ABSTRACT

Title of Document: Performance Analysis of
NAND Flash Memory Solid-State Disks

Cagdas Dirik
Doctor of Philosophy, 2009

Directed By: Professor Bruce Jacob
Department of Electrical and Computer Engineering
University of Maryland, College Park

As their prices decline, their storage capacities increase, and their endurance improves, NAND Flash Solid-State Disks (SSD) provide an increasingly attractive alternative to Hard Disk Drives (HDD) for portable computing systems and PCs. HDDs have been an integral component of computing systems for several decades as long-term, non-volatile storage in memory hierarchy. Today's typical hard disk drive is a highly complex electro-mechanical system which is a result of decades of research, development, and fine-tuned engineering. Compared to HDD, flash memory provides a simpler interface, one without the complexities of mechanical parts. On the other hand, today's typical solid-state disk drive is still a complex storage system with its own peculiarities and system problems.

Due to lack of publicly available SSD models, we have developed our NAND flash SSD models and integrated them into DiskSim, which is extensively used in

academe in studying storage system architectures. With our flash memory simulator, we model various solid-state disk architectures for a typical portable computing environment, quantify their performance under real user PC workloads and explore potential for further improvements. We find the following:

- The real limitation to NAND flash memory performance is not its low per-device bandwidth but its internal core interface.
- NAND flash memory media transfer rates do not need to scale up to those of HDDs for good performance.
- SSD organizations that exploit concurrency at both the system and device level improve performance significantly.
- These system- and device-level concurrency mechanisms are, to a significant degree, orthogonal: that is, the performance increase due to one does not come at the expense of the other, as each exploits a different facet of concurrency exhibited within the PC workload.
- SSD performance can be further improved by implementing flash-oriented queuing algorithms, access reordering, and bus ordering algorithms which exploit the flash memory interface and its timing differences between read and write requests.

PERFORMANCE ANALYSIS OF NAND FLASH MEMORY SOLID-STATE DISKS

by

Cagdas Dirik

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park, in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2009

Advisory Committee:

Professor Bruce Jacob, Chair
Professor Donald Yeung
Professor Gang Qu
Professor Jeff Hollingsworth
Professor Prakash Narayan

To my parents,

Zehra and Engin Dirik

Acknowledgements

First, I would like to thank my advisor, Professor Bruce Jacob. This thesis would not have been possible without his guidance, support, and endless patience.

I had the privilege to be friends and work with many great people. Mustafa Tikir has always been there for me through good times and bad times. He always listened to my drama and shared his valuable thoughts. I am very grateful to him for all the things he has done. Professor Yavuz Oruc has been a great mentor to me. He provided his advice throughout my struggles and kept me challenged. Thanks to Okan Kolak, Murat Ungor, Nihal Bayraktar, Betul Atalay, Parag Kulkarni, Samuel Rodriguez, Hongxia Wang, Amol Gole, Akin Akturk, Aydin Balkan, Fusun Yaman, Evren Sirin, Burcu Karagol, Fazil Ayan, Yakup Bayram for their friendship and emotional support.

Special thanks to my wife, Allyson. She had to witness the last years of a PhD student and I am afraid this has irreversibly damaged her perception of graduate school. I am grateful for her endless love and support, without which I could not survive.

Foremost, I would like to thank to my mother and father for their love and support. They have always been there for me, every step of my education. Long time ago they stopped asking me "when I will be graduating". This is the biggest relief for a graduate student. Thank you for believing in me.

Table of Contents

Chapter 1: Introduction

1.1. Problem Description

Flash-based solid-state disks are rapidly becoming a popular alternative to hard disk drives as permanent storage, particularly in netbooks, notebooks and PCs, because of flash's faster read access, low power consumption, small size, shock resistance, and reliability compared to hard disks. SSDs are commercially available in numerous commodity PC models today; they are considered a high-end option due to price-per-bit that is higher than HDDs, but that price gap is closing very quickly.

Flash technology has additional characteristics that have slowed its takeover of hard disks, including a lower bit density relative to HDDs, limited endurance (i.e., its limited number of write cycles), and write performance. Solutions have reached a level of maturity to place flash on a near-term crossover with disks. Rapid migration to later technology has been driving the bit cost of NAND flash significantly lower and its density higher. NAND flash capacity has doubled every year since 2001 and is expected to continue at that rate until 2010; by 2010 it is expected to reach 32/64 Gb single chip density [39, 58, 74]. Over the same period, cost of NAND flash memory has decreased 40-50% per year [69]. In addition, technological enhancements and architectural mechanisms have improved flash memory endurance - currently, NAND flash from several vendors is commercially available with an endurance rating of more than 50 years at 50 GB write per day. Soon, the limit on the number of writes will become a fading memory.

Today's typical hard disk drive is a highly complex electro-mechanical system which is a result of decades of research, development, and fine-tuned engineering. Despite this complexity, extremely detailed and accurate models of HDDs are publicly available [33]. Compared to HDD, flash memory provides a simpler interface, especially one without the complexities of mechanical parts. On the other hand, today's typical solid-state disk drive is still a complex storage system with its own peculiarities and system problems. NAND flash solid-state disks employ multiple flash memory arrays in parallel to increase storage system bandwidth and performance. When multiple flash memory arrays are available, data placement becomes a critical problem for performance and load balancing. Flash memory programming rate is considerably slow and in-place update of data is not allowed. Asymmetric read and write rates make solid-state disk performance more dependent on user workload. Effective wear leveling and block cleaning are two other issues unique to flash memory systems. As it is stated by Agrawal et. at., issues that arise in flash memory solid-state disk design mimic complex system problems that normally appear higher in the storage stack, or even in distributed systems [1].

The relationship between flash memory system organization and its performance is both complex and very significant [1, 40, 24]. Very little has been published on the internals of solid-state disk drives; even less has been published on the performance resulting from various flash memory design options. The most in-depth study to date has been by Agrawal et. al. [1], who analyzed different mapping and ganging/striping policies at the device level (i.e., assuming a flash device exported multiple array-select lines to

enable concurrent access within the device) and ganging at the system level, targeting both enterprise workloads and synthetic workloads. In this dissertation we study the full design space of system-level organization choices for solid-state disks, investigate device-level design trade-offs, and provide a model on how SSDs work. We address the following issues:

- *Concurrency:* By system-level organization we mean the design of the SSD, treating the individual flash devices as constants. Variables in this space include the number of independent busses, their organizations (widths, speeds, etc.), banking strategies, and management heuristics that connect the SSD's flash controller to the flash devices. As shown by Agrawal et al., increasing the level of concurrency in the flash SSD system by striping across the planes within the flash device can amortize the write overhead and increase throughput significantly [1]. Concurrency has been shown in the HDD space to provide tremendous bandwidth increases in interleaved organizations (e.g. RAID). Flash is interesting because unlike disks, its form factor need not change when accommodating interleaved organizations: one can achieve significant levels of concurrency in an SSD without significantly changing its overall size and shape. We investigate the effects of concurrent access to different flash banks via the same channel or by replicating resources and providing multiple independent channels to different flash banks, or by a combination of two.
- *Bandwidth issues:* Common wisdom holds that SSD performance is limited by its media transfer rate. Currently, access to a single flash memory chip is provided by

an 8-bit bus which limits the available bandwidth to 33 MB/s (30 ns bus speed is common) for read access. For write requests, single chip bandwidth can be much lower at 6-10 MB/s due to slow programming time (200 μs for programming 2KB page). As interface transfer rates increase with the introduction of serial I/O interfaces and fiber channel, HDD performance will continue to scale, but SSD performance is expected to be limited by the device's media transfer rate. Samsung's solution to this problem has been to move to a wider and higher performance bus, which can sustain 108 MB/s (16 bit, 54 MHz). Other vendors have followed suit. Two to three years ago, an 8-bit bus clocked at 50 ns was typical, whereas today most flash solid-state disks come with clocks speeds of 20–30 ns. There is also a push by other vendors to improve read/write performance of flash disks by access via 800 MB/s bus in a ring topology [37].

- *Write performance:* Another approach to improving flash performance is to reduce the programming time, thus improving the throughput of write requests. For example, Micron proposed using two-plane flash devices which can simultaneously read and program two pages (2 KBytes each) in the same flash die [59]. This effectively doubles sustainable read and write bandwidth (reported page program performance increases from 8.87 MB/s to 17.64 MB/s). Another approach taken by Micron is combining flash memory blocks into so-called superblocks, enabling the simultaneous read or write of 2 or 4 pages within a flash device or even across different flash dies [57]. This mechanism is similar to Agrawal's ganging and striping mechanisms. Samsung supports similar

architecture to hide programming latency wherein the flash controller controls 2 separate channels and supports 4-way interleaving (write throughput of 30 MB/s is reported) [64, 69].

In this dissertation we address the question; *which of these issues is the most significant - i.e., what approaches to improving solid-state disk drive performance provide the best performance at the lowest cost?* We model various flash solid-state disk architectures for a typical portable computing environment and quantify their performance under diverse user applications such as browsing, listening to music, watching videos, editing pictures, editing text and document creation, office application, and email applications. This dissertation also explores the potential for improvements to SSD organizations by flash oriented heuristics and policies. We study flash oriented queuing algorithms, access reordering, and bus ordering algorithms to accommodate asymmetric nature of read and write requests. We also address the question; *how to optimize the internal I/O access to SSD storage system without significant changes to its physical organization?*

1.2. Contribution and Significance

The contributions of this dissertation are three-fold:

1) We develop a solid-state disk simulator which can be used to measure performance of various NAND flash memory architectures. Our SSD simulator is designed as an extension to DiskSim v2.0 and models a generalized NAND flash solid-state disk by implementing flash specific read, program, erase commands, block cleaning and logical-to-physical address mapping, all while providing the illusion of an HDD. Our simulator is highly configurable and can simulate various solid-state disk architectures

while maintaining a view of a single disk drive to host system. We have used our own disk traces collected from portable computers and PCs running real user workloads to drive the SSD simulator. Our workloads represent typical multi-tasking user activity, which includes browsing files and folders, emailing, text editing and document creation, surfing the web, listening to music and playing movies, editing pictures, and running office applications. These workloads consist of not only I/O traffic generated by user applications, but also I/O read and write requests generated by system and admin processes [24].

2) We study NAND flash SSD architectures and their management techniques, quantifying SSD performance as a function of bandwidth, concurrency, device architecture, and system organization. We explore full design space of system-level organization choices for solid-state disks. Variables in this space include number of flash memory chips, number of independent busses, their organizations (widths, speeds, etc.), banking strategies, and management heuristics that connect the SSD's flash controller to the flash devices. We also investigate device-level design trade-offs as well, including pin bandwidth and I/O width [24]. We find the following:

- The flash memory bus does not need to scale up to HDD I/O speeds for good performance. Average read response times, a good indicator of system-level CPI [40, p. 52], do not improve much beyond 100 MB/s bus bandwidth.
- The real limitation to flash memory performance is not its bus speed but its core interface: the movement of data between the flash device's internal storage array and internal 2 KB data and cache registers.

- SSD organizations that exploit concurrency at both the system and device level (e.g. RAID-like organizations and Micron-style superblocks) improve performance significantly.
- These system- and device-level concurrency mechanisms are, to a significant degree, orthogonal: that is, the performance increase due to one does not come at the expense of the other, as each exploits a different facet of concurrency exhibited within the PC workload.
- NAND flash interface provides drastically different read and write timing which results in large performance disparities between reads and writes. Increasing the level of concurrency in SSD systems amortizes write overhead and increases write throughput significantly. However, asymmetry between reads and writes and the scale factor between their performance persists.

3) We explore the potential for further improvements to SSD organizations by flash oriented heuristics and policies. When distinctive differences between reading from and writing to flash memory and the impact of system- and device-level concurrency techniques are taken into account, there is potential for exploiting the performance disparity between reads and writes. We study flash oriented queueing algorithms, access reordering, and bus ordering algorithms to accommodate asymmetric reads and writes. Specifically:

- *Request scheduling heuristics:* Most disk-scheduling algorithms attempt to reduce seek time, since the majority of time spent in servicing an I/O request in conventional hard disk drives is seek time. Unlike HDDs, flash memory solid-

state disks do not have any mechanical components and therefore have a deterministic, uniform request service time. We show that; even with this limited potential, one can improve SSD storage system performance significantly by implementing flash-specific request-scheduling algorithms that exploit the flash memory interface and its timing differences between read and write requests.

- *I/O bus access policies:* We show that for a typical SSD physical organization (which provides concurrent access to different flash memory banks via the same I/O channel or multiple independent channels to different flash banks, or by a combination of the two), timing of I/O access requests and bus utilization is an important factor in performance. By taking into account the differences between read and write timing and using different I/O access policies, I/O bus utilization can be improved considerably.
- *Data burst size:* With a significant level of request interleaving, I/O bus utilization becomes critical, especially for read requests. By increasing the burst size in transferring data from/to the flash memory array, the number of I/O access requests can be reduced, thereby reducing I/O bus congestion.

1.3. Organization of Dissertation

The dissertation is organized as follows: Chapter 2 provides an overview of NAND flash memory. Characteristics of NAND flash memory is summarized and compared against other types of flash memory. Details of NAND flash memory solid-state disk architectures, including flash memory array organization, NAND flash interface, and flash specific algorithms, are covered in Chapter 3. Chapter 4 discusses related works for

the dissertation. Chapter 5 presents the methodology followed in the dissertation. Details of the SSD simulator designed and parameters for the simulations performed are also covered in chapter 5. Chapter 6 discusses the experimental results, mainly on the performance of NAND flash memory solid-state disks. Chapter 7 provides the conclusion to the dissertation.

Chapter 2: Overview of NAND Flash Memory

Flash memory is a type of electrically erasable programmable read only memory (EEROM) invented by Dr. Fujio Masuoka in the 1980s while working at Toshiba. Main characteristics of flash memory, which differentiate it from other types of EEPROM, are its ability to program in large blocks and its low cost per bit. NAND type flash memory was first introduced by Toshiba in the late 1980s, following NOR type flash memory by Intel [63]. Although other types of flash memory have been developed, NAND and NOR types are the two dominant ones in volume production. Starting from mid 1990s development of battery operated portable electronic appliances, such as PDAs and mobile phones, dramatically increased the popularity of flash memory and its market share. Driven by personal computer market and portable communications systems, flash memory will continue rise in popularity.

2.1. Non-Volatile Memory

Semiconductor memories can be grouped in two categories: volatile and non-volatile. Content of volatile memory (e.g., Random Access Memory) can be changed fast, easy, and unlimited number of times, but it is lost when the power is switched off. On the other hand, non-volatile memory (e.g., Read Only Memory and flash memory) can retain its content even when it is not powered. Early designs of non-volatile semiconductor memory were fabricated with permanent data and did not provide the ability to modify data content. Current designs can be erased and re-programmed a number of times although at a comparatively slow speed compared to RAM [12].

The very first non-volatile ROM was the mask-programmed ROM which was programmed during fabrication and could not be changed. Then in 1956, PROM was invented. PROM allowed engineers to program its content by using silicon or metal fuses. These fuses could be blown by a programmer to change the state of a memory cell from 0 to 1 exactly once. In the early 1970s EPROM was invented, which can be erased by exposure to ultraviolet light and programmed by applying high voltage. EPROM uses one transistor memory cell, thus it is a high density and low cost non-volatile memory. For example, early PC designs employed EPROM as their BIOS chip. In the 1980s EEPROM (Electrically erasable programmable ROM) introduced electrical erase capacity at byte granularity. Although a single byte could not be rewritten an unlimited number of times as in RAM, EEPROM provides good endurance - typically over 1 million program/erase cycle. However, EEPROM uses two transistors per memory cell and cell size cannot be easily scaled, therefore it is expensive and its density is much lower. Usually EEPROM has been used for storing parameters, user data, and configuration settings of a device. Flash memory provides a good compromise between EPROM and EEPROM. Its single transistor per cell architecture provides low cost per bit and high density comparable to EPROM. At the same time its ability to erase and program in large blocks provides flexibility comparable to EEPROM. Due to these characteristics, flash memory can be used both as code and user data storage and it has been successful in delivering to increasing demand for permanent storage driven by personal computer market and portable communications systems [12]. Figure 2.1 provides a comparison of various non-volatile memories.

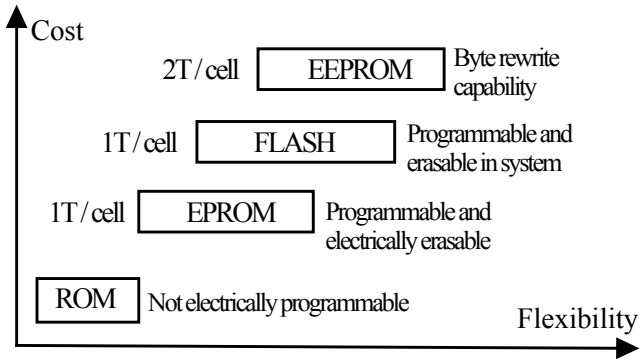


Figure 2.1: Comparison of non-volatile memories. Flash memory provides a good compromise between EPROM and EEPROM and can be used both as code and user data storage. Figure adopted from [12, 5].

Since the introduction of flash memory, many different flash technologies have been proposed. Among these, 4 types of flash memory have been adopted by industry: NOR type, divided bit line NOR (DINOR) type, NAND type, and AND type flash memory. Figure 2.2 shows a historical development of various flash memory technologies. Out of these four types of flash memory, NOR and NAND flash have been dominant in volume production and the most widely used among industry - NOR and NAND flash can be considered as the industry standard. DINOR type flash memory was introduced by Mitsubishi and AND type was introduced by Hitachi [12, 11]. NOR flash is mostly utilized as code and parameter storage in embedded systems due to is high speed random access, and ability to program at byte level. NAND flash is usually used for data storage in memory cards due to its high speed programming and high speed serial access [12]. Most recently, NAND flash solid-state disks (SSDs) are becoming popular as hard disk drive (HDD) replacements in the mobile personal computer market.

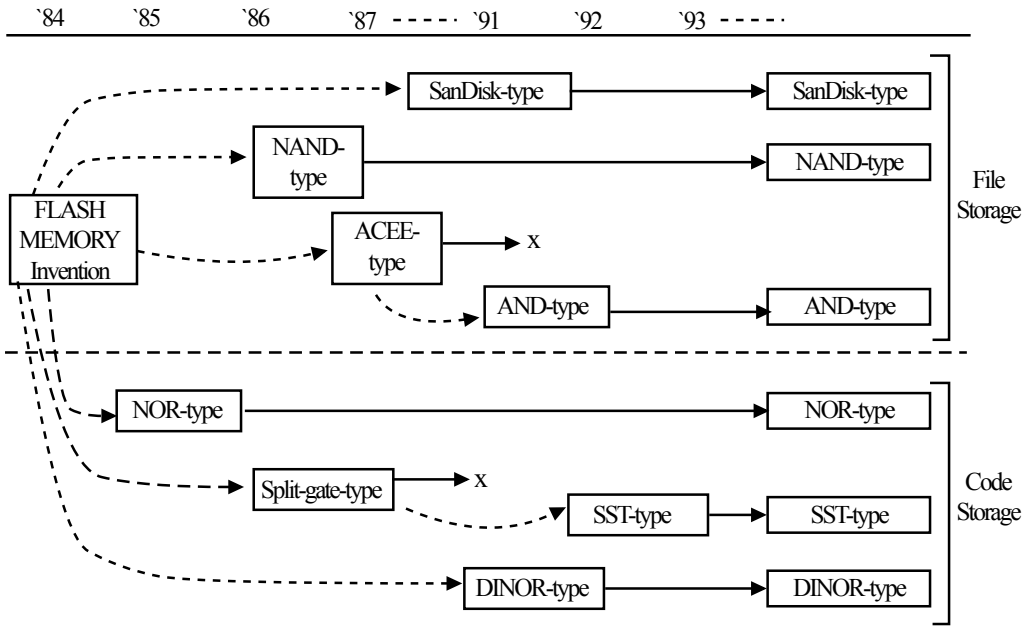
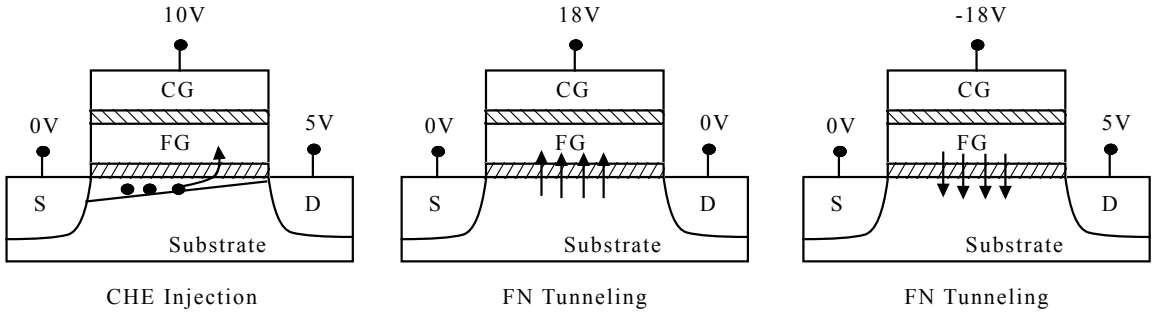


Figure 2.2: History of flash memory technologies. NOR and NAND type have been dominant flash memory types in volume production and the most widely used among industry. Figure adopted from [56].

2.2. Flash Memory Cell

Flash memory cell is a single transistor cell using a dual gate MOS device. A floating gate exists between the control gate and silicon substrate. Floating gate is completely isolated by dielectrics, therefore can trap electrons and keep its charge [5].

For programming memory cell, NOR flash uses channel-hot-electron (CHE) injection while NAND flash uses Fowler-Nordheim (FN) tunneling. With the CHE injection method, MOSFET is properly biased in drain and gate and a large current flows into the cell. Due to this large current, electrons in the channel gain sufficient energy to overcome the gate oxide barrier and get trapped in the floating gate. In FN tunneling, only drain of MOS device is biased and less current is used for programming. Therefore programming by FN tunneling takes longer than CHE injection but allows many cells to



(a) Program (b) Erase

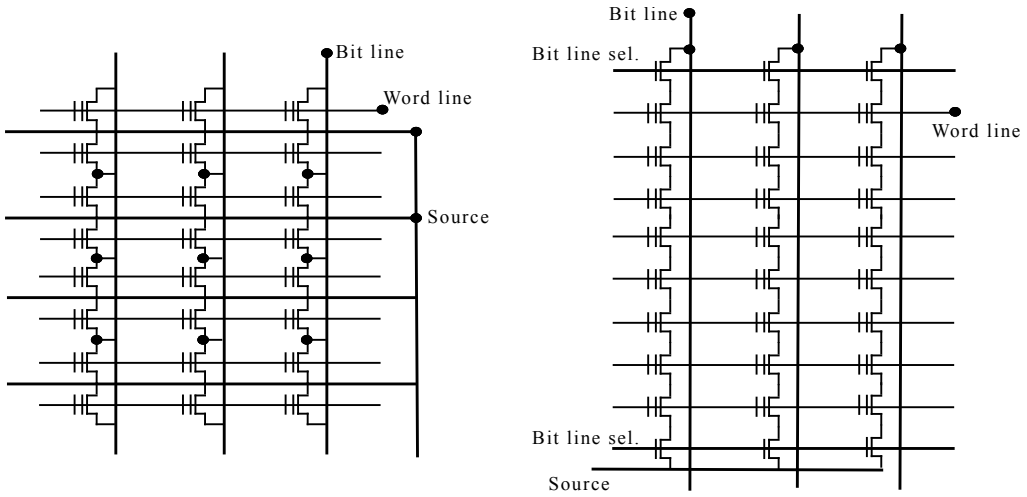
Figure 2.3: Flash memory programming and erase mechanisms. (a) In CHE injection, MOSFET is properly biased in drain and gate and electrons overcome the gate oxide barrier and get trapped in the floating gate. in FN tunneling, only drain of the MOS device is biased and less current is used. Once electrons are trapped in the floating gate, flash memory cell is programmed. (b) To erase flash memory cell, a high electric field is formed across gate oxide. This electric fields causes electrons to depart floating gate. Figure adopted from [78].

be programmed simultaneously. Once electrons are trapped in the floating gate, they cannot escape high energy silicon dioxide barrier even after the device is powered off. When a flash memory cell is programmed, it is considered logic “0” because when it is read it cannot conduct a current due to increased threshold voltage by the trapped charges in the floating gate [11, 63, 5].

In both NAND and NOR flash, cell erasure is performed by FN tunneling. With negative biasing of the cell gate, a high electric field is formed across gate oxide helping trapped electrons to overcome the high energy barrier and depart the floating gate. When a flash memory cell is erased, it is considered storing logic “1” value [11, 63, 5].

2.3. NOR vs. NAND FLASH

In NOR flash memory, memory cells are connected in parallel with common ground node. NOR type array organization as shown in Figure 2.4. Bitlines are formed by memory cells sharing the same drain contact and wordlines are formed by flash cells



(a) NOR array (b) NAND array

Figure 2.4: NOR and NAND flash memory array organizations. (a) In NOR flash memory, cells are connected in parallel. (b) In NAND flash, memory cells are connected in series resulting in increased density. Figure adopted from [12].

sharing gate contact. This array organization provides high speed direct access to a memory cell and high noise immunity - 100% guaranteed good bits [12, 5]. On the other hand, NAND flash memory employs a different array organization. In NAND flash, several memory cells are connected in series between bit line and ground, thus increasing the density compared to NOR flash - e.g., 4-5F^2 NAND flash memory cell size vs. 9-10F^2 memory cell size in NOR flash [63]. Actual sizes of two 64 MB flash memory dies are shown in figure 2.5 for comparison. Larger NOR flash memory cell size is due to bit line contact and ground contact for every two memory cells.

Although series connection of memory cells increases density in NAND flash, it reduces the current for read operation. Reading a single memory cell requires reading other cells in the same bit line, therefore NAND flash memory cannot provide fast random access and is usually employed as a serial memory. Moreover, reduced current in

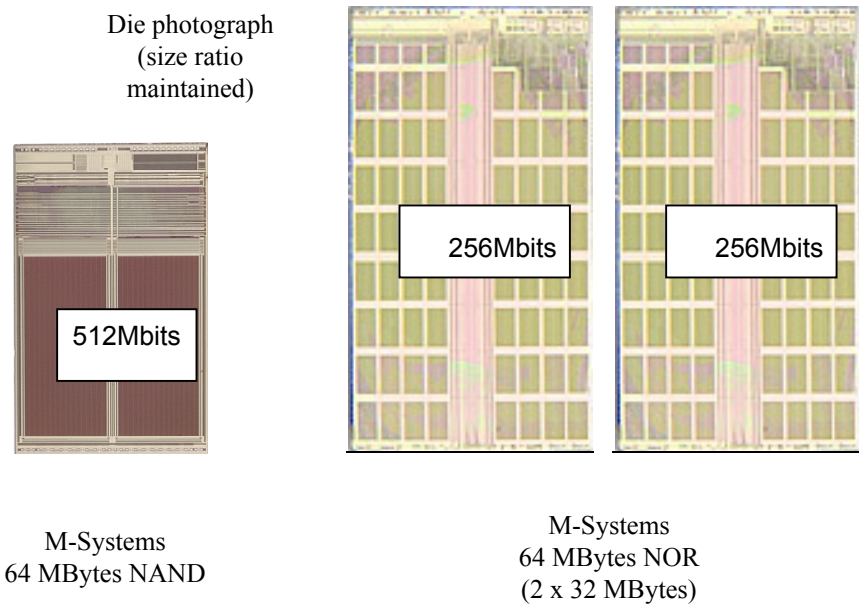


Figure 2.5: NAND and NOR flash density comparison. 64 MB NAND chip is about a third of the size of a 64 MB NOr chip [msystems2003comparison].

read operation makes NAND flash memory much more sensitive to noise and interference [12].

Fast random access time of NOR flash memory makes it ideal for code storage since the execution of code requires branching from one memory location to another. NOR flash memory can provide fully memory mapped random access interface with dedicated address and data lines, which also makes it better suited for code execution [5, 63]. Moreover, NOR flash memory is very reliable and guarantees 100% good bits. This eliminates the possibility of system faults and the need for error detection logic. On the other hand, NAND flash memory is used for data storage due to its serial access characteristics. Also NAND flash memory provides higher storage density at a lower cost compared to NOR flash, which makes it better suited for data storage [5].

Different programming mechanisms and array organizations of NAND and NOR flash memory result in different performance characteristics. Random access (read) time of NOR flash memory is significantly better than NAND - 60 ns compared to 10 μs. NOR flash memory allows writing at byte or word granularity at around 10 μs per byte or word. Although NAND flash memory write speed is much slower than NOR (200 μs per byte), simultaneous programming of cells is allowed. When this ability to program cells in parallel is accounted, NAND flash memory becomes much faster than NOR (200 μs per sector equivalent to 0.4 μs per byte) [78].

