A Direct-Access File System for a New Generation of Flash Memory

William K. Josephson

A Dissertation
Presented to the Faculty
of Princeton University
in Candidacy for the Degree
of Doctor of Philosophy

Recommended for Acceptance
by the Department of
Computer Science
Adviser: Kai Li

April 2011

© Copyright by William K. Josephson, 2011.

All rights reserved.

Abstract

Flash memory has recently emerged as an important component of the storage infrastructure, but presents several unique challenges to storage system designers: individual blocks must be erased before they can be rewritten, block erasure is time consuming, and individual blocks may be erased only a limited number of times. By combining hardware parallelism and a log-structured approach, state of the art flash storage systems can deliver two to three orders of magnitude more I/O operations per second than existing high-performance fibre channel disk drives. Despite the emergence of state of the art solid state disks (SSDs), the storage software stack has changed little and is still optimized for magnetic disks; the basic abstraction remains a linear array of fixed-size blocks together with a very large DRAM cache to convert user I/O to a smaller number of large I/O transfers.

This thesis presents the design, implementation, and evaluation of the Direct File System (DFS), describes the virtualized flash memory abstraction layer it depends upon, and introduces some simple heuristics for improved flash-aware buffer cache algorithms. We first examine the impact SSDs have had on the utility of the buffer cache. As SSD performance and density improves, the value of very large DRAM based buffer caches declines. We examine the change in tradeoffs through database buffer cache traces and simulation and offer some simple heuristics to take advantage of these changes. Second, we propose a richer interface more suitable for solid-state storage systems. This interface provides for sparse block or object-based allocation, atomic multi-block updates, and a block discard interface to facilitate reclamation of unused storage. Finally, we present DFS, a file system designed specifically for next generation flash memory systems that takes advantage of our proposed storage interface. The result is a much-simplified file system implementation that runs under Linux.

In both micro- and application benchmarks DFS shows consistent improvement over ext3, a mature and widely deployed file system for Linux, both in throughput and in CPU usage. For direct access, DFS delivers as much as a 20% performance improvement in microbenchmarks. On an application level benchmark, DFS outperforms ext3 by 7% to 250% while requiring less CPU power.

Acknowledgments

No thesis springs forth fully formed. Rather, it is the product of the advice, support, and encouragement of many.

First and foremost I would like to thank Kai Li, my advisor. I was first introduced to Kai by Margo Seltzer in the spring of my senior year at Harvard. Upon her suggestion and at Kai's invitation I found myself in the spring of 2002 in San Mateo working at Data Domain, then a small start-up in Greylock's Bay Area offices. I learned more about both research and entrepreneurship in a year at Data Domain than at any other time. Kai subsequently made it possible for me to come to Princeton as one of his students and I have continued to benefit from his insights and his uncanny talent for technological augury.

David Flynn, the then CTO and now CEO of FusionIO, has been a generous and helpful collaborator throughout the current project. David described his vision for flash memory based primary storage and provided the hardware and software tools that have formed a critical ingredient in the current study. He has balanced the needs of a new venture with those of research and I have benefited greatly.

Garret Swart and the engineers on the buffer cache team at Oracle have provided a wealth of information and insight into database cache behavior generally and online transaction processing with Oracle in particular. Garret made possible the instrumentation and benchmarking of Oracle which is, to say the least, a unique and valuable opportunity for those of us in research outside industry. The collaboration with Garret forms the basis for the work on caching discussed in Chapters 4 and 5.

I am also indebted to the members of my committee. Brian Kernighan has endured being a neighbor in the department since my third year at Princeton when Kai's students took up full time residence in 318a and has continued to provide sage advice. He also foolishly agreed to be a reader. Jennifer Rexford joined the faculty at Princeton shortly after I first arrived. A champion for graduate students in the department since her arrival, she, too, volunteered for duty as a reader and has provided both encouragement and invaluable feedback throughout the process. Michael Freedman and Doug Clark graciously agreed to be non-readers and through their insightful questions have helped me to improve the presentation of my research.

The National Science Foundation generously supported the first several years of my graduate career through its Graduate Research Fellowship. My work was also supported in part by Fusion I/O, by the Intel Research Council, and by National Science Foundation grants CSR-0509447, CSR-0509402, and SGER-0849512.

To my parents for their love, support, and patience

They provided the encouragement and made the way possible

To my friends for listening

For good company in a journey makes the way passable

Contents

Abstract . . . iii

Acknowledgments . . . iv

List of Figures . . . viii

List of Tables . . . x

1 Introduction **1**

2 Flash Memory and Existing Technology **7**

2.1 An Overview of NAND Flash Technology . . . 7

2.2 Flash Memory and Write Endurance . . . 13

2.2.1 Flash Translation Layer . . . 15

2.2.2 Garbage Collection . . . 19

2.3 Existing Flash File Systems . . . 22

2.4 Flash Memory and Caching . . . 28

2.5 Existing Flash Hardware . . . 31

2.6 Summary . . . 37

3 Rethinking the Storage Layer **39**

3.1 Existing vs. New Abstraction Layers . . . 41

3.2 Storage Interface . . . 43

3.3 Virtualized Flash Storage Layer . . . 50

3.4 Summary . . . 53

4 Rethinking the File System **55**

4.1 Storage Allocation 56

4.1.1 Crash Recovery 62

4.2 DFS Layout and Objects 66

4.2.1 Direct Data Accesses 69

4.3 Directories 70

4.4 Caching in DFS 73

4.4.1 Lazy LRU 75

4.5 Summary 78

5 Evaluation **80**

5.1 Evaluation of DFS 81

5.1.1 Virtualized Flash Storage Performance 82

5.1.2 Complexity of DFS vs. ext3 83

5.1.3 Microbenchmark Performance of DFS vs. ext3 84

5.1.4 Application Benchmarks Performance of DFS vs. ext3 89

5.2 Caching 97

5.2.1 Simulation Environment 98

5.2.2 Workloads 100

5.2.3 Simulation and Replay Results 101

5.3 Summary 112

6 Conclusion and Future Work **114**

Bibliography **119**

List of Figures

1.1 Cost Trends for NAND Flash and DRAM 3

2.1 Floating Gate Transistor 9

2.2 NAND Flash Array Organization 10

2.3 FTL Metadata 17

2.4 FTL Address Mapping 18

3.1 Flash Storage Abstractions 41

3.2 Storage API: Objects 44

3.3 Storage API: Block Storage 46

3.4 Storage API: Key-Value Pairs 48

3.5 Storage API: Atomic Update 50

4.1 File Size Histogram 59

4.2 File Size CDF 60

4.3 DFS logical block address mapping 61

4.4 DFS File Layout 67

4.5 DFS I-Node 68

4.6 DFS Directory Entry 72

5.1 DFS Throughput as a Function of Concurrency 86

5.2 TPC-C Logical Block Popularity 103

5.3 TPC-C Hit Rate vs Cache Size 104

5.4 TPC-C Wall Time in Seconds vs Cache Size 105
5.5 TPC-C Single Rate vs Cache Size 106
5.6 Run Time for TPC-H Queries 11 and 18 108
5.7 Run Time for TPC-H Query 19 110

6.1 TPC-C Runtime 116

List of Tables

2.1 Comparison of Storage Devices 34

5.1 Device 4KB Peak Random IOPS 83

5.2 Device Peak Bandwidth with 1MB Transfers 83

5.3 Lines of Code: DFS *vs* Ext3 84

5.4 Peak Bandwidth 1MB Transfers on ioDrive (Average of Five Runs) . 85

5.5 DFS CPU Utilization 87

5.6 Buffer Cache Performance with 4KB I/Os 88

5.7 Memory Mapped Performance of Ext3 & DFS 89

5.8 Application Benchmark Summary 90

5.9 Application Benchmark Execution Times 92

5.10 DFS *vs* Ext3 92

5.11 Zipf N-Gram Queries 95

5.12 TPC-H Descriptive Statistics 107

(a) Average Number of References Per Block 107

(b) Summary Statistics 107

Chapter 1

Introduction

Flash memory has become the dominant technology for applications that require large amounts of non-volatile solid-state storage. These applications include music players, cell phones, digital cameras, and shock sensitive applications in the aerospace industry. In the last several years, however, both the academic community and industry have shown increasing interest in applying flash memory storage technologies to primary storage systems in the data center. There are four trends driving the adoption of flash memory as an enterprise storage medium: performance, particularly random I/O performance, reduced power usage, cost, and reliability.

Random I/O presents a major challenge to primary storage systems, particularly those constructed with magnetic disks. Most improvements in disk technology have come in the form of higher recording density. This has dramatically reduced the cost of storage on a per-gigabyte basis and has improved sequential transfer rates. The time to service a random I/O request, on the other hand, is dominated by mechanical positioning delay, which has not kept pace. Flash memory has no electro-mechanical components. Instead, flash memory depends upon microchip feature size and process technologies, which have continued to improve exponentially according to Moore's Law. As a result, flash memory storage systems can deliver one to two orders of

magnitude more random I/Os per second than can a magnetic disk. The typical enterprise grade magnetic disk offers sequential read and write bandwidths of 200 MB/s and 120 MB/s, respectively. The average latency for random I/O on these same disks is between 3.4-4.0ms. By contrast, current solid state disks (SSDs) using NAND flash memory can achieve 170-250 MB/s for sequential I/O and can service random I/O requests with an average latency of 75-90μs. Next generation SSDs provide substantial further improvements. Several are able to deliver 750-1000 MB/s sequential I/O throughput and are able to service a random I/O with an average latency of as little as 8μs to 50μs.

Low power consumption is a further advantage of flash memory that has driven commercial adoption. A single high-performance magnetic disk may use as much as nine or ten watts when idle and fifteen or more watts when active. Existing SSDs designed to be drop-in replacements for magnetic disks consume as little as 60 milliwatts when idle and a few hundred milliwatts to two or three watts when active, depending on the underlying technology. Single-level cell, or SLC, NAND flash based SSDs typically consume more power than those using multi-level cell, or MLC, NAND flash. Some high-performance SSDs do consume more power than a single magnetic disk when active, but these devices also deliver substantially higher random I/O performance. As a result, some SSD vendors have advocated comparing the power consumption of SSDs to magnetic disks on the basis of power dissipated per I/O per second. For workloads that have a large random I/O component, this metric may be a good measure of power efficiency. In order to achieve the same random I/O performance with such a workload, a disk-based storage system may require an array of tens to hundreds of magnetic disks. The result is a gross over-provisioning of storage in order to achieve adequate performance. An SSD that with power consumption comparable to a single magnetic disk uses far less power than a large disk array.

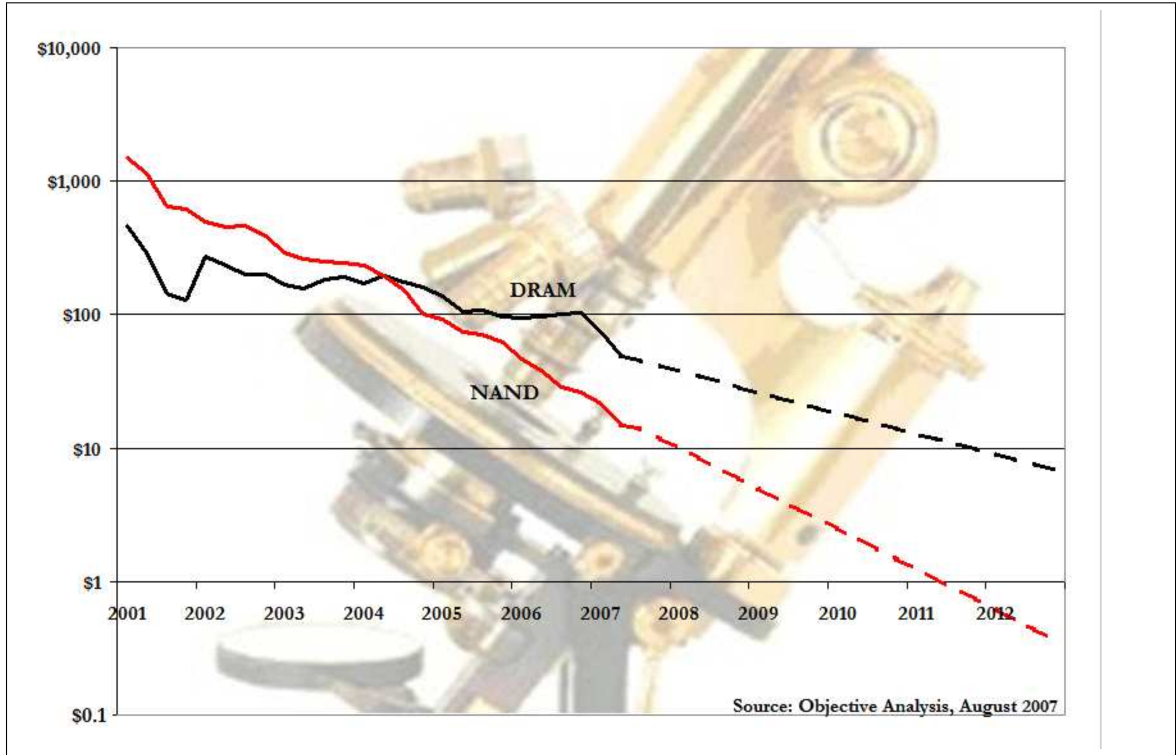

Figure 1.1: Cost Trends for NAND Flash and DRAM [30]

Over-provisioning of disks in order to have enough spindles to handle the random I/O component of a workload is also expensive in economic terms. Traditionally, system designers have added both more disks and more main memory. Large DRAM caches play an important role in achieving good performance in large storage systems. The memory subsystems necessary to use large DRAM arrays effectively are expensive to design and build and do not benefit from the economies of scale seen in systems also sold to consumers. Furthermore, flash memory is cheaper than DRAM on a per-byte basis and the gap between the two has been growing for some time. Figure 1.1 illustrates the trend as of about 2007. Hard data in the DRAM and NAND flash markets is difficult to find: most memory is sold under contract and the prices are not readily available. The spot market is very volatile, particularly when new, popular consumer devices such as the iPhone come to market. Spot price data from Spring 2009, Spring 2010, and Fall 2010 have continued to bear out the general trend shown in Figure 1.1 [20].

Early attempts to use flash memory as a primary storage medium for both personal computing and the data center have focused on replacing magnetic disks directly. First generation solid state disks (SSDs), including offerings intended for the enterprise, have replicated the physical form factor, electrical interconnect, and storage protocols used for magnetic disks. Furthermore, the file systems and buffer cache algorithms used to manage these SSDs have remained largely unchanged. Instead, the focus of the research community and industry has been on making improvements in the software and firmware that emulates the block-oriented disk interface. This layer, called the flash translation layer (FTL), is a critical component in high performance flash memory storage designs. It is also the software layer that is responsible for implementing the backwards compatible block interface. Recently proposed implementations of the flash translation layer are intended to be implemented in a solid state disk controller and to support the existing disk drive abstraction [1, 7, 35, 71]. System software then uses a traditional block storage interface to support file systems and database systems designed and optimized for magnetic disk drives.

Flash memory behavior and performance is substantially different from that of magnetic disks. As a result, we argue that shoehorning flash memory into the same interface as magnetic disks and using the cache algorithms and file systems designed for magnetic disks is not the best approach. The goal of our work is to identify new abstractions tailored for flash and to use them as the basis for file systems and cache designs intended specifically for flash memory. We have implemented these designs, including a file system designed to exploit the potential of NAND flash memory and buffer cache algorithms better suited to next-generation flash storage systems.

This thesis presents the design, implementation, and evaluation of the Direct File System (DFS), describes the virtualized flash memory abstraction layer it depends upon, and introduces some simple heuristics for improved flash-aware buffer cache algorithms. DFS is a file system designed specifically for an emerging generation

of flash storage device. The virtualized flash memory abstraction layer provides a uniform interface suited to emerging high-performance solid state disks such as FusionIO's ioDrive. With the advent of relatively inexpensive, high performance solid state disks and the gap in cost per gigabyte between DRAM and NAND flash memory, the marginal utility of additional DRAM cache has fallen dramatically. We argue that DRAM caches will not keep pace with the increase in the size of primary storage systems, if only for economic reasons. The usual approach has been to devise ever more clever cache algorithms to improve the cache hit ratio. We present instead a simple heuristic that seeks to cache only popular blocks that we call Lazy LRU. The motivation for this approach is that the cost of a miss is far lower with solid state disks that it is with storage systems constructed with traditional magnetic media. As a result, the modified version of LRU can improve improve system performance in primary storage systems using NAND flash by avoiding unnecessary copying of cache blocks to user buffers and by improving scan-resistance. In our experiments using the well-known TPC-C online transaction processing benchmark and the TPC-H decision support benchmark, Lazy LRU improves performance by about 4% with minimal changes to the cache algorithms or implementation.

DFS, which we first described at the USENIX File and Storage Technology (FAST) conference in February of 2010 [40], is designed to take advantage of the virtualized flash storage layer for simplicity and performance. The virtualized storage abstraction layer provides a very large, virtualized block addressed space, which can greatly simplify the design of a file system while providing backward compatibility with the traditional block storage interface. Instead of pushing the flash translation layer into disk controllers, this layer combines virtualization with intelligent translation and allocation strategies for hiding bulk erasure latencies and performing wear leveling on the host.

A traditional file system is known to be complex and typically requires four or more years to become mature. The complexity is largely due to three factors: complex storage block allocation strategies, sophisticated buffer cache designs, and methods to make the file system crash-recoverable. DFS dramatically simplifies all three aspects. It uses virtualized storage spaces *directly* as a true single-level store and leverages the virtual to physical block allocations in the virtualized flash storage layer to avoid explicit file block allocations and reclamations. By doing so, DFS uses extremely simple metadata and data layout. As a result, DFS has a short data path to flash memory and encourages users to access data directly instead of going through a large and complex buffer cache. DFS leverages the atomic update feature of the virtualized flash storage layer to achieve crash recovery.

We have implemented DFS for the FusionIO's virtualized flash storage layer and evaluated it with a suite of benchmarks. We have shown that DFS has two main advantages over the ext3 [85] file system. First, the number of lines of code in our file system implementation is about one eighth the number of lines of code in ext3 with similar functionality. Second, DFS has much better performance than ext3 while using the same memory resources and less CPU. Our microbenchmark results show that DFS can deliver 94,000 I/O operations per second (IOPS) for direct reads and 71,000 IOPS direct writes with the virtualized flash storage layer on FusionIO's ioDrive. For direct access performance, DFS is consistently better than ext3 on the same platform, sometimes by 20%. For buffered access performance, DFS is also consistently better than ext3, and sometimes by over 149%. Our application benchmarks show that DFS outperforms ext3 by 7% to 250% while requiring fewer CPU cycles.

Chapter 2

Flash Memory and Existing Technology

This chapter provides a high-level overview of NAND flash memory technology, the existing software used to manage it, and the primary challenges that must be overcome to obtain good performance. By virtue of its organization, NAND flash has performance and reliability characteristics that require specialized software in order to obtain good performance and longevity. The software that manages flash memory at the lowest level in traditional solid state disks is called the flash translation layer or FTL. We describe the organization of existing commodity solid state disks and the role of the FTL. We then discuss the primary challenges and existing approaches to achieving good performance using flash memory.

2.1 An Overview of NAND Flash Technology

Flash memory is a type of electrically erasable solid-state memory that has become the dominant technology for applications that require large amounts of non-volatile solid-state storage. These applications include music players, cell phones, digital cameras, and shock sensitive applications in the aerospace industry. More recently it

has been used as the primary storage medium for personal computers such as laptops that benefit from the low power requirements of NAND flash. It is also beginning to see use in enterprise data storage in conjunction with traditional disk-based storage systems because of its lower power consumption and the possibility of higher random I/O performance than traditional magnetic media.

Flash memory was originally invented at Toshiba in the early 1980s [22] and commercialized toward the end of the decade by Intel. Flash memory consists of an array of individual cells, each of which is constructed from a single floating-gate transistor [69]. A major advantage of flash memory is that the individual cells are extremely simple and regular, allowing for high-density fabrication. Flash is often the first type of device to be manufactured in quantity using a new fabrication technology as a result. This leads to even greater packaging density. Furthermore, flash chips must account for partial failure over time and it is therefore possible to mark blocks containing manufacturing defects at the time of production, increasing the yield and therefore decreasing the price per bit. This latter approach is comparable to the use of bad block maps on hard disk drives.

Floating gate transistors are similar to standard field-effect transistors (FETs) with a source, drain, and control gate [42]. A block diagram of a floating-gate transistor is shown in Figure 2.1. Unlike a standard FET, floating gate transistors have an additional floating gate that sits between the control gate and the channel between source and drain. This floating gate is electrically isolated by an insulating layer of metal oxide and therefore only capacitatively coupled to the control gate. The charge on the floating gate represents the value stored by the cell. When the gate is charged, it partially cancels the electric field applied via the control gate, changing the threshold voltage of the field-effect transistor. In order to read the contents of the cell, a voltage is applied to the control gate. The charge on the floating gate determines

the conductivity of the channel between source and drain. The resulting current flow through the channel is sensed to determine the value stored.

Traditionally, flash cells have stored a single bit [81]. These so-called single-level cell (SLC) flash devices are being replaced in some applications by multi-level cell (MLC) devices. Multi-level cell devices store more than one bit per cell. Multiple bits are stored in a single flash cell by sensing the charge on the floating gate more precisely. The charge determines the current flow through the FET channel. Rather than mere presence or absence of a current flow determining the value read from the cell, the amount of current flowing corresponds to one of several different possible values. Single-level cell flash devices are more robust, but multi-level cell devices offer higher densities and lower cost per bit.

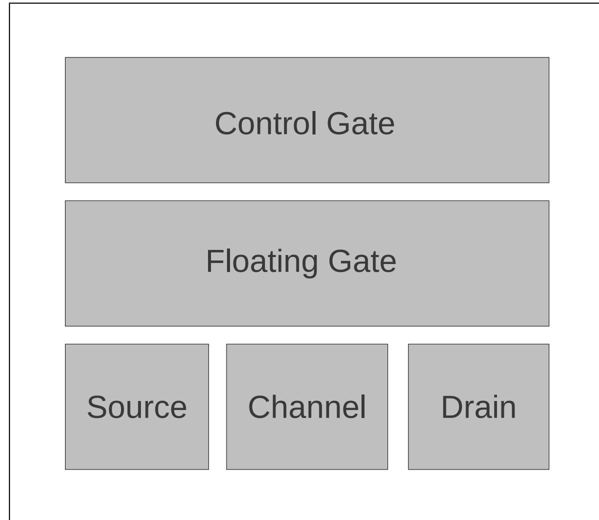

Figure 2.1: Floating Gate Transistor: Floating Gate is isolated from Source, Drain, and Control Gate by Dielectric.

Flash memory cells are programmed and erased through quantum tunneling [69]. A voltage significantly higher than the voltage used to read the flash cell is applied to the control gate. The application of a charge to the control gate allows a current to flow between source and drain. Depending upon the polarity of the charge applied between control gate and source, electrons either jump to or are pulled off of the floating gate via quantum tunneling. Any electrons deposited on the floating gate will be trapped there. The charge on the floating gate does leak, but very slowly over a period of years. As a result, flash memory cells can retain data for a decade or more.

Intel introduced the first commercial flash devices in 1988 as an electrically erasable replacement for ROM chips often used to store firmware. These early devices were so-called “NOR-flash” devices so named because the cells are arranged in a pattern similar to NOR gates [6]. NOR flash offers several benefits in the role of firmware storage: it can be erased and re-programmed *in situ* and it supports random access and word-level addressability. Furthermore, read and write operations typically take tens of microseconds. A major drawback of NOR flash is that the erase operation may take more than a millisecond. As with all types of flash memory, in order to change the value stored in a flash cell it is necessary to perform an erase before writing new data. Modern NOR flash chips are organized into blocks of cells. An erase operation applies only to an entire block at once [15]. Typically, NOR flash cells can be programmed a byte or entire word at a time.

Like NOR flash, NAND flash is constructed from arrays of floating-gate transistors. Unlike NOR flash, however, NAND flash cells are not word addressable. In a NAND flash device, several flash cells are connected in series. Several additional transistors are required to control the individual flash cells. Despite this added overhead, the reduction in the number of ground wires and bitlines for transferring data to and from cells allows for a significantly denser layout than is possible with NOR flash. The result is higher storage density per chip at the expense of word-addressability [15].

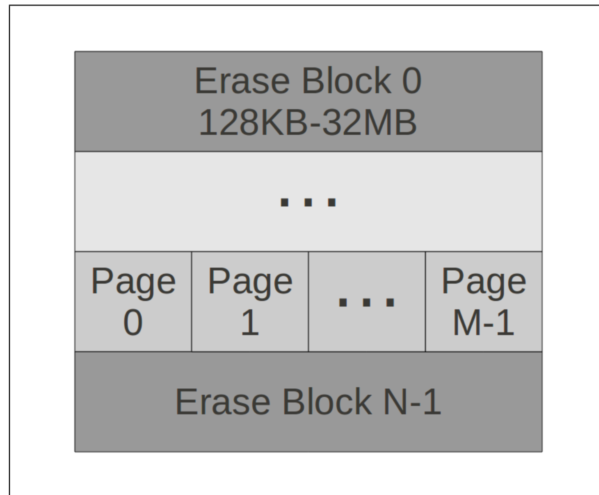

Figure 2.2: Block diagram of a NAND flash array with N erase blocks each containing M pages of data. Pages can be written or programmed individually, but entire erase blocks must be erased at once.

In a NAND flash array, the storage cells are logically organized into pages which are further grouped into erase blocks, as shown schematically in Figure 2.2 [15]. As in the case of NOR flash, all of the cells in a NAND flash erase block must be erased at the same time. Unlike NOR flash, however, NAND flash must be read and programmed an entire page at a time. The size of a NAND flash page varies, but in most cases is from 512 bytes to as much as 16KB. Typical erase block sizes run from roughly 32KB to a few tens of megabytes depending on the particular device. Any data in an erase block that is to be preserved must be copied first.

NAND flash block erasure is a time consuming operation, just as it is for NOR flash. Typical devices take about one or two milliseconds to erase a NAND flash erase block. Furthermore, an erase block may be erased successfully only a limited number of times. The number of successful erasures is known as the write endurance and depends on a number of factors, including the particular process technology used to manufacture the part and the size of individual semiconductor features. It usually ranges from as few as 5,000 erase cycles for consumer grade MLC flash to 100,000 to 1 million erase cycles for enterprise grade SLC flash [81]. Erase time, write endurance, and the block-oriented nature of NAND flash are the salient features for software system designers seeking reliability and performance from flash storage systems.

An additional concern in any storage system is failure – *i.e.,* the inability to correctly retrieve data previously stored in the system. For traditional disk-based storage systems, the failure rate is often expressed as the mean time between failures (MTBF). Informally, the MTBF is the predicted average elapsed time between failures of the storage system. Component parts of the storage system are most often assumed to have constant failure rates during their useful life after an initial "burn-in" period and therefore failures are assumed to be exponentially distributed. Although expressed as an elapsed time, MTBF is best understood as a failure rate during the useful life of the device. Disk manufacturers quote MTBF for their products on the basis of an

accelerated aging regimen which may or may not accurately reflect failure rates in the field.

Another reliability metric which is useful for mass storage systems and is more widely applied in solid-state storage systems is the uncorrectable bit error rate (UBER). Modern storage systems, including both disk-based systems and solid-state systems, use error correcting codes to detect and correct errors. The data to be stored by the system is divided into blocks and each block is encoded. The encoded block is called a code word and is larger than the original user data. Flash devices store the additional check bits out of band in each page. A decoding algorithm is used at retrieval time to detect and correct corruption of the data. The fraction of corrupted bits that can be tolerated is a function of the error correcting code used. Informally, the UBER is the probability of a code word having an uncorrectable bit error divided by the number of user-visible data bits per code word. That is, the UBER is the rate of bit errors after the application of the error correcting code (ECC). The UBER is a more nuanced measure of reliability which does not consider the entire device to have failed merely because of a single bit upset.

Both disks and flash memory devices employ error correcting codes to detect and correct multiple bit errors. In the case of NAND flash, additional flash cells are used to store ECC bits for each page [81]. A separate microcontroller is responsible for carrying out the error correcting code computations and signaling the host in the case of failure.

