Υλοποίηση Συστήματος Αρχείων με Καταγραφή Ενημερώσεων για Αποθηκευτικές Συσκευές Ασύμμετρης Προσπέλασης

Η ΜΕΤΑΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

υποβάλλεται στην

ορισθείσα από την Γενική Συνέλευση Ειδικής Σύνθεσης

Τμήματος Μηχανικών Η/Υ και Πληροφορικής
Εξεταστική Επιτροπή

από τον

Βασίλειο Παπαδόπουλο

ως μέρος των Υποχρεώσεων για τη λήψη του

ΜΕΤΑΠΤΥΧΙΑΚΟΥ ΔΙΠΛΩΜΑΤΟΣ ΣΤΗΝ ΠΛΗΡΟΦΟΡΙΚΗ

ΜΕ ΕΞΕΙΔΙΚΕΥΣΗ

ΣΤΑ ΥΠΟΛΟΓΙΣΤΙΚΑ ΣΥΣΤΗΜΑΤΑ

Ιούνιος 2015

Dedication

This is the beginning.

Table of Contents

List of Figures

List of Tables

List of Listings

Abstract

Papadopoulos, Vasileios, S.
MSc, Computer Science and Engineering Department, University of Ioannina, Greece.
June 2015
Implementation of a Journaling File System for Asymmetric Storage Devices
Thesis Supervisor: Stergios V. Anastasiadis

Asymmetric storage devices are increasingly offering competitive advantages as either standalone storage devices or as a distinct layer in the storage hierarchy. Compatibility with legacy systems is possible through a typical block interface offered by a special firmware running on the device's controller. By utilizing application semantical information, the file system running at the top is able to improve the performance of the device and exploit the intrinsic characteristics of asymmetric storage to its advantage.

More specifically, Log-Structured file systems offer increased performance and reduced latency compared to traditional file systems. In many cases however, they do not manage the available space effectively and as a result are hurdled by garbage collection overheads. In addition, the amount of data that can be recovered in case of a system crash depends on the pressure that was put into the system when the crash occurred and can be very limited.

In order to solve these problems we propose Metis, a composite file system that combines journaling with the log-structured file system in order to improve the garbage collection process, reduce write traffic and increase the utilization of the available storage space. We implemented our system in the Linux kernel. By using a set of tools that evaluate our system across different workloads, we are able to show that our implementation increases the utilization of the available storage space while also improving the recovery point objective of the system while its performance remains comparable to that of the

original implementation.

Εκτεταμενη Περιληψη στα Ελληνικα

Βασίλειος Παπαδόπουλος του Σωκράτη και της Γεωργίας.
MSc, Τμήμα Μηχανικών Η/Υ και Πληροφορικής, Πανεπιστήμιο Ιωαννίνων, Ιούνιος 2015.
Υλοποίηση Συστήματος Αρχείων με Καταγραφή Ενημερώσεων για Αποθηκευτικές Συσκευές Ασύμμετρης Προσπέλασης.
Επιβλέπων: Στέργιος Β. Αναστασιάδης

Οι συσκευές ασύμμετρης προσπέλασης, είτε ως κρυφή μνήμη είτε ως ξεχωριστά αποθηκευτικά μέσα, παρέχουν πολλαπλά πλεονεκτήματα σε σχέση με τα παραδοσιακά μέσα αποθήκευσης. Η επικοινωνία των συσκευών αυτών με τα υπάρχοντα συστήματα είναι εφικτή μέσω ειδικού λογισμικού που εκτελείται στον ελεγκτή της συσκευής και υποστηρίζει μια παραδοσιακή διεπαφή τύπου μπλοκ. Με κατάλληλη αξιοποίηση των σημασιολογικών πληροφοριών που προσφέρονται από τις εφαρμογές ένα σύστημα αρχείων είναι δυνατόν να βελτιώσει την απόδοση της συσκευής, καθώς επίσης και να εκμεταλλευτεί τις ιδιαιτερότητες της ασύμμετρης προσπέλασης προς όφελός των εφαρμογών.

Ειδικότερα, τα συστήματα με δομή αρχείου καταγραφής (log-structured) παρέχουν αυξημένη απόδοση και μειώνουν το χρόνο απόκρισης. Ωστόσο, σε αρκετές περιπτώσεις δεν διαχειρίζονται αποδοτικά τον αποθηκευτικό χώρο και επιβαρύνονται σημαντικά κατά τη διαδικασία εκκαθάρισής του. Επιπλέον, ο όγκος των δεδομένων που μπορεί να ανακτηθεί μετά από κατάρρευση του συστήματος εξαρτάται από τον φορτίο εργασίας τη χρονική στιγμή της κατάρρευσης και μπορεί να είναι περιορισμένος.

Προκειμένου να αντιμετωπίσουμε αυτές τις αδυναμίες προτείνουμε το Metis, ένα σύστημα με δομή αρχείου καταγραφής (log-structured) που χρησιμοποιεί ένα ξεχωριστό αρχείο καταγραφής ενημερώσεων (journal) προκειμένου να βελτιώσει τη διαδικασία εκκαθάρισης, να μειώσει τις απαιτήσεις σε εύρος ζώνης αποθήκευσης, και να πετύχει αποδοτικότερη

διαχείριση του αποθηκευτικού χώρου. Υλοποιήσαμε το σύστημά μας στον πυρήνα του λειτουργικού συστήματος Linux. Μέσω λεπτομερών πειραματικών μετρήσεων δείχνουμε ότι επιτυγχάνουμε αποδοτικότερη διαχείριση του αποθηκευτικού χώρου και βελτιωμένη ικανότητα ανάκτησης δεδομένων, ενώ η απόδοση του συστήματος παραμένει συγκρίσιμη με αυτή του αρχικού.

Chapter 1

Introduction

1.1 Thesis Scope

Technological advances increasingly make flash memory useful as either standard storage device in mobile systems or desirable storage layer in enterprise servers. Flash memory exhibits several attractive characteristics, such as low power consumption, high transaction throughput and reduced access latency. However, it has several limitations such as the erase-before-write requirement, the limited number of Program/Erase cycles per block and the need to write to erased blocks sequentially.

Early systems utilized "raw" NAND flash memory as their primary storage managed by a special purpose file system [66, 58]. As the storage needs grew however, a new layer of abstraction was added in the form of a firmware running on the controller known as the Flash Translation Layer (FTL). The FTL hides the underlying complexity of NAND flash memory by exposing a generic block interface to the system. Examples of such devices include SD memory cards, USB sticks and Solid State Drives. Solid State Drives in particular, are able to exploit the parallelism provided by the underlying architecture

and outperform their mechanical counterpart by orders of magnitude when it comes to random I/Os while offering very low access latency.

Even with the addition of the FTL however, flash memory bears idiosyncrasies that render its performance highly workload-dependent. Frequent random writes on an SSD lead to internal fragmentation of the underlying media and degrade its performance even by an order of magnitude [47]. More specifically, random writes trigger the garbage collection mechanism of the FTL and as a result, valid pages within victim blocks have to be relocated before the block itself can be erased. This phenomenon is known as write amplification [21, 41, 53] and is of utmost importance to the industry and the academia since the extra writes impair the device's performance and have also a negative impact on its lifetime.

Random-write patterns are quite common in modern applications. For example, the Facebook mobile application issues 150% more random writes that sequential ones while WebBench issues 70% more [30]. In addition, 80% of the total I/O operations issued are random and more than 70% of them use the *fsync* system call to force the data into persistent memory [25]. Therefore, unless handled carefully, random workloads can seriously affect the performance of the drive and reduce its lifetime.

Much effort has been devoted by researchers into resolving the random writes problem, reduce the write amplification factor and extend the device's lifetime. Most of the work focuses on the Flash Translation Layer with some studies attempting to categorize incoming writes with respect to the update frequency of the underlying pages, so that frequently modified pages (hot) do not pollute blocks with rarely modified pages (cold) [34, 18]. Others, exploit physical aspects of the device in order to improve its lifetime and increase its performance [26, 22]. Nonetheless, inefficiencies of the FTL have led to the proposal of a number of non-FTL solutions in the literature [36, 23, 61, 24].

One might expect that the use of a log-structured file system [57, 31] or one that follows the copy-on-write strategy like BTRFS [55] would help soothe the detrimental effects of random writes. However, these file systems do not consider the unique characteristics of flash storage and are therefore sub-optimal in terms of performance, reliability and device lifetime. As a result, a number of flash-aware file systems have been suggested that help optimize the usage of flash media [33, 47, 40]. Still, these file systems either use complicated mechanisms to identify hot and cold data in memory or exchange the

dependability of the drive for performance.

In this thesis we propose *Metis*, a composite flash-aware file system that combines journaling with the log-structured file system. By using a set of novel techniques that add minimal writing overhead, our system is able to:

1. Provide hot/cold block separation
2. Increase the efficiency of the garbage collection mechanism
3. Improve the recovery point objective of the system
4. Improve disk utilization

Experimental results show that our system is able to improve the utilization of the disk by up to 80% while performance remains comparable to that of the original file system. Our work is based on a previously proposed idea published by our group [19, 20].

1.2 Thesis Organization

The remainder of the thesis is organized as follows:

In **Chapter 2**, we provide the background required to understand the underlying problems of Solid State Drives. We then proceed to an overview of the various problems and techniques that have been proposed in order to improve their reliability and performance. In addition, we discuss the problem of file system consistency and present some basic information about Log-Structured File Systems.

In **Chapter 3**, the design goals of our system are defined and a general description of its key features and architecture is provided.

In **Chapter 4**, we introduce Metis, the composite file system that we have designed and developed. Our work is based on the idea of combining journaling with the Log-Structured File System. Frequently updated blocks are accumulated on the journal partition while only cold blocks are written on their final location in order to improve the reliability of the drive and increase its utilization.

In **Chapter 5**, we specify the characteristics of our experimentation platform. We then evaluate our system across different workloads and present our results graphically.

In **Chapter 6**, the concluding remarks of this thesis are drawn and future work is discussed.

Chapter 2

Background

2.1 Flash Memory

Semiconductor memories can be divided into two major categories: *Volatile memories* and *Non-Volatile memories*. Volatile memories can be programmed fast, easily and an unlimited number of times but require power to preserve the stored information. Non-Volatile memories are able to retain the stored information in the event of a power loss, but fall short in terms of speed and durability. Early designs of non-volatile memory were fabricated with permanent data and did not provide the ability to modify their content. Current designs allow their content to be erased and re-programmed a limited number of times although at a significantly slower speed compared to volatile memories [8, 6].

Flash Memory is a type of non-volatile memory that incorporates technologies from both EPROM and EEPROM memories. Invented in 1984 by Dr Fujio Masuoka, the term "flash" was chosen because a large chunk of memory could be erased at one time

compared to EEPROMs where each byte had to be erased individually. It is considered to be a very mature technology and has become extremely popular, especially since the development of battery operated portable electronic devices in the mid 1990s. Nowadays, flash memory is used on virtually all hand-held devices, USB memory sticks and memory cards with its main advantages being the high Read/Write speeds, small size, low power consumption and shock resistance. It also serves as the main component of Solid State Drives.

2.1.1 Introduction

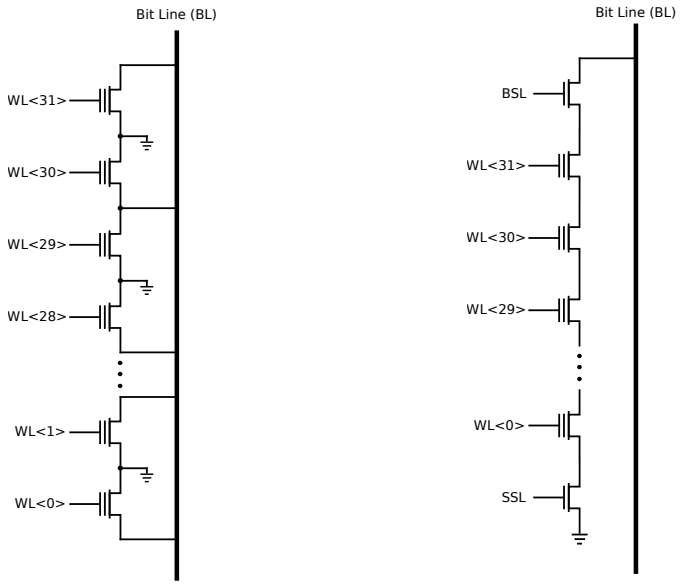
The heart of the flash memory cell is the *Floating-Gate Transistor*. Its structure is similar to a conventional MOSFET[1], with an additional gate added. The newly added gate called the *Floating Gate*, occupies the position of the original gate with the original gate (known as the *Control Gate*) now being on top of the floating gate (Figure A.2). The floating gate is insulated all around by an oxide layer and as a result, any electrons placed on it are trapped there and —under normal conditions— will not discharge for many years [46]. It is because of this phenomenon, that the floating-gate transistor can be used as non-volatile memory.

The reliability of a floating-gate transistor is one of its most important concerns. There are multiple leakage paths which can lead to loss of the programmed floating-gate electron charges however, with the use of proper monitoring systems flash memory is able to detect and correct bit errors in order to provide reliable storage. [46, 49]. Flash memory manufacturers typically need to guarantee at least 10 years of charge retention and $1k$ to $100k$ Program/Erase cycles for a product chip.

2.1.2 NOR and NAND Architectures

Depending on how the memory cells are organized within the chip, flash memory comes in two main architecture variants: NOR and NAND. NOR flash memory came to the market first as an EEPROM replacement and is mainly used as code and parameter storage where reliability and high speed random access is required. NAND flash is used mainly for data storage due to its high speed serial access and high density. We now look at the two

[1] Metal-Oxide-Semiconductor Field-Effect-Transistor



(a) (b)

Figure 2.1: NOR and NAND Flash architectures.

architectures and discuss their advantages and disadvantages.

NOR Flash Memory

In NOR flash memory, each cell has one end connected directly to ground, while the other end is connected to a *Bit Line (BL)* (Figure 2.1a). This arrangement resembles a NOR gate: When one of the *Word Lines* (WL) is brought high, the corresponding transistor pulls the output bit line low. The main advantage of this arrangement is that it allows each memory cell to be individually addressed, which results to high speed random access capability. In addition, NOR flash memory offers reliable storage due to the large size of the memory cell, which reduces the number of disturb faults which in turn, reduces the need for expensive error correction codes.

It is because of these features, that NOR Flash Memory is mainly used as code and parameter storage. When a program is executed, it generally requires fast random access reads, since the code usually jumps from one memory location to another while branching. Moreover, any errors in the code are likely to produce a system fault so a high degree of reliability is required. Since NOR flash provides all of the above features, it is an ideal solution for firmware storage and in-place execution.

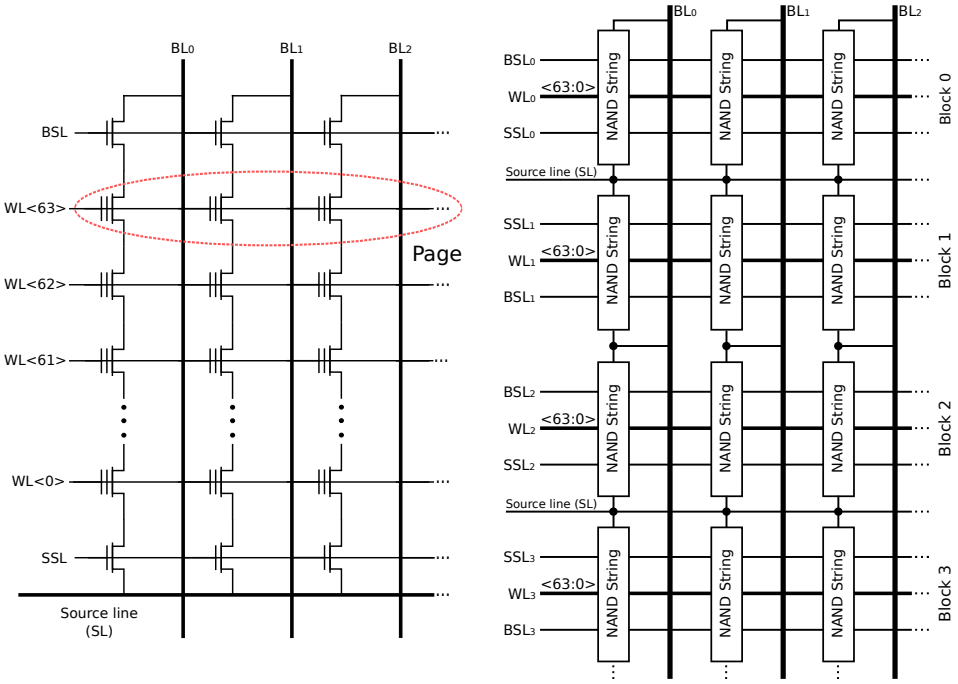


Figure 2.2: NOR and NAND Flash architectures.

NAND Flash Memory

In NAND flash memory, the transistors are connected in a way that resembles a NAND gate: Several transistors are connected in series and only if all word lines are pulled high is the bit line pulled low. The transistor group is then connected via some additional transistors to a bit line in the same way that a single transistor is connected to a bit line in NOR flash (Figure 2.1b).

In order to read a value, first the desired group is selected by using the Bit Select Line (BSL) and Source Select Line (SSL) transistors[2]. Next, most of the word lines are pulled high while one of them is read by applying the proper voltages to its connectors. Since there are no contacts directly accessing the memory cells, random performance is slow compared to NOR flash. However, the reduction in source lines and bit line contacts leads to a smaller effective cell size resulting in a smaller chip size and a lower cost per bit. More specifically, the cell size for NOR flash memory is in the $10F^2$ range (where F is the design rule of the chip), while for NAND flash memory it is reduced to $4F^2$ [44, 6].

When it comes to cell organization within a chip, NAND strings are packed to form a matrix in order to optimize silicon area occupation. Strings that share the same source

[2]When designing the chip, the ground contacts of figure 2.1 are replaced with *Source Lines* (SL)

	NOR	**NAND**
Read parallelism	8-16 Word	2Kbyte
Write parallelism	8-16 Word	2Kbyte
Read access time	$< 80ns$	$20\mu s$
Program time	$9\mu s/Word$	$400\mu s/Page$
Erase time	$1s/Sector$	$1ms/Sector$

Table 2.1: Comparison of NOR and NAND Flash architecture characteristics

line but have different bit lines form a logical *Block*. Transistors within a block that share the same word line form a logical *Page* (Figure 2.2). A typical page has a size of about 8-32kB while a block usually contains hundreds of pages. Read and write operations are performed at the granularity of a page and are relatively fast operations (25-60μs for Read and 250-900μs for Write). The erase operation is performed at the granularity of a block and is considered to be a very slow operation (about 3.5ms) [26].
To sum up, the main advantages of NAND Flash Memory are:

- Fast write performance, since multiple cells can be programmed simultaneously.
- High density and low cost per bit due to the small cell size.
- Low power consumption because of the use of the FN Tunneling mechanism (See Appendix A).

These characteristics make it an ideal candidate for data storage since most applications need to store large amounts of data and read them sequentially. A comparison of characteristics of NOR and NAND flash architectures can be seen in Table 2.1. For the rest of this thesis we are going to focus on NAND flash memory only as it is the main component of Solid State Drives discussed in the next section.