Power consumption and endurance of NAND and NOR flash memories is also different due to their programming mechanisms. Programming with CHE injection requires 0.3 to 1 mA of current, whereas FN tunneling uses less than 1 nA per cell for programming. NAND flash memory consumes less power during write operation even though it programs multiple cells in parallel. Since FN tunneling is used both for erase and program operations in NAND flash, its endurance is up to 10 times better compared to NOR flash [11, 63]. Endurance of flash memory is measured in number of cycles a memory cell can be erased and reprogrammed. Endurance cycles of 1,000,000 is typical for NAND flash memory [63]. Table 2.1 provides a comparison of two flash memory technologies.

2.4. Industry Trends

Flash memory is expected to keep its popularity in the digital consumer electronics market. At the same time, NAND type flash memory is expanding into high density storage media market as its bit cost is becoming comparable to the bit cost of

	NOR	NAND
Access	Random	Serial
Read Performance	60 - 120 ns 15 - 30 ns in burst mode	10 - 50 μs 25 - 50 ns in page mode
Write Performance	10 μs/byte or word	200 μs/byte 200 μs/page in page mode (0.4 μs/byte)
Cell Size (F^2)	10	5
Execute in Place (XIP) Capability	Yes	No
Erase Speed	Slow	Fast
Erase Cycles	10 K - 100 K	100 K - 1M
Interface	Full memory interface	I/O interface
Capacity	Low	High
Cost per Bit	High	Low
Active Power	High	Low
Standby Power	Low	High
Reliability	High	Low 1-4 bit EDC/ECC required
System Integration	Easy	Hard

Table 2.1: Comparison of NAND and NOR flash [78, 76, 65].

conventional storage media such as HDD. In 1994, 16 Mbit NAND flash memory was available and today 64 GByte NAND flash solid-state disks are replacing hard disks in high end portable computers. Moore's law in memory suggests two fold increase in density every one and a half years. While NOR flash memory scaling has been according to Moore's law, NAND flash density has been growing at a rate of two fold increase

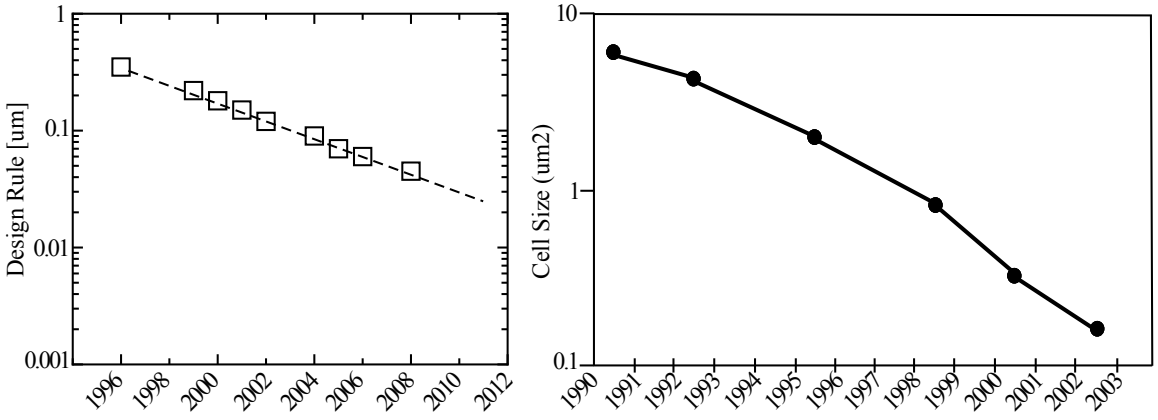


Figure 2.6: NAND flash scaling. Historical trend of NAND flash memory technology and cell size reduction [39, 74]

every year - Hwang's law [39, 74]. Figure 2.6 shows the historical trend of NAND flash scaling. This scaling of technology has reduced memory cell size more than 50 times in 10 years [41]. Major factors helping this aggressive scaling of NAND flash memory are: 30% lithographic shrinkage with each generation; new floating gate structure and device isolation, which scaled the cell area down by 50% in each generation; and increase in the number of memory cells in a string from 16 to 32, which reduced cell overhead by 15% [74]. Latest developments in NAND and NOR flash memory market by major manufacturers are listed in Table 2.2. By 2013, NAND flash memory technology is expected to move beyond 30 nm [6].

Despite this impressive growth, flash memory is also facing its technological challenges in scaling. More severe floating gate interference, lower coupling ratio, and less tolerant charge loss will limit NAND flash capacity to scale. Maintaining narrow erase cell threshold voltage, scaling drain program voltage, and gate length scaling are some of the technical challenges for NOR flash [74].

April 2006	90 nm, 512 Mbit NOR flash memory chips are available from Intel [49]
December 2006	Intel introduces 65 nm, 1 Gbit NOR flash memory chips [54]
March 2007	STMicroelectronics is offering 110 nm 32 Mbit NOR flash memory in automotive grade [75]
October 2007	Samsung develops 30 nm, 64 Gb NAND flash memory. 16 of these memory chips can be combined to achieve 128 GB memory card [27]
December 2007	Hynix will begin mass production of 48 nm, 16 Gb NAND flash memory chips in early 2008 [22]
February 2008	Toshiba and SanDisk co-developed 43 nm 16 Gb NAND flash memory chips (NAND strings of 64 cells aligned in parallel, resulting in 120 mm^2 chip area) [61]
May 2008	Intel and Micron introduced 34 nm 32 Gbit NAND flash chip. Each chips is 172 mm^2 and 16 of them can be combined for 64 GB data storage. [51]
December 2008	Toshiba unveils 512 GB NAND flash solid state disk based on 43 nm technology [73]
December 2008	Spansion ships 65 nm 1Gbit NOR flash memory [77]
January 2009	SanDisk and Toshiba expect to ship 32 nm NAND flash memory by the end of 2009 [52].

Table 2.2: Recent NAND and NOR flash memory developments on the news.

As flash memory is facing these limitations, another way to continue scaling of cell size per bit is offered by multilevel cell (MLC) concept. In a single level cell (SLC), bit 0 is stored if electrons are trapped within the floating gate and bit 1 is stored otherwise. In MLC memory cells, the amount of electrons trapped in the floating gate are precisely controlled. This results in a set of threshold voltages for the memory cell. When the memory cell is read, various threshold voltages cause current value to change within predetermined levels. Thus a MLC memory cell designed with 2^n different levels can

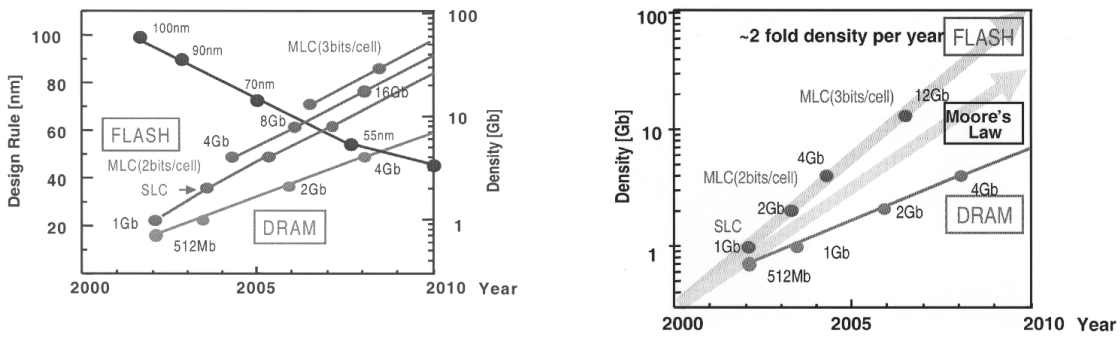


Figure 2.7: High density memory growth. Figure adopted from [39].

store n bits [5]. Figure 2.7 shows the impact of the MLC concept, especially in increasing flash memory density in recent years. Today 8Gbit MLC NAND flash memory is commercially available as a single chip at 0.0093 μm^2 per bit [74]. Figure 2.8 shows various SLC and MLC NAND flash memory chips commercially available from different vendors [78]. Although MLC has the potential to further increase density by using 4 or more bits per memory cell, it increases complexity and introduces new challenges to flash memory design. In order to precisely control the amount of electrons trapped in the floating gate, programming accuracy should be very high. Also reading a memory cell will require high precision current sensors and error correction circuitry. These additional circuitry and the requirement for high precision slows down programming and reading speeds (up to 3 - 4x) compared to SLC flash memory [5]. Moreover, endurance of MLC flash is not as good as SLC flash memory. Typically the number of endurance cycles for MLC flash with 2 bits per cell is 10 times less than SLC flash memory and further goes down with increasing number of bits per cell [28].

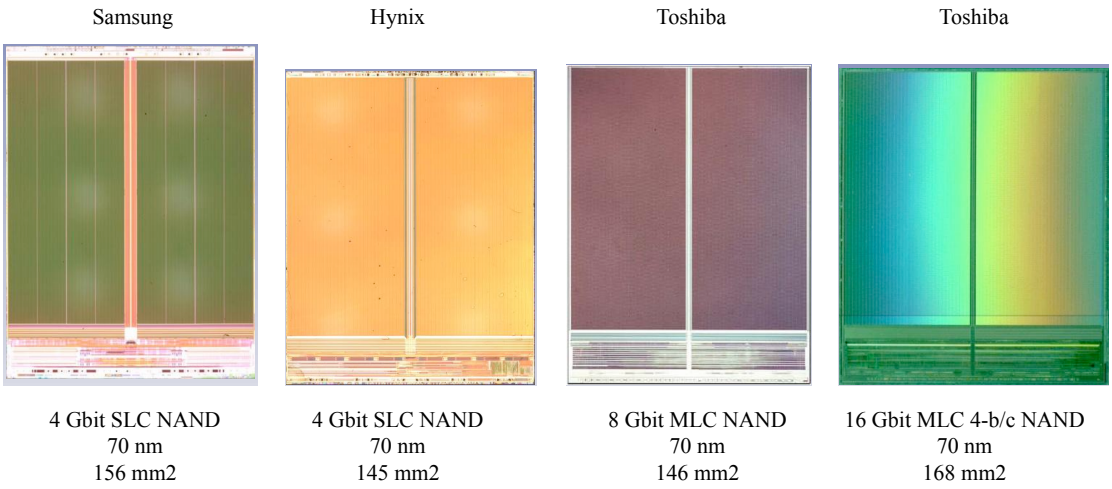


Figure 2.8: SLC and MLC NAND flash chips. Figure adopted from [78].

Flash memory scaling and density increase provides new applications and new directions in the consumer electronics market. For example, scaling of NOR flash enables the design of mobile DDR interfaced memory. Smaller memory cells make it possible to manufacture 1 Gbit or more die size with DDR interface speed, allowing chipsets with single memory controller and common execution bus (DRAM and flash memory operating at the same frequency). This high performance memory architecture can be used in the next generation of mobile phones and devices, which can support newer data transmission standards such as 3G [74]. Another popular item in the consumer electronics market is NAND flash solid-state disk (SSD). As NAND flash scales and its cost per bit decreases, SSD's are becoming a high performance, low power alternative to conventional hard disks (HDD). Today, consumers can buy 64 GB SSD in high end portable computers by paying an additional $400-500. Also NAND flash solid-state disks are in almost all netbooks, which are sold with 4 to 8 GB storage for under $500.

Chapter 3: NAND Flash Solid-State Disks

As NAND flash memory cost per bit declines, its capacity and density increase and endurance improves, NAND flash solid-state disks (SSD) are becoming a viable data storage solution for portable computer systems. Their high performance, low power and shock resistance provide an alternative to hard disk drives (HDD).

3.1. SSD vs. HDD: Cost, Performance and Power Comparison

As mentioned in chapter 2, NAND flash memory capacity has been doubling every year as Hwang's rule suggests. Migration to lower processing nodes and development of multi-level (MLC) cell technology have been driving its bit cost lower while improving its endurance and write performance. Therefore solid-state disks are becoming an attractive replacement for conventional hard disks.

Many manufacturer's are currently offering SSD as an optional upgrade for HDD on high end notebooks for a premium - $400-500 for 64 GB SSD. Although this price is still considerably expensive for a consumer market where the average notebook price is below $800, in a couple of years NAND flash SSD prices are expected to be the same as high end HDDs, as shown in figure 3.1. With this decrease on cost; by 2012, 83 million SSDs are expected to be sold - 35% notebook attachment rate [28]. In the meantime, MLC technology enables high density SSDs. Currently 64 GB SLC solid-state disks and 128 GB MLC solid-state disks are available. Toshiba recently announced its 512 GB NAND flash solid-state disk [73]. High cost and low density have long been considered as two barriers against straightforward adoption of SSD's. Once these barriers are

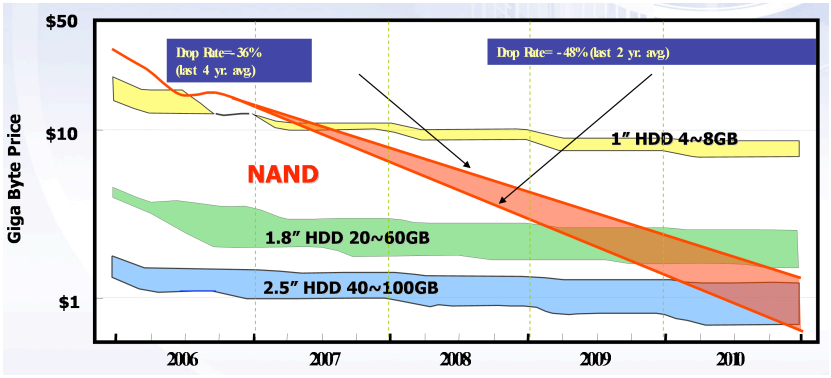


Figure 3.1: SSD and magnetic storage cost projection. Figure adopted from [62].

lowered, solid-state disks performance, reliability and low power consumption generates a great value proposition compared to conventional hard disks.

There are number of reviews available online each of which compares the performance of an SSD against an HDD using various applications as benchmarks. For example; a sample 64 GB SSD with SATA 3 Gb/s interface performs 9 times faster in loading applications (e.g., loading office work, outlook, internet explorer, adobe photoshop) compared to various 7K and 10K RPM hard disks [28]. Another popular benchmark is startup time for Windows Vista. In this benchmark, performance of a solid-state disk is reported to be 3 times better than conventional magnetic disks [28]. In order to better understand the performance difference between conventional hard disks and solid-state disks, we need to take a closer look into how an I/O request is serviced in both systems.

Typical characteristics of I/O traffic in personal computers can be described as bursty, localized in areas of the disk and partitioned 50:50 between reads and writes. Average I/O request size is 7-9 KB and I/O traffic load is estimated to be around 2.4

Mbits per second [38]. We can consider an 8 KB request as a benchmark point and estimate the average request service for a read and a write request. The conventional hard disk drive used for this example is 3.5" Deskstar 7K500 from Hitachi - 500 GB, 7200 RPM. For this HDD; command overhead is 0.3 ms, average seek time is 8.2 msec, write head switch time is 1 msec, average media transfer rate is 48 MB/sec and ATA 133 interface is supported [36]. Given these parameters, the average time for a random 8 KB read with Deskstar 7K500 is

$$0.3 + 8.2 + 4.2 + 0.16 + 0.004 = 12.864 \text{ msec}$$

and for a random 8 KB write is

$$0.3 + 9.2 + 4.2 + 0.16 + 0.004 = 13.864 \text{ msec}$$

As NAND flash memory, we used 16 Gb Micron NAND flash chip. 16 of these chips can be put together to form a 32 GB solid-state disk. For this SSD, read access time is 25 μs and the media transfer rate is 19 MB/s [60]. Given these parameters, the average time for a random 8 KB read with Micron SSD is

$$0.25\ \mu s + 411.2\ \mu s = 0.411 \text{ msec}$$

and for a random 8 KB write is

$$800\ \mu s + 102.8\ \mu s = 0.903 \text{ msec}$$

The significant amount of time spent on hard disk drives are due to mechanical components - seek time and rotational latency. NAND flash memory has an inherent performance advantage over HDD since there are no mechanical components in solid-state disks. Figure 3.2 shows the mechanical and electrical components of an HDD and an SSD. On the other hand, hard disks are considerably better in media transfer rate which can benefit sequential I/O requests. Also I/O traffic generated in personal computers is localized in areas of the disk, which minimizes disk seek time. Hard disk

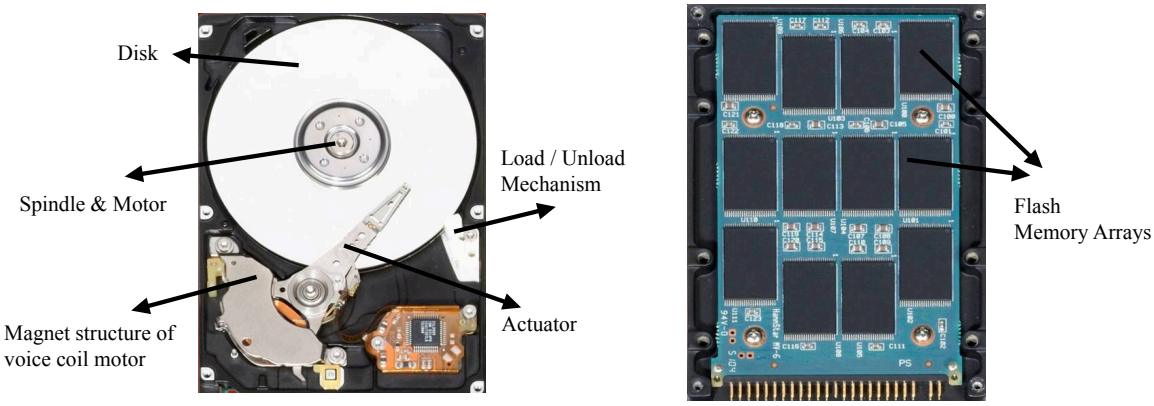
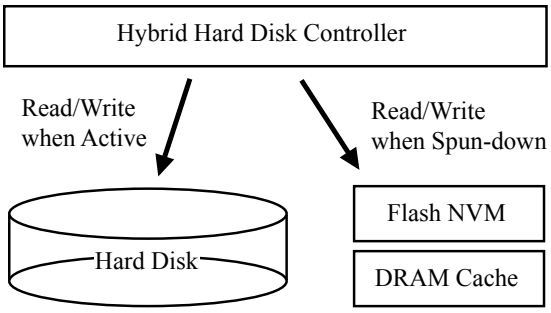


Figure 3.2: HDD vs SSD. Mechanical and electronic components of an HDD and an SSD.

drives also utilize scheduling algorithms which minimize seek times and access times. However, solid-state disk performance is still considerably better than hard disks since they have no mechanical component - essentially zero seek time.

The lack of mechanical components in solid-state disks not only permit better performance, it also results in significantly less power consumption. For example Hitachi Deskstar 7K500 consumes 11-13 W while reading or writing. Its idle average power is 9 W, while it consumes 0.7 W in sleep mode [35]. On the other hand a 64 GB 2.5" Samsung SSD consumes 1 W (200 mA typical active current operating at 5V) in active mode and 0.1 W in sleep mode [66]. Due to its low power consumption, one of the first proposals to use solid-state disks in storage systems was as a non-volatile cache for hard disks. When flash memory cost was high and density was low, hybrid hard disks were proposed for mobile consumer devices. In these hybrid designs, NAND flash memory was used as a high capacity (higher capacity than DRAM based disk buffers) standby buffer for caching and prefetching data, especially when the hard disk was spun down in



	Notebook Drive	Flash NVM
Read / Write	2 W	0.17 W
Idle	1.8 W	2.5 mW
Stand-by	0.2 W	2.5 mW
Capacity	60 GB	256 MB

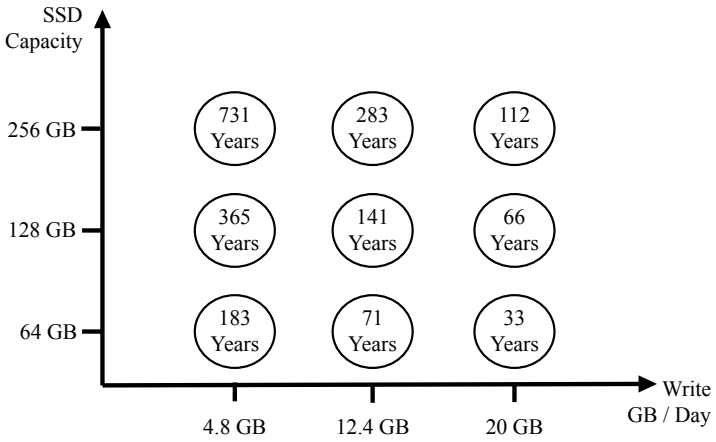
Figure 3.3: Hybrid Hard Disk Organization. An example hybrid hard disk organization with power specifications. Figure adopted from [8]

sleep mode. This way the hard disk idle times were extended, thus saving power [bisson_2006]. Figure 3.3 shows a typical hybrid disk drive organization and its power specifications.

3.2. Endurance: Reality or Myth?

Hard disk drives has always been prone to endurance or reliability problems due to wear and tear of their mechanical components. Hard disk drive manufacturers have implemented various techniques to increase the life span of the rotating media and its components. For example head load and unload zones protects disk during idle times, dynamic bearing motors help reduce vibrations and increase operational shock ratings. Today's hard disk drives typically report three to five years of service life, which is typically defined as the average time period before the probability of mechanical failures substantially increase.

For solid-state disks, reliability and endurance is a different concept. Due to lack of mechanical components solid-state disks are much more reliable and more resistant to shocks compared to hard disk drives. On the other hand, one of the main concerns with



	Writes (GB / Day)	5 Year Writes
Professional	12.4	22.6 TB
Student	5.3	9.6 TB
Personal	4.4	8 TB

Figure 3.4: SSD Endurance. LDE ratings of a typical SSD based on Bapco mobile user ratings. Figure adopted from [28, 4].

flash memory is its endurance. A flash memory cell has a lifetime, it can be erased and reprogrammed a limited number of times, after which it can no longer hold a charge. Consumers fear that once they buy a solid-state disk, it will only last a couple of months - a misconception that has slowed down the adoption of NAND flash solid-state disks.

Endurance of flash memory is measured in the number of cycles - the number of write updates on a memory location. Typical NAND flash endurance is 100,000 to 1,000,000 cycles for SLC type and 10,000 to 100,000 for MLC type. For example, assume we have a 64 GB NAND flash memory with an endurance rating of 100,000 cycles. If we are updating the same 64 MB disk block with a sustained write speed of 64 MB per second, then the lifetime of this solid-state disk would only be 28 hours. Fortunately, flash solid-state disks employ highly optimized wear leveling techniques, which ensure that write updates are written to different physical locations within the disk and each memory cell wears out evenly. When wear leveling techniques are considered, a 64 GB solid-state disk with 100,000 endurance cycles can sustain a 64 MB per second

write load for more than 3 years. Moreover if 1,000,000 endurance cycles are taken into account, this will increase to more than 30 years. One metric used by the industry to specify the endurance rating of a solid-state disk is longterm data endurance (LDE). LDE is defined as the total number of writes allowed in SSD's lifespan [4]. For example, a 32 GB SLC solid-state disk from SanDisk has an LDE spec of 400 TB, and a 64 GB MLC model with 100 TB LDE. If Bapco (Business Applications Performance Corporation) mobile user ratings for a professional is considered, 400 TB LDE corresponds to more than 17 years [4]. A 17 years lifespan for a solid-state disk is much more than a user expected lifespan of a data storage system. With these specifications, one can assume that this limit on the number of writes for NAND flash solid-state disks is theoretical and should not be a concern in the takeover of hard disks. As said by Jim Elliott VP of Marketing from Samsung: "Do you need a million mile auto warranty?" [28].

3.3. SSD Organization

In conventional hard disks, data is stored in disk platters and accessed through read and write heads. Surface number or head number identifies a platter. Each platter has multiple tracks and each track has a number of blocks depending on its zone. Tracks with the same track number in each platter form a cylinder. Access granularity is a sector (block) which has 512 bytes and the location of a block on disk is specified using PBA (Physical Block Address). PBA is formed by a combination of cylinder number, head number, and sector number.

Solid-state disks use a memory array structure different than hard disk drives. NAND flash memory is organized into blocks where each block consists of a fixed

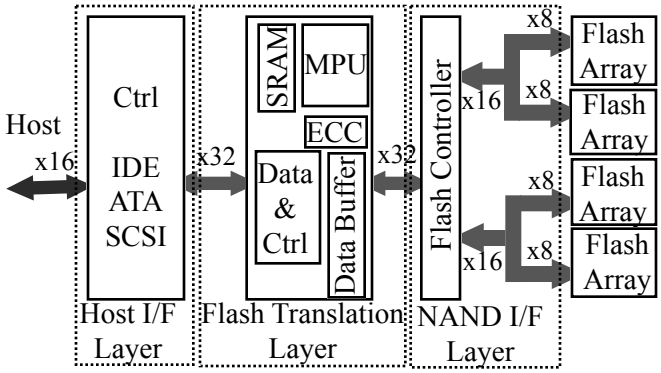


Figure 3.5: Solid-state Disk Organization. Organization of a conventional 32 GB NAND flash SSD. Figure adapted from [64].

number of pages. Each page stores data and corresponding metadata and ECC information. A single page is the smallest read and write unit. Data can be read from and written into this memory array via an 8-bit bus. This 8-bit interface is used both for data, address information, and for issuing commands. Flash memory technology does not allow overwriting of data (in-place update of data is not allowed) since a write operation can only change bits from 1 to 0. To change a memory cell's value from 0 to 1, one has to erase a group of cells first by setting all of them to 1. Also a memory location can be erased a limited number of times, therefore special attention is required to ensure that memory cells are erased uniformly. Despite these differences in storing and accessing data, solid-state disks still assume a block device interface. From host's file system and virtual memory perspective, there is no difference accessing a HDD or a SSD.

Figure 3.5 shows a 32 GB NAND flash solid-state disk architecture from Samsung [64]. Depending on the capacity of the disk, several flash memory arrays are banged together with dedicated or shared I/O bus. An 8 bit I/O bus is an industry wide standard to keep memory chip pin counts the same across different manufacturers. Once

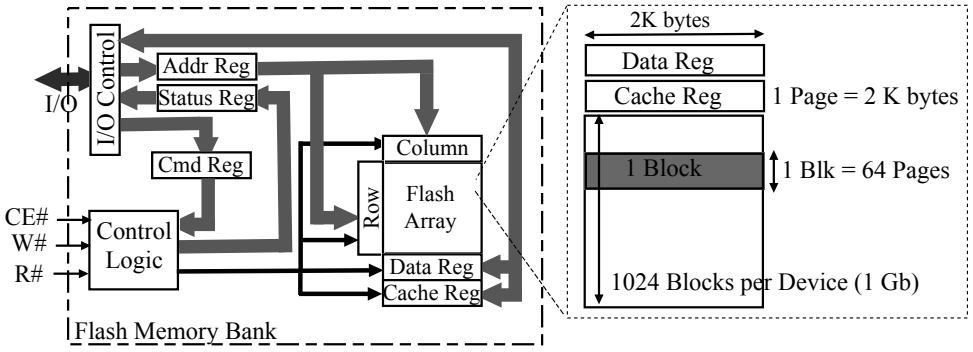
several flash memory arrays are organized to achieve desired capacity and bandwidth, they are accessed through NAND interface layer. This layer consists of a flash controller which implements internal read, write and erase commands and controls timing of address, data, and command lines. NAND interface does not specify dedicated lines for address, data, and command signals as an 8 bit I/O bus is shared for all - chip pin count is limited to reduce cost.

NAND flash solid-state disks assume a block device interface. Currently used interfaces are Fiber Channel (FC), parallel SCSI (Small Computer System Interface), parallel ATA (Advanced Technology Attachment), serial ATA (SATA), and serial attached SCSI (SAS). User's systems (traditionally called host) communicates to the block device through one of these protocols. The host interface layer is responsible for the decoding host system commands and transferring them to a flash translation layer (FTL). FTL layer converts requests' logical block address into physical page address in the flash memory and initiates read/write commands in the NAND interface layer. Address translation is one of the many activities of FTL later. Although flash memory lacks the mechanical complexities of a conventional hard disk, it has its own peculiarities. Since flash memory does not support the in-place update of data, every write request for a specific logical block address results in data to be written to a different physical address with every update. Therefore, logical to physical address mapping in flash memory is much more complicated and requires a dynamically updated address table. Moreover, FTL also implements wear leveling algorithms. Wear leveling ensures that memory cells in an array are equally used - homogeneous distribution of erase cycles. As mentioned before,

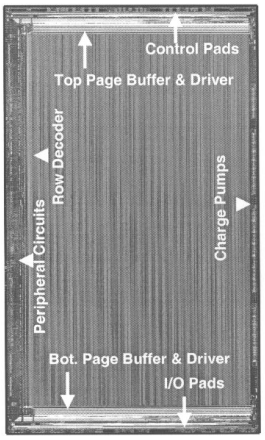
wear leveling is very important in ensuring a long lifespan for solid-state disks. In addition to wear leveling, FTL also implements other features of flash memory such as effective block management, erase unit reclamation, and internal data movements. Also, since there are multiple flash arrays, multiple I/O commands can be processed in parallel for improved performance. FTL layer is responsible in extracting maximum performance by using various types of parallelisms while keeping power consumption and cost at a minimum. One would consider FTL layer as the differentiating factor between different SSD manufacturers as it bundles proprietary firmware.