There are several common failure modes for flash devices which can be addressed with error correcting codes. Recent studies by researchers at Intel and Micron have shown that with suitable error correcting codes, modern flash devices can achieve a UBER on the order of 10^{-15} [63], which is similar to that for existing hard disk technology [27]. A simple high-level classification of these failure modes is:

- Program disturb

- Read disturb
- Data retention loss
- Limited write endurance

Each failure mode is the result of underlying physical effects such as quantum level noise effects, erratic tunneling, and disturbances during quantum tunneling. Program and read disturb are caused by non-selected cells receiving heightened voltage stress during a program or read operation to adjacent cells. This process is non-destructive and may be managed with ECC but eventually requires the data in the stressed cells to be copied to a new location and the old location erased. Similarly, data retention loss occurs due to dielectric leakage over time. It is a non-destructive process but requires data to be copied occasionally in order to guarantee long-term data retention. The fourth high-level failure mode is failure to erase or program successfully due to long-term changes in cell characteristics due to excessive program/erase cycles. This is the most well-known reliability concern with flash memory technology. A variety of wear leveling techniques are employed by flash storage systems to avoid "burning out" any one location. It is worth noting, however, that even excessive reading of a location may require moving data although no permanent damage is done to a cell.

2.2 Flash Memory and Write Endurance

The simplest approach to building a storage system using flash memory is to treat the entire flash device as an array of physically addressed blocks with each block represented by a single physical flash page. There are at least two significant problems with this method: first, some blocks may be written far more often than others and due to the limited write endurance of flash cells, these blocks will fail prematurely. Second, using a physically addressed linear array of blocks makes it impossible to update less than a full erase block at a time. In order to update a single page, the

entire erase block must be read, the block erased, and the constituent pages in the erase block re-written. If a power failure or crash occurs after erasure but before the pages in the erase block have been programmed, data blocks colocated in the same erase block but otherwise not involved in the write operation will be lost.

To illustrate the first problem, consider a program that updates a particular 4KB physical page once a second on a 2GB consumer MLC NAND flash chip. A common example of such a device are the Intel MD78 series of NAND flash devices [37]. Such chips typically have a write endurance of about 10,000 erasure/program cycles before a cell fails [81]. If a direct physical mapping is used to map logical file system blocks to physical flash pages, the block being updated may fail in as little as 10,000 seconds or less than three hours. If, on the other hand, the updates are evenly distributed over all of the $\sim 500,000$ pages on the device, the write endurance will not be exceeded for approximately eighty years, far longer than the designed useful life of the part. These two scenarios represent the two extremes for endurance given a very simple, regular workload. More generally, system designers are interested in an online algorithm that provides good write endurance even when the workload is not known in advance. Algorithms for distributing writes to avoid premature failure due to the limited write endurance of flash cells are called *wear leveling* algorithms.

The second drawback of the direct-mapped, physically addressed approach to managing flash memory is that updates smaller than the *erase block* size are inefficient and no longer atomic [1]. In order to update a single page under this scheme, it is necessary to read an entire erase block, modify the target page in main memory, erase the erase block, and then program all of the pages in the erase block with the updated values. This read-modify-write cycle is inefficient since it requires costly erase and program operations as well as two transfers of an entire erase block when only an individual page needs to be updated. This phenomenon is known as *write amplification* since a single small write operation may necessitate several much larger

writes [33]. The read-modify-write cycle to update a page also introduces the possibility of data loss in the event of a power failure after the erase operation but before the write operation has completed. In practice, the algorithms used to solve the wear leveling problem naturally address the problem of inefficient, non-atomic updates as well.

Because write endurance is not an issue for disk-based storage systems, existing file systems, database systems, and other clients of the storage system have not been designed with the write endurance problem in mind. There are two common methods for resolving this situation: the first is to implement wear leveling in firmware or in software as part of the host device driver. The first widely used implementation of this idea, the so-called flash translation layer, was patented by Ban and standardized by Intel [3, 35]. The second common approach is to solve the write-endurance problem directly in a file system specifically designed for flash memory. Examples of this second approach include JFFS2, YAFFS, and others [55, 88]. We describe both of these approaches in what follows.

2.2.1 Flash Translation Layer

Wear leveling algorithms are designed to distribute updates evenly across the pages of a flash device. At a high level, all wear leveling algorithms work by introducing a level of indirection. This level of indirection allows flash memory pages to be identified by a virtual block address rather than a physical block address. A mapping from virtual to physical block addresses is maintained by the system. In the case of NAND flash based storage, the block size typically corresponds to the size of a single page. When a block is to be updated, a fresh page is allocated from a pool of previously erased pages. The data to be written is programmed in the new page, the mapping from virtual to physical block is updated, and the old page is marked as unused and suitable for reuse. This basic approach has three advantages:

1. Each modification of a frequently updated block is written to a different physical location, avoiding premature wear.
2. The block-level virtual to physical mapping allows individual flash pages to be updated without erasing and re-writing an entire erase block.
3. Updates to a block can be made atomic by using well understood techniques such as write-ahead logging for the virtual address mapping.

The fundamental problem introduced by an additional level of indirection is how to store and maintain the virtual to physical mapping in an efficient and crash-recoverable manner. A secondary problem is to devise time and space efficient wear leveling policies.

Flash memory storage systems have traditionally solved the problem of translating virtual block addresses to physical flash addresses by maintaining two maps: a map from virtual to physical address and an inverse map from physical addresses back to virtual addresses. The inverse map is stored implicitly in the header of an erase block or flash page. This header contains the virtual block address of each page as well as bits indicating whether or not the page contains valid data and a version number. This additional metadata may be stored in the same physical location as the data or it may be stored in a dedicated metadata page elsewhere in the same erase block. In either case, the forward or virtual to physical mapping can be computed by scanning the entire flash device during initialization.

Unlike the implicit inverse mapping, the forward mapping is stored at least partially in main memory. For large flash devices, the entire mapping may be too large to fit in main memory, particularly in resource constrained environments. Portable flash devices often implement wear leveling in firmware running on an embedded processor with little volatile memory. One alternative is to store at least a portion of the forward mapping on the flash device as well. It is not practical to store the entire mapping in flash, however, as the forward mapping must support fast lookups and frequent fine-grained updates for which flash is ill-suited.

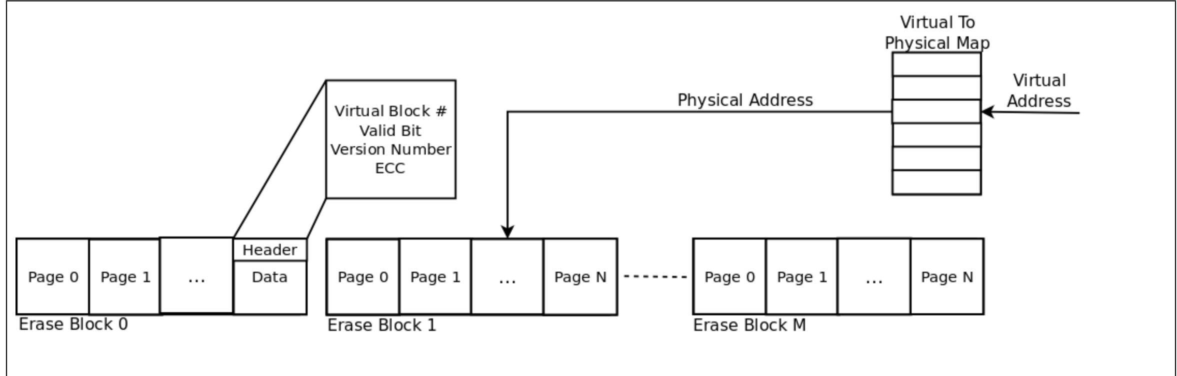

Figure 2.3: Metadata for Virtual Addressing of Flash Memory to Support Wear Leveling

Figure 2.3 shows the generic metadata used by wear leveling algorithms. Small devices may use the per-page header to store the entire inverse map and reconstruct the forward map when the device is first mounted. Many NAND flash devices reserve a small amount of extra storage in each page for out of band metadata. In the absence of such a reserved area, the FTL may reserve several pages at the beginning of each erase block to store metadata for all of the pages in the erase block.

Scanning the per-page metadata to reconstruct the address mapping is a viable approach for small devices, but is not practical for large devices as the time required to scan is prohibitive. In this case, the per-page metadata may be useful for detecting inconsistencies or in the case of catastrophic failure. As a practical matter, modern flash storage systems must store the forward mapping as well. A popular approach to doing so is the flash translation layer [3, 35]. The FTL stores the forward map in flash while attempting to reduce the cost of updates to this stable copy.

There have been many proposals for the design and implementation of a flash translation layer in recent years. Examples include the original Intel specification as well as a number of proposals from the research community, including Fully Associative Sector Translation (FAST) [47], Locality-Aware Sector Translation (LAST) [48], DFTL proposed by Gupta *et al.* [28], and the space-efficient FTL of Kim *et al.* [46]. More recent efforts focus on high-performance in SSDs, particularly for random writes. Birrell *et al.* [7], for instance, describe a design that significantly improves random

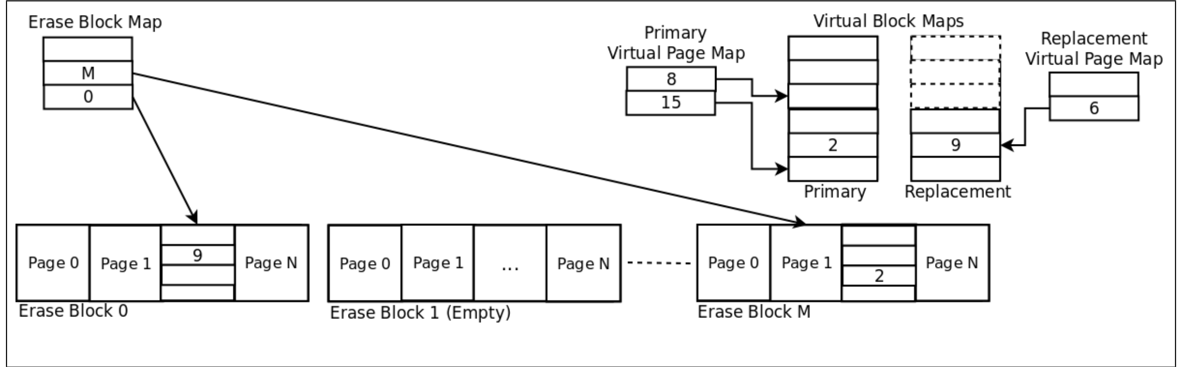

Figure 2.4

write performance by keeping a fine-grained mapping between logical blocks and physical flash addresses in RAM. Similarly, Agrawal *et al.* [1] argue that SSD performance and longevity is strongly workload dependent and further that many systems problems that previously have appeared higher in the storage stack are now relevant to the device and its firmware. This observation has led to the investigation of buffer management policies for a variety of workloads. We focus on the flash translation layer as standardized by Intel as it is the most widely known and deployed and illustrates the fundamental problems that any flash translation layer must solve.

The Intel FTL uses several data structures summarized in Figure 2.4. Two separate mappings are maintained by the flash translation layer. First, virtual block numbers are mapped to a pair consisting of a logical erase block number and a page index within the erase block. This is the Virtual Block Map. Second, logical erase blocks are mapped separately to physical erase blocks by the Erase Block Map. Since erase blocks are large, the size of this second mapping is modest enough to store efficiently in main memory. The Virtual Block Map, on the other hand, may be quite large and so it is partially stored in flash. When it becomes necessary to copy the valid or live contents of an erase block, only the Erase Block Map and not the Virtual Block Map need be updated. The separation of the two mappings simplifies the procedure for moving live data from an erase block that is about to be reclaimed to a new erase block that has yet to be populated.

The Virtual Block Map is actually a two-level hierarchical structure. The first level is stored entirely in RAM and is populated at initialization time by scanning the media. This first level is known as the Virtual Page Map. The Virtual Page Map makes it possible to look up pages of the Virtual Block Map stored in flash. The Virtual Block Map pages stored in flash contain multiple entries each of which maps a virtual block address to a logical erase block and page within that erase block. Since the first level of the mapping is stored in main memory, it is possible to make fine-grained updates to it in an efficient manner. The second level, which is stored in flash, can not be as efficiently updated since each modification requires copying the entire page and writing it to a new location. To reduce the number of updates to pages in the Virtual Block Map, two parallel maps are kept: the primary and replacement maps. The primary map is dense, but the replacement map may be sparse with only some pages actually present. If an entry in the map is to be updated and is free in the replacement map, the original entry in the primary map is cleared and the new value placed in the replacement map. Otherwise, the primary map is updated and the entire page in the primary map must be re-written. During lookup, if the primary map entry is cleared, the value in the replacement map is used instead. The virtue of this approach is that sequential access leads to several mappings being moved from the primary to the replacement map before a new page must be written.

2.2.2 Garbage Collection

The flash translation layer uses an additional level of indirection to facilitate sub-page updates and to place new data in erase blocks such a way as to spread wear evenly. Over time, however, the flash device will still accumulate pages filled with unused data. This “garbage” must be identified and recycled. Garbage collection algorithms for flash not only make unused pages available for reuse, but also interact with wear leveling algorithms to influence device longevity. The efficiency of flash garbage col-

lection algorithms also affects the overall performance of the device, particularly in the case of writes. Some flash storage systems run the garbage collection algorithm purely on demand, which slows write performance as erase operations are time consuming. Modern SSDs typically run the garbage collector in the background to make use of idle periods and keep a reserve of previously erased blocks.

Flash garbage collection proceeds in four phases:

1. Choose erase block, B, to collect
2. Copy any live data in B to a new location
3. Update virtual-to-physical mapping for copied live data
4. Erase B and update any data structures for tracking free pages

Garbage collection thus requires an answer to the three important policy decisions in a flash storage system:

1. Which erase blocks should be chosen for garbage collection?
2. Where should the system place new data? These data may be writes submitted by an external client or live data discovered during collection that must be moved so an erase block can be reclaimed.
3. When should collection be triggered?

Two natural policies for selecting blocks for erasure are to select the erase block either on the basis of the fraction of garbage in the block or on the basis of wear (*i.e.,* number of previous erasures). The advantage of selecting the erase block based on the amount of garbage it contains is that fewer erasures are necessary to reclaim a given amount of space. The simplest incarnation of this type of heuristic is a garbage collection algorithm that chooses to reclaim the erase block with the largest number of pages containing unused or obsolete data. This approach is efficient in that each erase block reclamation frees the largest possible number of pages for reuse. The disadvantage of this simplistic approach is that it does not take wear leveling into

account. Erase blocks that contain many hot pages will see high levels of churn and thus be candidates for erasure under this policy. However, such blocks will also see increased wear as compared to blocks containing cold pages.

An example of a reclamation algorithm specifically designed to reduce wear is that patented by Lofgren *et al.* [52]. Lofgren's algorithm adds a counter to each erase block that tracks the number of erasures. A spare erase block is reserved for use by the system. When an erase block is collected, its erase counter is compared to that of the erase block with the fewest erasures in the system. If the difference exceeds some pre-determined threshold, the block to be collected is swapped with the spare. The goal of the heuristic is to identify heavily worn erase blocks and cold data and to move cold data to worn blocks.

Algorithms such as Lofgren's do introduce a further complication: the erase counters are not updated atomically and are therefore susceptible to loss in the case of a crash or system failure. One solution to this problem, short of using logging, is that of Marshall and Manning [57]. Marshall's approach is to store the counter for erase block B both in B and in the metadata header of another block, $B' \neq B$. In case of a crash, the counters can be recovered during a scan of the device at boot. Other approaches include estimating wear on the basis of erase block erasure latencies, which tend to increase over time [29], approximations such as that of Jou and Jeppesen [41], and randomization as in the case of JFFS2 [88], which we discuss in greater detail when we consider file systems in Section 2.3.

In the system described by Jou and Jeppesen, erase blocks are kept on a priority queue in RAM that is ordered by erase count. When an erase block is collected, its live contents are moved and the block is added to the in-memory priority queue, but the block is not erased. When fresh blocks are required, they are drawn from the queue and erased on demand. Blocks that have been erased but have not had their counters updated can be detected after a crash as the erase counter will contain all

ones. The initialization logic assigns these blocks the minimum wear value plus one as an approximation.

One approach to satisfying competing demands of efficient reclamation and even wear is to employ two different erase block selection methods: one method reclaims free space quickly and efficiently, for instance by choosing the erase block with the most garbage; a second method reclaims erase blocks with the goal of ensuring even wear. The choice of when to switch from one algorithm to another may be randomized, periodic, or a function of measured wear. Ban and Hasharon describe one such method [4]. Their method runs the wear leveling reclamation method either after a fixed number of writes or with a fixed probability after every write. The victim erase block is chosen uniformly at random from all erase blocks in the system so that, with high probability, all erase blocks will be subject to reclamation and wear leveling. Wells has described yet another approach to address the competing demands of efficient reclamation of free space and even wear [87]. In the scheme introduced by Wells, every erase block is assigned a score of the form:

$$\text{score}(B) = \alpha \times \text{garbage}(B) + \beta \times \left[\max_{B'} \text{erase}(B') - \text{erase}(B)\right]$$

where garbage(B) is a measure of the amount of unreferenced data in the erase block B and erase(B) is the number of times B has been erased. When α is large in comparison to β, the score favors efficiency over wear leveling. After a threshold number of blocks have been erased, blocks are re-scored with $\alpha < \beta$, emphasizing wear.

2.3 Existing Flash File Systems

One common approach to using flash memory is to expose it as an ordinary block device via the flash translation layer and use an existing file system on top of the block

device. In the case of removable devices, the FTL is often implemented in the flash device itself. USB flash drives, for instance, implement the FTL in a microcontroller on the drive and export a block device via the USB mass storage protocol. The device is most often formatted for Microsoft's FAT file system [21]. FAT is favored in the context of removable devices as it represents a lowest common denominator that is compatible with a wide range of operating systems and embedded devices such as cameras and cellular telephones. Other commonly used file systems include NTFS, FFS, Ext2, and Ext3 [10, 58, 60, 85]. The advantages of this approach are cross-platform compatibility in the case of removable media and simplicity even in the case of non-removable media. The changes to the storage stack can be confined to the hardware device or possibly the software device driver layer depending upon where the flash translation layer is implemented. Implementing a new file system is typically a significant undertaking in a modern operating system and so the ability to reuse existing file systems is a major advantage.

An alternative approach is to expose the flash hardware directly to a file system designed specifically for flash memory. The new flash-specific file system is responsible for managing wear and reclaiming unreferenced flash blocks. This end-to-end approach exposes both the details of the flash hardware and the higher level file system interface to a monolithic software stack optimized for the task at hand rather than for traditional magnetic media.

A simple example of an optimization that this enables is the ability to return unused blocks in deleted files to the pool of free flash blocks. This optimization can improve both wear management and performance as garbage collection algorithms typically work best when a meaningful fraction of the storage they manage is unallocated. If a file system designed for magnetic disks is layered on top of the FTL, there is no simple way for the FTL to realize that a block belonging to a deleted file may be marked as garbage. The block will be marked as free by the file system, but

not in the FTL. The FTL will only be able to recycle the block when it is re-written, indicating that the current contents are stale and should be discarded. Recently, the block layer in some operating systems, notably FreeBSD, has been augmented to allow a file system to notify the FTL that a block is free. This extension is typically referred to as *trim*.

The trim directive is one example of the optimizations possible when the file system encompasses both the low-level flash management and the high-level file system interface. A further advantage is that only a single block allocator is necessary. When a traditional file system is layered on top of the FTL, both the file system and the FTL must allocate storage blocks. Block allocation policies can be unified in a flash-specific file system.

Conventional file systems designed for magnetic disks modify both user data and file system metadata in place. This approach has the advantage that when a block is modified, the new data can be found in the same physical location as the old data that has been replaced. In-place modification has two shortcomings in the case of file systems for magnetic disks: data must be placed at particular physical locations, requiring the client to wait for the disk arm to be correctly positioned, and it introduces the possibility of data loss or inconsistency in the case of crash. As a result, most modern disk-based file systems are journaling file systems that use write-ahead logging to avoid inconsistency in the case of crash. Examples of this approach include Microsoft's NTFS, IBM's JFS, SGI's XFS, and Apple's HFS+ [2, 12, 16, 79].

Log-structured file systems take the journaling approach one step further by keeping everything, including user data, in the log. The log is composed of fixed-size segments and all modifications are appended to the end of the log. A background garbage collection process called the "cleaner" is responsible for releasing unreferenced segments in the log. The Sprite LFS file system is an example of a research log-structured file system for disks [74]. Disk-based log-structured file systems eschew

update-in-place for write performance and to aid in crash recovery. Write operations at the file system interface level become append operations applied to the log. This helps to improve write performance as the sequential write performance of disks is much higher than the random write performance. Read performance, on the other hand, may suffer depending on the workload. Logically sequential blocks of a file may not be physically contiguous in a log-structured file system if the blocks were written at different times. The resulting poor read performance poses a significant obstacle to log-structured file systems for disks.

The log-structured approach has not seen widespread commercial use in disk-based file systems. Unlike file systems for disks, however, most flash-specific file systems do take a log-structured approach to managing flash memory. Flash file systems must avoid update-in-place entirely as the underlying hardware does not support it. Furthermore, flash memory supports nearly uniform access times for all storage blocks. As a result, the log-structured approach to constructing file systems is a good match for flash memory.

Some of the earliest work that applied the log-structured principle to flash memory was that of Kawaguchi *et al.* [44]. They describe a transparent device driver that presents flash as a disk drive rather than a file system. However, they did identify the applicability of a log-structured storage layer that dynamically maps logical blocks to physical addresses, provides wear leveling, and hides bulk erasure latencies using a log similar to that of LFS [74]. Most flash translation layer and flash-specific file system implementation have adopted this basic approach.

The Microsoft Flash File System is one of the best known early file systems [49]. Dating from the early 1990s, MFFS was originally intended to be a standard file system for removable flash media. Douglis *et al.* report particularly poor write performance with MFFS [19]. In fact, they report that both read and write latency increases linearly with file size. The increase in I/O latency is due to the inefficient

representation of files. In MFFS, each file is represented by a linked list of blocks. When a portion of the file is overwritten, this linked list is updated by "patching" a new block into the list. The resulting fragmentation and long chains of blocks results in poor performance for both reads and writes. Later versions of the file system appear to have provided for reclaiming data blocks and "defragmenting" files in the process, but the details are not documented.

eNVy [89] is an early file system design effort for flash memory. Unlike more traditional file systems, it was designed to sit on the memory bus and operate at the word rather than block level. Since it also provides non-volatile storage, it takes the place of both DRAM and disk. eNVy uses flash memory as fast non-volatile storage, a battery-backed SRAM module as a non-volatile cache for combining writes into the same flash block for performance, and copy-on-write page management to deal with bulk erasures. The processor in the eNVy hardware module directly addresses both a NOR flash array and the SRAM cache. Reads are serviced either out of the SRAM cache or directly from the NOR flash array. Write requests that do not hit in the cache are serviced by copying a flash page into the cache and modifying the cache page.

JFFS2, the successor to Axis Communication's Journaling Flash File System (JFFS) for embedded Linux platforms, is a POSIX compliant file system for NOR flash developed by David Woodhouse at RedHat [88]. JFFS2 files are named by versioned I-node numbers which are not reused by the system. Files are stored as a sequence of *nodes* which contain a copy of the relevant I-node and a either a contiguous range of data from a user file or a directory entry in the form of a name and I-node pair. The JFFS2 log consists of a linked list of nodes and is reconstructed at mount time by scanning the entire device. JFFS2 keeps a hash table in memory from I-node number to the most recent version of the I-node in flash. JFFS2 also keeps an in-memory data structure for each node. This data structure is kept on a

linked list ordered by physical address for garbage collection and on a separate per-file linked list. When new data is written to the file system, the in-memory metadata is updated and a new node is written to flash. While JFFS2 is very simple, it requires significant RAM resources to run and requires a scan of the flash device at mount time. In practice, this restricts it to small file systems and embedded devices.

YAFFS, or Yet Another Flash File System, is a NAND-specific file system designed to improve upon JFFS and JFFS2's perceived shortcomings when run on NAND flash devices [55]. YAFFS files are stored in fixed sized blocks of 512 bytes, 1KB, or 2KB. Each block is labeled with a small header that contains file system metadata. Each file in the system, including directories, is described by a single metadata block that contains the file's name, permissions, and attributes; subsequent blocks contain file data. Like JFFS2, all on-flash metadata is stored in block headers, forcing the file system to scan the entire device and reconstruct in-memory metadata at mount time. YAFFS reduces its memory footprint by using a tree to store the mapping from file offsets to physical flash addresses. The leaves of the tree point to a small contiguous range of blocks in flash that are scanned to determine the final physical address. This is possible because blocks in YAFFS are self-describing. To avoid updating the per-file metadata on every write, each block header contains enough metadata to reconstruct the most current version of the file header, including the size of the file itself. YAFFS2 further improves upon this scheme by using a sequence number in each block to determine which blocks are a part of the current version of a file and which are garbage. The YAFFS authors argue that wear leveling is less important for NAND flash than it is for NOR flash. As a result, their primary method for wear leveling is the infrequent randomized reclamation of an erase block; any live data in this block is moved to a new erase block before erasure.

2.4 Flash Memory and Caching

Cache management algorithms have been studied extensively at least since the early 1960s. The vast majority of these algorithms are designed to manage limited main memory used as a cache for a much larger storage system built either from individual disks or from arrays of disks. Relatively little attention has been paid in the literature to the problem of designing cache management algorithms specifically for flash memory based storage systems. Early work on flash-specific caching focused on the use of flash memory for mobile computers to reduce power consumption by replacing a conventional magnetic disk entirely [19], by building a second-level cache from flash memory [56], and by building a write cache from flash to avoid spinning up a magnetic disk and caching whole files in flash [50].

In this section we provide a brief overview of recent work on cache management algorithms for flash based storage. Some flash-specific cache management algorithms such as FAB [38], or Flash Aware Buffer Management, are designed with embedded or application-specific environments in mind. Others, such as BPLRU [45], are designed to tackle the shortcomings inherent to flash memory, such as its random write performance. Still others, such as CFLRU [67] and CFDC [66] are intended to be flash-aware improvements on venerable algorithms such as Least Recently Used (LRU).

FAB [38] is designed specifically for portable music player devices. As a result, FAB is designed for workloads in which the majority of write operations are sequential. FAB seeks to achieve three specific technical goals:

1. Minimize the number of write and erase operations since they are a significant source of I/O latency in flash storage systems

2. Encourage what the authors term a "switch merge" in which entire erase blocks are collected at once by swapping a block of garbage with a new log block with live data
3. Maximize overwrite of hot flash blocks in the buffer cache so as to avoid writes and the creation of garbage in flash memory

The FAB algorithm attempts to satisfy these goals using a combination of a victim cache block selection strategy and a buffer cache flushing strategy. The victim selection strategy selects an *erase block* rather than a single cache block. The erase block chosen is that erase block that has the most resident cache blocks. Any ties are broken using an LRU policy at the erase-block level. All cache blocks belonging to the victim erase block are flushed at once. One drawback of FAB is that it may flush recently used data blocks if the corresponding erase block has the largest number of pages in the buffer cache. This occurs because erase block level eviction occurs before data block recency is considered during the victim selection process.

Block Padding LRU, or BPLRU [45], was specifically intended to improve the random write performance of flash memory devices. Like FAB, BPLRU uses an erase block level strategy. Data blocks are grouped by erase block and entire erase blocks are placed on an LRU list; all of the data blocks in an erase block are flushed as a unit. Furthermore, when BPLRU flushes victim data blocks, it will first read any data blocks not already in the cache that belong to the same erase block as the victims. This process is referred to as block padding and encourages entire erase blocks to be replaced rather than requiring the garbage collector to sift through several erase blocks to identify live data. Finally, BPLRU attempts to identify sequential access and places blocks written sequentially at the end of the LRU list under the assumption that they are least likely to be accessed in the near future.