2.1.3 Multi Level Cell Memory

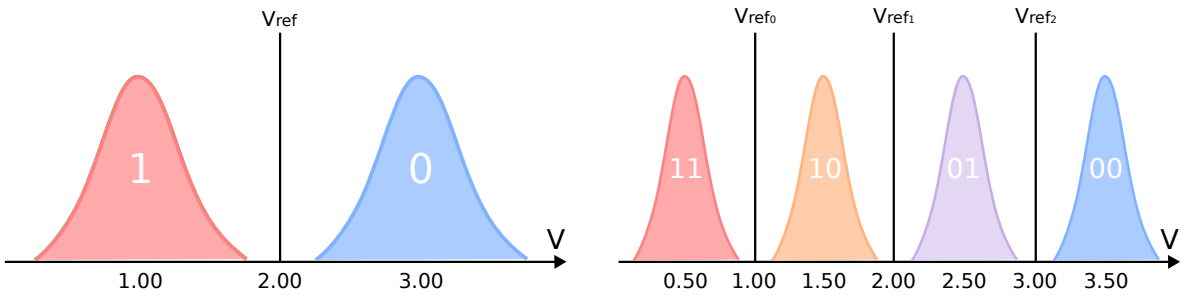
Until now, we assumed that only one bit of information could be stored in a single memory cell. This type of memory, where a memory cell can only store one bit and its value is determined by judging whether electrons are trapped in the floating gate or not, is known

as *Single-Level Cell* (SLC) memory. Let us now assume that the voltage applied to the control gate to produce the two states is 1.00 and 3.00 volts respectively, with a *Reference Voltage* V_{ref} of 2.00 volts (Figure 2.3a). The bell-shaped area exists because the threshold voltage is not going to be exactly the same for all the cells in a chip. In order to read the contents of a memory cell, the reference voltage is applied to the control gate. If the cell is in an erased state (bit value 1), current will flow through the cell. If there is no current flow, the floating gate must be occupied by electrons and therefore the cell is in a programmed state (bit value 0).

Multi Level Cell (MLC) memory technology uses multiple levels per cell to allow more bits to be stored in a single transistor. More specifically, MLC memory has 4 distinct states with each state representing 2 bits: 11, 10, 01 and 00. A cell in an erased state will be seen as the value 11 while the remaining three states are determined by accurately measuring the amount of current that flows through the cell during read operation. Instead of having two threshold voltages we now have four ($0.50V$, $1.50V$, $2.50V$ and $3.50V$ respectively), while the reference points between the states are now three (Figure 2.3b).

The main benefit of MLC memory, is that the storage capacity can be increased by using the same number of transistors and without an increase in process complexity. Specifically, transitioning from SLC to MLC technology is equivalent to a partial shrink of the device from one process technology generation to the next. 3-bit and 4-bit per cell memories have also been constructed (known as X3 and X4 MLC memories), however the cost per bit reduction in becoming smaller as we transition from one technology to another. When transitioning from SLC to MLC an approximate 40-50% cost reduction can be obtained but this figure drops to 20% for MLC to X3 MLC and to 10% for X3 MLC to X4 MLC [45, 38, 14].

Programming and reading a memory cell that stores multiple bits is also extremely demanding. Since we now need to control and sense the number of electrons trapped in the floating gate, high accuracy is required. In particular, high precision sensors and a more complex error correcting circuitry need to be used, which in turn leads to slower programming and reading speeds of about 3-4 times compared to those of SLC memory. Moreover, the bell shaped areas are now narrower since the voltage difference between the threshold voltages is now less and there is also a smaller voltage separation between the reference voltage and the upper and lower bounds of the threshold voltages on each side



(a) (b)

Figure 2.3: Single Level Cell and Multiple Level Cell memory voltages.

	SLC	**MLC**
Pages per Block	64	128
ECC bits	4-24 bits	16-40 bits
Program/Erase cycles	100.000	10.000
Raw bit error rate	10^{-4}	10^{-3}
Read latency	$25\mu s$	$50\mu s$
Program latency	$200 - 300\mu s$	$600 - 900\mu s$
Erase latency	$2ms$	$3ms$

Table 2.2: Comparison of SLC and MLC memory characteristics

of the reference voltage. As the floating gate will slowly lose its ability to retain electrons over time (remember that a memory cell can sustain a certain number of Program/Erase cycles), its threshold voltage will start to shift until it is off the bell shaped area and the chip will then be considered unusable since we cannot distinguish the value that is stored in it. Therefore, the smaller this area of distribution around the threshold voltage is, the less longevity the cell has. As a result, MLC memory cells can only sustain up to 10.000 P/E cycles while SLC cells can sustain up to 100.000 [9, 63]. A summary of the various characteristics of SLC and MLC memory can be found in Table 2.2.

2.2 Solid State Drives

NAND flash based Solid State Drives (SSDs) have been revolutionizing data storage systems in the latest years. A solid state drive is a purely electronic device based on semiconductor memory that has no mechanical parts compared to traditional Hard Disk Drives (HDDs). As a result, it has minimal seek time and provides very low access latencies. Moreover because of its electronic nature it offers low power consumption, lack of noise, shock resistance and most importantly very high performance for random data accesses.

Although the SSD technology has been around for more than 30 years, it remained too expensive for broad adoption. The introduction however of MP3 players, smartphones and a whole other class of devices that used flash memory as their main storage changed everything and led to the widespread availability of cheap non-volatile memory. As a result, consumer grade SSDs started to appear in the market in 2006 with their market share growing ever since [11, 46]. Still, two serious problems are limiting the wider deployment of SSDs: Limited lifespan and relatively poor random write performance [47]. In the next sections we look at the basic organization of an SSD and discuss its weaknesses.

2.2.1 Organization

Hard Disk Drives and SSDs are part of a class of storage called *block devices*. These devices use logical addressing to access data and abstract the physical media using small, fixed, contiguous segments of bytes as the addressable unit called *blocks*. A typical solid state drive incorporates three major parts: The NAND flash storage media, the controller and the host interface.

The NAND flash memory is the basic building block of an SSD and is organized in a hierarchical fashion (Figure 2.4). As we described in section 2.1.2, flash memory consists of Pages (2KB, 4KB or 8KB size) and Blocks (64-512 Pages). A page is the basic unit of read and write operations while erase operations are performed at the block level[3]. A *plane* is a matrix of few thousand blocks, while a *die* usually contains 4 planes and has a size of 2-16 GBytes. Each die can operate independently from one another and perform

[3]It is important not to confuse the block unit of a block device known as logical block with the flash block. A logical block usually has a size of 4KB and is used by the operating system. A flash block is about 256KB - 4MB, it is comprised of multiple pages and is used by the NAND flash memory.

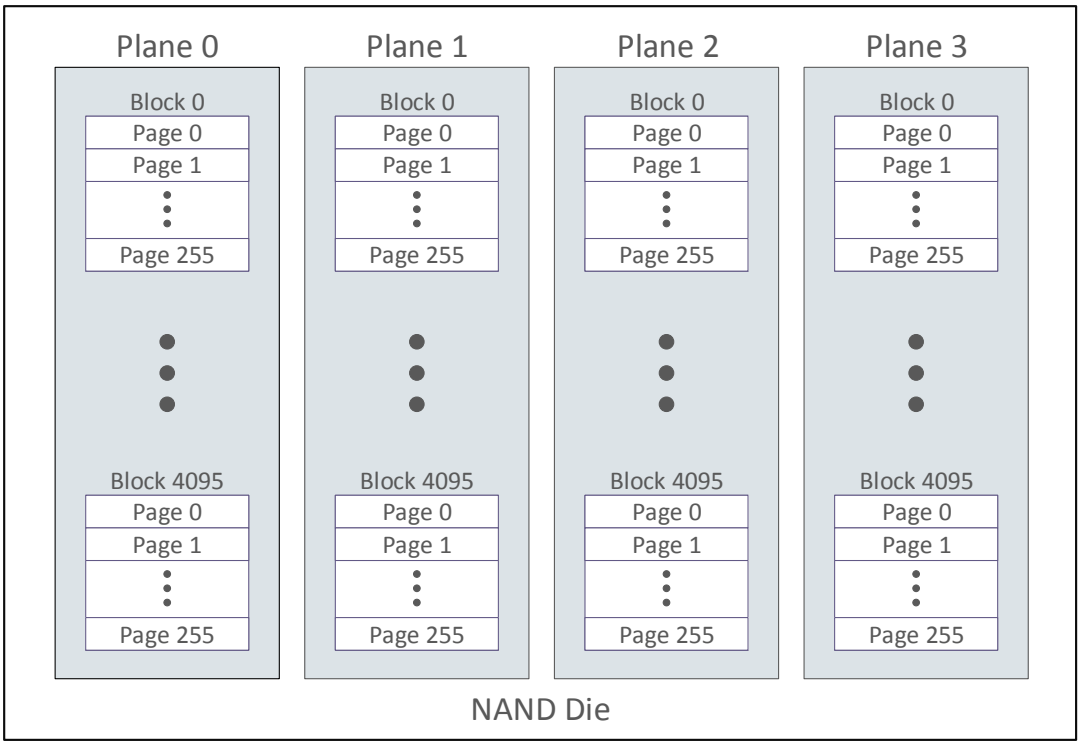


Figure 2.4: NAND Flash memory organization

operations in one or two planes. Multiple dies (2-32) can be organized to share the same *memory channel* and only one of them should be active at any given time. A typical SSD has multiple memory channels (usually 4-8) that are directly connected to the NAND Controller (Figure 2.5). It is because of this organization that SSDs are able to provide high bandwidth rates (about 500MB/s) even though a single flash package has limited performance (about 40MB/s). By splitting the data up into separate memory channels, a high degree of parallelism can be achieved which leads to a vast increase in performance [2, 11, 46].

Some file systems —mainly used for embedded applications— opt to use raw flash memory in order to store their data [58, 39, 66]. In contrast, solid state drives use the a special firmware that runs on the controller called the *Flash Translation Layer* in order to expose a linear array of logical block addresses to the host while hiding its inner workings. As a result, existing file systems can be used with SSDs without virtually any problems while the migration from a HDD to an SSD becomes seamless. The Flash Translation

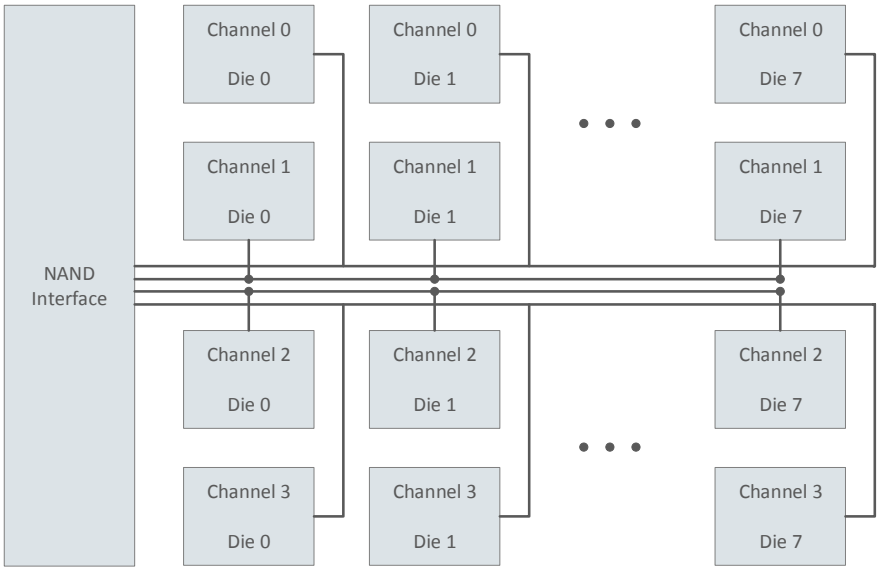


Figure 2.5: NAND Flash Channel Organization

Layer and its mechanics are discussed in detail in the next section.

The host interface also plays an important role in SSD design. The majority of solid state drives utilizes existing standards and interfaces such as SATA and SAS in order to achieve maximum compatibility with HDDs. Some high-end SSDs use the PCIe (Peripheral Component Interconnect Express) interface which is a more efficient interconnect since it does not use the host bus adapter that is required in order to translate the SATA and SAS protocols. This in turn leads to a decrease in latency and in power consumption.

The use of traditional block-access protocols however leads to the loss of higher-level semantics that could be utilized in order to make the SSD more efficient. This is one of the most important problems in Solid State Drive block management and while various solutions have been proposed in the literature[13, 15], most modern SSDs continue to use existing protocols and interfaces.

2.2.2 Flash Translation Layer

In order to protect and control the underlying NAND flash media solid state drives employ a complex embedded system with standalone processing in the memory controller that runs a special firmware called the *Flash Translation Layer* (FTL) (Figure 2.6). The FTL

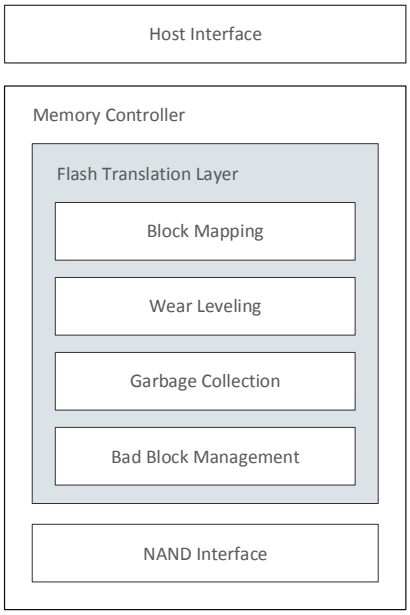


Figure 2.6: Block diagram of an SSD memory controller

exposes an interface similar to that of a HDD to the host system while hiding the unique flash physical aspects. Thus, it takes care of all the necessary operations that otherwise the host file system would have to in order to make the NAND flash robust and reliable. The FTL plays a key role in SSD design and many sophisticated mechanisms have been adopted over the years in order to optimize its performance [41, 26, 59]. A typical FTL performs four main operations: logical block mapping, garbage collection, wear leveling and bad block management.

Logical Block Mapping

As we mentioned in previous sections, pages in NAND flash memory cannot be overwritten and have to be erased before we are able to program them again. However, the program operation is performed at page granularity while the erase operation is performed at the block level. As a result writes in flash memory cannot be performed in place as in disks and a different page has to be used in order to store the updated data, a process known as *out-of-place update*.

The FTL maintains a *state* for every physical page while exposing an array of logical block addresses to the host system. A page can be in either one of three states:

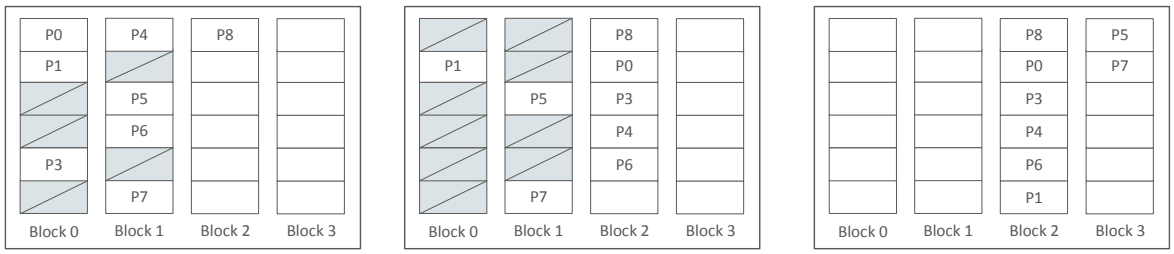


Figure 2.7: Page state transitions. Blank boxes stand for clean pages that can be used to write data, valid pages have their name assigned in the box while invalid pages are coloured grey.

- *Clean*, meaning that it is a page that has been erased and is now available for programming
- *Valid*, if it contains valid data
- *Invalid*, if the data contained is now stale and the page can be erased

Data is written into clean pages initially, which are then marked as valid. When an update occurs, a new clean page is used to store the updated data while the old page is now marked as invalid (Figures 2.7a & 2.7b). Hence, the FTL employs a mapping mechanism in order to map a *logical block address* (LBA) to a *physical block address* (PBA) and keep track of all the page state changes. The mapping granularity of such mechanism varies and it can be as large as a block or as small as a page. In general there are three mapping schemes that can be adopted [17, 9, 47]:

- In a *page-level* FTL scheme a logical block number is translated to a physical page number. This results to a fine-grained page-level mapping table that adds very low garbage collection overheads. Such granularity however comes at a price a large mapping table needs to be stored in RAM which might not always be possible.
- A *block-level* scheme maps a logical block number to a physical block number while the page offset within a block is fixed. The mapping is now coarse-grained and therefore the mapping table is small enough to be kept in memory entirely without

a problem however, garbage collection overheads have now raised especially when only a few pages within a block must be updated.

- An intermediate solution is to use a *hybrid-level* scheme that extends the block-level schemes by partitioning the flash blocks into data blocks and log blocks. Data blocks are mapped using the block-level scheme in order to reduce the required RAM size while log blocks are mapped using page-level mapping. The idea is to reduce the garbage collection overhead for the log blocks and use them as log buffers by directing small random writes to them that can be later grouped and written to data blocks.

The majority of SSDs today use a hybrid FTL scheme with the mapping table maintained in persistent flash memory and rebuilt in RAM at startup time.

Garbage Collection

The number of invalid pages grows as the device is written. Since writes can be written only to clean pages the FTL must maintain a pool of free blocks by triggering the conversion of invalid pages into clean ones. This process known as *garbage collection* is triggered either when the FTL runs out of free blocks or when the SSD is inactive for a period of time.

During the Garbage Collection process sparsely filled *victim blocks* with lots of invalid pages are selected, their valid pages copied into new blocks, while the old blocks are erased (Figure 2.7c). The newly erased blocks can now be used to accommodate new writes. Depending on the mapping scheme used, different policies need to be followed: If a page-level mapping is used, the valid pages in the victim block are copied out and *condensed* into a new block. If a block-level mapping is used, the valid pages need to be *merged* together with the updated pages in the same block.

Copying the remaining valid data of a victim block presents a significant overhead and therefore the garbage collection process should be triggered only when necessary since it has a great impact at the performance of the drive.

Wear Leveling

NAND blocks have limited endurance and are able to sustain a specific number of Program/Erase cycles. In addition, not all the information stored within the same memory location change with the same frequency: Because of the locality exhibited in most workloads writes are often performed in a subset of blocks which is frequently overwritten [26]. This can lead to certain blocks wearing out earlier than other blocks which poses a problem as it can lead to the device becoming unusable prematurely.

To counter this problem the FTL employs a wear-leveling mechanism in order to maximize the use of the blocks' limited endurance and guarantee a sufficient device lifetime. More specifically, the FTL keeps track of the number of P/E cycles for each block and aims to keep the wear as low and uniform as possible by "shuffling" blocks that are updated frequently (hot) with blocks of data that remain the same for a very long time (cold) and have a low P/E count. This poses a major challenge for SSD manufacturers and academic researchers and over the years various solutions and algorithms have been proposed. A discussion of the problem and its proposed solutions can be found in the next section.

Bad Block Management

No matter how good the wear leveling algorithm is, eventually *bad blocks* are going to start to appear in the NAND flash. A block is considered bad when a program or erase operation fails or when the bit error rate grows close to the Error Correcting Code capabilities.

Therefore, the FTL maintains a map of bad blocks that is created during the factory initialization of the device and contains a list of the bad blocks that have been found during factory testing [46]. When a block is found to be bad during the device lifetime it is added to the list and the FTL will stop using it. However, if there are no spare blocks to replace all the failing blocks the device will be considered unusable and will die.