3.3.1. Flash Memory Array

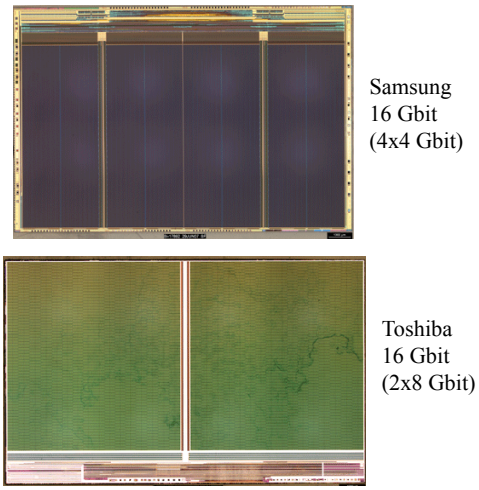
NAND flash memory is organized into blocks where each block consists of a fixed number of pages. Each page stores data and corresponding metadata and ECC information. A single page is the smallest read and write unit. Earlier versions of flash memory had page sizes of 512 Bytes and block sizes of 16 KBytes (32 pages). Currently a typical page size is 2 KBytes (4 sectors of 512 Bytes each), and a typical block size is 128 KBytes (64 pages). The number of blocks and pages vary with the size of the flash memory chip. Earlier flash devices with 512 Bytes page sizes are usually referred to as small-block NAND flash devices and devices with 2 KBytes page sizes are referred to as large-block. In addition to storage cells for data and metadata information, each memory die includes a command register, an address register, a data register, and a cache register. Figure 3.6 shows NAND flash memory array organization for a sample 1 Gbit flash memory from Micron [59]. Larger density flash arrays are usually manufactured by combining several lower density flash arrays in a single die - multiple planes in a die. For



(a) 1 Gbit Flash Memory



(b) 2 Gbit Flash Memory



(c) 16 Gbit Flash Memory

Figure 3.6: Flash Memory Array. (a) NAND flash memory organization. A typical 1 Gb flash memory array consists of 1024 blocks. Each block contains 64 pages of 2 KB each. Figure adapted from [59]. (b) Sample 2 Gbit NAND flash memory in production in 2003 using 90 nm technology [39]. (c) Die photograph of Samsung's 51 nm and Toshiba's 56 nm 16 Gbit NAND flash memory chips [20].

example, 2 Gbit Micron flash memory is two 1 Gbit arrays on a single die (2 planes in a die). Furthermore, two dies can be stacked together to form a 4 Gbit flash memory. These two dies can operate independently or together, depending on the model and configuration. Figure 3.6 also shows 16 Gbit flash devices from Samsung and Toshiba, where multiple planes are visible. Samsung's 16 Gbit NAND flash is MCL type and

manufactured using 51 nm technology. 16 Gbit is divided into 4 4-Gbit arrays. Page buffers are all in one side of the device but bonding pads are located in both sides for better power distribution. Toshiba's 16 Gbit NAND flash is also MLC type but manufactured using 56 nm technology. It uses two 8-Gbit memory arrays. One difference with this device is that data and cache registers are 4 KB instead of the more common 2 KBytes [20]. This suggests that each of two 8-Gbit arrays is indeed a two plane memory array operating in synch - 2 KBytes data and cache registers from both planes are combined and operate as one.

3.3.2. NAND Flash Interface

NAND flash memory supports 3 operations: read, write (program), and erase. 8 bit I/O bus is used for the interface without any dedicated address and command lines. As mentioned before, the smallest access unit for read and write is a page and erase operation is applied to an entire block.

To read a page, one issues a read command to the command register and writes the block number and the page number within the block into the address register. Complete page data (2 KBytes) will be accessed in t_R time and will be loaded into the data register. The typical value for t_R is 25 μs. Afterwards data can be read from data register via 8 bit I/O bus by repeatedly pulsing RE (Read Enable) signal at the maximum t_{RC} rate. Earlier solid-state disks could pulse RE at a rate of 20 MHz. Currently, a 33 or 50 MHz rate is common. In addition to a read command, NAND flash interface also supports random read and read cache mode operations. Random read can be used when only a sector is required from a page. When a page is accessed and loaded into the data

register, a specific sector (512 Bytes) within the page can be addressed and transferred via I/O bus. If sequential pages need to be accessed, the read command can be used in cache mode to increase the data transfer rate. In this mode, when the first page is loaded into the data register, it will be transferred from the data register to the cache register. Typically, copying data from the data register to the cache register takes 3 μs. While data is being read out from the cache register by pulsing RE, subsequent page can be accessed and loaded into the data register. Depending on the manufacturer, read in cache mode can have restrictions. One common limitation is that sequential pages have to be within a block - crossing block boundaries is not permitted. Figure 3.7a-b shows sample timing of read and read cache mode operations.

Similar to a read command, a write or program command has to be issued at the page level, and pages within a block have to be written in sequential order. To program a page, one issues a write command to the command register, writes a block number and page number into the address register, and loads data into the data register. The data will then be programmed into the target page in t_W. The typical value for t_W is 200 μs. To program more than a page, write command can be used in cache mode, which is similar to read in cache mode command. In write cache mode, data is first loaded into the cache register and then transferred from the cache register to the data register. While page is programmed using data from the data register, data for the subsequent page can be loaded into the cache register via 8 bit I/O bus. One of the limitations of page programming is that, pages within a block must be programmed consecutively from the first page of the block to the last page of the block. Figure 3.7c-d shows sample timing of write and write

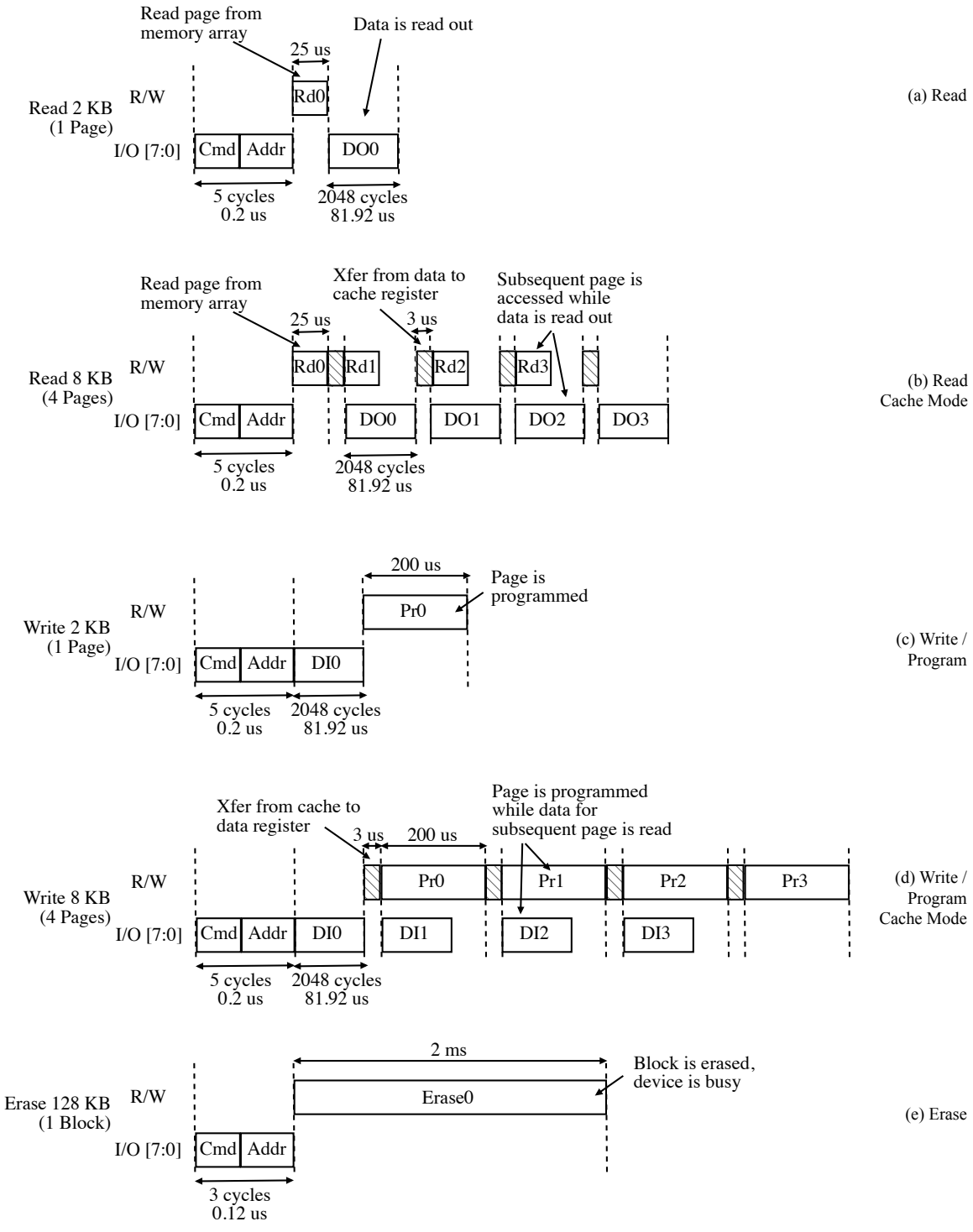


Figure 3.7: NAND Flash Interface. Timing diagram of read and write commands. 25 µs, 200 µs, 40 ns and 2 ms is used as typical values for t_R, t_W, t_{RC} and t_E successively

cache mode operations. Timing of a read request is heavily dependent on I/O bus speed, while timing of write requests is determined by how quickly a page can be programmed.

Write operation in flash memory can only change bit values from 1 to 0. The only way to change bit values from 0 to 1 is by erasing. Unlike read and write commands, erase command can only be performed at block level. Once issued, all bit values in all pages within a block are set to 1. To erase a block, one issues an erase command to the command register and loads the block number into the address register. Flash memory will then set its status to busy for t_E while the erase operation is performed and verified. The typical value for t_E is 2 ms. Figure 3.7e shows sample timing of an erase operation.

3.3.3. LBA-PBA Mapping

One of the limitations of flash memory is that memory cell bit values can only be changed from 0 to 1 by erasing blocks of memory. A typical block size used in current solid-state disks is 128 KB and a sector size for a block device is 512 bytes. Therefore, if a sector within a block needs to be updated, a sequence of operations must be performed. First, the entire 128 KB block is read out to RAM, which takes 5 to 6 ms. Second, the block will be erased to be ready for the update, which takes 2 ms. Then the sector within the block will be updated in RAM and the entire block will be written back to flash memory, which takes almost 13 ms. Given that the average write request size for a typical personal computer disk workload is 7-9 KB [38], this long sequence of events will be repeated with almost every write request. More important, some blocks of flash memory will be erased more often due to frequent localized I/O requests, which will cause flash memory to wear unevenly.

In order to address this performance and wear problem, flash memory solid-state disks do not support the in-place update of data. Rather, every write request for a specific logical address results in data to be written to a different physical address. Therefore, NAND flash memory uses dynamically updated address tables and employs various mapping techniques to match a logical block address requested by the host system to a physical page or block within flash memory. These mapping techniques are implemented at FTL layer and much more complicated than logical to physical address mapping in conventional hard disk drives. Block mapping, page mapping, virtual-physical address mapping, LBA-PBA mapping are all commonly used terms to address these sophisticated mapping algorithms and data structures.

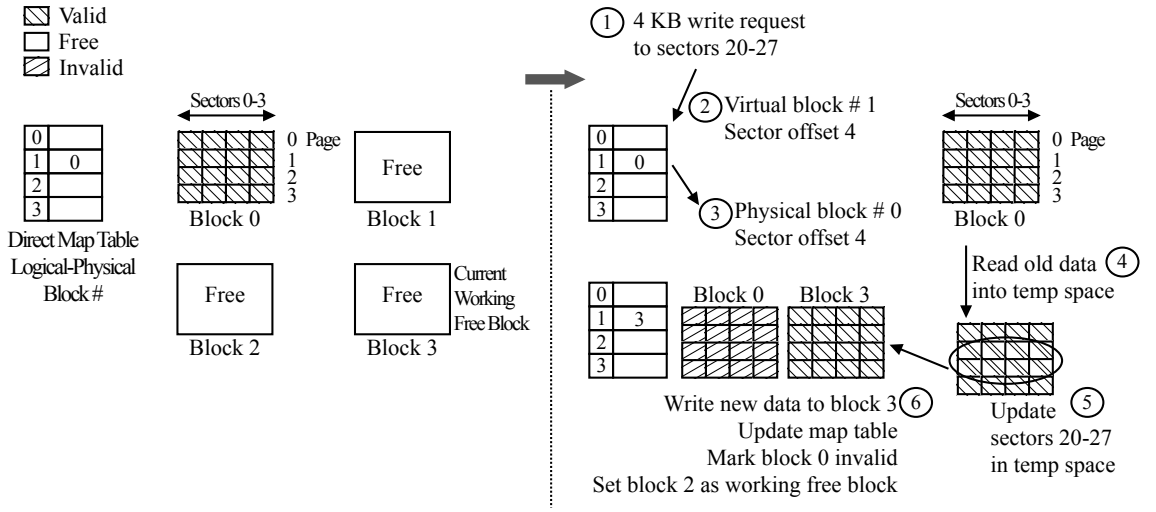
Most of the typical address mapping algorithms use two map tables. A direct map table will provide the physical location of data using its logical address. An inverse map table will store with each block of data its logical index and is used to generate the direct map table.

An inverse map table is distributed and stored in the flash memory with original data using header fields, preamble fields or ECC/CRC fields. Typical sector size for most disk drives is 512 Bytes. In addition to 512 Bytes of data, each sector also stores header information, preamble fields, ECC, and CRC fields. In flash memory solid-state disks each 2 KB page actually consists of 2 KB of data and 64 bytes of reserved space for ECC and metadata information. The logical index for a page can be included into the metadata information. Also in some flash memory implementations, one page in each block - typically the first page or the last page - may be reserved for logical index data. When

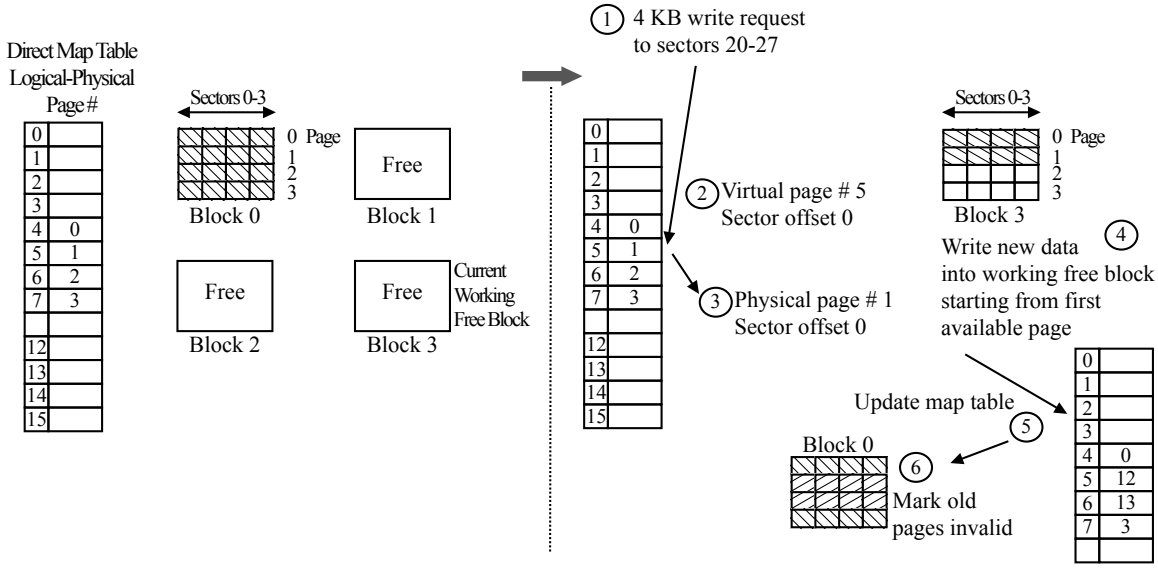
flash memory is powered up, the FTL layer will read through flash memory and use logical index data for each page or block to construct a direct map table.

A direct map table can be stored fully or partially in SRAM. Storing a direct map table fully in RAM would be the ideal case since it will be accessed for each read and write request and fast look-up is essential for performance. On the other hand, SRAM is one of the most expensive components of flash memory and its size is the deciding factor in the cost of a solid-state disk. Depending on performance and cost requirements, direct address mapping is implemented at block granularity or at page granularity or a combination of both.

When block mapping is implemented at block granularity, the logical address of a request is divided into two parts: virtual block number and sector offset within the block. Direct map table is then queried using a virtual block number and mapped to a physical block number. Combination of a physical block number and a sector offset is used to access data for a read request. In the case of a write request, all sectors of the physical block are read, requested sector is updated and data is written back into a new, free block. Afterwards the virtual block number in the direct map table is updated with the new physical block number and the old physical block is marked as invalid. When mapping is implemented at block granularity, only virtual to physical block addresses need to be stored in a direct map table. In a typical 32 GB solid-state disk, assuming the page size is 2 KB and the block size is 64 pages, there are 262144 blocks. This would require 18 bits to store a physical block address. Assuming direct map table is an array which stores physical address of logical block i at its i^{th} location, the size of the table would be less



(a) Block address mapping



(b) Page address mapping

Figure 3.8: Address Mapping. (a) 4 KB write request with address mapping at block granularity. (b) 4KB write request with address mapping at page granularity. (Page sizes of 2 KB - 4 sectors - and block sizes of 4 pages is assumed)

than 1 MB. The cost of storing this direct map table in an SRAM would not be high. On the other hand, performance would suffer because every write request involves reading valid sectors from memory, updating a sector and programming an entire block. Figure

3.8a shows a sequence of events for a sample write request when address mapping is implemented at block granularity.

When block mapping is implemented at page granularity, the logical address of a request is divided into a virtual page number and a sector offset within the page. Upon receiving a read or write request, the direct map table is queried to map the virtual page number to the physical page number. If the request is a write request, a free page is allocated from a working free block. New data is written into this free page while the old page is marked invalid and the direct map table is updated. Typically 2 KB is the standard page size and an average I/O request size is 7-9 KB [38]. This provides a good alignment between I/O requests and page boundaries, which results in better performance. On the other hand, mapping at page granularity would require 48 MB SRAM to store a direct map table for a typical 32 GB solid-state disk.

Although address mapping at page granularity delivers performance, SRAM costs can be too high, especially when solid-state disk densities are increasing beyond 64 GB. Therefore,a hybrid mapping technique is usually implemented. The typical personal computer I/O traffic is highly localized. There are hot spots in the disk accessed by frequent small writes and cold spots which are accessed infrequently by larger requests. Instead of implementing a direct map table as a static array, dynamic data structures can be used for increased flexibility. When a request pattern is detected as hot data, address mapping is implemented at page granularity. For other infrequent cold data, mapping is kept at block level. In some implementations, address mapping may be stored partially in flash memory itself. Address mapping algorithms and techniques for flash memory are an

active research area. There are also several patents filed, which are adopted as standard by industry. A comprehensive survey by Gal and Toledo provides details of various mapping algorithms and techniques used in flash memory [31].

3.3.4. Block Cleaning

Over time as write requests are serviced and new free physical pages are allocated for these write requests, the number of pages with invalid data increases. To service future write requests, blocks with these invalid pages need to be cleaned via the erase operation. This is a block cleaning process and is managed by the FTL layer and flash controller. Block cleaning may also be referred to as block reclamation or garbage collection.

Block cleaning is a long latency operation, which can have significant implications on the performance of a solid-state disk. When a block is claimed for cleaning, all valid user data must be moved before the block is erased. Block erasure is a long but fixed latency operation - typically 1.5 to 3 ms. However, copying valid used data to another location is dependent on the number of valid pages and the location of the target block. If valid user pages can be moved to another free block within the same flash memory chip or plane, fast internal data move operations may be used. Otherwise, valid data has to be read by the flash controller via 8-bit I/O bus and written to another flash memory array. With internal move operation, a page will be read into the cache register and then moved into the data register. While data from the data register is written to a different location, the next page will be read into the cache register. For example, assume a block is claimed for cleaning and half of the pages within the block have valid user data (32 pages out of 64 possible). If the valid user data can be moved to a different block

within the same memory bank or within the same die in a memory bank, the internal data move operation will take 1 page read (read first page), 31 interleaved page read and write operations and 1 page write (write last page). Assuming 25 μs for page read, 200 μs for page write and 3 μs for cache-to-date register transfer, it will take 6.521 ms to move valid data. This will add to the already long latency of the block erase operation. If copying of the valid data cannot be performed within the same memory bank or die, then data has to be read and written via 8-bit I/O interface, which will take even longer than 6.521 ms.

Firmware implemented at FTL or flash controller layer decides the timing of the block cleaning process, which blocks to claim for cleaning and where to move the valid user data in the reclaimed blocks. This is a run-time resource allocation problem with many constraints and several block cleaning algorithms have been suggested in the literature. Baek et. al. provides a comprehensive analysis of block cleaning, a detailed model for cost of block cleaning and an approach of clustering user data into hot and cold to reduce this cost [2, 3].

3.3.5. Wear Leveling

As mentioned in section 3.2., one of the main concerns with flash memory has been its endurance. Wear leveling techniques have been critical in overcoming this concern. Wear leveling is also a constraint in the block cleaning process - in deciding which block to claim for cleaning.

The main goal of wear leveling is to make sure that frequent localized write requests do not result in some blocks being erased more often, which will cause flash memory to wear unevenly. The ideal is all blocks are equally used - homogeneous

distribution of erase cycles among blocks. It is important to note that wear leveling and block cleaning place contrary restrictions on block management. Block cleaning requires some form of hot and cold data clustering so that frequent localized write requests may be separated from infrequent updates to reduce the cost of cleaning [2, 3]. But such clustering of data also results in frequent erasure for blocks storing frequently updated data, resulting in a non-homogeneous distribution of erase cycles. An efficient wear leveling algorithm for large scale flash memory systems is presented by Chang [17].

Chapter 4: Related Work

Prior research on flash memory systems has mostly been on the design of various FTL algorithms and the development of hybrid memory systems. There has been limited publications on the internals of solid-state disks and their performance characteristics. In this chapter, we will first discuss testing methodologies and simulation techniques used in prior research on flash memory systems. Second, we will discuss previous studies on the performance of flash memory systems and the interplay between memory architecture and performance. The rest of the chapter will be on hybrid memory systems and flash memory data structures and algorithms, such as page mapping, block cleaning, and wear leveling algorithms.

4.1. Flash Memory Simulations

Very little has been published on the internals of solid-state disk drives; less has been published on the performance resulting from various design options. The majority of studies have either been on the development of algorithms and data structures for various FTL functionality (such as address mapping, block cleaning, wear leveling) or the development of hybrid memory systems. In a hybrid setting, flash memory is often utilized as a cache for hard disk drives to reduce overall power consumption of I/O systems. There have been 2 different testing methodologies used in these studies.

One methodology is employing an embedded system or a prototype system with commodity flash memory attached to it. Baek et. al. uses an embedded system with an XScale PXA CPU, SDRAM, and NAND flash memory running Linux kernel and YAFFS to manage flash memory. In their simulations they compare modified YAFFS against

native YAFFS [2, 3]. Bisson and Brandt use a small amount of flash memory (NVCache) in a hybrid disk drive. They evaluate benefits of their hybrid disk drive by interconnectiong flash memory and hard disk drive over USB 2.0 and using a modified Linux kernel. By implementing NVCache in Linux Kernel, I/O requests are intercepted and redirected to flash memory [8, 9]. In his thesis Myers analyzed the use of NAND flash memory in relational databases and used two 32 GB commodity solid-state disks. Both disks are attached to a system using ext2 file system running with Linux kernel with 4 GM RAM [62]. Birrell et. al. uses a testing environment where commodity USB flash memory drives are attached to a Win32 file system. A USB analyzer is used to measure read and write latencies [7]. Dumitru uses a similar approach in testing read and write performance of commodity flash solid-state disks and compares them against commodity hard disk drives [26]. Gray and Fitzgerald use I/O benchmark tools SQLIO.exe and DiskSpd.exe in testing random read and write performance of a beta NAND flash 32 GB solid-state disk. These bechmark tools are used in generating 1, 2 or 4 outstanding I/O requests with variable block sizes [34]. In a similar study, Kim and Ahn use a real hardware prototype system with MLC NAND flash memory and test read/write performance using a series of benchmark tools such as PCMark, IOMeter, and Postmark [44]. Park et. al. uses a third party benchmark tools to compare performance of commodity HDDs and SSDs. Both disk drives are attached to a Samsung notebook [69]. Yoon et. al. uses a prototype system of their Chameleon Hybrid SSD architecture. This prototype system includes 4 flash memory modules attached to a development board together with an ARM7TDMI test-chip running at 20 MHz, a Xilinx Virte II FPGA chip

(implements FTL), and 64 KB SRAM. PCMark benchmark tool is used for testing [80]. In a more detailed study, Parthey and Baumgartl rely on black-box testing of 19 different commodity flash memory drives. The drives are connected via USB 2.0 card reader to an AMD64 Athlon computer running Linux [70]. Although testing commodity flash memory drives provides valuable information on the performance of real systems, the amount of information available is often times very limited. Without the knowledge of the internal workings of the item tested it is hard to identify design parameters and their impact on overall performance of the I/O system. On the other hand more information can be obtained by simulations.

Simulating an I/O system is flexible, detailed, convenient, and provides intuition. However there has not been any publicly available simulators for flash memory systems. In their study on I/O performance optimization techniques for hybrid hard disk drives, Kim et. al. utilized a hybrid disk simulator, SimHybrid. SimHybrid consists of a hybrid hard disk controller, DRAM device model, a disk device model and SimFlash which is a flash memory simulator. SimFlash models a 1GB NAND flash memory [48]. SimFlash's default flash memory of 1 GB does not scale to the capacity of solid-state disk drives and there is no information on the availability of it. In analyzing use of flash memory file cache for low power web servers, Kgil and Mudge also use a 1 GB flash memory model and integrate it as a page cache into M5, full system architectural simulator [43]. Other studies which focus entirely on flash memory use a series of simulators for performance verification [15, 16, 81, 17, 13, 25, 47, 18, 19]. There is no information on the implementation details of these simulators or on their availability. This suggests that these

simulators are only limited to testing and the verification of specific data structures and algorithms proposed in their respective studies. Therefore, we intend to develop a NAND flash memory simulator to be used in studying any solid-state disk storage system architecture, all while providing the illusion of representing a hard disk drive to the host system.

We have modeled our flash memory code and integrated it into DiskSim v2.0. DiskSim is an efficient and accurate disk system simulator and can be easily ported into any full-system simulator [32]. Our modified version of DiskSim can simulate a generalized NAND flash SSD by implementing flash specific read, program, erase commands, block cleaning, LBN-PBN mapping. Our default model simulates a 32 GB NAND flash SSD and can easily be extended beyond 32 GB capacity. More details of our NAND flash solid-state disk simulator is available in chapter 5. In a parallel effort, another extension of the DiskSim simulator for SSDs has been made available from Microsoft Research [71]. This SSD extension also provides an idealized NAND flash SSD model and provides limited support [1]. Our flash memory SSD simulator is based on DiskSim v.2.0 and Microsoft Research's SSD Extension is based on DiskSim v4.0. Other similarities or differences between our flash memory simulator and this SSD extension for DiskSim is not available to our knowledge.

