The Pennsylvania State University
The Graduate School

DESIGN CHALLENGES ON ENTERPRISE-SCALE STORAGE SYSTEMS EMPLOYING HARD DRIVES AND NAND FLASH BASED SOLID-STATE DRIVES

A Dissertation in
Computer Science and Engineering
by
Youngjae Kim

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Doctor of Philosophy

December 2009

The dissertation of Youngjae Kim was reviewed and approved* by the following:

Anand Sivasubramaniam
Professor of Computer Science and Engineering
Dissertation Co-Advisor, Co-Chair of Committee

Bhuvan Urgaonkar
Assistant Professor of Computer Science and Engineering
Dissertation Co-Advisor, Co-Chair of Committee

Prasenjit Mitra
Assistant Professor of Computer Science and Engineering

Qian Wang
Associate Professor of Mechanical and Nuclear Engineering

Raj Acharya
Professor of Computer Science and Engineering
Head of the Department of Computer Science and Engineering

*Signatures are on file in the Graduate School.

Abstract

Flash memory overcomes some key shortcomings of hard disk drives (HDDs), including faster access to non-sequential data and lower power consumption. Economic forces, driven by the desire to introduce flash into the enterprise market without changing existing software based, have resulted in the emergence of solid-state drives (SSDs), flash packaged in HDD from factors and capable of working with device drivers and I/O buses designed for HDDs. Unlike the use of DRAM for caching or buffering, however, certain idiosyncrasies of SSDs make their integration into HDD-based systems non-trivial. Flash memory suffers from limits on its reliability, in an order of magnitude more expensive than the disk, and can be sometimes even slower than the HDD (due to excessive GC induced by high intensity of random writes). Given the complementary properties of HDDs and SDDs in terms of cost, performance, and lifetime, the current consensus among several storage experts is to view SSDs not as a replacement for HDD but rather as a complementary device within the storage hierarchy.

In my dissertation, I designed and evaluated such a hybrid system called HybridStore to provide (a) improved capacity planning techniques to administrators with the overall goal of operating within cost-budgets and (b) improved performance/lifetime guarantees during episodes of deviations from expected workloads through several novel mechanisms such as fragmentation busting and write-regulation. As an illustrative example of HybridStore's efficacy, a combination of 1 SSD and 6 low-speed, cheaper and higher capacity HDDs is recommended the most cost-effective storage configuration in HybridStore for a predominantly random-write dominant I/O trace from an OLTP application running at a large financial institution. Also, HybridStore employing HDD with small SSD is able to reduce the average response time for Financial trace by about 71% as compared to a HDD-based system.

In addition to HybridStore project, I developed a novel design technique of

the Flash Translation Layer (FTL) in the SSD. It provides improved performance, reduced garbage collection overhead, and better overloaded behavior compared to state-of- the art FTL schemes. For example, the Financial trace shows a 78% improvement in average response time (due to a 3-fold reduction in operations of the garbage collector), compared to a state-of-the-art FTL scheme. Finally, I also developed and validated flash simulation framework call FlashSim. While a number of well-regarded simulation environments exist for HDDs, the same is not yet true for SSDs. This is due to SSDs having been in the storage market for relatively less time as well as the lack of information (hardware configuration and software methods) about state-of-the-art SSDs that is publicly available. FlashSim aimed at filling this void in performance evaluation of emerging storage systems that employ SSDs.

Table of Contents

List of Figures

List of Tables

Acknowledgments

I never could have written this dissertation without wonderful supports from many people. Most of all, I cannot fully express my gratitude to my advisors, Dr. Anand Sivasubramaniam and Dr. Bhuvan Urgaonkar for his immeasurable supports, superb guidance and excellent advises. They both have always given me inspiration and encouragement to finish my studies at Penn State. I also would like to thank my dissertation committee members and referees: Dr. Prasenjit Mitra and Dr. Qian Wang for allocating their time, being a part of my dissertation committee. Their strong support and valuable comments have made helped refine my dissertation. I also would like to acknowledge specially Dr. Piotr Berman for his advises and suggestions on my research. My gratitude is also to Prof. Sudhanva Gurumurthi at University of Virginia for supporting the project of thermal management at storage systems and encouraging me.

I have been privileged to be part of CSL members, who were also my friends at Penn State, Sudhanva Gurumurthi, Murali Vilayannur, Chun Liu, Jianyong Zhang, Partho Nath, Angshuman Parashar, Shiva Chaitanya, Byung Chul Tak, Niranjan Kumar Soundararajan, Sriram Govindan, Aayush Gupta, Jeonghwan Choi, Euiseong Seo, Arjun R. Nath, Kanishk Jain, Dharani Sankar Vijayakumar, Srinath Sridharan and Amitayu Das. Thanks to all of them for being supportive and helpful.

And finally, I would like to give all my love to the family, especially my parents and parents-in-law from bottom of my heart for their immense love and supports. To my loving wife, Yu Jung, without your love and dedication I would not have opportunity to finish the dissertation.

Chapter 1

Introduction

Hard disk drives (HDDs) have been the preferred media for data storage in enterprise-scale storage systems for several decades. Manufacturers of HDDs have been successful in ensuring sustained performance improvements while substantially bringing down the price-per-byte. However, there are several shortcomings inherent to HDDs that are becoming harder to overcome as we move into faster and denser design regimes. First, designers of HDDs are finding it increasingly difficult to further improve the rotational speeds (RPM) due to problems of dealing with the resulting increase in power consumption and temperature. Second, any further improvement in storage density is increasingly harder to achieve and requires significant technological breakthroughs such as perpendicular recording. Third, and perhaps most serious, despite a variety of techniques employing caching, pre-fetching, scheduling, write-buffering, and those based on improving parallelism via replication (e.g., RAID), the mechanical movement involved in the operation of HDDs can severely limit the performance that hard disk based systems are able to offer to workloads with significant randomness and/or lack of locality.

Alongside improvements in HDD technology, significant advances have also been made in various forms of solid-state memory such as NAND flash, magnetic RAM (MRAM), phase-change memory (PRAM), and Ferroelectric RAM (FRAM). Solid-state memory offers several advantages over hard disks: lower access latencies for random requests, lower power consumption, lack of noise, and higher robustness to vibrations and temperature. In particular, recent improvements in the design and performance of NAND flash memory (simply *flash* henceforth) have resulted

in its becoming popular in many embedded and consumer devices. Flash has, however, only seen limited success in the enterprise-scale storage market. Although (i) the aforementioned advances in flash technology and (ii) its dropping cost-per-byte had led several storage experts to predict the inevitable demise of HDDs, flash has so far not been able to make inroads into the enterprise-scale storage market to the extent expected.

Flash technology possesses a number of idiosyncrasies that have hindered the SSD from replacing HDD in the general enterprise market. First, there still exists a huge gap between the Cost/GB of HDDs and SSDs. Second, unlike HDD or DRAM, SSDs possess a asymmetry between the speeds at which reads and writes may be performed. As a result, the throughput a SSD offers a write-dominant workload is lower than for a read-dominant workload. Third, flash technology restricts the locations on which writes may be performed—a flash location must be *erased* before it can be written—leading to the need for a garbage collector (GC) for/within an SSD. Certain workload characteristics (in particular, the presence of randomness in writes) increase the fragmentation of data stored in flash, i.e., logically consecutive sectors become spread over physically non-consecutive blocks on flash. This exacerbates GC overheads, thereby significantly slowing down the SSD—even to an extent where it operates slower than a HDD! Furthermore, this slowdown is non-trivial to anticipate. Finally, to further complicate matters, unlike HDDs, SSDs have a life-time that is limited by the number of erases performed. Therefore, excessive writing to flash, while potentially useful for the overall performance of a flash-based storage system, limits its lifetime. This becomes an important concern in an enterprise-scale employing flash if its workload is write-intensive.

SSDs should be fairly complex devices. Their peculiar properties related to cost, performance, and lifetime make it difficult for a storage system designer to neatly fit them between HDD and DRAM. Given the complementary properties of HDDs and SSDs in terms of cost, performance, and lifetime, the current consensus among several storage experts is to view SSDs not as a replacement for HDD but rather as a complementary device within the storage hierarchy. I propose HybridStore, a hybrid storage system containing HDDs and SSDs. Besides this hardware, HybridStore comprises: (i) a *capacity planner* that makes long-term

resource provisioning decisions for the expected workload; it is designed to optimize the cost of equipment that needs to be procured to meet desired performance and lifetime needs for the expected workload and (ii) a *dynamic controller* whose goal is to operate the system in desirable performance/lifetime regimes in the face of deviations at short time-scales in workload. I develop simple statistical models that the capacity planner employs. These models are used in conjunction with *HybridSim* (a simulator I have developed for HybridStore by enhancing DiskSim to validate the efficacy of the capacity planner for a variety of well-regarded real-world storage traces). I implement the dynamic controller in our simulator. I enhance block device driver that employs online statistical performance and lifetime models for SSD (and a performance model for HDD) to dynamically partition incoming workload among the SSD and HDD, and two algorithms within the SSD controller (specifically, within the FTL layer) including reduction of fragmentation within the flash and a novel concept of *adaptive wear-leveling*.

Out of many concerns on employing flash in enterprise-scale storage systems, my next research focuses on improving random write performance of flash. Recent research solved such random write problems on flash by adding DRAM-backed buffers or buffering requests to increase their sequentiality. However, I focus on an intrinsic component of the flash, namely the *Flash Translation Layer (FTL)*, (which is the main cause of fragmentation on flash and hence slows down its performance) to provide a solution for performance problems of the flash. The small granularity page based FTLs have been known to be the best write performance on flash but, require large *on-flash SRAM-based cache* where it stores its mapping table. Thus, I propose and design a novel FTL, which is purely page-mapped. The idea behind this novel FTL scheme is simple: since most enterprise-scale workloads exhibit significant temporal locality, DFTL uses the on-flash limited SRAM to store the most popular mappings while the rest are maintained on the flash device itself. I implement a flash simulator called *FlashSim* [1] to evaluate the efficacy of DFTL and compare it with other FTLs. FlashSim is an open source and has been built by enhancing the popular Disksim 3.0 simulator.

[1] `http://csl.cse.psu.edu/hybridstore`

1.1 Read-map

This thesis aims to address challenges in a hybrid storage system employing HDDs and NAND Flash based SSDs. The subsequent chapters of this thesis are organized as follows. Chapter 2 describes background for NAND flash based Solid-state Disk Drive including NAND flash memory technology and their related work. In Chapter 3, based on observation that NAND flash performance becomes worse for small random write dominant workload patterns, a novel FTL scheme is proposed and evaluated. Also, I present a NAND flash based SSD simulator, called *Flsah-Sim* and validates it against real commercial SSDs for behavioral similarity and objected-oriented approach for the simulator design is further discussed. Chapter 4 presents an overview of HybridStore, a hybrid storage system employing multiple HDDs and SSDs and a long-term capacity planner in HybridStore is proposed and evaluated. Chapter 5 evaluates how long-term capacity planner and short-term dynamic controller could act in concert and present evaluation of all components of HybridStore. Finally, Chapter 6 concludes the dissertation with a summary and suggestions for future work.

Chapter 2

Background and Related Work

2.1 Basics of Flash Memory Technology

Recently, significant advances have been made in various forms of solid-state memory such as NAND flash, magnetic RAM (MRAM) [3], phase-change memory (PRAM) [4], and FeRAM [5]. In particular, improvements in the design and performance of NAND flash memory (simply flash henceforth) have resulted in it being employed in many embedded and consumer devices. Small form-factor hard disks have already been replaced by flash memory in some consumer devices, like music players. More recently, flash drives with capacities in the 32-64 GB range have become available and have been used in certain laptops as the secondary storage media [6].

Flash Operations - Flash is a unique storage device since unlike the hard disk drive and volatile memories, which provide read and write operations, it also provides an *erase operation* [7]. Flash provides three basic operations: (i) program or write, (ii) read, and (iii) erase. Salient operational characteristics of these operations are as follows [7]. The write operation changes the value of a bit in a flash memory cell from 1 to 0. The erase operation changes a bit from 0 to 1. Single bit erase operations are not typically supported. Erase operations are performed at the granularity of a *block* (a set of contiguous bits) by changing all the bits of the block to 1. Erase is the slowest operation while write is slower than read. The life-time of flash memory is limited by the number of erase operations.

Flash Type	Data Unit Size			Access Time		
	Page (Bytes)		Block	Page	Page	Block
	Data	OOB	(Bytes)	Read (us)	Write (us)	Erase (ms)
Small Block	512	16	(16K+512)	41.75	226.75	2
Large Block	2048	64	(128K+4K)	130.9	405.9	2

Table 2.1. NAND Flash organization and access time comparison for Small-Block vs. Large-Block schemes [1].

It has been reported that each flash memory cell can sustain about 10K-1M erase operations [8]. Moreover, flash memory can be composed of two types of memory cells: Single-Level-Cell (SLC) which stores one bit per cell and Multi-Level-Cell (MLC), introduced by Intel, which stores multiple bits of data per memory cell. However, improving the density of flash memory using MLC has been found to deteriorate its lifetime and performance [9]. In our research, we focus on SLC based flash memory.

An erase unit is composed of multiple *pages*. A page is the granularity at which reads and writes are performed. In addition to its data area, a page contains a small spare Out-of-Band area (OOB) which is used for storing a variety of information including: (i) Error Correction Code (ECC) information used to check data correctness, (ii) the logical page number corresponding to the data stored in the data area and (iii) page state. Each page on flash can be in one of three different states: (i) *valid*, (ii) *invalid* and (iii) *free/erased*. When no data has been written to a page, it is in the erased state. A write can be done only to an erased page, changing its state to valid. When data is written to an erased page, its state becomes valid. If the page contains an older version of data, it is said to be in the invalid state. As was pointed out, out-of-place updates result in certain written pages whose entries are no longer valid. They are called invalid pages.

Flash Types - Flash comes as a *small block* or *large block* device. Using fewer blocks not only improves read, write, and erase performance, but also reduces chip size by reducing gaps between blocks [1]. A small block scheme can have 8KB or 16KB blocks where each page contains 512B data area and 16B OOB. On the contrary, large block schemes have 32KB to 128KB blocks where each page contains 2KB data area and 64B OOB. Table 2.1 shows detailed organization and perfor-

mance characteristics for these two variants of state-of-the-art flash devices [1].

2.2 Characteristics of Flash Memory Operations

Flash memory includes the following operational characteristics: Basically the read and write speed of flash memory is asymmetric. Not only are erase operations done at the coarser granularity of a block and are significantly slower that reads/writes, there is an additional asymmetry between access times of reads and writes. As shown in Table 2.1, erase operations are significantly slower than reads/writes. Additionally, write latency can be higher than read latency by up to a factor of 4-5. This is because draining electrons from a flash cell for a write takes longer than sensing them for a read. Note that this is significantly different from hard disk and volatile memory.

In-place vs. Out-of-place updates - In flash memory, in-place update operations are very costly. Since an erase occurs at the block granularity whereas writes are done to pages, an in-place update to a page entails (i) reading all valid pages of the block into a buffer, (ii) updating the required page, (iii) erasing the entire block and (iv) then writing back all the valid pages to the block. Instead, faster out-of-place updates are employed that work as follows: An out-of-place update invalidates the current version of the page being updated and writes the new version to a free page. This introduces the need to keep track of the current page version location on flash itself, which is maintained by implementing an address translation layer (FTL). The OOB area of invalid pages are marked to indicate their changed states.

Out-of-place updates result in the creation of invalid pages on flash. A garbage collector is employed to reclaim invalid pages and create new erased blocks. It first selects a victim block based on a policy such as choosing a block with maximum invalid pages. All valid data within the block is first copied into an erased block. This data-rewrite operation can be quickly and efficiently processed by the special support of a *Copy-Back Program* operation where an entire page is moved into the internal data buffer first and then written [10]. Then the victim block is erased. The efficiency of garbage collector is one of the dominant factors affecting flash

memory performance.

Lifetime and Wear-leveling - The lifetime of flash memory is limited by the number of erase operations on its cells. Each memory cell typically has a lifetime of 10K-1M erase operations [8]. Thus, *wear-leveling* techniques [11, 12, 13] are used to delay the wear-out of the first flash block. They make flash memory last longer by evenly distributing the wear-out over all blocks. Since the MLC has much smaller voltage tolerance than SLC, each write-erase cycle tends to increase the variance in the voltage stored. Thus, the lifetime of MLC is more limited than that of SLC [9] whereas data density is more in MLC and benefited in cost by around two times more than SLC [9]. and it degrades the performance will need to differentiate data more precisely. The granularity at which wear-leveling is carried out impacts the variance in the lifetime of individual blocks and also the performance of flash. The finer the granularity, the smaller the variance in lifetime. However, it may impose certain performance overheads. Thus, this trade-off needs to be balanced to obtain optimal performance from the device while providing the minimal desired lifetime. The optimal selection for the victim block considers the copy overhead of valid pages and the wear-level of the block for wear-leveling.

2.3 Flash Memory based Solid State Disk Drive

A solid state disk-drive (SSD) can be composed of non-volatile memory such as battery-backed DRAM (DRAM-Based SSDs) or flash memory chips (Flash-based SSDs). There are also hybrid devices incorporating DRAM and flash memory [14]. RamSan-500, a cached flash SSD from Texas Memory Systems (TMS) is a hybrid of DDR RAM and NAND-SLC flash memory [15]. Symmetrix DMX-4 from EMC [14] is an enterprise networked storage system employing the flash drives. Since NAND flash memory based SSD has recently become popular, in this work we only concentrate on NAND flash memory based SSD.

2.3.1 NAND Flash based SSDs

Figure 2.1 describes the organization of internal components in a flash-based SSD from Mtron [16]. It possesses a host interface (such as Fiber-Channel, SATA,

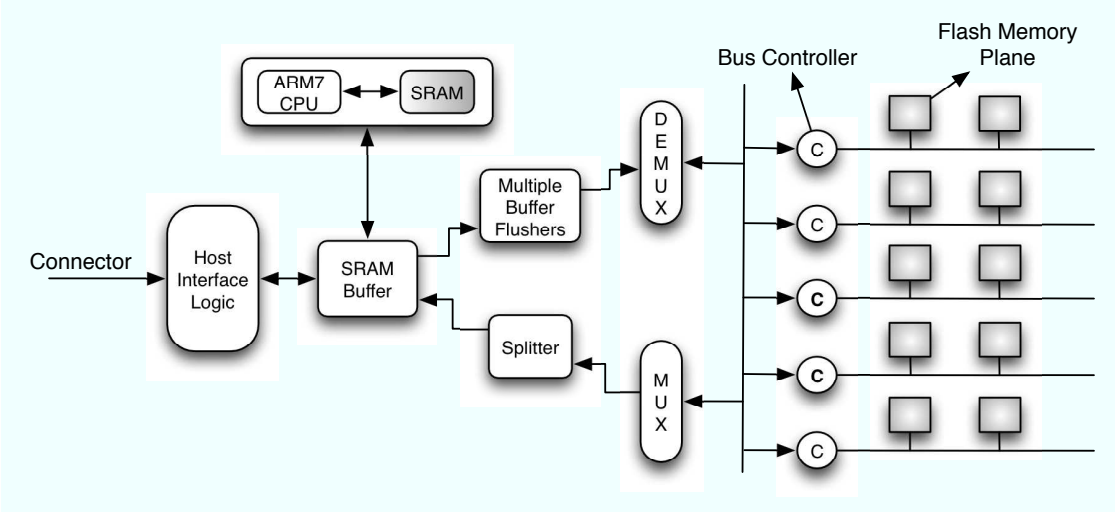


Figure 2.1. A Block Diagram of Flash based Solid State Disk Drive.

PATA, and SCSI etc.) to appear as block I/O device to the host computer. The main controller is composed of two units - processing unit (such as ARM7 processor) and fast access memory (such as SRAM). The virtual-to-physical mappings are processed by the processor and the data-structures related to the mapping table are stored in SRAM in the main controller. The software module related to this mapping process is called Flash Translation Layer (FTL). A part of SRAM can be also used for caching data.

A storage pool in a SSD is composed of multiple flash memory *Planes*. The *Planes* are implemented in multiple *Dies*. For example, Samsung 4GB flash memory has two *Dies*. A *Die* is composed of four planes, each of size is 512MB [10]. A *Plane* consists of a set of blocks. The block size can be 64KB, 128KB, 256KB etc. depending on the memory manufacturer. The SSD can be implemented multiple *Planes*. SSD performance can be enhanced by interleaving requests across the planes, which is achieved by the multiplexer and de-multiplexer between SRAM buffer and flash memories [10]. In our research, we deal with a simplistic, SLC flash device model with block size as 128KB.

2.3.2 Details of Flash Translation Layer

The mapping tables and other data structures, manipulated by the FTL are stored in a small, fast SRAM. The FTL algorithms are executed on it. FTL helps in

emulating flash as a normal block device by performing out-of-place updates which in turn helps to hide the erase operations in flash. It can be implemented at different address translation granularities. At two extremes are page-level and block-level translation schemes which we discuss next. As has been stated, we begin by understanding two extremes of FTL designs with regard to what they store in their in-SRAM mapping table. Although neither is used in practice, these will help us understand the implications of various FTL design choices on performance.

Page-level FTL Schemes - As shown in Figure 2.2, in a page-level FTL scheme, the logical page number of the request sent to the device from the upper layers such as file system can be mapped into any page within the flash. This should remind the reader of a fully associative cache [17]. Thus, it provides compact and efficient utilization of blocks within the flash device. However, on the downside, such translation requires a large mapping table to be stored in SRAM. For example, a 16GB flash memory requires approximately 32MB of SRAM space for storing a page-level mapping table. Given the order of magnitude difference in the price/byte of SRAM and flash; having large SRAMs which scale with increasing flash size is infeasible.

Block-level FTL Schemes - At the other extreme, in a block-level FTL scheme, as depicted in Figure 2.3, page offset within a block is fixed. The logical block number is translated into a physical block number using the mapping table similar to set-associative cache design [17]. The logical page number offset within the block is fixed. Figure 2.3 shows an example of block-based address translation. The Logical Page Number (LPN) is converted into a Logical Block Number(LBN) and offset. The LBN is then converted to Physical Block Number (PBN) using the block based mapping table. Thus, the offset within the block is invariant to address translation. The size of the mapping table is reduced by a factor of *block size/page size* (128KB/2KB=64) as compared to page-level FTL. However, it provides less flexibility as compared to the page-based scheme. Even if there are free pages within a block except at the required offset, this scheme may require allocation of another free block; thus reducing the efficiency of block utilization. However, since a given logical page may now be placed in only a particular physical

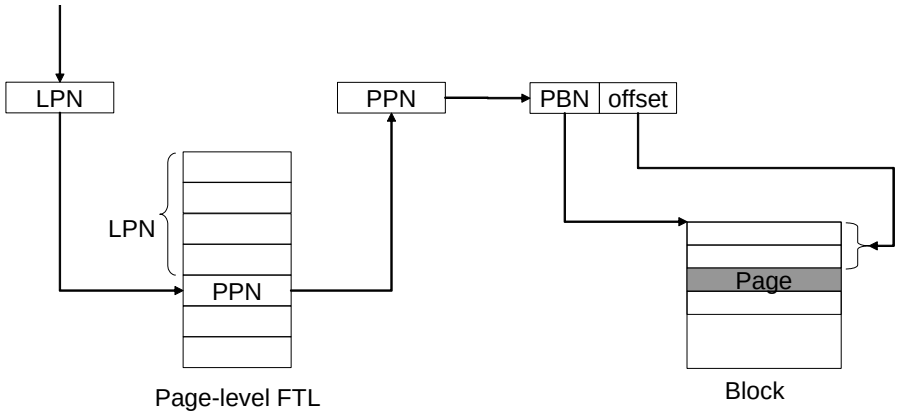


Figure 2.2. Page-level FTL scheme. LPN: Logical Page Number, PPN: Physical Page Number, LBN: Logical Block Number, PBN: Physical Block Number.

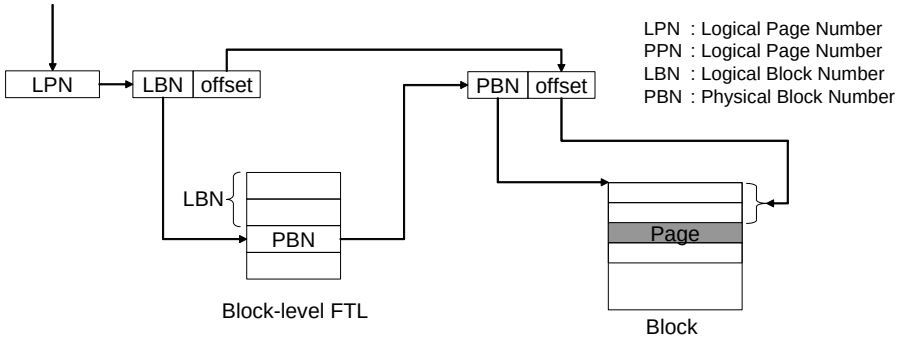


Figure 2.3. Block-level FTL scheme. LPN: Logical Page Number, PPN: Physical Page Number, LBN: Logical Block Number, PBN: Physical Block Number.

page within each block, the possibility of finding such a page decreases. As a result the garbage collection overheads grow. Moreover, the specification for large block based flash devices requiring sequential programming within the block [1] making this scheme infeasible to implement in such devices. For example, Replacement Block-scheme [18] is a block-based FTL scheme in which each data block is allocated replacement blocks to store the updates. The chain of replacement blocks along with the original data block are later merged during garbage collection.

A Generic Description of Hybrid FTL Scheme - To address the shortcomings of the above two extreme mapping schemes, researchers have come up with a variety of alternatives. Log-buffer based FTL scheme is a hybrid FTL which combines a block-based FTL with a page-based FTL as shown in Figure

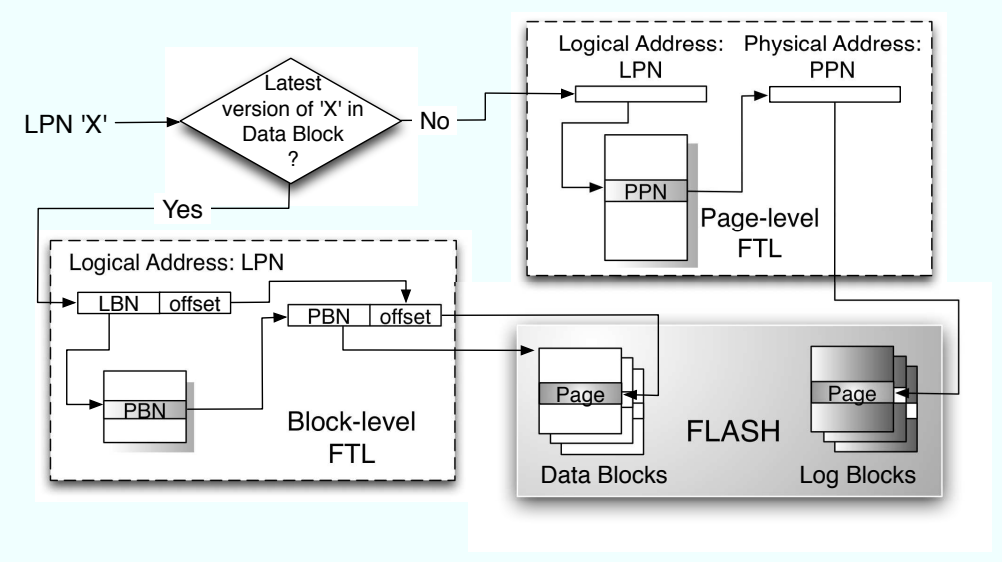


Figure 2.4. Hybrid FTL Scheme, combining a block-based FTL for data blocks with a page-based FTL for log blocks. LPN: Logical Page Number, PPN: Physical Page Number, LBN: Logical Block Number, PBN: Physical Block Number.

2.4. The entire flash memory is partitioned into two types of blocks - *Data* and *Log/Update* blocks. First write to a logical address is done in data blocks. Although many schemes have been proposed [19, 20, 21, 22, 23], they share one fundamental design principle. All of these schemes are a *hybrid* between page-level and block-level schemes. They logically partition their blocks into two groups - *Data Blocks* and *Log/Update Blocks*. Data blocks form the majority and are mapped using the block-level mapping scheme. A second special type of blocks are called log blocks whose pages are mapped using a page-level mapping style. Figure 2.4 illustrates such hybrid FTLs. Any update on the data blocks are performed by writes to the log blocks. The log-buffer region is generally kept small in size (for example, 3% of total flash size [23]) to accommodate the page-based mappings in the small SRAM. Extensive research has been done in optimizing log-buffer based FTL schemes [19, 20, 21, 22, 23].

2.3.3 Garbage Collection in Hybrid FTLs

The hybrid FTLs invoke a garbage collector whenever no free *log blocks* are available. Garbage Collection requires merging log blocks with data blocks. The merge

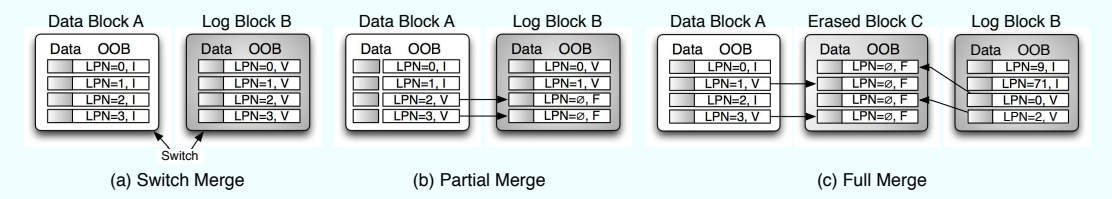


Figure 2.5. Various *Merge* operations (*Switch*, *Partial*, and *Full*) in log-buffer based FTL schemes. V: Valid, I: Invalid, and F: Free/Erased and LPN is Logical Page Number.

operations can be classified into: *Switch merge*, *Partial merge*, and *Full merge*. In Figure 2.5(a), since log block B contains all valid, sequentially written pages corresponding to data block A, a simple *Switch Merge* is performed, whereby log block B becomes new data block and the old data block A is erased. Figure 2.5(b) illustrates *Partial Merge* between block A and B where only the valid pages in data block A are copied to log block B and the original data block A is erased changing the block B's status to a data block. *Full Merge* involves the largest overhead among the three types of merges. As shown in Figure 2.5(c), Log block B is selected as the victim block by the garbage collector. The valid pages from the log block B and its corresponding data block A are then copied into a new erased block C and block A and B are erased.

Expensive Full Merge Operation Full merge can become a long recursive operation in case of a fully-associative log block scheme where the victim log block has pages corresponding to multiple data blocks and each of these data blocks have updated pages in multiple log blocks. This situation is illustrated in Figure 2.6.