2.2.3 Challenges

Despite their obvious advantage of providing performance that is orders of magnitude higher than that of hard disk drives, the design of solid state drives presents significant challenges that need to be addressed. We now look at the most important ones and

describe the various solutions that have been proposed.

Random Writes

Random writes are generally considered harmful in solid state drives. Although the random read performance of an SSD remains the same through the device's lifetime, random writes can cause internal fragmentation of the underlying media and thus lead to a decrease in performance even by an order of magnitude. The main reason behind this drop is the erase-before-write property and the asymmetry of the NAND flash memory. Multiple random writes result to multiple pages in different blocks becoming invalidated as their contents are updated and copied out to a new page. As a result, many blocks now contain a relatively small number of valid pages which leads to an increase in the garbage collection overhead or if a hybrid FTL scheme is used, a costly *full merge* operation has to be performed [47, 35].

To counter this problem many researchers have proposed the use of a *Log-Structured File System* (LFS) in the FTL [57, 17, 2, 18]. The LFS groups multiple write requests and writes them to the disk as a single segment. As a result multiple write requests are written to the disk sequentially in a log-like fashion which leads to higher write performance and less internal fragmentation.

Hot Block Management

Since copying the remaining data of a victim block into a new block during the garbage collection process adds a significant overhead, blocks selected for erasing should have as many invalid pages as possible. To achieve this many researchers have proposed the idea of grouping data with similar update frequencies, a process known as *Hot/Cold Block Separation*. More specifically, data blocks are separated into two (or sometimes more) categories: Data that is likely to be updated soon (i.e., hot blocks) and data that will probably remain the same for a long time (i.e., cold blocks). A block that contains hot data is likely to contain a small number of valid pages since most of its pages will have already been invalidated by the time garbage collection begins and therefore a lower garbage collection overhead is to be expected. To identify hot blocks, various heuristics have been suggested in the literature over time [50, 47, 37].

The process of separating data according to their temperature is crucial not only for performance reasons but also because it affects the endurance of the drive. Hot blocks can be kept in memory longer and wait for updates on their data before writing it to the SSD drive. Various schemes have also been proposed that use Non-Volatile RAM [23] or a hard disk drive [61] to direct hot blocks before writing them to their final location. In addition, the wear leveling algorithms can now be improved since hot blocks can be identified and sent to NAND blocks with a low P/E cycle count. Some researchers have also proposed directing hot blocks to MLC NAND blocks that have been modified to act as SLC blocks [36, 51] or to specific blocks that have been measured to have higher endurance than others [26]. The lifetime of the device can thus be extended and its performance improved without an increase in its resources.

Write Amplification

Operating systems assume that the time taken to complete an I/O operation is proportional to the size of the I/O request. While that might be true for HDDs, things differ when it comes to solid state drives. Since data in flash memory are always updated out of place, multiple updates can trigger the garbage collection process, which results to a series of write and erase operations being performed. This phenomenon is known as *write amplification* (WA) and may lead to a reduction of the drive's endurance since the extra writes wear out the flash cells more quickly [53].

This problem is addressed by employing one of the hot/cold block separation techniques mentioned before and by increasing the over-provisioning space of the device [60]. The *over-provisioning* (OP) space is a specially reserved portion of the flash blocks of the device that is not visible to the user. It is used to aid the garbage collection process by facilitating the migrated valid pages from victim blocks and also to extend the device lifetime by providing extra blocks that can be used to replace blocks marked as bad. The OP space in an SSD is typically defined as $\frac{C_{total}-C_{usr}}{C_{usr}}$ where C_{total} is the total amount of physical blocks and C_{usr} is the number of blocks that are available to the user. Most commodity SSDs set aside about 8% to 25% of their capacity as over-provisioning space while enterprise SSDs can set as much as 50% of their capacity. The use of more over-provisioning space greatly reduces the WA factor which is defined as the ratio of total physical writes to the writes perceived by the host and can increase the lifetime of the

device by up to 7 times [22].

2.3 Fast and Reliable File Systems

The main purpose of computers is to create, manipulate, store and retrieve data. A file system provides the mechanisms to support such tasks by storing and managing user data on disk drives and by ensuring that what is read from storage is identical to what was originally written. It considered as one of the most important parts of an operating system in order to manage permanent storage and besides storing user data in files, the file system also stores and handles information about files and about itself. A typical file system is expected to be extremely reliable and guarantee the integrity of all the stored data while providing very good performance.

2.3.1 Basic Concepts

Let us now review the basic building blocks of a file system:

- A *Volume* can be a physical disk or some subset (or superset) of the physical disk space known as a disk partition.
- A *Block* is the smallest unit writable by a disk or a file system. All file system operations are composed of operations done on blocks.
- Critical information about a particular file system is stored in a special structure called *Superblock*. A superblock usually contains information such as how large a volume is, the number of used and unused blocks, a magic number to identify the file system type and so on.
- *Data blocks* store the actual data of a file. No other information describing those data (such as the file's name) is stored in those blocks.
- The file system *Metadata* is a generic term that refers to its internal data structures concerning a file except the file's actual data. Metadata includes time stamps, ownership information, access permissions, storage location on the disk etc.

- All the information about a file except the data itself is stored in an *inode*. A typical inode contains file permissions, file type, number of links to the file and also some pointers to data blocks that contain the file's actual data. Each inode is assigned a unique inode number in order to distinguish it from all the others.

2.3.2 Caching and Buffering

Reading and writing to a file requires issuing many I/O operations to the disk, a process which takes a considerable amount of time. To counter this problem, most file systems use a part of the system memory as a cache for important blocks. The advantages of such approach are obvious: read and write requests now take place in the cache and the overall performance of the file system is improved.

Let us now consider an example where we want to read some data from the disk. If it was the first I/O request issued on the disk, a lot of I/O traffic is going to be generated since we have to access several directories before reaching the file inode. If however a similar read request was issued before, most of the requested data would now be available in the cache and thus no I/O is needed. Instead the only thing that we now have to read from the disk is the inode of the file and its corresponding data blocks.

Caching has also an important effect on write performance. Write requests are now being fulfilled by the cache and as a result require minimal time to complete. Besides the decrease in latency, write buffering has also a number of performance benefits. By delaying writes the system can batch some updates that would otherwise require multiple I/O requests into a single one (for example when a file is updated). In addition, the I/O controller can now schedule subsequent I/Os and thus increase the performance of the drive while sometimes it is even possible to avoid some requests (e.g., when writing a file and then erasing it).

Eventually however we need to flush the modified blocks to the disk in order to become persistent. Most modern file systems buffer writes in memory for 5 to 30 seconds and then flush them to the disk. If the system crashes before the updated blocks are sent to disk, the updates are lost. For that reason, applications that require a high degree of reliability are able to force the updates to be sent directly to the disk by using the *fsync()* system call.

Figure 2.8: Organization of a novel file system: Superblock, Inode bitmap, Data block bitmap, Inode blocks and Data blocks

Contemporary systems use file system pages that are integrated with virtual memory pages into a unified page cache to store disk blocks. Multiple block buffers are stored within a page and each block buffer is assigned a specialized descriptor called the *buffer head* that contains all the necessary information required by the kernel to locate the corresponding block on the disk. As a result, physical memory can be allocated much more flexibly between the virtual memory and the file system depending on the needs of the operating system at any given time.

Before a write occurs, the kernel verifies that the corresponding page exists in the page cache. If the page is not found and the write operation has the exact size of a complete page, a new page is allocated, filled with data and marked dirty. Otherwise the corresponding page is fetched from the disk, its contents are modified accordingly and then it is marked as dirty. During the *writeback* operation the kernel searches the page cache for dirty pages and flushes them to disk. Writeback occurs in two cases: Either when the age of dirty pages grows beyond a certain threshold or when the kernel wants to free some memory up.

2.3.3 Journaling File Systems

We now present a simple file system organization that will help us understand file system consistency. As we have already mentioned, the logical volume is split into blocks. Let us assume that the first block of the disk hosts the superblock structure that contains important information about the file system. After that we use two blocks to keep the inode and data bitmaps that keep track of the free inode and data blocks in the system. We then use some blocks to keep the inodes organized in a table fashion, a structure known as the *inode table*. A typical size for an inode is about 256 bytes while a typical disk block has a size of 4KB as a result, a single block is able to store multiple inodes.

After the inode blocks we have the data blocks which eventually take up most of the disk's space (Figure 2.8).

Suppose that we want to append some data to a file that is written on disk and extend its size by 5 blocks. Even though appending the new data seems to be a single atomic operation at the system call level, the actual process involves a number of steps that need to be carried out at the block level:

1. First, five new data blocks are allocated by searching the corresponding bitmap and marking them as in-use

2. Next, the file inode is updated to point to the new data blocks and record the new size of the file

3. Finally, the actual data are written into the data blocks

Remember now that the operating system buffers write requests to the system memory for some time before writing them to disk. If the system crashes before flushing the dirty blocks to the disk, the updates are lost. If however a failure occurs, *while* writing the blocks to the disk, the system is left at an *inconsistent state*. Any of the following outcomes is possible which impacts the integrity and the reliability of the file system:

- Just the bitmap changes are written to disk. We now have reserved some blocks but they do not contain data, nor there is an inode pointing at them.

- Just the inode is written. We have updated the inode, but it points to blocks that contain no valid data. Moreover, those blocks can be used by other files since they are not marked as in-use on the bitmap.

- Just the data blocks are written. As the file metadata have not been updated we cannot read the data stored in those blocks anyhow.

- The bitmap and the inode blocks are written to disk. The metadata have been fully updated however, the inode now points to data blocks that contain garbage.

- The bitmap and the data blocks are written. New data blocks have now been allocated and they are not going to be overwritten since they are marked as in-use in the bitmap, but there is no file inode to point at them.

- The inode and the data blocks are written. We now have the correct data on disk and an inode that points to them however, the data blocks are eventually going to be overwritten since they are marked as free in the data block bitmap.

To counter this problem, early file systems employed a tool called *fsck* to examine all of the system's blocks and detect, report and repair any inconsistencies before mounting it. The main problem with this approach is that the fsck tool is required to scan *all* of the file systems blocks which is an extremely time consuming process. More specifically, the time required to complete the scan grows linearly according to the size of the partition and can take up to 30 minutes for a 300GB disk. In addition such, an approach cannot fix all the problems (consider the fourth case above) but it is rather used to make sure that the file system metadata is internally consistent.

Recent file systems use *Journaling* to keep track of all the changes that are going to be made to the file system before applying them. The idea was originally invented by the database community (where it is known as write-ahead logging) and is based on the power of transactions. Just like in a database system a journaling file system treats a sequence of changes as a single atomic operation. However, instead of tracking updates to tables, it records changes to the file system metadata and/or data. The transaction scheme guarantees that either all or none of the file system updates are done.

The records of the file system changes are stored in a separate part of the file system known as the journal or log. The log can be either a separate partition or a special inode structure within the file system itself and is treated by the kernel as circular buffer. Journal records that store file system changes are initially written to the log and once they are safely stored, the changes are applied to the file system. If the write operation is completed successfully, the corresponding transaction is removed from the journal. If a system crash occurs, all we have to do is read the journal and replay valid transactions by copying the corresponding blocks to the file system.

The main advantage of such approach is that the time required in order to bring the file system into a consistent state is now minimized and is independent of the partition size. In addition, the chances of losing data due to file consistency problems are greatly reduced while scalability is also improved. We discuss the inner workings of the Journal Block Device layer (JBD2) which is used in most modern file systems today in the next

chapter.

2.3.4 Log-Structured File Systems

In 1991 John Ousterhout and Mendel Rosenblum proposed a new file system type known as the *Log-Structured File System* (LFS). Based on a number of observations they suggested that an ideal file system should focus on providing high write performance since more and more read requests could be serviced in the cache as a result of the rapidly growing the memory sizes. To achieve this, large chunks of data would have to be written sequentially to the disk in order to minimize seek costs and make use of the high sequential bandwidth while the file system performance would approach the peak performance of the disk [57, 3].

In a log-structured file system the disk is treated as a continuous, append-only log divided into *Segments*. A typical segment contains multiple disk blocks and has a size of 4-8MB. The LFS buffers all data *and* metadata updates in an in-memory segment before writing them to the disk in one long, sequential transfer. As a result, writes are becoming efficient and the overall performance of the system is increased.

We now describe an example of how the log-structured file system allocates and writes files (Figure 2.9): Let us suppose that a new file with 5 data blocks is created. The LFS writes the 5 data blocks and the corresponding inode into a disk segment along with other file updates. If we update the data on the last data block and extend the file size to include two more data blocks, a new segment will be written that contains the three data blocks (one updated and two new) and the updated inode. The file inode now points to the first four blocks in the old segment and also to the three new blocks while the fifth block in the old segment and the old inode are invalidated.

Since both the data and metadata updates are now scattered over the log, the LFS has to periodically write the complete and consistent file structures safely at a fixed location of the disk called the *Checkpoint Region*. In case of a system crash the LFS uses the checkpoint region to quickly recover by going to the end of the log and performing a roll forward operation in order to salvage any segments written after the latest checkpoint.

As the file system size grows, the number of free segments reduces and eventually the log will become full when all the free segments have been used. However as we described

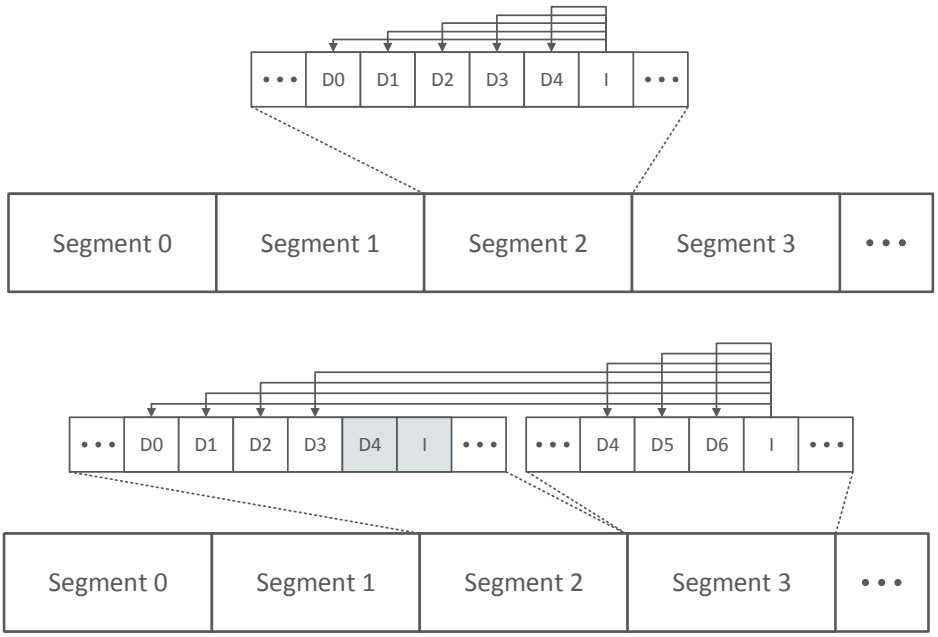


Figure 2.9: Creating and updating a file in a Log-Structured File System. The grey color indicates a block that contains stale data.

before, when updating a file the previous copy of the data will remain in its old location resulting in a "hole" in the segment. If this hole grows too big we could be using only a few blocks of a segment, while the rest of the space remains idle. Therefore periodically or when the number of segments in the disk falls below a threshold the LFS triggers the *Garbage Collection* process. During garbage collection a number of segments are scanned and the valid blocks are copied out to a new segment. The old segments are now free to be reused and the head of the log can wrap around to its tail and use the new segments.

The need for garbage collection is one of the major disadvantages of the Log-Structured File System. It is a very time consuming process that interrupts the normal disk operation and as a results no new writes can be serviced. Although it can be scheduled to run proactively when the disk is idle and while various algorithms have been suggested over time to improve its performance [4, 42], it still remains a burden to the wide adoption of LFS.

Various file systems based on the design of the LFS have been developed over time such as the Sprite FS [57], LinLog FS [12], F2FS [33] and some other prototypes for linux. Our design is based on the NILFS2 file system which is considered as one of the most

modern and active LFS implementations up to date [31].

2.4 Summary

Flash memory exhibits several attractive characteristics, such as low power consumption, high transaction throughput and reduced access latency. Even though it has several limitations such as the erase-before-write requirement and the limited number of Program/Erase cycles per block it has become extremely popular and is used by many devices as their primary storage mean.

Contemporary devices use a layer of abstraction known as the flash translation layer that hides the underlying complexity of NAND flash memory and exposes a generic block interface to the system. Moreover, the FTL takes care of wearing-out the device evenly and monitors the error correction codes of each block. Examples of such devices include SD memory cards, USB sticks and Solid State Drives. Solid State Drives in particular, exploit the parallelism provided by the underlying architecture and outperform their mechanical counterpart by orders of magnitude when it comes to random I/Os while offering very low access latency. Frequent random writes on SSDs however, trigger the garbage collection mechanism of the device and degrade their performance.

The file system used to store data on a solid state drive seriously impacts the device's performance and lifetime. Most file systems update the data in place and as a result, performance and consistency problems arise. Journaling file systems attempt to improve consistency by temporarily storing block updates into the journal and then transferring them to their final location. Traffic to the device however is increased, especially for sequential workloads. Log-structured file systems on the other hand, treat the disk as a continuous log and have shown to be beneficiary for flash media.

Chapter 3

Related Research

3.1 Log-Structured File Systems

Much work has been done in order to optimize log-structured file systems for conventional hard disks. Beginning with the original LFS proposal by Rosenblum et al. [57], Wilkes et al. proposed the *hole plugging method* [65] in which valid blocks in victim segments are overwritten to *holes*, i.e. invalid blocks, in dirty segments. This technique however, applies only to storage media that allow in-place updates. Matthews et al. proposed an adaptive cleaning policy that alternated between the original logging policy and the hole-plugging policy based on cost-benefit evaluation [42]. The cost model used however, is based on the performance characteristics of HDDs and takes into consideration the seek and rotational delay of the drive. Finally, Oh et al. introduced the idea of *threaded logging* that writes modified data to invalid areas of used segments in order to improve performance on a highly utilized volume [48].

A number of studies focus on separating hot and cold data. WOLF separates hot and cold pages into two different segments according to their update frequency and writes the

two segments to disk at once [62]. Nevertheless, this mechanism works well only when the number of hot and cold pages is approximately equal. HyLog adopts a hybrid approach and uses logging for hot pages to increase write performance and overwriting for cold pages to reduce the cleaning cost [64]. In order to determine the update policy however, it too uses a cost model that is based on the performance characteristics of HDDs.

SFS is a flash-aware file system that uses logging to eliminate random writes [47]. In order to reduce the cost of cleaning it separates hot and cold data based on the "update likelihood" of each block measured by tracking write counts and the block's age. Quantization is then used to partition segments into groups based on the measured hotness. This method adds significant overheads to the system since the update likelihood is calculated for each block that is sent to disk.

F2FS estimates the update likelihood using information readily available such as file operation, file type and file extensions [33]. It then categorizes blocks into hot, warm and cold and writes them into different parts of the disk. While this mechanism is fairly effective, the system uses threaded logging when it is under pressure in order to avoid garbage collection overheads. Such an approach however, negative impacts the device's lifetime since random writes cause fragmentation of the underlying media.