4.2. Flash Memory Architectures and Performance

What has received relatively little attention is the interplay between SSD organization and performance, including write performance. As previous studies have shown [23, 1], the relationship between memory-system organization and its performance is both

complex and very significant. Very little has been published on the internals of solid-state disk drives; less has been published on the performance resulting from various design options. Min and Nam described basics of flash memory and its technological trends in [58]. They also outlined various enhancements in the performance of flash memory such as write request interleaving and need for higher bus bandwidth. Request interleaving can be implemented within a chip or across multiple chips and will also benefit read and erase latency. Higher bus bandwidth is another technique which is already being employed by OneNAND bus architecture - 16-bit 108 MBps bus instead of a typical 8-bit 33 MBps [45]. A third technique highlighted in their study is the utilization of a dedicated communication path between host interface and flash interface which frees up system bandwidth. Birrell et. al. investigated write performance of flash disks and identified increased latency for non-sequential writes by running micro-benchmarks for commodity USB flash drives [7]. This increased latency of random writes (non-sequential write requests) is due to the difference between how disk space is addressed linearly by logical block address and how data is actually laid out in pages and blocks in flash memory. In a similar study Gray and Fitzgerald tested 32 GB Flash SSD from Samsung and reported average request time of 37 msec for 8 KB non-sequential writes [34]. Their study states benchmark test results but does not explain any of their findings. [26] provides a comparison of Flash SSD's from various vendors and suggests techniques such as write caching to improve performance. OS write caching, flash specific file systems, drive write caching, and block remapping are some of the proposed techniques, although evaluations of these proposed solutions are not available. Kim and Ahn implemented a

RAM buffer (similar to write buffers in hard disks) to improve latency of random writes in flash memory [44]. This write buffer is placed between the host buffer cache and the flash translation layer. They have also implemented three buffer management algorithms on a 1 GB NAND flash memory prototype on a target board with an ARM940T processor, 32 MB SDRAM, and a USB 2.0 interface. Although their prototype implements a USB flash memory instead of a larger capacity solid-state disk drive. Park et. al. provides details on the existing hardware of NAND solid-state disks and their multi-chip architecture. One of the important aspects of this architecture is the flash controller which can support 2 I/O channels and up to 4-way interleaving. Request interleaving has been an effective way in hiding request latencies. Within flash memory systems, it can be used to support parallel write requests for improved performance. Their study also provides a concise discussion of software architectures in NAND solid-state disks [69]. Another way to improve request latencies, especially with write requests, is using a bank assignment policy to utilize the multi-chip architecture. A static assignment policy is striping, which assigns write request to bank number N = LBA(mod *number of banks*). This is almost identical to RAID-0. For user workloads with heavy locality, this static bank assignment policy will result in an un-even distribution. Adaptive bank scheduling policies are proposed to provide an even distribution of write request to boost performance [14, 13].

One of the more recent and detailed studies on the performance of solid-state disks is [1]. Agrawal et. al. provides a detailed discussion on design tradeoffs for NAND flash SSDs by performing simulations using Microsoft Research's SSD extension to

DiskSim. Their work analyzes different SSD organizations using synthetic workloads and enterprise traces, and concludes that serial interface to flash memory is a bottleneck for performance. By employing parallelism within a flash memory package and interleaving requests to a flash memory die, the overall system bandwidth is doubled. Although this study is the most similar to ours, there are major differences on the methodology and areas of investigation. One of their conclusions is that SSD performance is highly workload sensitive; performance differs substantially if write requests are sequential or random (their synthetic traces are largely sequential; their enterprise traces are largely random; the performance improvements shown for the synthetic traces are far more significant than those shown for the real-world traces). Additionally, the workloads used in their study are read oriented, with roughly a 2:1 read-to-write ratio, which helps to hide the problem of slow writes in an SSD. However, in PC applications (user-driven workloads), there tends to be a much higher proportion of writes: in our workloads, we see a 50:50 ratio, which would tend to expose flash's write problem. User driven workloads are not biased towards sequential or random requests but provide a mix of random and sequential writes at a given time interval. Agrawal's study outlines core limitations of flash memory within the boundaries of a flash memory device/package; limitations such as logical to physical mapping granularity, limited serial interface, block erasure, cleaning frequency, and wear leveling. Our study extends their work by focusing on exploiting concurrency in SSD organizations at both the system and device level (e.g. RAID-like organizations and Micron-style superblocks). These system- and device-level concurrency mechanisms are, to a significant degree, orthogonal: that is, the performance

increase due to one does not come at the expense of the other, as each exploits a different facet of concurrency available within SSD organizations.

4.3. Hybrid Memory Systems

One of the main attractions of flash memory is its low power consumption. There have been several proposals to use flash memory in various parts of the memory and/or storage system to reduce overall power consumption. Bisson and Brandt integrated flash memory into hard disk drives [8, 9]. Flash memory in these hybrid disks are referred to as NVCache. When there is NVCache, I/O scheduler can redirect some I/O requests (for example long latency write requests) to flash memory to reduce overall request latency. This will also reduce power consumption as hard disk drives can be spun down for longer periods of time and expensive spin-up operations can be avoided. One limitation with using NVCache in hybrid disks is the additional cost of flash memory. Due to very low cost margins in disk drives, even adding a couple hundred MB flash memory into storage systems has a big impact on cost and may limit use and benefits of NVCache. Another area where flash memory can improve performance of conventional hard disks is in system boot time or application start time. For frequently used applications or applications which generate a significant amount of random I/O requests (which are troublesome for hard disks with long seek latency), data can be stored in flash memory or NVCache for better performance. This is often referred to as data pinning into flash memory. Host OS decides on which data to pin or provides an interface to the user for selecting applications to be pinned. If capacity allows, even the entire OS code and data can be pinned to flash memory for fast boot times. Kim et. al. provides a discussion and

evaluation of I/O optimization techniques for hybrid hard disk based systems [48]. In a similar study Chen et. al. also uses flash memory as a buffer for data prefetching and write caching [18]. The main goal of their algorithms on prefetching and caching data is to reduce power consumption of I/O systems by effectively extending disk idle periods. In [47], hard disk and flash memory is combined for an energy efficient mobile storage system. This hybrid system consumes 30-40% less power.

Instead of using flash memory as a caching mechanism in hybrid disk drives, Kgil and Mudge utilize it as a file cache in main memory [43]. Their proposal replaces 1 GB DRAM main memory with 128 MB DRAM and 1 GB Flash Memory combination. This FlashCache architecture is especially fitting for web servers. Typical web server workloads are not computationally intensive and they are heavily read biased. This requires use of large amounts of DRAM in web servers to mitigate I/O latency and bandwidth. However this significantly increased power consumption of the system even when it is idle. By replacing portions of DRAM with flash memory as a secondary file cache, power consumption of web server can be significantly reduced. The read heavy nature of web server workloads help in hiding write latency problems of flash memory and allows read performance to be effective.

4.4. Flash Memory Data Structures and Algorithms

Compared to hard disk drives, flash memory provides a simpler read/write interface, one without the complexities of mechanical parts. On the other hand, flash memory has its own peculiarities of not allowing in-place data updates, logical-physical address mapping, block erasing and wear leveling. To use flash memory as a storage device, one

needs to hide these peculiarities from the host system since the file system and virtual memory systems assume a block device interface when accessing the storage system. For this purpose, Flash SSDs implement a software layer called Flash Translation Layer and provide an illusion of HDD to host systems. Most studies on flash memory have been on the efficient implementation of these various flash translation layer functionality.

Park et. al. provides an overview of hardware architecture for a flash memory controller and summarizes basic functionality implemented at flash translation layer [69]. As mentioned in this study logical to physical address mapping can be performed at page level or block level. When implemented at page level write requests can be performed at page programming latency. However a large map table is required. If address mapping is implemented at block level, the size of the table is considerably reduced at the cost of increased write latency. In typical SSD architectures, hybrid mapping is performed, which is a combination of page and block mapping. In hybrid mapping, address space is typically divided into regions and either page or block mapping is implemented within a region. A very detailed understanding of their page mapping, block mapping and hybrid mapping schemes are available in [68].

Similar to hybrid mapping, a log block scheme is proposed in [46]. In this scheme some blocks are marked as log blocks and are mapped at page-level. Small, frequent writes are forwarded into log blocks but a majority of user data is kept in data blocks. Blocks which are addressed at block level are called data blocks. By keeping number of log blocks limited, address table size can be kept small. Log blocks can be converted into data blocks by a merge operation.

	Bytes for Addressing	Total Map Table Size
Page mapping	3 Bytes	3B * 8192 * 32 = 768 KB
Block mapping	2 Bytes	2B * 8192 = 16 KB
Hybrid mapping	(2 + 1) Bytes	2B * 8192 + 1B * 32 * 8192 = 272 KB

Table 4.1: Mapping Granularity. Table size to store mapping information for a sample 128 MB flash memory. Page size of 1 sector and block sizes of 32 pages is assumed. Flash memory consists of 8192 blocks. Table adopted from [21].

A comparison of mapping at different levels is provided by [21]. As highlighted in this study, additional data copy operations in block mapping imposes high cost. On the other hand, block mapping requires only limited space to store mapping information. An example given in this study compares mapping table sizes for 128 MB flash memory. As shown in table 4.1, block mapping requires the smallest SRAM to store mapping information. Chung et. al. also explains how this mapping information is managed using either map block method or per block method.

Another clustering based hybrid mapping technique is proposed by Chang and Kuo [15]. Based on user access patterns on a 20 GB hard disk, address space is divided into physical clusters. Physical clusters are stored in main memory in a tree based data structure and logical to physical address mapping is performed using a hash-based approach. This hash-based approach can be configured for performance [50]. For a 16 GB flash memory, their approach required 2.95 MB for mapping information. Mapping table would be 256 MB if page level mapping was used and 8 MB if block level mapping was used [16].

Another constraint with logical and physical address mapping is time to construct a mapping table when the device is powered up. With USB drives this was not a big issue, however as flash memory size increases to 16 GB, 32 GB or to 256 GB with solid-state disks, scanning pages to generate a mapping table at start-up time is not an easy task. Birrell et. al. proposes an address mapping technique which optimizes time to build the data structures and the mapping table [7]. Their approach uses mapping at page granularity for better write speed and can reconstruct mapping tables within a couple of seconds. Moreover their study explains in great detail all components of a mapping table. They divide mapping data into information stored in the flash memory itself and data held in RAM by the flash controller. They assume page sizes of 2KB and block sizes of 64 pages. For the 512 MByte flash memory modeled in their study, there are 4K blocks and 256K pages. The main mapping table is called an LBA table which is a 256K entry array. When indexed by logical address, LBA table maps logical disk address to flash page address. Each array element uses 22 bits; 18 bits for the flash page address and 4 bytes to indicate whether each of the 4 sectors within a page is valid or not. In addition to 256K entry LBA table, a smaller 4K entry table is used to keep block usage information. Indexed by block number, each entry contains the number of valid pages within each block. A 4K bit vector holds the free block list - which is a pool of free blocks ready to be re-used. These arrays are stored in volatile RAM and are managed by flash controller. When the device is powered up they have to be re-constructed from information stored in the flash memory itself. In order to perform fast data re-construction, the last page of each block is reserved to hold summary information. The first 63 pages can still hold user

data. Also 64 bytes reserved for each page is used to store metadata information. Typically logical block address is stored in metadata together with error correction code. In their proposed re-construction algorithm, Birrell et. al. stores additional information in these 64 bytes, which allows them to scan all the physical blocks and pages quickly during power-up. When the device is powered up for the very first time their algorithm takes around 16 sec to re-construct mapping information. Once the necessary data structures are built and metadata information is updated, the re-construction can be performed within couple seconds in successive power-ups [7].

A discussion of flash memory mapping algorithms and data structures is available in [29]. Open problems with logical to physical mapping is also presented in this study.

Logical to physical address mapping is required in flash memory because in-place update of data is not supported. Once a page is written, subsequent writes to the same page cannot proceed because bit values cannot be changed from 0 to 1. Erasing is the only way to change bit values from 0 to 1 in flash memory and can only be performed at block level. Block erasing is also a long latency operation. It is considered as a performance bottleneck and it is important to optimize its efficiency. Baek et. al. provides an understanding of cost of block cleaning [2, 3]. When a block is claimed for cleaning, all valid user data within this block has to be moved into another free location. Once all valid user data is moved, the block can be erased and added to the free block list. Block erasure is a fixed latency operation. However copying valid user data to another location is dependent on the number of pages with valid user data and the location of the target block. If valid user data can be moved to another free block within the same flash

memory chip or plane, fast internal data move operations can be used. To model cost of cleaning, Baek et. al. defines three key parameters [2, 3]:

Utilization (u): Percentage of valid pages in flash memory - indicates percentage of valid pages that need to be copied when a block is erased.

Invalidity (i): Percentage of invalid pages in flash memory - indicates percentage of blocks that are candidates for block cleaning

Uniformity (p): Percentage of blocks which are uniform in flash memory. A block is defined as uniform if it does not contain valid and invalid pages simultaneously.

Given these parameters cost of block cleaning is:

$$cleaningcost = B * ((1-p) + i * p) * e_t + P * (1-p) * (u / (u+1)) * (r_t + w_t)$$

where:

u: utilization ($0 \leq u \leq 1$)

i: invalidity ($0 \leq i \leq 1-u$)

p: uniformity ($0 \leq p \leq 1$)

B: Number of blocks in flash memory

P: Number of pages in flash memory

r_t: Page read latency

w_t: Page program latency

e_t: block erase latency

Their study also discusses the implications of these three parameters and finds that utilization and uniformity have a bigger impact that invalidity. Moreover, cost of block cleaning increases dramatically when uniformity is low. Uniformity is also used as "block cleaning efficiency" to determine the cost of cleaning a single block. Block

cleaning efficiency is defined (slightly different than uniformity) as the ratio of invalid pages to the total pages in a block during block cleaning [1]. Baek et. al. also proposes a page allocation algorithm which increases uniformity to reduce block cleaning costs. The main idea behind their page allocation technique is the ability to distinguish between hot and cold data. This distinction between user requests is common in flash memory community. Hot data is referred to as user data that is modified frequently and cold data is referred to as user data that is modified infrequently. If one can classify hot data, write requests to hot data may be allocated to the same block. Since hot data will be modified frequently, pages within this block will eventually become invalid - a uniform block. Their study proposes an algorithm in identifying hot and cold data and shows that performance may be improved by reducing the number of erase operations and data copy operations during block cleaning [2, 3].

A similar approach of clustering user data into hot and cold is also proposed by [19]. Their study provides an understanding of three stages of the block cleaning process. These three stages are identification of candidate blocks for cleaning, copying of valid pages into free space, and erasing blocks. Performance of the cleaning process is determined by when to clean, which blocks to choose as candidates, and where to copy valid data. Chiang et. al. provides details on policies on each one of these cleaning stages and proposes a new cleaning policy named Cost Age Time (CAT) [19]. Their CAT cleaning policy resulted in 50-60% fewer erase operations as they use a fine-grained system to cluster hot and cold data. Wu and Zwaenepoel also look into block cleaning cost in flash memory for their eNVy storage system [79]. Their study uses a hybrid

cleaning policy, which combines a greedy cleaning policy with locality gathering. Greedy cleaning policy chooses a block with the most number of invalid pages. Locality gathering is the same approach of clustering data into hot and cold regions.

Block cleaning policies not only consider cleaning efficiency and latency, but also wear leveling. As mentioned before, each flash memory cell has a lifetime. Each block in flash memory can be erased for a limited number of times, after which memory cells can no longer hold charge. Although endurance of flash memory increased to 100K or 1M cycles with recent improvements, and solid-state disks can sustain write loads for years with their current capacity, wear leveling algorithms have always been an integral part of block cleaning policies. The main goal of wear leveling is making sure that frequent and localized write requests do not result in some blocks to be erased more often, which will cause flash memory to wear unevenly. The desire is for all blocks to be equally used - homogeneous distribution of erase cycles among blocks. This will ensure a long life span of solid-state disks. However, wear leveling and block cleaning place contrary restrictions on the management of blocks. As explained, efficient block cleaning requires some form of clustering in page allocation so that frequent localized write requests to hot data can be separated from infrequently updated cold data. But such clustering of data also results in frequent erasures for blocks storing hot data, resulting in non-homogeneous distribution of erase cycles. Therefore, block cleaning policies usually not only take into account cleaning efficiency and erase latency, but also consider wear leveling.

An efficient wear leveling algorithm is presented in [81]. The main goal of this algorithm is to ensure that cold data is not stored in a block for very long periods of time.

Block management is performed in a pre-determined time frame, which is called as resetting interval. In a similar study, Chang proposes a wear leveling algorithm for large scale flash memory systems, called dual-pool algorithm [17]. Dual-pool algorithm ensures that blocks are not overly worn by storing infrequently updated data. Once a block participates in wear leveling, it has to wait for some time before being considered for wear leveling again. In order to achieve these, blocks are partitioned into two sets: a hot pool and a cold pool. Cold data is stored in blocks from the cold pool and hot data is stored using blocks from the hot pool. Hot and cold pools are resized adaptively as cold data is slowly moved away from the blocks with low erase cycles to more worn out blocks.

Flash memory solid-state disks emulate a block device interface. A different approach is using a file system specific for flash memory and letting system software manage flash storage [30, 42, 55, 53]. These file systems usually employ a log-structured approach [72]. A survey on flash specific file systems and related patents can be found in [31]. This survey also discusses various sophisticated data structures and algorithms designed to overcome the limitations of flash memory (block mapping, erase-unit reclamation and wear leveling).

Chapter 5: Methodology

One of the contributions of this dissertation is the development of a solid-state disk simulator which can be used to measure the performance of various NAND flash memory architectures. For accurate timing of disk requests our NAND flash SSD simulator is designed as an extension of DiskSim v2.0. Disk traces collected from portable computers and PCs running real user workloads are used to drive this flash simulator.

5.1. DiskSim Disk Simulator

DiskSim is an efficient, accurate disk system simulator from Carnegie Mellon University and has been extensively used in various research projects studying storage subsystem architectures [32]. It was originally developed at the University of Michigan and written in C. DiskSim not only simulates hard disk drives, but also includes modules for secondary components of the storage system such as device drivers, buses, disk controllers, request schedulers, and disk cache components. DiskSim may be used as a trace-driven simulator or can internally generate synthetic workloads. Accuracy of DiskSim v2.0 has been extensively validated against various hard disk drives from different manufacturers, such as Cheetah and Barracuda disk drives from Seagate, Ultrastar 18 ES from IBM, Atlas III, and 10K from Quantum. DiskSim can simulate disk array data organizations and can be integrated into full system simulators. DiskSim only models the performance behavior of disk systems; data is not actually read or written for I/O requests. Additionally, disk power models are available as extensions of DiskSim through other studies [gurumurthi2003drpm-}]. Figure 5.1 shows the storage system components modeled in DiskSim v2.0.

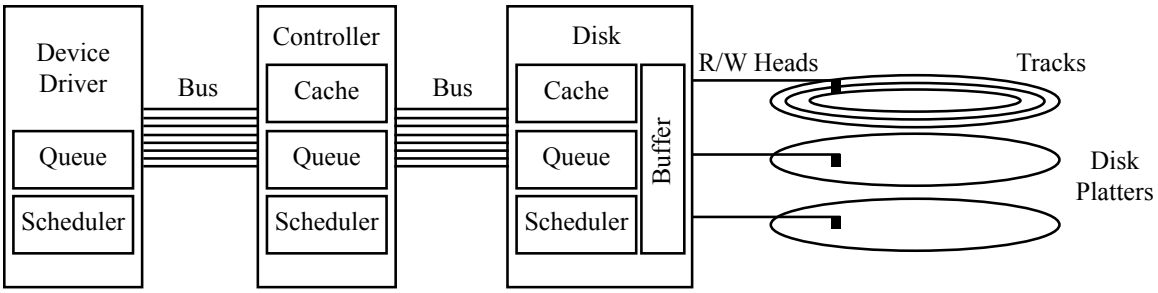


Figure 5.1: DiskSim storage system simulator components. DiskSim implements most components of a storage system; device driver, controller, disk controller with their cache, queue and scheduler components, disk platters and read/write heads.

5.2. NAND Flash Solid-State Disk Simulator

This dissertation uses a solid-state disk simulator, which is designed as an extension of DiskSim v2.0. This simulator models a generalized NAND flash solid-state disk by implementing flash specific read, program, erase commands, block cleaning, and logical-to-physical address mapping, all while providing the illusion of an HDD.

NAND flash interface commands supported by the simulator are page read, page read in cache mode, program page, program page in cache mode, and erase block.

Logical-to-physical address mapping is performed at the granularity of a page. For a 32 GB SSD, this requires the address mapping table to have 16777216 entries. Logical-to-physical address map table is modeled as an int array of size 16777216, which corresponds to 64 MB in table size. Map table is queried by the logical page address and stores the physical page address information and page status information (free/valid/ dirty). Implementation details of the address mapping table is shown in figure 5.2.

The flash simulator maintains a pool of free blocks and allocates pages in a consecutive order from current working free block when a write request is received. In

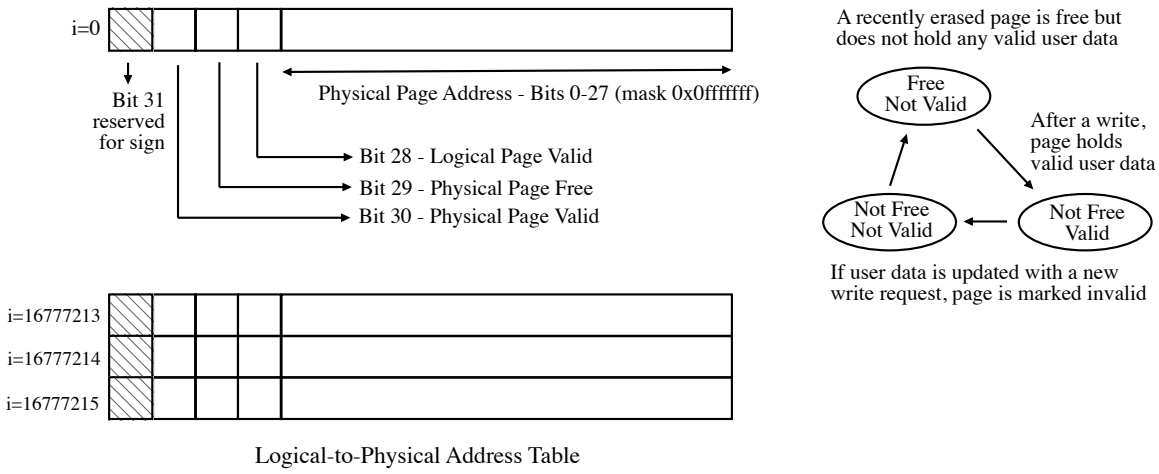


Figure 5.2: Logical-to-physical address mapping. Logical-to-physical address mapping is performed at the granularity of a page. Map table stores physical page address information and page status information.

order to process write requests, the flash simulator must have at least one free block available at all times. Otherwise, block cleaning operation is triggered to erase blocks with invalid pages and add to the list of free blocks. Simulator searches for blocks with the smallest erase latency during block cleaning since block reclamation causes all read and write requests to the device to be stalled in queue. For quick search of blocks eligible for cleaning, the simulator also maintains a list of blocks which include only invalid pages. Blocks with only invalid pages are the blocks with the smallest erase latency. The number of free blocks that trigger garbage collection may be configured within the simulator. Three data structures are used to maintain blocks within the simulator. An int array of 262144 entries (1 MB table size) is used to store block status information. An int array is used as a free block pool with a default size of 16384, which represents 2 GB free disk space. A separate table is used to store disk utilization information which is accessed during garbage collection. This table has 65 entries and stores the number of blocks with *i*

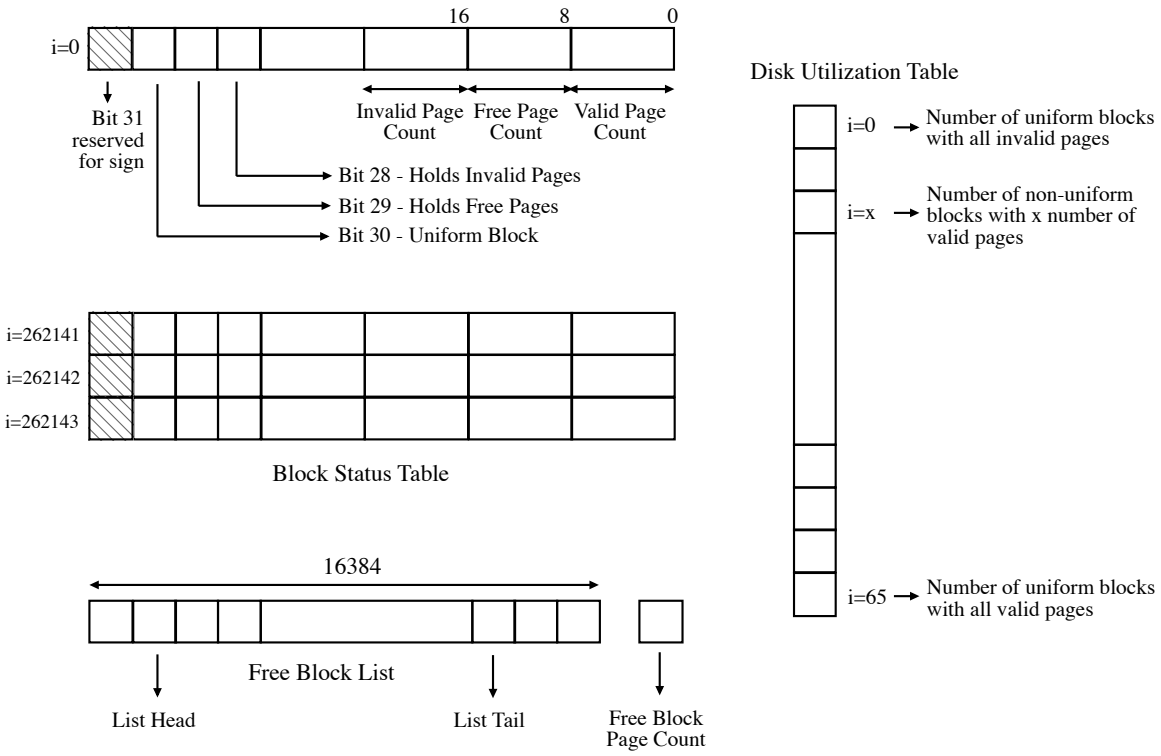


Figure 5.3: Block management. Block status table, free block list and disk utilization table is used to maintain blocks within the simulator

valid pages at location *i*. Implementation details of these data structures are shown in Figure 5.3.

A specific wear leveling algorithm is not implemented. However, write requests are forwarded to the least used physical locations within the address space of the storage system, providing simple wear leveling without overly complicating the performance models used. Figure 5.4 illustrates solid-state disk components modeled in our simulator.

DiskSim is a highly-configurable storage system simulator. Our NAND flash solid-state disk simulator thrives on this property and extends it further. Today's typical solid-state disks are organized as multiple chips connected in different ways using various number of buses. These multiple chips can operate independently or can be

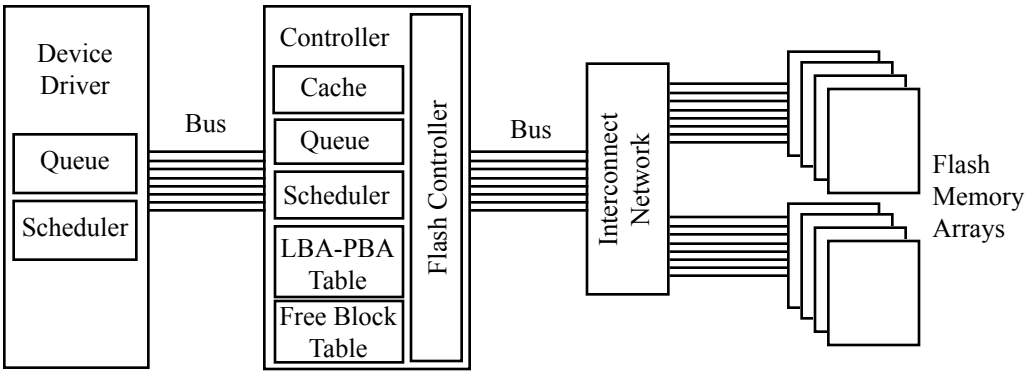


Figure 5.4: Flash SSD simulator components.

linked together in a Micron-style superblock. The size of flash memory chips can be changed while keeping the storage system capacity constant. Our flash simulator can simulate these various solid-state disk architectures while providing the illusion of a single HDD to host system.

5.3. Disk I/O Traces

In this dissertation, we have used our own disk traces to measure the performance of NAND flash solid-state disks. These traces are collected from portable computers and PCs running real user workloads. The workloads represent typical multi-tasking user activity, which includes browsing files and folders, emailing, text editing and compiling, surfing the web, listening to music and playing movies, editing pictures, and running office applications.

Our workloads consist of not only I/O traffic generated by user applications, but also read and write requests generated by system and admin processes. System processes are important as they generate I/O traffic comparable to the traffic generated explicitly by the user application. For example, half of the sectors read from the disk are requested by

the host system. File system cache update constitutes 40% of sectors written to disk. A snapshot of processes running during typical multi-tasking user activity is provided in Table 5.1.

We have collected our workloads by monitoring file system activity and filtering disk requests. We have used *fs_usage*, which is a general BSD command, to monitor file system. System calls reported by *fs_usage* include both activity by host system processes and user initiated I/O requests. Data reported by *fs_usage* includes a timestamp, file system call type, file descriptor, byte count requested by the call, disk block address, file offset, process name, and time spent in system call. A sample output of *fs_usage* command is shown in figure 5.5.