Log block L1 containing randomly written data is selected as a victim block for garbage collection. It contains valid pages belonging to data blocks D1, D2 and D3. An erased block is selected from the free block pool and the valid pages belonging to D1 are copied to it from different log blocks and D1 itself in the order shown. The other pages for D1 are copied similarly from log block L2 and L3. The valid page in D1 itself is then copied into the new data block. The data block D1 is then erased. Similar operations are carried out for data blocks D2 & D3 since L1 contains the latest version of some of the pages for these blocks. Finally, log block L1 is erased. This clearly illustrates the large overhead induced by full merge operations. Thus, random writes in hybrid FTLs induce costly garbage collection

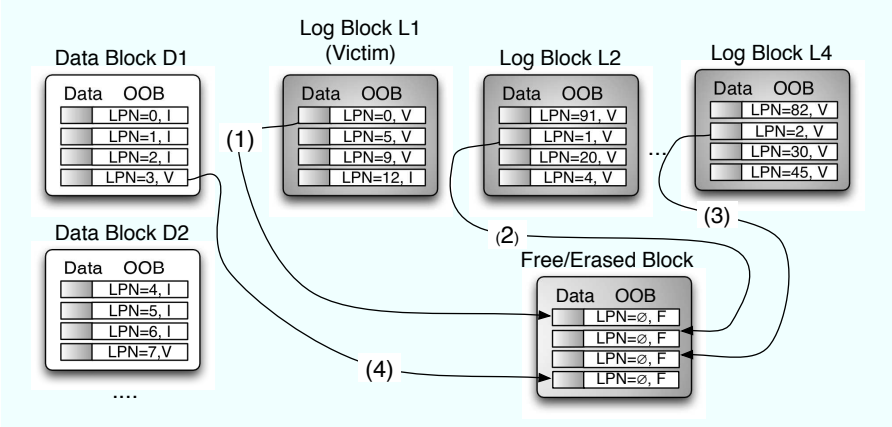


Figure 2.6. Expensive Full Merge.

which in turn affects performance of subsequent operations irrespective of whether they are sequential or random. Recent log buffer-based FTL schemes [22, 23] have tried to reduce the number of these full merge operations by segregating log blocks based on access patterns. Hot blocks with frequently accessed data generally contain large number of invalid pages whereas cold blocks have least accessed data. Utilizing hot blocks for garbage collection reduces the valid page copying overhead, thus lowering the full merge cost.

2.4 Related Work

State-of-the-art FTLs - Different FTL schemes have been proposed to improve flash device performance [24, 25]. State-of-the-art FTLs [20, 21, 22, 23] are based on hybrid log-buffer based approaches. They try to address the problems of expensive full merges, which are inherent to any log-buffer based hybrid scheme, in their own unique way. However, all of these attempts are unable to provide the desired results. As different from block-level FTL scheme [18], a log-based FTL scheme which is a hybrid approach have been exploited where BAST [20] is the first work. Block Associative Sector Translation (BAST) [20] scheme exclusively associates a log block with a data block. In presence of small random writes, this scheme suffers from *log block thrashing* [21] that results in increased full merge cost due to inefficiently utilized log blocks. Fully Associative Sector Translation (FAST) [21] allows log blocks to be shared by all data blocks. This improves the

utilization of log blocks as compared to BAST. FAST keeps a single sequential log block dedicated for sequential updates while other log blocks are used for performing random writes. Thus, it cannot accommodate multiple sequential streams. Further, it does not provide any special mechanism to handle temporal locality in random streams. SuperBlock FTL [22] scheme utilizes existence of *block level* spatial locality in workloads by combining consecutive logical blocks into a superblock. It maintains page-level mappings within the superblock to exploit temporal locality in the request streams by separating hot and cold data within the superblock. However, the three-level address translation mechanism employed by this scheme causes multiple OOB area reads and writes for servicing the requests. More importantly, it utilizes a fixed superblock size which needs to be explicitly tuned to adapt to changing workload requirements. The recent Locality-Aware Sector Translation (LAST) scheme [23] tries to alleviate the shortcomings of FAST by providing multiple sequential log blocks to exploit spatial locality in workloads. It further separates random log blocks into hot and cold regions to reduce full merge cost. In order to provide this dynamic separation, LAST depends on an external locality detection mechanism. However, Lee et al. [23] themselves realize that the proposed locality detector cannot efficiently identify sequential writes when the small-sized write has a sequential locality. Moreover, maintaining sequential log blocks using a block-based mapping table requires the sequential streams to be aligned with the starting page offset of the log block in order to perform switch-merge. Dynamically changing request streams may impose severe restrictions on the utility of this scheme to efficiently adapt to the workload patterns.

Flash as Cache and Write-Buffer - A lot of research has been conducted to improve performance of HDDs using non-volatile memory. eNVy [26] uses non-volatile memory for data storage wherein battery-backed SRAM is used to reduce the write overhead. HeRMES [27] uses magnetic RAM to reduce the overhead of frequently and randomly accessing meta-data. MEMS [28] has also been exploited to improve disk performance. Finally, storage architecture in which flash memory is used as a conventional disk cache has already been explored in [29] Our work goes beyond merely using flash as a cache/write-buffer—rather than treating flash as a *subordinate to the disk*, HybridStore views these as *complementary* storage

media.

Hybrid Disks - Samsung and Microsoft [30] have developed and deployed hybrid hard disks for laptops (where NAND flash is located at an upper level in the storage hierarchy as compared to hard disk). Booting time and resuming process from disk have been improved by overlapping the time for spinning up disk drive with the booting process from flash memory. Bisson et al. [31] have explored the use of a flash-based NVRAM as a write buffer to reduce write latency of hard disks for desktop environments. They employ I/O redirection to reduce seeking overhead from disk by directing requests likely to incur long seeks to the on-disk NVRAM. We view the MixDyn component of our system as conceptually close to Bisson et al.'s work and would be interested in comparing MixDyn with their I/O redirection technique in the future. To the best of our knowledge, this work is the closest to the MixDyn component of our system which will be described in Chapter 5. However, their model fails to effectively capture the intricacies of flash and thus is susceptible to poor performance induced by fragmentation caused by random writes. Our approach to modeling hybrid system, considers the performance variation of flash devices along with varying workload characteristics. A key difference is that our flash model additionally captures the fragmentation within flash (caused by random writes) and incorporates it into its redirection decision-making. This mechanism will be described in Chapter 5.

Flash-specific Improvements - Kim et al. [32] have developed a flash device buffer management scheme to reduce fragmentation caused by random writes. Park et al. [33] and Jo et al. [34] have proposed novel buffer cache management schemes to resolve this fragmentation issue due to random writes in SSDs. Kim et al [35] also developed a *FlashLite*, a user-level library to enhance durability of SSD for P2P file sharing by converting random writes to sequential writes. Different SSD designs including interleaving requests to obtain parallelism and ganging etc. have been proposed to improve flash device performance [10]. Further, Managed Flash Technology (MFT) [36] developed by EasyCo is a flash SSD acceleration software which tries to solve flash random write problem by converting random writes into sequential writes at block driver level. Transactional flash (named *TxFlash*)

recently proposed by Prabhkaran et al. is a novel SSD that uses flash memory and exports a transactional interface to the higher-level software [37]. They use the copy-on-write nature of the FTL on flash for supporting transaction operations in the SSD. Another orthogonal approach of exposing flash-based devices to the file system has been proposed. JFFS2 [38] and YAFFS2 [39] are the most popular file systems optimized for flash memories.

Flash in the Enterprise - Kgil et al. [40] propose a new architecture named *FlashCache* where they consider replacing 1GB DRAM with a combination of a smaller 256MB DRAM and 1GB NAND-based Flash. Their goal is to save memory power consumption while meeting performance requirement by using larger flash and a smaller DRAM. Sun Micro-systems has proposed a storage architecture incorporating flash-based SSDs as intent-log devices and read caches providing improved performance along with reduced power consumption [41]. They propose to use their ZFS file system [42] as an interface to these SSDs. We view Sun's proposed hybrid architecture as the closest in essence to HybridStore and believe that the models and techniques developed here are worth implementing and evaluating in the context of their system. Lee et al., [43] proposed an in-page logging approach in a flash-based DBMS to reduce random write overhead by updating in-place in the database buffer and hence reducing garbage collection overhead. A key contribution in this paper is the observation that workloads with extensive randomness can cause an SSD to perform worse than a HDD. We find similar results in our evaluation and build models that attempt to capture this aspect of an SSD's operation. Finally, Narayanan et al. [44] have also looked at the capacity provisioning in hybrid storage systems. Their work is complementary to our MixPlan and tries to establish the right balance of flash and disk capacity needed for different enterprise scale workloads. They utilize a large number of real data center traces in their work and it would help us in our validation/evaluation if we can get access to those traces. However, our work goes beyond provisioning to actual short-term dynamic control using MixDyn to handle the various idiosyncrasies of flash as well as to ensure that the lifetime and performance guarantees made by MixPlan are upheld.

Chapter 3

Improvements in Solid-state Disk Drive

3.1 Introduction

The FTL is one of the core engines in flash-based SSDs that maintains a mapping table of virtual addresses from upper layers (e.g., those coming from file systems) to physical addresses on the flash. It helps to emulate the functionality of a normal block device by exposing only read/write operations to the upper software layers and by hiding the presence of *erase* operations, something unique to flash-based systems. Flash-based systems possess an asymmetry in how they can read and write. While a flash device can read any of its *pages* (a unit of read/write), it may only write to one that is in a special state called *erased*. Flashes are designed to allow erases at a much coarser spatial granularity than pages since page-level erases are extremely costly. As a typical example, a 16GB flash product from Micron [45] (MT29F16G08MAAWP [1]) has 2KB pages while the erase blocks are 128KB [45]. This results in an important idiosyncrasy of updates in flash. Clearly, in-place updates would require an erase-per-update, causing performance to degrade. To get around this, FTLs implement *out-of-place updates*. An out-of-place update: (i) chooses an already erased page, (ii) writes to it, (iii) invalidates the previous version of the page in question, and (iv) updates its mapping table to reflect this change. These out-of-place updates bring about the need for the FTL to employ

a garbage collection (GC) mechanism. The role of the GC is to reclaim invalid pages within blocks by erasing the blocks (and if needed relocating any valid pages within them to new locations). Evidently, FTL crucially affects flash performance.

One of the main difficulties the FTL faces in ensuring high performance is the severely constrained size of the *on-flash SRAM-based cache* where it stores its mapping table. For example, a 16GB flash device requires at least 32MB SRAM to be able to map all its pages. With growing size of SSDs, this SRAM size is unlikely to scale proportionally due to the higher price/byte of SRAM. This prohibits FTLs from keeping virtual-to-physical address mappings for all pages on flash (page-level mapping). On the other hand, a block-level mapping, can lead to increased: (i) space wastage (due to internal fragmentation) and (ii) performance degradation (due to GC-induced overheads). Furthermore, the specification for large-block flash devices (which are the norm today) requires sequential programming within the block [1] making such coarse-grained mapping infeasible. To counter these difficulties, state-of-the-art FTLs take the middle approach of using a *hybrid* of page-level and block-level mappings and are primarily based on the following main idea (we explain the intricacies of individual FTLs in Chapter 2): most of the blocks (called Data Blocks) are mapped at the block level, while a small number of blocks called "update" blocks are mapped at the page level and are used for recording updates to pages in the data blocks.

3.2 Motivation

First, state-of-the-art hybrid FTL schemes suffer from poor garbage collection behavior. Second, they often come with a number of workload-specific tunable parameters (for optimizing performance) that may be hard to set. Finally and most importantly, they do not properly exploit the temporal locality in accesses that most enterprise-scale workloads are known to exhibit. Even the small SRAM available on flash devices can thus effectively store the mappings in use at a given time while the rest could be stored on the flash device itself. Our goal is that such a page-level FTL, based purely on exploiting such temporal locality, can outperform hybrid FTL schemes and also provide a easier-to-implement solution devoid of complicated tunable parameters.

Research Contributions - This work makes the following specific contributions:

- We propose and design a novel Flash Translation Layer called *DFTL*. Unlike currently predominant hybrid FTLs, it is purely page-mapped. The idea behind DFTL is simple: since most enterprise-scale workloads exhibit significant temporal locality, DFTL uses the on-flash limited SRAM to store the most popular (specifically, most recently used) mappings while the rest are maintained on the flash device itself. The core idea of DFTL is easily seen as inspired by the intuition behind the Translation Lookaside Buffer (TLB) [17] that stores most popular virtual-physical page translations for memory accesses to provide efficient CPU-cache management.

- Using a number of realistic enterprise-scale workloads, we demonstrate the improved performance resulting from DFTL. As illustrative examples, we observe 78% improvement in average response time for a random write-dominant I/O trace from an OLTP application running at a large financial institution and 56% improvement for the read-dominant TPC-H workload. Moreover, we see that DFTL even outperforms the ideal page-based FTL, reducing the system response time when write-back device cache is enabled.

3.3 Design of DFTL: Our Demand-based Page-mapped FTL

We have seen that any hybrid scheme, however well-designed or tuned, will suffer performance degradation due to expensive full merges that are caused by the difference in mapping granularity of data and update blocks. *Our contention is that a high-performance FTL should completely be re-designed by doing away with log-blocks.* This is exactly the key idea behind our scheme, which we describe next. Demand-based Page-mapped FTL (DFTL) is an enhanced form of the page-level FTL scheme described in Chapter 2. It does away completely with the notion of log blocks. In fact, all blocks in this scheme, can be used for servicing update requests. Page-level mappings allow requests to be serviced from any physical page

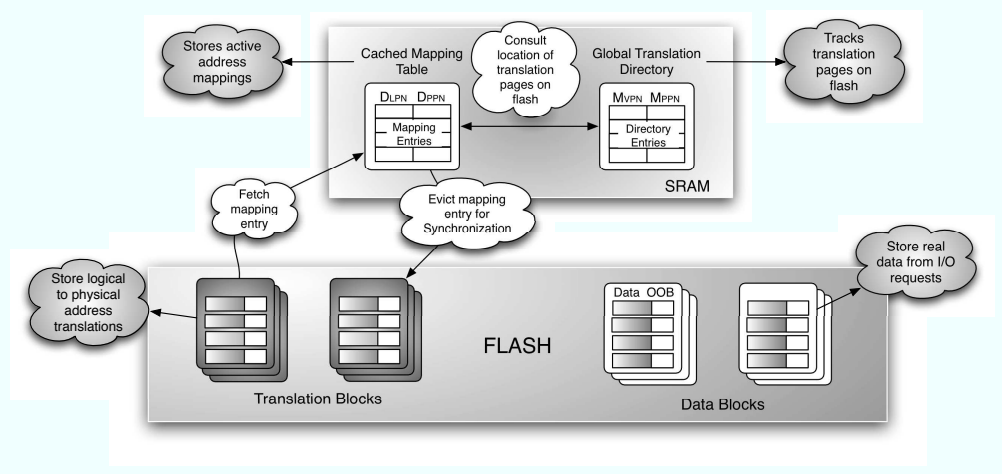


Figure 3.1. Schematic Design of DFTL. D_{LPN}: Logical Data Page Number, D_{PPN}: Physical Data Page Number, M_{VPN}: Virtual Translation Page Number, M_{PPN}: Physical Translation Page Number.

on flash. However, as we remarked earlier, the small size of on-flash SRAM does not allow all these page-level mappings to be present in SRAM. However, to make the fine-grained mapping scheme feasible with the constrained SRAM size, a *special address translation mechanism* has to be developed. In the next sub-sections, we describe the architecture and functioning of DFTL and highlight its advantages over existing state-of-the-art FTL schemes.

3.3.1 DFTL Architecture

DFTL makes use of the presence of temporal locality in workloads to judiciously utilize the small on-flash SRAM. Instead of the traditional approach of storing all the address translation entries in the SRAM, it dynamically loads and unloads the page-level mappings depending on the workload access patterns. Furthermore, it maintains the complete image of the page-based mapping table on the flash device itself. There are two options for storing the image: (i) The OOB area or (ii) the data area of the physical pages. We choose to store the mappings in the data area instead of OOB area because it enables us to group a larger number of mappings into a single page as compared to storing in the OOB area. For example, if 4 Bytes are needed to represent the physical page address in flash, then we can group 512 logically consecutive mappings in the data area of a single page whereas only

16 such mappings would fit an OOB area. Workloads exhibiting spatial locality can benefit since this storage allows pre-fetching of a large number of mappings into SRAM by reading a single page. This amortizes the cost of this additional page-read as subsequent requests are hit within the SRAM itself. Moreover, the additional space overhead incurred is negligible as compared to the total flash size. A 1GB flash device requires only about 2MB (approximately 0.2% of 1GB) space for storing all the mappings.

In order to store the address translation mappings on flash data area, we segregated *Data-Pages* and *Translation-Pages*. Data pages contain the real data which is accessed or updated during read/write operations whereas pages which only store information about logical-to-physical address mappings are called as translation pages. Blocks containing translation pages are referred to as *Translation-Blocks* and *Data-Blocks* store only data pages. It should be noted that we completely do away with log blocks. As is clear from Figure 3.1, translation blocks are totally different from log blocks and are only used to store the address mappings. They require only about 0.2% of the entire flash space and do not require any merges with data blocks.

3.3.2 Logical to Physical Address Translation

A request is serviced by reading from or writing to pages in the data blocks while the corresponding mapping updates are performed in translation blocks. In the following subsections, we describe various data structures and mechanisms required for performing address translation and discuss their impact on the overall performance of DFTL.

Global Mapping Table and Global Translation Directory - The entire logical-to-physical address translation set is always maintained on some logically fixed portion of flash and is referred to as the *Global Mapping Table*. However, only a small number of these mappings can be present in SRAM. These active mappings present in SRAM form the *Cached Mapping Table (CMT)*. Since out-of-place updates are performed on flash, translation pages get physically scattered over the entire flash memory. DFTL keeps track of all these translation pages on flash by using a *Global Translation Directory (GTD)*. Although GTD is perma-

```
Input: Request's Logical Page Number (request_lpn), Request's Size (request_size)
Output: NULL
while request_size ≠ 0 do
    if request_lpn miss in Cached Mapping Table then
        if Cached Mapping Table is full then
            /* Select entry for eviction using segmented LRU replacement algorithm */
            victim_lpn ← select_victim_entry()
            if victim_last_mod_time ≠ victim_load_time then
                /*victim_type : Translation or Data Block
                Translation_Page_victim : Physical Translation-Page Number containing
                victim entry */
                Translation_Page_victim ← consult_GTD (victim_lpn)
                victim_type ← Translation Block
                DFTL_Service_Request(victim)
            end
            erase_entry(victim_lpn)
        end
        Translation_Page_request ← consult_GTD(request_lpn)
        /* Load map entry of the request from flash into Cached Mapping Table */
        load_entry(Translation_Page_request)
    end
    request_type ← Data Block
    request_ppn ← CMT_lookup(request_lpn)
    DFTL_Service_Request(request)
    request_size- -
end
```

Algorithm 1: DFTL Address Translation

nently maintained in the SRAM, it does not pose any significant space overhead. For example, for a 1GB flash memory device, 1024 translation pages are needed (each capable of storing 512 mappings), requiring a GTD of about 4KB.

DFTL Address Translation Process - Algorithm 1 describes the process of address translation for servicing a request. If the required mapping information for the given read/write request exists in SRAM (in CMT), it is serviced directly by reading/writing the data page on flash using this mapping information. If the information is not present in SRAM then it needs to be fetched into the CMT from flash. However, depending on the state of CMT and the replacement algorithm being used, it may entail evicting entries from SRAM. We use the segmented LRU array cache algorithm [46] for replacement in our implementation. However, other algorithms such as evicting Least Frequently Used mappings can also be used.

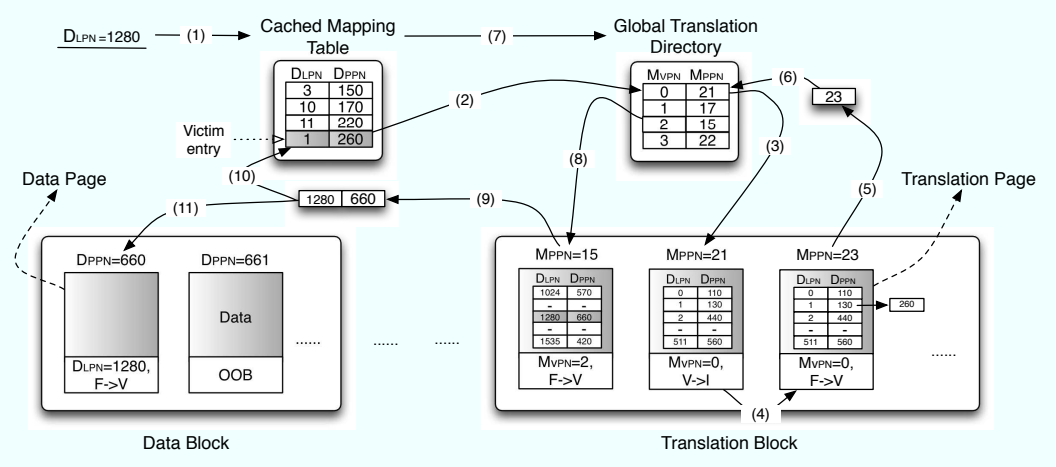


Figure 3.2. (1) Request to D_{LPN} 1280 incurs a miss in Cached Mapping Table (CMT), (2) Victim entry D_{LPN} 1 is selected, its corresponding translation page M_{PPN} 21 is located using Global Translation Directory (GTD), (3)-(4) M_{PPN} 21 is read, updated (D_{PPN} 130 $\rightarrow$ D_{PPN} 260) and written to a free translation page (M_{PPN} 23), (5)-(6) GTD is updated (M_{PPN} 21 $\rightarrow$ M_{PPN} 23) and D_{LPN} 1 entry is erased from CMT. (7)-(11) The original request's (D_{LPN} 1280) translation page is located on flash (M_{PPN} 15). The mapping entry is loaded into CMT and the request is serviced. Note that each GTD entry maps 512 logically consecutive mappings.

If the victim chosen by the replacement algorithm has not been updated since the time it was loaded into SRAM, then the mapping is simply erased without requiring any extra operations. This reduces traffic to translation pages by a significant amount in read-dominant workloads. In our experiments, approximately 97% of the evictions in read-dominant TPC-H benchmark did not incur any eviction overheads. Otherwise, the Global Translation Directory is consulted to locate the victim's corresponding translation page on flash. The page is then read, updated, and re-written to a new physical location. The corresponding GTD entry is updated to reflect the change. Now the incoming request's translation entry is located using the same procedure, read into the CMT and the requested operation is performed. The example in Figure 3.2 illustrates the process of address translation when a request incurs a CMT miss.

Overhead in DFTL Address Translation - The worst-case overhead includes two translation page reads (one for the victim chosen by the replacement algorithm and the other for the original request) and one translation page write

(for the victim) when a CMT miss occurs. However, our design choice is rooted deeply in the existence of temporal locality in workloads which helps in reducing the number of evictions. As discussed earlier, pre-fetching of mapping entries for I/O streams exhibiting spatial locality also helps to amortize this overhead. Furthermore, the presence of multiple mappings in a single translation page allows *batch updates* for the entries in the CMT, physically co-located with the victim entry. We later show through detailed experiments that the extra overhead involved with address translation is much less as compared to the benefits accrued by using a fine-grained FTL.

3.3.3 Read/Write Operation and Garbage Collection

Till now our focus was on performing address translation to locate the page to be read or updated. In this sub-section, we explain the actual data read and write operations along with the garbage collection mechanism involved. Read requests are directly serviced through flash page read operations once the address translation is completed. DFTL maintains two blocks, namely *Current Data Block* and *Current Translation Block*, where the data pages and translation pages are written, respectively. Page-based mappings allow sequential writes within these blocks, thus conforming to the large-block sequential write specification [1]. DFTL maintains pointers to the next free pages in the data and map blocks being currently written to. For write requests, DFTL allocates the next available free page in the Current Data Block, writes to it and then updates the map entry in the CMT.

However, as writes/updates propagate through the flash, over a period of time the available physical blocks (in erased state) decreases. DFTL maintains a high watermark called $GC_{threshold}$, which represents the limit till which writes are allowed to be performed without incurring any overhead of garbage collection for recycling the invalidated pages. This threshold can be adjusted with changing workload characteristics to optimize flash device performance. If it is set to a high level, the garbage collector will be invoked more often but the system will be able to maintain a high percentage of erased blocks. On the other hand, a lower setting helps to improve block utilization in the flash device while making the system operate at a resource-constrained level. Thus a delicate balance must be main-

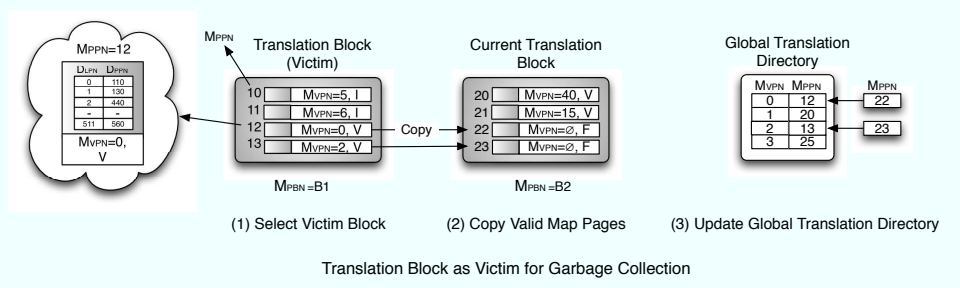


Figure 3.3. Example: (1) Translation Block (M_{PBN} B1) is selected as Victim for Garbage Collection. (2) Valid pages M_{PPN} 12 & M_{PPN} 13 are copied to the *Current Translation Block* (M_{PBN} B2) at free pages M_{PPN} 22 & M_{PPN} 23. (3) Global Translation Directory entries corresponding to M_{VPN} 0 & M_{VPN} 2 are updated (M_{PPN} 12 → M_{PPN} 22, M_{PPN} 13 → M_{PPN} 23.

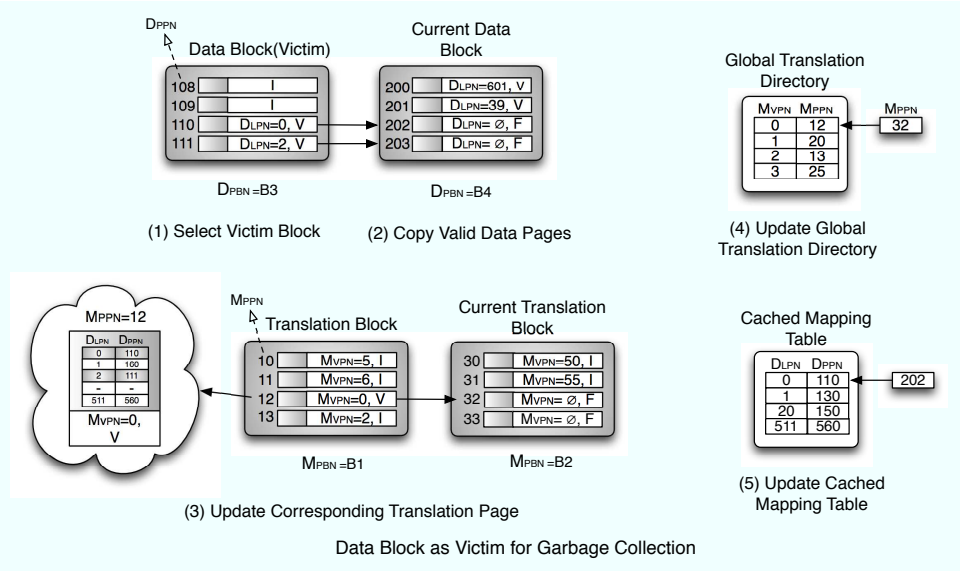


Figure 3.4. Example: (1) Data Block (D_{PBN} B3) is selected as Victim for Garbage Collection. (2) Valid pages D_{PPN} 110 & D_{PPN} 111 are copied to the *Current Data Block* (D_{PBN} B4) at free pages D_{PPN} 202 & D_{PPN} 203. (3) Translation page M_{PPN} 12 containing the mappings for the valid pages D_{PPN} 110 & D_{PPN} 111 is updated and copied to the Current Map Block (M_{PBN} B2). (4) Global Translation Directory entry corresponding to M_{VPN} 0 is updated (M_{PPN} 12 → M_{PPN} 32). (5) Since D_{LPN} 0 is present in Cached Mapping Table, the entry is also updated (D_{PPN} 110 → D_{PPN} 202). Note: We do not illustrate the advantages of batch updates and lazy copying in this example.

tained to optimize performance. This is one of the biggest advantages in DFTL as none of the other state-of-the-art hybrid FTL schemes provide this adaptability to changing workload environments. Once $GC_{threshold}$ is crossed, DFTL invokes the

garbage collector. Victim blocks are selected based on a simple cost-benefit analysis that we adapt from [12]. In this analysis, *cost* represents the overhead involved in copying valid pages from the victim block and *benefit* is the amount of free space reclaimed. These aspects of Garbage Collection are well studied and not the focus of our research. Any other garbage collection algorithm can be employed. However, we found empirically that minimizing the cost of valid page copying reduces the overall garbage collection overhead which in turn improves device performance during periods of intense I/Os by servicing the requests quicker and reducing the queuing delays in various storage sub-systems.

Different steps are followed depending on whether the victim is a translation block or a data block before returning it to the free block pool after erasing it. If it is a translation block, then we copy the valid pages to the Current Translation Block and update the GTD. However, if the victim is a data block, we copy the valid pages to the Current Data Block and update all the translation pages and CMT entries associated with these pages. In order to reduce the operational overhead, we utilize a combination of *lazy copying* and *batch updates*. Instead of updating the translation pages on flash, we only update the CMT for those data pages whose mappings are present in it. This technique of *lazy copying* helps in delaying the proliferation of updates to flash till the corresponding mappings are evicted from SRAM. Moreover, multiple valid data pages in the victim may have their virtual-to-physical address translations present in the same translation-page. By combining all these modifications into a single *batch update*, we reduce a number of redundant updates. The associated Global Translation Directory entries are also updated to reflect the changes. The examples in Figures 3.3 and 3.4 display the working of our Garbage Collector when the $GC_{threshold}$ is reached. Figure 3.3 is when a translation block is selected as victim and Figure 3.4 is when a data block is selected as victim. Algorithm 2 show the detailed description of read/write operation including garbage collection mechanism in our implementation consideration.

```
Input: NULL
Output: NULL
victim ← select_victim_entry();
/* Victim is TRANSLATION BLOCK */
if victim_type ∈ TRANSLATION_BLOCK_SET then
    foreach victim_page(i) do
        /* (i) Copy only valid pages in the victim block to the Current Translation
        Block, (ii) invalidate
        old pages, and update Global Translation Directory */
        if victim_page(i) is valid then
            curr_translation_blk ← get_curr_translation_blk();
            copy_page(victim_page(i), curr_map_blk);
            update_GTD(victim_page(i));
        end
    end
end
else
    /* Victim is DATA BLOCK */ foreach victim_page(i) do
        /* Copy only valid pages in the victim block to the
        current data block, invalidate old pages, and mark
        their corresponding translation pages for update */
        if victim_page(i) is valid then
            curr_data_blk ← get_curr_data_blk();
            copy_page(victim_page(i), curr_data_blk);
            translation_page_update_set[] ← mark_corr_translation_page_for_update
            (victim_page(i));
        end
    end
    /* perform batch update on the marked Translation Pages */
    foreach translation_page_i ∈ translation_page_update_set do
        curr_translation_blk ← get_curr_translation_blk();
        old_translation_page ← translation_page_i;
        update_translation_page(translation_page_i, curr_translation_blk);
        invalidate(old_translation_page);
        update_GTD(translation_page_i);
        if translation_page_i ∈ CachedMappingTable then
            update_CMT(translation_page_i);
        end
    end
end
erase_blk(victim); /* erase the victim block */
```

Algorithm 2: Garbage Collection

3.3.4 Dealing with Power Failure

Although flash is a non-volatile storage device, it relies on volatile on-flash SRAM which is susceptible to power failure in the host. When power failure occurs, all logical-physical mapping information stored in the Cached Mapping Table on

	Replacement Block FTL	BAST	FAST	SuperBlock	LAST	DFTL	Ideal Page FTL
FTL type	Block	Hybrid	Hybrid	Hybrid	Hybrid	Page	Page
Mapping Granularity	Block	DB-Block LB - Page	DB-Block LB-Page	SB - Block LB/Blocks within SB-Page	DB/Sequential LB - Block Random LB - Page	Page	Page
Division of Update Blocks (M)	-	-	1 Sequential + (M-1) Random	-	(m) Sequential-(M-m) Random (Hot and Cold)	-	-
Associativity of Blocks (Data:Update)	(1:K)	(1:M)	Random LB-(N:M-1) Sequential LB-1:1	(S:M)	Random LB-(N:M-m) Sequential LB-(1:1)	(N:N)	(N:N)
Blocks available for updates	Replacement Blocks	Log Blocks	Log Blocks	Log Blocks	Log Blocks	All Data Blocks	All Blocks
Full Merge Operations	Yes	Yes	Yes	Yes	Yes	No	No

Table 3.1. FTL Schemes Classification. N: Number of Data Blocks, M: Number of Log Blocks, S: Number of Blocks in a Super Block, K: Number of Replacement Blocks. DB: Data Block, LB: Log Block, SB: Super Block. In FAST and LAST FTLs, random log blocks can be associated with multiple data blocks.