Recently Evicted First (REF) [77], is a cache management algorithm for flash memory that is designed to integrate with a flash translation layer that uses a log

to manage updates. Much like FAB, REF chooses victim erase blocks rather than individual victim data blocks. Only data blocks residing in the victim erase blocks may be ejected from the cache. REF attempts to select victim erase blocks that correspond to FTL's active log blocks in order to improve the efficiency of garbage collection. Finally, REF takes data block level recency into account, unlike FAB and BPLRU. That is, data blocks that have not been used recently may be ejected even when other data blocks in the same erase block have been used recently and are retained in the cache.

Clean First LRU, or CFLRU [67], is a flash-aware modification of LRU which takes into consideration the higher cost of writes as opposed to reads in flash storage systems. CFLRU deliberately keeps some number of dirty pages in the buffer cache in an effort to reduce the number of writes to the flash device. CFLRU partitions the LRU list into two regions: the working region and the clean-first region. The working region contains recently used buffers and most cache hits. The clean-first region contains candidates for eviction. During victim selection, CFLRU prioritizes clean buffers in the clean-first region over dirty buffers. The size of the clean-first region is a parameter referred to as the *window size*; the authors provide both static and dynamic methods to select the window size. By prioritizing clean buffers over dirty buffers, CFLRU increases the effective size of the cache for dirty buffers and therefore reduces the number of writes to flash. However, clean buffers need not be at the tail of the LRU chain, forcing an expensive linear search. Worse, clean buffers tend to stay in the clean-first region near the working region as they are always chosen over dirty buffers, increasing the length of the linear search. Furthermore, CFLRU, like LRU, is inefficient when the workload contains long sequential access patterns as hot buffers are flushed by the sequentially accessed buffers.

Clean First Dirty Clustered, or CFDC [66], attempts to address the shortcomings of CFLRU. CFDC, like CFLRU, divides the cache into two regions: the working

region and the priority region. The relative size of these two regions is a parameter called the *priority window*. The priority region is represented by two lists: the clean queue and the dirty queue. Dirty buffers in the priority region are held on the dirty queue and clean buffers in the priority region on the clean queue. When a buffer is evicted from the working region, it is placed on the appropriate queue in the priority region. When the cache fills and a victim buffer must be chosen to make room for a new data block, the victim is chosen from the clean queue if it is not empty and from the dirty queue otherwise.

The dirty queue is organized as a list of data block clusters rather than a list of individual data blocks. Clustering is intended to combat the problem of long sequential access patterns flushing the cache. A cluster is a list of data blocks that are physically close on the backing storage medium and is therefore similar to the block-level LRU of FAB and BPLRU. The buffers in a cluster are ordered not by logical block number but by time of last reference in the cache. When a buffer in a dirty cluster is referenced, it is moved to the working region.

Different cache replacement algorithms can be used to manage the dirty and working regions. By default, the working region uses LRU to manage the buffer queue. The priority region orders the clean and dirty queues according to a priority function. The priority function tends to assign a lower priority to large clusters since flash devices tend to process larger clusters more efficiently as they exhibit greater spatial locality. It is also assumed that larger clusters are more likely to be the result of sequential access and therefore less likely to be accessed again in the near future.

2.5 Existing Flash Hardware

Flash storage has become a popular alternative to traditional magnetic disks in the last several years. As a result, most major storage vendors now offer products that

incorporate flash memory. These products run the gamut from removable media such as USB flash drives to large storage arrays that incorporate DRAM, flash, and magnetic disks in a single product. The most visible use of flash memory for most consumers is in consumer products such as MP3 players and smart phones. More recently, network routers and content delivery network appliances have begun to include embedded flash as a high-speed cache. The Cisco Content Delivery Engine [14] is one example of such a product.

In this section we provide a brief overview of commercial flash disks currently available. At one end of the spectrum are conventional magnetic disks with built-in NAND flash caches and SSDs packaged in a traditional disk form factor and intended as direct disk replacements. At the other end are sophisticated devices that connect directly to the I/O bus and use specialized device drivers rather than the existing low-level disk interface.

Both Seagate and Intel have introduced products that seek to provide the benefits of NAND flash without upsetting the existing storage infrastructure. Intel introduced its TurboMemory [59] product line in 2005 as a separate PCIe or mini-PCIe device intended for the laptop market. Subsequently, it has produced core chipsets, such as the X58 Express chipset for the x86 market that incorporate TurboMemory directly on the motherboard. TurboMemory acts as a flash memory cache in front of the existing magnetic disk drive to improve I/O performance, particularly random I/O performance, transparently. TurboMemory modules are intended to support existing operating system mechanisms such as Microsoft's ReadyDrive and ReadyBoost [72]. ReadyBoost uses a NAND flash SSD such as a USB thumb drive or a special purpose device such as TurboMemory as an extension of main memory. That is, ReadyBoost uses a NAND flash device to improve system throughput by providing a high performance swap partition. ReadyDrive, on the other hand, uses recent extensions to the ATA disk interface to cache data destined for a magnetic disk. ReadyDrive may

cache this data in a device such as a TurboMemory module or the disk itself may include a NAND flash cache. So-called hybrid disk drives are conventional magnetic disk drives with additional NAND flash memory used by the on-board disk controller as a cache. Hybrid drives may spin down the disk platters to save power while still serving data out of the flash cache. If a request to the disk misses in the cache, the disk is spun up on demand. Seagate has recently introduce a line of hybrid disks under the Momentus [75] name.

The most commonly available solid state disks are those that are designed to be direct replacements for magnetic disks. SSDs packaged in the same form-factor as magnetic disks have become popular, particularly in the consumer market, due to the ease with which they can be deployed. These devices use the same physical packages, the same electrical interconnects, and the same storage protocols and host bus adapters (HBAs) as magnetic disks. The design and implementation of this type of SSD is constrained by a form factor and interface designed for magnetic disks. As a result, these SSDs export only a simple block interface and must rely on a resource-constrained microcontroller to manage the underlying flash.

Typical examples of flash memory based SSDs using existing disk form factors include the Intel X25-E enterprise flash disk and the Sun SSD offering [36, 62]. Few technical details on the internals of these devices are publicly available. They do support so-called Native Command Queuing (NCQ), the Serial ATA equivalent of SCSI's Tagged Command Queuing. NCQ allows the SSD to receive multiple read or write requests at once and to respond to them out of order, allowing the device to take advantage of modest parallelism in the I/O request stream. Basic microbenchmark results for these devices may be found in Table 2.1. As a baseline for comparison, we also show the performance of a typical high-performance magnetic disk using Fiber Channel Arbitrated Loop (FC-AL), in this case a Seagate Cheetah 15K.7 disk. The flash storage devices provide a modest improvement in sequential read and write

	Disk	Intel X-25E	Sun SSD	Sun F20	FusionIO
Interconnect	FC-AL	SATA-II	SATA-II	PCIe	PCIe
Random Read	3.4ms	$75\mu s$	$90\mu s$	$220\mu s$	$8-50\mu s$
Random Write	3.9ms	$85\mu s$	$90\mu s$	$320\mu s$	$\sim$ 10-50μs
Random Read IOPS	300	35K	35K	101K	140K
Random Write IOPS	250	3.3K	3.3K	88K	135K
Seq. Read	116-194 MB/s	250 MB/s	250 MB/s	1092 MB/s	770 MB/s
Seq. Write	116-194 MB/s	170 MB/s	170 MB/s	500 MB/s	750 MB/s
Idle Power	~9W	60mW	100mW	?	3-5W
Active Power	12-16W	2.4W	2.1W	16.5W	10-17W

Table 2.1: Comparison of the Performance Characteristics of a Modern Fibre-Channel Disk, the Intel X-25E SSD, Sun Enterprise SSD, Sun Flash Accelerator F20, and the FusionIO ioDrive SSD

performance over existing high-performance magnetic disks. Random I/O latency, on the other hand, is roughly fifty times better and typical power dissipation is two orders of magnitude lower.

The second class of solid state disks are those that do not use the traditional HBA and interconnect. Two of the most widely used examples in this second class are the Sun Flash Accelerator F20 [62] and the FusionIO ioDrive [23]. Both of these devices connect directly to the PCI Express (PCIe) I/O bus rather than via a traditional HBA. Both also provide a higher degree of parallelism and performance than more traditional disk-like SSDs, however they are architecturally quite different. Table 2.1 shows the advertised performance of the Sun Flash Accelerator F20 with four Disks on Modules (DOMs) and the FusionIO ioDrive in its 160GB SLC NAND flash configuration. Both devices enjoy higher throughput, as measured in IOPS, when multiple concurrent threads issue requests. In the case of the Sun device, the performance numbers assume 32 concurrent threads; FusionIO does not state the number of threads used to collect the numbers reported in its specification sheet, but gen-

erally speaking peak performance is realized with about sixteen threads for writes and thirty-two threads for reads. It should also be noted that the performance numbers advertised in this table are those available at the time of writing in order to make an apples-to-apples comparison. The experimental results described in subsequent chapters were conducted with an earlier version of the FusionIO ioDrive which, as a pre-production beta sample, did not provide the same level of performance as currently shipping units.

The Sun Flash Accelerator is a PCIe card that consists of several Disk on Modules (DOMs) connected to a single on-board Serial Attached SCSI (SAS) HBA. In the typical configuration, there are four such DOMs, each consisting of 32GB of enterprise grade SLC NAND flash and 64MB of DRAM. Of the 32GB of NAND flash in each DOM, 24GB is available for use by consumers; the remaining 8GB is preserved as spare capacity to help improve performance and device longevity. Each DOM has its own SATA-II interface and flash controller that is responsible for garbage collection and bad block mapping. Front-end I/O requests to an individual DOM are automatically striped across the back end flash chips in the DOM. A super-capacitor provides backup power to the entire device in the case of failure and allows the DRAM buffers in each DOM to be flushed to stable storage. Each DOM is treated as a separate disk drive by the host operating system.

The Fusion ioDrive is also implemented as a PCIe card, however it uses a novel partitioning between the hardware and host device driver to achieve high performance. The overarching design philosophy is to separate the data and control paths and to implement the control path in the device driver and the data path in hardware. The data path on the ioDrive card contains numerous individual flash memory packages arranged in parallel and connected to the host via PCI Express. As a consequence, the device achieves highest throughput with moderate parallelism in the I/O request stream. The use of PCI Express rather than an existing storage interface such as SCSI

or SATA simplifies the partitioning of control and data paths between the hardware and device driver. The current hardware implementation of the I/O bus interface logic in the ioDrive consumes significant power, accounting for the high overall power usage of the device even when idle.

The device provides hardware support of checksum generation and checking to allow for the detection and correction of errors in case of the failure of individual flash chips. Metadata is stored on the device in terms of physical addresses rather than virtual addresses in order to simplify the hardware and allow greater throughput at lower economic cost. While individual flash pages are relatively small (512 bytes), erase blocks are several megabytes in size in order to amortize the cost of bulk erase operations.

The mapping between the logical addresses used by the block device layer and the physical addresses used by the hardware itself is maintained by the kernel device driver. The mapping between 64-bit logical addresses and physical addresses is maintained using a variation on B-trees stored in main memory[1]. Each address points to a 512-byte flash memory page, allowing a logical address space of 2^{73} bytes. Updates are made stable by recording them in a log-structured fashion: the hardware interface is append-only. The device driver is also responsible for reclaiming unused storage using a garbage collection algorithm. Bulk erasure scheduling and wear leveling algorithms for flash endurance are integrated into the garbage collection component of the device driver. A primary rationale for implementing the logical to physical address translation and garbage collection in the device driver rather than in an embedded processor on the ioDrive itself is that the device driver can automatically take advantage of improvements in processor and memory bus performance on commodity hardware without requiring significant design work on a proprietary embedded platform. This

[1]More recent versions have provisions for storing only a portion of the mapping in memory and dynamically paging it to flash

approach does have the drawback of requiring potentially significant processor and memory resources on the host.

2.6 Summary

NAND flash memory has become an important ingredient in high performance storage systems. Unlike magnetic disks, it contains no mechanical components and therefore does not suffer from mechanical positioning delay. As a result, NAND flash benefits from continued rapid improvements in process technology and offers exceptional random I/O performance. NAND flash does present its own set of challenges, however. Disks continue to be competitive with flash for sequential access patterns and on a cost per gigabyte basis. Moreover, flash requires sophisticated algorithms in the form of the flash translation layer in order to achieve high reliability, good write performance, and long life.

Most flash devices on the market have continued to use the existing block storage device interface. This approach has the advantage of backwards compatibility and allows existing file systems designed for magnetic disks to be deployed on flash memory devices with little or no change. There are a number of NAND flash specific file systems, including both JFFS2 and YAFFS, but these file systems are designed for small-footprint embedded applications and expect to manage the raw flash device directly.

Early flash devices were designed for embedded and shock-sensitive applications. These applications have expanded to include a variety of consumer electronics such as MP3 players and digital cameras as well as high performance network routers and Content Distribution Network (CDN) appliances. A number of vendors now sell flash disks that use the same electrical interface as conventional magnetic disks. More recently, several companies have begun to offer high-performance flash storage devices

that interface directly with the I/O bus and provide a potentially richer interface, higher concurrency, greater throughput, and average I/O latencies as low as $10\mu s$ as in the case of the FusionIO ioDrive.

Chapter 3

Rethinking the Storage Layer

The low-level abstraction and interface exported by most modern storage systems has not changed appreciably since the advent of modern magnetic disks. Whether the underlying storage device is a single disk or a large and sophisticated disk array with a multi-terabyte cache, the basic abstraction is the linear array of fixed size data blocks and the interface consists of four basic primitives: `read`, `write`, `seek`, and `sync`. This low-level interface to primary storage forms the basis for the wide variety of sophisticated storage systems in use today, including local file systems, database management systems, and the wide variety of networked and distributed applications found in the modern data center.

The logical block abstraction and the minimalist interface to primary storage has the twin advantages of simplicity and universality. Even with existing magnetic storage systems, however, it oversimplifies reality. In most cases, applications and even file systems and database storage systems must "reverse engineer" the true geometry of a modern disk from its performance behavior. Tuning applications to modern disk arrays is an even darker art with encoding, striping, and caching algorithms interacting. The increasing popularity of flash memory as a primary storage medium in the data center promises to simplify some of these concerns by providing high perfor-

mance random access without many of the quirks of magnetic disks and disk arrays such as positioning delay, stripe size, and so on.

Unfortunately, flash introduces its own set of performance anomalies. Whereas reads and writes to disk take similar amounts of time to execute, writes to flash memory are significantly more costly than reads. Furthermore, flash excels at random I/O, but disks are better able to keep pace when it comes to sequential I/O. In order to overcome these potential performance problems, flash storage devices have introduced the so-called flash translation layer which most often uses a variation on log-structured storage to provide good write performance. As a result, flash storage systems have pushed what were once higher level storage system questions down toward the hardware and firmware. These questions include block allocation, crash recovery, and layout. We believe that the resulting restructuring of the storage stack provides an outstanding opportunity to reconsider the basic storage abstractions and interfaces.

This chapter presents two important aspects of our approach to flash storage: (1) the introduction of new layers of abstraction for flash memory storage systems which yield substantial benefits in terms of both simplicity and performance (2) a virtualized flash storage layer, which provides a very large address space and implements dynamic mapping to hide bulk erasure latencies and to perform wear leveling. The virtualized flash storage layer is one particular proposed application programming interface (API) to the new layers of abstraction. In subsequent chapters, we introduce the design, implementation, and evaluation of DFS, a file system which takes full advantage of the new abstractions and the virtualized flash storage layer.

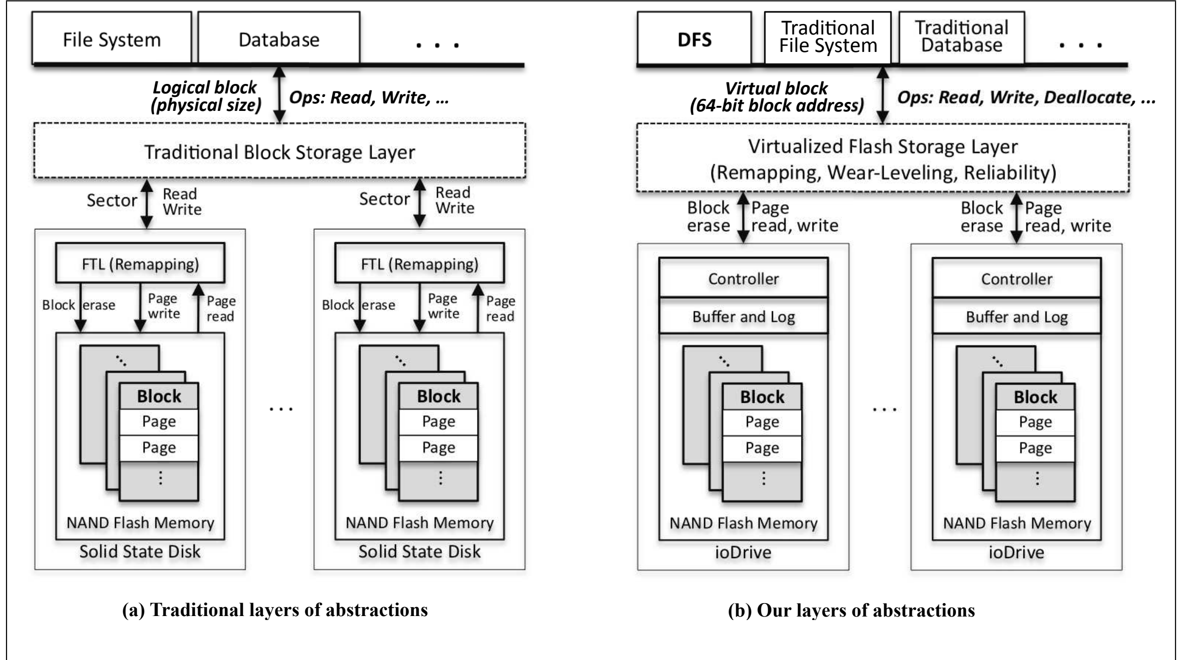

Figure 3.1: Flash Storage Abstractions

3.1 Existing vs. New Abstraction Layers

Figure 3.1 shows the architecture block diagrams for existing flash storage systems as well as our proposed architecture. The traditional approach, as shown on the left, is to package flash memory as a solid-state disk that uses a conventional disk interface such as SATA or SCSI for both the electrical and programming interfaces. An advanced SSD implements a flash translation layer in its on-board controller that maintains a dynamic mapping from logical blocks to physical flash pages to hide bulk erasure latencies and to perform wear leveling. Since the SSD uses the same interface as a magnetic disk drive, it supports the traditional block storage software layer which can be either a simple device driver or a sophisticated volume manager. The block storage layer then supports traditional file systems, database systems, and other software designed for magnetic disk drives. This approach has the advantage of disrupting neither the application-kernel interface nor the kernel-physical storage interface. On the other hand, it has a relatively thick software stack and makes it

difficult for the software layers and hardware to take full advantage of the benefits of flash memory.

We advocate an architecture in which a greatly simplified file system is built on top of a virtualized flash storage layer implemented by the cooperation of the device driver and novel flash storage controller hardware. The controller exposes direct access to flash memory chips to the virtualized flash storage layer. The virtualized flash storage layer exports an API that legacy applications can continue to use without disruption. The virtualized flash storage layer also exports an interface that a simplified file system such as DFS can use to achieve improved performance.

The virtualized flash storage layer is implemented at the device driver level which can freely cooperate with specific hardware support offered by the flash memory controller. The current implementation of the virtualized flash storage layer exports a large virtual block addressed space and maps it to physical flash pages. It handles multiple flash devices and uses a log-structured allocation strategy to hide bulk erasure latencies, perform wear leveling, and handle bad page recovery. This approach combines the virtualization and flash translation layer instead of pushing the FTL into the physical disk controller layer. The virtualized flash storage layer can still provide backward compatibility to run existing file systems and database systems. The existing software can benefit from the intelligence in the device driver and hardware rather than having to implement that functionality independently in order to use flash memory. More importantly, flash devices are free to export a richer interface than that exposed by disk-based interfaces.

DFS is designed to utilize the functionality provided by the virtualized flash storage layer. In addition to leveraging the support for wear leveling and for hiding the latency of bulk erasures, DFS uses the virtualized flash storage layer to perform file block allocations and reclamations and uses atomic flash page updates for crash recovery. This architecture allows the virtualized flash storage layer to provide an

object-based interface. Our main observation is that the separation of the file system from block allocations allows the storage hardware and block management algorithms to evolve jointly and independently from the file system and user-level applications. This approach makes it easier for the block management algorithms to take advantage of improvements in the underlying storage subsystem.

3.2 Storage Interface

In this section, we describe a richer interface to flash memory based primary storage than the traditional block-oriented storage interface. The interface presented here consists of a set of primitives for storage management that a file system or database system can use to manage flash storage. Our description of the storage API is in four parts: the primitives for manipulating objects, the core block storage API, the key-value store API, and additional primitives for atomic updates. We present a concrete implementation of the abstract storge interface in next section.

The basic unit of storage is the *object*, which is the low-level unit of naming. Garth Gibson *et al.* introduced the notion of object-based storage in their work on Network-Attached Secure Disks [26]. Object-based storage, as described by Gibson, consists of a set of flexibly sized containers called objects. An object has both data and metadata storage associated with it. Object metadata is stored, formatted, and interpreted by the storage system. The data associated with an object is an uninterpreted sequence of bytes. The object-based storage system is responsible for managing objects, object metadata, and the underlying physical storage.

In our proposed interface to flash memory, objects are named containers for uninterpreted data along with system defined metadata. An object is represented by a sequence of fixed-sized logical blocks each of which is identified by its logical block number or position in the sequence. Logical blocks are the unit of allocation and data

transfer and are the same size as the pages of the underlying flash memory. Although an object is logically a sequence of blocks, not all logical blocks in the sequence need be backed by allocated physical storage. Rather, objects may be sparse and regions of an object that are not backed by an allocated storage block are defined to contain all zeros.

The naming scheme for objects is flat – that is, objects are not named in a hierarchical or structured manner as they are in file systems. In general, a single file at the file system level may be represented by one or more objects at the object store level. Some implementations may elect to maintain metadata such as access control, ownership, quotas, access times, and so forth on a per-object basis. In the interest of simplicity, we expect a single pool of objects to be managed on behalf of a single client application, whether it be a database system, file system, or other program. The approach could be extended to allow the interface to manage separate volumes or pools of objects on behalf of multiple users and applications, perhaps even permitting over-commitment of resources.

1. ALLOCATE($\cdot$)

   The `allocate` operation allocates a fresh object, assigns a unique object ID, and initializes any per-object metadata.

2. DEALLOCATE(OID)

   The `deallocate` operation releases any storage blocks associated with a given object, including any per-object metadata, and frees the object ID for reuse.

3. UPDATE(OID, ...)

   The `update` operation allows a client to update user-defined metadata fields. For instance, a file system might choose to keep metadata such as a file's owner, group, or access rights in the metadata associated with the underlying object.

4. QUERY(OID)

   The `query` operation returns any metadata associated with a given OID.

Figure 3.2: Storage API: Object Interface Primitives

There are four primitives for manipulating objects as described in Figure 3.2: allocate, deallocate, update, and query. Allocating an object requires allocating a fresh object identifier and setting aside storage for tracking the object itself and any metadata associated with. Deallocating the object releases these resources for reuse. The update and query operations allow a client to update and interrogate the metadata associated with an object.

The next set of storage primitives consists of the block I/O primitives. Each block I/O primitive takes as an argument an object identifier which specifies the target of the operation. There may be multiple concurrent block I/O operations outstanding against a single object at the same time. The storage system makes no guarantees about the order in which concurrent block I/O operations execute or retire. Instead, the interface provides a barrier mechanism whereby all block I/O operations issued before the barrier are guaranteed to have completed before the barrier operation retires and returns a successful status to the client. The most common use of the storage barrier operation is to implement the UNIX `fsync` operation by waiting synchronously for a write barrier to complete.

The block I/O interface does not distinguish between storage allocation and update. That is, there is no explicit primitive to force the allocation of physical storage backing a logical block in an object. Instead, all allocation happens in the context of an update operation. When an update operation is issued, the storage system automatically allocates physical storage on an as-needed basis. An update to a previously allocated range of logical blocks may also create garbage that the system will subsequently reclaim during garbage collection.

The block I/O interface provides two separate primitives to handle sparse objects with ranges of logical blocks that are not backed by physical storage. The first, `trim`, allows a client to mark a range of logical block addresses as unused and therefore garbage subject to reclamation. The resulting "hole" in an object is defined to contain

1. READ(OID, OFF, LEN)

   The `read` operation returns the contents of the logical block range of length `len` starting at offset `off` in object `oid`.

2. UPDATE(OID, OFF, LEN, BUF)

   The `update` operation updates object `oid` so that the logical block range of length `len` starting at offset `off` contains the value held in the buffer `buf`. The storage system is responsible for allocating physical storage blocks as necessary.

3. TRIM(OID, OFF, LEN)

   The `trim` operation releases any physical storage associated with the logical block range of length `len` starting at offset `off` in object `oid`. Following the trim, the region is defined to contain all zeros.

4. EXISTS?(OID, OFF, LEN)

   The `exists?` operation determines whether or not the logical block range of length `len` starting at offset `off` in object `oid` is backed by physical storage.

5. COPY(OID_0, OFF_0, OID_1, OFF_1, LEN)

   The `copy` operation logically copies `len` bytes of data in object oid_0 starting at offset off_0 to the non-overlapping range of the same length starting at offset off_1 in object oid_1. The storage system is responsible for allocating physical storage blocks as necessary. Furthermore, where possible, the storage system should use copy-on-write techniques to avoid unnecessary data movement.

6. MOVE(OID_0, OFF_0, OID_1, OFF_1LEN, BUF)

   The `move` operation logically moves `len` bytes of data in object oid_0 starting at offset off_0 to the non-overlapping range of the same length starting at offset off_1 in object oid_1. It is equivalent to a `copy` operation followed by a `trim` of the source interval except that an implementation may choose to make `move` a single atomic operation.

7. BARRIER(OID)

   The `barrier` operation guarantees that before it completes, all preceding I/O operations to object `oid` have completed and any updates have been made stable.

Figure 3.3: Storage API: Block Storage Interface Primitives

only zeros following the successful completion of the `trim`. The second primitive for manipulating sparse objects is `exists?` which allows a client to determine whether or

not a range of logical block addresses is backed by physical storage. It is not possible to determine the allocation status of a range by using the `read` operation as it will return a buffer filled with zeros in two cases: when the underlying storage block is not allocated and when it is allocated but contains all zeros. The `exists?` operation is useful for archival programs that wish to store or transmit sparse files in an efficient manner.

The block storage API also provides primitives for copying or moving block ranges. Much as the virtual memory interface allows ranges of pages to be moved by adjusting the virtual address mappings, the block storage `copy` and `move` allow the mapping from object and logical block address to physical storage to be updated by an application. The copy operation copies a range of logical blocks to a new location. We expect that, where possible, the copy operation will be implemented as a copy-on-write update to the logical-to-physical mapping maintained by the storage system. This is another instance in which our interface takes advantage of the level of indirection already introduced in the flash translation layer to provide performance and wear leveling. Making `copy` and `move` primitive operations allows clients to move data in primary storage without requiring two transactions and data transfers across the I/O bus. The `move` operation is logically equivalent to a `copy` and `trim` of the source range except in the case of a crash. An implementation may choose to make the `trim` operation atomic.

Most modern file systems treat files as opaque, uninterpreted byte streams. It is up to the user application, not the file system, to impose internal structure on these files. The file as uninterpreted byte stream provides only a minimum of abstraction but allows for greater flexibility as altering the file system implementation for application specific purposes is often a prohibitive undertaking. Nonetheless, as the storage stack is pushed down into the device driver, firmware, and hardware of next-generation flash memory primary storage, we believe that there is a place for structured objects.

1. INIT(OID, FLAGS)

   The `init` operation initializes object `oid` for use as a key-value store. The `flags` parameter sets options for the newly created store such as whether or not to permit duplicate keys and in the case of unique keys whether or not to overwrite existing keys automatically. By default, keys must be unique.

2. INSERT(OID, KEY, VAL)

   The `insert` operation adds the key/value pair `key/val` to the store in object `oid`. By default, if a key-value pair with the same key already exists, it is replaced.