Not all file system activity reported by fs_usage generates a read or write request to the hard disk drive. Therefore, we have filtered out file system calls which generate an I/O request. Any system call which initiates an I/O request has to provide a block address (tagged as D=0x) and request size (tagged as B=0x). A sample output of our filtering process is shown in figure 5.6. After filtering out I/O requests, we have sorted them in time and generated an ascii trace file. Final format of our traces is in default ascii trace format, where each request is defined by 5 parameters: request arrival time, device number, block number, request size, and request flags [32]. A sample trace in ascii format is shown in figure 5.7.

The characteristics of our traces are in-line with expected I/O traffic for personal computer workloads reported by Hsu and Smith [38]. The average I/O per second in our traces range from 1.6 Mbps to 3.5 Mbps, similar to 2.37 Mbps reported in [38]. Our

User Applications	Adobe Reader - Pdf viewing
	Safari - Internet Browsing
	Finder - Folder browsing and editing
	iTunes - Listening to music
	Word & Excel - Office Application
	Preview - Picture viewing
	Quicktime Player - Watching movie
	Terminal
	srm - File deleting
User Support Applications	AppleSpell - Checking spelling
	iCal - Calendar and Alarm Scheduler
	ATSServer - System font manager
	mdimport - File indexing
	Dock - Application quick launch
	WindowServer - Manage application windows
Virtual Memory	kernel_task - Virtual memory manager
	dynamic_pager - Handling swap files
File System Cache	update - Flush file system cache to disk
Root Processes	configd - System and network configuration daemon
	automount - Auto mount and unmount network file systems
	diskarbitration - Mounting disk and file systems
	coreaudiod - Daemon used for core audio purposes
	syslogd - System log utility
	securityd - Access to keychain items
	mDNSResponder - Provides network services announcement

Table 5.1: Processes running during typical multi-tasking user activity.

Figure 5.5: *Fs_usage.* Sample output of file system activity reported by *fs_usage*

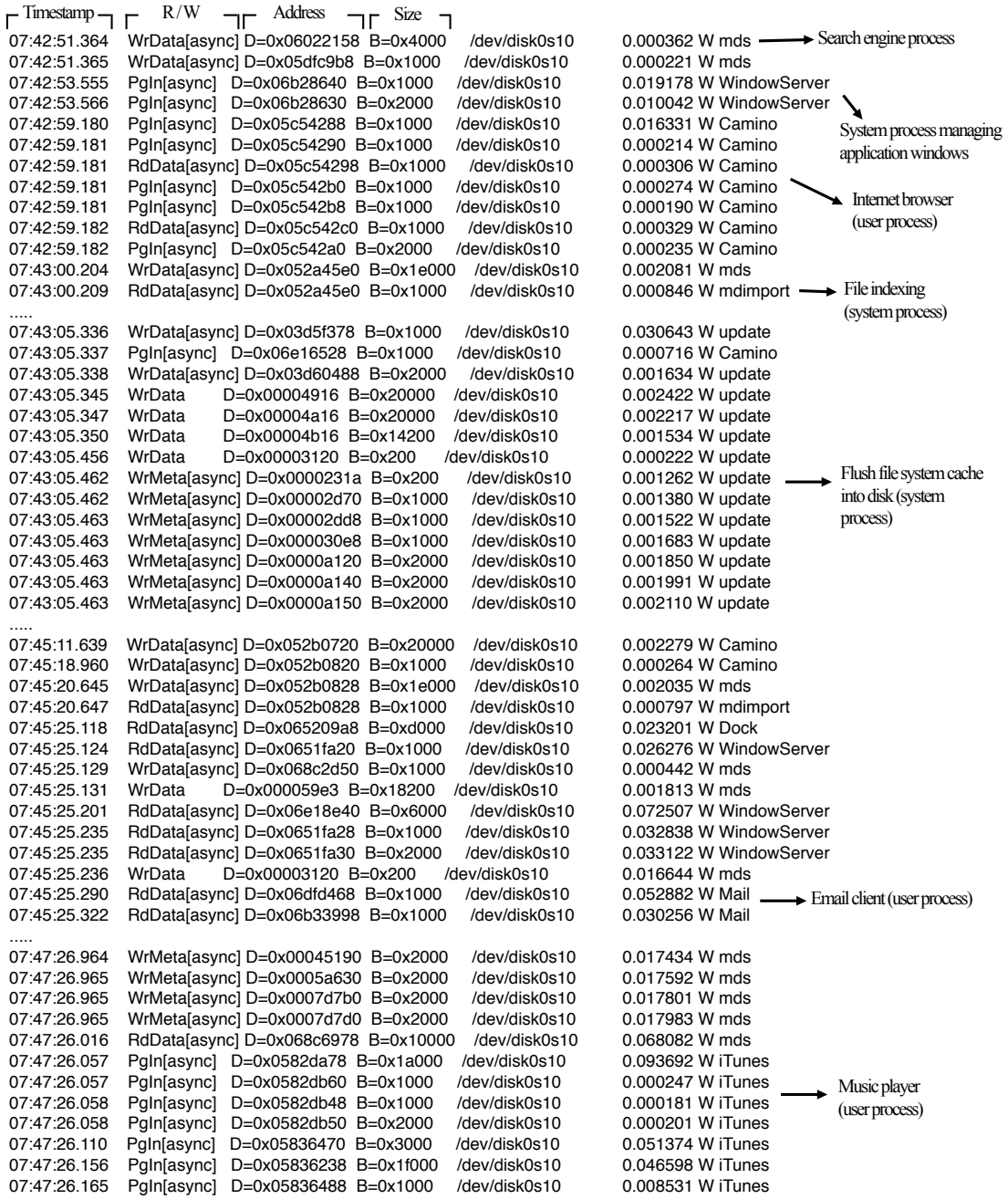


Figure 5.6: Disk I/O Requests. Sample output of file system activity. I/O requests are filtered.

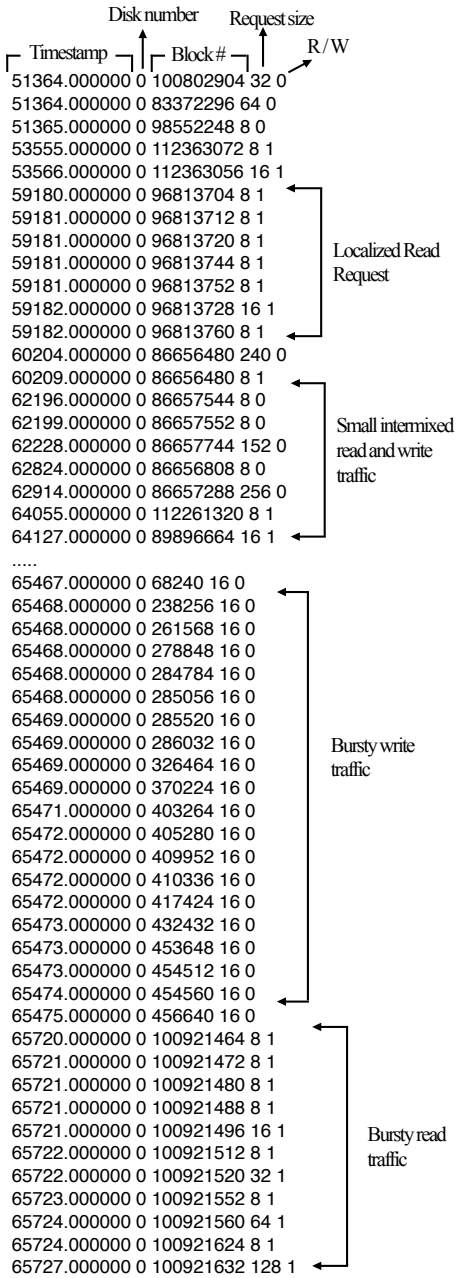


Figure 5.7: Input Trace. Sample ascii input trace showing characteristics of user activity.

personal computer workloads generate 4.6 to 21.35 I/O requests per second with an average request size of 26 KB. Although this average request size is much higher than the 7-9 KB expected by [38], it is weighted by a small number of large files; approximately half of the requests generated in our traces are 4-8 KB. We observed that the average request size in our personal workloads is skewed by the occasional very large write requests (of size 64 KB and higher). We have also confirmed that I/O traffic in our workloads is bursty, localized and balanced - I/O requests arrive in groups, frequently access localized areas of the disk, and are partitioned roughly 50:50 between reads and writes. Figure 5.8-14 summarize the properties of each trace and shows three different 4-minute snapshots, representing different mixes of reads and writes. Request size distribution and read:write ratio for each trace is also shown.

5.4. Simulation Parameters

In this dissertation, we have modeled a 32 GB ATA 133 MB/s NAND flash SSD. We modeled today's typical SSD which usually support multiple channels and multiple flash memory banks. We have simulated various configurations of flash memory banks on a shared bus or multiple independent channels to different flash banks or a combination of the two. Some of the configurations we have simulated are shown in figure 5.15. In each configuration modeled, the size of flash memory banks can change while the entire storage capacity is kept constant at 32 GB. For example, if 4 memory banks are connected via a single shared bus, figure 5.15b, then each bank is 8 GB in size. If a configuration with 2 independent I/O channels and 4 banks per channel is used, figure 5.15c, then each memory bank is 4 GB (system capacity is 8 x 4 GB).

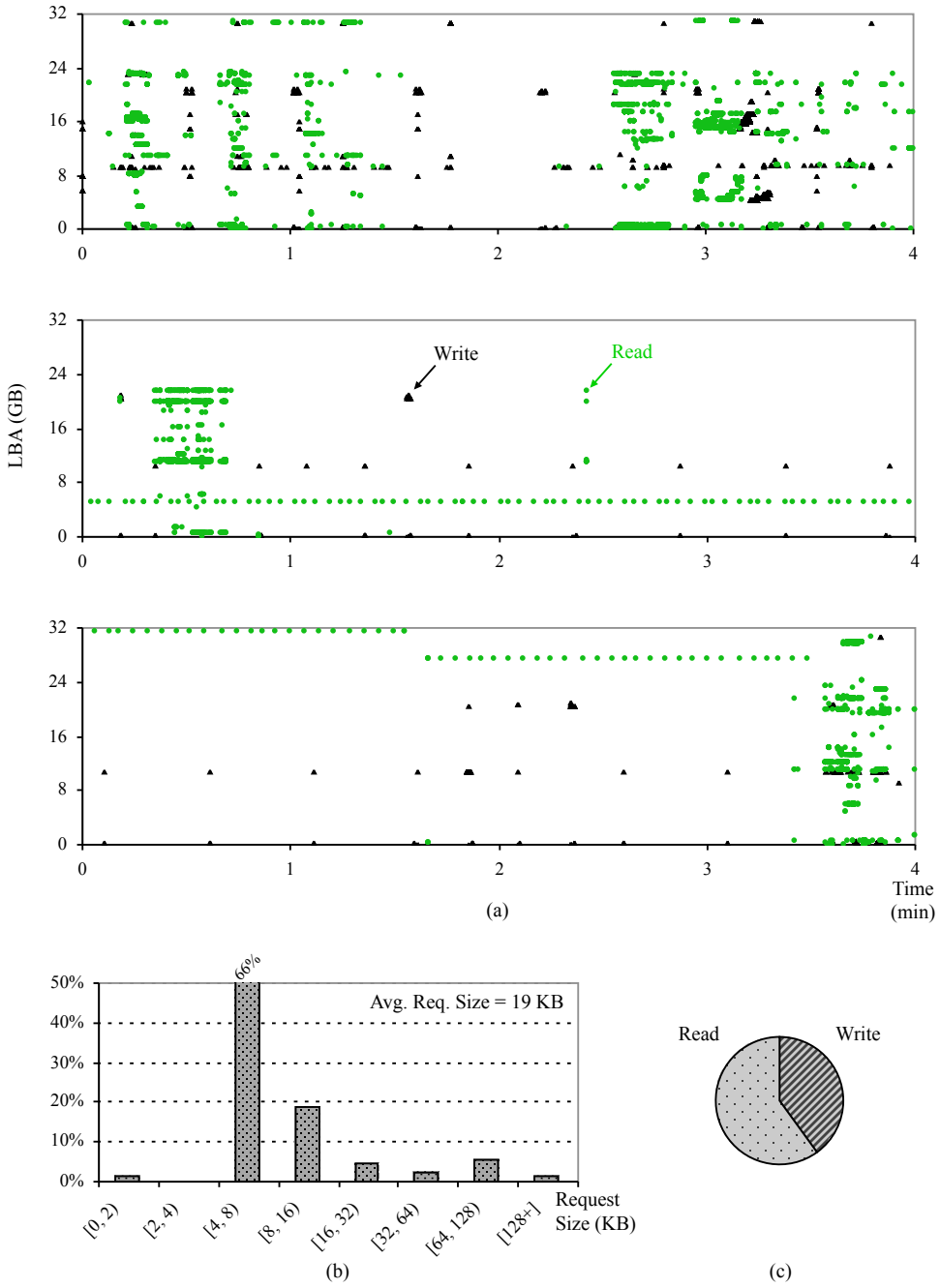


Figure 5.8: Trace 1 Characteristics. (a) 3 4-minute snapshots from trace 1 confirm bursty, localized traffic. (b) Request size distribution show that median and mean request size are different. (c) Read:write ratio for trace 1 is 60:40

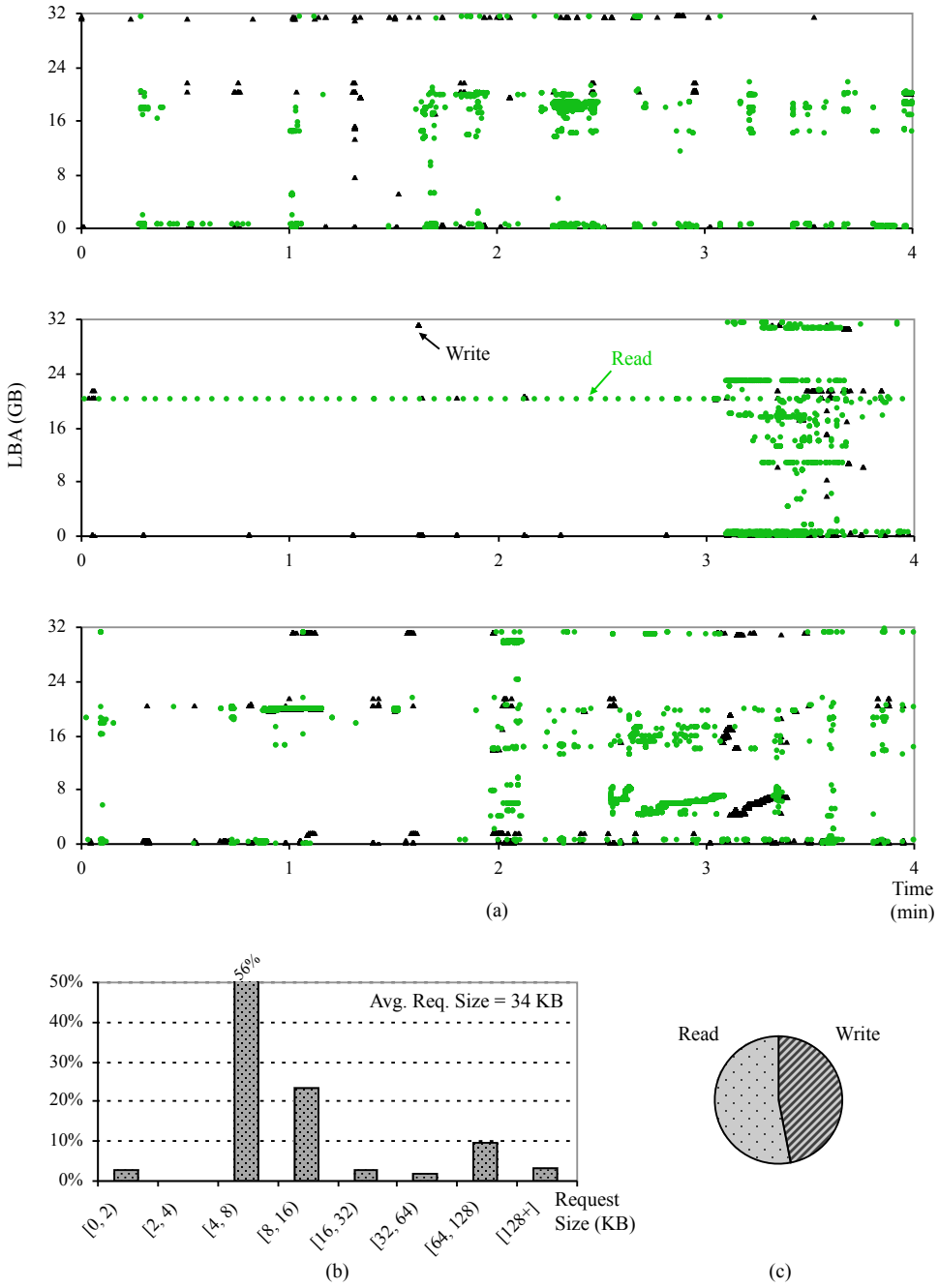


Figure 5.9: Trace 2 Characteristics. (a) 3 4-minute snapshots from trace 1 confirm bursty, localized traffic. (b) Request size distribution show that median and mean request size are different. (c) Read:write ratio for trace 1 is 53:47

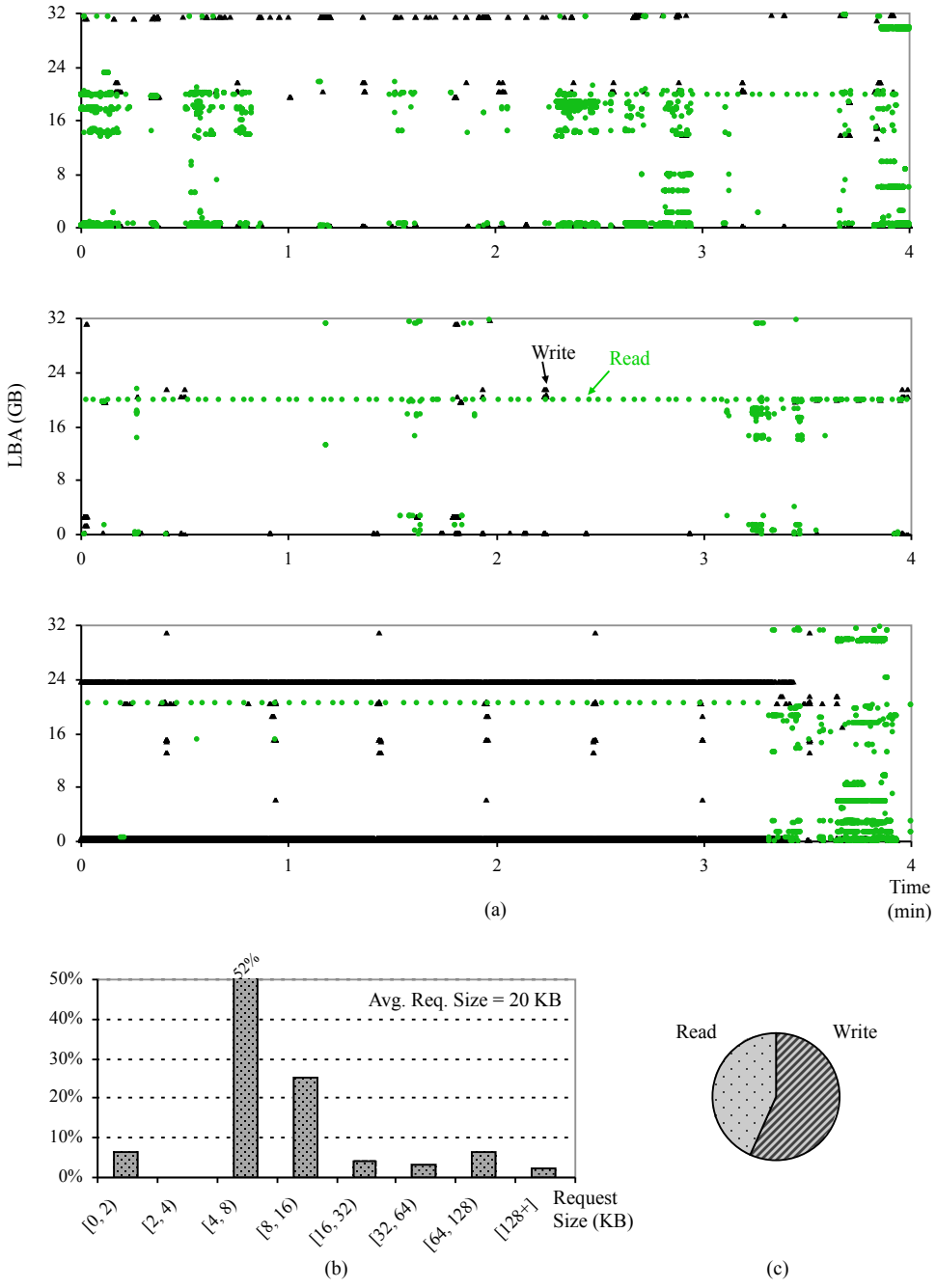


Figure 5.10: Trace 3 Characteristics. (a) 3 4-minute snapshots from trace 1 confirm bursty, localized traffic. (b) Request size distribution show that median and mean request size are different. (c) Read:write ratio for trace 1 is 43:57

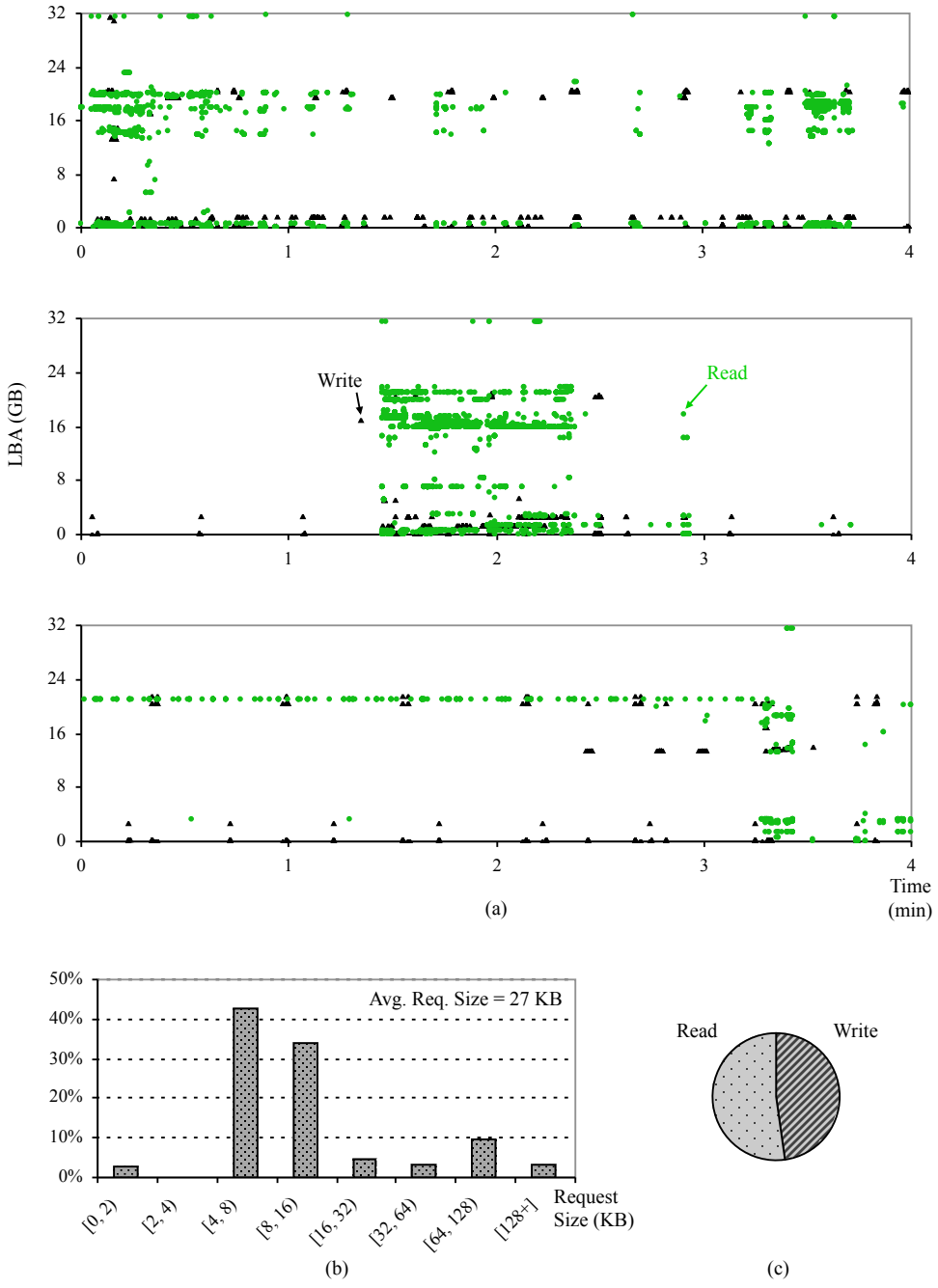


Figure 5.11: Trace 4 Characteristics. (a) 3 4-minute snapshots from trace 1 confirm bursty, localized traffic. (b) Request size distribution show that median and mean request size are different. (c) Read:write ratio for trace 1 is 52:48

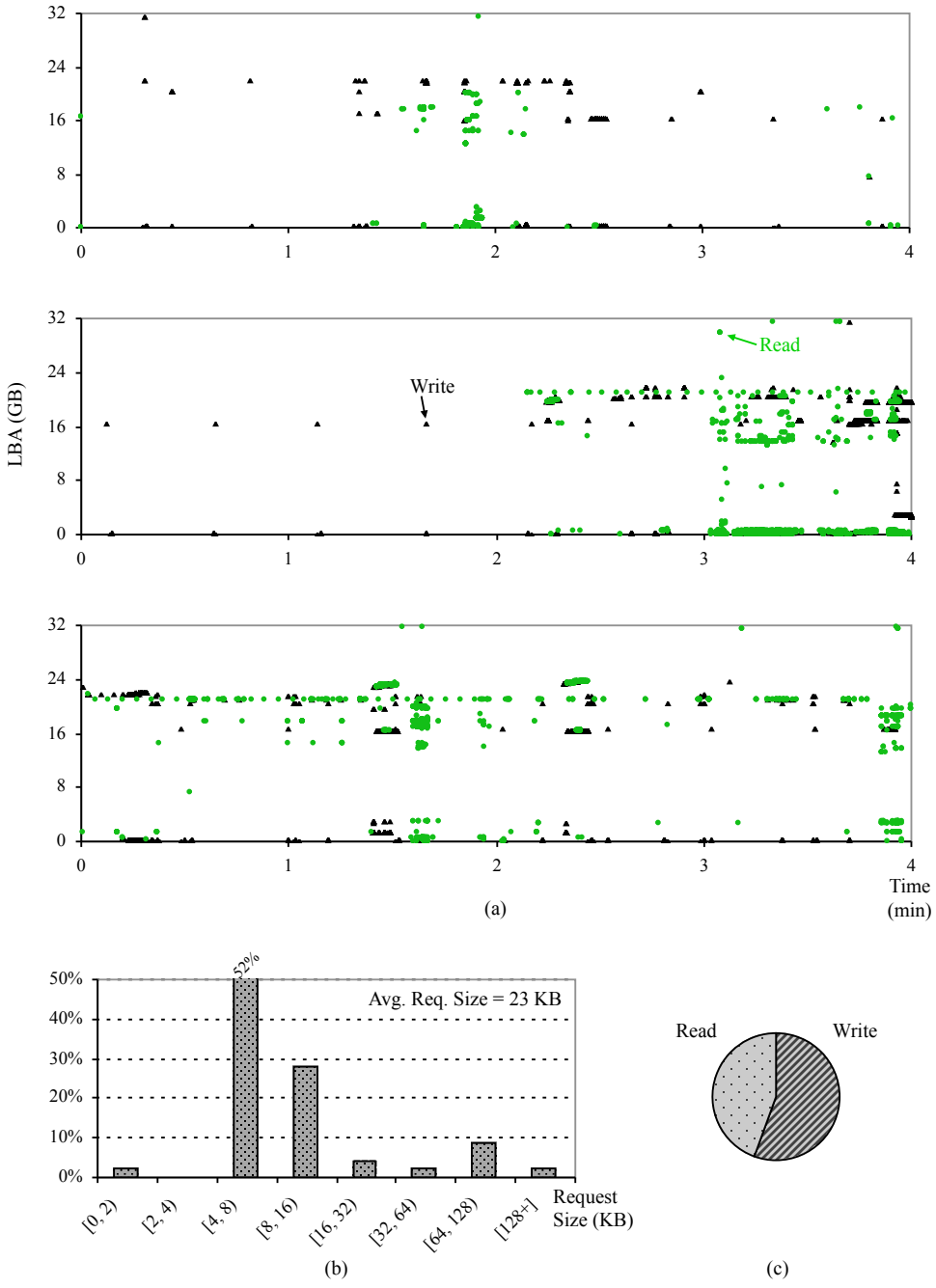


Figure 5.12: Trace 5 Characteristics. (a) 3 4-minute snapshots from trace 1 confirm bursty, localized traffic. (b) Request size distribution show that median and mean request size are different. (c) Read:write ratio for trace 1 is 45:55

