf,
+               get_block_t get_block, seft_iodone_t complete_unwritten);
+int __seft_fault(struct vm_area_struct *vma, struct vm_fault *vmf,
+                 get_block_t get_block, seft_iodone_t complete_unwritten);
+
+#define seft_mkwrite(vma, vmf, gb, iod)           seft_fault(vma, vmf, gb, iod)
+#define __seft_mkwrite(vma, vmf, gb, iod)         __seft_fault(vma, vmf, gb, iod)
+
+int seft_zero_page_range(struct inode *, loff_t from, unsigned len, get_block_t);
+int seft_truncate_page(struct inode *, loff_t from, get_block_t);
+
+#ifdef CONFIG_TRANSPARENT_HUGEPAGE
+int seft_pmd_fault(struct vm_area_struct *, unsigned long addr, pmd_t *,
+                   unsigned int flags, get_block_t, seft_iodone_t);
+int __seft_pmd_fault(struct vm_area_struct *, unsigned long addr, pmd_t *,
+                     unsigned int flags, get_block_t, seft_iodone_t);
+#else
+static inline int seft_pmd_fault(struct vm_area_struct *vma, unsigned long addr,
+                                 pmd_t *pmd, unsigned int flags, get_block_t gb,
+                                 seft_iodone_t di)
+{
+        return VM_FAULT_FALLBACK;
+}
+#define __seft_pmd_fault seft_pmd_fault
+#endif
+
+int seft_pfn_mkwrite(struct vm_area_struct *vma, struct vm_fault *vmf);
+
+static inline bool vma_is_seft(struct vm_area_struct *vma)
+{
+        printk(KERN_NOTICE "SEFT: vma_is_seft: entering");
+        return vma->vm_file && IS_SEFT(vma->vm_file->f_mapping->host);
+}
+
+#endif
diff --git a/mm/filemap.c b/mm/filemap.c
index 327910c..c9e032d 100644
--- a/mm/filemap.c
+++ b/mm/filemap.c
@@ -453,6 +453,7 @@ int filemap_write_and_wait_range(struct address_space
*mapping,
         } else {
                 err = filemap_check_errors(mapping);
         }
+
         return err;
 }
 EXPORT_SYMBOL(filemap_write_and_wait_range);
```

```
@@ -1719,7 +1720,7 @@ generic_file_read_iter(struct kiocb *iocb, struct iov_iter
*iter)
        loff_t *ppos = &iocb->ki_pos;
        loff_t pos = *ppos;

-       if (iocb->ki_flags & IOCB_DIRECT) {
+       if (iocb->ki_flags & IOCB_DIRECT) {
                struct address_space *mapping = file->f_mapping;
                struct inode *inode = mapping->host;
                size_t count = iov_iter_count(iter);
@@ -1727,9 +1728,11 @@ generic_file_read_iter(struct kiocb *iocb, struct iov_iter
*iter)

                if (!count)
                        goto out; /* skip atime */
+
                size = i_size_read(inode);
                retval = filemap_write_and_wait_range(mapping, pos,
                                        pos + count - 1);
+
                if (!retval) {
                        struct iov_iter data = *iter;
                        retval = mapping->a_ops->direct_IO(iocb, &data, pos);
@@ -1747,11 +1750,12 @@ generic_file_read_iter(struct kiocb *iocb, struct iov_iter
*iter)
                 * there was a short read because we hit EOF, go ahead
                 * and return.  Otherwise fallthrough to buffered io for
                 * the rest of the read.  Buffered reads will not work for
-                * DAX files, so don't bother trying.
+                * SEFT files, so don't bother trying.
                 */
                if (retval < 0 || !iov_iter_count(iter) || *ppos >= size ||
-                   IS_DAX(inode)) {
+                 IS_DAX(inode) || IS_SEFT(inode)) {
                        file_accessed(file);
+                         printk(KERN_NOTICE "SEFT: generic_file_read_iter: do not
perform buffered read\n");
                        goto out;
                }
        }
@@ -2395,6 +2399,7 @@ generic_file_direct_write(struct kiocb *iocb, struct
iov_iter *from, loff_t pos)
        }

        data = *from;
+       printk(KERN_NOTICE "SEFT: generic_file_direct_write: calling mapping-
>a_ops->direct_IO\n");
        written = mapping->a_ops->direct_IO(iocb, &data, pos);

        /*
@@ -2418,7 +2423,9 @@ generic_file_direct_write(struct kiocb *iocb, struct
iov_iter *from, loff_t pos)
```

```
                        mark_inode_dirty(inode);
                }
                iocb->ki_pos = pos;
-       }
+       } else {
+           printk(KERN_NOTICE "SEFT: generic_file_read_iter: written <= 0 after
.direct_IO (0x%zx)\n", written);
+       }
 out:
        return written;
 }
@@ -2495,6 +2502,7 @@ again:

                status = a_ops->write_begin(file, mapping, pos, bytes, flags,
                                                &page, &fsdata);
+
                if (unlikely(status < 0))
                        break;

@@ -2576,15 +2584,18 @@ ssize_t __generic_file_write_iter(struct kiocb *iocb,
struct iov_iter *from)
                loff_t pos, endbyte;

                written = generic_file_direct_write(iocb, from, iocb->ki_pos);
+
                /*
                 * If the write stopped short of completing, fall back to
                 * buffered writes.  Some filesystems do this for writes to
-                * holes, for example.  For DAX files, a buffered write will
-                * not succeed (even if it did, DAX does not handle dirty
+                * holes, for example.  For SEFT files, a buffered write will
+                * not succeed (even if it did, SEFT does not handle dirty
                 * page-cache pages correctly).
                 */
-               if (written < 0 || !iov_iter_count(from) || IS_DAX(inode))
+               if (written < 0 || !iov_iter_count(from) || IS_DAX(inode) ||
IS_SEFT(inode)) {
+                   printk(KERN_NOTICE "SEFT: __generic_file_write_iter: do not
performed buffered write\n");
                        goto out;
+              }

                status = generic_perform_write(file, from, pos = iocb->ki_pos);
                /*
diff --git a/mm/huge_memory.c b/mm/huge_memory.c
index bbac913..4c9ff01 100644
--- a/mm/huge_memory.c
+++ b/mm/huge_memory.c
@@ -17,6 +17,9 @@
 #include <linux/shrinker.h>
 #include <linux/mm_inline.h>
 #include <linux/dax.h>
```

```
+
+#include <linux/seft.h>
+
 #include <linux/kthread.h>
 #include <linux/khugepaged.h>
 #include <linux/freezer.h>
diff --git a/mm/page_io.c b/mm/page_io.c
index b995a5b..6d6adeb 100644
--- a/mm/page_io.c
+++ b/mm/page_io.c
@@ -239,6 +239,7 @@ int swap_writepage(struct page *page, struct
writeback_control *wbc)
                 end_page_writeback(page);
                 goto out;
         }
+
         ret = __swap_writepage(page, wbc, end_swap_bio_write);
 out:
         return ret;
diff --git a/mm/swapfile.c b/mm/swapfile.c
index 5887731..7e82a5e 100644
--- a/mm/swapfile.c
+++ b/mm/swapfile.c
@@ -151,6 +151,7 @@ static int discard_swap(struct swap_info_struct *si)

                 cond_resched();
         }
+
         return err;             /* That will often be -EOPNOTSUPP */
```