NVMFS is a composite file system assuming two distinct storage media: Non-Volatile RAM and Solid State Drive [52]. The fast, byte-addressable storage provided by the NVRAM is used to store hot data and metadata, while cold block updates are written to the SSD sequentially. TridentFS operates in a similar way, but also uses a Hard Disk Drive for cold data [23].

3.2 FTL Approaches

There has been much work in the industry and the academia aiming at improving random write performance at the FTL level. Most FTLs use a log-structured approach in order to overcome the erase-before-write limitation of flash memory and attempt to categorize data to hot and cold and store them to different parts of the disk.

FAST offers increased random performance by improving the log area utilization with flexible mapping [35]. LAST improves upon FAST by separating random log blocks

to hot and cold regions to reduce the full merge cost [37]. DAC provides a page-level mapping scheme that groups data according to their update frequency into the same segment to reduce garbage collection costs. A number of schemes that direct hot blocks to MLC NAND blocks that have been modified to act as SLC blocks [36, 51] or to specific blocks that have been measured to have higher endurance than others [26] have also been proposed in the literature. Nevertheless, FTL-level approaches face serious limitations since they rely only on logical block addresses to decide sequentiality, hotness and clustering.

3.3 Flash Memory File Systems

Raw flash memory file systems are common in embedded systems with limited resources. These file systems directly access NAND flash memory and address all the low-level issues such as wear leveling and bad block management by themselves. In order to handle the unique characteristics of flash memory most of these systems follow the log-structured approach. JFFS [66] and UBIFS [58] are two popular flash-based log-structured file systems that incorporate wear-leveling into the segment cleaning process. More specifically, the two file systems alternate between two different cleaning policies. The first one provides efficient garbage collection, while the other is used to wear-out blocks evenly.

A special case in flash memory file systems is the Direct File System (DFS) which leverages support from host-run FTL to simplify the file system design [27]. It is however, limited to specific flash devices and configurations and is not open source.

Chapter 4

System Design

4.1 Motivation

Even though Solid State Drives offer very low access latency and are able to outperform their mechanical counterpart by orders of magnitude when it comes to random I/Os, they bear idiosyncrasies that render their performance highly workload-dependent. Frequent random writes on an SSD lead to internal fragmentation of the underlying media and degrade its performance even by an order of magnitude [47]. More specifically, random writes trigger the garbage collection mechanism of the FTL and as a result, valid pages within victim blocks have to be relocated before the block itself can be erased. The extra writes impair the device's performance and have a negative impact on its lifetime.

Much effort has been devoted by researchers into resolving the random writes problem and extend the device's lifetime. Most of the work focuses around the Flash Translation Layer with some studies attempting to categorize incoming writes with respect to the

update frequency of the underlying pages, so that frequently modified pages (hot) do not pollute blocks with rarely modified pages (cold). Hot pages are then treated in a way that benefits the underlying flash media while cold pages are usually written to their final location as they would normally do [37, 29, 28, 34, 18, 36]. The physical aspects of the device have also been studied and a number of solutions that exploit its underlying characteristics in order to improve its endurance have also been proposed [26, 22]. Finally, writing mechanisms that allow the FTL to control block allocation decisions have been suggested in the literature in order to improve the utilization of the available disk space [67, 10].

The inability of the FTL to leverage semantic information about the application access patterns in order to optimize the traffic to the storage medium has led to the proposal of a number of non-FTL solutions in the literature. More specifically, various hybrid storage implementations that use Hard Disk Drives and/or Non-Volatile RAM alongside SSDs have been proposed [23, 61, 24]. By using a number of techniques these schemes are able to direct frequently updated data to the HDD or the NVRAM while cold blocks are written to the SSD.

One might expect that the use of a log-structured file system [57, 31] or one that follows the copy-on-write strategy like BTRFS [55] would help soothe the detrimental effects of random writes. However, these file systems do not consider the unique characteristics of flash storage and are therefore sub-optimal in terms of performance, reliability and device lifetime. As a result, a number of flash-aware file systems have been suggested that help optimize the usage of flash media [33, 47, 40]. Nonetheless, these file systems either use complicated mechanisms to identify hot and cold data in memory or exchange the dependability of the drive for performance.

4.2 Design Goals

We propose Metis, a composite file system that combines journaling with the log-structured file system. Our work is based on a previously proposed idea from our group[19, 20]. In the flash-aware file system that we propose, we set the following goals:

- Identify page access characteristics in host memory using existing mechanisms of

cache replacement in order to provide hot/cold data separation. Our system refrains from using complicated techniques that add significant overheads by calculating the hotness for each individual block. Instead, it uses existing mechanisms in order to find hot pages and direct them to the journal.

- Minimize the write traffic from host memory to flash storage without compromising data persistence. By sending hot data to the journal and cold data to the flash media the endurance of the device is improved. In addition, data written to the journal can be easily recovered in case of a system crash.

- Optimize the performance of the garbage collection mechanism by sending cold data to the LFS partition and by grouping blocks with similar hotness to the same segment.

- Use the journaling mechanism in order to ensure consistency and improve the recovery point objective of the log-structured file system.

- Improve the file system's segment utilization by batching synchronous block updates into the journal before writing them to the LFS. Since multiple block updates are written into the same segment, the overall disk utilization is increased.

4.3 Proposed Design

Flash storage in Metis is organized into two partitions: The journal partition and the LFS partition. The LFS partition stores the permanent state of written pages into a segmented log while the journal partition temporarily organizes data and metadata writes in the form of transactions. Even though journaling and LFS are already known individually, combining them into the same file system is novel and has numerous advantages.

4.3.1 Proactive Cleaning

In the past, journaling has been extensively used for fast metadata recovery from transient system failures. In our design however, the journal is designed to undertake the additional responsibility of proactively cleaning the permanent state from frequently updated data

and metadata. Written pages first reach the system cache. Flush daemons periodically transfer recently modified pages to the journal, where the pages are safe if a failure occurs. If a written page remains unmodified in memory until a timer expires, we transfer the page to LFS. Metis then informs the flash storage (e.g. by using the TRIM command) to erase those journal blocks, whose pages have been either transfered to LFS or invalidated by newer updates.

Instead of using complex methods to measure the hotness of each block, our system relies on cache timers to categorize pages to hot or cold and store them into the journal or LFS accordingly. The blocks in the LFS partition mostly contain valid pages, while the journal partition has valid blocks at the front part of the log and clean blocks at the rest. Journal transactions are written to/removed from the disk sequentially while the journal log is treated by the system as a circular buffer. Sequential workloads are beneficiary for the underlying media as they increase write performance and also reduce the device wear-out.

Arguably, writing all modified data first to the journal and subsequently to the LFS doubles the device traffic for sequential workloads. However, writes to the journal are relatively cheap. In addition, a page remapping scheme [10, 67] can be utilized in order to reduce the writing costs and improve performance.

The journal partition can also be hosted on a different device such as a hard disk drive in order to reduce write traffic to the SSD. In this case, write requests are firstly directed to the HDD before writing them to their final location in the SSD. Since hard disk drives are more resilient than SSDs the lifetime of the device is improved however, writes perform poorer than before. Read performance on the other hand remains optimal since read requests are serviced either by the host memory or the SSD and therefore the slow speeds of the HDD are completely avoided.

4.3.2 Garbage Collection

Log-Structured file systems are hurdled by garbage collection overheads. Cleaning occurs when the underlying storage capacity has been filled up and scattered blocks have to be reclaimed in order to obtain free segments for further writing. It is a very time consuming process that interrupts the normal disk operation and therefore, no new write requests can

be serviced. The garbage collector is responsible for selecting the appropriate segments for cleaning and copy their valid blocks into a new segment. It then informs the system of the newly reclaimed available space and normal operation is resumed. Making garbage collection executions faster or less frequent is critical to the performance of a log-structured file system. As a result, various cleaning algorithms have been proposed in the literature over time [57, 43, 32].

When selecting segments for cleaning, many garbage collection algorithms opt for segments with the smallest number of valid blocks, in hope of reclaiming as much space as possible with the least copy-out overhead. Such policies however, have shown to be ineffective since they do not consider the hotness of the data blocks contained within the reclaimed segment. Let us suppose for example that a segment that contains a large amount of frequently updated data is reclaimed by the garbage collector along with other segments that mostly contain cold data. After a short period of time the hot data in the newly written segment are updated and as a result a number of blocks within the segment are invalidated. The garbage collector would then have to be called again, in order to reclaim the blocks that were recently invalidated which results to a significant reduction in the performance of the system.

In order to reduce the cleaning overheads, modern garbage collection algorithms consider the hotness of the blocks contained within a segment and select those whose contents remain unchanged for a long time. In addition, it has been proven that grouping blocks with similar hotness into the same segment increases the effectiveness of garbage collection since the hotness of each segment can be calculated more accurately while the underlying flash media are also benefited [47].

Traditional Log-Structured file systems do not put any effort to distinguish hot and cold data or to group blocks with similar hotness into the same segment. Our design on the other hand, writes only cold data into LFS segments while hot data are placed on the journal where pages can be simply invalidated in the presence of updates. In addition, blocks are written to the LFS in a less-recently-modified order. As a result, blocks that remained unmodified are written together in the first same segment, while blocks that were recently updated are written into the last segment. The performance of the garbage collection mechanism is therefore improved, while cleaning also takes place less frequently since data on the LFS segments is not likely to be updated for a long time.

4.3.3 Crash Recovery

During normal operation, Log-Structured file systems buffer writes in a segment, write the segment to the disk and update the checkpoint region periodically (e.g. every 30 seconds or so) to ensure consistency. When recovering from a system crash, the checkpoint region is read and the last valid state of the file system is reconstructed. The system then tries to recover segments written after the latest checkpoint, a process known as roll-forward.

LFS places all metadata information right after writing all data blocks and just before updating the checkpoint region. As a result, in case of a system failure any changes in metadata written after the latest checkpoint are lost. During the process of roll-forward, LFS is able to recover only certain types of segments that contain data blocks written for data sync purposes. Therefore, the amount of data that would be lost in case of a system crash depends on the amount of pressure that was put into the system when the crash occurred and the time that had passed since the latest checkpoint.

Metis is designed to significantly improve the recovery point objective of the system since all data and metadata changes are sent periodically (i.e. every 5 seconds) to the journal. In case of a system crash, the journal is replayed and all the necessary blocks are recovered and written to the LFS partition. Although the process is relatively slower than that of LFS since all recovered blocks have to be written in the form of segments and therefore some processing is required, the recovery procedure is efficient and independent of the size of the file system.

4.3.4 Synchronous Writes

Synchronous writes pose a major problem for Log-Structured file systems. A typical LFS writes data in the form of segments that cannot be modified after they are placed in their final location. The synchronous write mechanism submits write requests and waits for them to be completed before submitting the next ones. In order to service these requests, LFS allocates segments and writes the requested data to the disk. Unfortunately, only a small portion of the segment is used for storage purposes while the rest of it cannot accommodate any extra writes unless reclaimed by the garbage collector. If the synchronous requests are frequent, the capacity of the drive is going to be quickly reduced and the garbage collector would have to be called repeatedly in order to reclaim disk space.

Metis services synchronous writes in a far more effective way than that of traditional log-structured file systems. In the event of a synchronous write request, buffered data is committed to the journal a process which is both fast and efficient. The segment containing this data can be constructed at a later time when the data becomes cold. Moreover, if the data is updated within a certain time period, the write operation is avoided altogether.

Multiple synchronous requests can also be grouped together in the same segment which leads to an improvement of the drive's utilization. In addition, since the available space is now organized in a more efficient way, the garbage collection process would not be called as early and as frequent as in the original implementation and therefore the performance of the system is also improved.

4.4 Summary

As it is clear from the above analysis, traditional log-structured file systems do not take into consideration the idiosyncrasies of flash memory and are sub-optimal in terms of performance and disk utilization. Even though the introduction of the flash translation layer has helped things, the use of a flash-aware file system can greatly improve the effectiveness and endurance of the system. However, all flash-aware file systems to date either use complicated mechanisms to separate hot and cold blocks or sacrifice the endurance of the disk for performance.

In this thesis we propose Metis, a flash-aware file system that utilizes existing cache replacement mechanisms in order to provide hot/cold block separation. It also reduces garbage collection overheads and improves the disk utilization of traditional log-structured file systems. In the next chapter, we provide more details on the underlying architecture of our prototype system.

Chapter 5

System Prototype

5.1 Overview

We implement our Metis prototype based on the NILFS2 file system. The New Implementation of the Log-Structured File System (NILFS2) was developed by Nippon Telegraph and Telephone corporation in 2005 [31]. It is considered to be a state-of-the-art Log-Structured file system that applies modern file system techniques such as B-tree structures and continuous snapshotting.

More specifically, NILFS2 creates a number of read-only checkpoints periodically of per synchronous write basis that users are able to mount in order to restore files mistakenly overwritten or destroyed a few seconds ago. Although checkpoints have a short lifespan, users have the ability to convert certain checkpoints to snapshots which will be preserved for as long as required.

In Metis we fully preserve those features. In addition to that we (i) provide hot/cold block separation, (ii) increase the efficiency of the garbage collection, (iii) improve the recovery point objective of the system (i.e., the maximum targeted period in which data might be lost in case of a system crash), and (iv) improve disk utilization. In order to meet our design goals, the current implementation of Metis integrates the Linux Journaling Block Device Layer (JBD2) with the NILFS2 filesystem.

5.2 Background

5.2.1 Linux Journaling

The journal in JBD2 can be implemented either as a hidden *.journal* file in the root directory of the file system or as a standalone disk partition. It is treated by the system as a circular buffer, meaning that the space that was used in order to store the various data blocks can be reclaimed and reused once they are written to the final location on the disk.

By design, journaling adds a bit of work during updates to greatly reduce the amount of recovery time required in the event of a system failure. Since logging every kind of write operation results to a significant performance overhead, some file systems opt to log only updates in the metadata while others choose to log all data and metadata updates. JBD2 is build upon three essential units: Log records, atomic operation handles and transactions.

- *Log records:* A log record describes the low-level operation issued by the file system in order to update a single disk block. JBD2 stores the updated blocks by saving the entire modified block rather than the range of bytes that were actually modified. As a result, log records inside the journal can be represented as normal blocks and do not require special handling.

- *Atomic operation handles:* Atomic operation handles are used to denote a set of low-level operations that represent a single high-level system operation. If a system crash occurs, either the whole high-level operation is applied during the recovery process or none of its low-level counterparts is.

- *Transactions:* Multiple atomic operation handles are grouped into a single *transaction* for reasons of efficiency. All the log records of a transaction are stored in consecutive blocks in the journal and are managed by the JBD2 as a whole. Thus, blocks used by a transaction can be reclaimed only after all data included in its log records are committed to the file system. During its life, a transaction goes through the following states: T_RUNNING, T_FLUSH, T_COMMIT and T_FINISHED. When recovering from a crash, the system checks the status of each transaction and only recovers the complete ones.

5.2.2 Log-Structured File System

On-Disk Layout

As in a traditional LFS, the disk in NILFS2 is divided in a number of segments known as *Full Segments* preceded by a superblock (Figure 5.1). A full segment is the basic division and allocation unit and is addressable by its index number. Given that the block size in Linux is usually 4KB, a typical full segment by default has the ability to store a number of 2048 blocks and therefore has a size of 8MB. Once a full segment is written and even if a very small portion of it is used, it cannot be re-written or filled with extra information until it is reclaimed by the garbage collector.

A *Partial Segment* is the basic write unit of the file system. Each partial segment consists of *Segment Summary* blocks and *Payload* blocks and its size cannot be greater than that of a full segment[1]. The segment summary area stores information about how the partial segment is organized while payload blocks can store any type of data and metadata information required by the file system. Typical information stored within the segment summary area are the length of the partial segment, a pointer to the next segment, checksum information, a number of status flags etc.

One or more partial segments are used to construct a *Logical Segment*. A logical segment is the basic recovery unit of NILFS2 and represents the difference between two consistent states of the file system. Typically it contains a number of file blocks, file B-tree node blocks, inode blocks, inode B-tree node blocks and a number of blocks for finding

[1]Even though it is called a "partial" segment, it can end up having the size of a full segment thus filling it up completely.

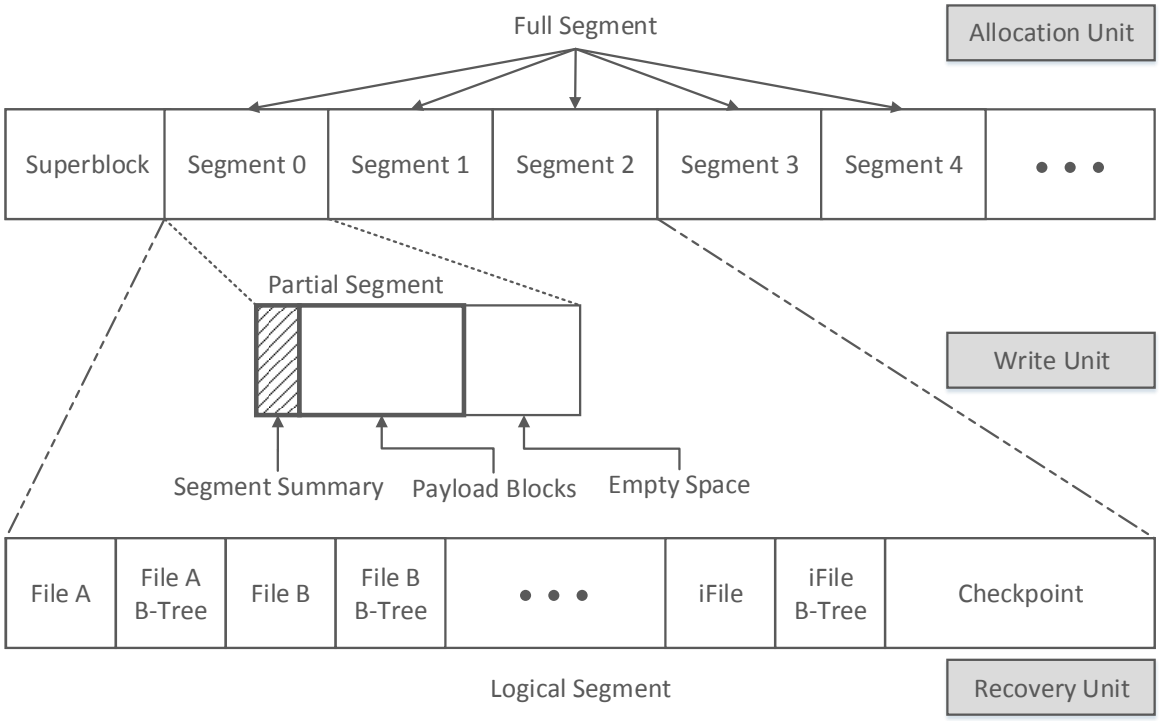


Figure 5.1: Allocation, Write and Recovery units of NILFS2

and managing checkpoints.

We now take a quick look on the various metadata structures used by the NILFS2 file system.

- *B-tree Inode Structures:* Inode structures for large files in NILFS2 are organized in B-tree form. B-tree structures have the main advantage of allowing for fast data block lookup since they remain balanced independently of the ordering or spacing of the keys inserted or removed.

- *Inode File:* Whereas in other file systems inodes are stored in fixed locations within the device, in NILFS2 the inode table is simply another file called the *iFile*. As in a regular file, the iFile has its own inode which describes the locations of the blocks that store the inode information.