SRAM will be lost. The traditional approach of reconstructing the mapping table utilizes scanning the logical addresses stored in the OOB area of all physical pages on flash [21]. However, the scanning process incurs high overhead and leads to long latencies while the mapping table is being recovered. In DFTL, the Global Translation Directory stores the locational information corresponding to the Global Mapping Table. Thus, storing the GTD on non-volatile storage resilient to power failure such as a fixed physical address location on flash device itself helps to bootstrap recovery. This can be performed periodically or depending on the required consistency model. Moreover, since GTD size is very small (4KB for 1GB flash), the overhead involved in terms of both space as well as extra operations is also very small. However, at the time of power failure there may be some mappings present in the Cached Mapping Table, that have been updated but not yet written back to map pages on flash. If strong consistency is required then even the Cached Mapping Table needs to be saved along with the GTD.

3.3.5 Comparison of State-of-the-art FTLs with DFTL

Table 3.1 shows some of the salient features of different FTL schemes. The DFTL architecture provides some intrinsic advantages over existing state-of-the-art FTLs which are as follows:

- Existing hybrid FTL schemes try to reduce the number of full merge operations to improve their performance. DFTL, on the other hand, completely

does away with full merges. This is made possible by page-level mappings which enable relocation of any logical page to any physical page on flash while other hybrid FTLs have to merge page-mapped log blocks with block-mapped data blocks.

- DFTL utilizes page-level temporal locality to store pages which are accessed together within same physical blocks. This implicitly separates hot and cold blocks as compared to LAST and Superblock schemes [22, 23] require special external mechanisms to achieve the segregation. Thus, DFTL adapts more efficiently to changing workload environment as compared with existing hybrid FTL schemes.

- Poor random write performance is argued to be a bottleneck for flash based devices. As is clearly evident, it is not necessarily the random writes which cause poor flash device performance but the intrinsic shortcomings in the design of hybrid FTLs which cause costly merges (full and partial) on log blocks during garbage collection. Since DFTL does not require these expensive full-merges, it it is able to improve random write performance of flash devices.

- All hybrid log-buffer based schemes maintain a very small fraction of log blocks (3% of total blocks [23]) to keep the page-level mapping footprint small (in SRAM). This forces them to perform garbage collection as soon as these log blocks are utilized. Some schemes [20, 22] may even call garbage collector even though there are free pages within these log blocks (because of low associativity with data blocks). DFTL, on the other hand, can delay garbage collection till $GC_{threshold}$ is reached which can be dynamically adjusted to suit various input streams.

- In hybrid FTLs, only log blocks are available for servicing update requests. This can lead to low block utilization for workloads whose working-set size is smaller than the flash size. Many data blocks will remain un-utilized (hybrid FTLs have block-based mappings for data blocks) and unnecessary garbage collection will be performed. DFTL solves this problem since updates can be performed on any of the data blocks.

3.4 FlashSim: A Simulator for NAND Flash-based Solid-State Drives

Recently, NAND Flash memory has become the main storage media for embedded devices, such as PDAs and music players. NAND Flash memory is now also being used in systems ranging from laptop and desktop computers to enterprise-scale storage servers. NAND Flash memory offers a number of benefits over the conventional hard disk drives (HDDs). These benefits include lower power consumption, lighter weight, higher resilience to external shock, the ability to sustain hotter operating regimes, and faster access times (with some exceptions that arise due to random writes). Unlike HDDs, NAND flash memory based Solid-State Disks (SSDs) have no mechanical moving parts, such as a spindle and voice-coil motors. Despite these benefits, a storage system designer needs to carefully consider the use of SSDs because they also have some notable weaknesses. The main weaknesses of SSDs include a higher price ($/GB) than HDDs, writes being 4-5 times slower than reads, slowdown in device throughput during periods of garbage collection that are hastened by small, random writes [43], and limited lifetime (10K-1M erase cycles per block) [47].

In order to overcome the limitations described above, a variety of complementary approaches have been proposed. For example, Multi-level Cell (MLC) technology gives higher density and cost per GB than Single-level Cell (SLC) [25]. The downside of MLC is that read and write times of MLC are slower. Consequently, there are current attempts to employ combinations of SLC and MLC Flash chips in SSDs. Numerous techniques for efficient address translation, garbage collection, and wear-leveling in the Flash Translation Layer (FTL) (more details in Chapter 2) have been explored to improve the performance of the SSD devices and/or providing longer lifetimes.

3.4.1 Motivation for A Simulator Framework

The design and implementation of cost efficient, reliable SSDs requires faithful and accurate evaluation test-beds for evaluating new algorithms for specific software components (such as those that constitute the FTL) within different hardware

configurations of the SSD before implementing them in the actual firmware. The fact that significant aspects of the techniques employed within SSDs are unknown to the pubic due to technology property issues further adds to the urgency of having such a test-bed for SSD research. With this motivation, we have designed and developed a simulation infrastructure.

Research Contributions - Here are the salient features and contributions of our work.

- The components of an SSD are can be classified as those belonging to the hardware and the software categories. The hardware component consists of a processing unit, memory, bus, and Flash chips. The software component (which executes on the processing unit) consists of a FTL. The price of the SSD depends on the hardware configuration in the SSD and the software running on the hardware, but there is a lack of test infrastructure to examine cost-effective hardware configurations and software algorithms in research environments outside those affiliated with manufacturers of SSDs. In this work, we provide an experimental test-bed to fill this void.

- The few efforts that have attempted to provide the simulation/emulation infrastructure [48, 10] lack desirable features, especially an object-oriented design. It is typically difficult to understand and enhance these simulators. Compared to other existing/evolving SSD simulators, FlashSim is entirely objected-oriented. Our approach allows the developers to easily understand, use, and extend our simulator. Furthermore, our simulator has been integrated with the well-regarded and popular DiskSim simulator [49] and validated for behavioral similarity with real SSD devices.

3.4.2 SSD Simulator Design

We have designed and implemented a SSD simulator that is based on the hardware diagram in Figure 3.5. The first version of our SSD simulator focused on software components (for instance, FTL schemes, garbage collection, and wear-leveling); we considered a simplified hardware model that simulated a single *Plane* with a simplified channel implementation.

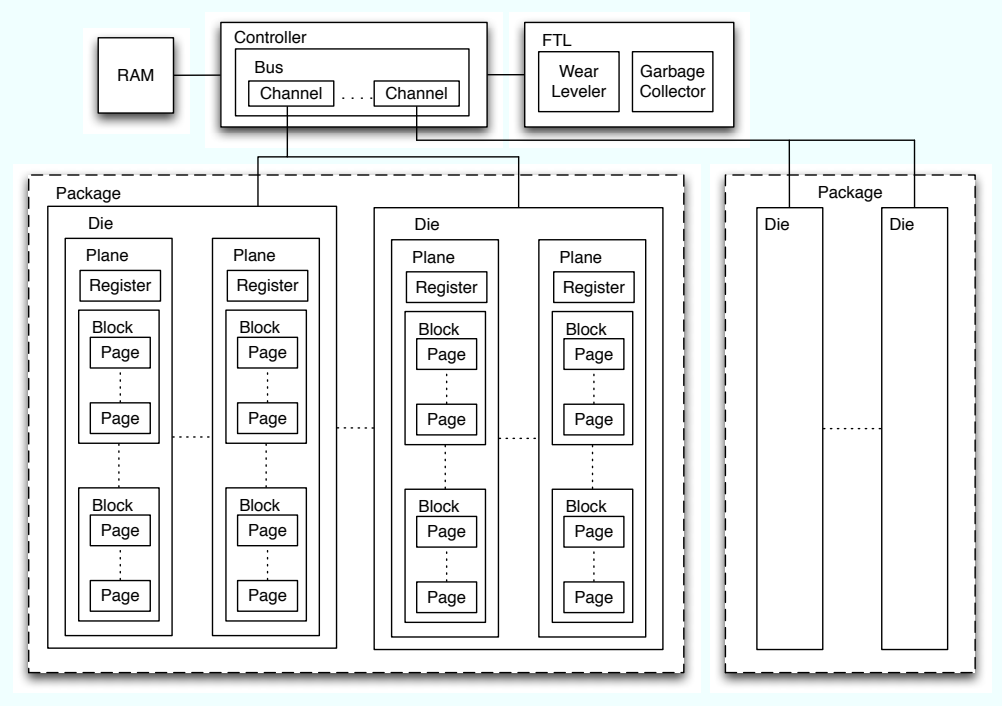


Figure 3.5. Hardware diagram for the SSD Simulator. Ellipses in between two of the same components indicate where more of the same components may be added. Only the full component break-down of the left-most package is shown.

Since this version of our simulator was limited by a simplified hardware model and not easy to extend due to a highly coupled implementation with DiskSim, we re-designed and re-implemented the simulator with an object-oriented approach. Our new simulator is entirely event-driven and written in a familiar language, C++; we achieve modularity, low coupling, and high cohesion. Our hardware-level diagram is shown in Figure 3.5.

3.4.2.1 Object-Oriented Component Design

The simulator was written as a single-threaded program in C++ for simplicity. C++ could provide a comprehensible object-oriented scheme where each class instance represented a hardware or software component. The UML diagram in Figure 3.6 of the Appendix contains all C++ classes used by the SSD simulator. FlashSim is integrated with Disksim's C code.

Hardware Component Design - The classes in the SSD simulator for hardware components are as follows:

- ***SSD:*** The SSD class serves to provide an interface to Disksim and provide a single class to instantiate in order to create the SSD simulator module. The SSD class creates event objects to wrap the Disksim *ioreq_event* structures and returns the event time to disksim.

- ***Package:*** The package class represents a group of flash dies that share a bus channel. The package class allocates its dies in its constructor and connects the dies to a bus channel. The package also facilitates addressing.

- ***Die:*** A die is a single flash memory chip that is organized into a set of planes. Dies are connected to bus channels, but individual planes contained in the die buffer bus transfers. In future development, the highest level at which merge operations may take place will be at the die level. The corresponding event object is updated with the merge delay time.

- ***Plane:*** Planes are comprised of blocks and provide a single page-sized register to buffer page data for bus transfers. The register is also used as a buffer for merge operations inside planes. The corresponding event object is updated with merge delays for merge operations and considers register delays.

- ***Block:*** A block is comprised of pages and is the smallest component that can be individually erased. When a block is erased, all pages in it are erased and can then be written to again. The corresponding event object is updated with the erase delay time. A block can only be erased a finite number of times because of reliability constraints [47].

- ***Page:*** Each page maintains its state and updates event objects with the read and write delays of the given flash technology. Page states include free/empty after erasure, valid after a successful write, and invalid after being copied to a new location in a merge operation.

- ***Controller:*** The controller class receives event objects from the SSD and consults the FTL regarding how to handle each event. The controller sends

the virtual data for events to the RAM for buffering before sending the event object to the bus.

- ***RAM:*** The RAM class calculates how long it takes to read or write data to itself. The RAM buffers virtual event data for the controller to send across the bus.

- ***Bus:*** The bus class has a number of channels that are each shared by all the dies in a package. The bus examines addresses in events and passes the event object on to the proper channel.

- ***Channel:*** Channels must schedule usage for events and update the event time values. Each channel keeps a scheduling table that keeps track of channel usage, and new events are scheduled at the next available free time slot after dependencies have been met. The scheduling table size is synonymous to queue length.

Software Component Design - The classes in the SSD simulator for software components are as follows:

- ***Event:*** First, the event class keeps track of its corresponding Disksim *ioreq_event* structure. Second, the event class holds methods and attributes to do all the record-keeping for the SSD simulator's state, including SSD addresses. Simulator objects pass event class objects and update the event objects statistics.

- ***Address:*** Addresses are comprised of a separate field for each hardware address level from the package down to the page. We provide an address class instead of a *struct* to help make a clear interface to assign and validate addresses.

- ***FTL:*** The FTL provides address translation from logical addresses to physical addresses. It determines how to process events that involve many pages by producing a list of single-page events to be processed in-order by the controller. The FTL is responsible for taking advantage of hardware parallelism

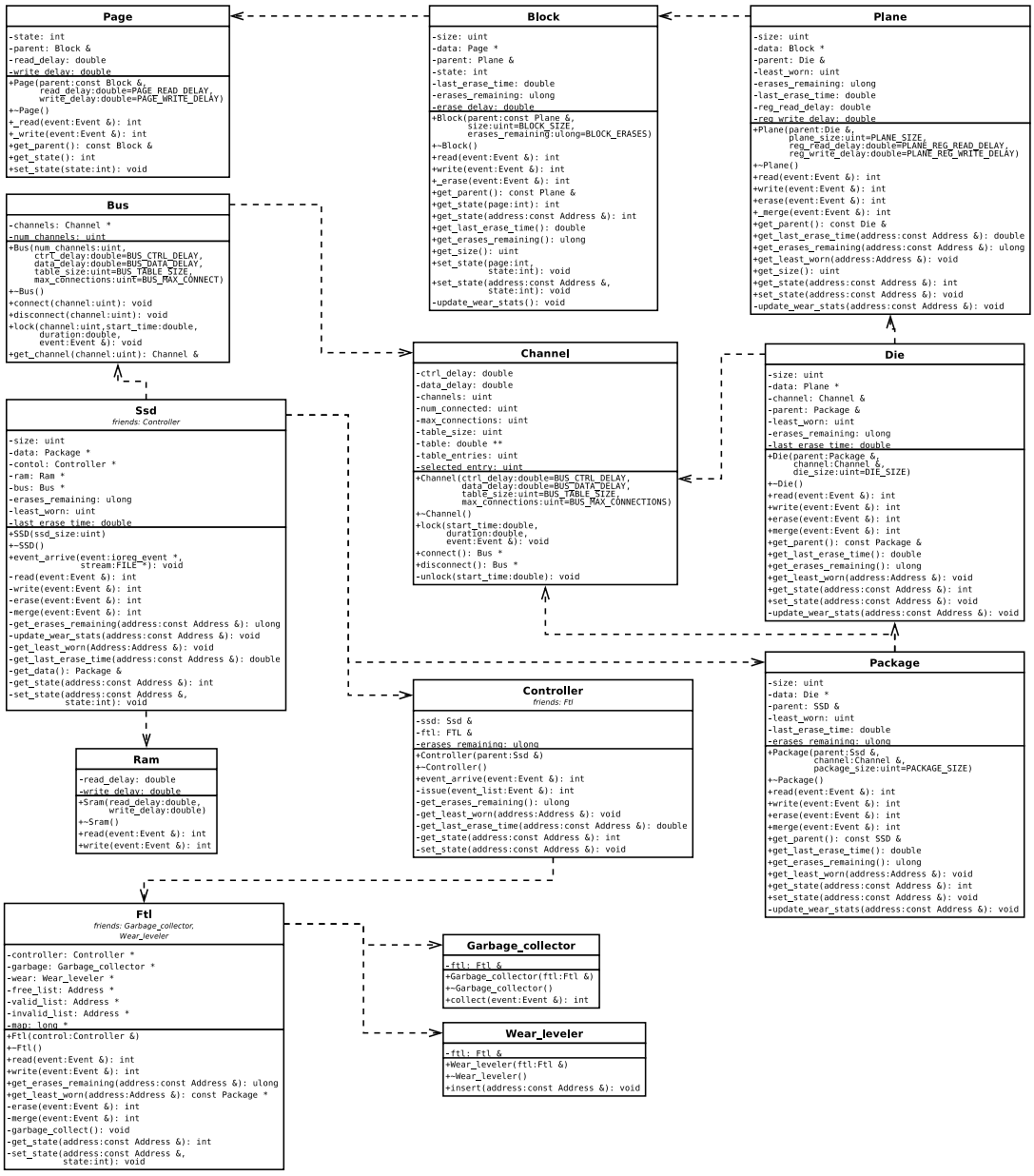


Figure 3.6. Arrows indicate dependencies of all types, including aggregation. Most dependencies arise from one class having references to another class, though many references are initialized by allocating a new instance of the aggregate class in the constructor.

for performance. The FTL also has a wear leveler and garbage collector to facilitate its tasks.

- ***Wear Leveler:*** The wear leveler class helps spread the block erasures over all blocks in the SSD. The wear leveler is responsible for keeping as many

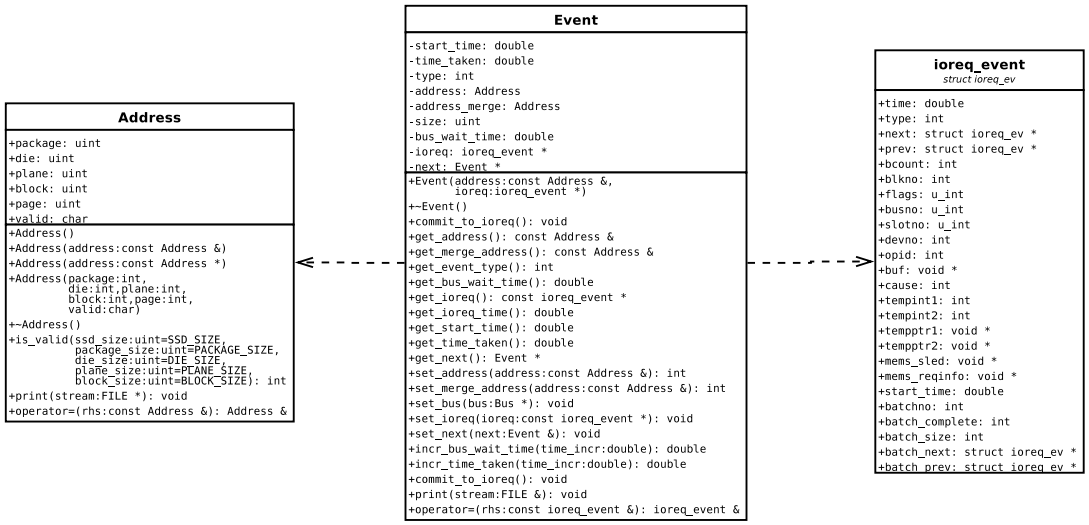


Figure 3.7. Arrows indicate dependencies of all types, including aggregation. Most dependencies arise from one class having references to another class, though many references are initialized by allocating a new instance of the aggregate class in the constructor.

blocks functional for as long as possible because blocks of pages can only be erased for reuse a finite number of times.

- ***Garbage Collector:*** The garbage collector is activated when a write request cannot be satisfied because the selected block is not writable or there is not enough free space in the selected block. The garbage collector seeks to merge partially-used blocks and free up blocks by erasing them. Any other algorithm for GC can also be simulated.

3.4.2.2 Bus Channel Interleaving

Figure 3.8 shows the interleaving of processing events for one bus channel. As per Figure 3.5, each bus channel connects to several flash dies that are grouped in a package. Each bus channel functions independently and in parallel; operations on different channels are not dependent on each other. The read interleaving for one bus channel is shown in Figure 3.8-(a). First, the control time signifies when the bus channel is locked for control signals that request a flash die to prepare data from a specific page. Next, the flash die processes the request for the data to be read. The bus channel is free to handle other requests at this time. Finally, the bus channel is locked for control signals that request the flash die to send data

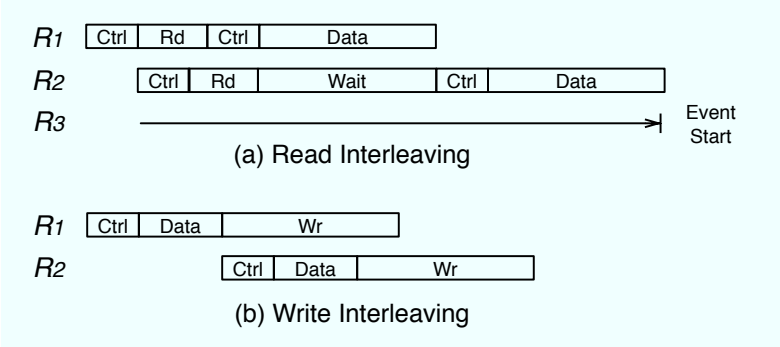


Figure 3.8. Interleaving for read/write requests

from a specific page and sending the data. The interesting part of this figure is the bus channel idle time period between the end of the control time for request two (R_2) and the beginning of the second control time period for request one (R_1). A control time period for request three cannot fit; request three (R_3) must be delayed until after request two finishes.

The write for one bus channel is shown in Figure3.8-(b). First, the bus channel must be locked for control signals to inform the proper flash die that it will receive data. Second, the bus remains locked to send the data. Finally, the flash die writes the data; the bus channel is free to handle other requests at this time. Since write requests only require one contiguous time block of bus channel time, write request happen in FIFO.

3.4.2.3 Event Flow

The SSD simulator is instantiated as a SSD object designed to accept *ioreq_event* structures from Disksim. Its functionality is described in detail in Algorithm 3. The SSD controller uses the FTL software module to create a list of events for a multi-page request. The controller issues each event in the list to the data hardware through corresponding bus channels. The bus channels handle the scheduling and interleaving of events for the controller; this simplifies our controller implementation. In Algorithm 4, events continue through the package and are handled starting at the die level; merge events can be handled inside flash dies or planes. Erase events are handled inside blocks, and read and write events are handled inside pages. The SSD and package components are included in the call stack

after consulting the bus channel because these components also keep track of wear statistics. Wear statistics stored in the SSD, package, die, plane, and block are updated every time an erase event occurs to keep a simple interface with lower algorithmic complexity for the FTL.

```
Input: Disksim's I/O Request Structure (ioreq_event)
Output: Device Service Time
foreach ioreq_event do
    begin SSD process ioreq_event
        wrap in event object;
        begin controller, FTL process event
            consult wearleveler and garbagecollector;
            create page-sized list of event objects;
            foreach e in event_list do
                begin SSD, bus, channel process e
                    lock for next available transfer time;
                    e_time ← e_time + channel_delay;
                end
                Package(e);
            end
            if e_type = erase then
                update SSD wear stats;
            end
            begin inform bus, channel: e finished
                channel update scheduling table for event dependencies;
            end
        end
    end
end
```

Algorithm 3: SSD simulator functionality

```
Input: Event object (e)
Output: NULL
begin package, die process e
    /* Merge event e in die                                                    */
    if e_type = mergeande_addr.plane ≠ e_addr_merge.plane then
        foreach valid page v in e_addr.block x do
            foreach empty page t in e_addr_merge.block y do
                t ← v;
                v_state ← invalid;
                t_state ← valid;
            end
            e_time ← e_time + die_merge_delay;
        end
    end
    /* Merge event e in plane                                                  */
    else plane process e
        plane_register ← e_data;
        if e_type = merge then
            foreach valid page v in e_addr.block x do
                foreach empty page t in e_addr_merge.block y do
                    t ← v;
                    v_state ← invalid;
                    t_state ← valid;
                end
                e_time ← e_time + die_merge_delay;
            end
        end
        /* e_type = read or write or erase                                     */
        else
            begin block process e
                if e_type = erase then
                    for each page in block x do
                        page_state ← empty;
                    end
                    e_time ← e_time + erase_delay;
                    update wear stats;
                end
                /* e_type = read or write                                      */
                else page process e
                    if e_type = read then
                        e_time ← e_time + read_delay;
                    end
                    else if e_type = write then
                        e_time ← e_time + write_delay;
                    end
            end
        end
    if e_type = erase then
        update plane, die, package wear stats;
    end
end
```

Algorithm 4: Package (event object) - SSD hardware functionality inside a package. This function is being called in Algorithm 3.

Flash Organization	
Flash Type	Large Block
Page (Data/OOB)	2KB/64Byte
Block	(128KB+4KB)
Latency & Energy Consumption	
Page Read	(130.9 us, 4.72uJ)
Page Write	(405.9 us, 38.04uJ)
Block Erase	(1.5 ms, 527.68uJ)
Firmware	
Garbage Collection	Yes
Wear-leveling	Implicit/Explicit
FTL	Page/DFTL/FAST

(a) Simulation Paremeters

MTron Real SSD1	
Model	MSP-7000
Specification	2.5 in, 4-way, SLC
Read/Write (MB/s)	120/90
Super Talent Real SSD2	
Model	FSD32GB25M
Specification	2.5 in, 1-way, SLC
Read/Write (MB/s)	60/45

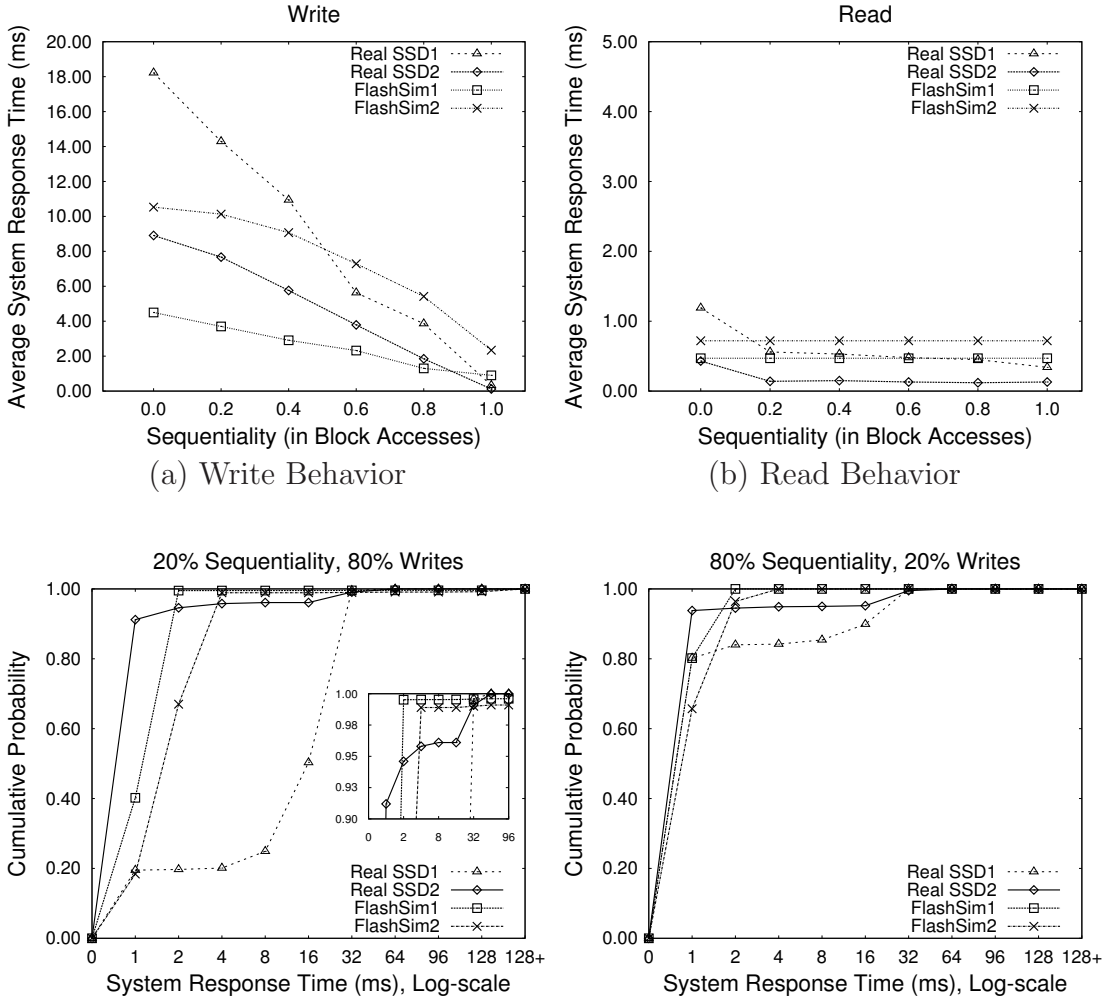
(b) Real SSD Devices

Table 3.2. Simulation parameters and real SSD device observed specifications.

3.4.3 Validation of SSD Simulator

We validated our simulator by comparing it to real SSDs for behavioral similarity; we compared the performance of different FTL schemes for realistic workload traces. We used the simplified version of the simulator that simulates a single *Plane* with with a simplified channel implementation for various software implementations, such as the FTL, garbage collector, and wear-leveler. More thorough evaluation that also considers interleaving with parallelism effects is left for future work.

Evaluation Setup - The specifications available for commercial SSDs are insufficient for modeling them accurately. For example, the memory cache size for FTL mappings and the exact FTL scheme used are not disclosed. Hence, it is difficult to simulate these commercial devices. We made assumptions for flash devices as



(a) Write Behavior (b) Read Behavior

(c) Random Write Dominant (d) Sequential Read Dominant

Figure 3.9. Validation of our SSD Simulator. Note that in the legends, Real SSD1, Real SSD2, FlashSim1, and FlashSim2 denote Mtron's SSD, SuperTalent's SSD, a SSD using a page-based FTL, and a SSD using DFTL.

described in Table 5.4 and configured our simulator accordingly.

Results - Using the parameters from Table 5.4, we validated our flash device simulator against commercial SSDs (MTron's SSD [16] and Super-Talent's SSD [50]) for *behavioral similarity*. For this purpose, we sent raw I/O requests to real SSDs and similar traces to our flash device simulator to measure device performance. As shown in Figure 3.9, our simulator was able to capture the performance trends exhibited by the real SSDs. With increasing sequentiality of writes (Fig-

ure 3.9-(a)), the performance of real SSDs improved, and our flash simulator with various FTLs was able to provide similar characteristics. When examining reads (Figure 3.9-(b)), real SSDs showed much less variation; the same was observed with our simulator. With a high degree of randomness in writes (80% random in Figure 3.9-(c)), real SSDs demonstrated long-tailed response time distribution (due to larger GC overhead); our simulator exhibited a similar trend.

Comparison of Existing SSD Simulators with FlashSim - Other research has been conducted to develop a simulator for NAND flash-based SSDs [10, 48]. Microsoft Research's simulator [10] is one of the first available SSD simulators; however, it is highly coupled with DiskSim. The strengths of their simulator include the implementation of parallelism effects across multiple channels and interleaving across different components within a single plane, but only a page-based FTL scheme is available. J. Lee et. al have developed a simple flash based SSD simulator [48]. This simulator is a stand-alone simulator that is limited by a single FTL scheme implementation, and they do not simulate I/O queueing effects.

Compared to the above simulators, our simulator has ability to simulate multiple FTL schemes, including page-based, block-based, FAST [21], and DFTL [51]. Our simulator is integrated with DiskSim to simulate queuing effects, and our simulator module can be instantiated multiple times within Disksim. Our single-threaded, event-driven, object-oriented approach is comprehensible and modular to allow for future extensions. Furthermore, we have validated FlashSim against real SSD devices for behavioral similarity.

3.4.4 Summary and Future Work

We have developed a flexible and robust simulator for SSDs that features an object-oriented design. We have validated our simulator with real SSD devices by demonstrating behavioral similarity and compared performance results for various FTL schemes. We also have analyzed the impact of various FTL schemes on performance and power consumption in the SSD.