3. LOOKUP(OID, KEY)

   The `lookup` operation retrieves the value, if any, associated with `key` in the object `oid`. If there are multiple values with the same key, the first is returned along with a cursor to retrieve subsequent pairs.

4. REMOVE(OID, KEY)

   The `remove` operation removes the pair with key `key`. It is an error if there are multiple values with the same key.

5. REMOVE(OID, CUR)

   The `remove` operation removes the key/value pair referenced by the cursor `cur`.

6. FIRST(OID)

   The `first` operation returns the first item in object `oid`.

7. NEXT(OID, CUR)

   The `next` operation returns the key/value pair referenced by the cursor `cur` and advances `cur` to the next pair. The ordering of keys is implementation defined.

Figure 3.4: Storage API: Key-Value Pair Interface Primitives

Key-value pair stores are common primitives in applications that manage large data sets and in distributed applications in the data center. Key-value stores are most often implemented using variations on hashing, B-trees, or a combination of the two. The level of indirection and the physical storage management algorithms necessary to impose internal structure on objects in the object store is already present in the form of the flash translation layer. We are not aware of a commercial flash storage product that provides a key-value interface, but we have discussed the design and

implementation in some detail with FusionIO. In the meantime, structured key-value pair objects can be implemented in the device driver or at higher levels in the software stack.

The idea of operating-system imposed structure on files is not new. IBM introduced ISAM files that support both sequential and key-value access as a part of its line of mainframe operating systems such as MVS and its successors [34]. Similarly, the Files-11 file system in conjunction with Record Management Services provides key-value access under OpenVMS [31]. Figure 3.4 summarizes the programmatic interface to our key-value store for objects. Our current implementation is a software-only implementation in the kernel and does not support duplicates as we have not yet had a need for either duplicates or cursors. We believe that the interface is both simple and generic enough that it would benefit greatly from integration into the flash translation layer in devices such as the FusionIO ioDrive.

In addition to the object store, block, and key-value interfaces, we propose adding an explicit atomic update interface. Database systems and file systems have traditionally offered transactional semantics using variations on techniques such as write-ahead logging and shadow paging. We believe that flash memory based storage should provide primitives to support transactional updates natively rather than forcing applications to reinvent the wheel. Modern flash memory storage uses logging techniques to implement copy-on-write semantics for performance. It is natural to expose an abstract interface to these low-level mechanisms rather than force applications to duplicate the mechanism. Figure 3.5 summarizes our proposed interface for atomic updates in flash memory. There are two primitives: `atomicupdate(`$\texttt{buf}_0$`, ..., `$\texttt{buf}_n$`)` and `abort(txn)`. The `atomicupdate` primitive takes a list of buffer descriptors and ensures that all of the described updates are performed or none are and that all preceding atomic updates that involved overlapping regions have completed before the current operation begins. A buffer descriptor consists of an object ID, a logical

offset into the object, a size, and a buffer containing the new data. The atomic update primitive thus provides atomicity and isolation. The `abort` primitive allows a client to terminate an active atomic update and discard its contents if it has not yet committed.

1. ATOMICUPDATE(BUF_0, . . . , BUF_N)
   The `atomicupdate` operation takes as arguments a set of buffers which include OID, offset, length, and buffer contents. It returns a transaction handle, `txn`. The updates are either applied as a group or none are applied. The transaction cannot proceed until any preceding `atomicupdate` operations that affect an overlapping region have committed.

2. ABORT(TXN)
   The `abort` operation terminates an uncommitted transaction, `txn`, abandoning all updates associated with the transaction. The `abort` function returns an error code indicating whether or not the operation succeeded. A transaction abort may fail if transaction has already committed or due to a fault in the underlying storage medium that prevents successful rollback.

Figure 3.5: Storage API: Atomic Update Interface Primitives

3.3 Virtualized Flash Storage Layer

The virtualized flash storage layer is an implementation of our flash storage API as described in Section 3.2 that additionally provides a backwards compatible block storage interface. That is, it also supports the conventional `read/write/seek` interface. It was also designed to require minimal firmware and device driver modifications on the FusionIO platform in order to support a research prototype. The current implementation of the virtual flash storage layer implements the object interface as a separate in-kernel library on top of the FusionIO kernel device driver. The block storage API, described in Figure 3.3, is implemented as a library that makes direct use of the FusionIO device driver's services. Barriers are implemented through the existing kernel infrastructure as calls into the device driver. The current implementation of

the virtual flash storage layer does not natively support the key-value pair interface of Figure 3.4, although we have experimented with software-only implementations in the kernel and in user space. Furthermore, the atomic multi-block update interface of Figure 3.5 is not exposed outside of the FusionIO device driver.

The primary novel feature of the virtualized flash storage layer is the provision for a very large, virtual block-addressed space. There are three reasons for this design. First, it provides client software with the flexibility to directly access flash memory in a single level store fashion across multiple flash memory devices. Second, it hides the details of the mapping from virtual to physical flash memory pages. Third, the flat virtual block-addressed space provides clients with a backward compatible block storage interface.

The mapping from virtual blocks to physical flash memory pages deals with several flash memory issues. Flash memory pages are dynamically allocated and reclaimed to hide the latency of bulk erasures, to distribute writes evenly to physical pages for wear leveling, and to detect and recover bad pages to achieve high reliability. Unlike a conventional flash translation layer, the mapping supports a very large number of virtual pages – orders-of-magnitude larger than the available physical flash memory pages.

The virtualized flash storage layer directly supports the primary block storage operations of Figure 3.3 except that the object ID is encoded in the high-order bits of the virtual block address. One way to think of the virtualized flash storage layer is as an analog of traditional virtual memory wherein logical block addresses are akin to virtual memory addresses and physical flash block addresses are analogous to physical DRAM page frames. The object ID of our proposed storage interface is folded into the virtual address by dividing the virtual address range into separate fixed-sized regions. We take this approach in the interest of simplicity, but anticipate that

future commercial implementations may decide to use a more traditional object-store interface.

All operations are block-based operations, and the block size in the current implementation is 512 bytes. We anticipate that future versions of the virtualized flash storage layer will also support a move operation to allow data to be moved from one virtual address to another without incurring the cost of a read, write, and deallocate operation for each block to be copied. A move operation requires only updating the logical-to-physical mapping already maintained by the virtualized flash storage layer.

The current implementation of the virtualized flash storage layer is a combination of a Linux device driver and FusionIO's ioDrive special purpose hardware. The ioDrive is a PCI Express card densely populated with either 160GB or 320GB of SLC NAND flash memory. The software for the virtualized flash storage layer is implemented as a device driver in the host operating system and leverages hardware support from the ioDrive itself.

The ioDrive uses a novel partitioning of the virtualized flash storage layer between the hardware and device driver to achieve high performance. The overarching design philosophy is to separate the data and control paths and to implement the control path in the device driver and the data path in hardware. The data path on the ioDrive card contains numerous individual flash memory packages arranged in parallel and connected to the host via PCI Express. As a consequence, the device achieves highest throughput with moderate parallelism in the I/O request stream. The use of PCI Express rather than an existing storage interface such as SCSI or SATA simplifies the partitioning of control and data paths between the hardware and device driver.

The device provides hardware support of checksum generation and checking to allow for the detection and correction of errors in case of the failure of individual flash chips. Metadata is stored on the device in terms of physical addresses rather than virtual addresses in order to simplify the hardware and allow greater throughput

at lower economic cost. While individual flash pages are relatively small (512 bytes), erase blocks are several megabytes in size in order to amortize the cost of bulk erase operations.

The mapping between virtual and physical addresses is maintained by the kernel device driver. The mapping between 64-bit virtual addresses and physical addresses is maintained using a variation on B-trees in memory. Each address points to a 512-byte flash memory page, allowing a virtual address space of 2^{73} bytes. Updates are made stable by recording them in a log-structured fashion: the hardware interface is append-only. The device driver is also responsible for reclaiming unused storage using a garbage collection algorithm. Bulk erasure scheduling and wear leveling algorithms for flash endurance are integrated into the garbage collection component of the device driver.

A primary rationale for implementing the virtual to physical address translation and garbage collection in the device driver rather than in an embedded processor on the ioDrive itself is that the device driver can automatically take advantage of improvements in processor and memory bus performance on commodity hardware without requiring significant design work on a proprietary embedded platform. This approach does have the drawback of requiring potentially significant processor and memory resources on the host.

3.4 Summary

We have argued for a richer storage abstraction for modern storage systems. The conventional abstraction of a large linear array of fixed sized data blocks is straightforward to implement and simple to understand but does not accurately reflect the performance characteristics of modern storage systems, particularly storage systems that incorporate NAND flash memory. While storage vendors have rushed to add flash

to their products as it promises to offer outstanding random I/O performance and the potential for reduced power consumption, flash introduces its own set of challenges. In particular, writes to raw NAND flash are much more costly than are reads and disks continue to outperform flash for sequential I/O. The solution to these performance challenges has been to introduce a level of indirection – the flash translation layer – which uses log-structured techniques to overcome the write bottleneck. The advent of the flash translation layer has started a trend to push high-level design decisions down the storage stack. We believe that this presents an opportunity for innovation as the flash translation layer offers the ability to implement new abstractions at little incremental cost. These abstractions include object based storage, atomic multi-block update, key-value pair interfaces, and block re-mapping techniques reminiscent of virtual memory that can reduce the number of I/O transfers. In the next chapter we describe DFS, a file system intended to take advantage of these new abstractions.

Chapter 4

Rethinking the File System

DFS is a file system designed to take advantage of the new features exported by the virtualized flash storage layer as described in Chapter 3. DFS makes use of the very large virtualized address space exported by the flash storage layer, direct access to flash storage, and the exported crash recovery mechanisms. Perhaps most importantly, DFS is designed to allow the critical storage management functions that are typically the purview of a file system to be delegated to the virtualized flash storage layer. DFS itself retains responsibility for directory management and access control.

By delegating most allocation and storage management to the virtualized flash storage layer, we are able to simplify the design and implementation of DFS. As we saw in Chapter 2, the flash translation layer plays a critical role in achieving good performance in modern flash memory based storage systems. The virtualized flash storage layer is the result of rethinking and extending the traditional flash translation layer beyond its basic block-oriented interface. The virtualized flash storage layer provides new, powerful primitives that do not impose significant new overhead beyond that already required to provide good performance in flash storage systems. DFS takes advantage of these new primitives and in so doing is able to avoid un-

necessary duplication of functionality between DFS and the virtualized flash storage layer. This approach has the added advantage that it allows modern flash memory storage hardware and the software layer that manages it to co-evolve while requiring few if any changes to the file system. Since the file system in modern operating systems such as Linux is deeply enmeshed with other significant subsystems such as the VFS, the buffer cache, and the virtual memory system, the separation of concerns is an advantage from a software engineering standpoint as well as a performance standpoint.

We have implemented DFS as a loadable kernel module for the Linux 2.6 kernel. The DFS kernel module implements the traditional Unix file system APIs via the Linux VFS layer. It supports all of the usual file system methods, including read, write, open, close, create, remove, rename, and seek. DFS also implements memory mapped I/O. Because of the design and implementation of the `exec` system call in Linux, file systems must implement memory mapped I/O in order to support program execution: Linux requires demand paging of executables.

4.1 Storage Allocation

DFS delegates I-node and file data block storage management to the virtualized flash storage layer. The virtualized flash storage layer is responsible for physical block allocations and deallocations, for hiding the latency of bulk erasures, and for wear leveling. The virtualized flash storage layer also exports primitives that DFS uses to ensure crash-recoverability.

We have considered two design alternatives. The first is to have the virtualized storage layer export an object-based interface in addition to the more conventional block-based storage interface. In an object-based storage system, the basic unit of organization is the object, which is a container that has both data and metadata

associated with it. The external interface to the storage system refers to logically addressed blocks of named objects rather than to raw physical block storage. In an object-based storage system, the storage system is responsible for managing the physical storage assigned to individual objects. Higher level software, including file systems, manipulates the objects rather than the physical storage medium. The object storage interface allows clients to create and destroy objects and to read and write data blocks and metadata associated with the object. Objects themselves may be identified by an OID or Object Identifier, which is an opaque identifier assigned by the storage system. Early academic work on object-based interfaces to storage include Network Attached Secure Disks [26] which spawned subsequent standards such as the SCSI Object Based Storage Device (OSD) standard. Several SSD vendors, including FusionIO, have expressed renewed interest in object-based storage in general and the OSD standard in particular as part of their product offerings.

We envisage DFS using a separate storage system object to represent each file system object. That is, a separate object would be allocated in the virtualized flash storage layer to represent each I-node, directory, and ordinary file containing user data. This approach allows DFS to delegate the task of physical storage allocation and management to the virtualized flash storage layer. Furthermore, this approach allows DFS to make effective use of the object storage interface as the higher-level abstractions required by the file system map naturally onto the lower-level abstractions exported by the object-based virtualized flash storage layer.

The disadvantage of an explicitly object-based interface is the additional storage system complexity. The virtualized flash storage layer already must handle logical to physical address translation. The addition of the object abstraction requires it to manage multiple objects backed by the same pool of physical storage. In practice, it must already support a traditional block device interface as well for backwards compatibility with existing software and file systems.

The second approach, and the one that we ultimately adopted for our current implementation, is to have the virtualized flash storage layer export a block device addressed by logical rather than physical block addresses. Most of the fundamental storage primitives are the same, but the implementation of the virtualized flash storage layer in the absence of an explicit storage objects is greatly simplified. The logical to physical mapping can be configured when the device is initialized and need not be dense. The basic abstraction remains a linear array of fixed sized blocks, but the array may be much larger than the physical storage available, allowing DFS to use some bits of a logical address for its own purposes. When configured to support DFS, the device exports a very large, sparse logical block address space. In the current implementation, logical block addresses may be as large as 64 bits. When this is the case, not all logical block addresses can correspond to physical storage and the mapping between logical and physical block addresses must be sparse. Furthermore, some fraction of the physical storage will be reserved by the system for its own metadata and as overhead to improve the efficiency of garbage collection.

The device can also be configured to support existing file systems and applications for backwards compatibility. When this is the case, the logical block address space will be the same size as the available physical block address space. The resulting mapping from logical to physical block addresses will be dense and the SSD will appear to be an ordinary disk.

The second approach of using a large, sparse logical address space has the advantage of simplicity. It is straightforward to support existing file systems and applications in a backwards-compatible fashion. While DFS makes use of a large, sparse logical address space, existing applications make use of a much smaller, dense logical address space. In both cases, the virtualized flash storage layer serves the same basic function of maintaining a dynamic mapping from logical block addresses to physical flash pages. The only difference between supporting DFS and a conventional dense

block device is the size of the logical address space. Since the flash device must implement the same level of indirection to support wear leveling and garbage collection in both cases, this approach is a simple, low-overhead way to support DFS. The drawback is that the logical address space must be much larger than the physical address space in order to allow for fragmentation in the mapping between logical address ranges and file system objects.

In our current implementation of DFS on the FusionIO ioDrive, we have configured the hardware to export a sparse 64-bit logical block address space. This allows logical block addresses to fit naturally in an 8-byte word that can be manipulated efficiently with the usual arithmetic instructions. By default, each logical block on the ioDrive is 512 bytes. This yields a byte-granular logical address space of 2^{73} bytes, although all address arithmetic is performed on block addresses. DFS file system objects, including I-nodes, directories, and files are allocated as contiguous logical block ranges.

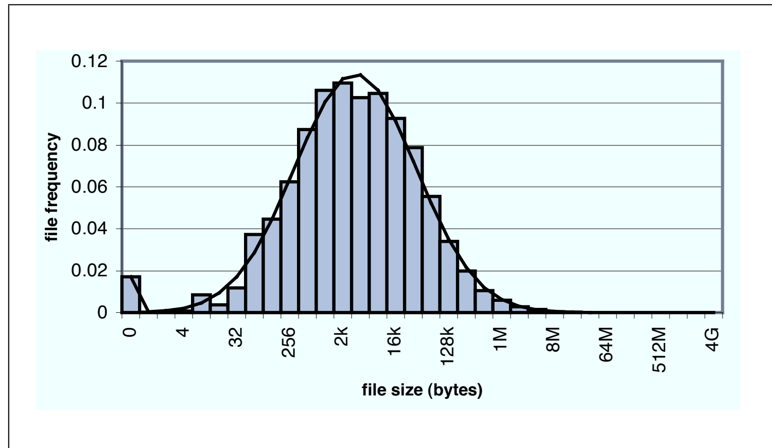

Figure 4.1: Histogram of observed file sizes as reported by Douceur and Bolosky [18]

DFS allocates logical address space in contiguous ranges called *allocation chunks*. The size of these chunks is configurable at file system initialization time but is 2^{32} blocks or 2TB by default. These allocation chunks are further subdivided into contiguous, equally sized *regions*. User files and directories are partitioned into two types: large and small. A large file occupies an entire allocation chunk whereas small files occupy at most one region. Multiple regions are packed into a single allocation chunk.

As a small file grows, it is eventually converted into a large file and its contents moved to a freshly allocated chunk. The current implementation must achieve this by copying the file contents, but we anticipate that future versions of the virtual flash storage layer will support changing the logical to physical translation map without having to copy data. The current implementation does not support remapping large files into the small file range should a file shrink.

When the file system is initialized, two parameters must be chosen: the maximum size of a small file and the size of an allocation chunk, which is also the maximum size of a DFS file. Both sizes must be powers of two. These two parameters are fixed once the file system is initialized and cannot be changed. It is possible to choose the parameters in a principled manner given the anticipated workload. There have been many studies of file size distributions in different environments, for instance those by Tanenbaum *et al.* [80] and Douceur and Bolosky [18]. Figure 4.1 shows the histogram of file sizes observed by Douceur and Bolosky in their study of over 10,000 file systems and 140 million files on the Microsoft corporate campus. The file systems studied were used for a variety of workloads, including technical support, development, and business and management functions. Figure 4.2 from Tanenbaum's study tells a similar story but also shows that the file size distribution for workstation workloads has been rel-

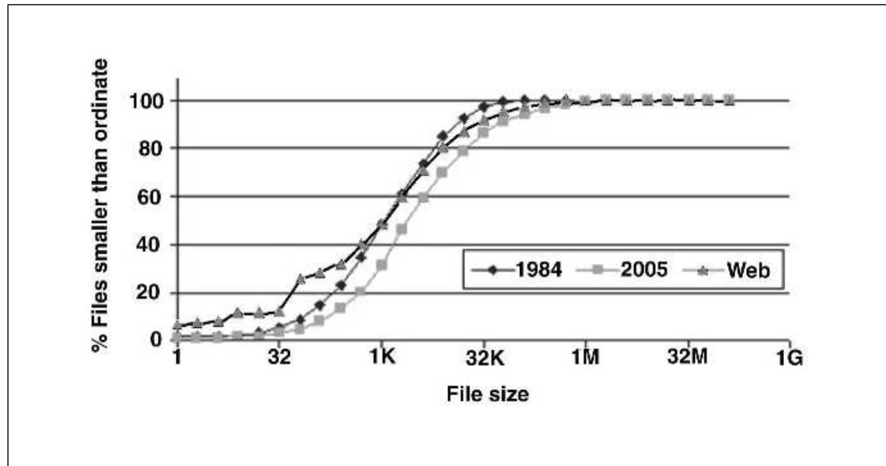

Figure 4.2: Cumulative distribution of observed file sizes as reported by Tanenbaum *et al.* [80]. The distribution for workstation workloads has been relatively stable over time.

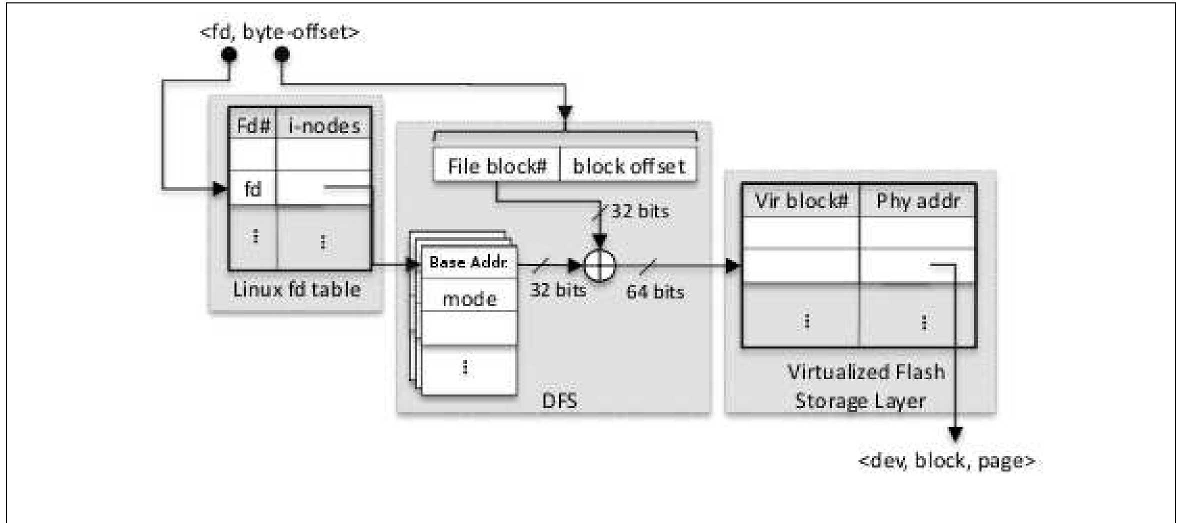

Figure 4.3: DFS logical block address mapping for large files; only the width of the file block number differs for small files

atively stable over time. Most files are small, but most bytes reside in large files. On the basis of the reported file size distributions, DFS by default sets the small file size to 32KB. The default allocation chunk size and therefore the default maximum file size is 2^{32} blocks which translates to 2^{41} bytes (2TB) on our development platform.

The current DFS implementation uses a 32-bit I-node number to identify individual files and directories and a 32-bit block offset into a file. This means that DFS can support up to $-1 + 2^{32}$ files and directories in total since the first I-node number is reserved for use by DFS itself. The largest supported file size is 2TB with 512-byte blocks since the block offset is 32 bits.

The DFS I-node stores basic per-file metadata. These data include the logical address extent assigned to the file, the type of the file, and access control information such as user and group. The I-node also stores creation, access, and modification times.

DFS assigns each a unique range of logical block addresses to each I-node. All read and write operations to a file operate on a block in this extent. The logical address of a file block is computed from the I-node's logical base address and the

logical block number. Figure 4.3 depicts the mapping from file descriptor and offset to logical block address maintained by DFS.

Since each file is represented by a single logical extent it is straightforward for DFS to combine multiple small I/O requests to adjacent regions into a single larger I/O. No complicated block layout policies are required at the file system layer. This strategy can improve performance because the flash device delivers higher transfer rates with larger I/Os. Our current implementation aggressively merges I/O requests; a more nuanced policy might improve performance further.

DFS leverages the three main operations supported by the virtualized flash storage layer: read from a logical block, write to a logical block, and discard a logical block range. The discard directive marks a logical block range as garbage for the garbage collector and ensures that subsequent reads to the range return only zeros. A version of the discard directive already exists in many flash devices as a hint to the garbage collector; DFS, by contrast, depends upon it to implement truncate and remove. It is also possible to interrogate a logical block range to determine if it contains allocated blocks. The current version of DFS does not make use of this feature, but it could be used by archival programs such as `tar` that have special representations for sparse files.

4.1.1 Crash Recovery

Modern file systems must be able to recover from unexpected crashes due to failures, particularly power failures. The degree of protection from data loss in the face of failure varies from one file system to the next. In general, there are three types of data loss that a file system may need to protect against. The most basic is the loss of the metadata used to track storage allocation. Since storage is recycled over the life time of the file system, it is necessary to maintain a mutable mapping from logical block number in a file to a particular physical location on the storage medium. If a

power failure occurs at an inopportune time, the stored copy of this mapping may become corrupt. For instance, if a block is removed from a free list before the crash but not yet assigned to a file, the block may be "leaked". This is a relatively benign fault, but it is also possible that a block may be assigned to multiple files as a result of corruption of the block allocator's data structures in which case unauthorized access may be granted to sensitive data. A second type of corruption that may occur is the corruption of higher-level metadata such as that stored in directories. If directories are corrupted, a file may be orphaned – *i.e.,* still be allocated but not appear in any directory. Other typical faults include directory entries that point to deleted files, incorrect permissions, owners, and so on. Most often, file systems use the same techniques to guarantee the consistency of directory updates as they do for block allocations. The third type of data loss that some file systems protect against is the loss of user data. Most modern file systems provide a system call such as the UNIX `fsync` system call that allows applications to flush the contents of the buffer cache for a particular file to ensure that all data for that file is made stable. A few file systems also provide for logging of application updates to a file. Logging of user data provides stronger consistency guarantees for the data contained in a user file. This additional guarantee comes at the expense of performance. Most applications that use the file system interface either ignore the consistency problem or provide their own facilities using the `fsync` system call.

The traditional UNIX approach to file system crash recovery is the `fsck` program. `Fsck` scans the entire file system to identify inconsistencies after a crash and attempts to correct them. `Fsck` is not a suitable solution to the crash recovery problem in modern operating systems for two reasons: first, it is not reliable as it cannot correct all problems that might arise from a crash. In fact, `fsck` has historically been associated with both data loss and security problems. Second, since `fsck` must examine the entire storage volume after a crash, it is only suitable as a method of last resort as

storage devices become prohibitively large to scan efficiently. Some versions of `fsck` support a so-called "background" mode which allows the file system to be mounted and used by applications while the file system is checked for inconsistencies. Even this optimization is insufficient due to the inordinate I/O bandwidth and time consumed by `fsck` on large modern storage systems.

Modern file systems have taken one of two approaches to eliminating the dependency on `fsck`. One is to order on-disk metadata updates to guarantee consistency of the metadata in stable storage. The most common example of this approach is soft updates as implemented in recent versions of FFS [24]. The more common approach taken by a variety of file systems including JFS, XFS, HFS+, and NTFS is write-ahead logging or journaling [76]. Logging file systems maintain the consistency of critical file system data structures by tracking updates in a log and ensuring that updates to the log are stable before modifying on-disk data structures. High performance implementations often rely on specialized hardware such as non-volatile random access memory (NVRAM), most often in the form of battery-backed DRAM, to make updates to the log both persistent and efficient.

NVRAM and file system level logging require complex implementations and introduce additional costs for the traditional file systems. NVRAM is typically used in high-end file systems so that the file system can achieve low-latency operations while providing fault isolations and avoiding data loss in case of power failures. The traditional logging approach is to log every write and performs group commits to reduce overhead. Logging writes to disk can impose significant overheads. A more efficient approach is to log updates to NVRAM, which is the method typically used in high-end file systems [32]. NVRAMs are typically implemented with battery-backed DRAM arrays on a PCI card. The NVRAM array typically provides several gigabytes of storage for logging. Achieving good performance with an NVRAM is a significant engineering challenge because every update must be logged in NVRAM before being

sent to stable storage. In a typical commodity PC based storage system, using a separate NVRAM card for logging means that each update must traverse the I/O bus an additional time, requiring additional resources and increasing I/O latency. In a network file system, each write will have to go through the I/O bus three times, once for the network interface card, once for NVRAM, and once for writing to disks.

Since flash memory is a form of NVRAM, DFS leverages the support from the virtualized flash storage layer to achieve crash-recoverability. The virtualized flash storage layer implements the basic functionality of crash recovery for the mapping from logical block addresses to physical flash storage locations. DFS leverages this property to provide crash recovery. Unlike traditional file systems that use and their own logging implementation, DFS piggybacks on the flash storage layer's log. When a DFS file system object is updated, DFS passes the request to the virtualized flash storage layer. If the request is a write request that requires allocation to satisfy, the virtualized flash storage layer is responsible for allocating a physical page of the flash device and logging the result internally. Similarly, when a page or range of pages backing a DFS object are released, the virtualized flash storage layer logs the update and releases the underlying physical storage. This approach guarantees the consistency of block allocations in DFS. It does not, however, guarantee the consistency of application data stored in a DFS object nor does it guarantee the consistency of directory updates. Directory updates, for instance, may require changes to more than one flash page. The most common example of an operation requiring multiple updates is `rename`. When a file is renamed, both the old and new parent directories, which need not be the same, must be updated. We discuss the case of directory updates separately in Section 4.3.