- *Checkpoint File:* To record checkpoints and snapshots that are present in the file system, NILFS2 uses a special file known as the *Checkpoint File* (cpFile). Every time a checkpoint is created, a new record is added at the end of the cpFile that

includes (among others) the current timestamp and the inode for the iFile.

- *Disk Address Translation File:* Relocating a block that belongs to multiple snapshots can be quite difficult since the inode of every snapshot that the block belongs to would have to be updated to point to the new address of the block. This could result in a very intensive process since there can be tens or even hundreds of snapshots active at any given time. To counter this problem, NILFS2 adds a level of indirection by using a special file called the *Disk Address Translation* (DAT) file. The DAT file includes an array of 64-bit addresses that translates logical addresses used by the file system to actual device addresses.

- *Segment Usage File:* The *Segment Usage* (SU) file records the usage details of each segment. More specifically, it records when the segment was written and also the number of active blocks within the segment.

- *The Super Root:* The *Super Root* block is inserted at the end of every logical segment and includes three inodes: the inode for the DAT file, the inode for the cpFile and the one for the SU file. By using these three inodes, NILFS2 is able to find all the necessary structures needed for accessing and managing the files stored within the filesystem. A pointer to the latest super root block is kept in the superblock structure.

Segment Construction

Dirty blocks in NILFS2 are written to the disk in the form of segments by using the *Segment Constructor*. Instead, however, of planning the whole image of the logical segment and how it should be split in partial segments in order to write it to the disk beforehand, the segment constructor employs a special buffer called the *Segment Buffer*. The segment buffer has the capacity of a full segment and is used in order to build partial segments one by one rather than handling long logical segments that cross over multiple full segments.

To ensure that data blocks are placed in the correct order within the segment and to also make sure that a file has been completely updated before writing it to the disk, the original NILFS2 implementation uses a system of novel transactions. Only completed transactions are written to disk while several transactions can be nested inside of one

"outer" transaction. It is important to note through, that all nested transactions commit only when the outermost transaction is successfully completed.

Committing transactions are responsible for calling the segment constructor either when the number of dirty blocks in the dirty files list has exceeded the capacity of a full segment or 5 seconds after the first outermost transaction commits. In case of a write-intensive workload, the segment constructor is called to continuously flush dirty file blocks to the disk while the metadata files are written periodically every 30 seconds. If a crash happens in before writing the super root block, all data and metadata changes are lost.

5.3 Implementation

5.3.1 Journal Integration

In order to integrate the linux journaling block device layer with the NILFS2 file system, a number of steps had to be taken. We began by adding the following fields to the superblock of the system in order to store all the necessary information required by the JBD2 layer:

Listing 5.1: Added journal superblock fields

```
struct block_device *journal_bdev; /* Journal Block Device */
unsigned long ns_journal_devnum;   /* Journal Device Number */
struct journal_s * s_journal;      /* JBD Journal Structure */
unsigned long s_commit_interval;   /* Journal Commit Interval */
```

We then proceeded by adding a new option to the command line interface (`jdev`) so that users could specify the location of the journaling partition. Consecutively, we implemented a set of functions in order to load the journal during the mount process and initialize the JDB Journal Structure. More specifically the following functions were added to NILFS2:

Listing 5.2: Journal support functions

```
/* Open the external journal device */
```

```
struct block_device *nilfs_blkdev_get(dev_t, struct super_block)
/* Release external journal device */
void nilfs_blkdev_put(struct block_device)
/* Report and clear any journal errors */
void nilfs_clear_journal_err(struct super_block, struct the_nilfs)
/* Setup per-fs journal parameters */
void nilfs_init_journal_params(struct super_block, struct the_nilfs,
    journal_t)
/* Get journal structure from external device */
journal_t *nilfs_get_dev_journal(struct super_block, struct the_nilfs,
    dev_t)
/* Load the JBD2 journal structure on the NILFS2 superblock */
int nilfs_load_journal(struct the_nilfs, struct super_block)
```

The above functions issue calls to the appropriate JBD2 functions in order to load the journal during mount, detect any errors caused by a system crash and start recovery if necessary. They also release the journal during unmount and write a clean bill of health that can be used in order to detect whether a system failure occurred.

To achieve a seamless integration, the `load_nilfs` function of the original file system was modified in order to issue a call to `nilfs_load_journal`. This function in turn, calls the `nilfs_get_dev_journal` function to load the journal structure into the `s_journal` superblock field. To achieve this it uses `nilfs_blkdev_get`, which loads the block device descriptor on the `journal_bdev` field of the superblock. The system then checks whether the device was properly unmounted. If a crash took place, the recovery procedure of JBD2 is triggered, else the journal is erased and its parameters are initialized by calling the `nilfs_init_journal_params` function.

To properly unmount the system, the `nilfs_put_super` function was properly modified in order to destroy the journal structure by calling `jbd2_journal_destroy`. This function flushes any data blocks stored in the journal to its final location and writes a clean bill of health in order to indicate that the system was properly unmounted. The `nilfs_blkdev_put` function is then called to release the journal device.

5.3.2 Transactions

To ensure that data and metadata blocks are placed in the correct order within the segment and to also make sure that a file has been completely updated before writing it to the disk, the original NILFS2 implementation uses a system of novel transactions. Only completed transactions are written to disk while several transactions can be nested inside of one "outer" transaction. It is important to note through, that all nested transactions commit only when the outermost transaction is successfully completed.

The JBD2 layer also works by wrapping file system changes into transactions. More specifically two basic functions, `jbd2_journal_start` and `jbd2_journal_stop` are provided, which are used to indicate the beginning and the end of a transaction. When the `jbd2_journal_start` function is used, a transaction handle is returned that the system can use in order to indicate which blocks are going to be part of this transaction. The two transaction functions are nestable, however, each task can only have a single outstanding transaction at any one time and therefore, nothing commits until the outermost `jbd2_journal_stop`.

We implement a set of wrapper functions in order to call the `journal_start/stop` functions from the NILFS2 file system. The fact that NILFS2 already uses some sort of transaction mechanism allows us to conveniently place JBD2 transaction calls just outside NILFS2 transactions. We refrain from putting JBD2 transaction calls inside NILFS2 transactions in order to avoid deadlocks during the checkpoint process (discussed below).

A serious problem emerges however: both mechanisms use the `journal_info` field of the task scheduler in order to store and access their transaction information. Therefore, the two mechanisms cannot be used in parallel since the transaction information of the one mechanism will be overwritten by that of the other mechanism, resulting in a system crash. To solve this problem we added a new field to the superblock structure (`struct *nilfs_transaction_info ns_ti`) and modified the transaction mechanism of NILFS2 to store its transaction information into that field. As a result, a seamless integration of the two transaction mechanisms has been achieved.

5.3.3 Journaling Data

The modifications of each individual buffer need to be wrapped with calls to the JBD2 layer in order to let it know which blocks should be sent to the journal. In order to do this, JBD2 provides the following functions: `jbd2_journal_get_create_access`, `jbd2_journal_get_write_access`, `jbd2_journal_dirty_metadata`. Prior to modifying a buffer, a call to the `jbd2_journal_get_write_access` function is issued in order to ensure that the function has exclusive access of the buffer. In case of a newly created buffer, `jbd2_journal_get_create_access` is called instead. After the modification is complete, `jbd2_journal_dirty_metadata` is called in order to mark the buffer as ready to be committed to the journal.

The original NILFS2 implementation simply marked the blocks as dirty, when they had to be written back to disk. In case of a journaling file system however, marking blocks directly as dirty could lead to serious consistency problems as some of them could be written to the disk by the writeback mechanism before being committed to the journal. Instead, when a call to the `jbd2_journal_dirty_metadata` function is issued, buffers are marked as `JBDDirty` in order to be copied to the journal first.

To solve this problem, all buffer changes in our system were wrapped with calls to the appropriate JBD2 functions in order to mark them as `JBDDirty`. Even though NILFS2 transactions indicated most of the places where buffer changes were being made, we had to modify several generic functions that the original implementation used and replace them with our own customized implementations that added transaction support and/or calls to the appropriate functions in order to mark the buffers as JBDDirty. In particular, the following generic functions have been replaced: `generic_write_begin`, `generic_write_end`, `block_commit_write`, `block_write_end`, `walk_page_buffers`.

5.3.4 Commit

Dirty data and metadata blocks need to be flushed to the journal after a short period of time before being written to their final location on the disk, a process is known as the *commit phase*. A specialized kernel thread known as *kjournald* is employed by the JBD2 layer in order to trigger the commit phase periodically. The thread starts its execution when the journal is initialized and terminates when the journal is destroyed.

As already mentioned in the previous section, pages that contain dirty data that has not yet been written into the journal is marked as *JBDDirty*. During the commit phase, the buffers that were marked as JBDDirty are flushed into the journal in the form of transactions and the corresponding pages are marked as *Dirty*. Subsequently, dirty pages can be written to their final location by the writeback mechanism or by the checkpoint process.

The elapsed time between two commit phases is known as the *commit interval* and by default it is set to 5 seconds. A new option was added during mount (`jcinv`) so that users can adjust it according to their needs.

5.3.5 Checkpoint

Dirty data and metadata buffers in our system should be flushed into their final location on the disk in the form of segments after being committed to the journal. This process is known as the *checkpoint phase*, and in order achieve this, a mechanism used by the original implementation is employed known as the *Segment Constructor*. More specifically, the segment constructor scans the page cache for dirty pages, organizes them into segments, and writes them to their final location.

In the original NILFS2 implementation, committing NILFS2 transactions are responsible for calling the segment constructor, either when the number of dirty blocks has exceeded the capacity of a full segment or 5 seconds after the first outermost transaction commits. In case of a write-intensive workload, the segment constructor is called to continuously flush dirty file blocks to the disk, while the metadata files are written periodically every 30 seconds. We modify this behavior, in order to trigger the checkpoint process in two cases:

- Each JBD2 transaction requests a number of guaranteed buffers in order to start the logging operation. If the number of free buffers is smaller than the number of buffers requested, checkpointing needs to be performed in order to reclaim space. The amount of free space in the journal is checked every time a write request is completed. If the ratio of free journal space is less than 1/4 of the journal size, the segment constructor is called to start the checkpoint process.

- After completing the first write request, a timer is started in our system. When

the timer expires the checkpoint process is triggered since the data that are stored in the journal can now be characterized as cold. By default the timer is set to 30 seconds, but users can change its value during mount by using the new `cinv` option that we have implemented.

Calling the segment constructor unconditionally can result to serious problems if the commit phase is initiated by kjournald while data is checkpointed. Therefore we modified the main loop of the segment constructor by adding calls to functions that prevent any journal updates while checkpointing takes place (Listing 5.3). In particular, the `jbd2_journal_lock_updates` function blocks any further transactions from being started while running ones are waited upon completion. The NILFS locking transaction mechanism does have this feature where further updates are simply blocked and the execution continues. Had we selected to insert JBD2 transactions inside NILFS2 transactions, then a deadlock could be very easily caused since multiple transactions lie nested within one another. Finally, calls to the appropriate functions that remove written back buffers from the journal were added in order to reclaim journal space after the segment constructor finishes its work.

Listing 5.3: Main loop of the segment constructor

```
1  static void nilfs_segctor_thread_construct(struct nilfs_sc_info *sci,
2                                              int mode)
3  {
4    struct nilfs_transaction_info ti;
5    struct the_nilfs *nilfs = sci->sc_super->s_fs_info;
6
7    if(nilfs_doing_construction())
8      return;
9    /* Lock journal updates */
10   jbd2_journal_lock_updates(nilfs->s_journal);
11   /* Lock NILFS2 updates */
12   nilfs_transaction_lock(sci->sc_super, &ti, 0);
13   /* Start segment constructor */
14   nilfs_segctor_construct(sci, mode);
15   /* Start the timer for unclosed segment */
16   if (test_bit(NILFS_SC_UNCLOSED, &sci->sc_flags))
```

```
17      nilfs_segctor_start_timer(sci);
18    /* Unlock NILFS2 updates */
19    nilfs_transaction_unlock(sci->sc_super);
20    /* Unlock journal updates */
21    jbd2_journal_unlock_updates(nilfs->s_journal);
22    /* Remove written back buffers from the journal */
23    spin_lock(&nilfs->s_journal->j_list_lock);
24    __jbd2_journal_clean_checkpoint_list(nilfs->s_journal);
25    spin_unlock(&nilfs->s_journal->j_list_lock);
26
27    mutex_lock(&nilfs->s_journal->j_checkpoint_mutex);
28    jbd2_cleanup_journal_tail(nilfs->s_journal);
29    mutex_unlock(&nilfs->s_journal->j_checkpoint_mutex);
30  }
```

It is important to note that only committed buffers are written to the LFS partition. As we have already mentioned, pages that contain data that has not yet been committed to the journal is marked as JBDDirty in the page cache. During the segment construction process, the segment constructor scans the page cache for *dirty* pages in order to write them to the LFS partition. As a result, pages that are marked as JBDDirty are not written to the LFS.

This approach makes sense since data that were recently updated and have not been committed to the journal are considered hot and should not be written to their final location. Instead, they will be written to the journal and transfered to the LFS if and only if they remain unchanged after next commit in which they will be marked as dirty.

Each time a file is modified or when a new file is created in NILFS2, its inode is added to the *dirty files list*. When the segment constructor needs to search for dirty buffers, in uses the dirty files list in order to search the proper address space and avoid exhaustive lookups over the page cache.

Inodes in the dirty files list are organized in a first-modified fashion. If a file is modified and its inode is already on the dirty files list, it is not inserted again. Given that files are written into segments in the order they appear on the dirty files list, it would be a good idea to organize the inodes in a less-recently-modified fashion. That way, files that have remained unchanged will be written together in the first segments while hot files will

probably be written on the last segment. This approach increases the efficiency of the garbage collection mechanism since blocks are grouped into segments according to their hotness.

5.3.6 Failure Recovery

In the original NILFS2 implementation recovering from a system crash is a fast and efficient process. Initially, the superblock is read and the location of the last valid super root block is found. If the superblock is corrupted, information is taken from the secondary superblock which is kept in another part of the disk. After finding the super root it is easy to locate all the necessary files in order to mount the file system in a consistent state. NILFS2 then tries to recover segments written after the latest super root block, a process know as *roll forward*. It begins by reading the segment summary of the last segment where a pointer to the next segment is stored. It then reads the next segments and tries to recover the data from them however, it is able to recover *only* data sync segments since the metadata files have been lost.

Our system retains the original recovery mechanism of NILFS2 that stores and mounts the last consistent state of the device and improves it using journaling techniques. More specifically, after a system crash occurs the superblock is read and the system is mounted in a consistent state. Metis then replays the journal in order to recover all data and metadata blocks that were stored in it. However at this point, a serious problem emerges. For every block that is committed to the journal, its corresponding block number is saved in a special descriptor block. Traditional journaling file systems use this number in order to recover the block and copy it to its final location on the disk. In Log-Structured file systems the position of the blocks is "transiently indeterminable" and cannot be decided until just before writing them since multiple file updates are going to be grouped together and written into a segment. As a result we are unaware of where to copy the block.

To counter this problem, we modified the journaling mechanism provided by the JBD2 layer so that the number of the block's inode is saved within the journal descriptor block. Since the inode numbers of metadata files are statically defined, we can easily distinguish between data and metadata block updates. We also modified the original recovery mechanism of the JBD2 layer. In case of a system crash the blocks stored in the journal

are recovered and then by using a set of high level operations they are written to the NILFS2 partition. Metadata block updates are written first while data blocks follow. Since the journal has not yet been loaded, the writing procedure is the same as in the original NILFS2 implementation. Finally, the segment constructor is started in order to collect the data and write them to their final location on the disk. Although the recovery procedure in Metis is relatively slower than that of NILFS2 since the segment constructor is involved, it remains efficient and independent of the size of the file system.

5.3.7 Synchronous Writes

Synchronous writes pose a major problem for Log-Structured file systems. As in a typical LFS, NILFS2 writes data in the form of segments that cannot be modified after they are placed in their final location. The synchronous write mechanism submits write requests and waits for them to be completed before submitting the next ones. In order to serve these requests, NILFS2 triggers the segment construction mechanism. Depending on the nature of the file update, either a logical segment is constructed that includes all the necessary data and metadata of the file, or a *data sync segment* can be created. A data sync segment is a special type of segment that is written only when the file's blocks are updated while the inode remains unchanged. Dsync segments only contain the updated file blocks and do not contain any metadata.

Constructing a segment for syncing purposes negatively impacts the utilization of the disk since a small amount of data blocks have to be written to a full segment. In our system, we modify the behavior of the original mechanism in order to trigger the commit phase by invoking calls to the `jbd2_journal_force_commit` function. During the commit phase, all data and metadata updates are sent to the journal where they are safely stored in case of a system crash. The segment containing those data can be constructed at a later time when the data becomes cold while multiple synchronous requests can also be grouped together in the same segment which leads to an improvement of the drive's utilization.

5.4 Summary

In this chapter, the underlying architecture of our prototype system was presented. We have based our Metis implementation on the NILFS2 file system and the Journaling Block Device layer. The internal operations of both systems were discussed in detail and a number of changes that had to be done in order to create our composite file system were presented.

As we can see, the integration of journaling with the log-structured file system poses many challenges especially with the checkpoint and recovery procedures. However, by adding the proper functions and by precisely modifying the operation of existing mechanisms we are able to successfully overcome any problems and create a prototype system that offers the numerous advantages of Metis.

Chapter 6

Experimental Results

6.1 Experimentation Environment

We implemented Metis in the Linux kernel version 3.14.17. We evaluate our prototype implementation using x64-based server nodes running the Debian distribution. For all of our experiments we used nodes with two 2.33GHz processors and 4GB of RAM Memory. The majority of our experiments were run on a SATA 7200RPM disk with a capacity of 250GB and 16MB of on-disk cache. A solid state drive with a capacity of 60GB was also used to run a smaller set of experiments. In both disks the write cache was enabled in order to take advantage of the increased performance.

In all of the described scenarios, two separate disk partitions were used: one for the journal and the other for the LFS partition. The default parameters of linux were kept and the block size was defined to 4K. In addition, the writeback and expiration period as well

as the dirty pages ratio of the virtual memory manager were tuned for best performance. All write requests issued to the system where within the range of 1K-4K.

Three different schemes were tested: one that used hard disks for both the journal and the LFS partition, one that placed the LFS partition in a solid state drive and the journal partition in a hard drive and one that placed both in a solid state drive. The journal partition in our setup had a size of 2GB and in the first two cases it was hosted by the same disk that hosted the root partition. This configuration was selected in order to decouple the journal traffic from the traffic directed to the LFS partition.

In our evaluation, we used the *fio* micro-benchmark tool [1] in order to stress the system under different workloads. In addition, we experimented with different checkpoint intervals. We examine the number of Input/Output Operations Per Second (IOPS), the average latency, the number of segments constructed in the LFS partition and the amount of traffic directed to the disk and analyze the results.

6.2 Asynchronous Writes

We began by evaluating the performance of our system in the case of asynchronous write requests. We experimented with three types of workloads: Sequential, Random and Random that followed a Zipfian distribution. The last type of workload was chosen in order to simulate a case with frequent block overwrites.