Future Work - This project is a work in progress. Since the simulator has only been validated with a simple behavioral model for a single plane and simplified

channel implementation, we will continue with more thorough validation methods that include bus channel interleaving effects. Caching and I/O scheduling effects will be added and examined. Since our simulator module can have multiple instances in Disksim, we can simulate disk arrays that contain a combination of both SSDs and HDDs. In addition to performance simulation, our simulator is able to incorporate power models and other extensions. We plan to combine our thermal-performance simulator of disk drives [52] with our future work involving hybrid disk arrays that contain a combination of both SSDs and HDDs.

Download - Source-code is available for download from `http://csl.cse.psu.edu/hybridstore`.

3.5 Experimental Results Using FlashSim

We use FlashSim to evaluate the performance of DFTL and compare it with both (i) a state-of-the-art hybrid FTL FAST [21]) and an (ii) idealized page-based FTL with sufficient SRAM (called *Baseline* FTL henceforth). The Baseline scheme assumes the availability of sufficient SRAM to store the *entire* address translation table, thus doing away with all overheads with regards to the translation pages.

Evaluation Setup We simulate a 32GB NAND flash memory with specifications shown in Table 2.1. To conduct a fair comparison of different FTL schemes, we consider only a portion of flash as the *active region* which stores our test workloads. The remaining flash is assumed to contain cold data or free blocks which are not under consideration during the evaluation. We assume the SRAM to be just sufficient to hold the address translations for FAST FTL. Since the actual SRAM size is not disclosed by device manufacturers, our estimate represents the minimum SRAM required for the functioning of a typical hybrid FTL. We allocate extra space (approximately 3% of the total active region [22]) for use as log-buffers by the hybrid FTL.

We use a mixture of real-world and synthetic traces to study the impact of different FTLs on a wide spectrum of enterprise-scale workloads. Table 5.3 presents salient features of our workloads. We employ a write-dominant I/O trace from an

Workloads	Average Request Size (KB)	Read (%)	Sequentiality (%)	Average Request Inter-arrival Time (ms)
Financial (OLTP)	4.38	9.0	2.0	133.50
Cello99	5.03	35.0	1.0	41.01
TPC-H (OLAP)	12.82	95.0	18.0	155.56
Web Search	14.86	99.0	14.0	9.97

Table 3.3. Enterprise-Scale Workload Characteristics.

OLTP application running at a financial institution [53] made available by the Storage Performance Council (SPC), henceforth referred to as the *Financial trace*. We also experiment using Cello99 [54], which is a disk access trace collected from a time-sharing server exhibiting significant writes; this server was running the HP-UX operating system at Hewlett-Packard Laboratories. We consider two read-dominant workloads to help us assess the performance degradation, if any, suffered by DFTL in comparison with other state-of-the-art FTL schemes due to its address translation overhead. For this purpose, we use TPC-H [55], which is an ad-hoc, decision-support benchmark (OLAP workload) examining large volumes of data to execute complex database queries. Also, we use a read-dominant Web Search engine trace [56] made available by SPC. Finally, apart from these real traces we also use a number of synthetic traces to study the behavior of different FTL schemes for a wider range of workload characteristics than those exhibited by the above real-world traces.

The *device service time* is a good metric for estimating FTL performance since it captures the overheads due to both garbage collection and address translation. However, it does not include the *queuing delays* for requests pending in I/O driver queues. In this study, we utilize both (i) indicators of the garbage collector's efficacy and (ii) response time as seen at the I/O driver (this is the sum of the device service time and time spent waiting in the driver's queue, we will call it the *system response time*) to characterize the behavior/performance of the FTLs. The garbage collection overhead is demonstrated through the impact of merges, the copying of valid pages, and the erasing of the blocks in these operations. In subsequent subsections, we highlight the cost of full merges, examine the performance of different FTL schemes, and evaluate their ability to handle overload conditions in different workloads.

3.5.1 Garbage Collection and Address Translation Overheads

So far we have argued qualitatively that GC is the main source of overheads in an FTL scheme. As explained earlier, the garbage collector may have to perform merge operations of various kinds (switch, partial, and full) while servicing update requests. Recall that merge operations pose overheads in the form of block erases. Additionally, merge operations might induce copying of valid pages from victim blocks—a second kind of overhead. We report both these overheads as well as the different kinds of merge operations in Figure 3.11 for our workloads. As expected from Section 3.3 and corroborated by the experiments shown in Figure 3.11, read-dominant workloads (TPC-H and Web Search)—with their small percentage of write requests—exhibit much smaller garbage collection overheads than Cello99 or Financial trace. The number of merge operations and block erases are so small for the highly read-dominant Web Search trace that we do not show these in Figures 3.11(a),(b), and (c).

Switch merges - Hybrid FTLs can perform switch merges only when the victim update block (selected by garbage collector) contains valid data belonging to logically consecutive pages. DFTL, on the other hand, with its page-based address translation, does not have any such restriction. Hence, *DFTL shows a higher number of switch merges* for even random-write dominant Financial trace as seen in Figure 3.11(a).

Full merges - As shown in Figure 3.10, with FAST, about 20% of the full merges in the Financial trace involve 20 data blocks or more. This is because state-of-the-art hybrid FTLs allow high associativity of log blocks with data blocks while maintaining block-based mappings for data blocks, thus requiring a costly operation of merging data pages in the victim log block with their corresponding data blocks (recall Figure 2.6 in Section **??**). For TPC-H, although DFTL shows a higher number of total merges, its fine-grained addressing enables it to *replace full merges with less expensive partial merges.* With FAST as many as 60% of the full merges involve more than 20 data blocks. As we will observe later, this directly impacts FAST's overall performance. Figure 3.11(b) shows the higher number of

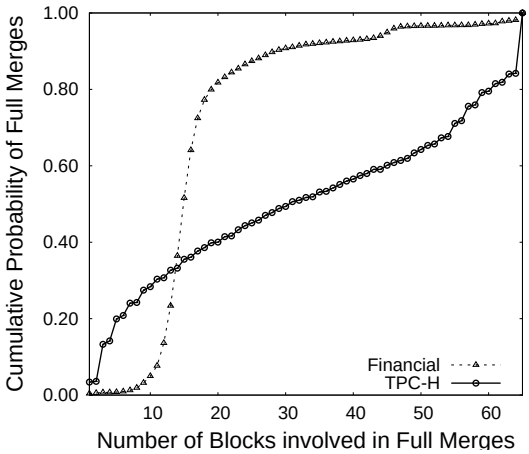
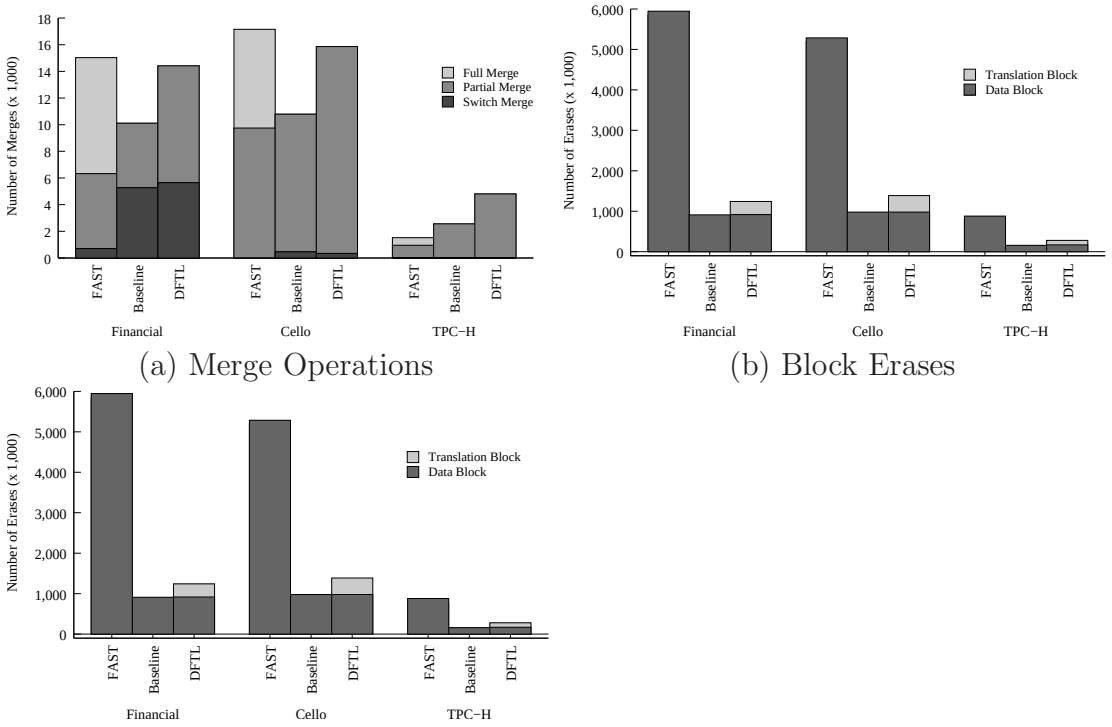


Figure 3.10. Illustration of the large number of expensive full merge operations induced by the hybrid FAST FTL scheme. About 20% of full merges involve 20 data blocks or more for the Financial trace.

Workloads	FTL Type	Erase (#)		Merges			Read Overhead				Write Overhead			
		Data	Map	S	P	F	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)
Financial	Baseline	10,111	-	5,275	4,836	-	15,573	-	-	-	15,573	-	-	-
	DFTL	10,176	4,240	5,650	8,766	-	19,369	5,945	517,456	7,582	19,369	5,945	258,017	7,582
	FAST	151,180	-	374	5,967	8,865	1,508,490	-	-	-	1,508,490	-	-	-
Cello	Baseline	10,787	-	447	10,340	-	81,109	-	-	-	81,109	-	-	-
	DFTL	10,795	5,071	353	15,513	-	82,956	42,724	730,107	29,010	82,956	42,724	251,518	29,010
	FAST	134,676	-	1	9,763	7,694	3,149,194	-	-	-	3,149,194	-	-	-
TPC-H	Baseline	2,544	-	14	2,530	-	102,130	-	-	-	102,130	-	-	-
	DFTL	2,678	2,118	5	4,791	-	110,716	75,255	1,449,183	10,018	110,716	75,255	50,242	10,023
	FAST	19,476	-	5	949	568	618,459	-	-	-	618,459	-	-	-
Web-Search	Baseline	-	-	-	-	-	-	-	-	-	-	-	-	-
	DFTL	15	350	-	365	-	480	6391	1,588,120	51	480	6,391	16,390	51
	FAST	-	-	-	-	-	-	-	-	-	-	-	-	-

Table 3.4. Analysis of garbage collection overhead for various FTLs. (1): Number of data page reads in GC, (2): Number of map page reads in GC, (3): Number of map page reads for address translation, and (4):Number of map page reads when victim block is a data block. (5): Number of data page writes in GC, (6): Number of map page writes in GC, (7): Number of map page writes for address translation, and (8):Number of map page writes when victim block is a data block. *Base* in FTL type denotes a Baseline FTL scheme.

block erases with FAST as compared with DFTL for all our workloads. This can be directly attributed to the large number of data blocks that need to be erased to complete the full merge operation in hybrid FTLs. Moreover, in hybrid FTLs only a small fraction of blocks (log blocks) are available as update blocks, whereas DFTL allows all blocks to be used for servicing update requests. This not only improves the block utilization in our scheme as compared with FAST but also contributes in reducing the invocation of the garbage collector.



(a) Merge Operations

(b) Block Erases

(c) Extra Read/Write Operations

Figure 3.11. Overheads with different FTL schemes. We compare DFTL with FAST and Baseline for three workloads: Financial, Cello99, and TPC-H. The overheads for the highly read-oriented Web Search workload are significantly smaller than others and we do not show them here. In (c), Address Translation (Read) and Address Translation (Write) denote the extra read and write operations for address translations required in DFTL, respectively. All extra read/write operations have been normalized with respect to FAST FTL.

Translation and valid page copying overheads - DFTL introduces some extra overheads due to its address translation mechanism (due to missed mappings that need to be brought into the SRAM from flash). Figure 3.11(c) shows the normalized overhead (with respect to FAST FTL) from these extra read and write operations along with the extra valid pages required to be copied during garbage collection. Even though the address translation accounts for approximately 90% of the extra overhead in DFTL for most workloads, overall it still performs less extra operations than FAST. For example, DFTL yields a 3-fold reduction in extra read/write operations over FAST for the Financial trace. Our evaluation supports the key insight behind DFTL, namely that the temporal locality present in workloads helps keep this address translation overhead small, i.e., most requests are

serviced from the mappings in SRAM. DFTL is able to utilize page-level temporal locality in workloads to reduce the valid page copying overhead since most hot blocks (data blocks and translation blocks) contain invalid pages and are selected as victims by our garbage collector. In our experiments, we observe about 63% hits for address translations in SRAM for the financial trace even with our conservatively chosen SRAM size. Furthermore, the relatively high address translation overhead can be attributed to the minimal size of SRAM that we have used in our experiments. In a later sub-section, we investigate how this overhead reduces further upon increasing the SRAM size. All the values used in Figure 3.11 are in Table 3.4.

3.5.2 Performance Analysis

Having seen the comparison of the overheads of garbage collection and address translation for different FTLs, we are now in a position to appreciate their impact on the performance offered by the flash device. The performance of any FTL scheme deteriorates with increase in garbage collection overhead. The Baseline scheme does not incur any address translation overhead and is also able to prevent full-merges because of fine-grained mapping scheme. Thus, it shows the best performance amongst all other *implementable* FTL schemes. The Cumulative Distribution Function of the average system response time for different workloads is shown in Figure 3.12. DFTL is able to closely match the performance of Baseline scheme for the Financial and Cello99 traces, both random-write dominant workloads. In case of the Financial trace, DFTL reduces the total number of block erases as well as the extra page read/write operations by about 3 times , thus decreasing the overall merge overhead by about 76%. This results in improved device service times and shorter queuing delays (refer to Table 3.5) which in turn improve the overall I/O system response time by about 78% as compared to FAST.

For Cello99, the improvement is much more dramatic because of the high I/O intensity which increases the pending requests in the I/O driver queue, resulting in higher latencies. Reviewers should be careful about the following while interpreting these results: we would like to point out that Cello99 represents only a point within a much larger enterprise-scale workload spectrum for which the gains of-

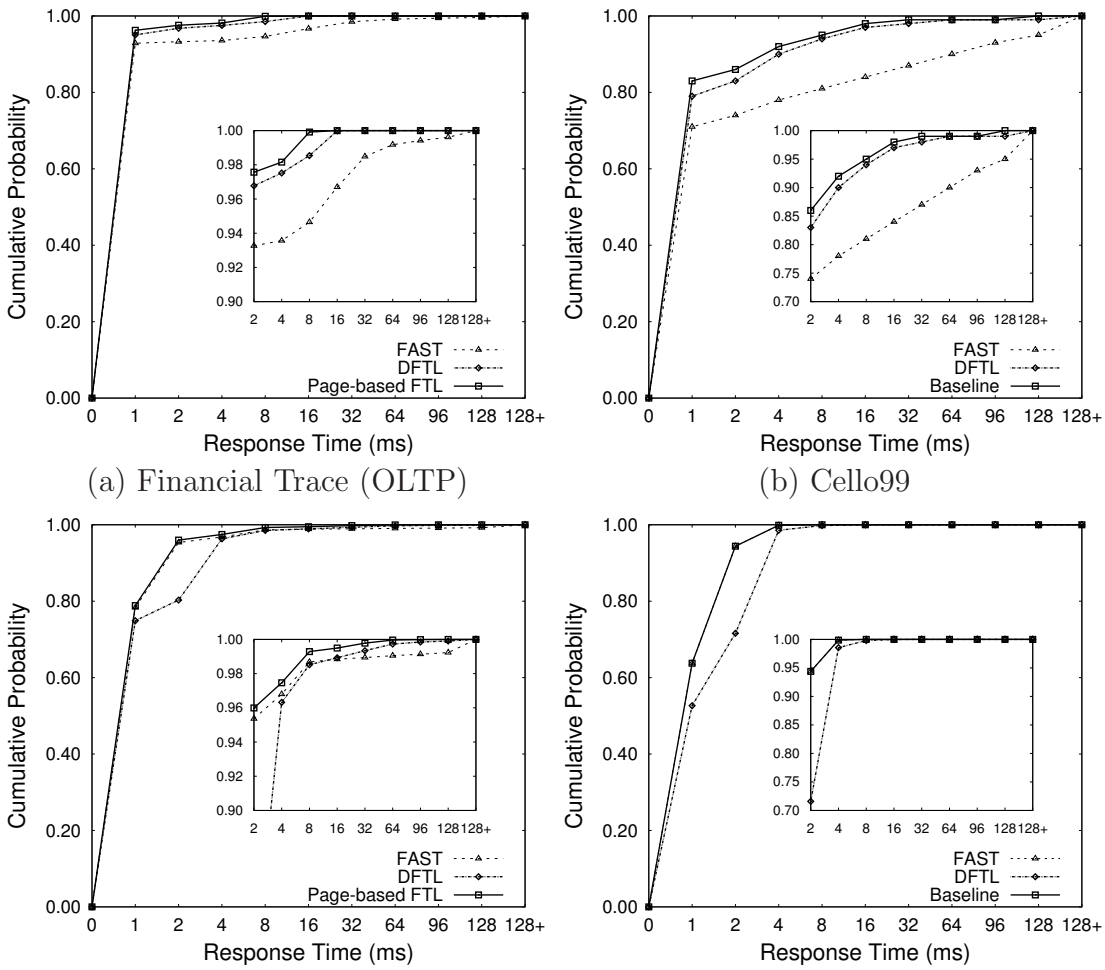


Figure 3.12. Each graph shows the Cumulative Distribution Function (CDF) of the average system response time for different FTL schemes.

fered by DFTL are significantly large. More generally, DFTL is found to improve the average response times of workloads with random writes with the degree of improvement varying with the workload's properties.

For read-oriented workloads, DFTL incurs a larger additional address translation overhead and its performance deviates from the Baseline (Figure 3.12(c) & (d)). Since FAST is able to avoid any merge operations in the Web search trace, it provides performance comparable to Baseline. However, for TPC-H, it exhibits a *long tail* primarily because of the expensive full merges and the consequent high latencies seen by requests in the I/O driver queue. Hence, even though FAST services about 95% of the requests faster, it suffers from long latencies in

Workloads	FTL Type	System Response Time (ms)		Device Response Time (ms)		I/O driver Queuing Delay (ms)	
		Avg.	std.dev	Avg.	std.dev	Avg.	std.dev
Financial	Baseline	0.43	0.81	0.39	0.79	0.04	0.19
	FAST	2.75	19.77	1.67	13.51	1.09	13.55
	DFTL	0.61	1.52	0.55	1.50	0.06	0.29
Cello99	Baseline	1.50	4.96	0.41	0.80	1.08	4.88
	FAST	16.93	52.14	2.00	14.59	14.94	50.20
	DFTL	2.14	6.96	0.59	1.04	1.54	6.88
TPC-H	Baseline	0.79	2.96	0.68	1.78	0.11	2.13
	FAST	3.19	29.56	1.06	11.65	2.13	26.74
	DFTL	1.39	7.65	0.95	2.88	0.44	6.57
Web Search	Baseline	0.86	0.64	0.68	0.44	0.18	0.46
	FAST	0.86	0.64	0.68	0.44	0.18	0.46
	DFTL	1.24	1.06	0.94	0.68	0.30	0.78

Table 3.5. Performance metrics for different FTL schemes with enterprise-scale workloads.

the remaining requests, resulting in a higher average system response time than DFTL.

For FAST to match the performance of DFTL for random-write dominant workloads, it needs a faster flash device. Figure 3.13 shows the necessary flash device speed-up required for FAST to achieve the performance comparable with our FTL scheme. A 4 times faster flash will require more investment to attain similar results. Thus, DFTL even helps in reducing deployment costs for flash based SSD devices in enterprise-servers.

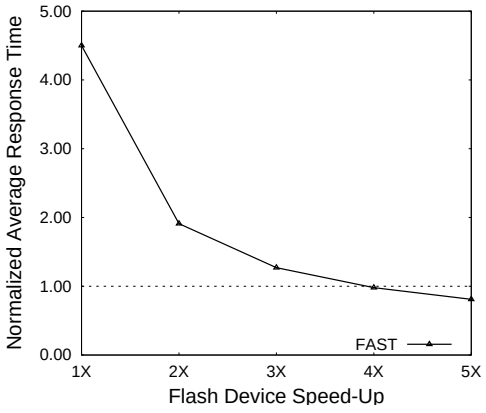
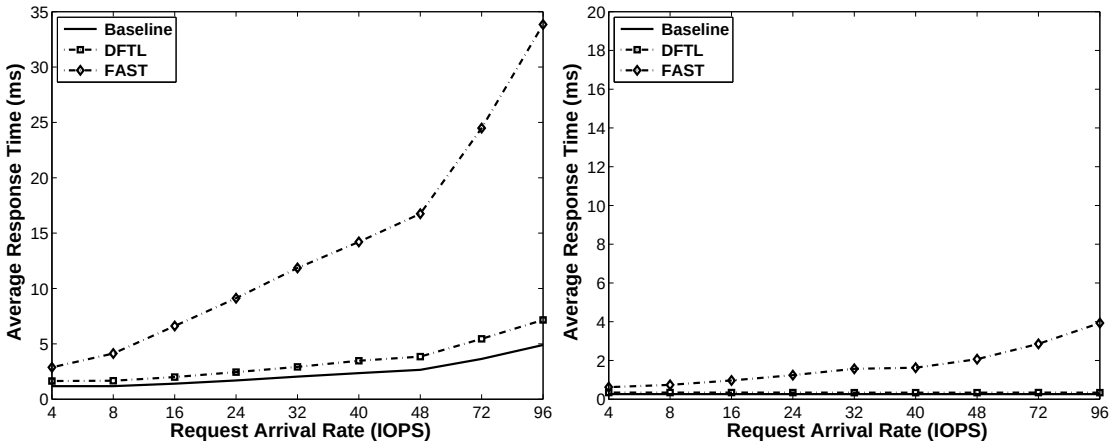


Figure 3.13. Performance improvement of FAST FTL with Flash Device Speed-Up for the Financial. Average Response times have been normalized with respect to DFTL performance without any speed-up (1X).

In the following section, we examine the the various overheads associated with different FTL schemes including the cost imposed by garbage collection, especially full-merges in state-of-the-art FTLs and the address translation overhead in DFTL.



(a) Workload A (b) Workload B

Figure 3.14. Performance comparison of various FTLs with changing I/O intensity for synthetic workloads. DFTL is able provide improved performance as well as sustain overloaded behavior in workloads much better than FAST. The 99% confidence intervals are very small and hence not shown.

3.5.3 Exploring a Wider Range of Workload Characteristics

We have seen the improvement in performance for different realistic workloads with DFTL as compared to state-of-the-art FTLs. Here, we widen the spectrum of our investigation by varying one workload property, namely I/O request arrival intensity. An enterprise-scale FTL scheme should be robust enough to sustain periods of increased I/O intensity, especially for write dominant workloads. In order to simulate such changing environments we use two synthetic workloads with varying characteristics: Workload A is predominantly random write-dominant whereas Workload B has a large number of sequential writes. With increasing request arrival rate, the flash device transitions from a *normal operational region* to an *overloaded region*.

As shown in Figure 3.14, for Workload A the transition into overloaded region is marked by very high gradient in response times pointing to the un-sustainability of such an environment using FAST. On the other hand, DFTL is not only able to provide improved performance in the operational region but is also able to sustain higher intensity of request arrivals. It provides *graceful degradation* in performance to sustained increase in I/O intensity, a behavior especially desirable

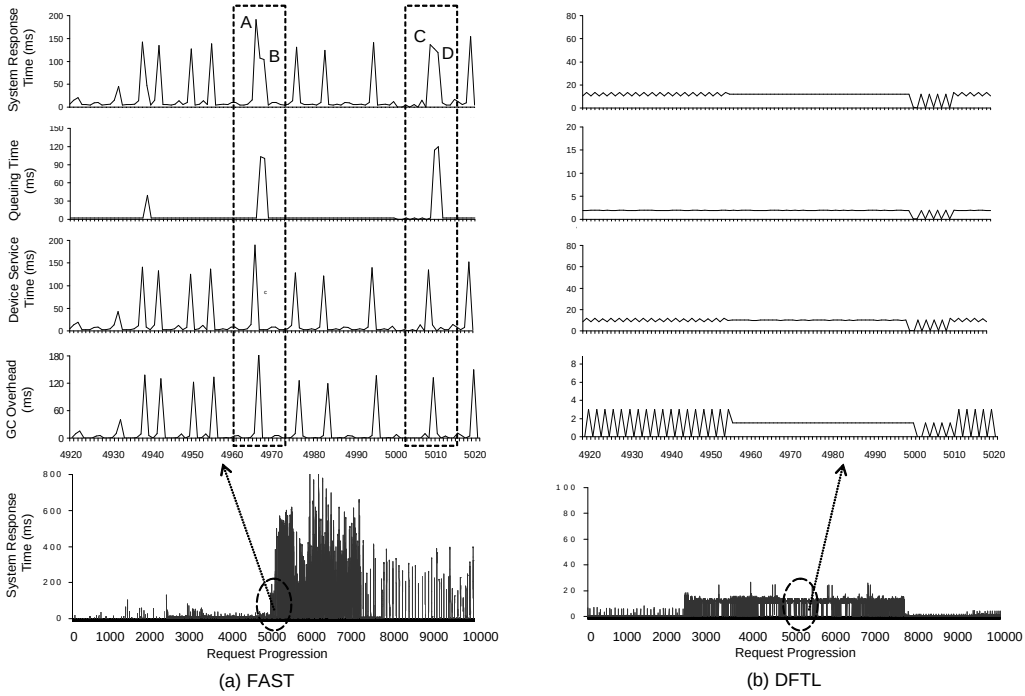


Figure 3.15. Microscopic analysis of DFTL and FAST FTL schemes with Financial trace. The selected region (requests 4920 to 5020) represents transition from normal operational region to overloaded region. Requests A & C undergo full-merges in FAST. However, their impact is also seen on requests B & D through long queuing latencies. Meanwhile, DFTL is able to provide much better performance in the same region.

in enterprise-scale systems. For sequential workload B, the merge overhead is reduced because of higher number of switch merges as compared to full-merges. Thus, FAST is able to endure the increase in request arrival rate, much better than its own performance with random-write dominant workload A. However, we still observe better performance from DFTL, which is able to approximate the performance of Baseline scheme because of the availability of all blocks to service the update requests.

3.5.4 Microscopic Analysis

In this sub-section, we try to perform a microscopic analysis of the impact of garbage collection on instantaneous response times by installing probes within FlashSim to trace individual requests.

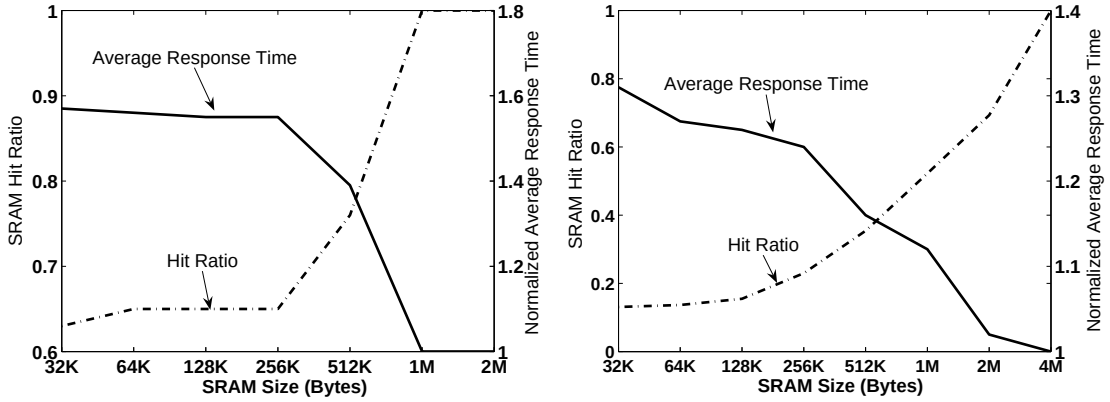
Figure 3.15 represents a same set of 100 consecutive requests being serviced

by FAST and DFTL for the Financial trace. This region illustrates transition from a sustainable I/O intensity (operational region) to a period of very intense I/Os (overloaded region) in the Financial trace. As is clearly visible, FAST suffers from higher garbage collection overhead and requests undergo higher latencies as compared to DFTL. Full merges cause a large number valid pages to be copied and the corresponding blocks to be erased. This results in higher device service time for the request undergoing these operations. This in turn causes the pending requests in the I/O driver queue to incur longer latencies. Thus, even though the device service time for these requests is small; the overall system response time increases. For example, in the top highlighted region in Figure 3.15, request A undergoes full merge resulting in very high device service time. While A is being serviced, the pending request B incurs high latency in the I/O driver queue (spike in queuing time for B) which increases its overall system response time. The same phenomenon is visible for requests C and D. Thus, full merges not only impact the current requests but also increase the overall service times for subsequent requests by increasing queuing delays. In sharp contrast, during the same period, DFTL is able to keep garbage collection overhead low and provide sustained improved performance to the requests as it does not incur any such costly full merge operations.

3.5.5 Impact of SRAM size for Mapping Entries

All the experiments in the preceding subsections were done by utilizing the bare minimum amount of SRAM necessary for implementing any state-of-the-art hybrid FTL scheme. Even with this constrained SRAM size, we have shown that DFTL outperforms the existing FTL schemes for most workloads. The presence of temporal locality in real workloads reduces the address-translation overhead considerably.

Figure 3.16 shows the impact of increased available SRAM size on DFTL. As seen, greater SRAM size improves the hit ratio, reducing the address translation overhead in DFTL, and thus improving flash device performance. As expected, with the SRAM size approaching the working set size (SRAM hit ratio reaches 100%), DFTL's performance becomes comparable to Baseline. Increasing SRAM



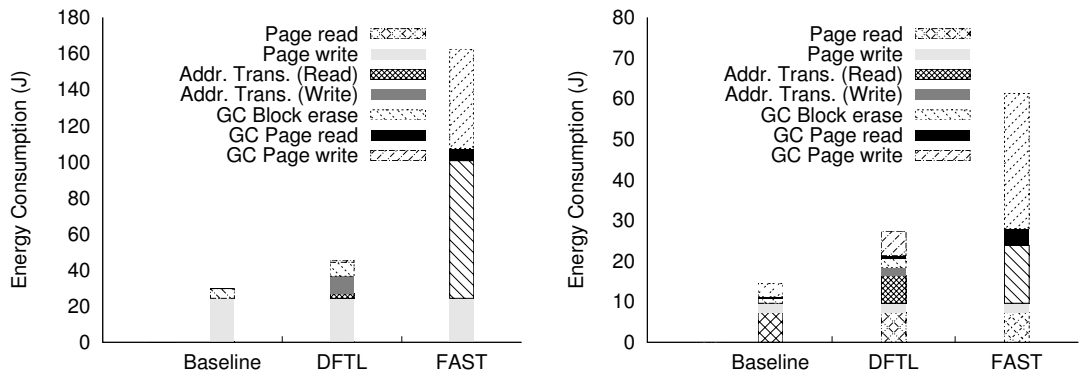
(a) Financial Trace (b) TPC-H Benchmark

Figure 3.16. Impact of SRAM size on DFTL. Response times have been normalized with respect to the Baseline FTL scheme. For both the Financial trace and TPC-H, there is performance improvement with increased SRAM hit-ratio. However, beyond the working-set size of workloads there is no benefit of additional SRAM for address translation. The 99% confidence intervals are very small and hence not shown.

size for holding address translations beyond the workload working-set size does not provide any tangible performance benefits. It would be more beneficial to utilize this extra SRAM for caching popular read requests, buffering writes, etc. than for storing unused address translations.