After a crash, the virtualized flash storage layer must recover using the internal log. The consistency of the contents of individual files is the responsibility of applications, but the on-flash state of the file system is guaranteed to be consistent. Since the

virtualized flash storage layer uses a log-structured approach to tracking allocations for performance reasons and must handle crashes in any case, DFS does not impose any additional onerous requirements.

4.2 DFS Layout and Objects

DFS uses a straightforward approach to organize files and their associated metadata in flash. The basic abstraction used by DFS to manage storage is the file. The system file stores the superblock and I-nodes, directories manage the mapping from file names to I-nodes, and ordinary files store user data. DFS files are represented in flash as contiguous logical extents and the flash device is addressed using a single 64-bit linear block address space. This logical address space is subdivided into allocation chunks which are further subdivided into regions. The size of both allocation chunks and regions are fixed at file system initialization time. Small files reside in regions; when a file grows too large to fit in a single region, it is assigned an entire allocation chunk. DFS does not support files of any type that are larger than the allocation chunk size.

As shown in Figure 4.4, there are three kinds of files in a DFS file system: the system file, directories, and plain files which contain user data. Plain files and directories may be either large files that occupy an entire allocation chunk or may be one of several small files packed together in a single allocation chunk.

The first allocation chunk in a DFS volume is occupied entirely by the system file. The system file contains the boot block, superblock and all of the I-nodes. The boot block occupies the first few blocks or sectors of the system file. The system and flash device firmware cooperate to load the boot block from the raw device and bootstrap the system.

The superblock immediately follows the boot block. At mount time, the file system can compute the location of the superblock without reading any other on-flash data

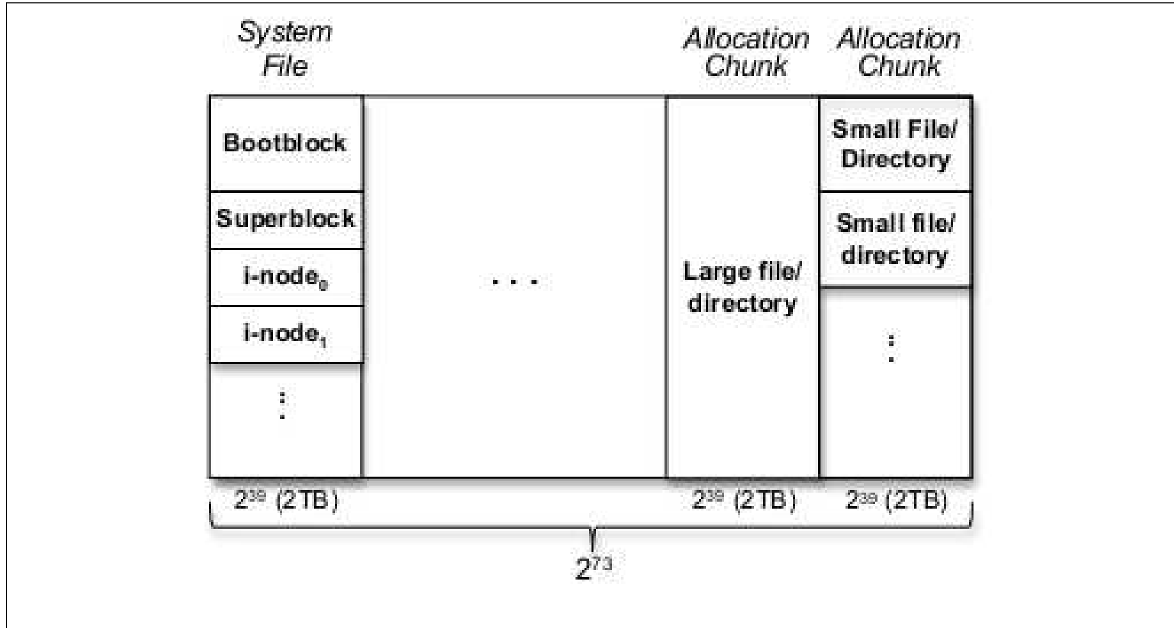

Figure 4.4: Layout of DFS system and user files in virtualized flash storage. The first 2TB is used for the system file. The remaining 2TB allocation chunks are for user files or directory files. A large file occupies the whole chunk; multiple small files are packed into a single chunk.

structures. The superblock contains a magic number, file system version number, a hint indicating the most efficient size for data transfers, and the persistent metadata for the I-node allocator. The remainder of the system file consists of an array of flash page aligned DFS I-node data structures.

The I-node allocator is organized as a LIFO stack. When the file system is initialized, the free I-nodes are linked together as a stack and the first free I-node is noted in the superblock. When an I-node is allocated, the first I-node in the stack is popped off and the superblock is updated with the location of the next free I-node, if any. When an I-node is freed, it is updated with a link to the current top of the stack and its I-node number is placed in the link to the top of the stack in the superblock. An I-node is not considered free until the superblock has been updated. In order to avoid leakage of I-nodes, DFS must ensure an atomic update of both the superblock and a directory entry. This is achieved using the atomic multi-block update primitive of the virtualized flash storage layer.

```
struct IODInode {
    u32int      inumber;
    u16int      mode;
    u16int      nlink;
    u32int      uid;
    u32int      gid;
    u32int      flags;
    u32int      gen;

    u64int      size;       /* size, including holes */
    u64int      atime;      /* 32-bit sec, 32-bit nsec */
    u64int      mtime;
    u64int      ctime;
    union {
        u64int      btime;  /* birth time; files & dirs only */
        u64int      rdev;   /* If device/FIFO/socket, etc. file */
    };
};
```

Figure 4.5: The DFS On-Flash I-Node Data Structure

Each I-node is identified by a 32-bit unique identifier or I-node number. The logical address of an I-node within the system file can be computed from the I-node number. Each I-node data structure is stored in a single 512-byte flash block. Figure 4.5 shows the contents and layout of an I-node in the system file. Each I-node contains the I-number and base virtual address of the corresponding file. It also contains a variety of generic metadata typically found in a UNIX file system, including nano-second resolution creation, modification, and access times. The metadata fields in the I-node consume a total of 72 bytes. An additional 440 bytes remain for other attributes and future use. Since an I-node is page-aligned and fits in a single flash page, it can be updated atomically by the virtualized flash storage layer without any special considerations.

The implementation of DFS uses a 32-bit block-addressed allocation chunk to store the content of a regular file. Since a file is stored in a contiguous, flat space, the address of each block offset can be simply computed by adding the offset to the

virtual base address of the space for the file. A block read simply returns the content of the physical flash page corresponding to the virtual block. A write operation writes the block to the mapped physical flash page directly, causing the device driver and hardware to allocate physical storage as necessary. Since the virtualized flash storage layer triggers a mapping or remapping on write, DFS performs the write operation without an explicit block allocation.

DFS permits "holes" in a file; that is, portions of a file that are not backed by allocated physical flash storage. A hole in a DFS file is a block-aligned interval of logical addresses that are not backed by physical storage. Such an interval is defined to consist of all zeroes when read. If a user application writes less than a full block and the write forces a physical block allocation, DFS and the virtualized flash storage layer guarantee that the unwritten portion of the newly allocated block is filled with zeroes.

A hole in a file may arise in one of two ways. First, a user application may write to disjoint regions of a file by using the `write` and `lseek` system calls. Second, on DFS file systems, an application may directly invoke the `trim` primitive of the virtualized flash storage layer on an interval of logical addresses in a file. The `trim` directive indicates to DFS and the virtualized flash storage layer that the physical flash storage backing an an interval of logical block addresses may be released and marked as garbage. The only physical storage used to represent a hole is that required for the logical to physical address mapping maintained by the virtualized flash storage layer. DFS also uses the `trim` directive to release the physical storage claimed by a file when it is deleted.

4.2.1 Direct Data Accesses

DFS promotes direct data access. The current Linux implementation of DFS allows the use of the buffer cache in order to support memory mapped I/O which is

required for the `exec` system call. However, for many workloads of interest, particularly databases, clients are expected to bypass the buffer cache altogether. The current implementation of DFS provides direct access via the direct I/O buffer cache bypass mechanism already present in the Linux kernel. Using direct I/O, page-aligned reads and writes are converted directly into I/O requests to the block device driver by the kernel.

There are two main rationales for this approach. First, traditional buffer cache design has several drawbacks. The traditional buffer cache typically uses a large amount of memory. Buffer cache design is quite complex since it needs to deal with multiple clients, implement sophisticated cache replacement policies to accommodate various access patterns of different workloads, and maintain consistency between the buffer cache and disk drives, and support crash recovery. In addition, having a buffer cache imposes a memory copy in the storage software stack.

Second, flash memory devices provide low-latency accesses, especially for random reads. Since the virtualized flash storage layer can solve the write latency problem, the main motivation for the buffer cache is largely eliminated. Thus, applications can benefit from the DFS direct data access approach by utilizing most of the main memory space typically used for the buffer cache for a larger in memory working set.

4.3 Directories

DFS supports the hierarchical naming scheme typical of modern file systems. Each file or directory in a DFS file system is identified by its I-node number. Applications and users, however, identify files and directories by a human-readable file name. Directories map file names to I-node numbers. A directory may contain files or other directories, resulting in a tree of directories rooted at the root directory with plain files at the leaves. A path name as used by an application is merely a sequence of

file names from the root directory of the file system to a file or directory in this tree. The root directory in a DFS file system always has the I-node number 1. Any other directories in a DFS file system are allocated I-node numbers on demand.

Directories implement the mapping from human readable file names to DFS I-node numbers which uniquely identify DFS files. Conceptually, a DFS directory is a mapping from name to an I-node number. DFS directories do not store file metadata. Rather, file metadata is located in the file's I-node in the system file. The reason for this design decision is that DFS supports multiple hard links to a file. That is, several different names, possibly in different directories, may refer to the same DFS I-node. By placing only the I-node number in the directory, the metadata for a file may be stored in exactly one place, avoiding consistency problems. In practice, DFS directory entries also contain a type field that indicates how the directory entry should be interpreted. This field is not strictly speaking necessary as the same information may be gleaned from an I-node, but it is conventionally found in UNIX file systems and can be used as an additional consistency check. Given the I-node number, the kernel and file system can retrieve the corresponding I-node and any metadata stored in it and take an appropriate action.

Figure 4.6 contains a list of the supported file types. There are three categories: regular files and directories, symbolic links, and special files. Special files include block and character device files, FIFOs (also known as named pipes), and UNIX domain sockets. The I-node for a special file contains sufficient metadata for the kernel to implement the special semantics of the file. DFS is only involved in providing naming for these types of files; the implementation details are the purview of the kernel.

DFS supports symbolic links. A symbolic link is a special type of file that, instead of containing data, contains another file name and acts as a layer of indirection in the file system naming scheme. When a symbolic link is opened, the client follows the link by instead opening the file named by the symbolic link. The kernel is responsible

```
/* Directory entry types */
enum {
    IO_DT_NONE      = 0,
    IO_DT_FIFO      = 001,  /* Named pipe */
    IO_DT_CHR       = 002,  /* Character device */
    IO_DT_DIR       = 004,  /* Directory file */
    IO_DT_BLK       = 006,  /* Block device */
    IO_DT_REG       = 010,  /* Regular file */
    IO_DT_LNK       = 012,  /* Symbolic link */
    IO_DT_SOCK      = 014,  /* UNIX domain socket */
    IO_DT_WHT       = 016   /* Whiteout */
};

/* UFS/FFS style directory entries */
struct IODirect {
    u32int  inode;              /* inode # */
    u16int  reclen;             /* length of this record */
    u8int   type;               /* file type */
    u8int   namelen;            /* length of name */
    uchar   name[0];            /* actually longer */
};
```

Figure 4.6: The DFS On-Flash Directory Entry Data Structure

for breaking cycles, usually by limiting the number of links followed. The target of a DFS symbolic link is an arbitrary path name stored as ordinary data. As an optimization, if the target path name is short enough, it may be stored in the extended metadata region of the link's I-node. Unlike a hard link, which is merely an additional reference to a particular I-node in the same file system, a symbolic link may cross volume boundaries.

A DFS directory is merely an ordinary file with special semantics and a particular internal structure. The virtualized flash storage layer is responsible for allocating physical flash storage for a directory just as it is for a file. A DFS directory supports four operations: look up a file, insert a new file, remove an existing file, rename a file, and enumerate the files in the directory. These operations must be efficient and crash-recoverable. The rename operation is the most complicated as a file may be

moved from one directory to another as a part of the renaming process. As a result, it may be necessary to update more than one location in the process of a directory operation, introducing the possibility of inconsistency after a crash.

There are a number of common approaches to making directory operations efficient. Some versions of FFS keep a hash table in memory for recently visited directories [54]. Other file systems, such as XFS, use B^+-trees on disk to represent directories [79]. The DFS design anticipates that file names will be hashed to a four or eight byte key and that the key-value pair mechanism of the virtualized flash storage layer will be used to store directory entries. The current version of the virtualized flash storage layer does not implement this interface, however, nor does it export the atomic multi-block update interface.

In the current implementation, a DFS directory is mapped to flash storage in the same manner as ordinary files are. The only difference is its internal structure. A directory contains contains an array of name, I-node number, type triples as described in Figure 4.6. The current implementation is very similar to that found in FFS [60]. Updates to directories, including operations such as rename, which touch multiple directories and the on-flash I-node allocator, are made crash-recoverable through the use of a separate write-ahead log. Although widely used and simple to implement, this approach does not scale well to large directories as it requires a linear scan of the directory for lookup, removal, and insertion.

4.4 Caching in DFS

Existing main memory buffer cache algorithms are designed either for DRAM or second-level caches backed by magnetic disks. The most commonly implemented algorithms in the former category include variations on LRU, CLOCK, and occasionally MRU [25,64,73]. ARC and its variants [5,61] have received wide attention recently in

both categories, most notably in Sun's ZFS file system. More recent innovations for second-level caches include 2Q and MQ [39, 90]. Although the strategies employed in these two cases are different due to differing workloads, they share the common goal of improving cache hit rates at all costs. As the gap between DRAM and magnetic storage access times has widened, the cost of a miss has increased significantly. In a modern storage system, a cache hit may be served in microseconds while a cache miss may take milliseconds to satisfy.

The rise of high-performance enterprise flash storage poses both a challenge and an opportunity. The marginal utility of an additional DRAM cache block is reduced significantly by the high random read performance of modern flash memory. Indeed, existing and announced flash storage devices offer between 100K and 300K random I/Os per second (IOPS) and further improvements are already foreseen; the use of multiple devices in RAID-like configurations can further improve aggregate I/O performance. As a result, the time to read a block from flash is within a small multiplicative factor of the time to copy a small block already in DRAM but not in the processors' caches.

One approach to the problem might be to add additional DRAM to the buffer cache. The cost of DRAM has fallen precipitously in the last decade and a large server-class machine today might have several tens of gigabytes of DRAM. There are several problems with this approach: DRAM itself is cheap, but the cost of a large high-performance memory subsystem is still significant; error detection and correction in main memory becomes imperative as main memory grows and the probability of corruption increases, but error detecting and correcting memory subsystems are slower and more expensive; the cost of flash on a per-gigabyte basis is actually lower than that of DRAM and so the available budget may well be better spent on a larger or higher-performance flash memory subsystem. The already low and still falling marginal utility of DRAM exacerbates this problem.

Another approach is to attempt to continue to improve the cache hit rate. This avenue is not a new one as cache algorithms have been studied for close to fifty years. Furthermore, it fails to take advantage of the inherent properties of flash memory storage: any improvement in cache hit rate applies equally to existing storage systems and next-generation flash storage systems. Instead, we advocate an approach whereby popular and infrequently used objects are segregated and only popular objects are cached. The basis for this approach is the realization that the cost for being wrong is much lower for flash storage subsystems than it is for traditional magnetic storage.

One can imagine a number of viable mechanisms for segregating popular and unpopular objects in a storage system. One could attempt to learn the popularity of an object with a machine learning algorithm or use application level hints. For many interesting workloads, however, an even simpler mechanism suffices: an object is deemed unpopular until it has been requested at least twice over a suitable interval of time. When an unpopular object is requested, it is read from flash and transferred directly to the user application, bypassing the buffer cache entirely. Only on subsequent requests is the buffer cache populated and then copied from the buffer cache into the application. Of course, this algorithm can be further improved if application hints are available. For instance, a database may suggest that I/Os to index blocks be cached unconditionally while I/Os to data blocks may be cached only on second use.

4.4.1 Lazy LRU

Lazy LRU is a simple extension to the ordinary Least Recently Used (LRU) algorithm. In a typical cache with LRU replacement there are two important data structures: a cache directory indexed by disk block address and a queue of unpinned cache blocks maintained in order of recency. In the steady state, the cache is fully occupied and all cache blocks contain valid data although many of them may be unpinned. When

a request is made for a cache block, the address is looked up in the cache directory. On a hit, if the block is in the queue, it is moved to the head of the queue. On a miss, the block must be read from backing storage and inserted into the cache. In order to insert the block, a victim must be chosen and ejected. Under the LRU policy, the victim is the least recently used, unpinned block. This block resides at the tail of the queue.

The Lazy LRU heuristic differs from ordinary LRU as follows. First, we maintain a data structure to track which blocks have been seen in the past. In order to avoid using significant additional storage in main memory, we use a Bloom Filter [8]. When a block is requested from the cache, we query the cache directory. On a cache hit, the algorithm is unchanged. On a cache miss, we consult the Bloom Filter; if the block has been seen before, the algorithm proceeds as in the case of ordinary LRU. If the block has not been encountered before, however, we mark it as seen and instead of populating the cache transfer the block directly from the underlying flash storage device to the user buffer, bypassing the cache. A subsequent request for the block will trigger a second cache miss, but this time the cache will be populated and the ordinary LRU algorithm will prevail.

Several points about the Lazy LRU algorithm are in order: first, there are clearly access patterns for which ordinary LRU is superior. If the cache is sufficiently large that the entire working set will fit and so the only misses are mandatory misses, then Lazy LRU will cause twice as many cache misses as ordinary LRU. We anticipate, however, that the gap in size between main memory and flash storage will continue to grow. Furthermore, for access patterns with significant skew, the penalty of additional cache misses may actually be outweighed by the cost of an additional copy from the buffer cache to a user application buffer. For instance, applications that exhibit large linear scans are unlikely to see a significant number of cache hits. Since Lazy LRU eliminates a large number of small block copies in this case it will outperform ordinary

LRU. This scan-resistant behavior is particularly useful in data-mining workloads such as TPC-H.

Second, the inter-reference gap for infrequently accessed blocks is significant. The inter-reference gap between two accesses of a block is the time measured either in wall time or in the number of intervening I/O requests. Since the cache is of finite size, infrequently accessed blocks will be ejected from the cache before a second request to the block occurs. In this case, the Lazy LRU algorithm improves performance by not ejecting an existing block or copying the new block unnecessarily. Indeed, Lazy LRU need not remember which blocks it has seen for all time. It may instead impose a window of several seconds to a few minutes or an absolute number of I/O requests after which blocks are forgotten. One particularly simple way to implement this approach is to periodically reset the Bloom Filter since deletions can not be simply accommodated. In this way, Lazy LRU can adapt to a slowly changing access pattern.

Third, an access pattern which we see frequently in some workloads is a read-modify-write cycle with short inter-reference gap. In this case, we see two accesses to a block in quick succession followed by a long period in which no accesses to the block are made. The first access is a read operation and the second a write or dirty operation. Lazy LRU can improve performance in this case as well. Rather than populating the cache, copying the data to a user buffer, modifying it, copying it back to the cache, and then flushing it at a later date, we can bypass the cache altogether. On a read, the user buffer is populated using DMA and modified locally. Using an asynchronous interface such as POSIX AIO, the dirtied buffer can be written back to the underlying storage as part of a larger request. If consistency across multiple threads or processes is an issue, a ghost entry can be maintained in the cache directory for dirtied blocks.

Fourth, yet another common access pattern in some workloads is a burst of read requests from a single thread for a single block with very short inter-reference gaps. In our simulation environment, this access pattern appears as a large number of distinct requests for a single block. In actual fact, however, these requests serve primarily to update cache metadata such as pin counts. For very short inter-reference gaps, it is plausible that the application instead should be modified to keep a local copy of the buffer rather than requesting the same buffer repeatedly provided the number of such local buffers required is small. Often, this access pattern is due to a number of small records being packed into a single larger storage block – *i.e.,* false sharing. If write sharing between multiple writers is involved, coordination overhead may be significant; if a single reader thread is involved, a local copy may be in order.

4.5 Summary

We have described the design and implementation of the Direct File System as well as the virtualized flash memory abstraction layer that it depends upon. DFS delegates storage block management, including allocation, layout, wear leveling, and bulk erasure, to the virtualized flash storage layer. DFS further depends upon low-level support for atomic updates to ensure reliable recovery in the event of a crash. Our current implementation of DFS relies upon the implementation of the virtualized flash storage layer as a very large virtually addressed single-level store. This approach yields good performance, a simple interface, and backwards compatibility with software systems accustomed to the conventional block-oriented storage interface. In order to use the available virtual address space efficiently, DFS does differentiate between large and small files. This partitioning is guided by distribution of file sizes reported by studies of deployed file systems.

We have also described some simple heuristics for improved flash-aware buffer cache algorithms. We believe that one important consequence of the emergence of large, high-performance, and relatively inexpensive solid state storage systems. As a result, the marginal utility of additional DRAM buffer cache blocks is much reduced. One avenue for further work is to continue the program of improving buffer cache hit rates. An additional approach, which we also advocate, is a form of selective caching wherein popular blocks or objects are preferentially cached while less popular blocks are not inserted into the cache in the first place. The Lazy LRU heuristic is one example of this latter approach.

The current implementation of DFS does not fully realize our design goals. For instance, the DFS implementation does not yet support snapshots. The primary reason we have not implemented snapshots is that we expect future versions of the virtualized flash storage layer to provide primitives for snapshots. Native support of snapshots in the virtualized flash storage layer will greatly simplify the snapshot implementation in DFS. Since the virtualized flash storage layer is already log-structured for performance and hence takes a copy-on-write approach by default, one can implement snapshots in the virtualized flash storage layer efficiently.

Second, we are currently implementing support for atomic multi-block updates in the virtualized flash storage layer. The log-structured, copy-on-write nature of the flash storage layer makes it possible to export such an interface efficiently. For example, Prabhakaran *et al.* recently described an efficient commit protocol to implement atomic multi-block writes [70]. These methods will allow DFS to guarantee the consistency of directory contents and I-node allocations without an auxiliary log. In the interim, DFS uses a straightforward extension of the traditional UNIX file system directory structure.

Chapter 5

Evaluation

In this chapter we evaluate the current implementation of DFS as described in Chapter 4 running on top of FusionIO's ioDrive hardware and device driver, which together implement a preliminary version of the proposed virtualized flash storage layer of Chapter 3. We evaluate the performance and assess the implementation complexity of DFS using a combination of microbenchmarks and an application benchmark representative of modern file system workloads.

The existing DFS implementation is a loadable Linux kernel module that implements the VFS API and relies on the existing Linux buffer cache implementation. The current version of DFS does not include the modifications to the buffer cache proposed in Section 4.4. Rather, we evaluate the proposed approach to caching for flash storage separately. Our analysis relies on buffer cache traces taken from well-known database workloads. We have instrumented both Oracle [53] and Postgres [78]. The opportunity to instrument the Oracle RDBMS and collect cache trace data is itself interesting as source level access and trace data are not generally available to the research community. We built a simple buffer cache simulator based on the Oracle RDBMS architecture and use it for the trace driven simulations presented here.

5.1 Evaluation of DFS

In the analysis of DFS and its performance that follows, we are first and foremost interested in answering two questions:

- How do the layers of abstraction perform?
- How does DFS compare with existing file systems?

To answer the first question, we use a microbenchmark to evaluate the number of I/O operations per second (IOPS) and bandwidth delivered by the virtualized flash storage layer and by the DFS layer. To answer the second question, we compare DFS with ext3 by using a microbenchmark and an application suite. Ideally, we would compare with existing flash file systems as well, however file systems such as YAFFS and JFFS2 are designed to use raw NAND flash and are not compatible with either existing SSDs that export a block interface or the sort of next-generation flash storage devices that we envisage that support the virtualized flash storage layer API.

All of our experiments were conducted on a desktop with Intel Quad Core processor running at 2.4GHz with a 4MB L2 cache and 4GB of main memory. The host operating system was a stock Fedora Core installation running the Linux 2.6.27.9 kernel. Both DFS and the virtualized flash storage layer implemented by the FusionIO device driver were compiled as loadable kernel modules using the default optimization flags set by the Linux kernel build environment.

We used a FusionIO ioDrive with 160GB of SLC NAND flash connected via PCI-Express x4 [23]. The advertised read latency of the FusionIO device is $50\mu s$. For a single reader, this translates to a theoretical maximum throughput of 20,000 IOPS. Multiple readers can take advantage of the hardware parallelism in the device to achieve much higher aggregate throughput. For the sake of comparison, we also ran the microbenchmarks on a 32GB Intel X25-E SSD connected to a SATA II host bus adapter [36]. This device has an advertised typical read latency of about $75\mu s$.

Our results show that the virtualized flash storage layer delivers performance close to the limits of the hardware, both in terms of I/Os per second and aggregate bandwidth. Our results also show that DFS is much simpler than ext3 and achieves better performance in both the micro- and application benchmarks than ext3. Furthermore, DFS is able to achieve these results more using fewer resources.

5.1.1 Virtualized Flash Storage Performance

We have two goals in evaluating the performance of the virtualized flash storage layer. First, to examine the potential benefits of the proposed abstraction layer in combination with hardware support that exposes parallelism. Second, to determine the raw performance in terms of bandwidth and IOPs delivered in order to compare DFS and ext3. For both purposes, we designed a simple microbenchmark which opens the raw block device in direct I/O mode, bypassing the kernel buffer cache. Each thread in the program attempts to execute block-aligned reads and writes as quickly as possible.

To evaluate the benefits of the virtualized flash storage layer and its hardware, one would need to compare a traditional block storage software layer with flash memory hardware equivalent to the FusionIO ioDrive but with a traditional disk interface FTL. Since such hardware does not exist, we have used a Linux block storage layer with an Intel X25-E SSD, which is a well-regarded SSD in the marketplace. Although this is not a fair comparison, the results give us some sense of the performance impact of the abstractions designed for flash memory.

We measured the number of sustained random I/O transactions per second. While both flash devices are enterprise class devices, the test platform is the typical white box workstation we described earlier. The results are shown in Table 5.1. Performance, while impressive compared to magnetic disks, is less than that advertised by the manufacturers. We suspect that the large IOPS performance gaps, particularly

for write IOPS, are partially limited by the disk drive interface and limited resources in a drive controller to run sophisticated remapping algorithms.

Device	Read IOPS	Write IOPS
Intel	33,400	3,120
FusionIO	98,800	75,100

Table 5.1: Device 4KB Peak Random IOPS

Device	Threads	Read (MB/s)	Write (MB/s)
Intel	2	221	162
FusionIO	2	769	686

Table 5.2: Device Peak Bandwidth with 1MB Transfers

Table 5.2 shows the peak bandwidth for both cases. We measured sequential I/O bandwidth by computing the aggregate throughput of multiple readers and writers. Each client transferred 1MB blocks for the throughput test and used direct I/O to bypass the kernel buffer cache. The results in the table are the bandwidth results using two writers. The virtualized flash storage layer with ioDrive achieves 769MB/s for read and 686MB/s for write, whereas the traditional block storage layer with the Intel SSD achieves 221MB/s for read and 162MB/s for write.

5.1.2 Complexity of DFS vs. ext3

Table 5.3 shows the number of lines of code for the major modules of DFS and ext3 file systems. Although both implement Unix file systems, DFS is much simpler. The simplicity of DFS is mainly due to delegating block allocations and reclamations to the virtualized flash storage layer. The ext3 file system, for example, has a total of 17,500 lines of code and relies on an additional 7,000 lines of code to implement logging (JBD) for a total of nearly 25,000 lines of code compared to roughly 3,300 lines of code in DFS. Of the total lines in ext3, about 8,000 lines (33%) are related to

Module	DFS	Ext3
Headers	392	1583
Kernel Interface (Superblock, *etc.*)	1625	2973
Logging	0	7128
Block Allocator	0	1909
I-nodes	250	6544
Files	286	283
Directories	561	670
ACLs, Extended Attrs.	N/A	2420
Resizing	N/A	1085
Miscellaneous	175	113
Total	3289	24708

Table 5.3: Lines of Code in DFS and Ext3 by Module

block allocations, deallocations and I-node layout. Of the remainder, another 3,500 lines (15%) implement support for on-line resizing and extended attributes, neither of which are supported by DFS.