For each workload a number of checkpoint interval was tried and tested in order to better understand the impact of the checkpoint phase to the performance of the system. More specifically, intervals of 10, 30 and 60 seconds were evaluated. A special case in which the checkpoint interval was set to a very high value (marked as Inf in our experiments) was also tested. The Inf checkpoint interval was chosen in order to check our system under the optimal case where no checkpoints occur and all updates are sent to the journal. Finally, the size of the dataset was selected to be 1GB as we did not wanted to push the system to its limits rather than investigate its performance by experimenting with various options.

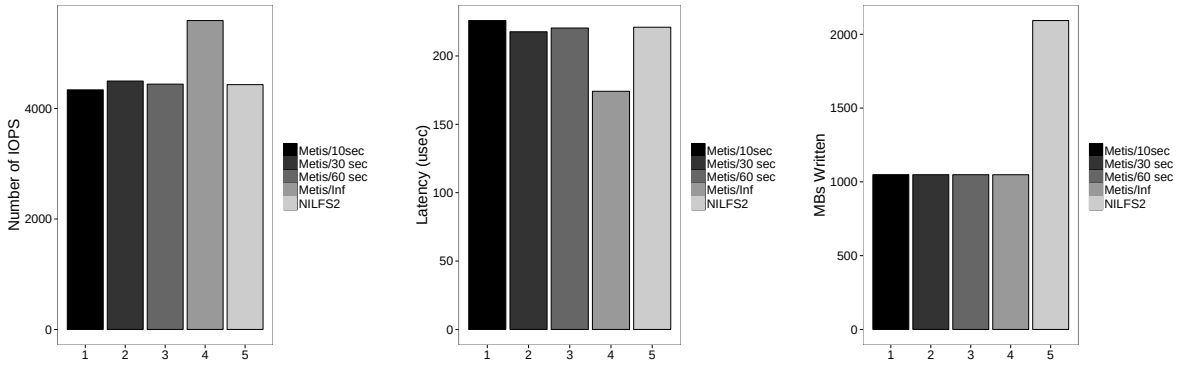


Figure 6.1: Sequential writes tests on disk-based configuration.

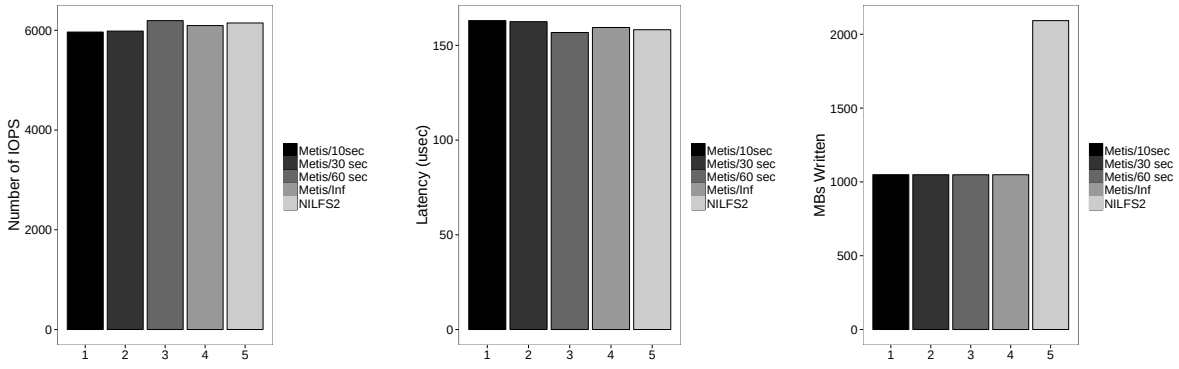


Figure 6.2: Sequential writes tests on hybrid configuration.

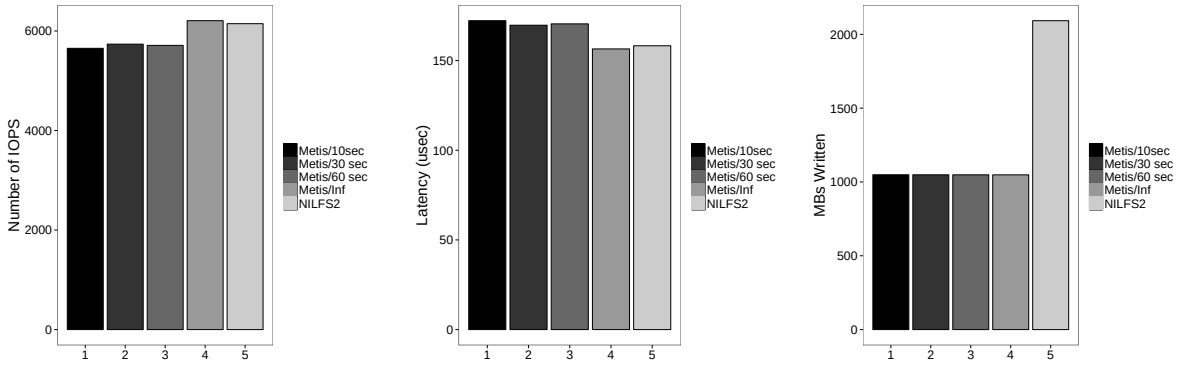


Figure 6.3: Sequential writes tests on ssd-based configuration.

6.2.1 Sequential Workloads

Our first experiment writes updates to the disk sequentially. We run the experiment on the disk-based, hybrid and ssd-based schemes and examine the number of IOPS, the

average latency and the number of MBs written to the LFS partition. The results are illustrated in Figures 6.1, 6.2 and 6.3 respectively.

As we can see, performance and latency in the first two configurations remain comparable to that of the original system. In addition, the performance in case of the disk based configuration that uses the Inf checkpoint interval is increased by about 20% while the latency also drops by 22%. In the ssd-based configuration performance drops slightly except from the case where the Inf checkpoint interval is used. When it comes to the amount of traffic written to the disk, our system achieves optimal performance and essentially halves the number of MBs written to the disk.

6.2.2 Random Workloads

Uniform Distribution

We then proceed to workloads that write data to the disk in a random manner. We expect the performance of our system to be lower than that of the traditional LFS in the first two schemes due to the way the original implementation applies the random updates. In addition, we anticipate that the drop will be much higher in the hybrid configuration, since the original implementation writes directly to the solid state drive while our system has to direct block updates to the disk-based journal first. Experimental results shown in Figures 6.4, 6.5 and 6.6 confirm our suspicions. Performance in the disk based scheme is decreased by about 10%, while in the hybrid configuration it is decreased by 36%. However, performance in the ssd-based scheme remains comparable to that of the original file system and increases as the checkpoint interval is increased. We believe that this behavior is a result of the sequential way that updates are written to/removed from the journal. Latency in the first two schemes is increased by 18% and 38% respectively while for the third scheme it is reduced by 10%.

The amount of MBs written to the disk on the other hand, is reduced while the checkpoint interval increases. In the optimal case traffic is reduced by 82% in the disk-based configuration, by 60% in the hybrid scheme and by 76% in the ssd-based configuration. When using a modest checkpoint interval value such as 30 seconds, it is reduced by 48%, 64% and 59% respectively. In general, the performance of random workloads in our system is slightly worse than that of the original implementation in the disk configuration

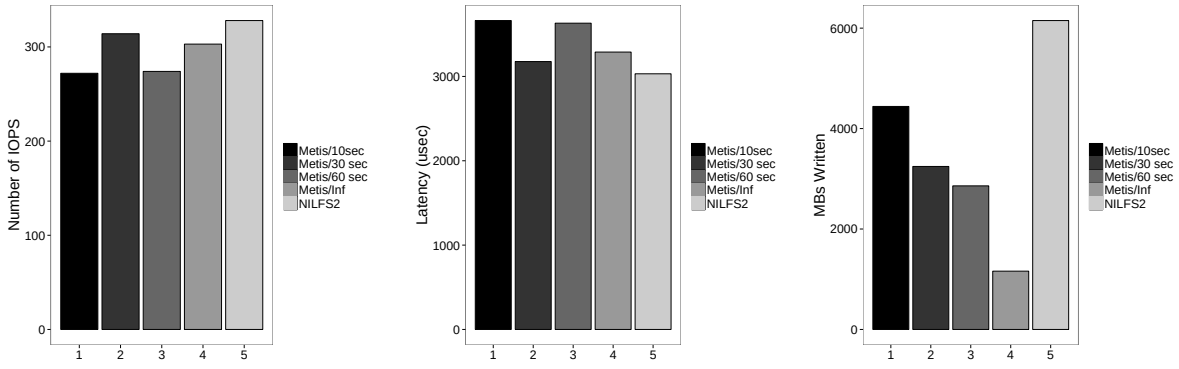


Figure 6.4: Random writes tests on disk-based configuration.

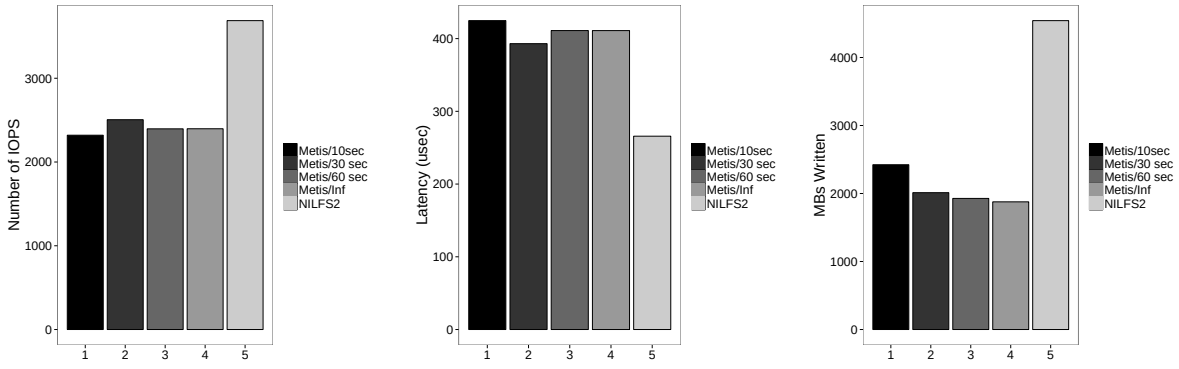


Figure 6.5: Random writes tests on hybrid configuration.

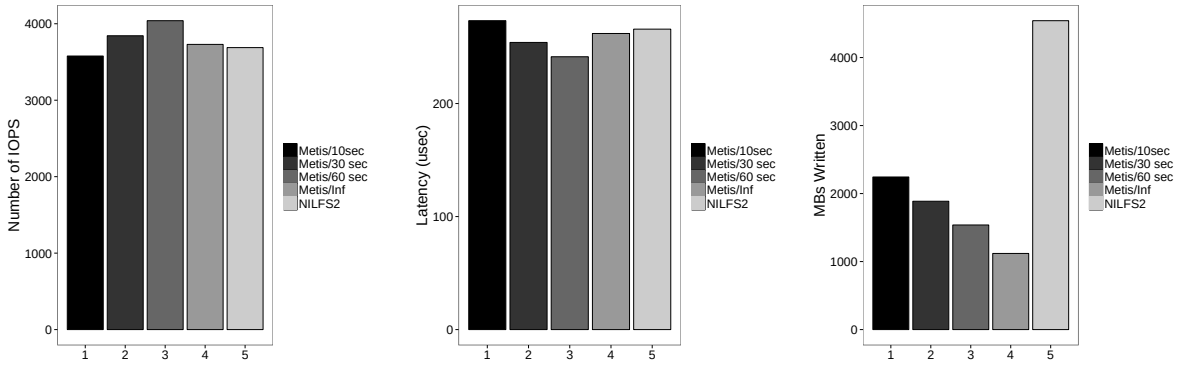


Figure 6.6: Random writes tests on ssd-based configuration.

and slightly better is the ssd configuration. In the hybrid configuration the system is significantly slower however, our system greatly improves the disk utilization.

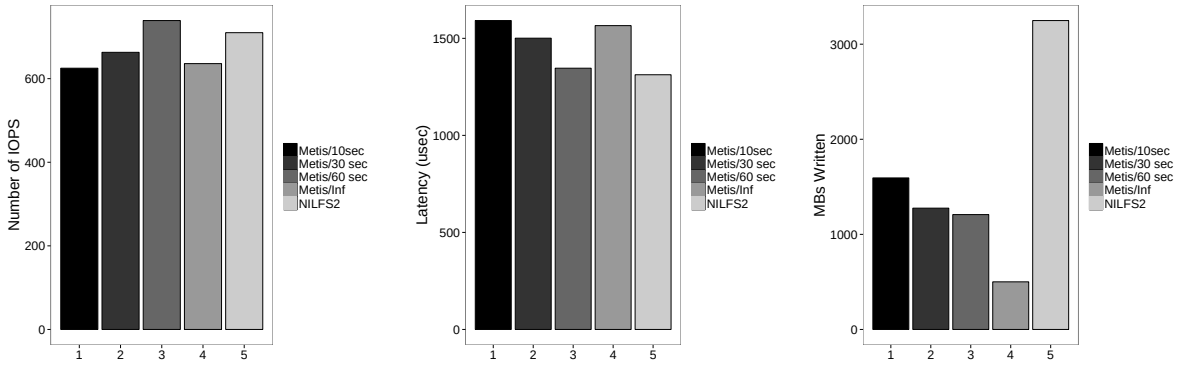


Figure 6.7: Random writes - Zipfian distribution tests on disk-based configuration.

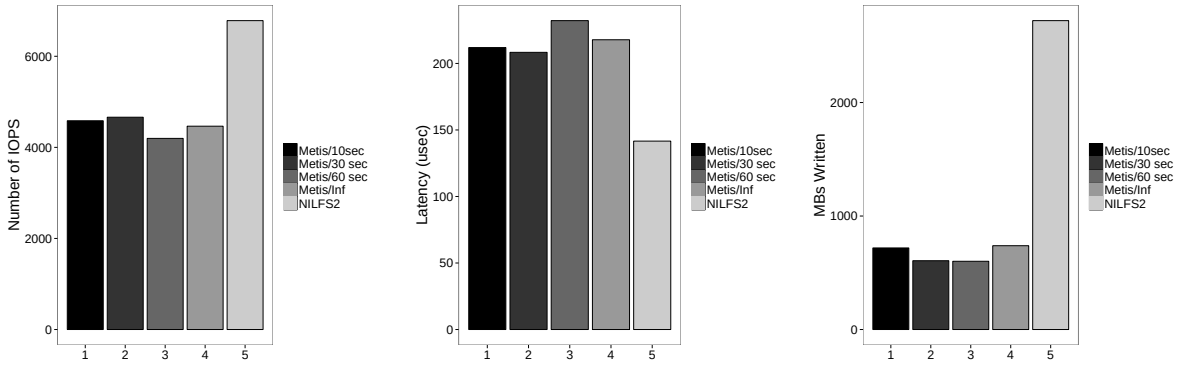


Figure 6.8: Random writes - Zipfian distribution tests on hybrid configuration.

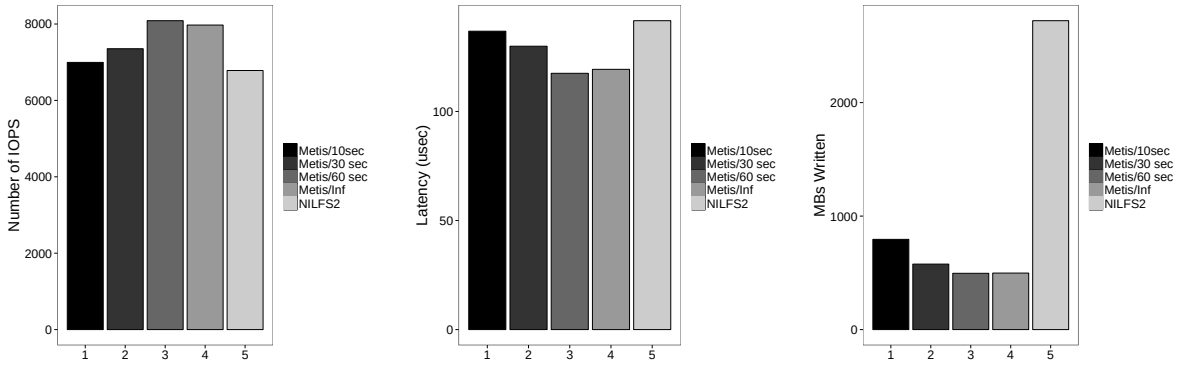


Figure 6.9: Random writes - Zipfian distribution tests on ssd-based configuration.

Zipfian Distribution

Since we have examined the performance of our system under a purely random workload, we now need to investigate what will happen in the case of frequent block overwrites.

In order to do this, we test our system by using a random workload that follows the zipfian distribution. We expect that some of the overwrites will be caught by the journal and therefore the amount of traffic that will be directed to the LFS partition will be considerably lower that that of the original implementation. The results are shown in Figures 6.7, 6.8 and 6.9. Performance shows a similar behavior as before however this time the amount of disk space used by all three schemes is reduced dramatically. In the optimal case the amount of traffic in the hybrid scheme is reduced by 78% while by using a 30 seconds commit interval it is reduced by 73%. We suspect that the reason behind this drop is because the hybrid scheme is able to apply more data updates within a given time period due to the use of the SSD. As a result, more overwrites can be "caught" by the journal before the checkpoint process begins. In the ssd configuration traffic is reduced by up to 82% in the case of the Inf checkpoint interval.

To sum up our system offers the same performance as in the previous case where random workloads were used, which is lower than that of the original implementation in the first two schemes and a slightly better in the ssd scheme. The disk utilization on all three cases however, is significantly improved since less writes are directed to the LFS partition.

6.3 Synchronous Writes

We now evaluate the performance of our system in the presence of synchronous writes. In order to achieve this, we use a sequential workload that synchronizes the data after a number of write requests. We experiment with the number of requests and examine three cases: When synchronization occurs every 1 write request, every 10 and every 100 requests. Once again we compare our system with the original implementation in both disk-based and hybrid configurations and examine the number of IOPS, the average latency and the amount of traffic directed to the LFS partition. The results for the disk-based scheme are shown in Figure 6.10, for the hybrid scheme in Figure 6.11 and for the ssd-based scheme in Figure 6.12.

As we can see performance increasingly drops as the number of writes requests between synchronous requests decreases. The average latency on the other hand, remains lower

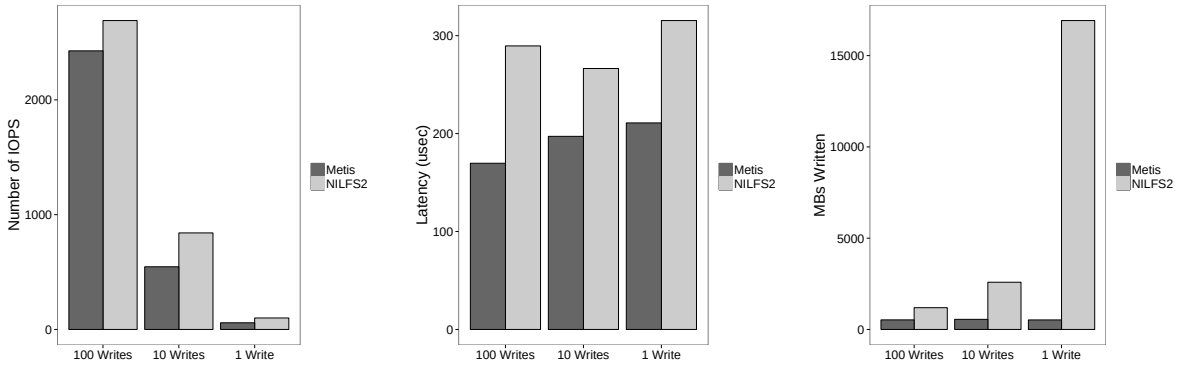


Figure 6.10: Synchronous Writes - Sequential tests on disk-based configuration.