3.5.6 Analysis of Energy Consumption

Power consumption of the flash memory in the SSD may not be significant when compared to other components (CPU and Memory), but as shown in Table 5.4, erase operations consume significant power. Unlike individual read and write operations, erase operations have a greater impact on the overall SSD's energy consumption, and the number of erase operations for a given workload varies according to the current FTL scheme. Figure 3.17 shows the energy consumption by operations for different FTL schemes in the Financial and TPC-H traces. The Financial trace is mostly random-write-dominant, while TPC-H is read-dominant (see Table 5.3). Thus, the energy consumption for the Financial trace is much higher than that of TPC-H due to the power consumptions caused by GCs. DFTL requires additional page read and write operations due to mapping table entry misses in the memory, causing additional energy consumption in both traces. As expected,



(a) Financial Trace (b) TPC-H Benchmark

Figure 3.17. Energy consumption by different FTL schemes.

FAST FTL consumes significantly more energy than other FTL schemes due to more erase and write operations during GC.

In addition to power consumption by flash operations, the processor power consumption can be considerably high during GC. GC involves victim block searching overhead, which aims at finding the block with the least number of valid pages in order to reduce page copying overhead. Figure 3.18 shows the tradeoff between normalized average response time and the number of FTL search operations during GC for the Financial trace. Higher search operations decreases the response time while consuming more energy because (i) blocks with fewer valid pages require fewer copy operations, and (ii) the search operations induce energy consumption by processor and system bus usage. Thus, the energy consumption during GC can be reduced by balancing fewer search operations with a greater number of copy operations.Fewer search operations will slightly increase response time because an incomplete search may select blocks with more val id pages that must be copied.

On-board RAM is another considerable factor in the power consumption in the SSD. Since the page-based FTL requires more memory as compared to the block-based FTL, the idle power consumption of the additional memory will be larger. FAST maintains block-level mapping for data regions and page-level mapping for log regions; the on-board RAM's energy consumption is as close to that of the block-level FTL. DFTL requires the same memory as the block-level FTL; the idle power consumption is the same as that of the block-level FTL.

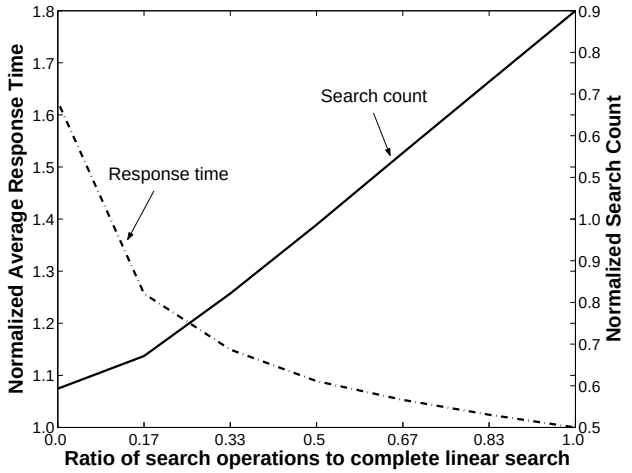
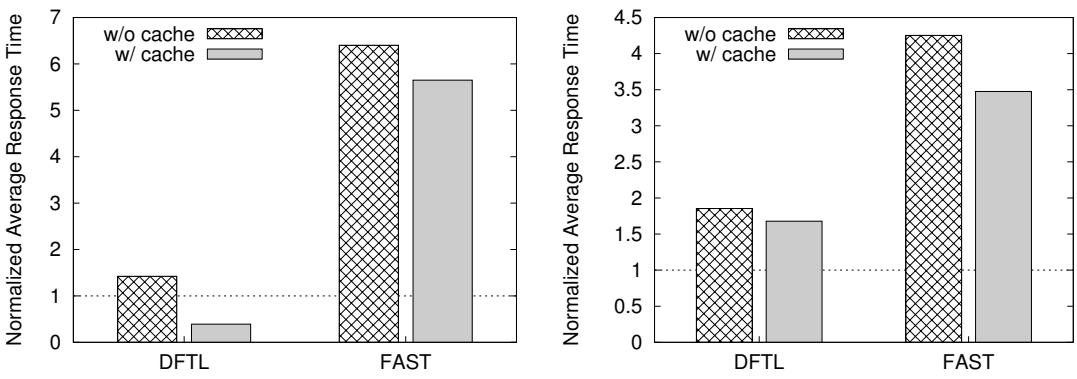


Figure 3.18. Tradeoff between performance and search operation energy consumption. This experiment has been conducted with DFTL for the Financial trace. We varied the number of search operations. Note that 0.0 on the X-axis means that the victim block is selected randomly without any search, and 1.0 means the victim block with the least number of invalid pages is selected after a complete linear search.

3.5.7 Impact of SSD Cache

All the experiments in the preceding subsections were done by ignoring the effect of SSD cache. However, it will be interesting to see the effect of SSD cache on FTL performance. As mentioned earlier, DFTL and FAST FTLs require less SRAM space for mapping entries compared to ideal page-based FTL. We consider 32GB SSD. The ideal page-based FTL needs 16MB SRAM to maintain all mapping entries whereas DFTL and FAST FTLs require only 32KB SRAM for the mapping entries. Thus, the DFTL and FAST FTLs can utilize the remaining huge memory space except for the memory space for mapping entries from entire 16MB SRAM. However, there is no available memory space used for data cache in the ideal page-based FTL. Figure 3.19 shows the impact of SSD cache on FTL performance when those remaining memory spaces are used for data cache. As expected, the cache improves flash device performance, reducing the amount of requests sent to the flash device (by about 85% for Financial trace and 58% for TPC-H). The results show normalized average response times with respect to baseline. DFTL scheme with cache even further outperforms the baseline, improving their response times by 72% in Financial trace, however, it is still worse than the baseline in TPC-



(a) Financial Trace (b) TPC-H Benchmark

Figure 3.19. Performance improvement of DFTL and FAST FTLs with device caches. Average Response times have been normalized with respect to baseline (Ideal Page-based FTL) performance without cache.

H. It is because our cache is a write-back cache which is highly optimized for write-dominant workloads (note that TPC-H is a read dominant workload) and the DFTL still suffers from extra overhead which is inevitable for the management of mapping entries. Thus, in these workloads, it would be better to increase the SRAM area for mapping tables while allocating less SRAM for data cache when DFTL scheme is used. FAST FTL scheme with cache also improves their response times in both workloads. However they are still worse than the baseline schemes because the FAST FTLs still suffer from expensive full merge operations.

3.6 Concluding Remarks

We argued that existing FTL schemes, all based on storing a mix of page-level and block-level mappings, exhibit poor performance for enterprise-scale workloads with significant random write patterns. We proposed a complete paradigm shift in the design of the FTL with our Demand-based Flash Translation Layer (DFTL) that selectively caches page-level address mappings. We developed and validated a flash simulation framework called FlashSim. Our experimental evaluation using a comprehensive flash simulator called FlashSim with realistic enterprise-scale workloads endorsed DFTL's efficacy for enterprise systems by demonstrating that DFTL offered (i) improved performance, (ii) reduced garbage collection overhead,

(iii) improved overload behavior and (iv) most importantly unlike existing hybrid FTLs is free from any tunable parameters. We have developed a simulator for SSDs and validated our simulator with real SSD devices by demonstrating behavioral similarity and compared performance results for various FTL schemes. As a representative example, a predominantly random write-dominant I/O trace from an OLTP application running at a large financial institution showed a 78% improvement in average response time due to a 3-fold reduction in garbage collection induced operations as compared to a state-of-the-art FTL scheme. For the well-known read-dominant TPC-H benchmark, despite introducing additional operations due to mapping misses in SRAM, DFTL improved response time by 56%. Moreover, our DFTL scheme even outperforms the ideal page-based FTL scheme, improving the response times by 72% in OLTP trace.

Chapter 4

Capacity Planning in HybridStore

4.1 Introduction

Hard disk drives (HDDs) have been the preferred media for data storage in enterprise-scale storage systems for several decades. The disk storage market totals approximately $34 billion annually and is continually on the rise [57]. Manufacturers of HDDs have been successful in ensuring sustained performance improvements while substantially bringing down the price-per-byte. During the past decade, the maximum internal data rate (IDR) for hard disks has witnessed a 20-fold increase resulting from improvements in rotational speeds (RPM) and storage densities; seek times have improved by a factor of 4 over the same period. However, there are several shortcomings inherent to HDDs that are becoming harder to overcome as we move into faster and denser design regimes. First, designers of HDDs are finding it increasingly difficult to further improve the RPM (and hence the IDR) due to problems of dealing with the resulting increase in power consumption and temperature [58, 59, 52, 60]. Second, any further improvement in storage density—another way to increase the IDR—is increasingly harder to achieve and requires significant technological breakthroughs such as perpendicular recording [61, 62, 63]. Third, and perhaps most serious, despite a variety of techniques employing caching, pre-fetching, scheduling, write-buffering, and those based on improving parallelism via replication (e.g., RAID), the mechanical movement involved in the operation of HDDs can severely limit the performance that hard disk based systems are able to offer to workloads with significant randomness and/or lack of locality. Specific

to our interest in this work, in an enterprise-scale system, *consolidation* (e.g., as proposed/explored in [64]) can result in the multiplexing of unrelated workloads imparting/exaggerating the randomness. Furthermore, such consolidated workloads are likely to exhibit degraded temporal and (more seriously for HDD-based systems) spatial locality, thereby potentially adversely affecting performance [64, 65].

Improvements in Flash Memory Technology - Alongside improvements in HDD technology, significant advances have also been made in various forms of solid-state memory such as NAND flash [66], magnetic RAM (MRAM) [3], phase-change memory (PRAM) [67], and Ferroelectric RAM (FRAM) [5]. Solid-state memory offers several advantages over hard disks: lower access latencies for random requests, lower power consumption, lack of noise, and higher robustness to vibrations and temperature. In particular, recent improvements in the design and performance of NAND flash memory (simply *flash* henceforth) have resulted in its becoming popular in many embedded and consumer devices. Small form-factor HDDs have already been replaced by flash in some consumer devices like music players, PDAs, digital cameras, etc. Flash has, however, only seen limited success in the enterprise-scale storage market [41]. Although (i) the aforementioned advances in flash technology and (ii) its dropping cost-per-byte [68] had led several storage experts to predict the inevitable demise of HDDs [69], flash has so far not been able to make inroads into the enterprise-scale storage market to the extent expected [41].

Solid-state Drives - Borrowing a few sentences from an excellent paper on this topic by Leventhal [41], *"The brunt of the effort to bring flash to primary storage has taken the form of solid-state disks (SSDs), flash memory packaged in hard-drive form factors and designed to supplant conventional drives. This technique is alluring because it requires no changes to software or other hardware components, but the cost of flash per gigabyte, while falling quickly, is still far more than hard drives. Only a small number of applications have performance needs that justify the expense".* [1] As evidence of this, major storage vendors producing flash-based large-scale storage systems such as RamSan-500 from Texas Memory Systems,

[1] We will use the terms *SSD* and *flash* interchangeably in the rest of this work.

Media	Access Time (μs)	Lifetime	Cost($/GB)
DRAM	0.9	N/A	125
SSD	(45) Read , (200) Write	10K-1M Erase Cycles	25
HDD	< 5500	MTTF=1.2Mhr	3

Table 4.1. Performance, lifetime, cost comparison among different storage media.

	MTron SSD	Western Digital HDD
Model	MSP 7000	Raptor X
Flash Type/RPM	SLC	10,000
Capacity	16GB	150GB
Interface	SATA 1.5GB	SATA 1.5GB

Table 4.2. Specification of the tested storage device.

Symmetrix DMX-4 from EMC, ioDrive from ioFusion, etc. are catering only a select class of applications such as large database servers rather than the general enterprise storage market.

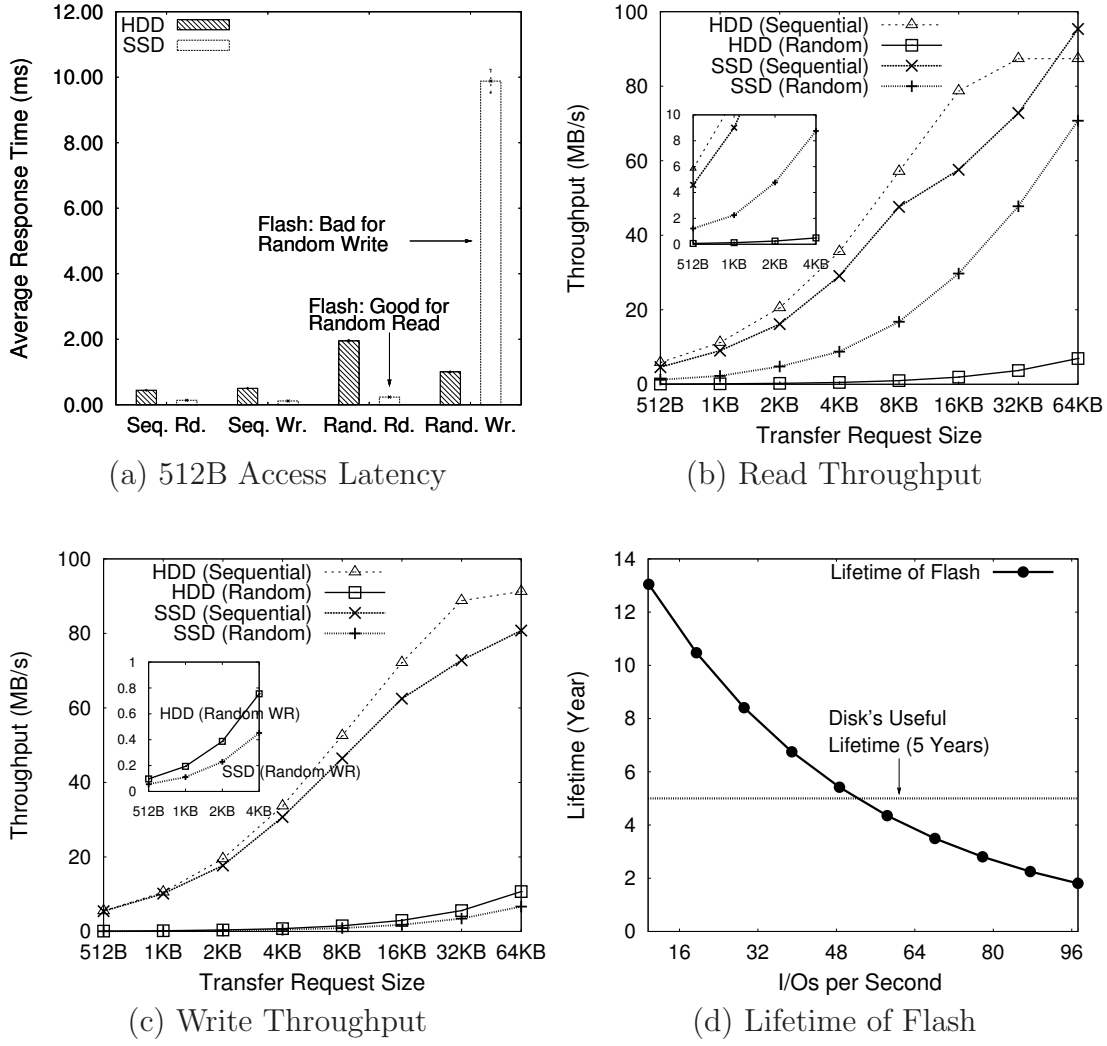
Table 4.1 (all values are based on [41]) presents a comparison of the performance, lifetime, and cost of representative HDDs, SSDs, and DRAM used in the enterprise. There are several important implications of how these properties compare with each other. Flash technology possesses a number of idiosyncrasies that have hindered the SSD from replacing HDD in the general enterprise market. First, it is evident that there exists a huge gap between the Cost/GB of HDDs and SSDs. [2] Second, unlike HDD or DRAM, SSDs possess a huge asymmetry between the speeds at which reads and writes may be performed. As a result, the throughput a SSD offers a write-dominant workload is lower than for a read-dominant workload. Third, flash technology restricts the locations on which writes may be performed—a flash location must be *erased* before it can be written—leading to the need for a garbage collector (GC) for/within an SSD. Certain workload characteristics (in particular, the presence of randomness) increase the fragmentation of data stored in flash memory, i.e., logically consecutive sectors become spread over physically non-consecutive blocks on flash. This exacerbates GC overheads, thereby significantly slowing down the SSD—even to an extent where it operates

[2] A similar gap exists between SSD and DRAM. Furthermore, it is projected to worsen in the near future: up to a factor of 13 by 2010 [70]. This rules out major changes in the role played by DRAM in future systems that employ SSDs. DRAM will continue to retain both of its important roles related to caching and buffering.

slower than a HDD! [43]. Furthermore, this slowdown is non-trivial to anticipate. A given set of random writes may themselves experience good throughput, but increase fragmentation, thereby degrading the performance of requests (read or write) arriving much later in future. Finally, to further complicate matters, unlike HDDs, SSDs have a life-time that is limited by the number of erases performed. Therefore, excessive writing to flash, while potentially useful for the overall performance of a flash-based storage system, limits its lifetime. This becomes an important concern in an enterprise-scale employing flash if its workload is write-intensive.

4.2 Motivation for HybridStore

From the above description, it should be clear that SSDs are fairly complex devices. Their peculiar properties related to cost, performance, and lifetime make it difficult for a storage system designer to neatly fit them between HDD and DRAM. To illustrate the complexity of the relationship between HDD and SSD, we perform a simple experiment using the devices described in Table 4.2. We send raw I/O requests to these actual devices and measure throughput and access latencies. Next, we use our MixedSim simulator (described in detail in Chapter 5.4.1) to estimate the useful lifetime of flash-based SSD in HybridStore. As has been observed in other recent research, under certain workload conditions, an SSD can perform worse than the HDD [43] and in certain SSDs, read throughput can be slower than write throughput for small random workload patterns [71, 72]. A look at Figures 4.1(a)-(c) provides an illustration of such behavior and calls for careful design to gainfully utilize them in conjunction with HDDs in the enterprise. The degrading lifetime with increased write-intensity, as shown in Figure 4.1(d), may result in premature replacement of these devices, adding to deployment, procurement, and administrative costs. Note that we have picked a lifetime of 5 years for a HDD just for illustrative purposes. An excellent study of the useful lifetimes of disks based on data from real enterprise-scale systems appears in a paper by Schroeder and Gibson [73]. Finally, the low throughput offered by SSDs to random write-dominated workloads (Figure 4.1(c)), which are frequently encountered in enterprise-scale systems [43], necessitates intelligent partitioning of data in such



(a) 512B Access Latency (b) Read Throughput

(c) Write Throughput (d) Lifetime of Flash

Figure 4.1. A comparison of the performance and lifetime characteristics of representative SSD and HDD. Although MTTFs for HDDs tend to be of the order of several decades, recent analysis has established that other factors (such as replacement with next, faster generation) implies a much shorter actual lifetime and hence we assume a nominal lifetime of 5 years in the enterprise. Note that Seq., Rand., Wr., and Rd. respectively denote Sequential, Random, Write, and Read. I/O request size in (d) is a page size (2KB). *Each bar in (a) is shown with 99% confidence interval.*

hybrid environments while ensuring that the management costs do not overwhelm the performance improvements. As already alluded to and explained in more detail in Chapter 5.3, compared to the HDD, an SSD require a longer history to be incorporated into a performance predictor. Modeling these characteristics is an

unexplored area and a significant part of our work as well as the foundation of the overall functioning of HybridStore.

Research Contributions - This work makes the following specific contributions.

- We propose HybridStore, a hybrid storage system containing HDDs and SSDs. Besides this hardware, HybridStore comprises: (i) a *capacity planner* (*MixPlan* henceforth) that makes long-term resource provisioning decisions for the expected workload; it is designed to optimize the cost of equipment that needs to be procured to meet desired performance and lifetime needs for the expected workload (Refer to Chapter 4) and (ii) a *dynamic controller* (*MixDyn* henceforth) whose goal is to operate the system in desirable performance/lifetime regimes in the face of deviations at short time-scales in workload from those anticipated by MixPlan (Refer to Chapter 5).

- We investigate how MixPlan can find the economical optimal configuration in a heterogeneous storage environment containing multiple HDDs and SSDs in this Chapter. We provide a general form of comprehensive methodology using a well-known technique for optimization problems, Linear Programming (LP). As an illustrative result, MixPlan is able to identify close to minimum SSD capacity needed to meet a specified performance goal for realistic workloads while ensuring similar performance as compared to a comparatively more over-provisioned system.

- We develop simple statistical models that MixPlan employs. Here, we consider a simplified hybrid storage system containing 1 HDD and 1SSD. The details are in Chapter 5, These models are used in conjunction with MixedSim (a simulator we have developed for HybridStore by enhancing DiskSim [49]) to validate the efficacy of MixPlan for a variety of well-regarded real-world storage traces. Moreover, we implement MixDyn in our simulator. In a HybridStore prototype, MixDyn would have two components: (a) an enhanced block device driver that employs online statistical performance and lifetime models for SSD (and a performance model for HDD) to dynamically partition incoming workload among the SSD and HDD, and (b) two algorithms within

the SSD controller (specifically, within the FTL layer) including reduction of fragmentation within the flash (fragmentation buster) and a novel concept of *adaptive wear-leveling*. As an illustrative result of our empirical evaluation of the efficacy of MixDyn, it is able to prolong the life of SSDs in HybridStore by about 33% in the face of an unexpected increase in I/O activity.

- Finally, we present ideas on how MixPlan and MixDyn could act in concert and present a preliminary validation and evaluation of all components of HybridStore in Chapter 5.

4.3 Overview of HybridStore

Figure 4.2 depicts the interaction between various components of HybridStore. In this study, we utilize a simplified model of an enterprise-scale storage system consisting of a single HDD and a single SSD connected to the same I/O bus. We will deal with more complex configurations consisting of RAID arrays etc. in future work. HybridStore consists of two major components: (i) a long-term resource provisioning tool (MixPlan) for system administrators to optimize the procurement or deployment or maintenance costs while adhering to the performance budgets (specific to workloads) and lifetime budgets, and (ii) a short term dynamic controller (MixDyn) which is part of the HybridStore internal structure. MixPlan utilizes statistical models for performance and lifetime to determine the optimal SSD capacity (we assume a static HDD size for our study) for different workloads.

4.4 Capacity Planning: MixPlan

Given the large price gap between flash-based SSDs and HDDs, it is essential to determine appropriate capacities of these devices for the workload the system expects to support. We define this process of determining the right capacities of devices in HybridStore as *capacity planning*. Our goal is to determine the right number of SSDs and HDDs which need to be deployed in a heterogeneous storage environment. In this section, we provide a general form of comprehensive methodology using a well-known technique for optimization problems, Linear Pro-

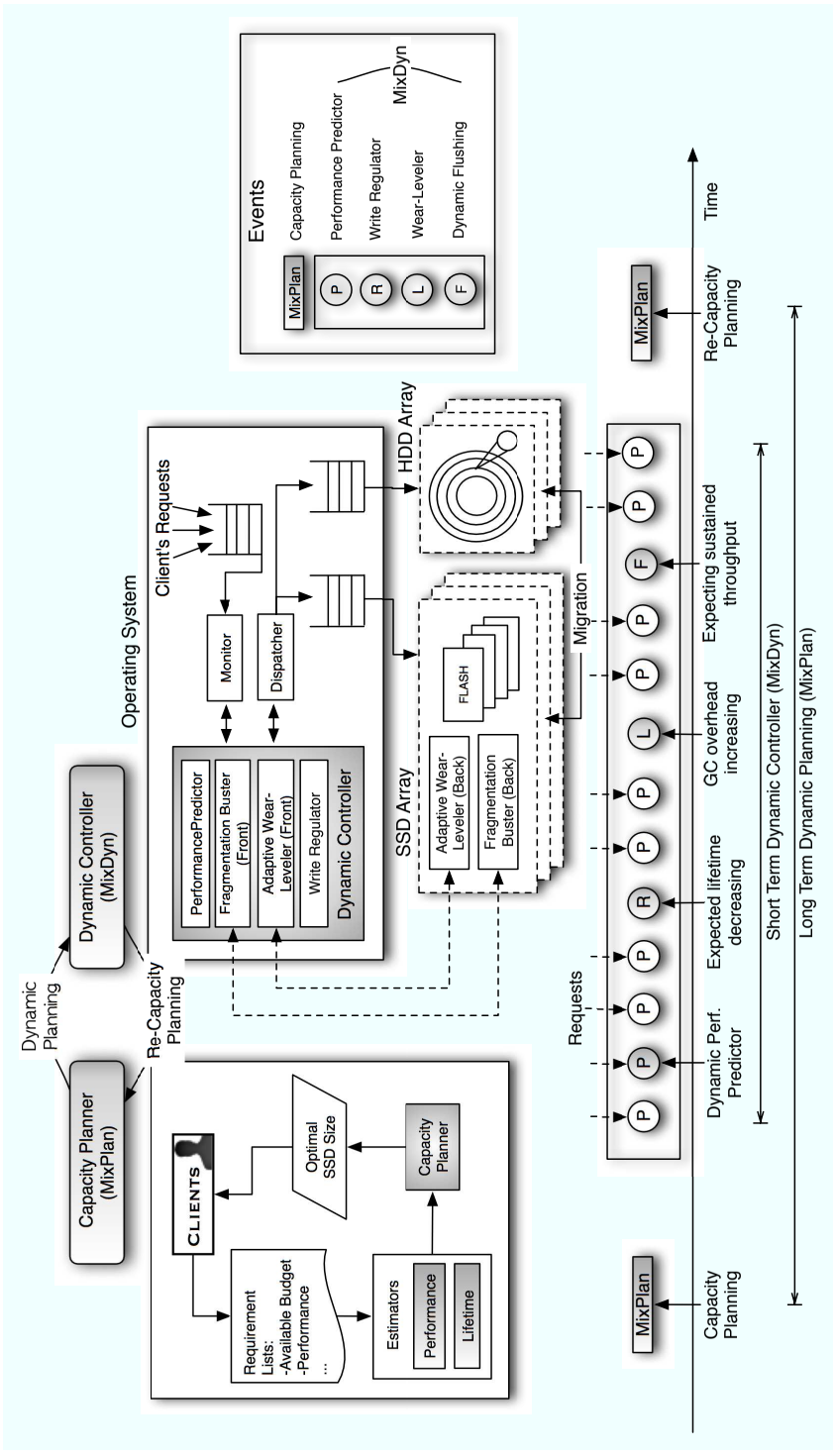


Figure 4.2. Depiction of various components of HybridStore and how they interact.

$$\textbf{\textit{Minimize}}\ Cost_{HybridStore}$$

$$\textbf{\textit{Subject to the constraints}} \begin{cases} Perf_{Hybridstore} \geq Perf_{Budget} \\ Life_{Hybridstore} \geq Life_{Budget} \end{cases}$$

$$\textbf{\textit{Where}}\ Cost_{HybridStore} = Cost_{Installation} + Cost_{Recurring}$$

gramming (LP).

4.4.1 Problem Formulation

It is challenging to find the most economical combination of SSDs and HDDs because the performance of devices varies a lot according to device's manufacturer and workload patterns [74]. Especially, the performance and lifetime of flash-based SSDs is highly dependent on not only the workload characteristics but also the internal intricacies of flash such as design of FTL, efficiency of GC, amount of on-board DRAM etc. In such an environment, storage system architects are faced with tough questions about whether to incorporate flash based SSDs in their systems, in what quantity (capacity), at which layers in the storage hierarchy. This makes their job really challenging and mandates the design of a robust capacity planner (MixPlan) tool which can equip them with a means to design a highly efficient storage system while meeting their requirements. We formulate our capacity planning problem as a means of minimizing the cost of acquiring/installing HybridStore while meeting the workload-specified performance ($Perf_{Budget}$) and useful lifetime budget ($Life_{Budget}$). Let $Cost_{Installation}$ indicate the installing cost of devices. Apart from these, costs associated with power consumption, thermal consumption (cooling), other maintenance and management activity form the recurring costs denoted by $Cost_{Recurring}$. However, information in the academic domain about the management/maintenance costs of these devices (- HDDs and SSDs) is still sparse and inconclusive. Furthermore, management costs vary with

legal contracts and are highly subjective. Hence, we only consider electricity cost of operation due to power consumption and thermal management (cooling) as recurring cost in this study. Thus, the total HybridStore cost $Cost_{HybridStore}$ is the sum of these individual costs. Our optimization problems are summarized as follows:

Storage Models - Figure 4.3 shows an example of a storage system model employing different device types. The objective of capacity planning is to minimize the cost of HybridStore (deployment, management and maintenance) while meeting the service level agreements(SLAs). The constraints can vary from guaranteeing some minimum performance requirements to reducing management and re-deployment costs, ensuring system reliability etc. For the purpose of our study, we try and minimize the deployment and operation cost (in terms of $) subject to a combination of both performance and re-deployment constraints lifetime of flash memory. We use IOPS as a metric of HybridStore's performance and term this metric as the system's *Performance Budget*. In addition, we need to consider lifetime issues in the flash because the blocks in SSDs become unreliable beyond 10K-1M erase cycles [8]. This poses a significant challenge for a system administrator whose objective is to keep system re-deployment frequency and costs under control. We capture these objectives in terms of a *Lifetime Budget* (years) for the system, which is the time between successive capacity planning decisions and equipment procurement/installation.

Multiple steps to economical storage configuration - In order to build cost-effective storage system called HybridStore, we need a framework (we call it MixPlan henceforth) to satisfy the given workload requirements using known device characteristics. As we will describe later, we can extract workload requirements (space or bandwidth requirement) by analyzing their IO traces. And the device characteristics can can be obtained not only from their data sheets but also from performance tests. Figure 4.4 shows the process of decision-making for cost-effective capacity planning using MixPlan. The trace analyzer of MixPlan extracts the workload characteristics (utilization, peak arrival rate of the requests, read/write ratio, average request size) and generates the predicted workload characteristics for the next few years (3-5 years). Once these inputs are ready, Mix-

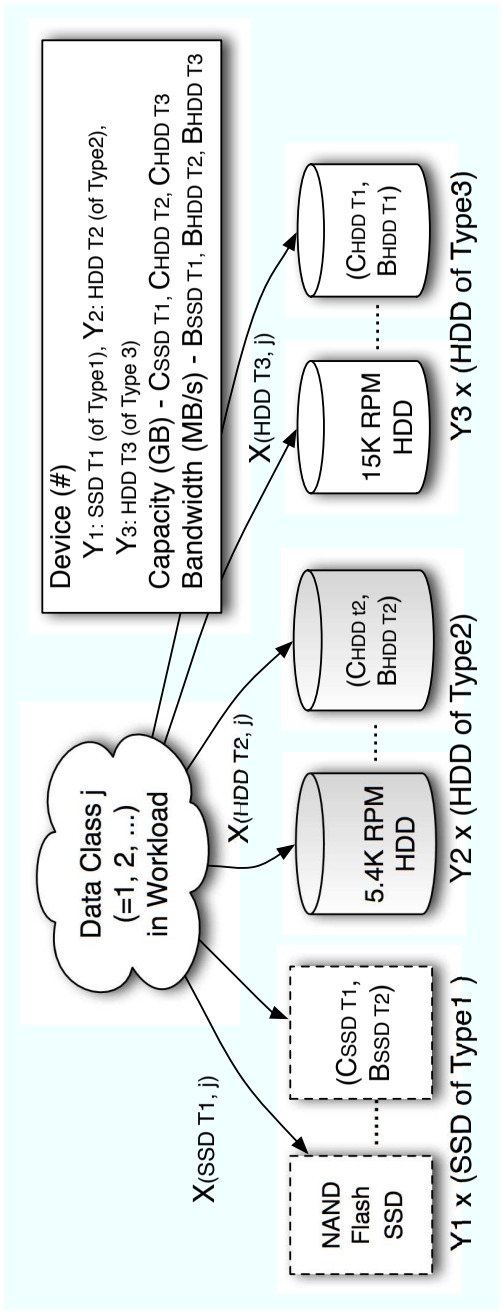


Figure 4.3. Storage System for HybridStore. j: jth data class, $X_{SSD\ T1,j}$: jth data class on Y_1 devices of SSD type1, $X_{HDD\ T2,j}$: jth data class on Y_2 devices of HDD type2, $X_{HDD\ T3,j}$: jth data class on Y_3 devices of HDD type3

Plan finds the most economical storage configuration as well as data placement between devices that minimizes total cost (including storage installment and recurring costs) while meeting the afore-mentioned constraints.