Although it may not be fair to compare a research prototype file system with a file system that has evolved for several years, the percentages of block allocation and logging in the file systems give us some indication of the relative complexity of different components in a file system.

5.1.3 Microbenchmark Performance of DFS vs. ext3

We use Iozone [65] to evaluate the performance of DFS and ext3 on the ioDrive when using both direct and buffered access. We record the number of 4KB I/O transactions per second achieved with each file system and also compute the CPU usage required in each case as the ratio between user plus system time to elapsed wall time. For both file systems, we ran Iozone in three different modes: in the default mode in which I/O requests pass through the kernel buffer cache, in direct I/O mode without the buffer cache, and in memory-mapped mode using the `mmap` system call.

File System	Threads	Read (MB/s)	Write (MB/s)
ext3	2	760	626
DFS	2	769	667

Table 5.4: Peak Bandwidth 1MB Transfers on ioDrive (Average of Five Runs)

In our experiments, both file systems run on top of the virtualized flash storage layer. The ext3 file system in this case uses the backward compatible block storage interface supported by the virtualized flash storage layer.

Direct Access

For both reads and writes, we consider sequential and uniform random access to previously allocated blocks. Our goal is to understand the additional overhead due to DFS compared to the virtualized flash storage layer. The results indicate that DFS is indeed lightweight and imposes much less overhead than ext3. Compared to the raw device, DFS delivers about 5% fewer IOPS for both read and write whereas ext3 delivers 9% fewer read IOPS and more than 20% fewer write IOPS. In terms of bandwidth, DFS delivers about 3% less write bandwidth whereas ext3 delivers 9% less write bandwidth.

Table 5.4 shows the peak bandwidth for sequential 1MB block transfers. This microbenchmark is the file system analog of the raw device bandwidth performance shown in Table 5.2. Although the performance difference between DFS and ext3 for large block transfers is relatively modest, DFS does narrow the gap between file system and raw device performance for both sequential reads and writes.

Figure 5.1 shows the average direct random I/O performance on DFS and ext3 as a function of the number of concurrent clients on the FusionIO ioDrive. Both of the file systems also exhibit a characteristic that may at first seem surprising: *aggregate* performance often increases with an increasing number of clients, even if the client requests are independent and distributed uniformly at random. This behavior is

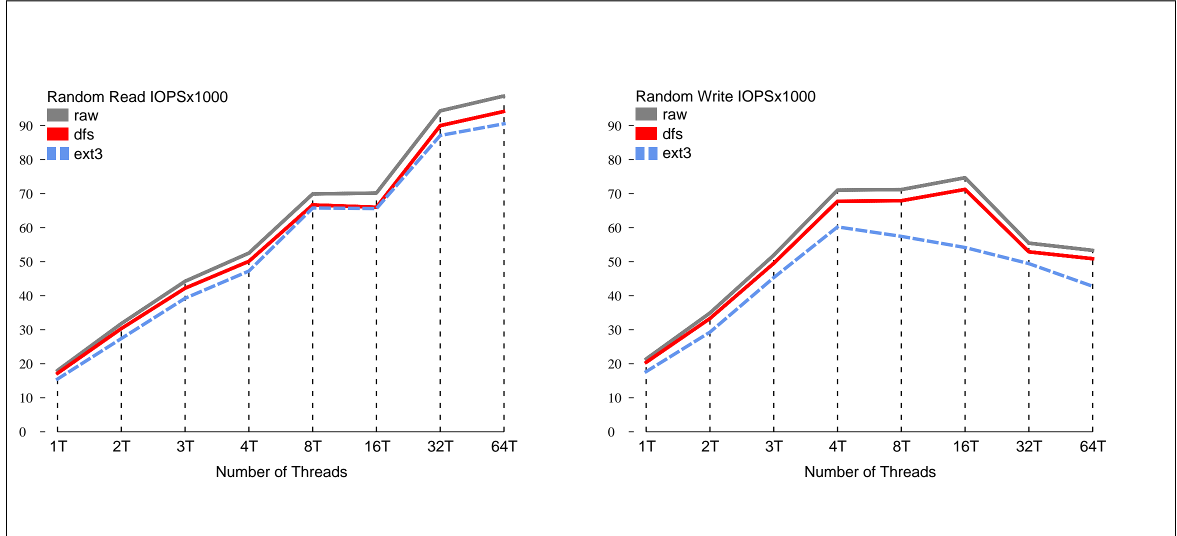

Figure 5.1: Aggregate IOPS for 4K Random Direct I/O as a Function of the Number of Threads

due to the relatively long latency of individual I/O transactions and deep hardware and software request queues in the flash storage subsystem. This behavior is quite different from what most applications expect and may require changes to them in order to realize the full potential of the storage system.

Unlike read throughput, write throughput peaks at about 16 concurrent writers and then decreases slightly. Both the aggregate throughput and the number of concurrent writers at peak performance are lower than when accessing the raw storage device. The additional overhead imposed by the file system on the write path reduces both the total aggregate performance and the number of concurrent writers that can be handled efficiently.

We have also measured CPU utilization per 1,000 IOPS delivered in the microbenchmarks. Table 5.5 shows the improvement of DFS over ext3. We report the average of five runs of the IOZone based microbenchmark with a standard deviation of one to three percent. For reads, DFS CPU utilization is comparable to ext3; for writes, particularly with small numbers of threads, DFS is more efficient. Overall, DFS consumes somewhat less CPU power, further confirming that DFS is a lighter weight file system than ext3.

Threads	Read	Random Read	Write	Random Write
1	8.1	2.8	9.4	13.8
2	1.3	1.6	12.8	11.5
3	0.4	5.8	10.4	15.3
4	-1.3	-6.8	-15.5	-17.1
8	0.3	-1.0	-3.9	-1.2
16	1.0	1.7	2.0	6.7
32	4.1	8.5	4.8	4.4

Table 5.5: Improvement in CPU Utilization per $1,000$ IOPS using 4K Direct I/O with DFS relative to Ext3

One anomaly worthy of note is that DFS is actually more expensive than ext3 per I/O when running with four clients, particularly if the clients are writers. This is due to the fact that there are four cores on the test machine and the device driver itself has worker threads that require CPU and memory bandwidth. The higher performance of DFS translates into more work for the device driver and particularly for the garbage collector. Since there are more threads than cores, cache hit rates suffer and scheduling costs increase; under higher offered load, the effect is more pronounced, although it can be mitigated somewhat by binding the garbage collector to a single processor core.

Buffered Access

To evaluate the performance of DFS in the presence of the kernel buffer cache, we ran a similar set of experiments as in the case of direct I/O. Each experiment touched 8GB worth of data using 4K block transfers. The buffer cache was invalidated after each run by unmounting the file system and the total data referenced exceeded the physical memory available by a factor of two. The first run of each experiment was discarded and the average of the subsequent ten runs reported.

Tables 5.6 and 5.7 show the results via the Linux buffer cache and via memory-mapped I/O data path which also uses the buffer cache. There are several observa-

Thr.	Seq. Read IOPS x 1K		Rand. Read IOPS x 1K	
	ext3	DFS (Speedup)	ext3	DFS (Speedup)
1	125.5	191.2 (1.52)	17.5	19.0 (1.09)
2	147.6	194.1 (1.32)	32.9	34.0 (1.03)
3	137.1	192.7 (1.41)	44.3	46.6 (1.05)
4	133.6	193.9 (1.45)	55.2	57.8 (1.05)
8	134.4	193.5 (1.44)	78.7	80.5 (1.02)
16	132.6	193.9 (1.46)	79.6	81.1 (1.02)
32	132.3	194.8 (1.47)	95.4	101.2 (1.06)
	Seq. Write IOPS x 1K		Rand. Write IOPS x 1K	
Thr.	ext3	DFS (Speedup)	ext3	DFS (Speedup)
1	67.8	154.9 (2.28)	61.2	68.5 (1.12)
2	71.6	165.6 (2.31)	56.7	64.6 (1.14)
3	73.0	156.9 (2.15)	59.6	62.8 (1.05)
4	65.5	161.5 (2.47)	57.5	63.3 (1.10)
8	64.9	148.1 (2.28)	57.0	58.7 (1.03)
16	65.3	147.8 (2.26)	52.6	56.5 (1.07)
32	65.3	150.1 (2.30)	55.2	50.6 (0.92)

Table 5.6: Buffer Cache Performance with 4KB I/Os

tions. First, both DFS and ext3 have similar random read IOPS and random write IOPS to their performance results using direct I/O. Although this is expected, DFS is better than ext3 on average by about 5%. This further shows that DFS has less overhead than ext3 in the presence of a buffer cache.

Second, we observe that the traditional buffer cache is not effective when there are a lot of parallel accesses. In the sequential read case, the number of IOPS delivered by DFS basically doubles its direct I/O access performance, whereas the IOPS of ext3 is only modestly better than its random access performance when there are enough parallel accesses. For example, when there are 32 threads, its IOPS is 132,000, which is only 28% better than its random read IOPS of 95,400!

Third, DFS is substantially better than ext3 for both sequential read and sequential write cases. For sequential reads, it outperforms ext3 by more than a factor of

Thr.	Seq. Read IOPS x 1K ext3	DFS (Speedup)	Rand. Read IOPS x 1K ext3	DFS (Speedup)
1	42.6	52.2 (1.23)	13.9	18.1 (1.3)
2	72.6	84.6 (1.17)	22.2	28.2 (1.27)
3	94.7	114.9 (1.21)	27.4	32.1 (1.17)
4	110.2	117.1 (1.06)	29.7	35.0 (1.18)
Thr.	**Seq. Write IOPS x 1K ext3**	**DFS (Speedup)**	**Rand. Write IOPS x 1K ext3**	**DFS (Speedup)**
1	28.8	40.2 (1.4)	11.8	13.5 (1.14)
2	39.9	55.5 (1.4)	16.7	18.1 (1.08)
3	41.9	68.4 (1.6)	19.1	20.0 (1.05)
4	44.3	70.8 (1.6)	20.1	22.0 (1.09)

Table 5.7: Memory Mapped Performance of Ext3 & DFS

1.4. For sequential writes, it outperforms ext3 by more than a factor of 2.15. This is largely due to the fact that DFS is simple and can easily combines I/Os.

The story for memory-mapped I/O performance is much the same as it is for buffered I/O. Random access performance is relatively poor compared to direct I/O performance. The simplicity of DFS and the short code paths in the file system allow it to outperform ext3 in this case. The comparatively large speedups for sequential I/O, particularly sequential writes, are again due to the fact that DFS readily combines multiple small I/Os into larger ones. In the next section we show that I/O combining is an important effect; the quicksort benchmark is a good example of this phenomenon with memory mapped I/O. We count both the number of I/O transactions during the course of execution and the total number of bytes transferred. DFS greatly reduces the number of write operations and more modestly the number of read operations.

5.1.4 Application Benchmarks Performance of DFS vs. ext3

We have used five applications as an application benchmark suite to evaluate the application-level performance on DFS and ext3.

Applications	Description	I/O Patterns
Quicksort	A quicksort on a large dataset	Mem-mapped I/O
N-Gram	A program for querying n-gram data	Direct, random read
KNNImpute	Processes bioinformatics microarray data	Mem-mapped I/O
VM-Update	Update of an OS on several virtual machines	Sequential read & write
TPC-H	Standard benchmark for Decision Support	Mostly sequential read

Table 5.8: Applications and their characteristics.

Application Benchmarks

Table 5.8 summarizes the characteristics of the applications and the reasons why they are chosen for our performance evaluation.

In the following, we describe each application, its implementation and workloads in detail:

Quicksort. This quicksort is implemented as a single-threaded program to sort 715 million 24 byte key-value pairs memory mapped from a single 16GB file. Although quicksort exhibits good locality of reference, this benchmark program nonetheless stresses the memory mapped I/O subsystem. The memory-mapped interface has the advantages of being simple, easy to understand, and a straightforward way to transform a large flash storage device into an inexpensive replacement for DRAM as it provides the illusion of word-addressable access.

N-Gram. This program indexes all of the 5-grams in the Google n-gram corpus by building a single large hash table that contains 26GB worth of key-value pairs. The Google n-gram corpus is a large set of n-grams and their appearance counts taken from a crawl of the Web that has proved valuable for a variety of computational linguistics tasks. There are just over 13.5 million words or 1-grams and just over 1.1 billion 5 grams. Indexing the data set with an SQL database takes a week on a computer with only 4GB of DRAM [11]. Our indexing program uses 4KB buckets with the first 64 bytes reserved for metadata. The implementation does not support overflows, rather an occupancy histogram is constructed to find the smallest k such

that 2^k hash buckets will hold the dataset without overflows. With a variant of the standard Fowler-Nolls-Vo hash, the entire data set fits in 16M buckets and the histogram in 64MB of memory. Our evaluation program uses synthetically generated query traces of 200K queries each; results are based upon the average of twenty runs. Queries are drawn either uniformly at random or according to a Zipf distribution with $\alpha = 1.0001$. The results were qualitatively similar for other values of α until locking overhead dominated I/O overhead.

KNNImpute. This program is a very popular bioinformatics code for estimating missing values in data obtained from microarray experiments. The program uses the KNNImpute [84] algorithm for DNA microarrays which takes as input a matrix with G rows representing genes and E columns representing experiments. Then a symmetric GxG distance matrix with the Euclidean distance between all gene pairs is calculated based on all experiment values for both genes. Finally, the distance matrix is written to disk as its output. The program is a multi-threaded implementation using memory-mapped I/O. Our input data is a matrix with 41,768 genes and 200 experiments results in a matrix of about 32MB, and a distance matrix of 6.6GB. There are 2079 genes with missing values.

VM Update. This benchmark is a simple update of multiple virtual machines hosted on a single server. We choose this application because virtual machines have become popular from both a cost and management perspective. Since each virtual machine typically runs the same operating system but has its own copy, operating system updates can pose a significant performance problem. Each virtual machine needs to apply critical and periodic system software updates at the same time. This process is both CPU and I/O intensive. To simulate such an environment, we installed 4 copies of Ubuntu 8.04 in four different VirtualBox instances. In each image, we downloaded all of the available updates and then measured the amount of time it

Application	Wall Time Ext3	DFS	Speedup
Quick Sort	1268	822	1.54
N-Gram (Zipf)	4718	1912	2.47
KNNImpute	303	248	1.22
VM Update	685	640	1.07
TPC-H	5059	4154	1.22

Table 5.9: Application Benchmark Execution Time Improvement: Best of DFS *vs* Best of Ext3

	Read IOPS x 1000		Read Bytes x 1M	
Application	Ext3	DFS (Change)	Ext3	DFS (Change)
Quick Sort	1989	1558 (0.78)	114614	103991 (0.91)
N-Gram (Zipf)	156	157 (1.01)	641	646 (1.01)
KNNImpute	2387	1916 (0.80)	42806	36146 (0.84)
VM Update	244	193 (0.79)	9930	9760 (0.98)
TPC-H	6375	3760 (0.59)	541060	484985 (0.90)
	Write IOPS x 1000		Write Bytes x 1M	
Application	Ext3	DFS (Change)	Ext3	DFS (Change)
Quick Sort	49576	1914 (0.04)	203063	192557 (0.95)
N-Gram (Zipf)	N/A	N/A	N/A	N/A
KNNImpute	2686	179 (0.07)	11002	12696 (1.15)
VM Update	3712	1144 (0.31)	15205	19767 (1.30)
TPC-H	52310	3626 (0.07)	214265	212223 (0.99)

Table 5.10: Application Benchmark Improvement in IOPS Required and Bytes Transferred: Best of DFS *vs* Best of Ext3

took to install these updates. There were a total of 265 packages updated containing 343MB of compressed data and about 38,000 distinct files.

TPC-H. This standard benchmark, formally known as the Transaction Processing Council's Benchmark H (TPC-H) [82], is intended to be typical of decision support workloads. The TPC-H benchmark models a large-scale enterprise database application for analyzing business transactions for patterns and trends. The application uses a very large relational database store in persistent storage to manage the data. This

database consists of eight tables which model the various distributors and suppliers in different geographical regions.

We used the Ingres database to run the benchmark, which consists of 22 business oriented queries and two functions that respectively insert and delete rows in the database. We used the default configuration for the database with two storage devices: the database itself, temporary files, and backup transaction log were placed on the flash device and the executables and log files were stored on the local disk. We report the results of running TPC-H with a scale factor of 5, which corresponds to about 5GB of raw input data and 90GB for the data, indexes, and logs stored on flash once loaded into the database.

Performance Results of DFS vs. ext3

This section first reports the performance results of DFS and ext3 for each application, and then analyzes the results in detail.

The main performance result is that DFS improves applications substantially over ext3. Table 5.9 shows the elapsed wall clock time of each application running with ext3 and DFS on the same execution environment mentioned at the beginning of the section. The results show that DFS improves the performance all applications and the speedups range from a factor of 1.07 to 2.47.

To explain the performance results, we will first use Table 5.10 to show the number of read and write IOPS, and the number of bytes transferred for reads and writes for each application. The main observation is that DFS issues a smaller number of larger I/O transactions than ext3, though the behaviors of reads and writes are quite different. This observation explains partially why DFS improves the performance of all applications, since we know from the microbenchmark performance that DFS achieves better IOPS than ext3 and significantly better throughput when the I/O transaction sizes are large.

One reason for larger I/O transactions is that in the Linux kernel, file offsets are mapped to block numbers via a per-file-system `get_block` function. The DFS implementation of `get_block` is aggressive about making large transfers when possible. A more nuanced policy might improve performance further, particularly in the case of applications such as KNNImpute and the VM Update workload which actually see an increase in the total number of bytes transferred. In most cases, however, the result of the current implementation is a modest reduction in the number of bytes transferred.

But, the smaller number of larger I/O transactions does not completely explain the performance results. In the following, we will describe our understanding of the performance of each application individually.

Quicksort. The Quicksort benchmark program sees a speedup of 1.54 when using DFS instead of ext3 on the ioDrive. Unlike the other benchmark applications, the quicksort program sees a large increase in CPU utilization when using DFS instead of ext3. CPU utilization includes both the CPU used by the FusionIO device driver and by the application itself. When running on ext3, this benchmark program is I/O bound; the higher throughput provided by DFS leads to higher CPU utilization, which is actually a desirable outcome in this particular case. In addition, we collected statistics from the virtualized flash storage layer to count the number of read and write transactions issued in each of the three cases. When running on ext3, the number of read transactions is similar to that found with DFS, whereas the number of write transactions is roughly twenty-five times larger than that of DFS, which contributed to the speedup. The average transaction size with ext3 is about 4KB instead of 64KB with DFS.

Google N-Gram Corpus. The N-gram query benchmark program running on DFS achieves a speedup of 2.5 over that on ext3. Table 5.11 illustrates the speedup

Threads	Wall Time in Sec. Ext3	DFS	Ctx Switch x 1K Ext3	DFS
1	10.82	10.48	156.66	156.65
4	4.25	3.40	308.08	160.60
8	4.58	2.46	291.91	167.36
16	4.65	2.45	295.02	168.57
32	4.72	1.91	299.73	172.34

Table 5.11: Zipf-Distributed N-Gram Queries: Elapsed Time and Context Switches ($\alpha = 1.0001$)

as a function of the number of concurrent threads; in all cases, the internal cache is 1,024 hash buckets and all I/O bypasses the kernel's buffer cache.

The hash table implementation is able to achieve about 95% of the random I/O performance delivered in the `Iozone` microbenchmarks given sufficient concurrency. As expected, performance is higher when the queries are Zipf-distributed as the internal cache captures many of the most popular queries. For Zipf parameter $\alpha = 1.0001$, there are about 156,000 4K random reads to satisfy 200,000 queries. Moreover, query performance for hash tables backed by DFS scales with the number of concurrent threads much as it did in the `Iozone` random read benchmark. The performance of hash tables backed by ext3 do not scale with the number of threads nearly so well. This is due to increased per-file lock contention in ext3. We measured the number of voluntary context switches when running on each file system as reported by `getrusage`. A voluntary context switch indicates that the application was unable to acquire a resource in the kernel such as a lock. When running on ext3, the number of voluntary context switches increased dramatically with the number of concurrent threads; it did not do so on DFS. Although it may be possible to overcome the resource contention in ext3, the simplicity of DFS allows us to sidestep the issue altogether. This effect was less pronounced in the microbenchmarks because `Iozone` never assigns more than one thread to each file by default.

Bioinformatics Missing Value Estimation. KNNImpute takes about 18% less time to run when using DFS as opposed to ext3 with a standard deviation of about 1% of the mean run time. About 36% of the total execution time when running on ext3 is devoted to writing the distance matrix to stable storage. Most of the improvement in run time when running on DFS is during this phase of execution. CPU utilization increases by almost 7% on average when using DFS instead of ext3. This is due to increased system CPU usage during the distance matrix write phase by the FusionIO device driver's worker threads, particularly the garbage collector.

Virtual Machine Update. On average, it took 648 seconds to upgrade virtual machines hosted on DFS and 701 seconds to upgrade those hosted on ext3 file systems, for a net speed up of 7.6%. In both cases, the four virtual machines used nearly all of the available CPU for the duration of the benchmark. We found that each VirtualBox instance kept a single processor busy almost 25% percent of the time even when the guest operating system was idle. As a result, the virtual machine update workload quickly became CPU bound. If the virtual machine implementation itself were more efficient or more virtual machines shared the same storage system we would expect to see a larger benefit to using DFS.

TPC-H. We ran the TPC-H benchmark with a scale factor of five on both DFS and ext3. The average speedup over five runs was 1.22. For the individual queries DFS always performs better than ext3, with the speedup ranging from 1.04 (Q1: pricing summary report) to 1.51 (RF2: old sales refresh function). However, the largest contribution to the overall speedup is the 1.20 speedup achieved for Q5 (local supplier volume), which consumes roughly 75% of the total execution time.

There is a large reduction (14.4x) in the number of write transactions when using DFS as compared to ext3 and a smaller reduction (1.7x) in the number of read transactions. As in the case of several of the other benchmark applications, the large

reduction in the number of I/O transactions is largely offset by larger transfers in each transaction, resulting in a modest decrease in the total number of bytes transferred.

CPU utilization is lower when running on DFS as opposed to ext3, but the Ingres database thread runs with close to 100% CPU utilization in both cases. The reduction in CPU usage is due instead to greater efficiency in the kernel storage software stack, particularly the flash device driver's worker threads.

5.2 Caching

Production quality buffer cache implementations are complex pieces of software that are typically tightly coupled either to an operating system kernel or a large application such as a database. Major modifications to a buffer cache can be a time consuming undertaking. As a case in point, the Oracle database buffer cache implementation is several hundred thousand lines of code that is tightly coupled to much of the rest of the I/O subsystem even though it is by design logically distinct. Rather than make significant changes to existing buffer cache implementation, we have chosen to take a trace-driven approach.

Our trace driven analysis of the proposed cache algorithms described in Section 4.4 depends upon a set of relatively modest changes to the buffer cache that log each read, dirty, flush, pin and unpin operation. A simplified cache simulator subsequently reads the cache operation trace and simulates the actions of the cache algorithm for a given set of parameters. The cache simulation is offline which further simplifies the implementation of the simulator and allows detailed instrumentation if so desired. For each workload, the captured cache trace allows us to simulate a range of cache algorithms and sizes. Each run of the simulator produces an I/O trace. This I/O trace drives an on-line replay engine that issues I/Os in a realistic test environment that allows us to measure real-world performance of the storage system.

5.2.1 Simulation Environment

Our simulation environment consists of three components: the instrumented buffer cache, the cache simulator, and the replay engine. The instrumentation added to the buffer cache emits a trace of cache operations arriving at the cache, not I/O operations emitted by the cache. This is an important distinction as most available traces in the storage literature are I/O traces. Furthermore, it is a relatively simple matter to capture I/O traces but in general it requires source-level access and the ability to recompile the application of interest to capture cache traces. Open source software such as the Linux kernel and databases such as Postgres are comparatively easy to modify so as to capture cache traces. Current open source databases, however, do not support query-level parallelization and have not been tuned as aggressively as the major proprietary database management systems such as Oracle.

Our cache simulator consumes previously captured cache traces and outputs an I/O trace. The cache simulator is run offline. It is a single-threaded, event driven program that simulates a multi-threaded cache implementation. Cache events read by the simulator are ordered by a global event sequence number or wall time. The simulator's scheduler considers each simulated thread in round-robin order and selects the event with the lowest sequence number that is currently runnable. The simulator tracks per-block lock state to decide which events are runnable and does not permit writers to bypass readers. The simulator generally speaking does not emit single write I/O requests. Rather, several dirty cache blocks are bundled into a single write episode that is scheduled as a batch. The simulator supports several cache replacement algorithms each compiled as a separate linkable module. The replacement algorithm is responsible for selecting victims when a cache block must be ejected from the cache and for scheduling "write episodes".

In our simulator, reads are assumed to be issued synchronously by individual threads while writes are handled asynchronously by one or more background cleaner

threads. This is the same approach taken by the Oracle RDBMS and is typical of many high-performance storage intensive applications. The simulator emits enough information for the replay engine to approximate the data flow dependencies for each block in the system. Each read request emitted by the simulator is tagged with the disk address of the requested block, a read sequence number, the write sequence number of the preceding write to this block (if any), the reader's thread ID and the frame number of an in-core buffer. Each write episode is tagged with a write sequence number, the last read sequence number for each thread that dirtied at least one buffer in the episode and the disk address and buffer frame number for each write in the episode.

The replay engine is responsible for taking the I/O trace output by the cache simulator and transforming it into a sequence of real I/O requests while satisfying some basic ordering constraints. The replay engine is thus able to produce an I/O workload very similar to what the original software system would have produced given the same input workload, cache algorithm, and cache parameters. The detailed instrumentation of the cache simulator and the runtime behavior of the replay engine provide the basis for our analysis of cache algorithms and behaviors.

For each thread in the I/O trace, the replay engine maintains a thread that issues reads synchronously. Writes are issued asynchronously using one of a variety of techniques, including a pool of writer threads and a small number of threads using the POSIX AIO interface. No matter what the configuration of the replay engine, it enforces the following two I/O ordering rules:

1. Reads to a block b must wait until all preceding writes to b have completed (no stale data)

2. Writes of a dirty buffer block until all threads that have touched the buffer have completed all preceding reads (no dirty data)

The replay engine must be able to enforce these basic ordering constraints while still being efficient enough to remain I/O bound. The current implementation of the replay engine is I/O bound on our current hardware platform.

5.2.2 Workloads

In this section, we provide a short description of each of the workloads used to evaluate the Lazy LRU heuristic. In each case, we use an application benchmark to capture a trace of cache operations. This trace serves as input to our simulation environment which outputs I/O traces for subsequent replay.

Our first workload is the Transaction Processing Council's Benchmark C (TPC-C) [83], a standard benchmark for online transaction processing (OLTP). TPC-C simulates a complete environment in which a set of users executes transactions for order entry and fulfillment. These transactions include entering and delivering orders, recording payments, checking the status of orders, and monitoring stock. As is typical of OLTP workloads, the TPC-C benchmark consists of a mix of random reads and writes with a block reference pattern that exhibits significant skew.

We collected our TPC-C trace using an Oracle database environment set up by Oracle engineers for the express purpose of running the benchmark [53]. The tracing environment consisted of a modified version of Oracle that emits a trace of cache operations and a 1000 warehouse TPC-C installation. The resulting trace consists of almost one hundred and fifty million cache operations.

Our second workload is the Transaction Processing Council's Benchmark H (TPC-H) [82], a standard benchmark for decision support workloads described previously. The TPC-H benchmark consists of twenty-two business oriented queries. Except for the initial loading of the database and the insert/delete functions, TPC-H consists almost entirely of reads with many sequential or looping scans of the database.