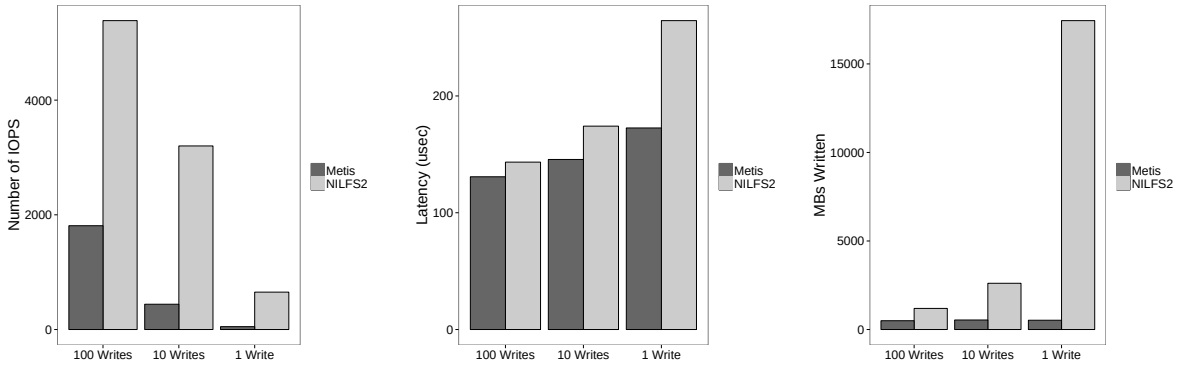


Figure 6.11: Synchronous Writes - Sequential tests on hybrid configuration.

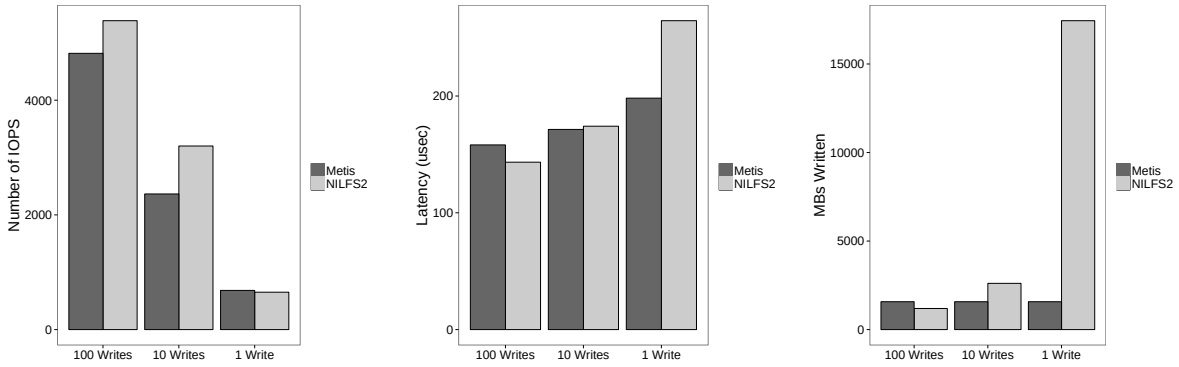


Figure 6.12: Synchronous Writes - Sequential tests on ssd-based configuration.

than that of the original implementation and increases much slower. In case the case of 1 write per synchronous request in particular, the average latency is decreased by 35%. Disk utilization remains optimal and dramatically improves that of the original NILFS2 file system. In the case of 10 writes per synchronous request, the average traffic is reduced

by 80% while in the case of 1 write per request it is reduced by 99.4%. In the general case our system improves the latency of the original implementation, and dramatically improves the disk utilization however, the performance of the system is reduced, especially when a hybrid configuration is used.

6.4 Summary

We evaluate our system by using the fio micro-benchmark tool in order to stress our system under different workloads. We examine the number of Input/Output Operations Per Second (IOPS), the average latency, the number of segments constructed in the LFS partition and the amount of traffic directed to the disk and analyze the results. Three different schemes were tested depending on the type of media that the journal and the LFS partition resided in.

Results show that our system offers performance comparable to that of the original file system with sequential workloads, while improving the overall disk utilization. Random workloads negatively impact the performance of our system, especially in the hybrid configuration mode however, the write traffic directed to the LFS partition is reduced by up to 78%. Finally, our systems offers reduced latency and dramatically increases disk utilization by up to 99.4% with synchronous workloads, although at reduced performance.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

Asymmetric storage devices are becoming increasingly useful as either standard storage device in mobile systems or desirable storage layer in enterprise servers. Though such devices exhibit several attractive characteristics and are able to outperform their mechanical counterpart by even by an order of magnitude when it comes to random workloads, they have several limitations such as the erase-before-write requirement and limited block endurance. As a result, their performance and lifetime is highly workload-depended.

Researchers have tried to solve this problem by adding a level of indirection between the device and the file system known as the flash translation layer. This solution however, is not able to utilize application semantic information in order to exploit the intrinsic characteristics of asymmetrical storage to its advantage. As a result, a number of flash-aware file systems have been suggested that help optimize the usage of flash media. Still, these file systems either use complicated mechanisms to identify hot and cold data in memory or exchange the dependability of the drive for performance.

In this thesis we proposed *Metis*, a composite flash-aware file system that combines journaling with the log-structured file system. Our work is based on previously published work of our group [19, 20]. By using a set of novel techniques that add minimal writing overhead, our system is able to provide hot/cold block separation, increase the efficiency of the garbage collection mechanism, reduce the recovery point objective of the system, and improve disk utilization.

We implemented Metis in the Linux kernel version 3.14.17. We evaluated our prototype implementation by using a set of micro-benchmark tools that stressed our system under different types of workloads. In addition, we tried and tested three different setup schemes: One that used hard disks for both the journal and the LFS partition, one that placed the journal partition in a hard disk drive and the LFS partition in a solid state drive and one where both partitions resided in an SSD. Results show that our system offers performance comparable to that of the original file system with sequential workloads, while improving the overall disk utilization. Random workloads on the other hand negatively impact the performance of our system in the disk-based and hybrid configuration modes. However, the performance of the ssd-based configuration is slightly improved while write traffic directed to the LFS partition is reduced by up to 78% for all three schemes. Finally with synchronous writes, our system offers reduced latency and dramatically increases disk utilization by up to 99.4% although at reduced performance.

7.2 Future Work

There are many directions for future work, especially regarding the performance evaluation of our implementation. In the future we plan to extend our experiments in order to stress our system using real-world application workloads. Such tests would allow us to study the behavior of our system under extensive pressure and discover its advantages and weaknesses. In addition, a configuration during which synchronous writes over the same blocks are applied should be tested, in order to investigate the capability of our system to "catch" synchronous overwrites in the journal and not send them to the LFS partition. Finally, the effect of the journal partition size should be tested in order to get a clear view of its role in the block separation process.

Bibliography

[1] Fio micro-benchmark tool. `http://freecode.com/projects/fio`. Accessed: 2015-06-05.

[2] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D Davis, Mark S Manasse, and Rina Panigrahy. Design tradeoffs for ssd performance. In *USENIX Annual Technical Conference*, pages 57–70, 2008.

[3] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 0.80 edition, May 2014.

[4] Trevor Blackwell, Jeffrey Harris, and Margo I Seltzer. Heuristic cleaning algorithms in log-structured file systems. In *USENIX*, pages 277–288, 1995.

[5] Daniel P Bovet and Marco Cesati. *Understanding the Linux kernel.* " O'Reilly Media, Inc.", 2005.

[6] Joe Brewer and Manzur Gill. *Nonvolatile Memory Technologies with Emphasis on Flash: A Comprehensive Guide to Understanding and Using Flash Memory Devices*, volume 8. Wiley. com, 2011.

[7] Neil Brown. A NILFS2 score card. `https://lwn.net/Articles/522507/`. Accessed: 2015-05-07.

[8] Giovanni Campardo, Rino Micheloni, and David Novosel. *VLSI-design of non-volatile memories*. Springer, 2005.

[9] Feng Chen, David A Koufaty, and Xiaodong Zhang. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In *ACM SIGMETRICS Performance Evaluation Review*, volume 37, pages 181–192. ACM, 2009.

[10] Hyun Jin Choi, Seung-Ho Lim, and Kyu Ho Park. Jftl: A flash translation layer based on a journal remapping for flash memory. *ACM Transactions on Storage (TOS)*, 4(4):14, 2009.

[11] Michael Cornwell. Anatomy of a solid-state drive. *Commun. ACM*, 55(12):59–63, 2012.

[12] Christian Czezatke and M Anton Ertl. Linlogfs-a log-structured file system for linux. In *USENIX Annual Technical Conference, FREENIX Track*, pages 77–88, 2000.

[13] Timothy E Denehy, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Bridging the information gap in storage protocol stacks. In *USENIX Annual Technical Conference, General Track*, pages 177–190, 2002.

[14] Eran Erez. Methods for optimizing page selection in flash-memory devices, April 28 2009. US Patent 7,525,870.

[15] Gregory R Ganger. *Blurring the line between OSes and storage devices.* School of Computer Science, Carnegie Mellon University, 2001.

[16] Dominic Giampaolo. *Practical file system design with the Be file system.* Morgan Kaufmann Publishers Inc., 1998.

[17] Garth Goodson and Rahul Iyer. Design tradeoffs in a flash translation layer. In *Proceedings of Workshop on the Use of Emerging Storage and Memory Technologies*, 2010.

[18] Aayush Gupta, Youngjae Kim, and Bhuvan Urgaonkar. *DFTL: a flash translation layer employing demand-based selective caching of page-level address mappings*, volume 44. ACM, 2009.

[19] Andromachi Hatzieleftheriou and Stergios V. Anastasiadis. Jlfs: Journaling the log-structured filesystem for proactive cleaning in flash storage. In *USENIX Annual Technical Conference (ATC)*, Portland, OR.

[20] Andromachi Hatzieleftheriou and Stergios V. Anastasiadis. Improving bandwidth efficiency for consistent multistream storage. In *ACM Transactions on Storage (TOS)*, volume 9, March 2013.

[21] Xiao-Yu Hu, Evangelos Eleftheriou, Robert Haas, Ilias Iliadis, and Roman Pletka. Write amplification analysis in flash-based solid state drives. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, page 10. ACM, 2009.

[22] Ping Huang, Guanying Wu, Xubin He, and Weijun Xiao. An aggressive worn-out flash block management scheme to alleviate ssd performance degradation. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 22:1–22:14, New York, NY, USA, 2014. ACM.

[23] Ting-Chang Huang and Da-Wei Chang. Tridentfs: a hybrid file system for non-volatile ram, flash memory and magnetic disk. *Software: Practice and Experience*, 2014.

[24] Yeonseong Hwang, Hyunho Gwak, and Dongkun Shin. Two-level logging with non-volatile byte-addressable memory in log-structured file systems. In *Proceedings of the 12th ACM International Conference on Computing Frontiers*, CF '15, pages 38:1–38:2, New York, NY, USA, 2015. ACM.

[25] Sooman Jeong, Kisung Lee, Seongjin Lee, Seoungbum Son, and Youjip Won. I/o stack optimization for smartphones. In *USENIX Annual Technical Conference*, pages 309–320, 2013.

[26] Xavier Jimenez, David Novo, and Paolo Ienne. Wear unleveling: improving nand flash lifetime by balancing page endurance. In *12th USENIX Conference on File and Storage Technologies (FAST 14)*, pages 47–59, 2014.

[27] William K Josephson, Lars A Bongo, Kai Li, and David Flynn. Dfs: A file system for virtualized flash storage. *ACM Transactions on Storage (TOS)*, 6(3):14, 2010.

[28] Dawoon Jung, Jeong-UK Kang, Heeseung Jo, Jin-Soo Kim, and Joonwon Lee. Superblock ftl: a superblock-based flash translation layer with a hybrid address translation scheme. *ACM Transactions on Embedded Computing Systems (TECS)*, 9(4):40, 2010.

[29] Sanghyuk Jung, Yangsup Lee, and Yong Ho Song. A process-aware hot/cold identification scheme for flash memory storage systems. *Consumer Electronics, IEEE Transactions on*, 56(2):339–347, 2010.

[30] Hyojun Kim, Nitin Agrawal, and Cristian Ungureanu. Revisiting storage for smartphones. *ACM Transactions on Storage (TOS)*, 8(4):14, 2012.

[31] Ryusuke Konishi, Yoshiji Amagai, Koji Sato, Hisashi Hifumi, Seiji Kihara, and Satoshi Moriai. The linux implementation of a log-structured file system. *SIGOPS Oper. Syst. Rev.*, 40(3):102–107, July 2006.

[32] Ohhoon Kwon, Kern Koh, Jaewoo Lee, and Hyokyung Bahn. Fegc: An efficient garbage collection scheme for flash memory based storage systems. *Journal of Systems and Software*, 84(9):1507–1523, 2011.

[33] Changman Lee, Dongho Sim, Jooyoung Hwang, and Sangyeun Cho. F2fs: A new file system for flash storage. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 273–286, Santa Clara, CA, February 2015. USENIX Association.

[34] Jongsung Lee and Jin-Soo Kim. An empirical study of hot/cold data separation policies in solid state drives (ssds). In *Proceedings of the 6th International Systems and Storage Conference*, SYSTOR '13, pages 12:1–12:6, New York, NY, USA, 2013. ACM.

[35] Sang-Won Lee, Dong-Joo Park, Tae-Sun Chung, Dong-Ho Lee, Sangwon Park, and Ha-Joo Song. A log buffer-based flash translation layer using fully-associative sector translation. *ACM Transactions on Embedded Computing Systems (TECS)*, 6(3):18, 2007.

[36] Sungjin Lee, Keonsoo Ha, Kangwon Zhang, Jihong Kim, and Junghwan Kim. Flexfs: A flexible flash file system for mlc nand flash memory. In *USENIX Annual Technical Conference*, 2009.

[37] Sungjin Lee, Dongkun Shin, Young-Jin Kim, and Jihong Kim. Last: locality-aware sector translation for nand flash memory-based storage systems. *ACM SIGOPS Operating Systems Review*, 42(6):36–42, 2008.

[38] Yan Li, Seungpil Lee, Yupin Fong, Feng Pan, Tien-Chien Kuo, Jongmin Park, Tapan Samaddar, Hao Thai Nguyen, Man L Mui, Khin Htoo, et al. A 16 gb 3-bit per cell

(x3) nand flash memory on 56 nm technology with 8 mb/s write rate. *Solid-State Circuits, IEEE Journal of*, 44(1):195–207, 2009.

[39] Seung-Ho Lim and Kyu-Ho Park. An efficient nand flash file system for flash memory storage. *Computers, IEEE Transactions on*, 55(7):906–912, 2006.

[40] Youyou Lu, Jiwu Shu, and Wei Wang. Reconfs: a reconstructable file system on flash storage. In *FAST*, pages 75–88, 2014.

[41] Youyou Lu, Jiwu Shu, Weimin Zheng, et al. Extending the lifetime of flash-based storage through reducing write amplification from file systems. In *FAST*, pages 257–270, 2013.

[42] Jeanna Neefe Matthews, Drew Roselli, Adam M. Costello, Randolph Y. Wang, and Thomas E. Anderson. *Improving the Performance of Log-structured File Systems with Adaptive Methods*. SOSP '97. ACM, New York, NY, USA, 1997.

[43] Jai Menon and Larry Stockmeyer. An age-threshold algorithm for garbage collection in log-structured arrays and file systems. In *High Performance Computing Systems and Applications*, pages 119–132. Springer, 1998.

[44] Rino Micheloni, Giovanni Campardo, and Piero Olivo. *Memories in wireless systems*. Springer, 2008.

[45] Rino Micheloni, Luca Crippa, and Alessia Marelli. *Inside NAND flash memories*. Springer, 2010.

[46] Rino Micheloni, Alessia Marelli, and Kam Eshghi. *Inside Solid State Drives (SSDs)*. Springer, 2013.

[47] Changwoo Min, Kangnyeon Kim, Hyunjin Cho, Sang-Won Lee, and Young Ik Eom. Sfs: random write considered harmful in solid state drives. In *10th USENIX Conference on File and Storage Technologies (FAST 12)*, page 12, 2012.

[48] Yongseok Oh, Eunsam Kim, Jongmoo Choi, Donghee Lee, and Sam H Noh. Optimizations of lfs with slack space recycling and lazy indirect block update. In *Proceedings of the 3rd Annual Haifa Experimental Systems Conference*, page 2. ACM, 2010.

[49] Alan R Olson and Denis J Langlois. Solid state drives data reliability and lifetime. *Imation White Paper*, 2008.

[50] Dongchul Park, Biplob Debnath, Youngjin Nam, David HC Du, Youngkyun Kim, and Youngchul Kim. Hotdatatrap: a sampling-based hot data identification scheme for flash memory. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, pages 1610–1617. ACM, 2012.

[51] Jung-Wook Park, Seung-Ho Park, Charles C Weems, and Shin-Dug Kim. A hybrid flash translation layer design for slc–mlc flash memory based multibank solid state disk. *Microprocessors and Microsystems*, 35(1):48–59, 2011.

[52] Sheng Qiu et al. Nvmfs: A hybrid file system for improving random write in nand-flash ssd. In *Mass Storage Systems and Technologies (MSST), 2013 IEEE 29th Symposium on*, pages 1–5. IEEE, 2013.

[53] Abhishek Rajimwale, Vijayan Prabhakaran, and John D Davis. Block management in solid-state devices. In *USENIX Annual Technical Conference*, 2009.

[54] Bruno Ricco, Gianfranco Gozzi, and Massimo Lanzoni. Modeling and simulation of stress-induced leakage current in ultrathin sio 2 films. *Electron Devices, IEEE Transactions on*, 45(7):1554–1560, 1998.

[55] Ohad Rodeh, Josef Bacik, and Chris Mason. Btrfs: The linux b-tree filesystem. *ACM Transactions on Storage (TOS)*, 9(3):9, 2013.

[56] Elyse Rosenbaum and Leonard F Register. Mechanism of stress-induced leakage current in mos capacitors. *Electron Devices, IEEE Transactions on*, 44(2):317–323, 1997.

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

[58] Andreas Schierl, Gerhard Schellhorn, Dominik Haneberg, and Wolfgang Reif. Abstract specification of the ubifs file system for flash memory. In *FM 2009: Formal Methods*, pages 190–206. Springer, 2009.

[59] Ilhoon Shin. An optimal ftl for ssds.

[60] Kent Smith. Understanding ssd over provisioning. *LSI Corporation,[online], http://www. edn. com/design/systems-design/4404566/Understanding-SSD-over-provisioning*, 2013.

[61] Gokul Soundararajan, Vijayan Prabhakaran, Mahesh Balakrishnan, and Ted Wobber. Extending ssd lifetimes with disk-based write caches. In *FAST*, volume 10, pages 101–114, 2010.

[62] Jun Wang and Yiming Hu. Wolf-a novel reordering write buffer to boost the performance of log-structured file systems. In *1st USENIX Conference on File and Storage Technologies (FAST 02)*, pages 47–60, 2002.