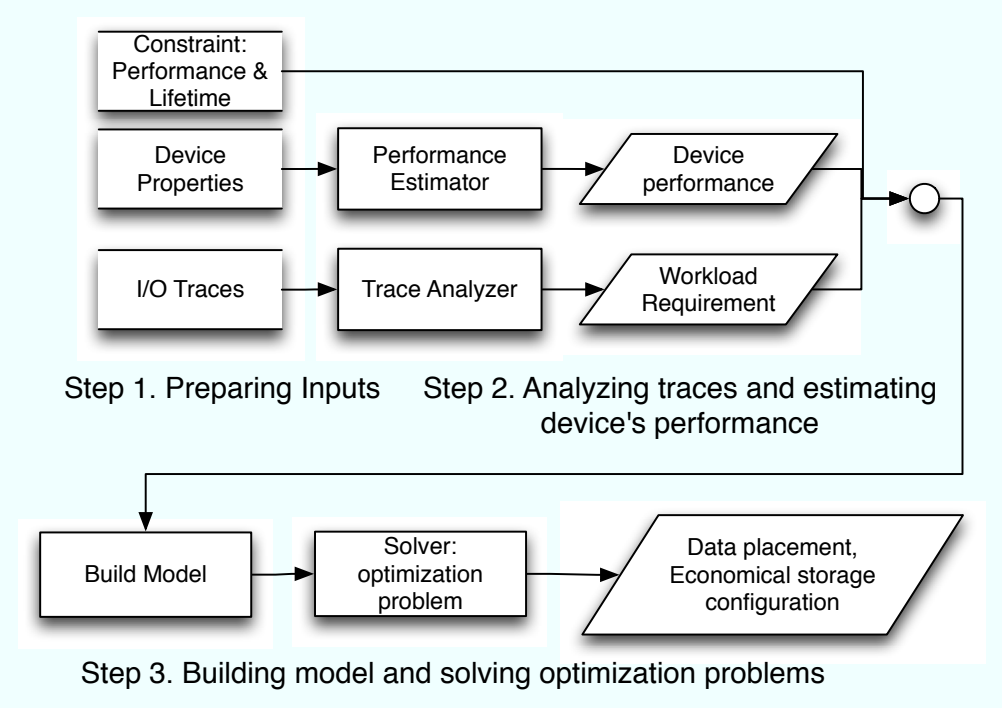


Figure 4.4. Decision process for capacity planning by MixPlan

4.4.2 Finding optimal solution by MixPlan

We describe a tool which finds the most cost-effective storage configuration using available devices for the provisioned workloads by reducing our optimization problem to a LP problem.

Declaration of Variables - Device type i (i=1, 2, 3, ..., I) has capacity C_i and bandwidth B_i. Data class j (j=1, 2, 3, ..., J) has size S_j and frequency F_j. U_i is the utilization of the device of type i. All the variables declared in our formulation are as follows:

$$
\begin{aligned}
I \ \& \ J &= \text{number of device types or number of data classes respectively} \\
C_i &= \text{capacity of device type } i \\
U_i &= \text{utilization of device type } i \\
B_i &= \text{maximum bandwidth of device type } i \\
S_j &= \text{size of data class } j
\end{aligned}
$$

$$F_i = \text{frequency of data class } i$$
$$W_{ij} = \text{weight factor for bandwidth of data class } j \text{ on } y_i \text{ devices of type } i$$

Decision Variables - x_{ij} is data of class j on y_i devices of type i. y_i is the number of devices of type i.

$$x_{ij} = \text{data of class } j \text{ on } y_i \text{ devices of type } i$$
$$y_i = \text{number of devices of type } i$$

Installation Cost of Storage - As described earlier, we consider installation cost and electricity cost for the total cost of the storage systems. Given the properties of I different types of devices, the overall installation cost of storage systems is highly dependent on the numbers of each device type $i \in I$, and its individual device cost which can be calculated ($= D_\$ \times C_i$) where $D_\$$ is the capacity cost per unit size:

$$Cost_{Installation} = \sum_{i=1}^{k} D_\$(i) \times C_i \times y(i) \tag{4.1}$$

Electricity Cost of Operation and Cooling Cost - Given the electricity cost per time ($= K_\$$) and the power consumption of device type i ($= P(i)$), the energy consumption of overall storage system ($= E$) over time followed by the overall electricity cost of operation by the energy consumption can be calculated as: Since the cooling cost is equal to cost of power consumption [75], we estimate the cooling cost is the same as the electricity:

$$Cost_{Recurring} = K_\$ \times E_{Operation} \tag{4.2}$$
$$(\ where\ E_{Operation} = \sum_{i=1}^{I} y(i) \times \int_t P(i)\ dt\ )$$

Objective Function - Putting these together, we get the dollar cost of installing storage system and its operation. The objective function to minimize is: Note that one could choose to minimize this objective function by reducing the number of devices. But, this can result in violating capacity requirement or bandwidth

requirement that the storages need to support. Consequently, when minimizing the above objective function, we need to satisfy the following constraints.

$$\begin{aligned} Cost_{HybridStore} &= Cost_{Installation} + Cost_{Recurring} \\ &= (\sum_{i=1}^{I} y(i)) \times D_{\$}(i) \times C_i \\ &\quad +2 \times (K_{\$} \times \sum_{i=1}^{I} y(i) \times \int_t P(i)dt)) \end{aligned} \tag{4.3}$$

Constraints - The constraints are related to (i) data groups, (ii) device's capacity, (iii) device's bandwidth, and (iv) lifetime of the SSD.

$$\sum_i x_{ij} = S_j, \ (\forall \ j \in J) \tag{4.4}$$

$$\sum_j x_{ij} \leq (U_i \times C_i) \times y_i, \ (\forall \ i \in I) \tag{4.5}$$

$$F_j \times \frac{x_{ij}}{S_j} \leq B_{ij} \times y_i, \ (\forall \ i \in I, \forall \ j \in J) \tag{4.6}$$

$$Lifetime \ (i, x) \leq \textit{Useful Lifetime of HDD } (i \in \textit{Flash based SSDs}) \tag{4.7}$$

Equation 4.4 constraints that all data in the data class j partitioned between all types of devices should be the same as the size of data class j (S_j). Equation 4.5 reflects the capacity constraint of devices of type i i.e. the total amount of data assigned to devices of type i should not exceed the total capacity that they provide. Equation 4.6 is the bandwidth constraint of devices of type i for x_{ij} i.e. the bandwidth request should not exceed total bandwidth that they can provide. Equation 4.7 is the lifetime constraint of device of type i for the bandwidth request, the rate of which can be easily calculated from data classes. x is the write bandwidth. The estimated lifetime of Flash based SSDs should not be less than the useful lifetime of HDDs (3-5 years) [73]. The expected NAND flash lifetime can be calculated as follows:

$$Expected \ lifetime = \frac{Size \ of \ NAND \ flash \ \times \# \ of \ erase \ cycles}{bytes \ written \ per \ day} \tag{4.8}$$

4.5 Evaluation for MixPlan

We evaluate MixPlan using a well-regarded Integer Linear Programming (ILP) solver (*lp_solve 5.5.0.14*) written in C [76]. *lp_solve* is a free (GNU licensed) linear programming solver based on simplex method. Also, we have written the trace analyzer in C. The source codes are less than 500 lines of code. The Solver execution time is extremely short (in seconds), however the analyzer execution time is dependent on the trace size and can run into minutes for large traces.

Workload		Total Volume Size (GB)			Bandwidth (KB/s)	Request Size (KB)	Read (%)
		Hot	Cold	Total			
Large-scale	Financial* [53]	114	2,649	2,763	9,220	3.42	23
	TPC-H* [55]	87	813	900	16,050	25.6	80
Small-scale [77]	/usr	244	1,115	1,359	36.7	45.6	93
	/proj	193	1,690	1,883	30.3	44.4	91
	/print	46	384	430	2.1	13.3	9
	/hm	2	37	39	0.2	7.9	38
	/resproj	1	79	80	0.1	8.9	7
	/proxy	3	85	88	14.8	11.8	61
	/src1	124	431	555	22.5	46.5	95
	/src2	1	100	101	0.06	6.62	18
	/webstg	3	105	108	0.15	8.55	12
	/term	1	21	22	0.12	10.05	31
	/websql	60	381	441	9.0	45.0	94
	/media	13	496	509	0.13	5.59	57
	/webdev	1	101	102	0.08	18.07	20

Table 4.3. Description of Enterprise-scale Workloads Used. * denotes scaled trace.

4.5.1 Experimental Setup and Workloads

Workloads We employ the write-dominant I/O traces of an OLTP application running at a financial institution [53] made available by the Storage Performance Council (SPC), henceforth referred to as the *Financial trace*. TPC-H [55] is an ad-hoc, decision-support read dominant benchmark (OLAP workload) examining large volumes of data to execute complex database queries. The summary of these traces used is shown in Table 4.3. We considered four representative storage devices: high-speed HDD, low-speed HDD, Flash based SSD, and FusionIO's ioDrive. Detailed description of these devices is given in Table 4.4.

Parameter Settings in MixPlan Device utilization ratio (ratio of amount of actual data stored in the device to its entire storage capacity) needs to be properly set in capacity planning. We set the expected utilization ratio of flash device as 70%. This is based on the observation of Kgil et al. that garbage collection

Type	RPM	Capacity (GB)	Price ($/GB)	IDR (MB/s)		Power (W)		Erase Cycles
				Read	Write	Device	Supplement	
SSD [78]	-	80	4.25	220	160	1.0	1.8	10K-1M
High-end HDD [79]	15K	300	1.50	128	128	9.19	17.0	-
Low-end HDD [80]	5.4K	1K	0.12	42.66	42.66	5.8	10.7	-

Table 4.4. Storage device characteristics. Each device can consumes extra energy (Watts/drive) to each device [2] when they are built in a fashion of device array. Note that 10 cents per kilowatt-hour (kWh) is used to estimate electricity cost in our evaluation.

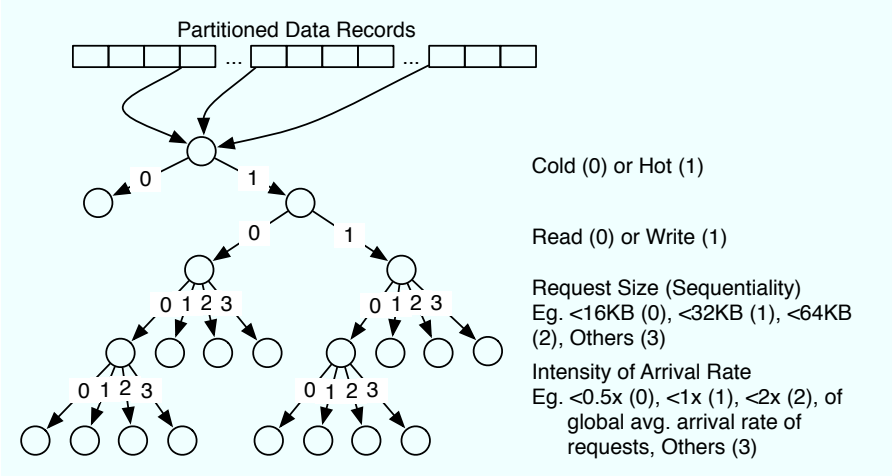


Figure 4.5. Hierarchical data classification.

overhead in flash dramatically increases if the utilization exceeds 70% [40]. Also, we have a similar observation in experiment using our flash simulator. The expected disk utilization is set as 50% to provide sufficient storage space. Moreover, we need to consider device use duration in order to consider the recurring cost of the storage system. We used this period as 5 years for our evaluation. Note that 10 cents per kilowatt-hour (kWh) is used to estimate electricity cost in our evaluation.

Metrics The main metrics used in our study include (i) storage installment cost (in $), recurring cost (in $) including operation cost of power consumption and cooling cost, (iii) the number of each type of devices, and (iv) the amount of each partitioned data class.

4.5.2 Hierarchical Data Classification

We partition the entire logical address spaces by a certain size of record (logically consecutive area, 1MB in size). The record size of this is set by considering read-ahead effect of hard disk drive for spatial locality. The entire logical address space is partitioned into *Hot* and *Cold* regions where hot regions are sets of data records whose I/O Operations Per Second (IOPS) is greater than 1 while the rests of them form cold regions. They are further classified by read ratio, request size, and intensity of arrival rate. Since many workloads show time-varying load due to diurnal effect, average values of the arrival rates are not adequate for classification purposes. Thus, we use peak arrival rates of the data records. We use 95th percentile of peak bandwidth (measured every second) of data class as bandwidth requirement. Figure 4.5 shows our data classification which has been consideration for our evaluation. We have experimented with 17 data classes (without consideration of data classification by intensity of arrival rate of the requests) versus 33 data classes. We found considering 33 data classes enables MixPlan to find more economically optimal solution than 17 data classes. The number of data classes can be optimized by reduction and merging techniques.

4.5.3 Bandwidth of Devices

Bandwidth of each device is dependent on the input data category. Hence, we need to consider the variance of device bandwidth with each data category. In order to do this, we simulated the performance of the HDDs (high-end and low-end HDDs) to each data category. We estimated these performance values using average seek time and average rotational delay for each data category. We used a simple performance model for HDDs because there already exists a lot of research [81] in this regard and this is not the major focus of our study. We leave use of sophisticated device performance models as part of our future work. Unlike HDDs, performance of SSDs is highly incumbent on its internal mechanisms such as FTL, GC etc. The non-disclosure of this important information by major SSD manufacturers makes developing performance models for SSDs a big challenge. Thus, we have developed a performance model for the SSD using conditional regression model. Our performance model for the SSD is based on real measurement values of the

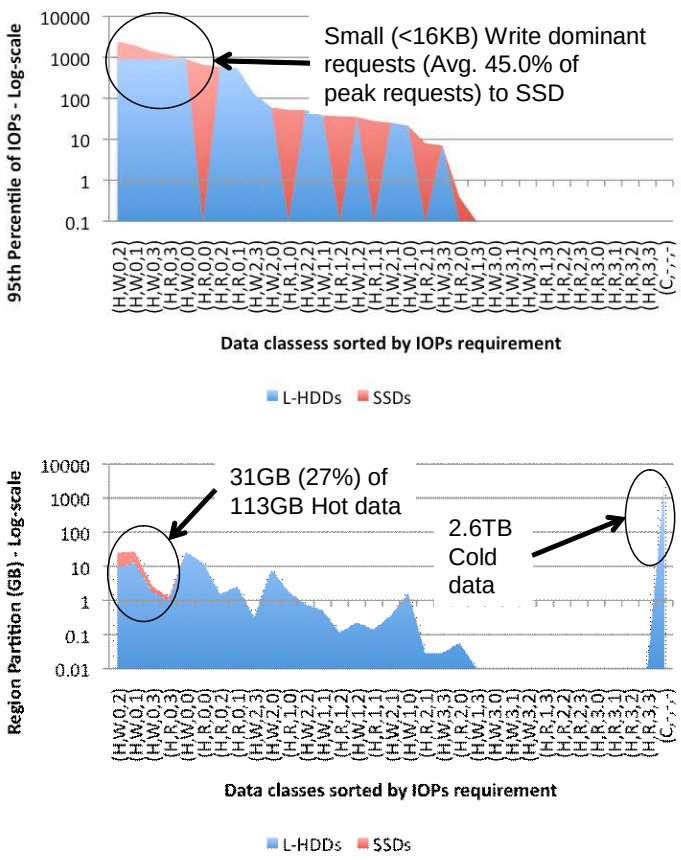


Figure 4.6. Data partitioning in hybrid systems by MixPlan for the Financial trace. Each tuple at x-axis shows hot or cold, read or write, request size, and intensity of arrival rate of data class. In the tuple, "H" and "C" respectively denote hot and cold data while"R" and "W" respectively denote read and write data. Request size and intensity of arrival rate of the data class can be represented by the numbers, 0-3. The bigger value denotes large request size and higher intensity of request's arrival rate.

SSD performance values available in [82]. The performance model for the ioDrive of FusionIO has been developed by linear regression technique with its read and write bandwidth values of the data sheet.

4.5.4 Economical Storage Configuration

4.5.4.1 Can SSDs replace HDDs?

In [44], Agrawala et al. observe that replacing HDDs with SSDs is not a economically viable solution. We try and examine if SSDs can actually replace HDDs at

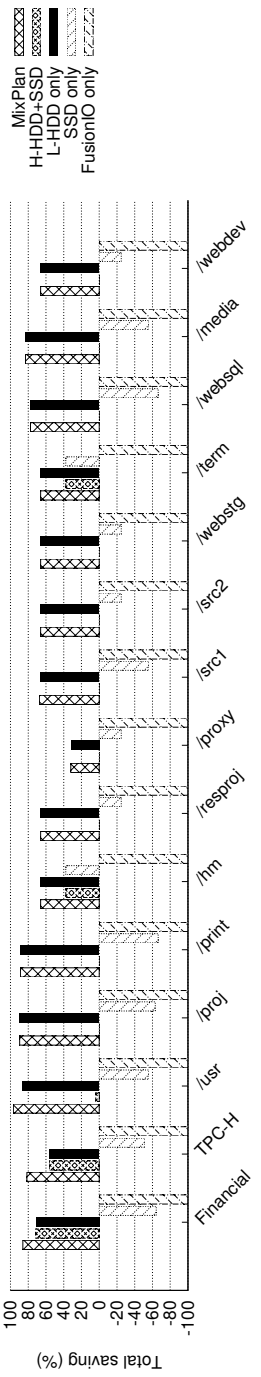


Figure 4.7. Total cost savings (%) compared to high-end HDD only system. Note that the values of y-axis has been cut at -100 (%). However, the values can be worse than this lower bound (-100(%)).

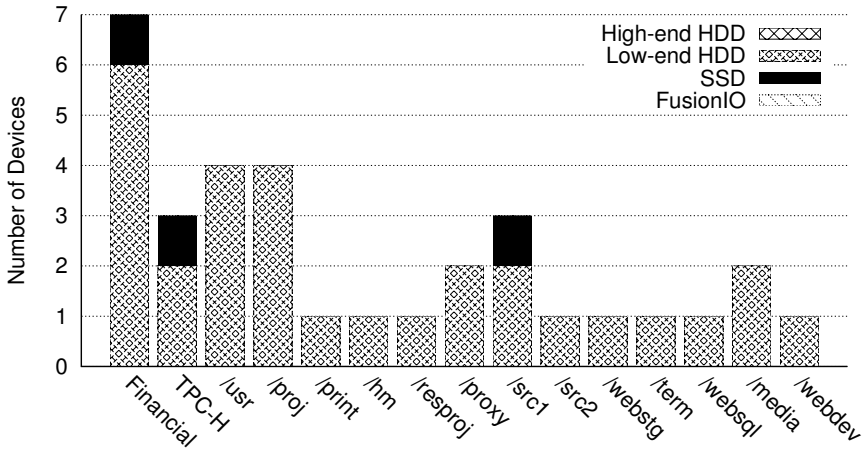


Figure 4.8. Economical storage configurations

current price points and if not, then at what price points does it become viable to use a SSD only storage system. In this experiment, we see that MixPlan can find the most economic storage composition for a workload, given the available device characteristics and their prices. From the results in Figure 4.8, we observe that low-end HDD can replace high-end HDD in MSR traces except in "src1" trace under current price ratios of devices. It is because I/O bandwidth requirements of these traces to each device are much lower than those of high-end HDDs. However, from the results of Financial trace and TPC-H, we observe that 6 low-end HDDs and 1 SSD for Financial trace 2 low-end HDDs and 1 SSD for TPC-H need to be employed. Figure 4.6 shows what data, and how much of them needs to be partitioned in the hybrid system employing SSDs and low-end HDDs for Financial traces.

4.5.4.2 Efficiency of Hybrid System

We compared the total cost of hybrid system with those of other storage configurations. Figure 4.7 shows total cost savings (%) of hybrid system compared to high-end HDD only system. We assumed that the baseline system is high-end only system. From Figure 4.7, we observe that the storage configuration by MixPlan economically outperforms than others. For example, in overall, the configuration by the MixPlan can save total cost of storage system by average 73% to high-end

HDD only system. Moreover, we can observe that SSD can replace high-end HDD in "/hm" and "/term" traces. For "/hm" trace, SSD solution requires $340 installing cost and $12.4 recurring cost while high-end HDD solution requires $450 installing cost and $114.7 recurring cost.

4.5.5 Impact of variable factors

Workload characteristics are known to show deviation from their normal behavior and with greater adoption of flash technology, the prices of SSDs are also coming down. In this subsection, we examine how it finds the most economical combination of devices, while dealing with variation in device prices and workload characteristics. Moreover, we investigate how does the recurring cost of devices affect the decisions by MixPlan.

4.5.5.1 What if price fluctuation of device?

Figure 4.9(a) show the results when the recurring cost of devices is not taken into account for capacity planning. To allow price fluctuation, we varied the price of devices with respect to low-end HDD. Since low-end HDD HDDs are much cheaper than others, we set the price of low-end HDD HDD as baseline and instead, we relatively lowered the prices of high-end HDD, SSD, and FusionIO's device. In the result of Financial from Figure 4.9(a), for low intensity Financial trace workload (1x) MixPlan outputs similar combination of devices (1 SSD and 6 low-end HDD HDDs) as was demonstrated in earlier experiments. This suggests that the price variation is not sufficient to influence the decision. We also conducted experiments with varying arrival rates to change workload intensity. Interestingly, as shown from the results of TPC-H in Figure 4.9(a), with times more intense workloads (8x), MixPlan suggests employing a combination of all 3 types of devices 1 FusionIO's device, 1 SSD, and 2 low-end HDD.

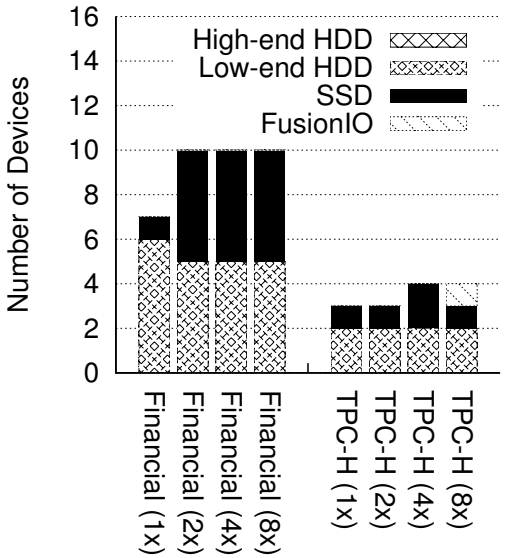
4.5.5.2 What if not considering recurring cost?

We include the recurring cost of devices into our experiments and study its impact on MixPlan's decision making. Comparing Figure 4.9(a) which includes the recurring cost along with the installation costs with Figure 4.9(b) which only includes

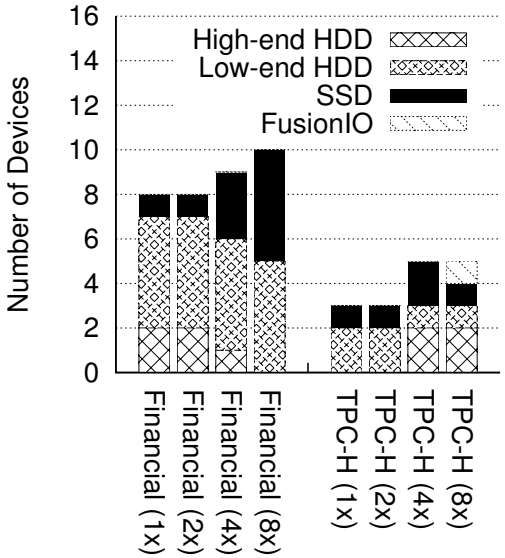
the installation cost clearly demonstrates that recurring cost can play a significant role in the capacity planning process. For both Financial and TPC-H, we find that employing high-end HDDs becomes economically inviable when the recurring costs are considered. This is primarily because of the high power consumption and cooling costs involved with these devices.

4.6 Concluding Remarks

This research was based on the needs for finding right capacities of SSDs and HDDs in hybrid storage systems. We also provide a general form of comprehensive methodology using a well-known technique for optimization problems, Linear Programming (LP). Based on this technique, we developed an capacity planner, called *MixPlan* that finds the most economically efficient storage configuration while meeting the performance and lifetime requirements of SSDs and HDDs. As an illustrative result, we showed that MixPlan is able to identify close to minimum SSD capacity needed to meet a specified performance goal for a realistic workloads while ensuring similar performance as compared to a comparatively more over-provisioned system.



(a) Considering Installation and Recurring Cost



(b) Considering Installation Cost Only

Figure 4.9. When the price gap between devices decreases for Financial trace. The new price ratio of each device : (high-end, low-end, SSD, FusionIO's device) = (1.5, 1, 4, 8).

Chapter 5

Dynamic Management in HybridStore

5.1 Introduction

Workloads are known to exhibit variation from their predicted behavior. In such circumstances, capacity planning alone is not sufficient to meet the lifetime and performance budgets. With higher intensity of writes, the garbage collector is invoked more often; thus degrading the system's performance. Moreover, it results in higher number of block erases in flash, reducing the flash lifetime. Thus, we require additional sophisticated data partitioning mechanisms which can dynamically adapt to these changing workload environments. In the next Chapter, we describe some techniques employed by our dynamic controller (MixDyn) to meet the various budgets and thus work in synchronization with MixPlan. We have established the need for a fine-grained control mechanism which should be able to manage the requests and hence ensure sustained throughput from the storage system which is able to meet our lifetime and performance budgets.

In this chapter, we see how MixPlan and MixDyn could act in concert and present a evaluation of all components of HybridStore. Before investigating how they act in concert, we try to model performance and lifetime of the SSD to not only see if a black-box modeling approach for SSD is possible, but also motivate the needs for dynamic controller in HybridStore. We used our FlashSim which has

been introduced in Chapter 3 for model developments and evaluations.

5.2 Modeling Performance and Lifetime of Flash Memory

We employ a "black-box" modeling approach for estimating a given SSD's useful lifetime and performance. Our model makes no assumptions about the inner configurations (such as FTL employed, SRAM cache size etc.). We do find its efficacy varies depending on the internals of the SSD. For example, the predictor performs better with our page-based like FTL than other state-of-the-art hybrid FTLs. For this purpose, we need to identify statistically significant workload characteristics that impact the SSD's lifetime and performance. Performance is directly impacted by data fragmentation caused by random writes which invoke costly GC operations. Moreover, high write intensity increases the number of erase operations required to reclaim invalid space on flash, thus reducing lifetime of blocks. Based on these observations, we consider the following workload characteristics as significant independent variables: (i) average read/write ratio, (ii) spatial locality captured in the form of average sequentiality among requests, (iii) average request inter-arrival time, (iv) average request size, and (v) flash utilization defined as the ratio of the working set size to the total flash size. We install probes in MixedSim to capture data relating to the above parameters. Our interest is flash. We now, do not change disk. Given a fixed disk configurations, we are interested in flash configurations and models in the hybrid environment.

We consider the following properties on flash. First, operational behavior of flash, especially for writes is highly dependent on FTL design and implementation (which has been explained in Chapter 2. Out-of-place update properties in flash imposes the intricacy of FTL design. Second, the lifetime of flash memory is highly dependent on frequency of erase operations occurring by garbage collection followed by write rates to flash. Third, flash capacity and its utilization impacts on performance as well as lifetime on flash. We describes the models that we use in our study for capturing capacity, average response time, and lifetime of flash memory for given workloads and flash utilization.

5.2.1 Regression Based Modeling

Using multiple linear regression, we first find significant predictor variables which affect the variables being predicted: (i) average system response time (ms) for performance budget, (ii) average block erase rate (erases/second) for lifetime budget. The underlying assumption on this linear regression modeling approach is an assumption of linearity. It is assumed that the relationship between variables is linear. Moreover, there is a normality assumption that the residuals (predicted minus observed values) are distributed normally (i.e., follow the normal distribution). We start with the general approach in multiple regression of finding significant predictor variables while plugging in as many predictor variables as we can think of. In order to avoid multicollinearity problems, we also perform correlation analysis on predictor variables to ensure that they are all independent variables.

Performance Model for SSD - We use average I/O system response time (R_{avg}) as a predictor of flash performance. I/O system response time represents the time interval between the issuance of request to the SSD by the I/O driver and its completion notification to the driver. It includes queuing delay, bus delay and controller overhead in the device. We first experimented with a multiple linear regression based model. Upon finding this model unsatisfactory, we moved towards a slightly more complicated multiple log-linear model [81]. It can be represented as

$$\log(R_{avg}) = a_0 + \sum_{i=1}^{n} a_i \cdot W_{avg}(i) + \epsilon_1 \tag{5.1}$$

where (W_{avg}) represents the average of a particular workload characteristic selected from a set of n parameters discussed earlier (Section 5.2) and ϵ_1 is a small error. The coefficients (a_0, a_1, ... , a_n) are estimated during the learning phase of the experiments.

Lifetime Model for SSD - Erase rate (block erases per second) denoted by E_{avg}, represents the lifetime of a flash device since each block typically has a life of about 10K-1M erase cycles [8]. As in the case of performance modeling, we start by fitting a multiple linear regression model. Again, we observe that a multiple

log-linear regression technique, similar to the one used for performance budget is able to model the lifetime budget. The similarity between the two models arises from the fact that higher response times are a function of garbage collection which require block erases and hence impact lifetime. Thus, the lifetime model can be represented as

$$\log(E_{avg}) = b_0 + \sum_{i=1}^{n} b_i \cdot W_{avg}(i) + \epsilon_2 \tag{5.2}$$

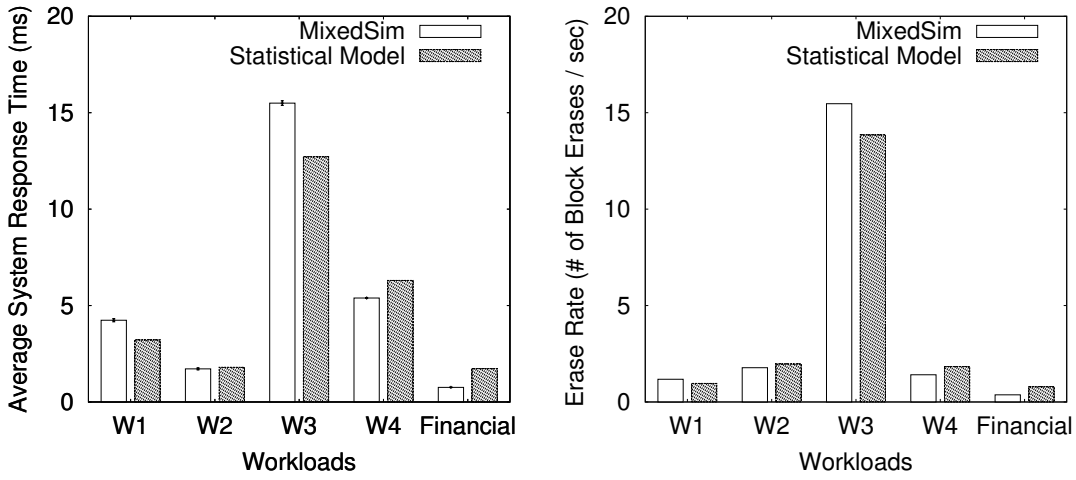
where (W_{avg}) represents the average of a particular workload characteristic selected from a set of n parameters discussed earlier (Section 5.2) and ϵ_2 is a small error. The coefficients (b_0, b_1, ... , b_n) are estimated during the learning phase of the experiments.