We were unable to capture TPC-H traces using the same Oracle environment as used by TPC-C for logistical reasons. We therefore chose to capture our traces using Postgres 8.3.7 [78]. Of the twenty-two queries, Postgres is unable to process queries Q17, Q20, and Q22 in a reasonable time so we have excluded them from our evaluation. Furthermore, some queries, such as query Q2, are CPU intensive, rather than I/O intensive and so are of less interest in our current study.

We report the results of running TPC-H with a scale factor of 10 which corresponds to about 10GB of raw data. In this configuration, the database system consumes roughly 20GB of storage space for data, indexes, logs, and temporary files. Choosing this scale factor allowed us to place the entire contents of the database on any of the SSDs used for evaluation purposes. The resulting cache traces vary from about seven hundred thousand cache operations for query sixteen to eight hundred million operations for query twenty one with most queries requiring three and half to four and a half million cache operations to complete.

5.2.3 Simulation and Replay Results

We have argued that the marginal utility of an additional DRAM cache block in a flash memory storage system is relatively modest. The value of the buffer cache has decreased with the advent of flash memory for two reasons: first, data set sizes have grown much faster than economical DRAM memory subsystems and second, the time to service a random read in flash memory is similar to the time to copy data from a DRAM buffer whereas the time to read a random block from disk is more than two orders of magnitude longer.

We have proposed a simple heuristic called Lazy LRU for managing the buffer cache in the presence of flash memory. We evaluate this heuristic using trace driven simulation and replay using the TPC-C and TPC-H workloads. Our primary evaluation criteria are cache hit rate and elapsed wall time for replay. We believe that

caches will not continue to grow in proportion with primary storage. As we observed in Section 4.4, large DRAM caches are expensive to build. The new generation of flash storage devices are able to service cache misses within a small multiple of the time to copy a block from the cache on a hit. If future caches are smaller as a fraction of primary storage, cache hit rates will inevitably fall. This places Lazy LRU at an interesting design point, particularly if it exhibits cache hit rates similar to LRU in practice. Lazy LRU adds the additional benefit of scan resistance and the ability to avoid unnecessary copying in a number of common cases such as isolated read-modify-write cycles.

TPC-C

A common rule of thumb for storage cache designers is that roughly 80% of the references in a storage system are to approximately 20% of the blocks. This skew in the reference pattern is an important ingredient in any effective caching strategy. Figure 5.2 shows the distribution of block popularities for the TPC-C trace. Blocks are sorted by the number of references and the resulting rank versus the natural logarithm of the number of references is plotted. In the case of the TPC-C trace, 80% of the references are to 22% of unique blocks referenced.

Given the presence of skew in the block reference distribution, it is natural to examine the effect of increasing the size of the cache. One way to evaluate the effect of the cache size to is plot the cache hit rate as a function of cache size. Another is to consider the elapsed wall time for trace replay, also as a function of cache size. In both cases, we have configured the simulator to use LRU to manage a cache with 8KB cache blocks, the same size as used in the default configuration of the Oracle RDBMS. We then vary the cache size from 64MB to 16GB. The simulator allows us to measure the cache hit rate and to generate an I/O trace for subsequent replay.

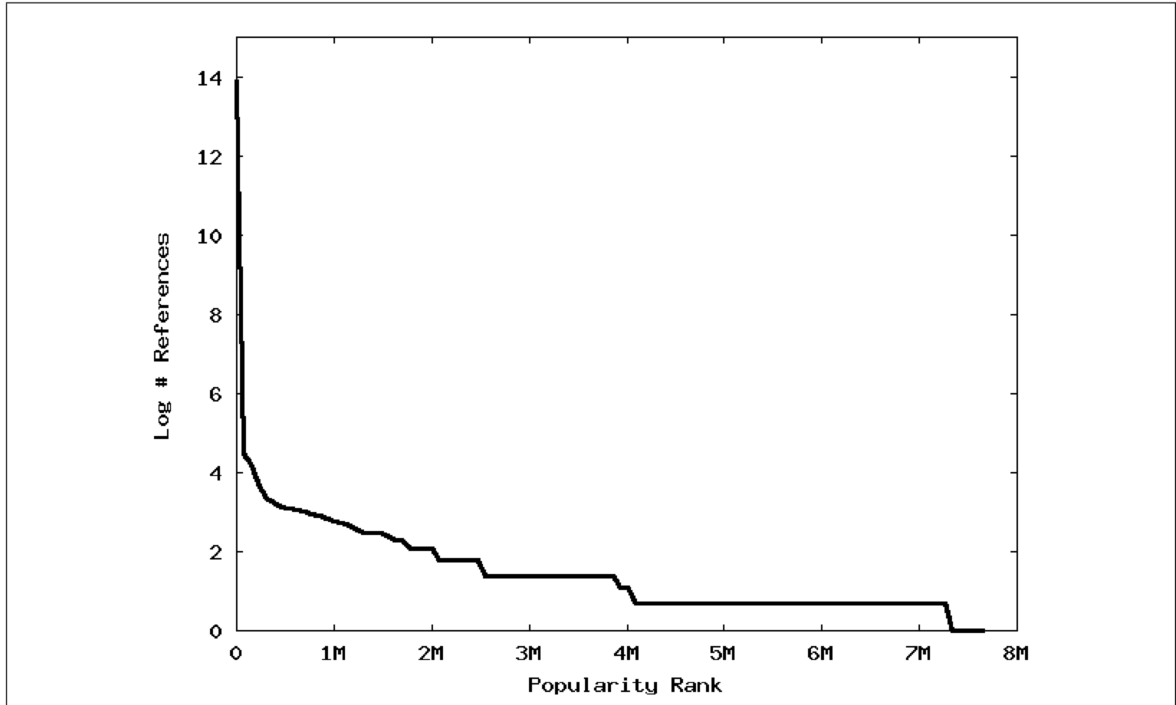

Figure 5.2: TPC-C: Number of Block References vs Logical Block Popularity Rank

Figure 5.3 shows the cache hit rate as a function of cache size. With a cache of 8192 8KB blocks consuming 64MB of DRAM, the cache hit rate is about 63%. With a cache that is over two orders of magnitude larger, the cache hit rate increases to approximately 80%. Although this is a substantial improvement in hit rate, it comes at a significant cost in DRAM: the total number of unique blocks touched during the 1000 warehouse TPC-C trace is about 7.2M or almost 60GB. A 64MB cache is about one-tenth of one percent of the total footprint of the trace whereas the 16GB cache is over twenty-five percent of the active data in the workload! In traditional storage systems where the cost of a main-memory cache miss is significant, it may make sense to rely on a cache that is between 10% and 25% the size of the dataset in order to achieve good performance. However, as dataset sizes continue to grow and the amount of flash available on a server grows faster than the available DRAM, this approach is unlikely to be economical.

An alternative way to look at the marginal utility of an additional DRAM cache block is consider the change in run time as a function of cache size. In other words,

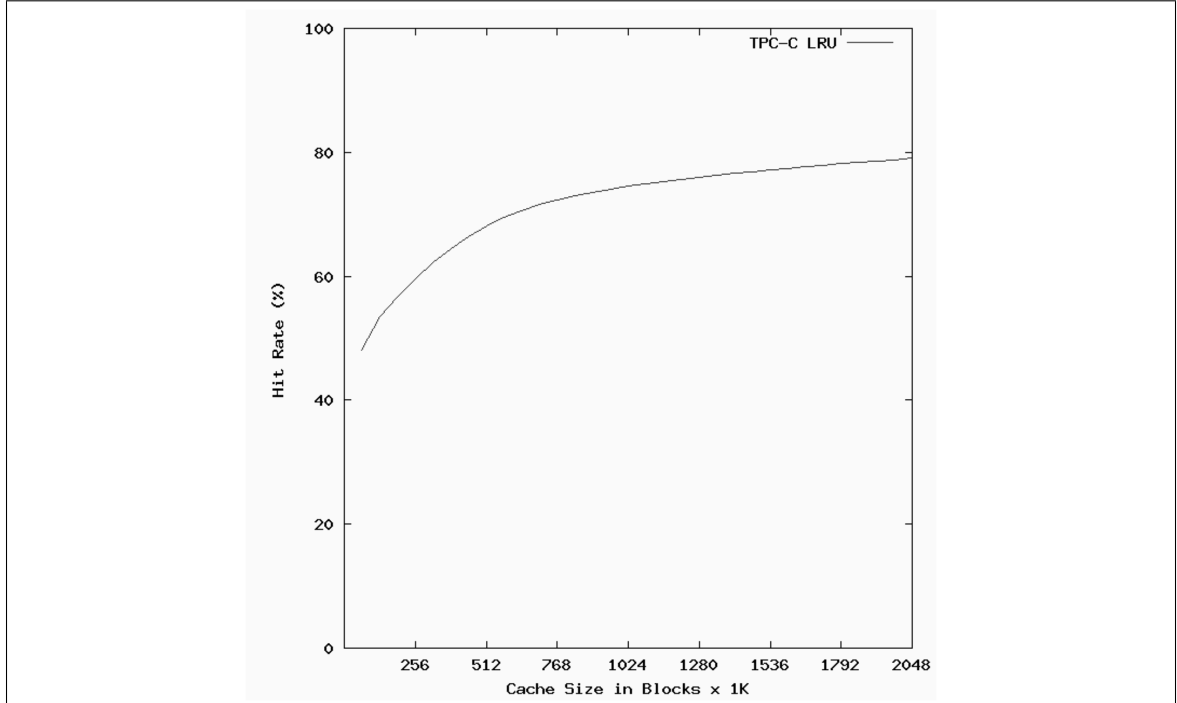

Figure 5.3: Hit Rate vs Cache Size for TPC-C With 1000 Warehouses

the marginal utility of a DRAM cache block is proportional to the derivative of the cache-size versus run-time curve. Figure 5.4 shows this tradeoff for the TPC-C trace over the same range of cache sizes as in Figure 5.3. We use the cache simulator to generate two I/O traces from the TPC-C cache request trace: one with the cache managed using ordinary LRU and one with the cache managed using Lazy LRU. For each cache size and cache management algorithm, we report the average elapsed wall time for five runs of the replay engine.

With a cache of 64K blocks, the I/O traces take almost 1200 seconds to run. Increasing the cache to 896K cache blocks cuts the runtime in half to about 600 seconds. An additional 896K cache blocks, or a total of 1792K cache blocks, yields a run a run time of approximately 475 seconds for a speedup of only 1.25. By the time the cache contains 2M blocks, the cache holds fully one quarter of the active data set, yet only provides a speed-up of about three over the original cache which held less than one tenth of one percent of the data set.

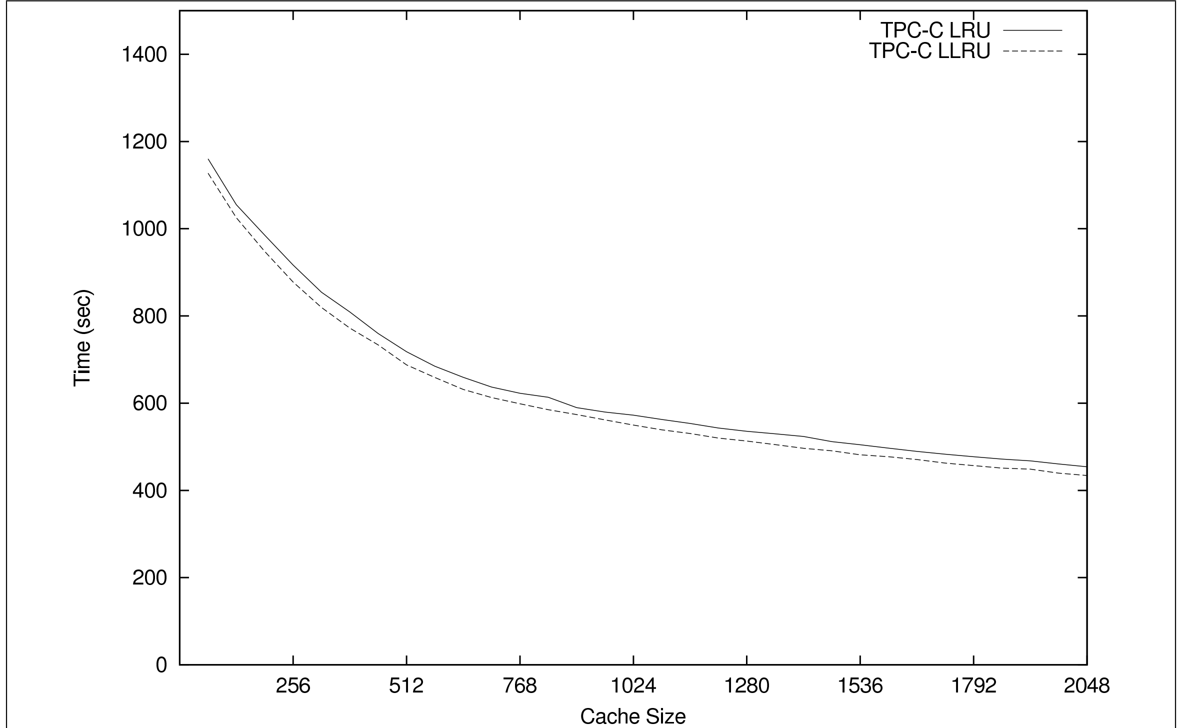

Figure 5.4: Execution Time vs Cache Size for TPC-C With 1000 Warehouses

Both the distribution of block popularities and the decreasing marginal return on additional cache blocks suggests that many blocks populating the cache are referenced only infrequently. Of particular interest are those blocks that are read precisely once. There is little advantage to placing blocks that are read precisely once in the cache at all. If, instead of caching these blocks, we transfer directly from flash to the user buffer, we can avoid an unnecessary in-memory copy. As the cost of a DMA transfer from flash approaches the cost of small block copies in main memory, this simple approach may yield real benefits. It is this observation that is the motivation for the Lazy LRU heuristic.

We classify blocks in three categories: (1) those blocks that are read once and then not touched again; (2) those blocks that are read once, modified once, and then written back; (3) all other blocks. Particularly if the inter-reference gap between read and modify is small in case (2), we can bypass the cache both on the initial read and on the subsequent write of the modified block. Figure 5.5 shows the fraction of reads

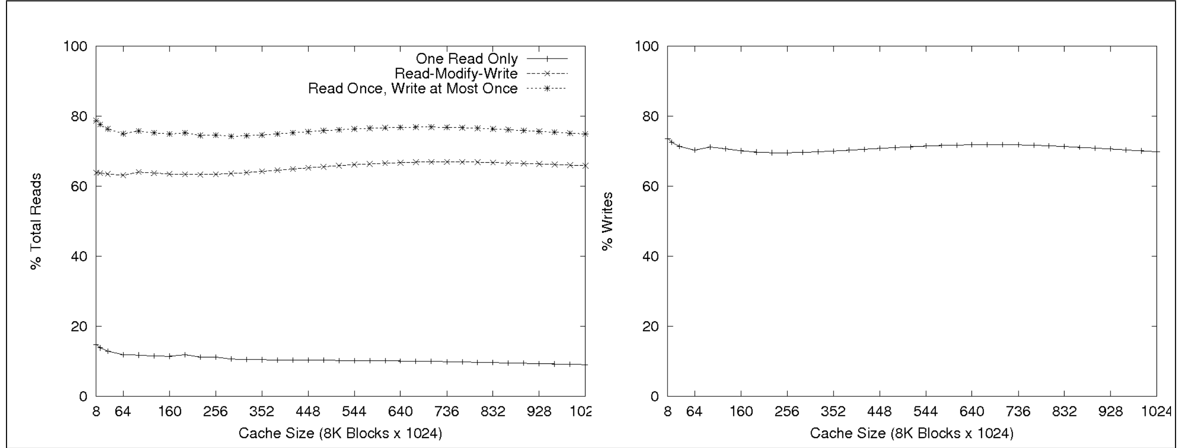

Figure 5.5: TPC-C Single Rate vs Cache Size

that fall in categories (1) and (2) as well as the fraction of writes that fall in category (2). Almost 80% of blocks read are read only once and between 70 and 75% of blocks written are read once, modified once, and then written back. In other words, between seventy and eighty percent of blocks are "cold" blocks while the remaining twenty to thirty percent of blocks are "hot". Moreover, this classification is stable even as the cache size scales from 8K cache blocks to 2M cache blocks.

It is worth noting that an implementation may modify the definition of read-once to mean read-once before being ejected from the cache. Although we adopt the more stringent definition for the purposes of analysis, there is little difference between two blocks that are each read once in the entire trace and one that is read twice but ejected from the cache in the interim due to insufficient capacity. This alternate definition of the read-once criterion can be used to simplify the practical implementation of the Lazy LRU heuristic.

Figure 5.4 illustrates the result of the Lazy LRU heuristic for the TPC-C workload over a range of cache sizes. Lazy LRU is consistently 4-4.5% faster than LRU. This performance improvement, while modest, is achieved with no improvement in the hit rate. In fact, LRU has a negligibly better cache hit rate for this workload. The performance difference between the two algorithms is entirely due to the reduced number of block copy operations in the lazy variant. Given the level of optimization

and the effort expended to achieve good performance in modern storage systems, even a 4% improvement is probably worthwhile given the simplicity of the heuristic.

TPC-H

Query Number	Cache Size in Blocks: 8192	262144	524288	1048576
1	1.00	1.00	1.00	1.00
2	11.34	12.01	12.01	12.01
3	1.00	1.00	1.00	1.00
4	1.01	1.01	1.01	1.01
5	1.01	1.01	1.01	1.01
6	1.00	1.00	1.00	1.00
7	1.00	1.00	1.00	1.00
8	1.00	1.00	1.00	1.00
9	8.48	8.49	8.49	8.49
10	1.01	1.01	1.01	1.01
11	1.01	2.01	2.01	2.01
12	1.00	1.00	1.00	1.00
13	5.74	5.75	5.75	5.75
14	1.00	1.00	1.00	1.00
15	1.00	1.00	1.00	1.00
16	1.01	1.01	1.01	1.01
18	1.00	1.00	1.00	1.00
19	1.00	1.00	1.00	1.00
21	282.09	283.42	283.42	283.42

(a) Average Number of References Per Block as a Function of Cache Size

Query Number	Number of Refs. x 1K	Inter-Ref. Gap
1	1189	4.6
2	2789	1,900.0
3	1492	8.6
4	270	—
5	270	—
6	1187	4.3
7	1494	6.1
8	1536	8.1
9	14579	11.5
10	270	6.3
11	363	90,000.0
12	1187	7.1
13	1775	28.2
14	1188	7.3
15	1187	6.7
16	224	10.5
18	2410	575,000.0
19	1229	8.3
21	412989	129.8

(b) Number of Block References and Inter-Reference Gap

Table 5.12: TPC-H Descriptive Statistics

The TPC-H benchmark models a decision support workload which is essentially read only after the database has been initialized and populated. Decision support workloads tend to be characterized by large sequential table scans, loops, and random reads. For a sufficiently large data set, both sequential scans and random I/O preclude effective caching but can benefit from intelligent prefetching. The effectiveness of a caching strategy in the case of looping I/O patterns depends upon the ability to

detect the behavior, identify the data blocks referenced as part of the loop, and the number of blocks in the loop body relative to the size of the cache.

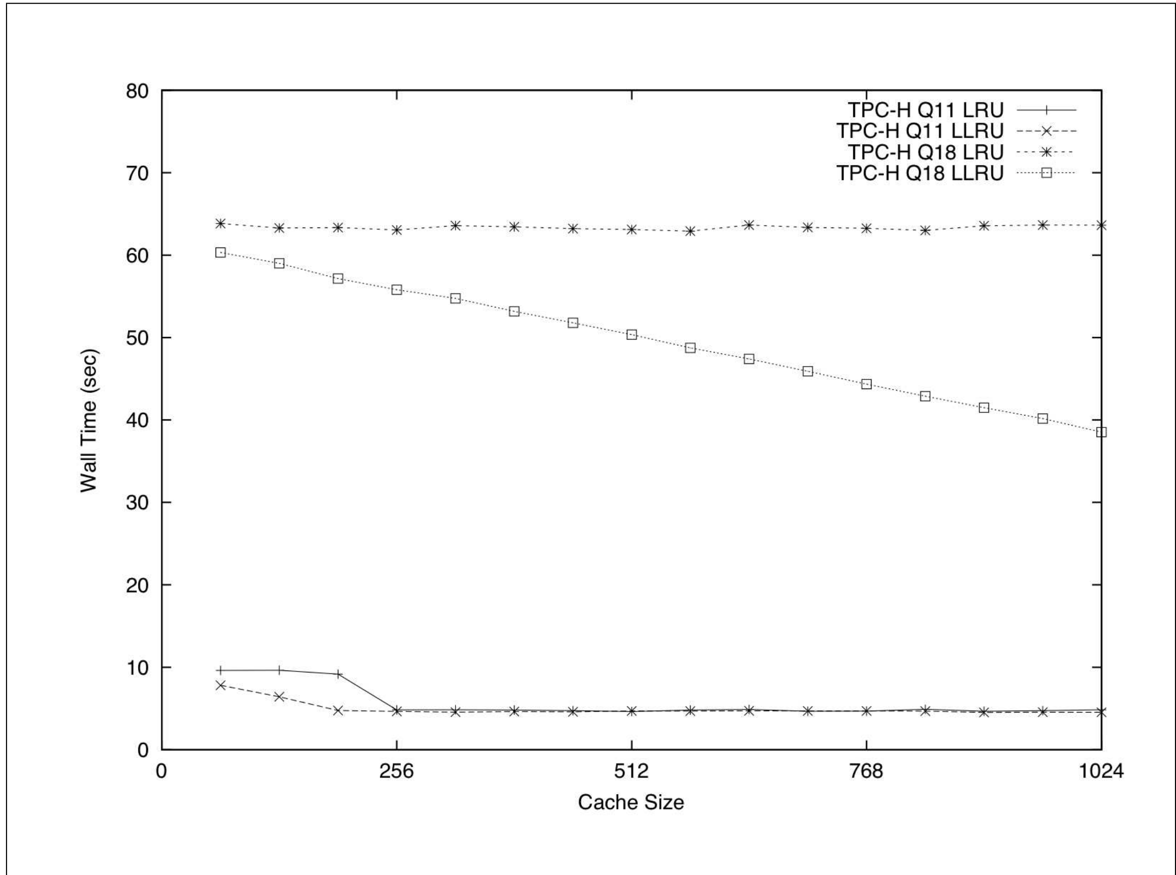

Figure 5.6: Run time for TPC-H Queries 11 and 18 using LRU and Lazy LRU to manage the cache. Both queries benefit from the scan resistance of Lazy LRU.

Table 5.12(a) shows for each query the average number of references per block for all blocks read during the trace. For most queries, the average number of references to each logical block is very nearly one. In fact, with the exception of the five queries Q2, Q9, Q11, Q13, and Q21, the average number of hits is essentially one at all cache sizes from eight thousand cache blocks up to one million cache blocks. Of these five queries, only query Q11 shows an appreciable difference in the average number of references to data blocks as a function of cache size: once the cache is large enough, the average increases from 1.01 to 2.01 references. The average of roughly one hit per block in the remaining fourteen queries is indicative of either random access or

sequential scanning behavior common to relational table joins. In both cases, blocks are typically read in and used only once during the processing of an individual query. Lazy LRU is well suited to this type of I/O behavior: there is little benefit to keeping a block in the cache if it used only once and there is no need to incur the overhead of a copy from the cache to the user buffer in this case either.

The relatively small number of references per block for most TPC-H queries is common to decision support workloads. Common primitives include table scans, nested loops to implement inner joins, and random access as a part of hash joins. With the exception of some looping constructs, these types of primitive generate I/O workloads that are difficult or impossible for cache management algorithms to capture and capitalize upon. These types of workloads are instead better suited to the wide variety of prefetching algorithms, particularly application directed or informed prefetching [68].

A second metric of interest in addition to the number of references per data block is the average inter-reference gap. Table 5.12(b) shows the total number of block references and the average inter-reference gap for each of the queries in our TPC-H benchmark. Queries Q4 and Q5 determine product order priority and local supplier volume, respectively. Typical query plans for these queries consist of a sequence of inner joins, often implemented as hash joins, a hash-based aggregation, and a final sort phase. So few blocks are referenced more than once using our database setup that the average inter-reference gap for these queries is not informative. For most other queries, the inter-reference gap is relatively small – less than two hundred block references. When the inter-reference gap is small relative to the cache size, the LRU replacement algorithm is well suited to capturing the locality present in the reference pattern. Queries Q2, Q11, and Q18, on the other hand have average inter-reference gaps across the entire query that are several orders of magnitude larger. Query Q2 is

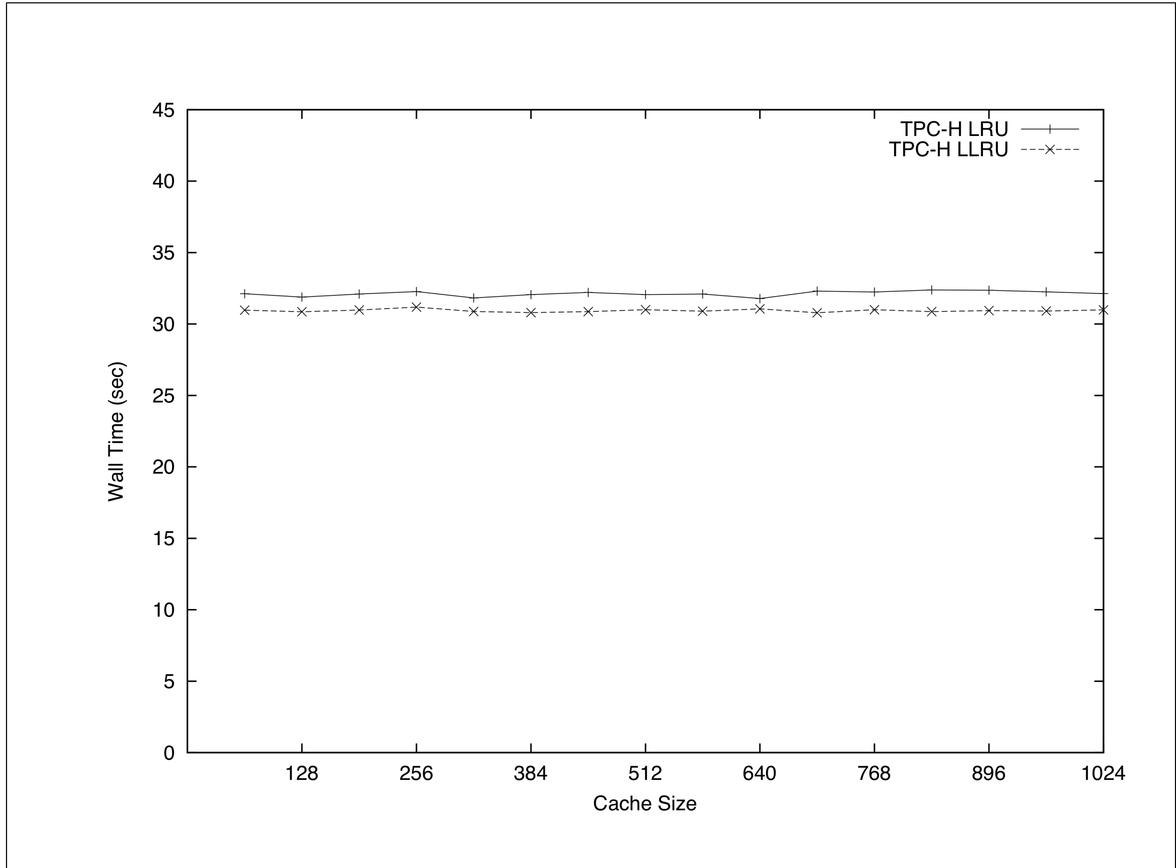

Figure 5.7: Run time for TPC-H Query 19 using LRU and Lazy LRU to manage the cache. Query 19 is representative of many TPC-H queries that require scanning tables larger than the available cache. Lazy LRU benefits from reduced number of copy operations between the cache and user buffers.

essentially CPU bound in our database configuration. Queries Q11 and Q18, on the other hand, exhibit interesting cache behavior.