[63] Wei Wang, Tao Xie, and Deng Zhou. Understanding the impact of threshold voltage on mlc flash memory performance and reliability. In *Proceedings of the 28th ACM International Conference on Supercomputing*, ICS '14, pages 201–210, New York, NY, USA, 2014. ACM.

[64] Wenguang Wang, Yanping Zhao, and Rick Bunt. Hylog: A high performance approach to managing disk layout. In *3rd USENIX Conference on File and Storage Technologies (FAST 04)*, volume 4, pages 145–158, 2004.

[65] John Wilkes, Richard Golding, Carl Staelin, and Tim Sullivan. The hp autoraid hierarchical storage system. *ACM Transactions on Computer Systems (TOCS)*, 14(1):108–136, 1996.

[66] David Woodhouse. Jffs: The journalling flash file system. In *Ottawa Linux Symposium*, volume 2001, 2001.

[67] Yiying Zhang, Leo Prasath Arulraj, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. De-indirection for flash-based ssds with nameless writes. In *FAST*, page 1, 2012.

Appendix A

Appendix

Non-Volatile Memory

Semiconductor memories can be divided into two major categories: *Volatile memories* and *Non-Volatile memories*. Volatile memories can be programmed fast, easy and an unlimited number of times but require power to preserve the stored information. Non-Volatile memories are able to retain the stored information in the event of a power loss, but fall short in terms of speed and durability. Early designs of non-volatile memory were fabricated with permanent data and did not provide the ability to modify their content. Current designs allow their content to be erased and re-programmed a limited number of times although at a significantly slower speed compared to volatile memories [8, 6].

Read-Only Memory

The earliest form on non-volatile semiconductor memory was the Read-Only Memory (ROM). As the name implies, data stored in a ROM cannot be modified and even if they can, the process for altering the data is either too slow or too difficult.

In it's strictest sense, the term ROM refers to *Mask ROM* which is fabricated with the desired data permanently stored in it. In fact, the contents of a Mask ROM are programmed by the integrated circuit manufacturer and cannot be changed at all. The term "Mask" refers to the fabrication process where regions of the chip are actually masked of during the process off *photolithography*.

Mask ROM is the oldest type of ROM and while the process of creating a mask is costly, laborious and prone to errors, once it is completed chips are fabricated with a very low cost per bit, use very little power, are extremely reliable and are cheaper than any other kind of semiconductor memory.

Programmable Read-Only Memory

Creating ROM chips from scratch and for small quantities is very expensive and time-consuming. As a result, a new type of ROM was created in 1956 known as the *Programmable Read-Only Memory* (PROM). PROM chips are manufactured with all bits set to 1 and by using a special tool called *programmer*, selected bits can be changed from 1 to 0 and therefore store the desired data into the chip. Blank PROM chips can be bought inexpensively and coded by anyone with a programmer, which leads to a reduced manufacturing cost.

As illustrated in Figure A.1, a typical PROM chip consists of a grid of columns and rows. At every intersection of a column and a row, there is a fuse connecting the two lines. If a charge is sent through a column and passes through the fuse in a cell to a grounded row, it indicates a value of 1. The memory can thus be programmed by *Blowing* the fuses, which is an irreversible process. By blowing a fuse the connection opens and indicates a value of 0 to the user. Although it is physically impossible to un-blow a fuse, it is often possible to change the contents of the memory after initial programming by blowing additional fuses and change some of the remaining 1 bits to 0s.

Erasable Programmable Read-Only Memory

Even though the invention of PROM reduced the manufacturing cost for a ROM chip, it was not a flexible solution. Changes or updates to the stored data meant that a new chip had to be used, leading to an increase in cost. To put it differently even though the PROM was inexpensive per chip, the cost could add up over time if multiple chips had to be used. To address this issue a new type of ROM was invented in 1971, the *Erasable Programmable Read-Only Memory* (EPROM).

Much like the PROM an EPROM chip consists of a grid of rows and columns. The main difference is that instead of using a fuse to connect the two lines, a new type of

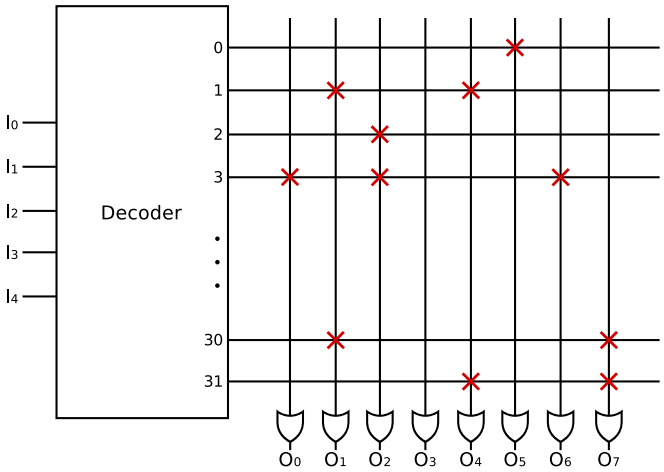


Figure A.1: A typical PROM. The red X marks a fuse which has been blown.

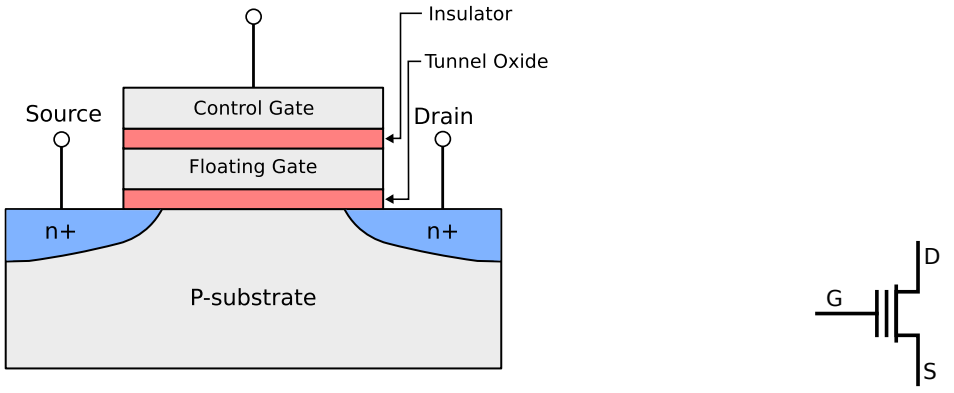
field-effect transistor is used, the *Floating-Gate Transistor*. When an EPROM chip is manufactured, all bits are set to 1. In order to change a transistor's state from 1 to 0 a special programmer is used that applies an electrical charge of 10 to 13 volts to the floating gate, a process known as *Fowler-Nordheim Tunneling*.

In order to erase an EPROM chip, a special tool that emits a certain frequency of UV light has to be used. It is a very sensitive and time consuming procedure during which specific rules have to be followed in order to prevent the chip from being destroyed[1]. To allow the exposure to UV Light during erasing, a transparent quartz window is installed at the top of the chip through which the silicon chip is visible.

Electrically Erasable Programmable Read-Only Memory

The invention of EPROM was a major step forward for non-volatile memories, especially with the introduction of the floating-gate transistor. However, it still required dedicated equipment to program the chips and a labor-intensive process to remove and reinstall them each time a change in the data was necessary. Moreover, one of the main disadvantages of EPROM was that changes could not be made incrementally and the whole chip had to be erased and reprogrammed. Therefore, a new type of ROM was invented in 1978, the *Electrically Erasable Programmable Read-Only Memory* (EEPROM).

[1]Typical setup: The chip has to be exposed to UV light at $253.7nm$ of at least $15W - \frac{sec}{cm^2}$ for about 20 to 30 minutes with a lamp at a distance of about $2.5cm$



(a) (b)

Figure A.2: A cross-section sketch of the Floating-Gate Transistor and its Circuit symbol.

As the term "Electrically Erasable" implies, EEPROMs can be erased and rewritten on-chip by using a high-voltage pulse. This new technology eliminated the biggest drawbacks of EPROM since the chip does not have to be removed to be rewritten, it does not have to be completely erased to change a specific portion of it and it does not require additional equipment in order to change its contents. The Write/Erase procedure on an EEPROM however, damages the layer of insulating material on the chip, so the number of Write/Erase cycles is limited. Early models would fail after tens or hundreds of thousand times, while modern EEPROMs can sustain one million Write/Erase operations or more.

Flash Memory

Flash memory is specialized descendant of EEPROM that is also based on floating-gate transistors that can be electrically erased and reprogrammed. Although EEPROM memories are versatile, their contents are changed 1 byte at a time making them extremely slow. Moreover, EEPROM uses two transistors per memory cell compared to EPROM that uses one therefore it is more expensive and its density is much lower.

To address this problem, Dr. Fujio Masuoka invented the Flash Memory technology in 1984. Flash memory provides a good compromise between EPROM and EEPROM. It uses a single transistor per cell providing low cost per bit and high density much like an EPROM while at the same time its contents can be electrically altered like an EEPROM.

Moreover, it uses in-circuit wiring to erase data by applying an electrical field to the entire chip, or to predetermined sections of the chip called *blocks* leading to a much faster design. Finally, it writes data in chunks (usually 512 bytes in size) instead of 1 byte at a time making it ideal for both code and data storage. In the following sections, we are going to examine the main components and architectures of modern flash memory.

Floating-Gate Transistor

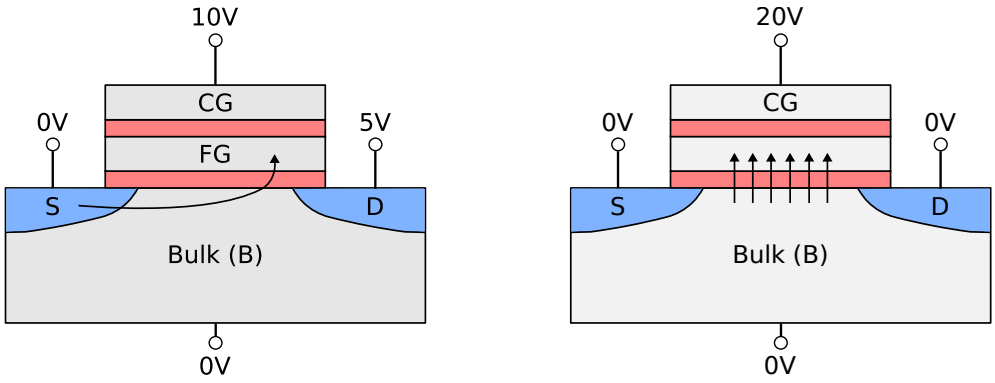
The heart of the flash memory cell is the *Floating-Gate Transistor*. Its structure is similar to a conventional MOSFET[2], with an additional gate added. The newly added gate called the *Floating Gate*, occupies the position of the original gate with the original gate (now known as the *Control Gate*) now being on top of the floating gate (Figure A.2). The floating gate is insulated all around by an oxide layer and as a result, any electrons placed on it are trapped there and - under normal conditions - will not discharge for many years [46]. It is because of this phenomenon, that the floating-gate transistor can be used as non-volatile memory. Let us now look at the basic operating principles of the floating gate transistor and then discuss its reliability.

Program

In order to program a floating-gate transistor two basic physical mechanisms are exploited, depending on the flash memory architecture: *Channel Hot Electron (CHE) Injection* and *Fowler-Nordheim (FN) Tunneling* [6, 44, 8, 45].

CHE Injection applies a relatively high voltage (between $4V$ and $6V$) to the transistor's Drain (D), a high voltage ($8V$ to $11V$) to the Control Gate (CG) while the Source (S) and Bulk (B) terminals are kept at $0V$. As a result, a large current (between $0.3mA$ to $1mA$) flows in the cell and the hot electrons generated in the channel acquire sufficient energy to jump the gate oxide barrier and get trapped into the Floating Gate (FG) (Figure A.3a). CHE Injection is typically used with NOR-type flash architectures and is a relatively fast procedure since it takes only a few microseconds to shift the threshold voltage from the

[2]Metal-Oxide-Semiconductor Field-Effect-Transistor



(a) (b)

Figure A.3: Programming a Floating-Gate Transistor: CHE Injection and FN Tunneling.

erased value to the programmed value. However, it is also an inefficient method since less than 0.001% of the channel current will be directed to the floating gate.

In NAND architectures, Fowler-Nordheim Tunneling is used to program the floating gate. A high voltage of about $20V$ is applied to the control gate, while the drain, source and bulk terminals are kept at $0V$ (Figure A.3b). Once again, electrons gain sufficient energy to overcome the oxide barrier, but the process is now better controlled and more efficient. Even though FN Tunneling is slower than CHE Injection, it requires very small programming current per cell (less than $1nA$) which allows many cells to be programmed simultaneously.

Erase

To erase a floating-gate transistor, both architectures use the FN Tunneling mechanism although with different parameters. In NAND architectures, the control gate is biased with a negative voltage of about -$20V$, between $4V$ and $6V$ are applied to the drain, while the source and bulk terminals are kept at $0V$ (Figure A.4a). In NOR architectures a similar setup is used however this time the source terminal is floated and a lower voltage of about -$10V$ is applied to the control gate.

During the erasure process a high electric field is applied across the gate oxide that pushes the electrons out of the floating gate and reduces the transistor's threshold voltage. When erased, a NOR memory cell conventionally stores a logic 0 value while a NAND

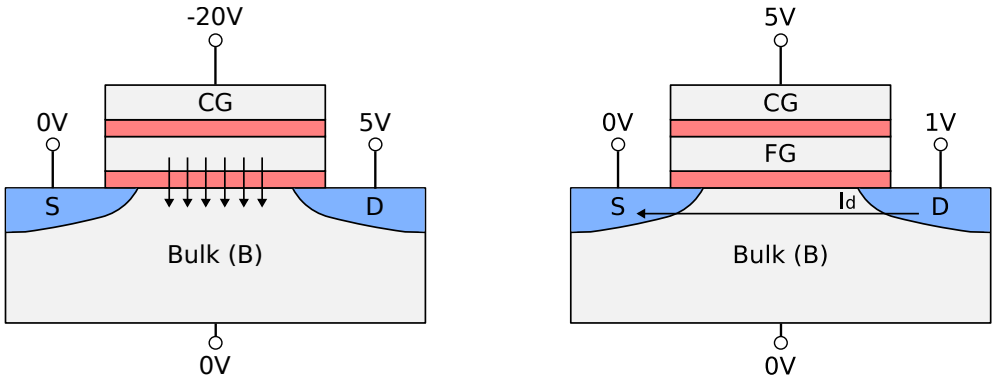


Figure A.4: Erasing and Reading a Floating-Gate Transistor.

memory cell a logic 1 value. It is considered to be a relatively slow procedure compared to programming as it usually takes a few milliseconds.

Read

With the mechanism to store a bit as either 0 or 1, all that is left is to read the stored value. In read mode, the control gate is biased at about $5V$, the source and bulk terminals are kept grounded while a small voltage of about $1V$ is applied to the drain. If electrons are trapped in the floating gate, the *Threshold Voltage* V_{th} required for electrons to be pulled from the source and form a channel between the Source and the drain will be higher than the applied *Reference Voltage* V_{ref}. As a result, there will be no current flow through the cell. However, if no electrons are trapped in the floating Gate the behavior of the control gate is similar to that of a regular MOSFET and current will start flowing through the channel (Figure A.4b). This current flow can be monitored by an external circuit in order to determine the state of the cell as being either 0 or 1.

In flash technology, the process of reading does not destroy the data however, caution has to be exercised when applying voltage to the cell drain during a read operation. If higher voltages are applied, then the cell would experience *Read-Disturb* caused by Channel Hot Electron generation. Although the generated current is relatively small (10 to $50\mu A$), it has to be considered that a flash cell is in read mode most of the time and therefore, even small amounts of read-disturb can have an impact on data integrity over

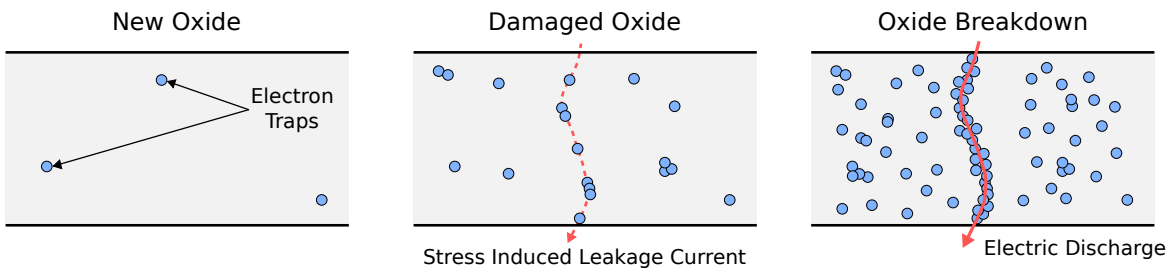


Figure A.5: Stages of Oxide Layer Breakdown.

time [6].

Reliability

The reliability of a floating-gate transistor is one of its most important features since flash memory manufacturers typically need to guarantee at least 10 years of charge retention and $1k$ to $100k$ Program/Erase cycles for a product chip. There are multiple leakage paths which can lead to loss of the programmed floating-gate electron charges [46, 49]. Electrons can be lost through current flowing from the floating gate to the control gate ($I_{IPD-leak}$) or through the side wall oxide ($I_{SW-leak}$) however, the main cause of *Charge Loss* is the leakage though the tunnel oxide layer separating the floating gate from the substrate ($I_{TOX-leak}$).

The tunnel oxide insulation layer is very thin (less than $10nm$) and the erasure and programming processes subject it to stress from large electric fields. Such changes can result to structural changes in the SiO_2 layer and lead to defects that trap electrons in the oxide layer. As a result, tiny "cracks" start to appear in the insulation material that permit the charge on the floating gate to leak into the substrate by *Stress Induced Leakage Current* (SILC) [56, 54]. These cracks are in fact broken bonds of the atoms in the SiO_2 layer (electron traps) that are caused by the electron tunneling processes. As the number of Program/Erase cycles is increased more and more defects are starting to appear and eventually lead to a *Breakdown* of the oxide layer (Figure A.5).

Programming, erasing and reading memory cells can cause charge disruptions within adjacent memory cells and lead to *Disturb Faults*. By taking into consideration the high

density of the memory cells in modern flash memories, voltage changes that are capacitively coupled between adjacent memory cells can lead to random bit errors in the stored data. Disturb faults are not detected easily and usually the flash controller has to keep track of the number of Read/Program/Erase operations across the whole storage device in order to prevent them.

The deterioration of the tunnel oxide over time and disruptions from adjacent memory cells can eventually lead to random bit errors in the stored data and while the chances of any given data bit becoming corrupted are extremely small, the number of data bits stored in a system increases the likelihood of data corruption. Therefore, Error Detection and Correction Codes (ECC) are used in modern most flash memory systems to prevent the data from becoming corrupted. Although the use of ECCs limits the performance of the system, it is necessary in order to correct bit errors and provide reliable storage.

Short Vita

Vasileios Papadopoulos was born in Ioannina, Greece in 1989. He graduated from the 1st High School of Ioannina in 2006 and in 2012 he obtained his Diploma in Computer Engineering from the Computer Engineering and Informatics Department of the University of Patras. Currently he is a Postgraduate student at the Department of Computer Science and Engineering of the University of Ioannina and a member of the Systems Research Group. His research interests include Storage Systems, Solid-State Drives and Operating Systems.