5.2.2 Validation

In this sub-section, we validate our models by comparing against the actual values measured using MixedSim. We generate a large number of synthetic traces by varying workload characteristics described in Section 5.2 to train the models and randomly select 900 of these traces to form our training set. The adjusted R-square [1] is found to be around 90% for both the multiple log-linear models [81]. The average error rate is about 25% for the training set.

We validate our performance and lifetime models by comparing their results with the corresponding values measured using MixedSim. Table 5.1 shows the salient characteristics of some of the synthetic and real workloads. We choose write-only synthetic traces for validation since flash performs very well for read dominant workloads. Moreover, lifetime is not an issue for such workloads since they encounter very few erase operations. For W2, the error in the performance model is only about 4% whereas it rises to about 21% for W3 which has the highest erase rate and response time values (owing to large request sizes and low inter-arrival times) in the traces shown. For the Financial trace [53], the observed

[1] Adjusted R-square defines the proportion of variability that is accounted for by a statistical model. Unlike R-square, it only increases if a newly added predictor statistically improves an existing model.



a) Performance Model (b) Lifetime Model

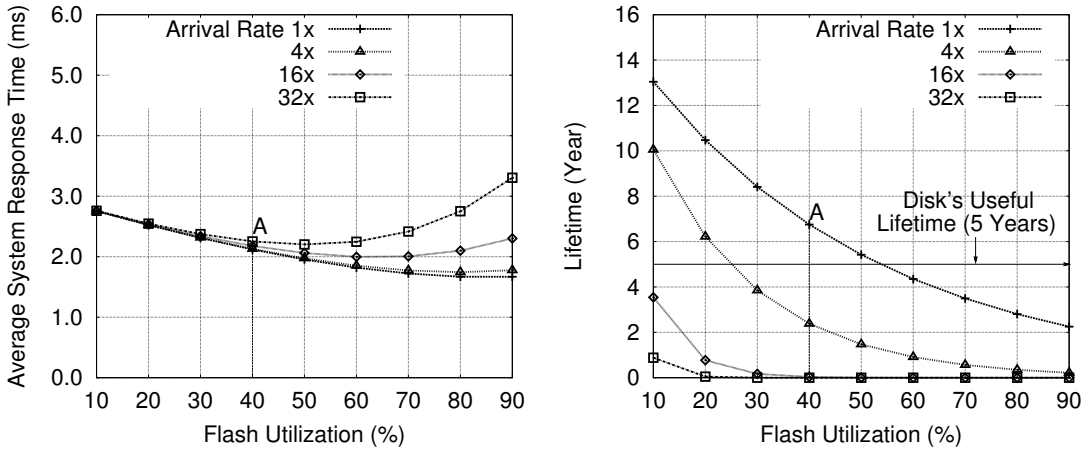
Figure 5.1. Validation of performance and lifetime models compared with values measured using MixedSim. *Each bar of MixedSim in (a) is shown with 99% confidence interval. The 99% confidence intervals of MixedSim in (b) are very small and hence not shown.*

Index	Sequentiality (Ratio)	Request Size (Sectors)	Utilization (Ratio)	Inter-Arrival (ms)
W1	0.10	41.54	0.89	322.18
W2	0.70	16.90	0.89	79.90
W3	0.30	115.71	0.94	80.24
W4	0.70	115.44	0.58	319.74
Financial	0.03	6.57	0.91	164.49

Table 5.1. Some of the synthetic write-only workloads (W1,W2,W3,W4) used to train the performance and lifetime models and a realistic Financial Trace workload used for evaluating the models.

performance as well as lifetime errors are high. The major cause of this discrepancy is that our black-box model assumes no information about the internal state of the flash and hence is liable to errors. Arguably, by incorporating more information about flash internals we can improve our model further. However, as explained in Chapter 4, for HybridStore, having a MixPlan suffices so long as MixDyn can handle the inaccuracies in the former models. To summarize our validation, we have demonstrated the possibility of developing a performance and lifetime estimation methodology with reasonable accuracy with simple log linear regression models.

5.2.3 Why MixPlan Alone Doesn't Suffice



(a) Overall Performance (b) Lifetime of flash

Figure 5.2. Capacity planning for Financial-like trace. Note that we increase arrival rate as shown in legends. Flash utilization signifies the amount of space being utilized for sending requests. All other requests are serviced from HDD.

Workloads are known to exhibit variation from their predicted behavior. In such circumstances, capacity planning alone is not sufficient to meet the lifetime and performance budgets. Figure 5.2(a)-(b) show the impact of increased arrival rate on performance and lifetime budgets for a write-dominant workload. If the system designer had provisioned the system at point A to keep the flash lifetime around a disk's useful life while satisfying the performance needs, these guarantees do not hold if the workload changes. With higher intensity of writes, the garbage collector is invoked more often; thus degrading the system's performance. Moreover, it results in higher number of block erases, reducing the flash lifetime. Thus, we require additional sophisticated data partitioning mechanisms which can dynamically adapt to these changing workload environments. In the next section, we describe some techniques employed by our dynamic controller (MixDyn) to meet the various budgets and thus work in synchronization with MixPlan.

5.3 Dynamic Controller: MixDyn

We have established the need for a fine-grained control mechanism which should be able to manage the requests and hence ensure sustained throughput from the storage system which is able to meet our lifetime and performance budgets. In this section, we discuss the core of MixDyn—the performance prediction module—and then elaborate other components, namely *(i) Fragmentation Buster*, *(ii) Write Regulator*, and *(iii) Adaptive Wear-leveler* which try to ensure that the guarantees made by MixPlan are upheld.

5.3.1 Short-Term Performance Prediction Model for SSD

The performance of the SSD is highly dependent on the workload incident on it. Since out-of-place updates are performed on the flash, GC resulting from fragmentation has an important impact on response time. We build upon our learning from capacity planning and try to develop time-scale performance models suitable for the MixDyn. Although the large-body of work on modeling disk performance is of use here, there are certain salient novel aspects of flash operation that MixDyn's SSD model must capture. Perhaps the most important such feature is that unlike a disk, *an SSD performance model needs to incorporate a much longer history*, since a large enough number of random writes (that might themselves experience good performance) might cause fragmentation over time and the resulting GC invocation would then degrade the performance of requests that arrive much later.

Again we start with identifying the crucial workload characteristics which play a major role. However, contrary to the earlier MixPlan performance model here, we work with a sliding window of requests. This sliding window acts as a short term history of requests and enable us to make fair short term decisions. The main workload characteristics used in the model are: (i) *Average Read to write ratio* of a window of requests, (ii) *Spatial locality*—average sequentiality of a window of requests, (iii) *Request inter-arrival time*, and (iv) *Current request size*. Since this performance model needs to make predictions about the performance of requests in the immediate future, and as seen how performance depends on long-term history, we need to capture and preserve certain aspects of the *current state* of the flash device. However, this information about state of the flash device might require

information about SSD internals that may not be feasible (e.g., in the SSD that HybridStore assumes).

In order to build a feasible as well as efficient black-box performance model, we use the history of previous device service times as an indicator of flash device state. For simplicity, we use the average of the service times (S_{avg}). Moreover, we use system response time ($R_{current}$) as a measure of flash device performance. Thus, our multiple linear regression model can be represented as

$$R_{current} = c_0 + c_1 \cdot W_{window} + c_2 \cdot S_{avg} + \epsilon$$
$$S_{avg} = \frac{\left(\sum_{j=1}^{w} S(j)\right)}{w} \tag{5.3}$$

where ϵ is a small error and W_{window} is the workload during window w. The coefficients (c_0, c_1, c_2) are estimated during a learning/training phase of our experiment which consists of half of the workload. We believe converting our learning-based prediction technique can be easily adapted to operate on-line, although we do not evaluate that here.

5.3.2 Evaluation with Dynamism-Aware Performance Predictor

We use the Financial trace [53] and TPC-H [55] workload to validate our model. Contrary to our performance predictor for MixPlan, our empirical evaluation suggests a simpler multiple linear regression to be satisfactory. For Financial trace, we observe the measured R-square value to be 98% (as shown in Table 5.2). We compare the accuracy of our model with a simple baseline—a *last value-based* prediction model for SSD which uses the last service time value as its prediction. Figure 5.3 demonstrates the superior prediction quality of our model for both TPC-H and Financial trace. Our model is able to predict the state of the flash better than the last value predictor and hence shows much small error rate.

Dynamism-Aware Performance Prediction - The performance of the SSD is highly dependent on GC overhead resulting from fragmentation on the SSD as well as workloads running on it. SSD could show worse performance than

Multiple R	0.98
R Square	0.98
Adjusted R Square	0.98
Standard Error	0.27
Observations	32289

(a) Regression Statistics

	Coefficients	Standard Error	P-value
Intercept	0.13145	0.002982131	0
Previous Device Service Time	0.01223	0.003517166	0.000505959
Real/Write Ratio	-0.66566	0.019041997	8.0937E-263
Sequentiality	-0.78597	0.159642759	8.55189E-07
Inter-Arrival	-0.00007	8.29538E-06	3.00991E-16
Request Size	0.12127	0.000381543	0

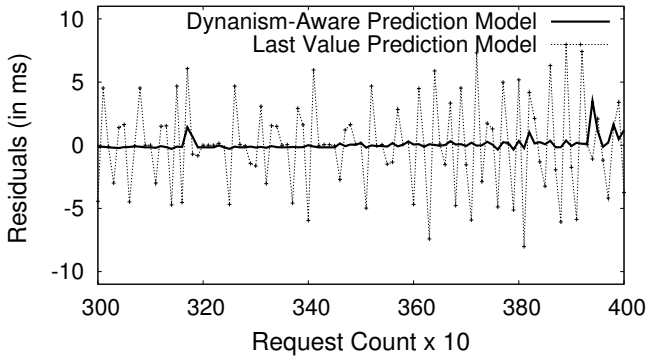
(b) Significance of Predictor Variables

Table 5.2. Statistical Statistics for Performance Prediction of Flash for Financial Trace. Correlations between all predictor variables are almost zero.

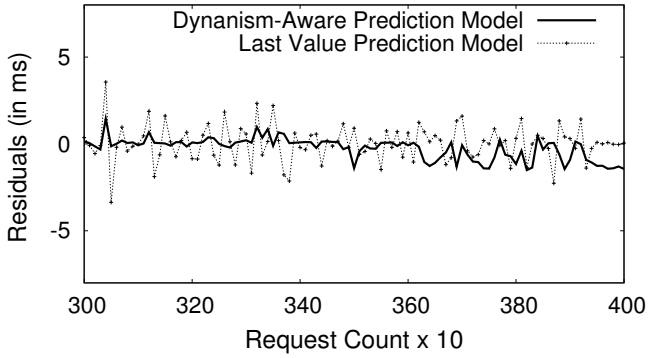
HDD if many small random writes are requested to the SSD for a long time. We build a performance predictor module for the SSD and provide a comprehensive dynamic-aware data partitioner for the best performance of both SSD and HDD. We will evaluate the superiority of our dynamic-aware performance prediction in HybridStore in Section 5.4.

5.3.3 Fragmentation Busting

As described earlier, small random write increases data fragmentation on flash, thus exacerbating garbage collection overhead. We demonstrate this impact in Figure 5.4 by alternating sequential and small random write requests for synthetic workloads. The presence of random writes in region "B" increases the average response time for requests in "C" as compared to writes in "A" although both "A" and "C" represent regions with sequential write activity. In order to prevent such fragmented zones on flash, we try to develop a flushing methodology called *Fragmentation Busting*. As shown in Figure 5.4, flushing some portion of these small random writes to disk (periodically moving 25% of random writes for this experiment), we can reduce the variation in response times and improve the



(a) Financial



(b) TPC-H

Figure 5.3. Comparison of our dynamic SSD performance prediction model with a simple last value-based prediction model. *The 99% confidence intervals are very small and hence not shown.*

performance.

Flushing prevents the impact of random writes from increasing. For experiments, we peoridically flushed 25% valid data on flash into disk. During the beginning of the experiment, since the flash was empty Figure 5.4 demonstrates the improvement in response time with fragmentation buster. impact in Figure 5.4 by (which means that flash hasn't yet been affected by fragmentation), it shows sustained low throughput regardless of randomness in the requests (device response time is less than 0.25 ms). However, the performance begins to decrease according to level of randomness increased in the workloads. "A" in the figure shows a regions that flash's device service time is steeply increasing. However, sequential stream significantly improves the degraded performance (due to small random writes) in previous phases (refer to a "B" region in the figure). Also the fragmentation buster

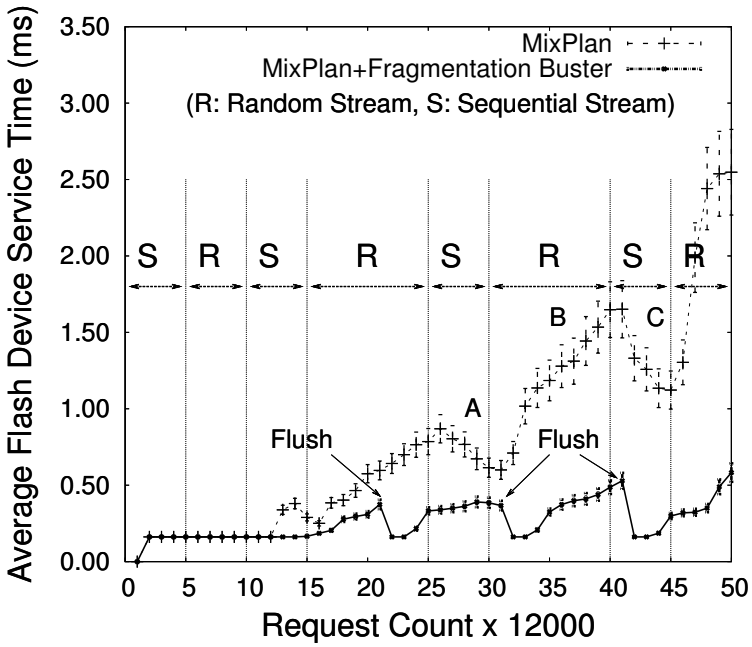


Figure 5.4. Performance degradation due to fragmentation on flash and subsequent performance improvement with fragmentation buster. *Flush* indicates periods of migration activity from flash to disk. *Each point is shown with 95% confidence interval.*

keeps sustained throughput by flushing unnecessary data into disk at the moment indicated by "Flush" in the figure. In this experiment, we considered flushing as a background job, which doesn't affect the foreground performance.

Workloads are known to exhibit periods of idleness between bursts of requests [83]. Lot of research has gone into developing techniques to identify and utilize these idle periods. Specifically, Mi et al. categorized workloads based on idle periods into tail-based, body-based and body+tail based [83]. and found the presence of heavy-tailed inter-arrival times in enterprise-scale workload implying the presence of significant idle periods along with those of intense activity. Currently, we do not incorporate any specific policy in our HybridStore design. Flushing requires co-operation from the device since the effective mapping tables are present within the device and are not exposed to outer systems. Thus, only a part of the flushing mechanism, specifically the scheduler, can be implemented with MixDyn. In order to decide which data needs to be flushed, the device controller needs to pin the pages causing this fragmentation. We maintain a LRU (Least Recently Used) list of the valid pages using the logical page number of the requests. This represents

the cold data on flash and its migration to disk does not have any major impact on HybridStore's performance. When the idle period kicks in, the fragmentation buster directs the flash controller to start flushing the data fragments. A small DRAM-based buffer needs to be maintained so that any request to the data being migrated can be serviced. Since we flush mostly cold data, such requests are rare. Moreover, since this activity can be delayed until an idle period is available, in this work we consider it a pure background activity that does not interfere with the real workload and hence we ignore its possible degrading effects on overall performance.

5.3.4 Handling Uncertainties in Enterprise-scale Workloads

We pointed out to one of the challenges in capacity planning (Section 5.2.3) as the unpredictability in workloads. A prolonged and/or recurring period of unanticipated random writes detrimentally impact on lifetime of flash. In this sub-section, we develop techniques for handling sudden unanticipated bursts in requests.

5.3.4.1 Write Regulation

The workload projections made by MixPlan are dictated by general workload characteristics and are subject to violations during operation. The write regulator monitors the rate of incoming writes and comes into action if sustained violations are observed. This is essential to preserve the lifetime budget requirements. When violations are detected, it starts to regulate the writes being sent to flash by overriding the decisions made by the performance model in MixDyn. Currently, we use a policy which randomly picks the requests being sent to flash and diverts them to disk instead. As part of future work, we plan to develop more sophisticated models.

5.3.4.2 Adaptive Wear-Leveling

As described earlier, wear-leveling requires swapping of data between blocks which have high erase count with blocks which have relatively lower erase count. This swapping operation results in additional erase operations which reduce the lifetime

of blocks [47, 13]. These extra erases start to play a significant role towards the end of a flash device's life and indeed accelerate its death. Traditional wear-leveling algorithms define the lifetime of flash based on the reduction in the capacity of the device as compared with the original capacity and hence aim to achieve uniform distribution of erases across all blocks on flash. We propose a paradigm shift in this philosophy by defining the useful lifetime of flash in hybrid environment to be the time while HybridStore is meeting the performance/lifetime guarantees. This provides us the flexibility to allow wear-out of few blocks on flash by temporarily halting wear-leveling mechanism if it helps in meeting the overall lifetime budget. We propose an *adaptive wear-leveling mechanism*—a novel idea to the best of our knowledge—which, like the write regulator monitors the erase rate of blocks and during periods of prolonged unanticipated write activity, co-ordinates with the flash controller to prevent the extra erases caused by wear-leveling by temporarily halting the leveling algorithm. Once normal I/O activity starts (as projected by MixPlan to uphold the lifetime budget), it allows the device to revert to its wear-leveling mechanism.

5.4 Evaluation

5.4.1 Experimental Setup and Workloads

In this sub-section, we describe the enterprise-scale workloads and our hybrid simulator used to evaluate HybridStore. For evaluation purposes, we have developed our own simulator named *MixedSim*.

Workloads - Table 5.3 illustrates the characteristics of enterprise-scale workloads used in our evaluation. We employ a write-dominant I/O trace from an OLTP application running at a financial institution [53] made available by the Storage Performance Council (SPC), henceforth referred to as the *Financial trace*. We also experiment using Cello99 [54], which is a disk access trace collected from a time-sharing server exhibiting significant writes; this server was running the HP-UX operating system at Hewlett-Packard Laboratories. TPC-H [55] is is an ad-hoc, decision-support read dominant benchmark (OLAP workload) examining large vol-

umes of data to execute complex database queries. Finally, we also use a number of synthetic traces to study the efficacy of MixPlan and MixDyn for a wider range of workload characteristics than those exhibited by the above real-world traces.

Workloads	Request Size (KB)	Read (%)	Sequentiality (%)	Inter-arrival Time (ms)
Financial (OLTP) [53]	4.38	9.0	2.0	133.50
Cello99 [54]	5.03	35.0	1.0	41.01
TPC-H (OLAP) [55]	12.82	95.0	18.0	155.56

Table 5.3. Enterprise-Scale Workload Characteristics. All values are average. Note that sequential requests means that the address of incoming request is next to that of the previous request.

Flash Device	
Parameter	Value
Page (Data)	2KB
Page (OOB)	64B
Block	(128KB+4KB)
Page Read Time	130.9 us
Page Write Time	405.9 us
Block Erase Time	1.5 ms
Interface	SATA
Garbage Collector	Yes
Wear-leveling	Implicit/Explicit
FTL Type	Page/DFTL

Hard Disk Drive (HDD)	
Parameter	Value
Disk Model	IBM Ultrastar 36Z15
Interface	SATA
Storage Capacity	36.7 GB
RPM	15,000
Seek Time	3.4 msec
Rotation Time	2 msec
Internal Tx Rate	55 MB/sec

Table 5.4. Default simulation parameters.

MixedSim - We develop a simulation framework for integrated disk and flash based storage systems, called MixedSim. It is built by enhancing Disksim 3.0 [49], a well-regarded HDD simulator. Disksim is an event-driven simulator which has been extensively used in different studies [84, 52, 60] and validated with several disk models. It simulates storage-system components including disk drives, controllers, caches, and various interconnects etc. However, it does not allow the modeling of flash based devices. MixedSim is designed with a modular architecture with the capability to model a holistic storage environment. It is able to simulate different storage sub-system components including device drivers, controllers, caches, flash devices, disks, and various interconnects. In our integrated simulator, we add the

Workload	Lifetime (Yr)		
	Over Provisioned (2GB)	MixPlan (1GB)	Under Provisioned (0.5GB)
Financial	52.69	7.29	2.67
TPC-H	97.50	21.65	-

Table 5.5. Lifetime observations with different approaches. A block is assumed to possess 10K reliable erase cycles.

basic infrastructure required for implementing the internal operations (page read, page write, block erase etc.) of a flash-based device. The core FTL engine is implemented to provide virtual-to-physical address translations along with a garbage collection mechanism. Table 5.4 describes the various flash device and HDD parameters used in MixedSim. [2] The details of flash simulator used in MixedSim is in Chapter 3.

5.4.2 Evaluation of MixPlan

In this sub-section, we compare the performance of MixPlan capacity planner with other generic capacity planning methodologies which either under-provision or over-provision the flash capacity in HybridStore.

5.4.2.1 Lifetime Budget Constraint

Table 5.5 shows the flash device lifetime for various capacity planning techniques with dynamism-aware data partitioning policy for different workloads.
TPC-H is read dominant and hence performance budget is of greater concern than lifetime. For Financial trace which is a write-dominant workload, we observe that under-provisioning capacity would necessitate flash device replacement within 3 years and hence would impact the overall lifetime budget of HybridStore. We want the flash device to last till around the useful life of disk (approximately 5 years) and both over-provisioning and MixPlan are able to achieve this mandate. Over-provisioning flash capacity should reduce the request response times from flash device since the garbage collection overheads will be reduced and hence improve HybridStore performance as compared to MixPlan. However, as we observe in the

[2]Simulations using current state-of-the-art HDDs such as Seagate's Cheetah15K and SSDs such as Intel's X25-M SSD are part of our future work.

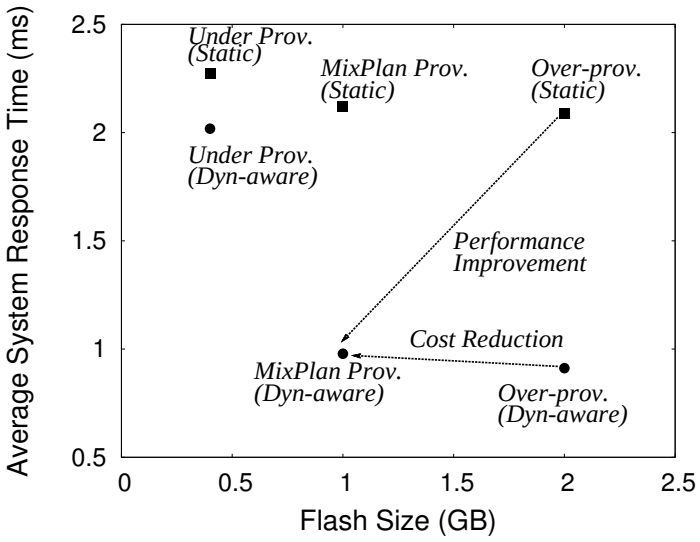


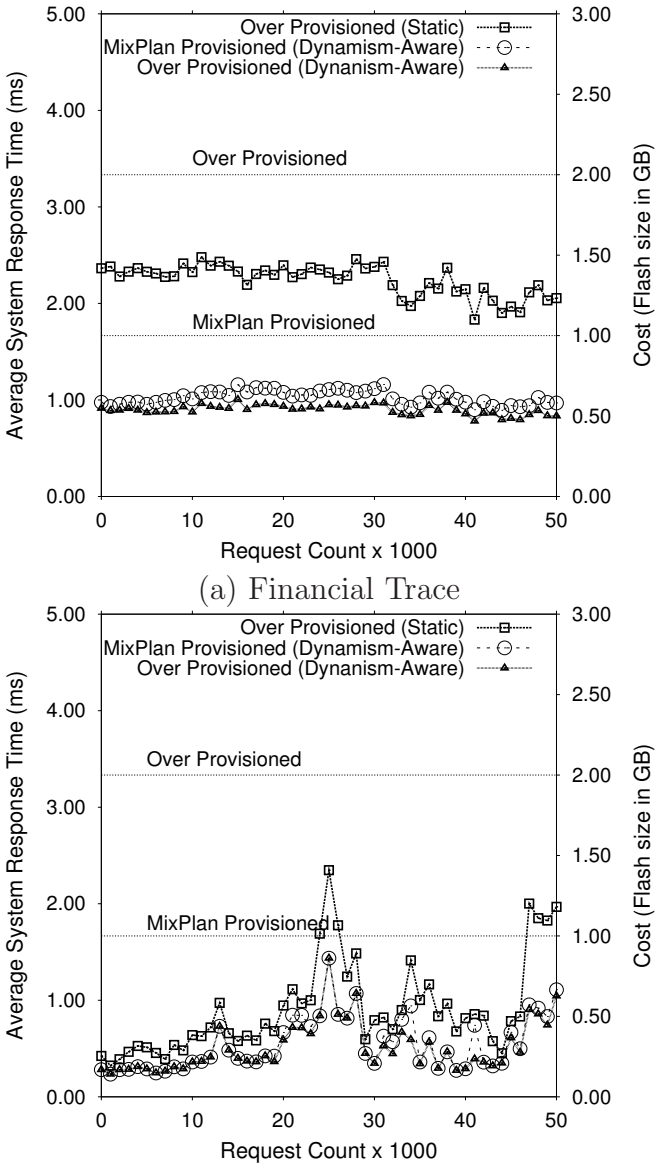
Figure 5.5. Capacity Planning for Financial Trace. "Static" denotes a static data-partitioning policy where write requests larger than 4KB are assumed to be sequential and are serviced by the HDD and others are serviced by SSD. "Dyn. aware" denotes an intelligent data partitioning.

next sub-section, benefits accrued with this extra flash are much less as compared to the increased cost due to larger flash.

5.4.2.2 Performance Budget Constraint

Figure 5.5 demonstrates the improvement in performance and reduction in cost using MixPlan along with dynamism-aware performance predictor for Financial Trace. Both MixPlan and Over-provisioning with static data partitioning are able to meet the lifetime guarantees and improve the response time as compared to an under-provisioned system. However, as illustrated in Figure 5.6(a), if dynamism-aware data partitioner is utilized along with an over-provisioned flash, we observe a slight improvement in performance as compared to MixPlan. But this small improvement comes at an additional cost of bigger flash memory. Thus, the cost-to-benefit (Figure 5.5) ratio advocates the use of MixPlan for capacity planning in enterprise-scale systems.

As shown in Figure 5.6(b), for read-dominant TPC-H [55], both MixPlan and over-provisioned models provide similar performance. This can be directly attributed to the fact that read-oriented workloads have very small amount of writes,

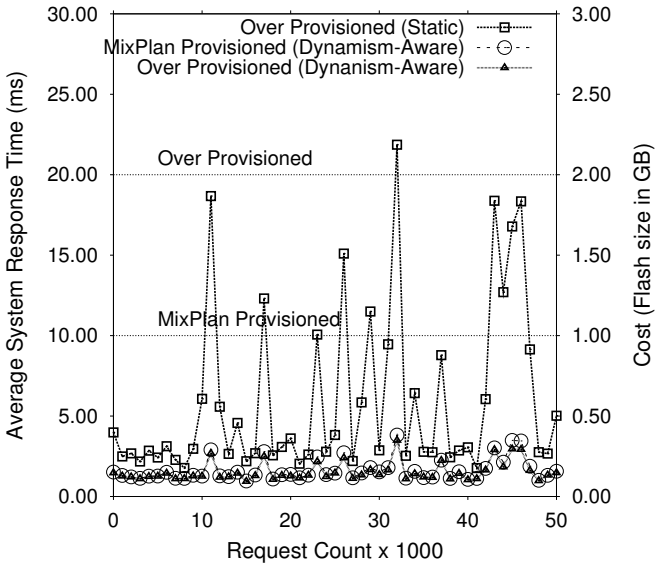


(a) Financial Trace

(b) TPC-H

Figure 5.6. Capacity Planning: MixPlan is not only able to reduce the cost but also improve performance in conjunction with dynamism aware data partitioning. "Static" denotes a static data-partitioning policy where write requests larger than 4KB are assumed to be sequential and are serviced by the HDD and others are serviced by SSD.

thus the garbage collector is invoked very infrequently and the service patterns remain similar for both the capacity planning methodologies. Figure 5.7 clearly illustrates the need for dynamism-aware data partitioner (MixDyn). Static partitioning is unable to handle periods of bursts in Cello resulting in poor response



Cello99

Figure 5.7. Capacity Planning for Cello99. All other assumptions are the same as those in Figure 5.6.

times. On the other hand, MixDyn is able to respond effectively to such situations and hence provide better performance. Now we evaluate these benefits in the next sub-sections.

5.4.3 MixDyn Acting in Concert with MixPlan

We evaluate the performance of prediction models in MixDyn along with our novel three mechanisms such as (i) adaptive wear-leveling, (ii) write-regulation and (iii) fragmentation busting.