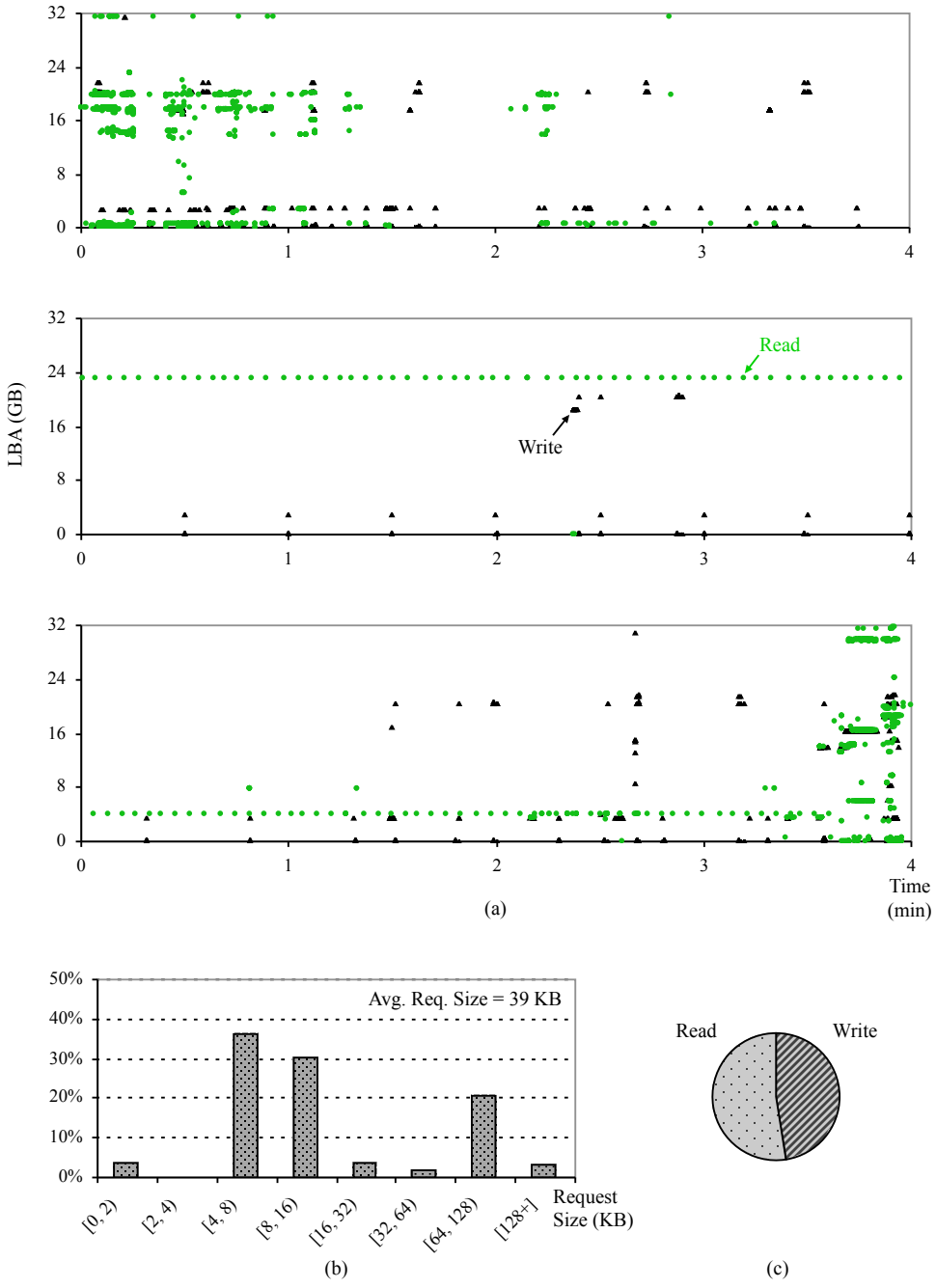


Figure 5.13: Trace 6 Characteristics. (a) 3 4-minute snapshots from trace 1 confirm bursty, localized traffic. (b) Request size distribution show that median and mean request size are different. (c) Read:write ratio for trace 1 is 53:47

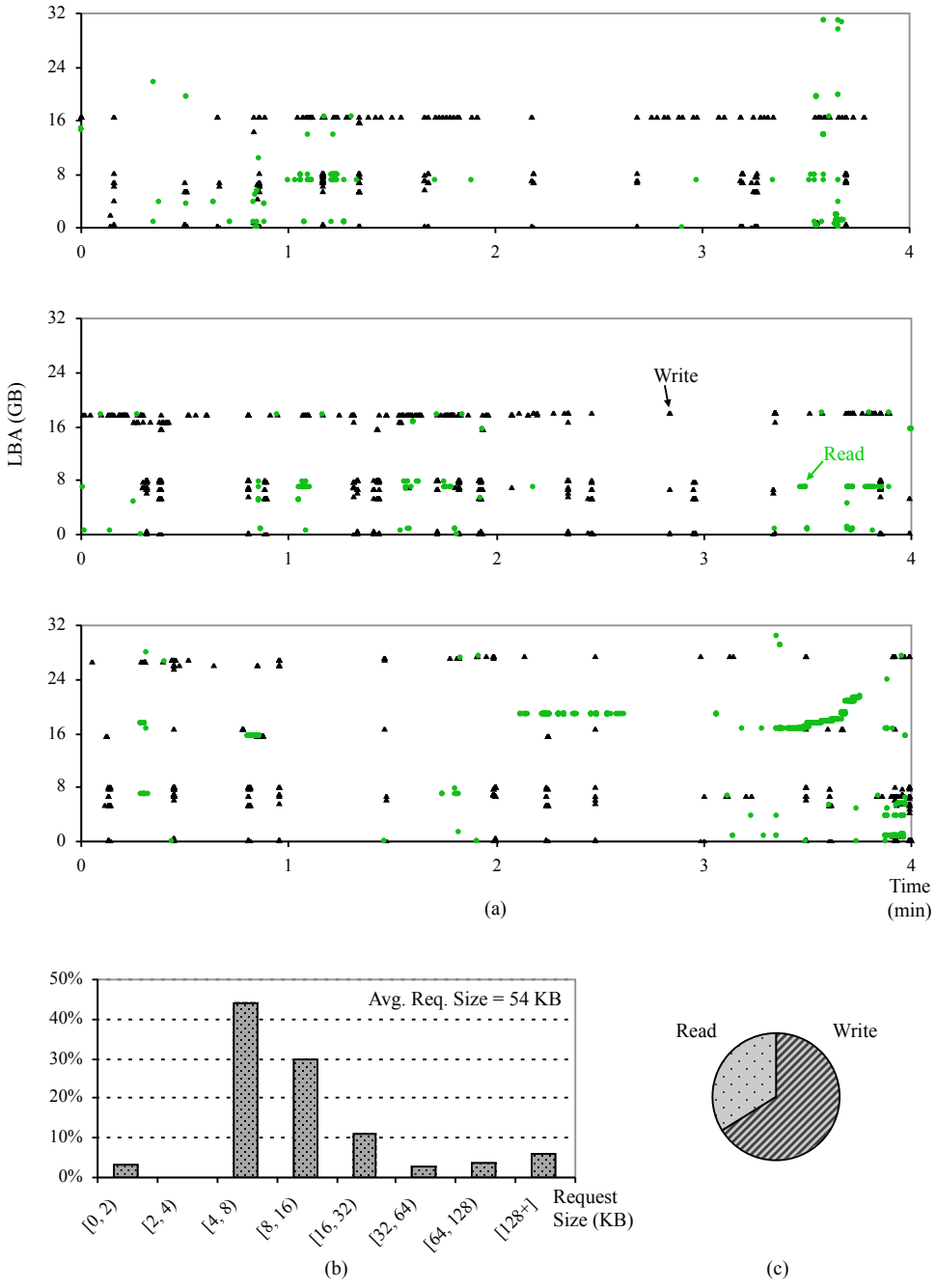


Figure 5.14: Trace 7 Characteristics. (a) 3 4-minute snapshots from trace 1 confirm bursty, localized traffic. (b) Request size distribution show that median and mean request size are different. (c) Read:write ratio for trace 1 is 34:66

In our models, we have used the following timing parameters for our flash memory model: page access time of 25 μs, page program time of 200 μs and block erase time of 1.5 ms. Our flash memory model is a large block flash memory with page sizes of 2 KBytes and block sizes of 64 pages. Logical to physical address mapping is performed at the granularity of a page. The speed at which data can be read from the flash memory banks to the external flash controller also varies throughout our simulations. We have modeled 8-, 16- and 32-bit wide I/O busses with speeds of 25, 50 and 100 MHz.

To simulate a realistic flash management model, we have assumed modular striping for write requests. If we have a total of x memory banks, the Nth write request is assigned to bank number N(mod x). We have maintained a pool of free blocks for each bank and have allocated pages from current working free blocks when a write request is received. Modular striping also provides simple yet effective wear leveling, as each memory bank handles an equal amount of write traffic.

Disk Parameters	
Disk capacity	32 GB
Host I/F	ATA 133 MB/s
Configuration	
I/O channels	1, 2, 4
Memory banks per channel	1, 2, 4
Page size	2 KB
Block size	128 KB
Memory bank density	2, 4, 8, 16, 32 GB
LBN-PBN mapping	Page mapping
Timing parameters	
Page read	25 µs
Page write	200 µs
Block erase	1.5 ms
I/O transfer rate	25, 50, 100 MHz

Table 5.2: Simulation parameters. Base configuration for flash memory storage system.

Chapter 6: Experimental Results

In this dissertation, we explore the system-level organization choices for NAND flash memory solid-state disks - we study a full design space of system-level organizations, varying number of busses, speeds and widths of busses, and the degree of concurrent access allowed on each bus. To compare with system-level details, we also investigate device-level design trade-offs, including pin bandwidth and I/O width. We present scheduling heuristics and I/O access policies of NAND flash SSD storage systems, which exploit the distinctive differences between reading from and writing to flash memory.

6.1. Banking and Request Interleaving

One way to increase the performance of flash memory disks has been utilizing request parallelism among memory banks in the flash array where each bank can read/write/erase independently. As we have mentioned before, writing (programming) a page into flash memory is significantly longer than reading a page from flash. In a typical user workload, read and write requests come in batches - sequential reads and writes. One way to hide write latency in flash memory would be interleaving sequential writes by assigning write requests to individual flash memory banks. Since flash memory allocates a newly erased page for a write request, choosing an empty page for each write becomes a run-time decision of resource allocation. When sequential write requests arrive, one can assign free pages for these writes from different banks. This way sequential writes are dispatched to multiple independent banks in parallel, and page write times are interleaved. Figure 6.1 shows a flash array organization with 4-way banking and a timing diagram for 4 sequential write requests of 2 KB each.

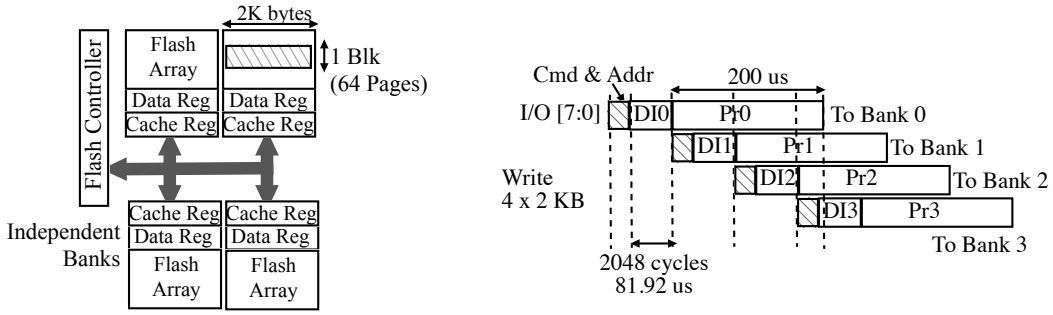


Figure 6.1: Request interleaving. 4-way banking and timing diagram of 4 subsequent write requests of 2 KB each. 8 bit I/O bus is shared by independent banks.

On the other hand, sequential read requests do not benefit from multiple banks as much as write requests. Only the bank which holds the requested physical page can serve a read. However once write requests are serviced faster, read requests will wait less time in queue and read performance will also increase.

In order to understand the performance impact of banking and request interleaving, we have simulated sample configurations where 1, 2, 4 or 8 flash memory banks are attached to a single I/O bus and configured to read/write/erase independently. Modular striping is implemented as write policy - if we have a total of *x* memory banks, the Nth write request is assigned to bank number N(mod *x*). Average disk-request response time is reported, as a sum of physical access time (time to read/write data from/ to flash array) and queue wait time. Figures 6.2-4 show the effect of increasing the degree of banking on average disk-request response time for various user workloads.

As summarized in table 6.1, one sees significant improvements in both read and write request times (75-90%) when the level of banking is increased from 1 to 2. Request times can be further improved by 30-50% by increasing the level of interleaving from 2

All requests	1-way	2-way	4-way	8-way
25 MHz I/O	30.56	6.85	4.92	4.68
Reads				
25 MHz I/O	15.55	3.01	2.41	2.38
Writes				
25 MHz I/O	45.13	10.53	7.31	6.91

Table 6.1: 1-, 2-, 4-, and 8-way banking. Average request service times in milliseconds using a shared 8-bit 25 MHz I/O bus.

to 4. However, from 4- to 8-way banking, reads and writes start to show different performance characteristics. While request times continue to improve for writes; read-request performance begins to flatten, moving from 4- to 8-way banking. This is explained by an increase in the physical access times at high levels of banking due to bus contention - especially for low bandwidth 8-bit 25 MHz bus configuration. The more banks per channel, the larger degree of bus utilization, to the point of traffic congestion. As shown in figure 3.7 in chapter 3, read request timing mostly consists of time spent in reading data from flash memory via I/O bus. It is more sensitive to degradation in the I/O channel than writes because any congestion in the I/O bus will impact reads more than writes. For a typical 4K read request, 90% of the physical access is reading data from flash memory through I/O interface. On the other hand, for a typical 4K write request only 40% of the physical access is transferring data into flash memory and requires I/O bus. Delays in acquiring the I/O bus in a shared bus configuration will have a large impact on the read request timing. In our simulations with 8-way banking and low I/O

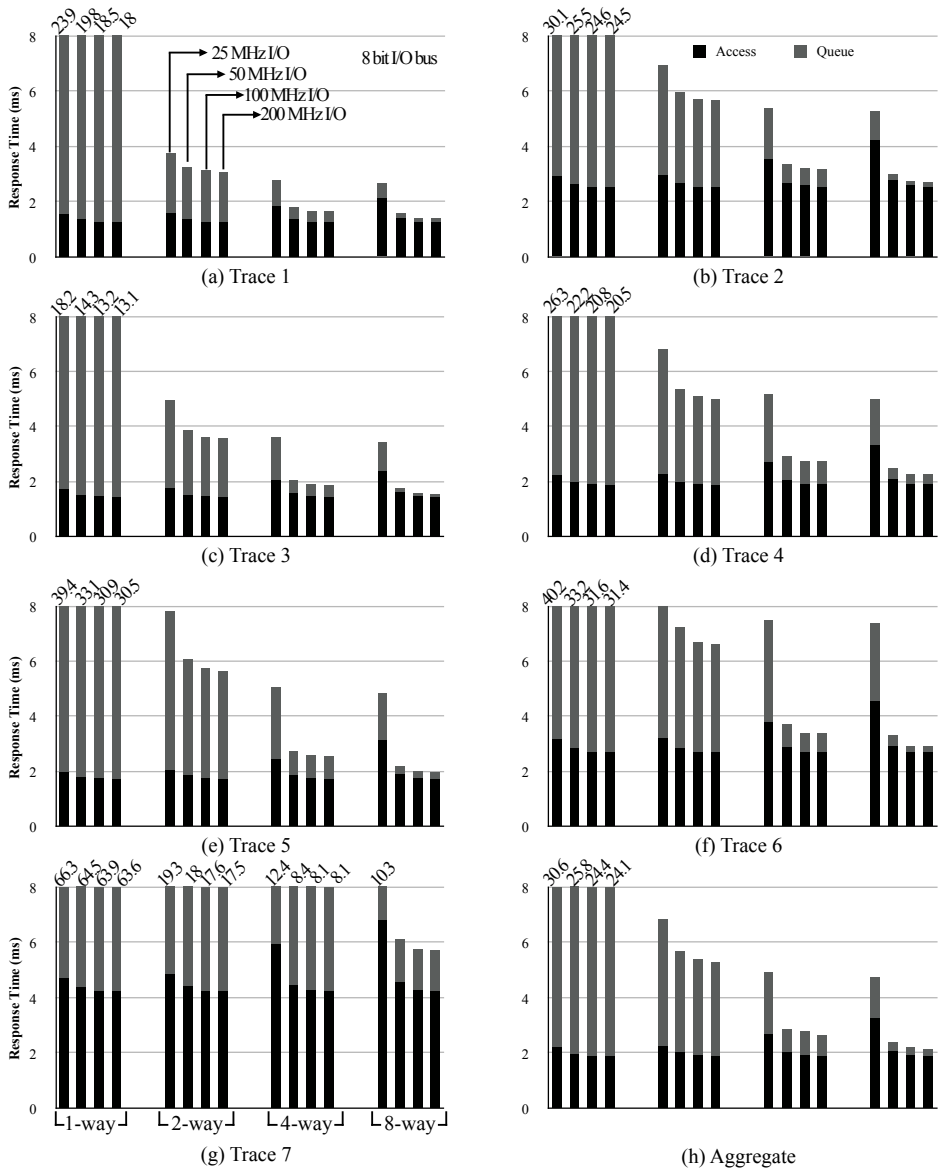


Figure 6.2: Single Shared I/O bus, 1-, 2-, 4-, and 8-way banking. Average request service time in milliseconds using a shared 8-bit I/O bus for various user workloads is shown. Request times improve significantly moving from 1-way to 2-way banking or from 2-way to 4-way banking.

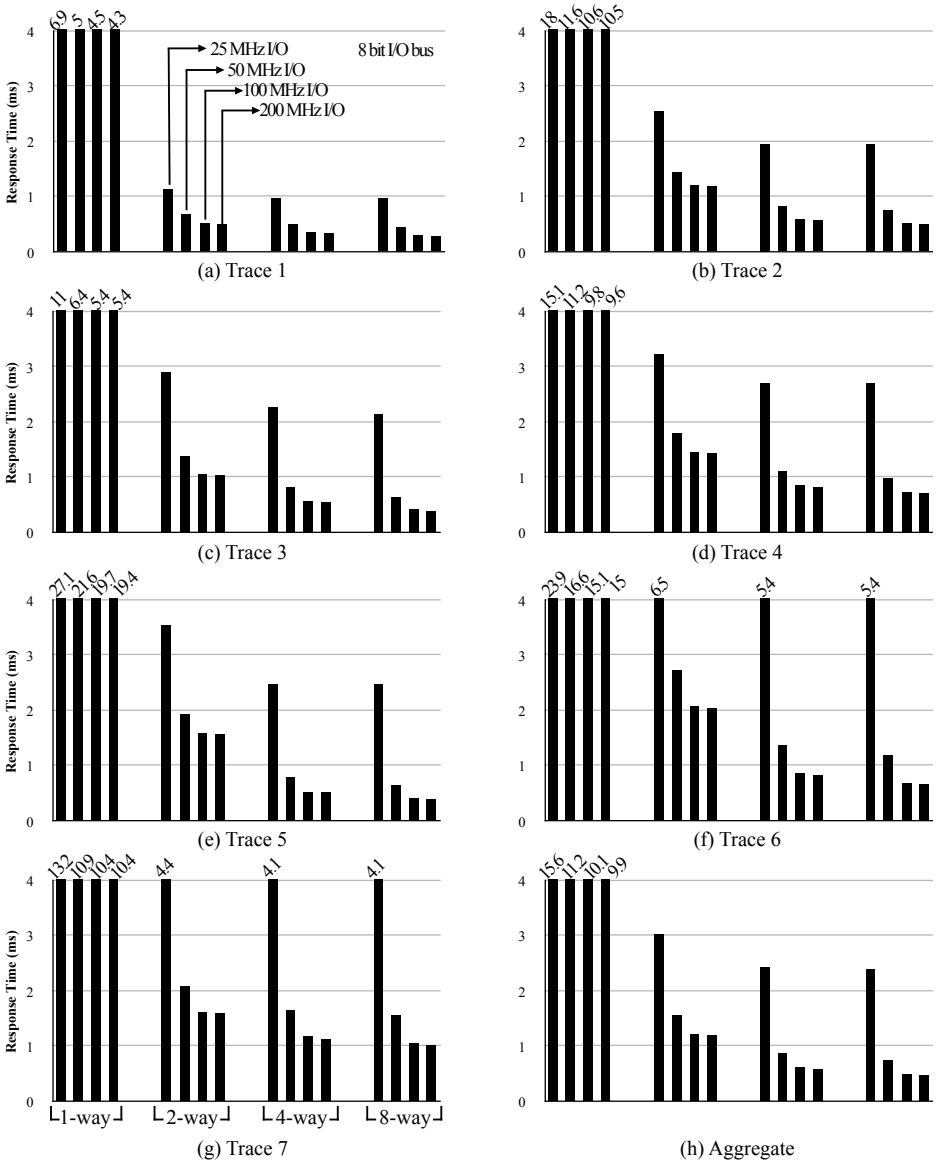


Figure 6.3: Reads with single shared I/O bus, 1-, 2-, 4-, and 8-way banking. Average read request service time in milliseconds using a shared 8-bit I/O bus for various user workloads is shown. Request times improve significantly moving from 1-way to 2-way banking or from 2-way to 4-way banking. Performance does not improve much with 8-way banking due to the increase in the physical access times.

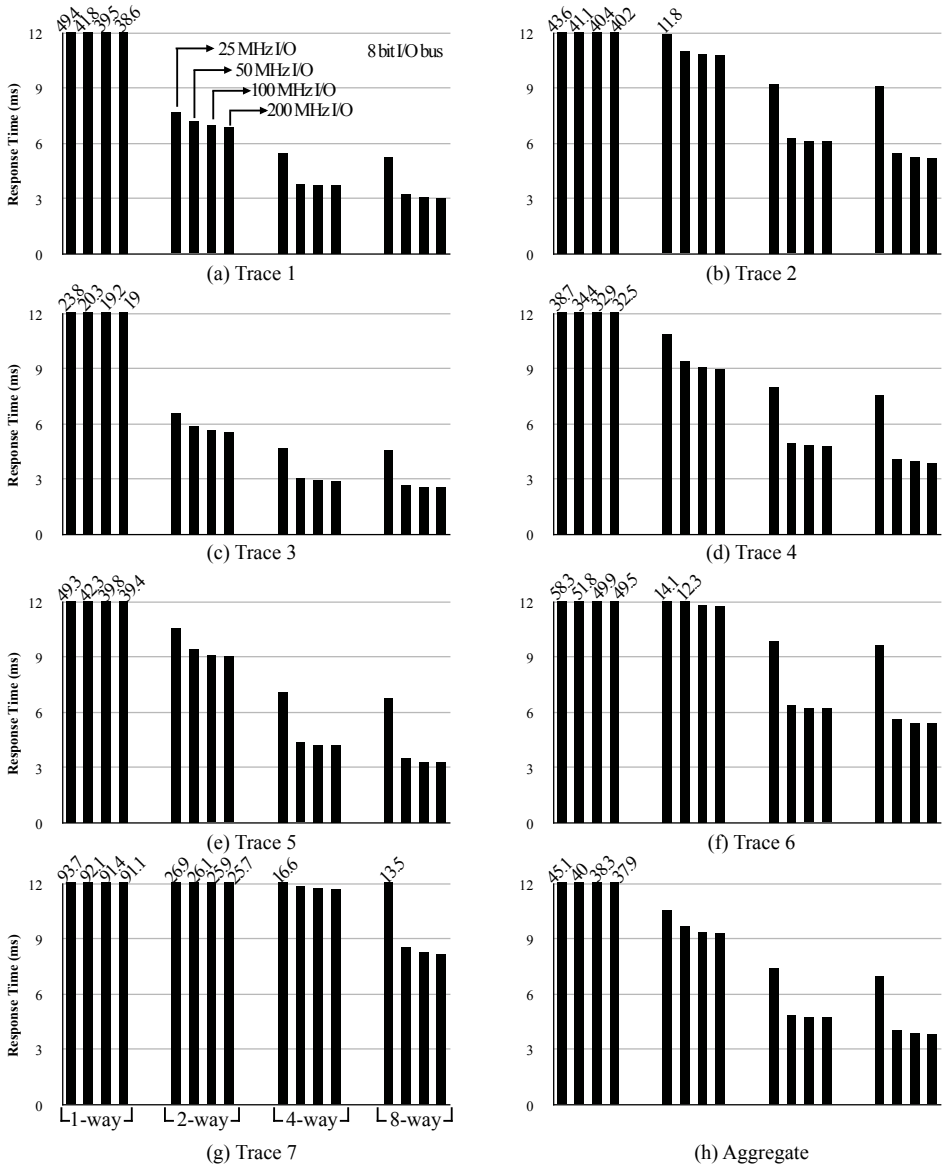


Figure 6.4: Writes with single shared I/O bus, 1-, 2-, 4-, and 8-way banking. Average write request service time in milliseconds using a shared 8-bit I/O bus for various user workloads is shown. Request times improve significantly moving from 1-way to 2-way banking or from 2-way to 4-way banking. Performance continues to improve when 8-way banking is used. Moreover, write request times show very high variation depending on the workload even with a high degree of banking.

bus bandwidth, we have observed 1.3 milliseconds spent in average on bus arbitration for each request. This corresponds to more than 50% of average read request service time. A good example is comparing user trace 1 and user trace 7, as trace 1 is read oriented and trace 7 is write heavy. With trace 1, moving from 4-way to 8-way banking performance improved 4-18% depending on the speed of I/O bus. On the other hand, trace 7 performance improvement was 15-30% from 4-way to 8-way banking. Read performance only improved 2-15% for both traces.

Performance of read requests is critical because the overall system performance tracks the disk's average read response time [40]. Therefore, one does not gain much by increasing the level of interleaving from 4 to 8 in a single channel configuration. Attaching more banks to a single channel increases the electrical load and may limit the maximum clock rate of the I/O bus. More load draws more power. When combined with increased access time due to bus contention, more power is drawn for a longer time. These factors increase the cost of 8-way banking and a 5-20% performance improvement may not be enough to justify the cost increase.

Another good example would be comparing 2-way and 4-way banking and analyzing performance improvements with cost and power considerations. If only write requests are taken into account, moving from 2-way banking with high speed 100-200 MHz I/O bus to 4-way banking with a slower 25 MHz I/O bus will always see performance improvements. Increasing the load on the I/O channel, but slowing the clock rate may result in similar cost and power consumption. However, if we only focus on

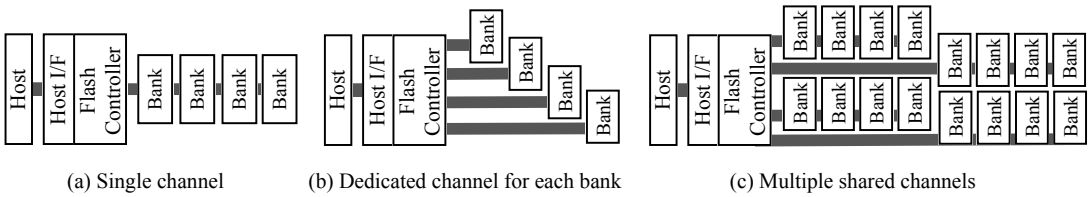


Figure 6.5: Flash SSD Organizations. (a) Single I/O bus is shared - 1, 2, or 4 banks; (b) dedicated I/O bus; 1, 2, or 4 buses and single bank per bus; (c) multiple shared I/O channels - 2 or 4 channels with 2 or 4 banks per channel.

reads, the same will result in 100% or more performance degradation. Then, one needs to consider workload characteristics - is expected usage read oriented or write heavy?

Additional observations regarding read and write performance are: Write request timing show very high variation depending on the workload even with a high degree of banking. Asymmetry in read and write timing results in 2-3x scale difference between their performances. Load dependent variation in request times and significant difference time difference between reads and writes indicate that there is room for further improvements by employing various request scheduling techniques and run-time resource allocation policies.

If better performance is aimed by using higher degree of interleaving (higher than 8-way), more I/O channels may be used to access memory array to support increased concurrency. We have simulated configurations with multiple independent flash memory banks per channel in combination with multiple independent channels. Various flash memory organization simulated are shown in figure 6.5.