Queries Q11 and Q18 identify important stock items and large customers, respectively. Query Q11 has an average inter-reference gap of 90K references and query Q18 an average inter-reference gap of 575K references. Both are typically implemented as a sequence of nested loops computing inner joins. In the latter case, the average number of references per block is very nearly one, indicating significant outliers. In the former case, the average number of hits per block is one for small caches, but once the cache is sufficiently large, each block is referenced twice before being ejected. In both cases, the large average inter-reference gap means that a relatively large cache

is necessary to capture the underlying block reference loop. This is particularly true for ordinary LRU as the cache must be large enough to hold the number of distinct intervening blocks. As illustrated in Figure 5.6, the Lazy LRU heuristic is able to capture the block reference loop more effectively than is ordinary LRU. Lazy LRU is also subject to capacity constraints, of course, but only need keep very basic, compact metadata for blocks that are infrequently referenced and therefore not placed in the cache. As a result, the Lazy LRU heuristic automatically keeps more blocks that are referenced multiple times in the loop body in the cache. The Lazy LRU heuristic is thus able to improve the cache hit rate and consequently reduce both the number of I/O transactions and elapsed wall time without application input. Although Figure 5.6 shows elapsed wall time, the results are qualitatively similar if one plots the number of I/O transactions instead. In fact, even query Q2 shows similar behavior when I/O transactions are plotted. Since query Q2 has a significant CPU intensive component, the reduction in the number of I/O transactions at small cache sizes has only a very modest impact on run time.

We have thus far identified several queries in the TPC-H traces where performance and cache behavior differ significantly when the cache is managed with the Lazy LRU heuristic as opposed to ordinary LRU. For these queries, the average number of references per block is greater than one, sometimes significantly so, and the inter-reference gap is also significantly larger than it is for most other queries. Lazy LRU is also a reasonable choice for the remaining queries that do not exhibit these features. The remaining queries have reference patterns in which logical blocks are typically referenced once and then discarded, either due to index scans or random access patterns in hash joins. Query Q19 is representative of the remaining queries and, as shown in Figure 5.7, the Lazy LRU heuristic again gives a modest performance improvement. Figure 5.7 shows the average elapsed wall time for five runs for each cache size. The difference in run time between the Lazy LRU heuristic and ordinary LRU varies

from as little as 2.3% to as much as 4.7%, but on average is about 3.8%. Although the difference in performance is relatively small, the heuristic is applicable to other cache management algorithms as well, is simple to implement, and is applicable to workloads with both scanning and looping I/O behavior.

5.3 Summary

DFS is a simple, robust, high-performance file system that takes advantage of the implementation and performance characteristics of modern solid state storage devices. DFS delegates the majority of storage management and layout to the underlying device, resulting in a significant reduction in complexity. The prototype implementation of DFS is about one-eighth the size of the implementation of ext3, a mature and widely deployed Linux file system. Much of this difference is in the logging, block allocation, and I-node manipulation routines which are essential to any file system. Microbenchmark results further demonstrate that DFS is able to deliver performance closer to that of the raw hardware than existing file systems such as ext3. For direct access, DFS delivers as much as 20% performance improvement. Furthermore, DFS continues to outperform ext3 by 7% to as much as 250% in application level benchmarks while consuming fewer CPU resources.

File system performance is intimately tied to buffer cache performance. We have argued that the marginal value of additional DRAM cache has fallen with the advent of high performance solid state storage. For instance, we found through trace driven simulation and replay that increasing the buffer cache from less than one tenth of one percent of the data set size to one quarter the data set size only improved performance by a factor of three. For common data mining workloads that scan large volumes of data, the situation is often worse. As a result, we have proposed the Lazy LRU heuristic which defers populating the cache until a block has been determined to be

sufficiently popular and avoids unnecessary copies in the buffer cache. We found that this approach yields roughly a 4% performance improvement at essentially no cost for transaction processing workloads. For some data mining workloads, the improvement is much larger as Lazy LRU is scan resistant, permitting frequently used data such as that subject to looping reference patterns, to remain in the cache while infrequently used data subject to scanning reference patterns is prevented from polluting the cache.

Chapter 6

Conclusion and Future Work

We have argued that the recent rush to adopt flash memory in enterprise grade storage systems is driven chiefly by a combination of three factors: power consumption, random I/O performance, and cost. Disk based storage systems require significant over-provisioning in order to provide good random I/O performance. High performance disks can deliver roughly two hundred random I/Os per second per spindle. Continued advances in magnetic recording technology have dramatically improved disk density but have not appreciably improved disk arm positioning delay. As a result, high performance storage systems must provision disks based on random I/O performance first and size second. Individual magnetic disks consume roughly fifteen watts when active and almost ten when idle but not powered down. Depending on the implementation, existing SSDs may consume twenty to eighty percent of the power that a magnetic disk does when active. When idle, SSDs may use as little as 60mW or two orders of magnitude less power than magnetic disks. The gap in power usage between disks and SSDs widens by one to two orders of magnitude when over-provisioning for performance in magnetic storage systems is taken into account.

A persistent concern about flash memory storage systems is reliability. Magnetic disks are a venerable technology and the storage community has had long experience

with them. Flash memory failure modes are substantially different from those of traditional storage systems. Limited write endurance is a real problem and the recent adoption of MLC NAND flash in enterprise products has exacerbated concerns. We have argued that with proper software support, even MLC NAND flash is suitable for building enterprise grade storage systems. Log-structured storage and garbage collection are a fact of life if flash storage systems are to achieve good performance and these same techniques are effective for managing the wear leveling problem as well.

The advent of sophisticated flash translation layers to manage flash memory resources for performance and wear leveling has tended to push what were once high-level storage system design problems further down into the storage stack. The copy-on-write nature of flash memory has revived interest in log-structured storage. Some flash storage devices have chosen to implement the log-structured flash translation layer in the storage device itself as firmware; others, such as FusionIO, have pushed much of the intelligence into the device driver in the host kernel. In both cases, the flash translation layer has co-evolved with the storage hardware architecture in order to achieve the highest level of performance possible. We have observed that the problems the flash translation layer must solve are closely related to the problems that the file system and other storage consumers have traditionally had to solve as well. These problems include storage allocation, layout, and provisions for crash recovery. Rather than duplicate the solution to these problems at two points in the storage stack, we advocate delegating the solution to the flash translation layer. The flash translation layer can continue to co-evolve with the hardware while providing an interface to consumers that avoids duplication. We have shown that this approach greatly simplifies the design and implementation of a file system such as DFS. Moreover, DFS is able to deliver higher performance using the same hardware. Figure 6.1 illustrates the advantages of DFS and the new generation of flash memory based primary storage over

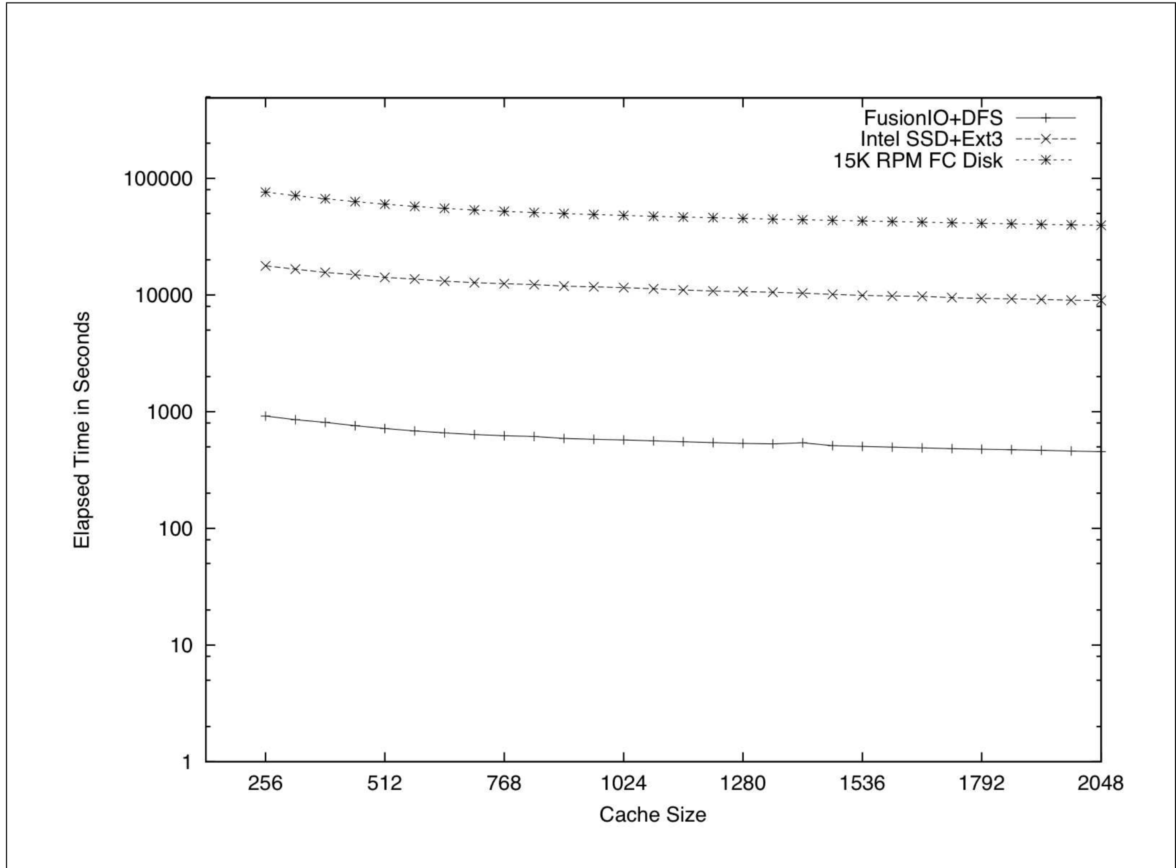

Figure 6.1: Elapsed Time for TPC-C Running on FusionIO and Intel SSDs as a Function of Cache Size

existing technologies. The combination of the new generation of SSDs and the elimination of an unnecessary layer of indirection by DFS yields an order of magnitude improvement over existing SSDs with legacy file systems which themselves offered an order of magnitude performance improvement over magnetic disks.

We have further argued that as flash memory storage systems improve, the utility of the traditional buffer cache decreases. With existing flash storage subsystems that achieve between 100K and 300K random I/Os per second, the time to copy a block from the buffer cache to a user buffer or processor cache is within a small factor of the time to serve the miss from flash. In other words, with a system such as the FusionIO ioDrive and DFS, the marginal utility of each additional DRAM buffer is relatively modest. Furthermore, large main memories require sophisticated and

expensive memory system designs so that doubling the size of main memory more than doubles memory system cost. DRAM based memory systems also consume a significant amount of power even when quiescent. As a result, we have argued for a more aggressive approach to cache design that attempts to segregate blocks on the basis of popularity and does not even attempt to cache anything but the most popular blocks. A straightforward example of this approach is the Lazy LRU heuristic which only populates the cache on the second reference to a block. Although it is possible to construct workloads for which Lazy LRU is inefficient, it is a reasonable heuristic for several common and important workloads.

In Section 4.5 we discussed some of the limitations of the current implementation of DFS and some concrete future work required to turn the existing prototype into a commercial product. Certainly a more complete implementation of the virtualized flash storage layer and DFS atop it is one avenue for future engineering work. There is also room for further improvement of the Lazy LRU heuristic specifically and its application to other common cache algorithms. There is a wealth of research on cache algorithms, but as flash storage system continue to grow, the gap between the main memory buffer cache and the underlying storage system will continue to grow as well. At the same time, the gap between the cost of a cache hit and a cache miss is much smaller for these new storage systems. Existing research is not well suited to this environment.

In our discussion of flash memory we have focused on local storage. Most applications in the data center are now distributed across multiple hosts either in a single data center or possibly in the wide area. We still believe that focusing on the local storage case is a sensible starting point. Applications distributed across a cluster still must manage local storage effectively. File systems such as DFS also find application inside network attached storage filer heads and in many database applications. Nonetheless, an important avenue for future research is how to adapt flash storage

to a distributed environment. We believe that the virtualized flash storage layer provides a good foundation for further work in this area. We have believe that both local and distributed applications will benefit from delegating storage management and that specific interfaces such as key/value storage can form the basis for richer data structures in both local and distributed applications.

Our description of the virtualized flash storage layer and our implementation and evaluation of DFS have focused on the use of flash memory to replace magnetic disks for primary storage. Indeed, DFS is intended to be a primary storage file system. For many applications, flash memory based storage systems are now inexpensive enough that it is feasible to replace disks entirely with flash memory. This does not mean that magnetic disks are doomed to obsolescence. Rather, we anticipate that magnetic disks will be relegated to the role of secondary storage. Disks still offer good sequential performance and high density at low cost. Flash memory shines when random I/O performance is critical. The longer term research question is how best to marry the two.

Bibliography

[1] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy, "Design tradeoffs for SSD performance," in *Proceedings of the 2008 USENIX Technical Conference*, June 2008.

[2] *Technical Note TN1150: HFS Plus Volume Format*, Apple Corporation, March 2004.

[3] A. Ban, "Flash file system," U.S. Patent 5 404 485, April 4, 1995.

[4] A. Ban and R. Hasharon, "Flash file system," U.S. Patent 6 732 221, June 1, 2001.

[5] S. Bansal and D. S. Modha, "CAR: Clock with adaptive replacement," in *Proceedings of the 2004 Conference on File and Storage Technologies*, 2004, pp. 187–200.

[6] R. Bez, E. Camerlenghi, A. Modelli, and A. Visconti, "Introduction to flash memory," in *Proceedings of the IEEE*, vol. 91, no. 4, 2003, pp. 489–502.

[7] A. Birrell, M. Isard, C. Thacker, and T. Wobber, "A design for high-performance flash disks," *ACM Operating Systems Review*, vol. 41, no. 2, April 2007.

[8] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, pp. 422–426, 1970.

[9] T. Brants and A. Franz, "Web 1T 5-gram version 1," 2006.

[10] R. Card, T. T'so, and S. Tweedie, "The design and implementation of the second extended filesystem," in *First Dutch International Symposium on Linux*, December 1994.

[11] A. Carlson, T. M. Mitchell, and I. Fette, "Data analysis project: Leveraging massive textual corpora using n-gram statistics," Carnegie Mellon University Machine Learning Department, Tech. Rep. CMU-ML-08-107, May 2008.

[12] A. Chang, M. F. Mergen, R. K. Rader, J. A. Roberts, and S. L. Porter, "Evolution of storage facilities in AIX version 3 for RISC system/6000 processors," *IBM Journal of Research and Development*, vol. 34, no. 1, Jan. 1990.

[13] L.-P. Chang, "On efficient wear leveling for large-scale flash-memory storage systems," in *Proceedings of the 2007 ACM Symposium on Applied Computing*. New York, NY: ACM, 2007, pp. 1126–1130.

[14] Cisco Systems, "Cisco content delivery engines generation 2," http://www.cisco.com/en/US/prod/collateral/video/ps7191/ps7126/product_data_sheet0900aecd8057f446.pdf, 2010.

[15] J. Cooke, "Flash memory 101: An introduction to NAND flash," *EETimes*, March 2006.

[16] H. Custer, *Inside the Windows NT File System*. Microsoft Press, 1994.

[17] C. Dirik and B. Jacob, "The performance of PC solid-state disks (SSDs) as a function of bandwidth, concurrency, device architecture, and system organization," in *ISCA 2009: Proceedings of the 36th Annual International Symposium on Computer Architecture*, 2009, pp. 279–289.

[18] J. R. Douceur and W. J. Bolosky, "A large scale study of file-system contents," in *Proceedings of the 1999 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, 1999.

[19] F. Douglis, R. Caceres, M. F. Kaashoek, K. Li, B. Marsh, and J. A. Tauber, "Storage alternatives for mobile computers," in *Operating Systems Design and Implementation*, 1994, pp. 25–37.

[20] "DRAM and NAND Flash Contract Prices," http://www.dramexchange.com, DRAM Exchange, a TrendForce Company.

[21] "Volume and file structure of disk cartridges for information interchange," http://www.ecma-international.org/publications/standards/Ecma-107.htm, ECMA International, June 1995.

[22] M. Fujio and I. Hisakazu, "Semiconductor memory device and method for manufacturing the same," U.S. Patent 4 531 203, 1981.

[23] FusionIO Corporation, "FusionIO ioDrive specification sheet," http://www.fusionio.com/PDFs/Fusion%20Specsheet.pdf, 2009.

[24] G. R. Ganger, M. K. McKusick, C. A. N. Soules, and Y. N. Patt, "Soft updates: A solution to the metadata update problem in file systems," *ACM Transactions on Computing Systems*, vol. 18, no. 2, pp. 127–153, 2000.

[25] J. Gecsei, D. R. Slutz, and I. L. Traiger, "Evaluation techniques for storage hierarchies," *IBM Systems Journal*, vol. 9, no. 2, pp. 78–117, 1970.

[26] G. Gibson, D. Nagle, K. Amiri, F. Chang, E. Feinberg, H. Gobioff, C. Lee, B. Ozceri, E. Riedel, D. Rochberg, and J. Zelenka, "File server scaling with network-attached secure disks," in *SIGMETRICS '97: Proceedings of the 1997*

ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, 1997, pp. 272–284.

[27] J. Gray and C. van Ingen, "Empirical measurements of disk failure rates and error rates," Microsoft Research, Tech. Rep. MSR-TR-2005-166, December 2005.

[28] A. Gupta, Y. Kim, and B. Urgaonkar, "DFTL: A flash translation layer employing demand-based selective caching of page-level address mappings," in *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*. New York, NY: ACM, 2009, pp. 229–240.

[29] S.-W. Han, "Flash memory wear leveling system and method," U.S. Patent 6 016 275, November 4, 1998.

[30] J. Handy, "The changing relationship of flash and DRAM SSDs," *Objective Analysis*, August 2007.

[31] *OpenVMS System Manager's Manual, Volume 2: Tuning, Monitoring, and Complex Systems*, Hewlett-Packard Development Company, September 2003.

[32] D. Hitz, J. Lau, and M. Malcom, "File system design for an NFS file server appliance," NetApp Corporation, Tech. Rep. TR-3002, September 2001.

[33] X.-Y. Hu, E. Eleftheriou, R. Haas, I. Iliadis, and R. Pletka, "Write amplification analysis in flash-based solid state drives," in *Proceedings of SYSTOR 2009*, 2009, pp. 1–9.

[34] *OS/VS2 MVS Overview*, IBM Corporation, June 1978.

[35] Intel Corporation, "Understanding the flash translation layer (FTL) specification," December 1998.

[36] ——, "Intel X25-E SATA solid state drive," http://download.intel.com/design/flash/nand/extreme/extreme-sata-ssd-datasheet.pdf, 2009.

[37] ——, "Product brief: Intel NAND flash memory," ftp://download.intel.com/pressroom/kits/vssdrives/Nand_PB.pdf, 2009.

[38] H. Jo, J.-U. Kang, S.-Y. Park, J.-S. Kim, and K. Lee, "FAB: Flash-aware buffer management policy for portable media players," *IEEE Transactions on Consumer Electronics*, vol. 52, no. 2, pp. 485–493, 2006.

[39] T. Johnson and D. Shasha, "2Q: A low overhead high performance buffer management replacement algorithm," 1994.

[40] W. K. Josephson, L. A. Bongo, D. Flynn, and K. Li, "DFS: A filesystem for virtualized flash storage," in *Proceedings of the 8th USENIX Conference on File and Storage Technologies*, February 2010.

[41] E. Jou and J. Jeppesen III, "Flash memory wear leveling system providing immediate direct access to microprocessor," U.S. Patent 5 568 423, April 14, 1995.

[42] D. Kahng and S. M. Sze, "A floating-gate and its application to memory devices," *The Bell System Technical Journal*, vol. 46, no. 4, pp. 1288–1295, 1967.

[43] M. A. Kandaswamy and R. L. Knighten, "I/O phase characterization of TPC-H query operations," in *Proceedings of the 4th International Computer Performance and Dependability Symposium*. IEEE Computer Society, 2000, p. 81.

[44] A. Kawaguchi, S. Nishioka, and H. Motoda, "A flash-memory based file system," in *Proceedings of the Winter 1995 USENIX Technical Conference*, 1995.

[45] H. Kim and S. Ahn, "BPLRU: A buffer management scheme for improving random writes in flash storage," in *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, February 2008.

[46] J. Kim, J. M. Kim, S. H. Noh, S. L. Min, and Y. Cho, "A space-efficient flash translation layer for compactflash systems," *IEEE Transactions on Consumer Electronics*, vol. 48, pp. 366–375, 2002.

[47] S.-W. Lee, W.-K. Choi, and D.-J. Park, "FAST: An efficient flash translation layer for flash memory," in *Lecture Notes in Computer Science*, August 2006, pp. 879–887.

[48] S. Lee, D. Shin, Y.-J. Kim, and J. Kim, "LAST: Locality-aware sector translation for NAND flash memory-based storage systems," *SIGOPS Operating Systems Review*, vol. 42, no. 6, pp. 36–42, 2008.

[49] M. Levy, "Interfacing Microsoft's flash file system," in *Memory Products*. Intel Corporation, 1993, pp. 4–318 – 4–325.

[50] K. Li, "Towards a low power file system," University of California at Berkeley, Tech. Rep. CSD-94-814, May 1994.

[51] D. R. Llanos, "TPCC-UVa: An open-source TPC-C implementation for global performance measurement of comptuer systems," *ACM SIGMOD Record*, vol. 35, no. 4, pp. 6–15, December 2006.

[52] K. Lofgren, R. Normand, G. Thelin, and A. Gupta, "Wear leveling techniques for flash EEPROM systems," U.S. Patent 7 353 325, Jan., 2005.

[53] K. Loney, *Oracle Database 10g: The Complete Reference*. Oracle Press, 2004.

[54] D. Malone, "UFS/FFS optimisations: Softupdates, dirpref, and dirhash," in *BSDCon 2001*, Nov. 2001.

[55] C. Manning, "YAFFS: The NAND-specific flash file system," *LinuxDevices.Org*, September 2002.

[56] B. Marsh, F. Douglis, and P. Krishnan, "Flash memory file caching for mobile computers," in *Proceedings of the Twenty-Seventh Hawaii International Conference on Architecture*, January 1994.

[57] J. Marshall and C. Manning, "Flash file management system," U.S. Patent 5 832 493, April 24, 1997.

[58] A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, and L. Vivier, "The new ext4 filesystem: Current status and future plans," in *Ottawa Linux Symposium*, June 2007.

[59] J. Matthews, S. Trika, D. Hensgen, R. Coulson, and K. Grimsrud, "Intel turbo memory: Nonvolatile disk caches in the storage hierarchy of mainstream computer systems," *ACM Transactions on Storage*, vol. 4, no. 2, May 2008.

[60] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry, "A fast file system for UNIX," *ACM Transactions on Computer Systems*, vol. 2, no. 3, August 1984.

[61] N. Megiddo and D. Modha, "ARC: A self-tuning, low overhead replacement cache," in *Proc. of the 2003 Conference on File and Storage Technologies*, 2003, pp. 115–130.

[62] S. Microsystems, "Sun enterprise solid-state drives and flash drives," http://www.oracle.com/us/products/servers-storage/storage/flash-storage/index.html, August 2010.

[63] N. Mielke, T. Marquart, N. Wu, J. Kessenich, H. Belgal, E. Schares, F. Trivedi, E. Goodness, and L. R. Nevill, "Bit error rate in NAND flash memories," in *Reliability Physics Symposium, 2008. IRPS 2008. IEEE International*, July 2008, pp. 9–19.

[64] V. F. Nicola, A. Dan, and D. M. Dias, "Analysis of the generalized clock buffer replacement scheme for database transaction processing," *SIGMETRICS Performance Evaluation Review*, vol. 20, no. 1, pp. 35–46, 1992.

[65] W. Norcott, "Iozone filesystem benchmark," http://www.iozone.org, 2009.

[66] Y. Ou, T. Härder, and P. Jin, "CFDC: A flash-aware replacement policy for database buffer management," in *DaMoN '09: Proceedings of the Fifth International Workshop on Data Management on New Hardware*, 2009, pp. 15–20.

[67] S.-Y. Park, D. Jung, J.-U. Kang, J.-S. Kim, and J. Lee, “CFLRU: A replacement algorithm for flash memory,” in *Procieedings of the 2006 International Conference on Compilers, Architecture and Syntehsis for embedded Systems*, 2006.

[68] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka, “Informed prefetching and caching,” in *In Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, 1995, pp. 79–95.

[69] P. Pavan, R. Bez, P. Olivo, and E. Zanomi, “Flash memory cells – an overview,” in *Proceedings of the IEEE*, vol. 85, no. 8, August 1997, pp. 1248–1271.

[70] V. Prabhakaran, T. L. Rodeheffer, and L. Zhou, “Transactional flash,” in *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, December 2008.

[71] A. Rajimwale, V. Prabhakaran, and J. D. Davis, “Block management in solid state devices,” January 2009, unpublished Technical Report.

[72] “Windows vistas features explained,” http://www.microsoft.com/windows/products/windowsvista/features/details/performance.mspx, 2007, microsoft Corporation.

[73] J. T. Robinson and M. V. Devarakonda, “Data cache management using frequency-based replacement,” in *Proceedings of the 1990 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, 1990, pp. 134–142.

[74] M. Rosenblum and J. K. Ousterhout, “The design and implementation of a log-structured file system,” *ACM Transactions on Computer Systems*, vol. 10, pp. 1–15, 1992.

[75] Seagate Corporation, “DS Momentus: Powerful, Intelligent, Affordable Solid State Hybrid Drive,” http://www.seagate.com/docs/pdf/datasheet/disc/ds_momentus_xt.pdf, 2010.

[76] M. I. Seltzer, G. R. Ganger, M. K. McKusick, K. A. Smith, C. A. N. Soules, and C. A. Stein, “Journaling versus soft updates: Asynchronous meta-data protection in file systems,” in *Proceedings of the 2000 USENIX Annual Technical Conference*, 2000.

[77] D. Seo and D. Shin, “Recently-evicted first buffer replacement policy for flash storage devices,” *IEEE Transactions on Consumer Electronics*, vol. 54, no. 2, August 2008.

[78] M. Stonebraker, L. A. Rowe, and M. Hirohama, “The implementation of Postgres,” in *IEEE Transactions on Knowledge and Data Engineering*, 1990, pp. 340–355.

[79] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck, “Scalability in the XFS file system,” in *In Proceedings of the 1996 USENIX Annual Technical Conference*, 1996, pp. 1–14.

[80] A. S. Tanenbaum, J. N. Herder, and H. Bos, “File size distribution in UNIX systems: Then and now,” *ACMSIGOPS Operating Systems Review*, vol. 40, no. 1, pp. 100–104, January 2006.

[81] J. Thatcher, T. Coughlin, J. Handy, and N. Ekker, *NAND Flash Solid State Storage for the Enterprise: An In-Depth Look At Reliability*. Storage Network Industry Association, 2009.

[82] Transaction Processing Performance Council, “TPC Benchmark H: Decision Support,” 2008, http://www.tpc.org/tpch.

[83] Transaction Processing Performance Council (TPC), “TPC Benchmark C: Online Transaction Processing,” 1992, http://www.tpc.org/tpcc.

[84] O. Troyanskaya, M. Cantor, G. Sherlock, P. Brown, T. Hastieevor, R. Tibshirani, D. Botstein, and R. B. Altman, “Missing value estimation methods for DNA microarrays,” *Bioinformatics*, vol. 17, no. 6, pp. 520–525, 2001.

[85] S. Tweedie, “Ext3, journaling filesystem,” in *Ottawa Linux Symposium*, July 2000.

[86] C. Ulmer and M. Gokhale, “Threading opportunities in high-performance flash-memory storage,” in *High Performance Embedded Computing*, 2008.

[87] S. Wells, “Method for wear leveling in a flash EEPROM memory,” U.S. Patent 5 341 339, August 23, 1994.

[88] D. Woodhouse, “JFFS: The journalling flash file system,” in *Ottawa Linux Symposium*, 2001.

[89] M. Wu and W. Zwaenepoel, “eNVy: A non-volatile, main memory storage system,” in *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1994.

[90] Y. Zhou, J. F. Philbin, and K. Li, “The multi-queue replacement algorithm for second level buffer caches,” in *Proceedings of the 2001 USENIX Annual Technical Conference*, 2001, pp. 91–104.