5.4.3.1 Dynamism-Aware Performance Prediction

We integrate our SSD prediction model with an admittedly simple disk performance predictor. We use a model based on the average response time observed during the training phase to predict disk performance. The dynamic controller (MixDyn) partitions write requests depending on the least response times predicted by the SSD and HDD models. MixDyn maintains a table to store information about the current location of data (device id) and updates it whenever some

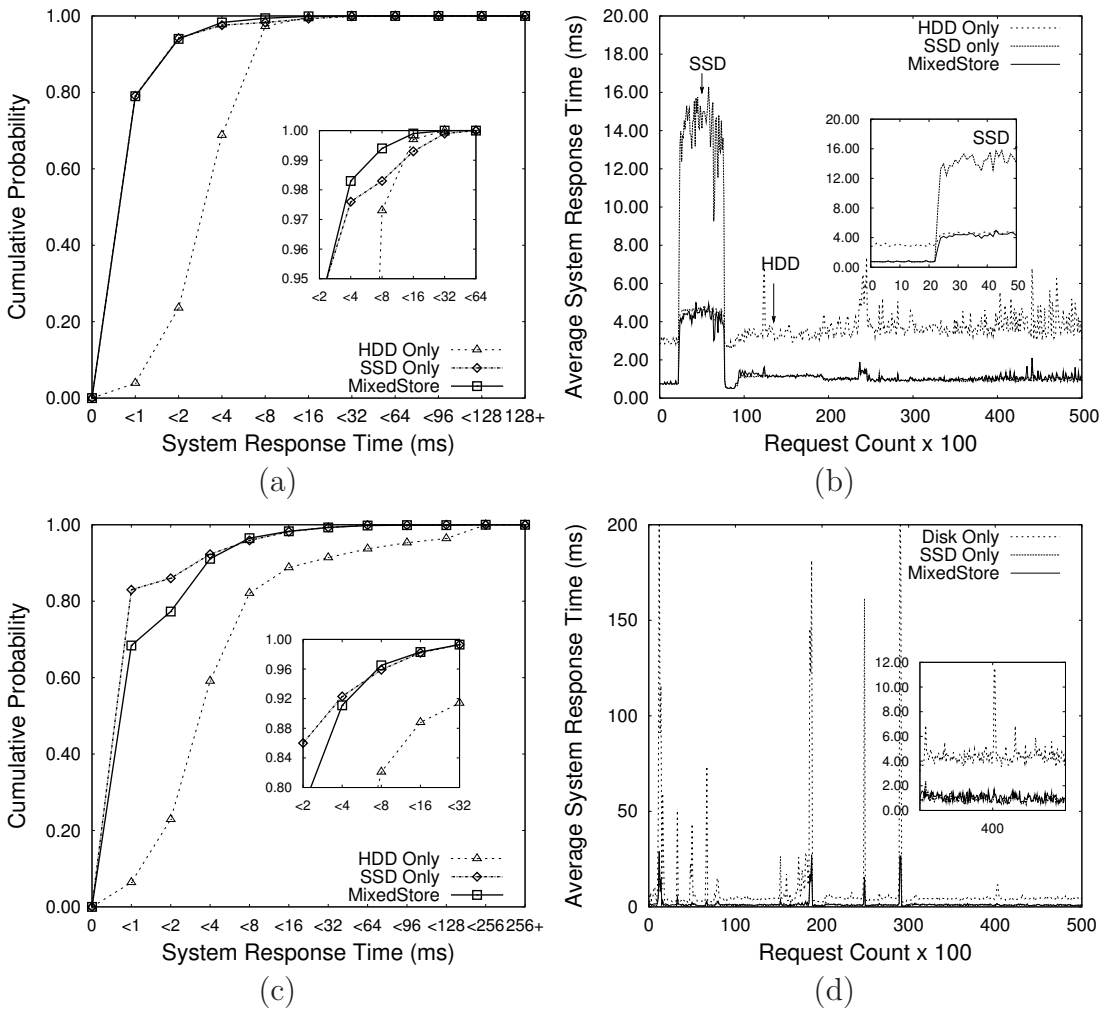
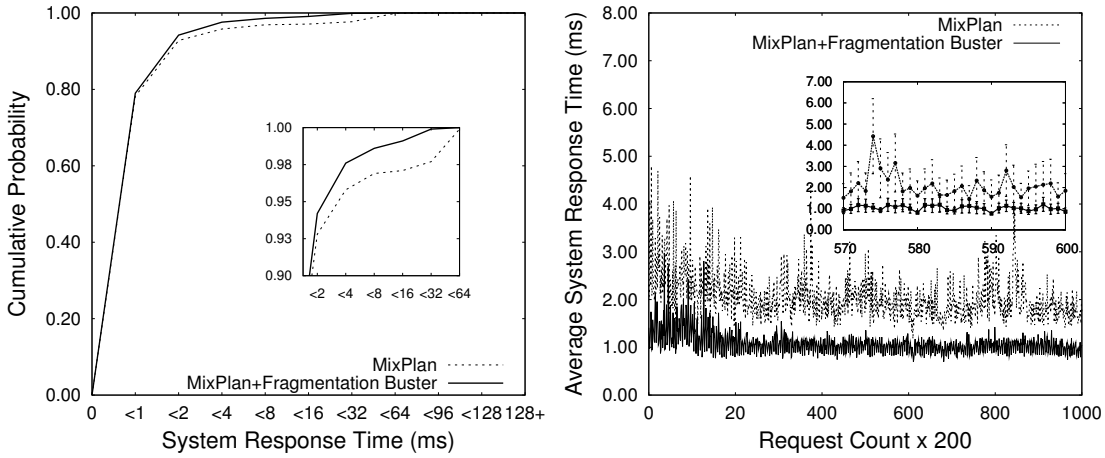


Figure 5.8. Performance of HybridStore compared with a disk-only and a flash-only system. (a) and (b) respectively shows CDF and performance behavior in time series for Financial Trace. (c) and (d) are the same experiments for Cello99. *The 99% confidence intervals are very small and hence not shown.*

data is migrated from one device to the other. Read requests are always serviced from the device which contains the data.

Figure 5.8(a) illustrates the performance of HybridStore incorporating the prediction models in MixDyn with respect to a disk-only and flash-only system for the random write dominant Financial trace. Although we observe good performance from flash device for servicing most requests, but as shown in Figure 5.8(b), some requests suffer from extensive GC overhead and exhibit high response time on flash. Our prediction model is able to move these requests to the disk and achieve



(a) (b)

Figure 5.9. (a) Performance improvement of MixDyn with fragmentation buster for Financial Trace. (b) Sustained improved performance (consistently reduced response times) obtained using fragmentation buster for Financial Trace. *Each point in the small zoomed-in graphs in (b) is shown with 95% confidence interval.*

better performance for HybridStore. Moreover, HybridStore reduces the average system response time by about 71% as compared to a disk-only system. Similar performance improvement is observed for Cello. However, the limitation of simplistic disk prediction model is observed for Cello in Figure 5.8(c) where flash-only system improves response time by about 20% as compared to HybridStore. The disk prediction model (in HybridStore) is unable to capture the high intensity of random writes resulting in incorrect prediction by MixDyn since some high latency requests are now inevitably wrongly serviced from disk. We believe with a more sophisticated disk performance prediction model will alleviate such discrepancies and improve the performance of MixDyn. We plan to pursue this as part of our future work.

5.4.3.2 Fragmentation Busting

We have already established in Section 5.3.3 that data fragmentation leads to degradation of flash device's performance and necessitates fragmentation busting. We perform offline profiling of enterprise-scale workload to identify idle periods. This enables us to schedule fragmentation busting activity in the background. As shown in Figure 5.9(a), it helps in reducing the tail of CDF for the Financial

Technique	Average Erase Rate (Erases/Sec.)	Ratio of Requests Serviced by Flash (Flash/(Flash+Disk))	Average System Response Time (ms)
MixPlan	0.16	0.69	1.15
MixPlan (Red_{25})	0.12	0.54	1.56
MixPlan (Red_{50})	0.09	0.40	1.98

Table 5.6. Evaluation of Write Regulation.

trace i.e., reducing the number of requests that experienced high response time. This allows sustained improved performance from the flash device by reducing the variance in response time as illustrated in Figure 5.9(b).

5.4.3.3 Write Regulation

We experiment with a write regulator that detects increased I/O activity and consistently monitors the expected flash life through the lifetime model of MixPlan. When violations are detected, it starts to regulate the writes being sent to flash by over-riding the decisions made by the performance model in MixDyn. Currently, we use a policy which randomly picks the requests being sent to flash and diverts them to disk instead. As part of future work, we plan to develop more sophisticated models. We experiment with two models of a static write rate regulator that pick 25% or 50% (uniformly at random) of the requests being sent to flash and redirects them to HDD during periods of higher-than-expected I/O intensity. Let us call these policies Red_{25} *and* Red_{50}*, respectively.* For this experiment, we synthesize a workload with similar characteristics as Financial trace but with periods of reduced inter-arrival time between requests. Table 5.6 shows that we are able to reduce the flash block erase rate by about 25% while reducing the requests being serviced by flash by about 21% using Red_{25}. An additional 19% reduction in the erase rate is observed using Red_{50}. However, it results in an increase of 0.83ms in average system response time. Thus, the rate of write regulation must be chosen judiciously so as to meet the performance budget while ensuring that lifetime guarantees are satisfied.

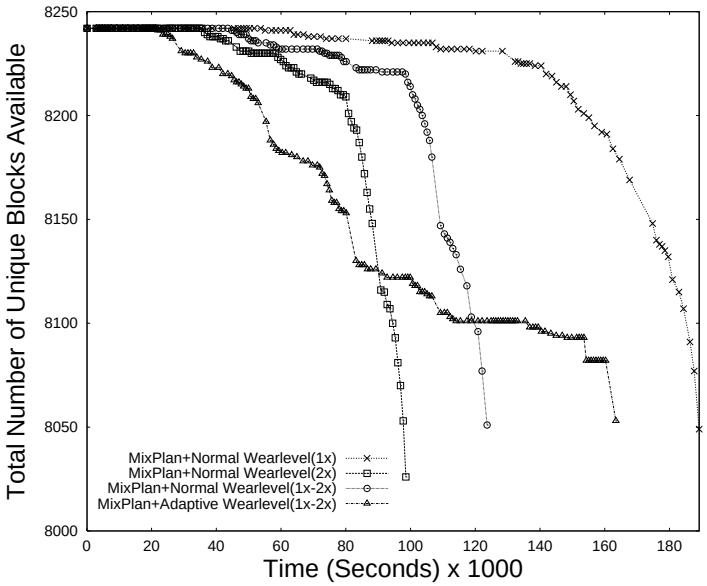


Figure 5.10. Adaptive Wear-Leveling. 1x and 2x denote the normal and unanticipated increase (2 times speed-up) in I/O intensity in the Financial trace. 1x-2x denotes a trace with regions of normal and increased I/O activity. Normal wear-leveling refers to continuously invoking wear-leveling algorithms irrespective of the available useful lifetime of blocks on flash.

5.4.3.4 Adaptive Wear-Leveling

The lifetime projections made by MixPlan are subject to violations due to uncertainties (increased unanticipated I/O activity) in the enterprise-scale workloads. Our novel adaptive wear-leveler helps MixDyn in upholding these guarantees. Figure 5.10 shows the impact of our adaptive wear-leveler on the Financial trace with modified inter-arrival times to resemble a workload with periods of high I/O activity. It halts the wear-leveling algorithm when it detects prolonged unanticipated I/O activity (we use static profiling to detect these periods). As compared to the normal wear-leveling algorithm which continuously performs leveling irrespective of the residual lifetime of blocks, our adaptive algorithm is able to improve the useful lifetime of flash by about 33%. This helps in delaying the need for replacement and reducing re-deployment costs. This enables MixDyn to achieve the lifetime guarantees as projected by MixPlan; hence both our capacity planning and dynamic-controller tools act in tandem to achieve the lifetime and performance budgetary requirements.

5.5 Concluding Remarks

This research was based on the emerging consensus among several storage experts that in the foreseeable future, with the exception of certain specialized domains, SSDs should be used as a complementary device to HDDs in enterprise-scale storage hierarchy. We attempted to address two problems in a simplified version of such a hybrid system consisting of one HDD and one SSD sharing the I/O bus. First, we developed an on-line capacity planner called *MixPlan* that used LP models to meet the performance and lifetime requirements of SSDs and HDDs to provide storage administrators with guidelines on provisioning such a system in a cost-effective manner. Second, we developed a dynamic controller, *MixDyn*, that used shorter time-scale SSD and HDD models along with regulation of write rate to the SSD and a novel idea of adaptive wear-leveling within the SSD to operate the storage system within regions of desirable cost, performance, and lifetime budgets. We evaluated these systems using a simulator (MixedSim) developed by us using a variety of well-regarded benchmarks. We found that HybridStore is able to reduce the average system response time by about 71% as compared to a HDD-based system for a enterprise-scale Financial trace. Moreover, our innovative adaptive wear-leveling mechanism was able to prolong the life of SSDs by about 33% in the presence of unanticipated increase in I/O intensity. In essence, our research opened up new vistas for not only designing a well-provisioned enterprise-scale storage system consisting of SSDs and HDDs but also established a need for re-looking at the design of SSDs to incorporate some innovative mechanisms such as fragmentation buster and adaptive wear-leveler.

Chapter 6

Conclusion and Future Work

6.1 Summary

Certain idiosyncrasies of SSDs make their integration into HDD-based systems non-trivial. Their peculiar properties related to cost, performance, and lifetime make it difficult for a storage system designer to neatly fit them between HDD and DRAM. Given the complementary properties of HDDs and SSDs in terms of cost, performance, and lifetime, the current consensus among several storage experts is to view SSDs not as a replacement for HDD but rather as a complementary device within the storage hierarchy.

In Chapter 3, I propose and design a novel FTL which is purely page-mapped. This work was motivated to improve random write performance of flash. Recent research solved such random write problems on flash by adding DRAM-backed buffers or buffering requests to increase their sequentiality. However, I focused on an intrinsic component of the flash, namely the *Flash Translation Layer (FTL)* to improve poor performance due to random writes in SSDs. Moreover, I developed and validated flash simulation framework call *FlashSim* [1]. to provide an experimental test-bed for SSD related research.

In Chapter 4, I propose HybridStore, a hybrid storage system containing HDDs and SSDs. Besides this hardware, I present a *capacity planner* that makes long-term resource provisioning decisions for the expected workload; it is designed to

[1] `http://csl.cse.psu.edu/hybridstore`

optimize the cost of equipment that needs to be procured to meet desired performance and lifetime needs for the expected workload. I investigate how the capacity planner can find the economical optimal configuration in a heterogeneous storage environment containing multiple HDDs and SSDs. I also presents a general form of comprehensive methodology using a well-known technique for optimization problems, Linear Programming (LP). As an illustrative result, the capacity planner is able to identify close to minimum SSD capacity needed to meet a specified performance goal for a realistic workloads while ensuring similar performance as compared to a comparatively more over-provisioned system.

In Chapter 5, I present a *dynamic controller* whose goal is to operate the system in desirable performance/lifetime regimes in the face of deviations at short time-scales in workload. In order to answer to where to send the requests in hybrid storage system, I develop simple statistical models that HybridStore employs. These models are used in conjunction with *MixedSim*. I implement the dynamic controller in our simulator. I enhance block device driver that employs online statistical performance and lifetime models for SSD and a performance model for HDD to dynamically partition incoming workload among the SSD and HDD, and two algorithms within the SSD controller (specifically, within the FTL layer) including reduction of fragmentation within the flash and a novel concept of *adaptive wear-leveling*.

6.2 Future Work

Immediate Research Direction - There are several unresolved issues related to resource and data management in hybrid storage systems. One of interesting set of problems comes from the needs of new storage hierarchies. These call for changes of traditional file systems to consider different storage alternatives. A design and implementation of hybrid file systems (in which data blocks/files can move over any storage media in local or distributed storage systems and the file system efficiently handles this movement) is also interesting research topic. An equally important set of problems is concerned with modeling and developing a tool to extract unknown internal information from the flash device. I have already experience in developing black box models to understand the performance/lifetime behavior of a given flash

device. I can extend this statistical approach to provide a more sophisticated tool that understands more details of internal structures/operations of the flash, such as DRAM size, caching policies on DRAM, parallelism effects, and etc.

Long-term Research Direction - I anticipate that my long term future work will continue towards designing new memory/storage hierarchies by considering all possible storage alternatives. Foremost to this agenda are the ever-increasing abundance of storage alternatives (such as flash memory, MRAM, PRAM, and etc.) and the needs of new memory/storage hierarchy designs to provide high performance and energy efficient memory/storage organization. New designs of memory/storage hierarchies should be done carefully because it could impose dramatic changes in system software design. I foresee that resource management as well as data management in such systems will raise interesting research questions.

For example, can we potentially exploit such storage alternatives to create new memory/storage hierarchies? Traditional memory hierarchy in a system simply consists of three levels, such as processor caches, system caches, and storage device. we can study more to find the proper levels of hierarchies with new storage alternatives and the proper size of each level under performance, power, and price constraints. Besides, system support from operating systems in these new hierarchies will be another interesting problems. Since we will consider more complex hierarchies, the system support from the operating systems will need to be efficiently designed and developed. These will raise a number of interesting algorithmic problems such as efficient management of data across hierarchies. For example, what kinds of file system support will be needed? Can we identify principles and solutions common to these new hierarchies so that design and development efforts may benefit from efforts in the rest?

Bibliography

[1] Micron (2007) *Technical Report (TN-29-07): Small-Block vs. Large-Block NAND Flash Devices*, *Tech. rep.*, Micron, `http://www.micron.com/products/nand/technotes`.

[2] EMC (2006), "Emc Symmetrix DMS-3 Electrical Power Estimation and Configuration Planning ," `http://www.sandirect.com/documents/symmetrix_dmx3_elec_power_est_wp.pdf`.

[3] Tehrani, S., J. Slaughter, E. Chen, M. Durlam, J. Shi, and M. DeHerren (1999) "Progress and Outlook for MRAM Technology," *IEEE Transactions on Magnetics*, **35**(5), pp. 2814–2819.

[4] Raoux, S., G. W. Burr, M. J. Breitwisch, C. T. Rettner, Y.-C. Chen, R. M. Shelby, M. Salinga, D. Krebs, S.-H. Chen, H.-L. Lung, and C. H. Lam (2008) "Phase-Change Random Access Memory: A Scalable Technology," *IBM J. Res. Dev.*, **52**(4), pp. 465–479.

[5] Shimada, Y. (2008) "FeRAM: Next Generation Challenges and Future Directions," *Electronics Weekly*.

[6] Samsung (2006), "Samsung's Q30-SSD Laptop with 32GB Flash Drive," .

[7] Niijima, H. (1995) "Design of a Solid-State File Using Flash EEPROM," *IBM Journal of Research and Developement*, **39**(5), pp. 531–545.

[8] Gal, E. and S. Toledo (2005) "Algorithms and Data Structures for Flash Memories," *ACM Computing Survey*, **37**(2), pp. 138–163.

[9] Atwood, G., A. Fazio, D. Mills, and B. Reaves (1997) "Intel StrataFlash^{TM} Memory Technology Overview," *Intel Technology Journal*.

[10] Nitin, A., P. Vijayan, and W. Ted (2008) "Design Tradeoffs for SSD Performance," in *Proceedings of the USENIX Annual Technical Conference*, pp. 57–70.

[11] Jung, D., Y. Chae, H. Jo, J. Kim, and J. Lee (2007) "A Group-based Wear-Leveling Algorithm for Large-Capacity Flash Memory Storage Systems," in *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pp. 160–164.

[12] Kawaguchi, A., S. Nishioka, and H. Motoda (1995) "A Flash-Memory based File System," in *Proceedings of the Winter 1995 USENIX Technical Conference*, pp. 155–164.

[13] Lofgren, K. M. J., R. D. Norman, G. B. Thelin, and A. Gupta (2005) "Wear Leveling Techniques for Flash EEPROM," *United States Patent, No 6,850,443*.

[14] EMC, "Symmetrix DMX-4, EMC," `http://www.emc.com/products/detail/hardware/symmetrix-dmx-4.htm`.

[15] Texas-Memory-Systems, "RamSan-500, Texas Memory Systems," `http://www.ramsan.com/products/ramsan-500/`.

[16] Mtron (2008), "2.5-Inch Mtron SATA Solid State Drive - MSP 7000," `http://www.mtron.net/English/Product/ec_msp7000.asp`.

[17] Hennessy, J. and D. Patterson (2003) *Computer Architecture - A Quantitative Approach*, Morgan Kaufmann.

[18] Ban, A. (1993) "Flash File System," *United States Patent, No 5,404,485*.

[19] Kim, J., J. Kim, S. Noh, S. Min, and Y. Cho (2002) "A Space-Efficient Flash Translation Layer for Compactflash Systems," *IEEE Transactions on Consumer Electronics*, **48**(2), pp. 366–375.

[20] Chung, T., D. Park, S. Park, D. Lee, S. Lee, and H. Song (2006) "System Software for Flash Memory: A Survey," in *Proceedings of the International Conference on Embedded and Ubiquitous Computing*, pp. 394–404.

[21] Lee, S., D. Park, T. Chung, D. Lee, S. Park, and H. Song (2007) "A Log Buffer based Flash Translation Layer Using Fully Associative Sector Translation," *IEEE Transactions on Embedded Computing Systems*, **6**(3), p. 18.

[22] Kang, J., H. Jo, J. Kim, and J. Lee (2006) "A Superblock-based Flash Translation Layer for NAND Flash Memory," in *Proceedings of the International Conference on Embedded Software (EMSOFT)*, pp. 161–170.

[23] Lee, S., D. Shin, Y. Kim, and J. Kim (2008) "LAST: Locality-Aware Sector Translation for NAND Flash Memory-based Storage Systems," *SIGOPS Oper. Syst. Rev.*, **42**(6), pp. 36–42.

[24] Choudhuri, S. and T. Givargis (2007) "Performance improvement of block based NAND flash translation layer," in *Proceedings of the Fifth IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pp. 257–262.

[25] Park, S., J. Park, J. Jeong, J. Kim, and S. Kim (2008) "A Mixed Flash Translation Layer Structure for SLC-MLC Combined Flash Memory System," in *Proceedings of the 1th International Workshop on Storage and I/O Virtualization, Performance, Energy, Evaluation and Dependability (SPEED2008)*.

[26] Wu, M. and W. Zwaenepoel (1994) "eNVy: a Non-Volatile Main Memory Storage System," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 86–97.

[27] Miller, E. L., S. A. Brandt, and D. D. E. Long (2001) "HeRMES: High-Performance Reliable MRAM-Enabled Storage," in *HotOS*, pp. 95–99.

[28] Uysal, M., A. Merchant, and G. A. Alvarez (2003) "Using MEMS-Based Storage in Disk Arrays," in *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, USENIX Association, Berkeley, CA, USA, pp. 89–101.

[29] Marsh, B., F. Douglis, and P. Krishnan (1994) "Flash Memory File Caching for Mobile Computers," in *Proceedings of the Twenty Seventh Hawaii Conference on Systems Science*.

[30] Samsung, "Samsung Hybrid Hard Drive," `http://www.samsung.com/Products/Semiconductor/Support/ebrochure/hddodd/hybrid\_hard\_drive\_datasheet\_200606.pdf`.

[31] Bisson, T. and S. A. Brandt (2007) "Reducing Hybrid Disk Write Latency with Flash-Backed I/O Requests," in *Proceedings of the 2007 15th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, IEEE Computer Society, Washington, DC, USA, pp. 402–409.

[32] Kim, H. and S. Ahn (2008) "BPLRU: A Buffer Management Scheme for Improving Random Writes in Flash Storage," in *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, pp. 1–14.

[33] Park, S., D. Jung, J. Kang, J. Kim, and J. Lee (2006) "CFLRU: A Replacement Algorithm for Flash Memory," in *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, pp. 234–241.

[34] Jo, H., J. Kang, S. Park, J. Kim, and J. Lee (2006) "FAB: Flash-Aware Buffer Management Policy for Portable Media Players," *IEEE Transactions on Consumer Electronics*, **52**(2), pp. 485–493.

[35] Kim, H. and R. Umakishore (2009) "FlashLite: A User-Level Library to Enhance Durability of SSD for P2P File Sharing," in *Proceedings of the International Conference on Distributed Computing Systems (ICDCS)*.

[36] "Managed Flash Technology (MFT) Flash SSD Acceleration Software," `http://www.easyco.com/zx1285301141358249458/mft/index.htm`.

[37] Prabhakaran, V., T. L. Rodeheffer, and L. Zhou (2008) " Transactional Flash ," in *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI) (To Appear)*.

[38] Woodhouse, D., "JFFS2: The Journalling Flash FileSystem," .

[39] Manning, C., "YAFFS: Yet Another Flash File System," `http://www.aleph1.co.uk/yaffs`.

[40] Kgil, T., D. Roberts, and T. Mudge (2008) "Improving NAND Flash Based Disk Caches," in *Proceedings of the 35th International Symposium on Computer Architecture (ISCA)*, pp. 327–338.

[41] Leventhal, A. (2008) "Flash Storage Memory," *Communications of the ACM*, **51**(7), pp. 47–51.

[42] "ZFS L2ARC," `http://opensolaris.org/os/community/zfs/`.

[43] Lee, S. and B. Moon (2007) "Design of Flash-based DBMS: An In-Page Logging Approach," in *Proceedings of the International Conference on Management of Data (SIGMOD)*, pp. 55–66.

[44] Narayanan, D., E. Thereska, A. Donnelly, S. Elnikety, and A. Rowstron (2009) "Migrating Enterprise Storage to SSDs: Analysis of Tradeoffs," in *Proceedings of the ACM European Conference on Computer Systems (Eurosys)*, pp. 145–158.

[45] Micron (2008), "Micron 16GB Mass Storage," `http://www.micron.com/products/partdetail?part=MT29F16G08DAAWP`.

[46] Karedla, R., J. S. Love, and B. G. Wherry (1994) "Caching Strategies to Improve Disk System Performance," *IEEE Transactions on Computer (TC)*, **27**(3), pp. 38–46.

[47] Chang, Y.-H., J.-W. Hsieh, and T.-W. Kuo (2007) "Endurance Enhancement of Flash-memory Storage Systems: An Efficient Static Wear Leveling Design," in *Proceedings of the 44th Annual Conference on Design Automation*, ACM, New York, NY, USA, pp. 212–217.

[48] Lee, J., E. Byun, H. Park, J. Choi, D. Lee, and S. H. Noh (2009) "CPS-SIM: Configurable and Accurate Clock Precision Solid State Drive Simulator," in *Proceedings of the Annual ACM Symposium on Applied Computing (SAC)*, pp. 318–325.

[49] Ganger, G. R., B. L. Worthington, and Y. N. Patt (2003) *The DiskSim Simulation Environment Version 3.0 Reference Manual*, Carnegie Mellon University.

[50] Super-Talent-Technology (2007), "2.5-Inch Super-Talent SATA Solid State Drive," `http://www.supertalent.com/products/ssd-commercial.php?type=SATA`.

[51] Gupta, A., Y. Kim, and B. Urgaonkar (2009) "DFTL: A Flash Translation Layer Employing Demand-based Selective Caching of Page-level Address Mappings," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS)*, pp. 229–240.

[52] Kim, Y., S. Gurumurthi, and A. Sivasubramaniam (2006) "Understanding the Performance-Temperature Interactions in Disk I/O of Server Workloads," in *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 179–189.

[53] Storage-Performance-Council, "OLTP Trace from UMass Trace Repository," `http://traces.cs.umass.edu/index.php/Storage/Storage`.

[54] HP-Labs., "HP Labs. Tools and Traces," `http://tesla.hpl.hp.com/public_software/`.

[55] Zhang, J., A. Sivasubramaniam, H. Franke, N. Gautam, Y. Zhang, and S. Nagar (2004) "Synthesizing Representative I/O Workloads for TPC-H," in *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, pp. 142–151.

[56] Storage-Performance-Council, "Websearch Trace from UMass Trace Repository," `http://traces.cs.umass.edu/index.php/Storage/Storage`.

[57] International-Data-Group (2008), "Datacenter SSDs: Solid Footing for Growth," .

[58] Charrap, S. H., P. L. Lu, and Y. He (1997) "Thermal Stability of Recorded Information at High Densities," *IEEE Transactions on Magnetics*, **33**(1), pp. 978–983.

[59] Gurumurthi, S., A. Sivasubramaniam, and V. Natarajan (2005) "Disk Drive Roadmap from the Thermal Perspective: A Case for Dynamic Thermal Management," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pp. 38–49.

[60] Kim, Y., J. Choi, S. Gurumurthi, and A. Sivasubramaniam (2008) "Managing Thermal Emergencies in Disk-Based Storage Systems," *Journal of Electronic Packaging, Transactions of the ASME (Invited Paper)*, **130**(4).

[61] Schirle, N. and D. F. Lieu (1996) "History and Trends in the Development of Motorized Spindles for Hard Disk Drives," *IEEE Transactions on Magnetics*, **32**(3), pp. 1703–1708.

[62] Mallary, M., A. Torabi, and M. Benakli (2002) "One Terabit Per Square Inch Perpendicular Recording Conceptual Design," *IEEE Transactions on Magnetics*, **38**(4), pp. 1719–1724.

[63] Chen, J. and J. Moon (2001) "Detection Signal-to-Noise Ratio versus Bit Cell Aspect Ratio at High Areal Densities," *IEEE Transactions on Magnetics*, **37**(3), pp. 1157–1167.

[64] Goyal, P., D. Modha, and R. Tewari (2003) "CacheCOW: Providing QoS for Storage System Caches," *SIGMETRICS Performance Evaluation Review*, **31**(1), pp. 306–307.

[65] Gulati, A., A. Merchant, and P. J. Varman (2007) "pClock: An Arrival Curve based Approach for QoS Guarantees in Shared Storage Systems," in *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, pp. 13–24.

[66] Samsung (2008), "Samsung 256GB Flash SSD With High-Speed Interface," `http://www.i4u.com/article17560.html`.

[67] Intel, "STMicroelectronics Deliver Industry's First Phase Change Memory Prototypes," `http://www.intel.com/pressroom/archive/releases/20080206corp.htm`.

[68] (2008), "Flash Price Drop Spurs Innovation," `http://www.washingtonpost.com/wp-dyn/content/article/2008/02/01/AR2008020101313.html`.

[69] Leventhal, A. (2008), "Can flash memory become the foundation for a new tier in the storage hierarchy?" `http://sun.systemnews.com/articles/125/1/storage/20231`.

[70] Handy, J. (2009), "Flash vs. DRAM Price Projections," http://www.storagesearch.com/ssd-ram-flash%20pricing.html.

[71] Chen, F., D. A. Koufaty, and X. Zhang (2009) "Understanding Intrinsic Characteristics and System Implications of Flash Memory based Solid State Drives," in *Proceedings of the eleventh international joint conference on Measurement and modeling of computer systems (SIGMETRICS)*, pp. 181–192.

[72] Rajimwale, A., V. Prabhakaran, and J. D. Davis (2009) "Block Management in Solid-State Devices," in *Proceedings of the USENIX Annual Technical Conference*.

[73] Schroeder, B. and G. A. Gibson (2007) "Understanding Disk Failure Rates: What does an MTTF of 1,000,000 hours mean to you?" in *Proceedings of the Annual Conference on File and Storage Technology (FAST)*, pp. 1–16.

[74] Gasior, G. (2009), "Six Pack of Solid-State Drives Compared," http://techreport.com/articles.x/16848.

[75] Belady, C. (2007) "In the Data Center, Power and Cooling Costs More Than The IT Equipment it Supports," *ElectronicsCooling*, **13**(1).

[76] "LP SOLVE: Linear Programming Code," http://lpsolve.sourceforge.net/5.5/.

[77] Narayanan, D., "Enterprise-scale I/O Traces from Microsoft Research," ftp://ftp.research.microsoft.com/pub/austind/MSRC-io-traces/.

[78] Intel, "Intel X25-M SATA II, SSD, 80GB," http://www.intel.com/design/flash/nand/mainstream/index.htm.

[79] Seagate, "Seagate Cheetah 15K.5," http://www.seagate.com/docs/pdf/datasheet/disc/ds_cheetah_15k_5.pdf.

[80] Samsung (2009), "Samsung "Green" 1TB Serial ATA 3Gbps Internal Hard Drive," http://www.iconocast.com/EB000000000000030/R4/News1.htm.

[81] Jain, R. (1991) *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*, Wiley-Interscience.

[82] Mearlan, L. (2009), "Analysis: SSD performance – is a slowdown inevitable?" http://www.computerworld.com/s/article/9132668/Analysis_SSD_performance_is_a_slowdown_inevitable_.

[83] Mi, N., A. Riska, E. Smirni, and E. Riedel (2008) "Enhancing Data Availability through Background Activities," in *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, pp. 492–501.

[84] Gurumurthi, S., A. Sivasubramaniam, M. Kandemir, and H. Franke (2003) "DRPM: Dynamic Speed Control for Power Management in Server Class Disks," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pp. 169–179.

Vita

Youngjae Kim

Youngjae Kim did his schooling in Seoul, South Korea. Subsequently, he received his undergraduate B.S. degree in Computer Science and Engineering from Sogang University, Seoul, Korea in the year 2001. During his undergraduate school, he joined military army as an active service for 2 years and 2 months (Aug. 1996 Oct. 1998). After he completed his B.S. in Computer Science at Sogang, he attended KAIST (Korea Advanced Institute of Science and Technology) where he completed a M.S. in Computer Science. Then, he worked at ETRI (Electronics and Telecommunications Research Institute - National Research Lab. in Korea). He expects to get his Ph.D degree in 2009 from the Department of Computer Science and Engineering at the Pennsylvania State University. His research interests are in the design, implementation, and evaluation of file and storage systems, operating systems, and distributed systems and power and thermal management at datacenter.