As summarized in table 6.2 and shown in figures 6.6-11, using multiple I/O channels to connect the same number of memory banks only improves performance when I/O bandwidth is low. With a moderate speed I/O channel (50 MHz), increasing the

I/O Rate	Number of I/O channels x Number of banks per channel									
All requests	**1 x 1**	**1 x 2**	**2 x 1**	**1 x 4**	**2 x 2**	**4 x 1**	**1 x 8**	**2 x 4**	**4 x 2**	**4 x 4**
25 MHz	30.56	6.85	6.41	4.92	3.44	3.35	4.68	3.11	2.8	2.83
50 MHz	25.8	5.64	5.57	2.86	2.81	2.8	2.38	2.29	2.27	2.13
Reads										
25 MHz	15.55	3.01	2.59	2.41	1.85	1.75	2.38	1.75	1.63	1.59
50 MHz	11.19	1.52	1.47	0.84	0.8	0.79	0.73	0.69	0.67	0.62
Writes										
25 MHz	45.13	10.53	10.08	7.31	4.97	4.89	6.91	4.41	3.93	4.01
50 MHz	39.98	9.58	9.51	4.79	4.74	4.72	3.97	3.82	3.8	3.58

Table 6.2: Request time of Flash SSD Organizations. Average request service time in milliseconds using a shared 8-bit I/O bus.

number of I/O channels while keeping the same level of concurrency only results in a maximum of 5% performance improvement. This very small increase in performance will be costly. Adding more channels will increase the cost of a system as well as its power consumption. For typical user workloads, NAND flash memory interface is the limiting factor rather than the total system bandwidth. When I/O channels are added, they are idle most of the time. On the other hand, system behavior might be different when server workloads are used - where one would expect sustained read or write traffic over longer periods of time.

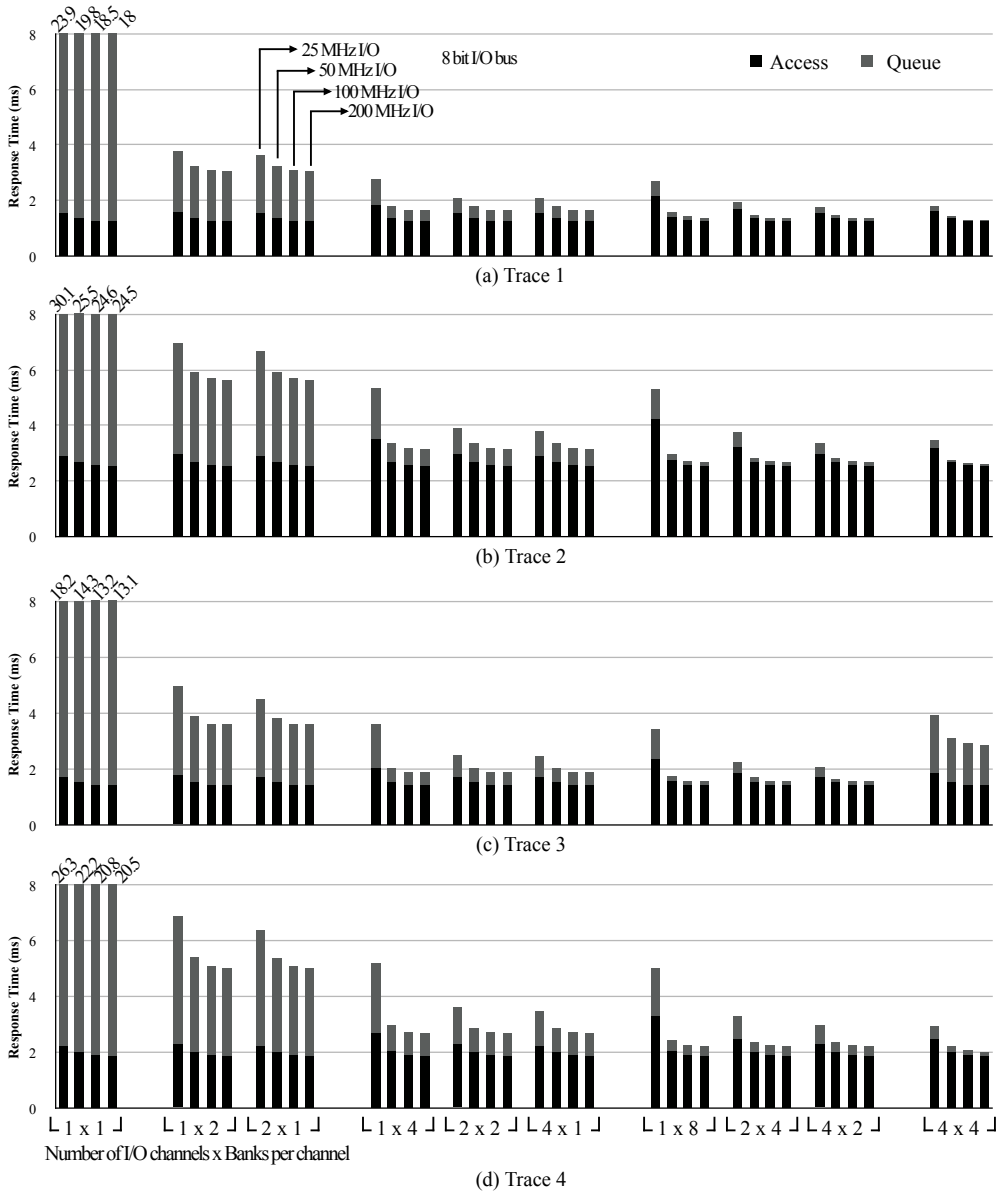


Figure 6.6: 1-, 2-, 4-, and 8-way banking with multiple bus configurations. Average request service time in milliseconds using a shared 8-bit I/O bus for user workloads 1-4 is shown. Using multiple I/O channels to connect the same number of memory banks only improve performance when I/O bandwidth is low - 8 bit 25 MHz I/O channel.

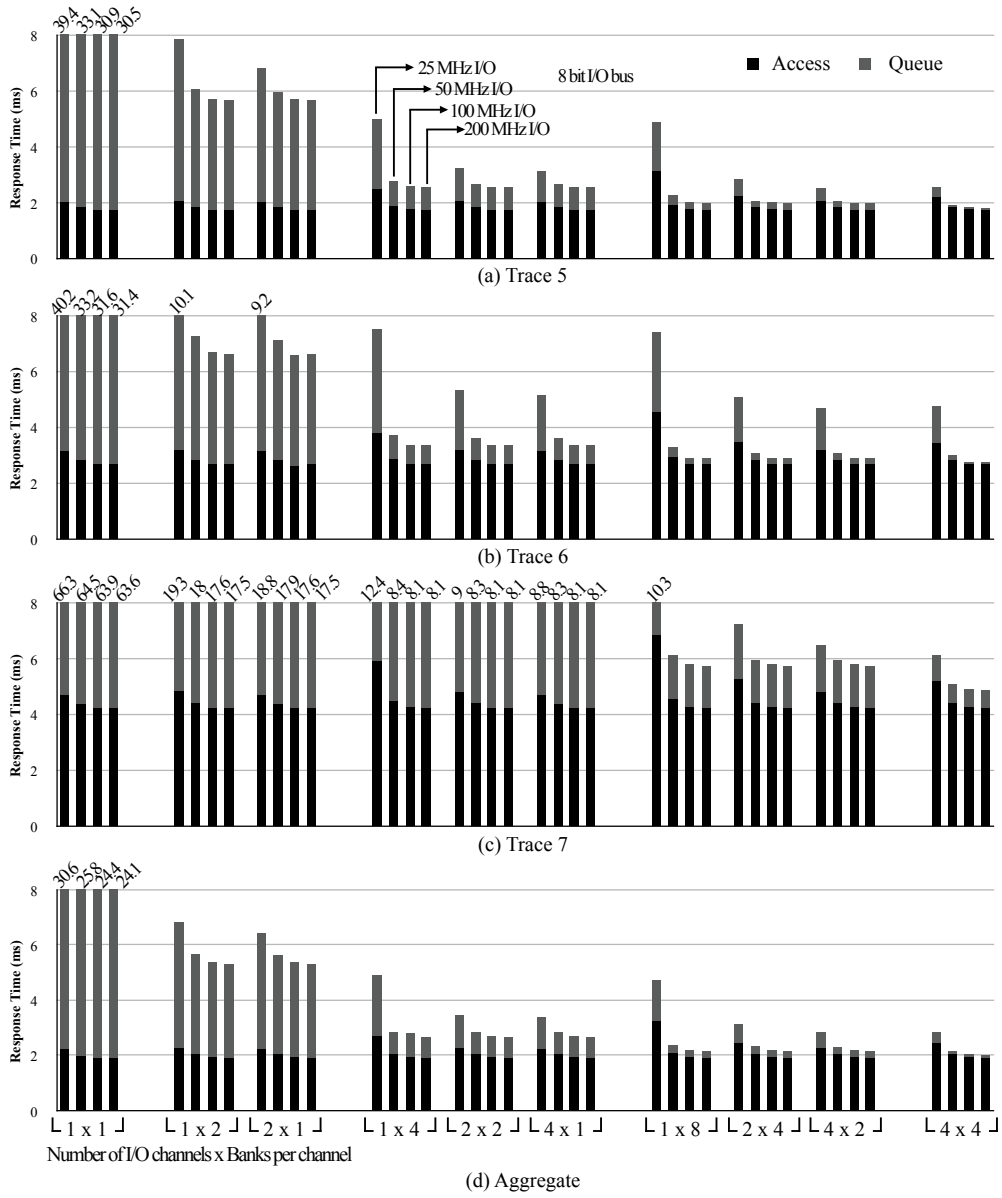


Figure 6.7: 1-, 2-, 4-, and 8-way banking with multiple bus configurations. Average request service time in milliseconds using a shared 8-bit I/O bus for user workloads 5-7 is shown. Using multiple I/O channels to connect the same number of memory banks only improve performance when I/O bandwidth is low - 8 bit 25 MHz I/O channel.

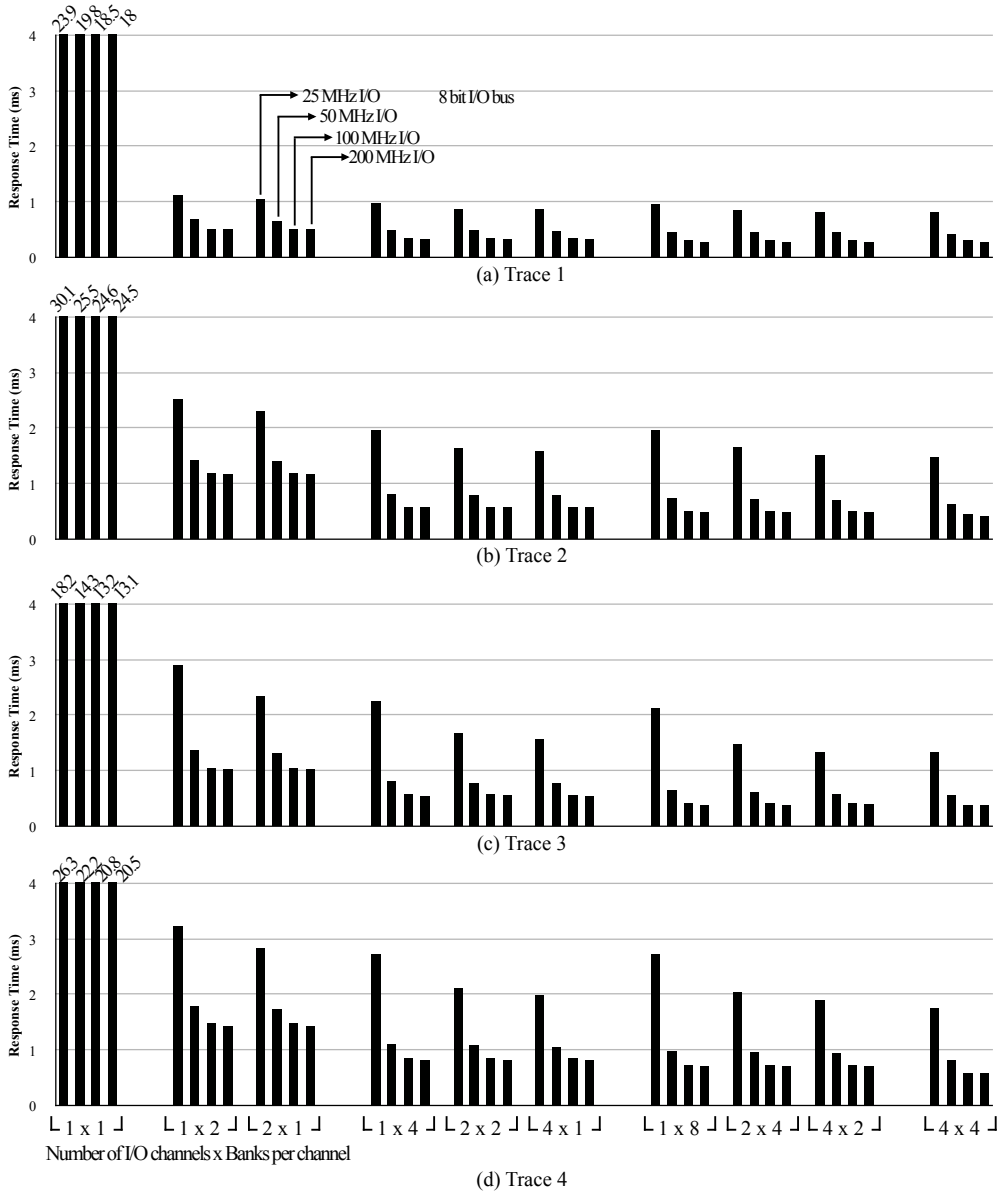


Figure 6.8: Reads with 1-, 2-, 4-, and 8-way banking and multiple bus configurations. Average read request service time in milliseconds using a shared 8-bit I/O bus for user workloads 1-4 is shown. Using multiple I/O channels to connect the same number of memory banks only improve read performance when I/O bandwidth is low - 8 bit 25 MHz I/O channel. Reads benefit when a faster I/O bus is used.

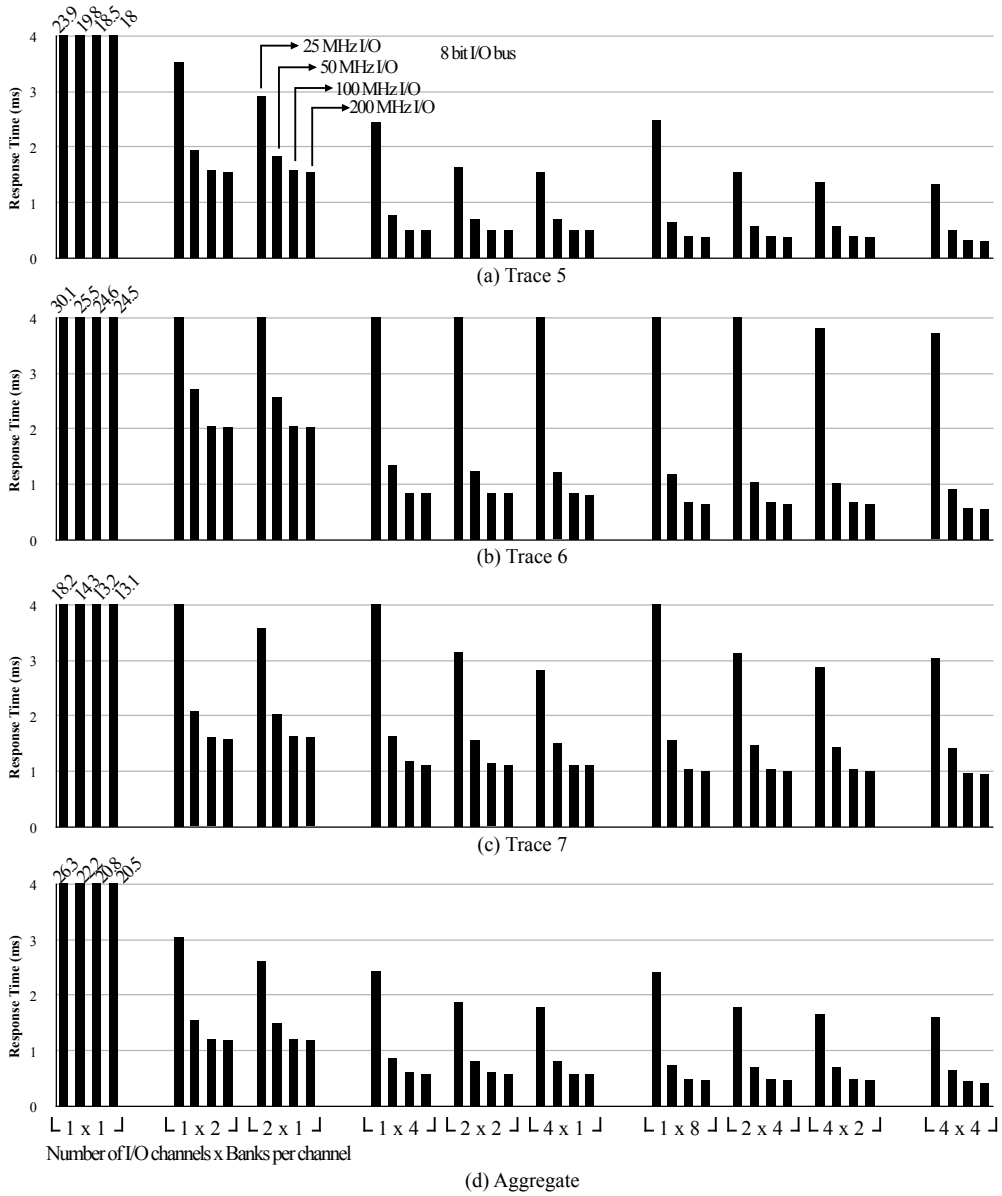


Figure 6.9: Reads with 1-, 2-, 4-, and 8-way banking and multiple bus configurations. Average read request service time in milliseconds using a shared 8-bit I/O bus for user workloads 5-7 is shown. Using multiple I/O channels to connect the same number of memory banks only improve read performance when I/O bandwidth is low - 8 bit 25 MHz I/O channel. Reads benefit when a faster I/O bus is used.

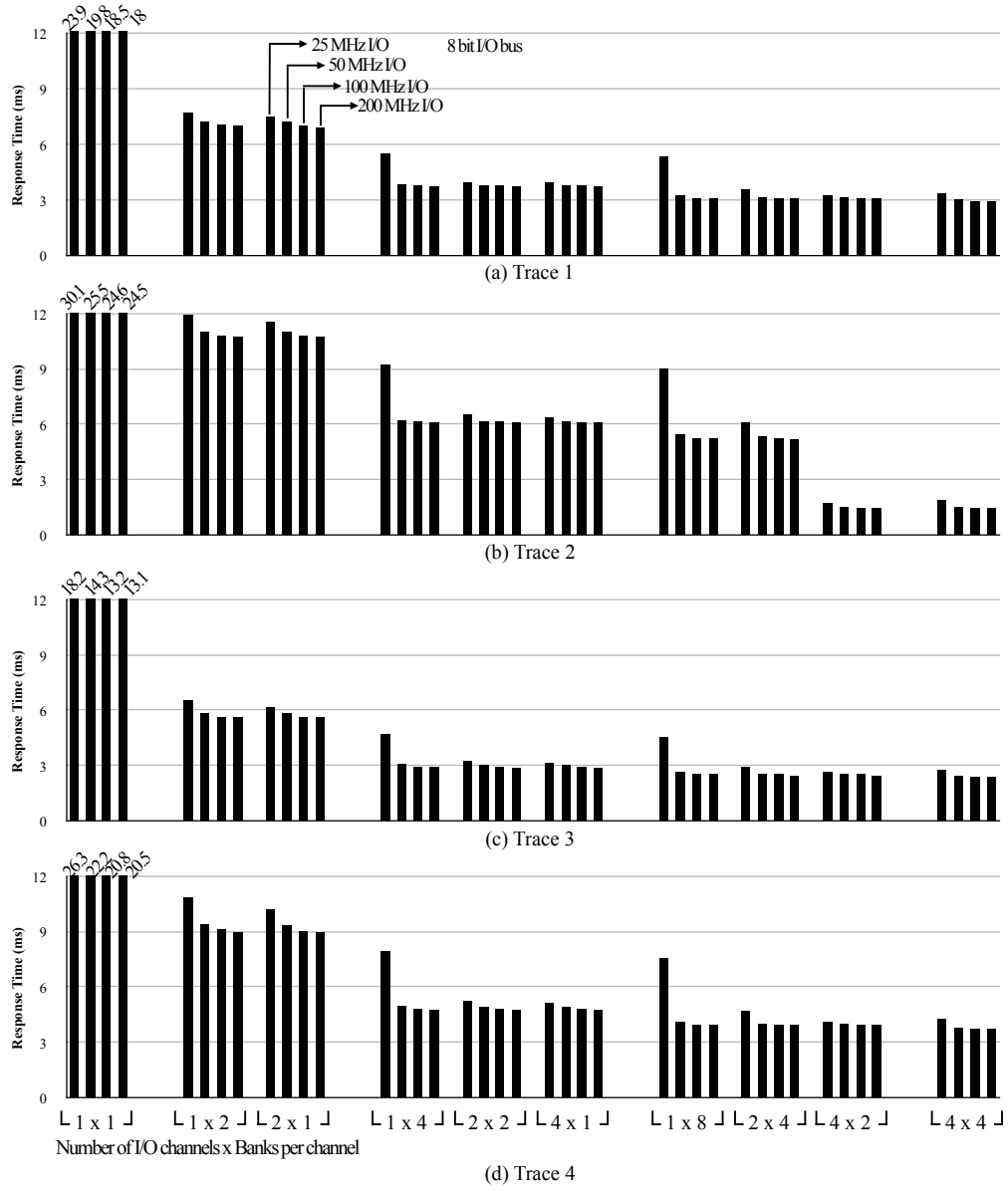


Figure 6.10: Writes with 1-, 2-, 4-, and 8-way banking and multiple bus configurations. Average write request service time in milliseconds using a shared 8-bit I/O bus for user workloads 1-4 is shown. Write performance is more dependent on the total number of memory banks and does not change much with the number of I/O channels used to connect to memory banks.

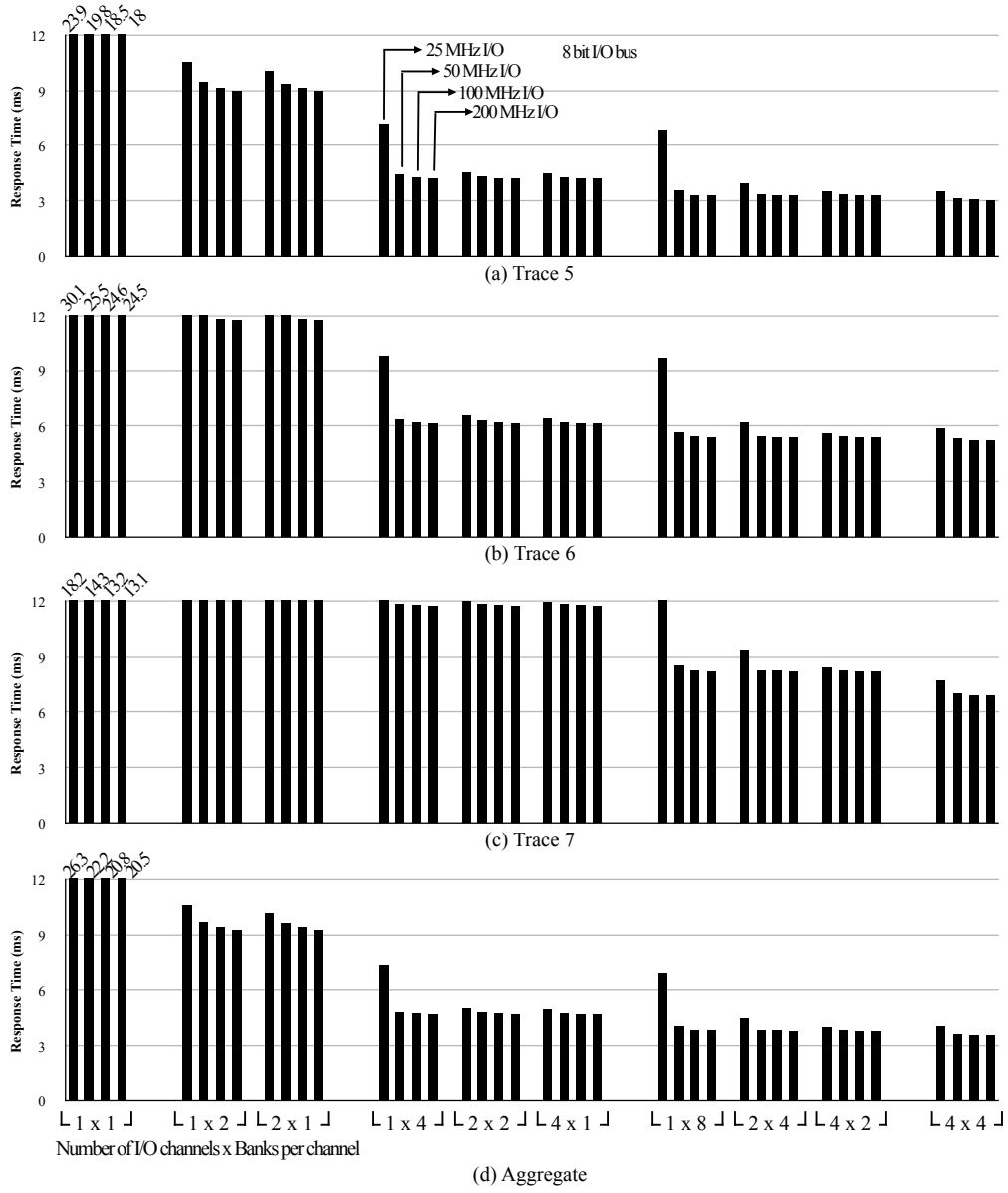


Figure 6.11: Writes with 1-, 2-, 4-, and 8-way banking and multiple bus configurations. Average write request service time in milliseconds using a shared 8-bit I/O bus for user workloads 5-7 is shown. Write performance is more dependent on the total number of memory banks and does not change much with the number of I/O channels used to connect to memory banks.

As mentioned, increasing the number if I/O channels only benefit when bandwidth is low (8-bit, 25 MHz I/O bus). Although more I/O channels will cost more and draw more power, 35% overall performance improvement might justify it in this case. On the other hand, one could get close to 50% improved performance if instead a higher bandwidth bus is utilized. This demonstrates another design choice where cost and power consumption should be weighted in. Several factors such as technology, manufacturing costs, pin count, and operating voltage will have an impact. When performance/cost is the deciding factor, design with one more I/O channel and 35% better performance may be better or worse than a design with a higher bandwidth I/O bus and 50% better performance.

6.2. Superblocks

Another way to hide write latency in flash memory and to improve both read and write performance is ganging blocks across individual flash memory banks to create superblocks [57]. In this array organization, individual flash memory banks are combined by shared chip-enable, command signals, and I/O signals. Sharing of command signals enables merging physical blocks across flash arrays to create designated superblocks. This effectively increases the size of available data and cache registers and enables a superblock to process a higher volume of data in one step. Figure 6.12 shows a sample flash array organization with 4-way superblocks and timing diagram for an 8 KB write request.

One advantage of superblocks would be lower pin count due to shared chip-enable, command signals, and I/O signals. Also there is no need for a bus arbitration logic

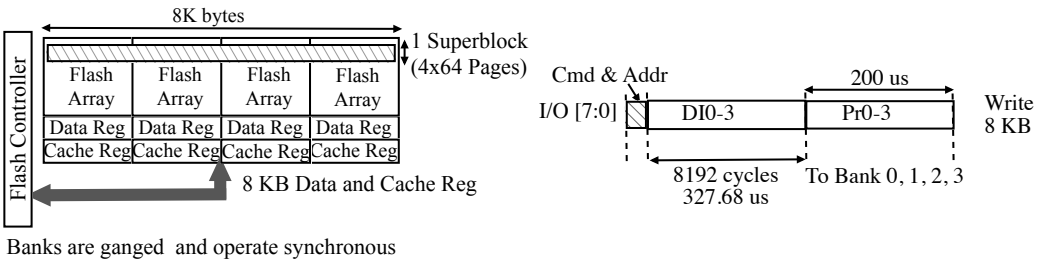


Figure 6.12: Superblocks. 4-way superblocks and timing diagram of an 8 KB write request. I/O bus used in example is 8-bit wide and clocked at 25 MHz.

since all memory banks operate in lock step. One the other hand, flash controller complexity will increase to manage blocks across flash memory banks, as each bank must be checked independently (for example, compare figures 5 & 6 in [57]. Another limitation with superblocks is the fact that blocks are linked together permanently, due to the hardwiring of control and I/O pins. If any one of the blocks in a superblock becomes bad, all blocks in that superblock are considered unusable, thus reducing available storage.

In order to understand the performance impact of superblocks, we have simulated sample configurations where 1, 2, 4 or 8 flash memory banks are configured to form 1-, 2-, 4- or 8-way superblocks and attached to a single I/O bus. Average disk-request response time is reported, which is a sum of physical access time (time to read/write data from/to flash array) and queue wait time. Figures 6.13-15 show the effect of increasing the level of superblocking on average disk-request response time for various user workloads.

As summarized in table 6.3, there are significant improvements in both read and write request times (80%) when the level of superblocking is increased from 1 to 2. Request times may be further improved up to 60% by increasing the level of

All requests	1-way	2-way	4-way	8-way
25 MHz I/O	30.56	6.45	4.95	4.91
100 MHz I/O	24.39	4.43	1.49	0.76
Reads				
25 MHz I/O	15.55	2.53	2.09	2.07
100 MHz I/O	10.06	0.9	0.35	0.27
Writes				
25 MHz I/O	45.13	10.25	7.73	7.68
100 MHz I/O	38.29	7.85	2.59	1.23

Table 6.3: 1-, 2-, 4-, and 8-way superblocks. Average request service time in milliseconds using single shared 8-bit I/O bus.

superblocking from 2 to 4. However, from 4- to 8-way superblocks, reads and writes start to show different performance characteristics. While request times continue to improve for reads; write-request performance improves even more, moving from 4- to 8-way banking. This is due to the fact that programming in flash memory takes several times longer than reading and superblocking directly attacks this programming latency problem.

At the same time, performance of superblocking is dependent on the speed of I/O bus. When I/O bandwidth is low, moving from 4-way to 8-way superblocks hardly improves the performance. But when I/O bandwidth is high, performance improvements are 50-60%. In order to achieve better performance at high levels of superblocking, one also has to increase the bandwidth of the system. Cost and power consumption of higher

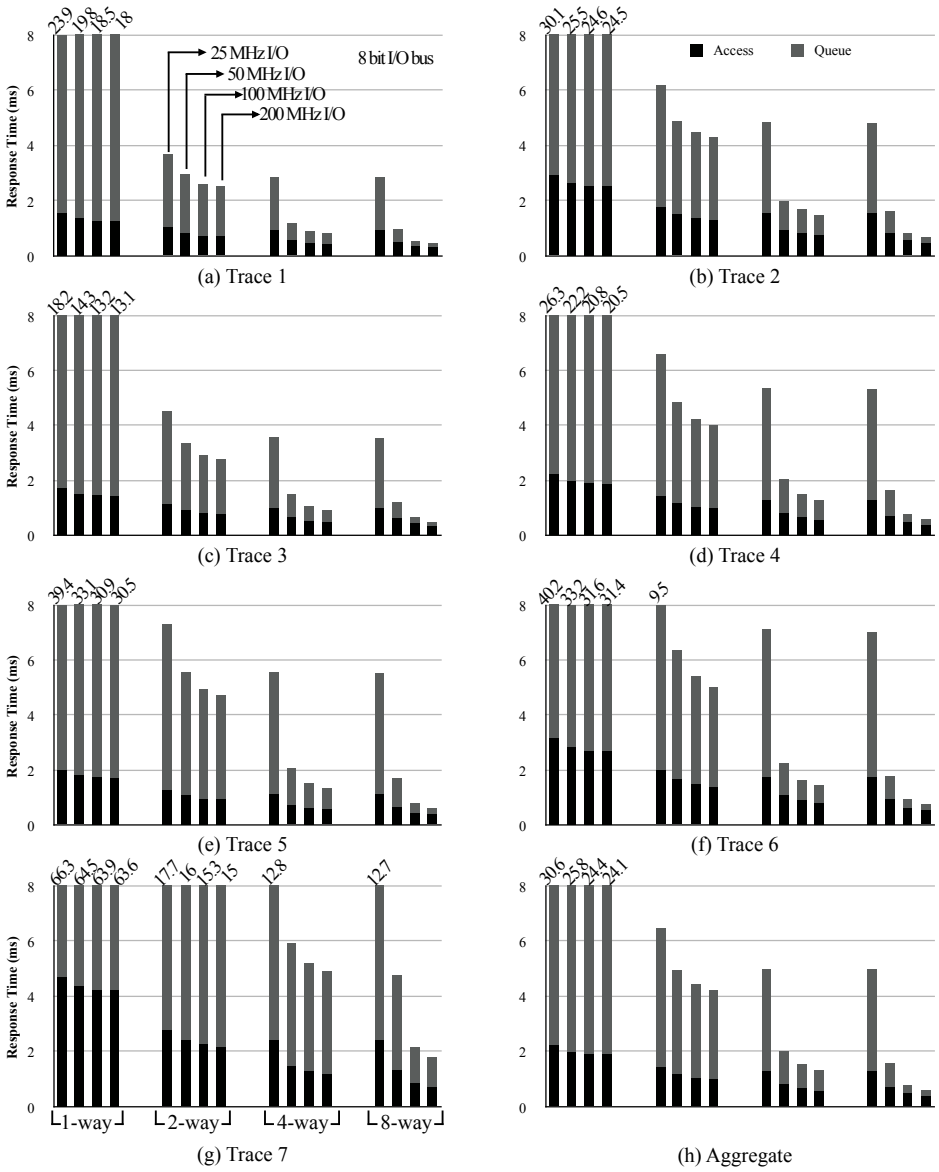


Figure 6.13: Single I/O bus, 1-, 2-, 4-, and 8-way superblocks. Average request service time in milliseconds using a shared 8-bit I/O bus for various user workloads is shown. Request times improve significantly when the level of superblocking is increased and I/O bandwidth is high.

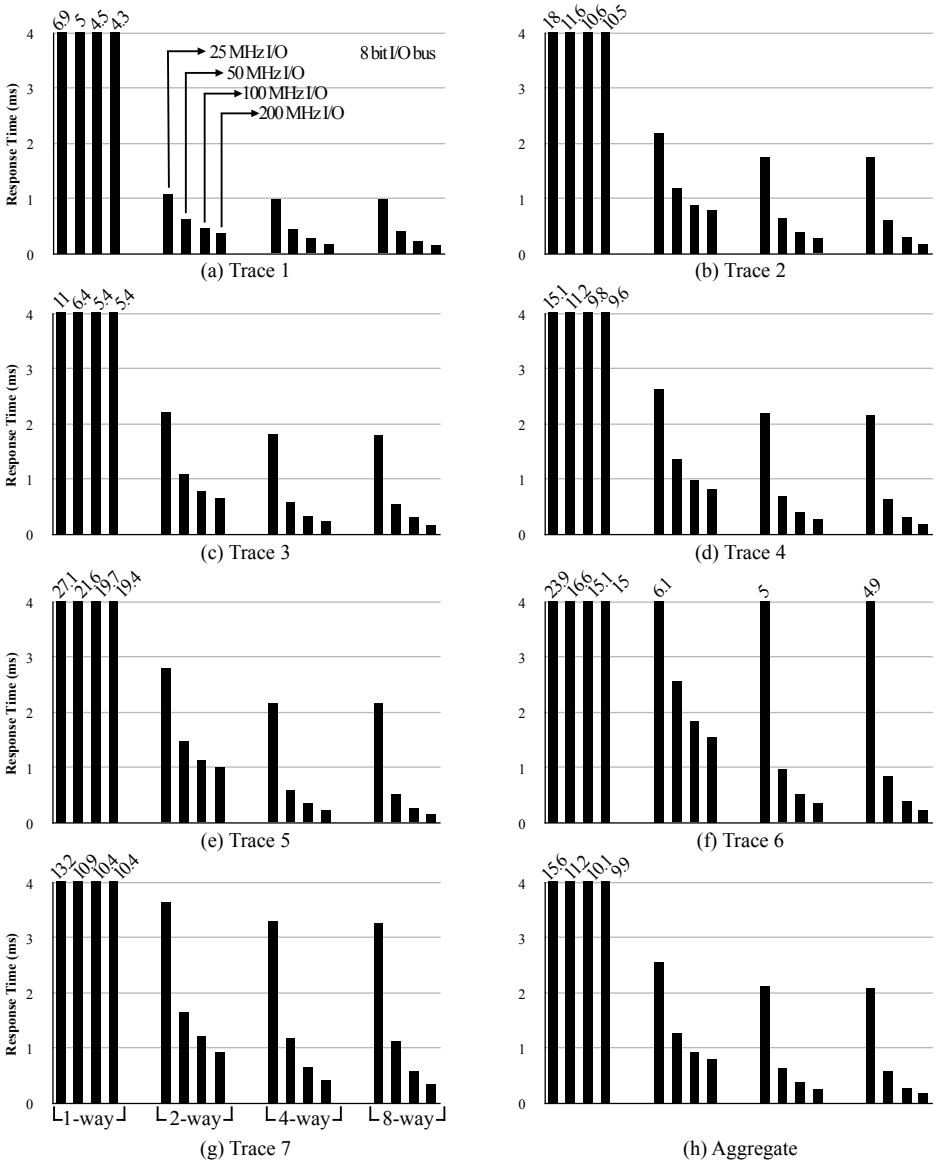


Figure 6.14: Reads with single I/O bus, 1-, 2-, 4-, and 8-way superblocks. Average read request service time in milliseconds using a shared 8-bit I/O bus for various user workloads is shown. Request times improve significantly when the level of superblocking is increased and I/O bandwidth is high.

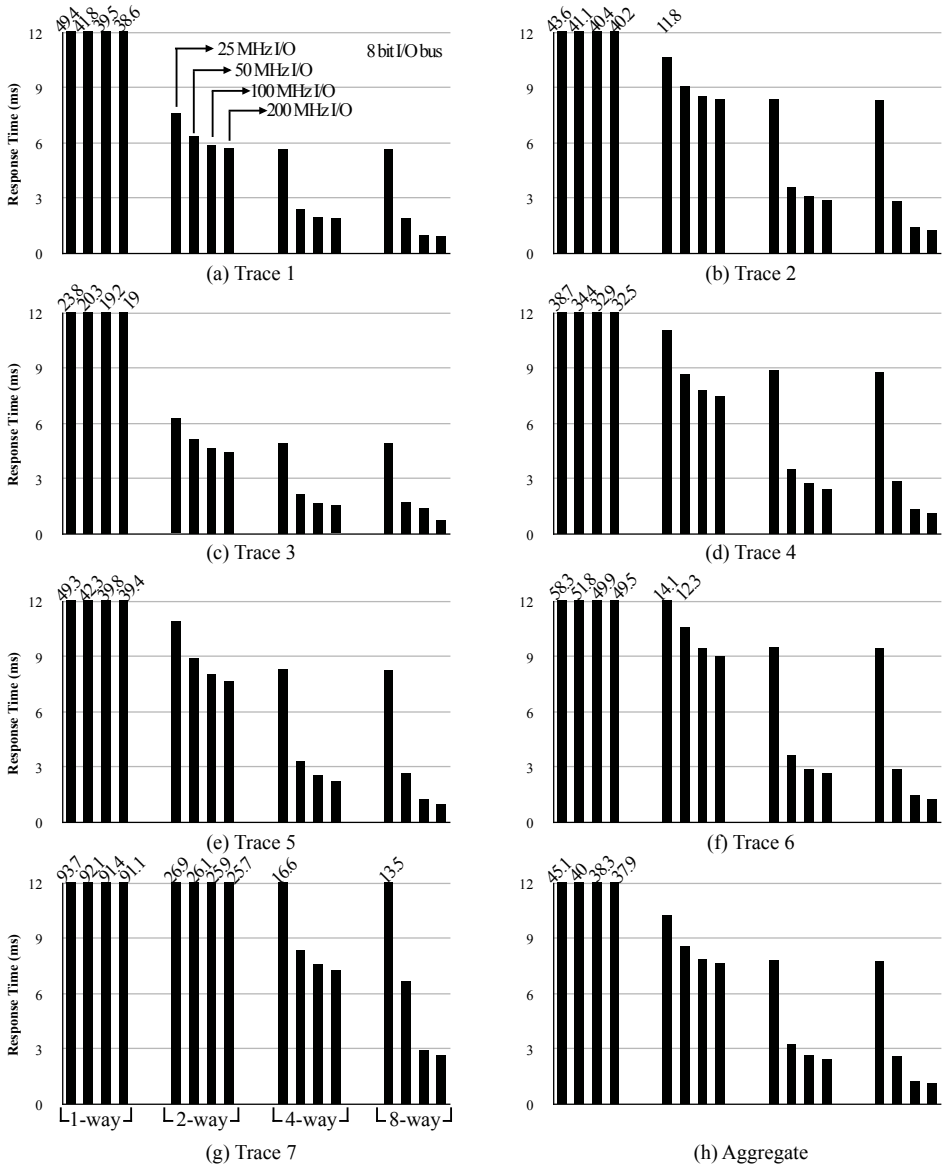


Figure 6.15: Writes with single I/O bus, 1-, 2-, 4-, and 8-way superblocks. Average write request service time in milliseconds using a shared 8-bit I/O bus for various user workloads is shown. Request times improve significantly when the level of superblocking is increased and I/O bandwidth is high.

bandwidth I/O channels should be added to the cost of higher controller complexity, increasing the overall cost of superblocking.

High variation on write request performance (depending on the workload) and the 2-3x scale difference between read and write performance continue to be the case for superblocks as well. These indicate that it is possible to further improve system performance by employing various scheduling techniques and run-time resource allocation policies. Also, as shown in figure 6.13, requests spend a significant portion of time waiting in queue even with high levels of superblocking. One way to reduce queue times may be combining superblocks with banking and request interleaving.

6.3. Concurrency: Banking vs. Superblocks

Superblocks and request interleaving are methods that exploit concurrency at device- and system-level to significantly improve performance. These system- and device-level concurrency mechanisms are, to a significant degree, orthogonal: that is, the performance increase due to one does not come at the expense of the other, as each exploits a different facet of concurrency exhibited within the PC workload. One can formulate concurrency as “number of banks times superblocking within a bank”.

Banking and request interleaving exploits concurrency at system-level by treating flash memory banks as individual flash devices. As shown in section 6.1; by accommodating interleaved organizations, one can achieve significant levels of concurrency in an SSD without changing its overall size and shape. If we consider the anatomy of an I/O, banking and request interleaving improves performance by reducing queueing times. If more banks are available in the system, more requests can be

dispatched to these banks and each request will wait less time in queue. Although physical access times should not change by the level of banking in theory, there will be additional overheads when multiple independent banks share a limited system bandwidth.

Superblocks enable simultaneous read or write of multiple pages within a flash device or even across different flash dies [57]. By exploiting concurrency at the device level, one can effectively double NAND flash memory read and write bandwidth. If we consider anatomy of an I/O, superblocking improves performance by reducing physical access times. As the level of superblocking increases, read and write request access times improve.

The obvious question is which of these concurrency techniques is better? In order to answer this question, we have simulated different SSD organizations at different levels of concurrency. Variables in this space include bus organizations (widths, speeds, etc), banking strategies, and superblocking methods. We have simulated concurrency levels of 2, 4, 8 and 16. Within each level, we exploited concurrency by banking, by superblocking ,or by a combination of two. For example; concurrency of 4 can be achieved by connecting 4 individual memory banks, or 1 memory bank using 4-way superblocking, or connecting 2 individual memory banks, each of which use 2-way superblocking. Single I/O bus at 25, 50, 100 and 200 MHz is used. Average disk-request response time is reported, which is a sum of physical access time (time to read/write data from/to flash array) and queue wait time.

I/O Rate	Concurrency = Number of Banks x Superblocks												
All Req.	**2x1**	**1x2**	**4x1**	**2x2**	**1x4**	**8x1**	**2x4**	**4x2**	**1x8**	**8x2**	**4x4**	**2x8**	**1x16**
25 MHz	6.85	6.45	4.92	4.56	4.95	4.68	4.37	4.29	4.91	4.27	4.48	4.39	4.89
100 MHz	5.34	4.43	2.67	1.77	1.49	2.16	1.24	0.84	0.76	1.13	0.72	0.66	0.67
Read													
25 MHz	3.01	2.53	2.41	2.28	2.09	2.38	2.17	2.17	2.07	2.15	2.26	2.17	2.06
100 MHz	1.2	0.9	0.58	0.4	0.35	0.47	0.32	0.29	0.27	0.3	0.27	0.27	0.27
Write													
25 MHz	10.5	10.3	7.31	6.75	7.73	6.91	6.48	6.31	7.68	6.31	6.61	6.5	7.66
100 MHz	9.31	7.85	4.68	3.09	2.59	3.79	2.13	1.37	1.23	1.92	1.16	1.04	1.07

Table 6.4: Request time of Flash SSD Organizations. Average request service time in milliseconds using 8-bit I/O bus.

As summarized in table 6.4 and shown in figures 6.16-21, increasing concurrency by means of either superblocking or banking results in different performance depending on the system bandwidth.

For low bandwidth systems (8 bit 25 MHz I/O bus) increasing concurrency beyond 4 (4 banks, 2 banks with 2-way superblocking each, or one bank with 4-way superblocking) does not result in any further performance gains. When the level of concurrency is 4 or more, all configurations of banking and superblocks perform within 7% of each other. This proves that at 25 MBps the storage system is limited by I/O

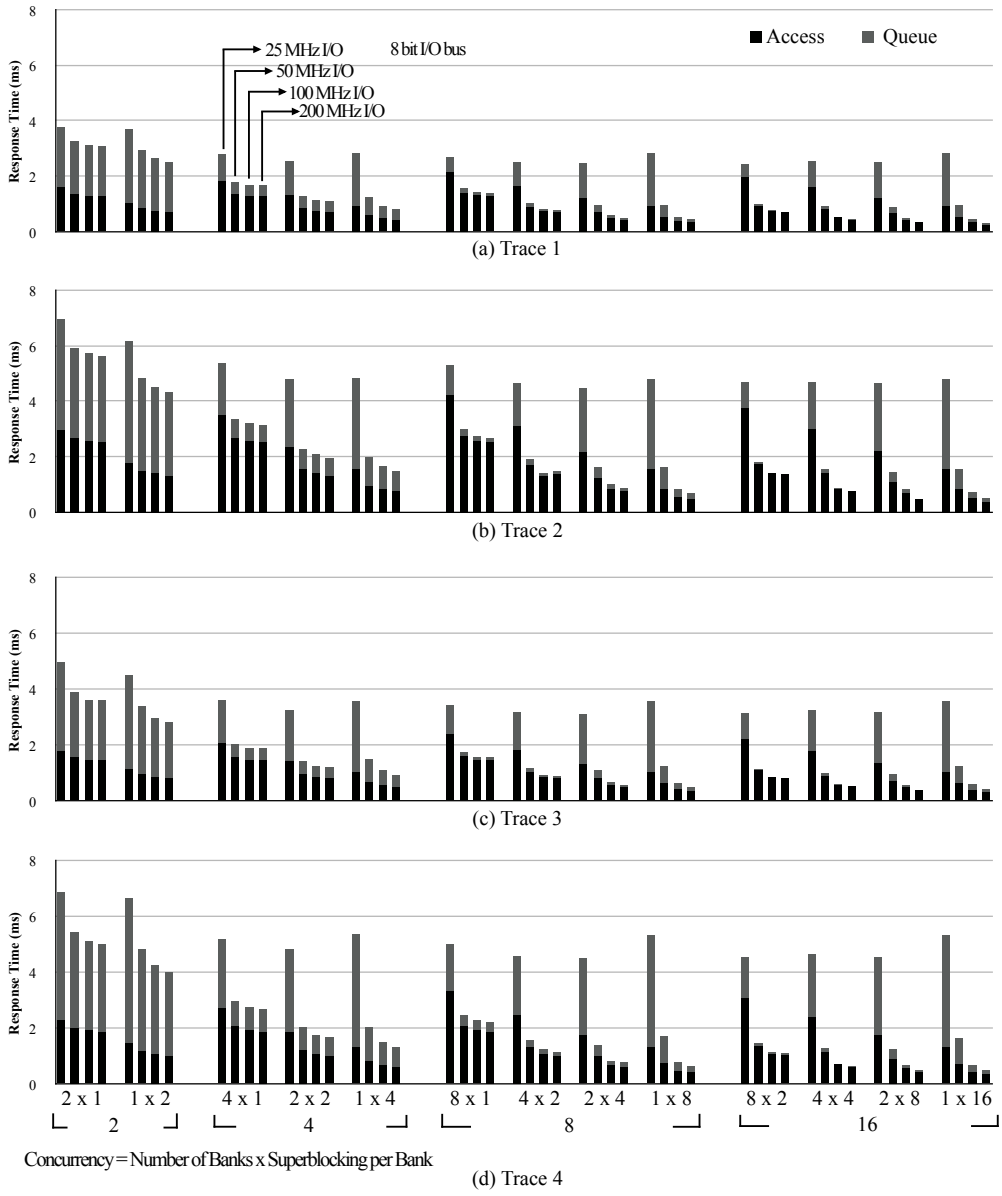


Figure 6.16: 2, 4, 8, and 16 level concurrency. Average request service time in milliseconds using a shared 8-bit I/O bus for user workloads 1-4 is shown. Concurrency is defined as "Number of Banks x Level of Superblocking per Bank". For low bandwidth systems, increasing concurrency beyond 4 does not result in any further performance improvements.

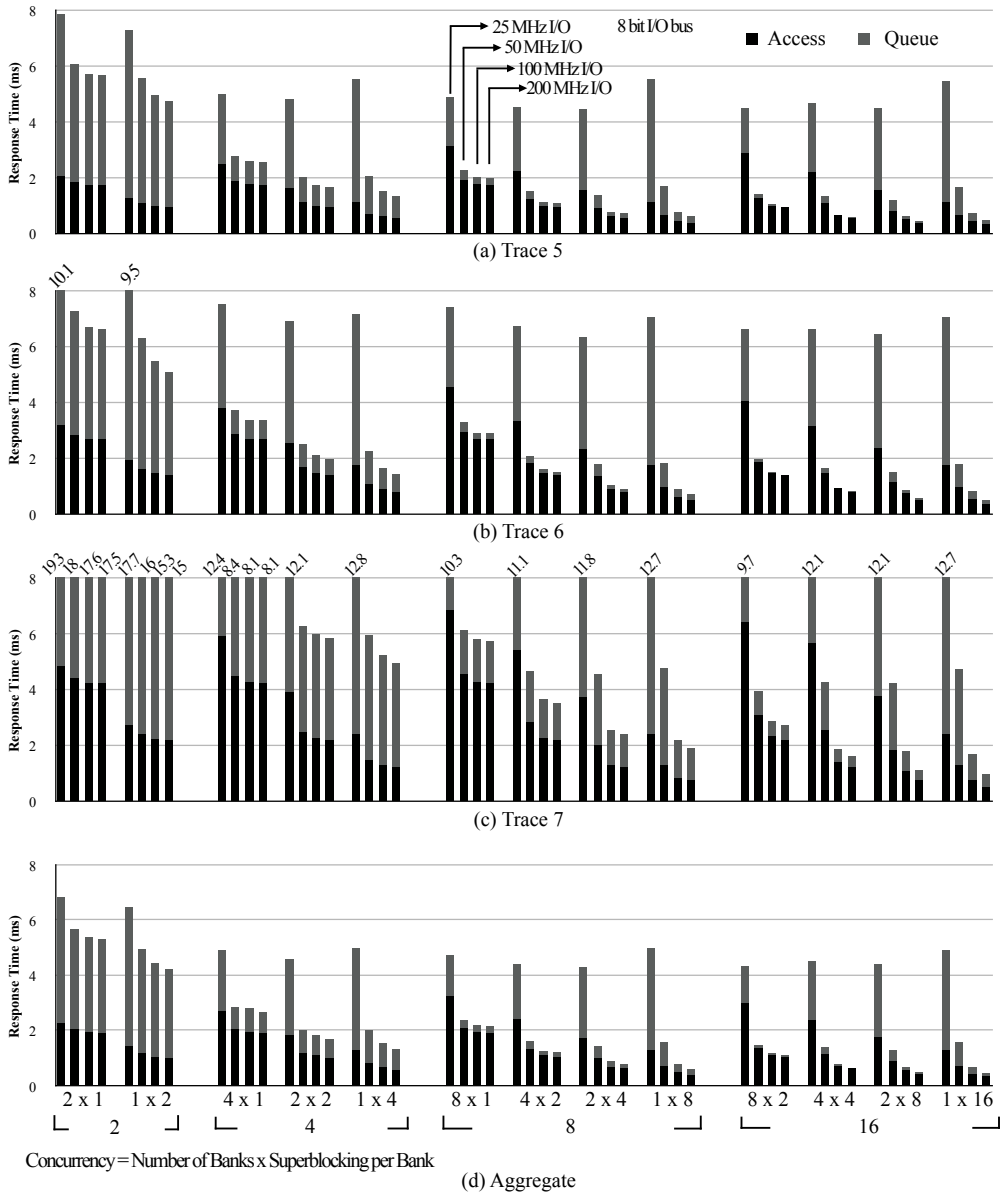


Figure 6.17: 2, 4, 8, and 16 level concurrency. Average request service time in milliseconds using a shared 8-bit I/O bus for user workloads 5-7 is shown. Concurrency is defined as "Number of Banks x Level of Superblocking per Bank". For low bandwidth systems, increasing concurrency beyond 4 does not result in any further performance improvements.

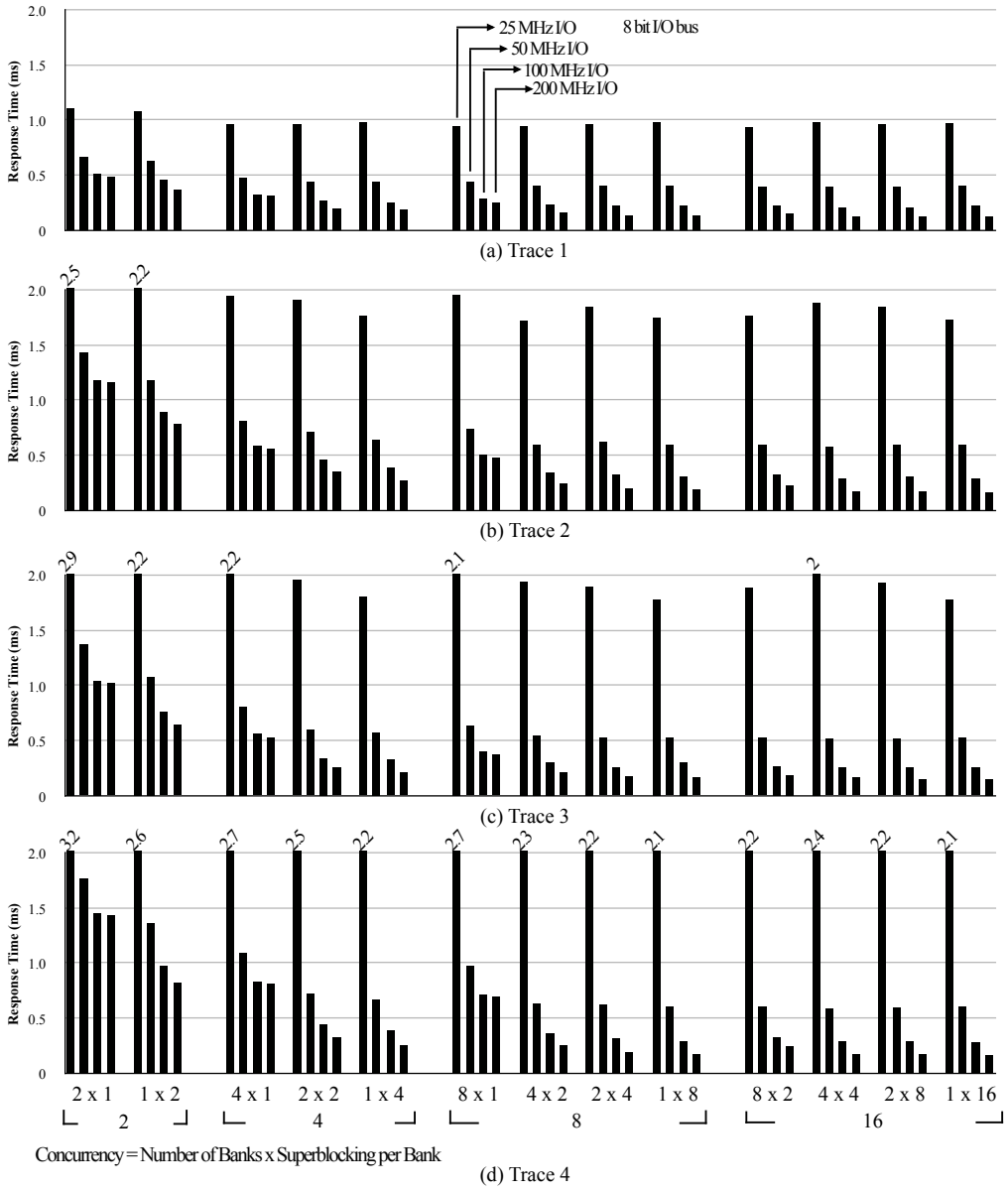


Figure 6.18: Reads with 2, 4, 8, and 16 level concurrency. Average read request service time in milliseconds using a shared 8-bit I/O bus for user workloads 1-4 is shown. Concurrency is defined as "Number of Banks x Level of Superblocking per Bank".

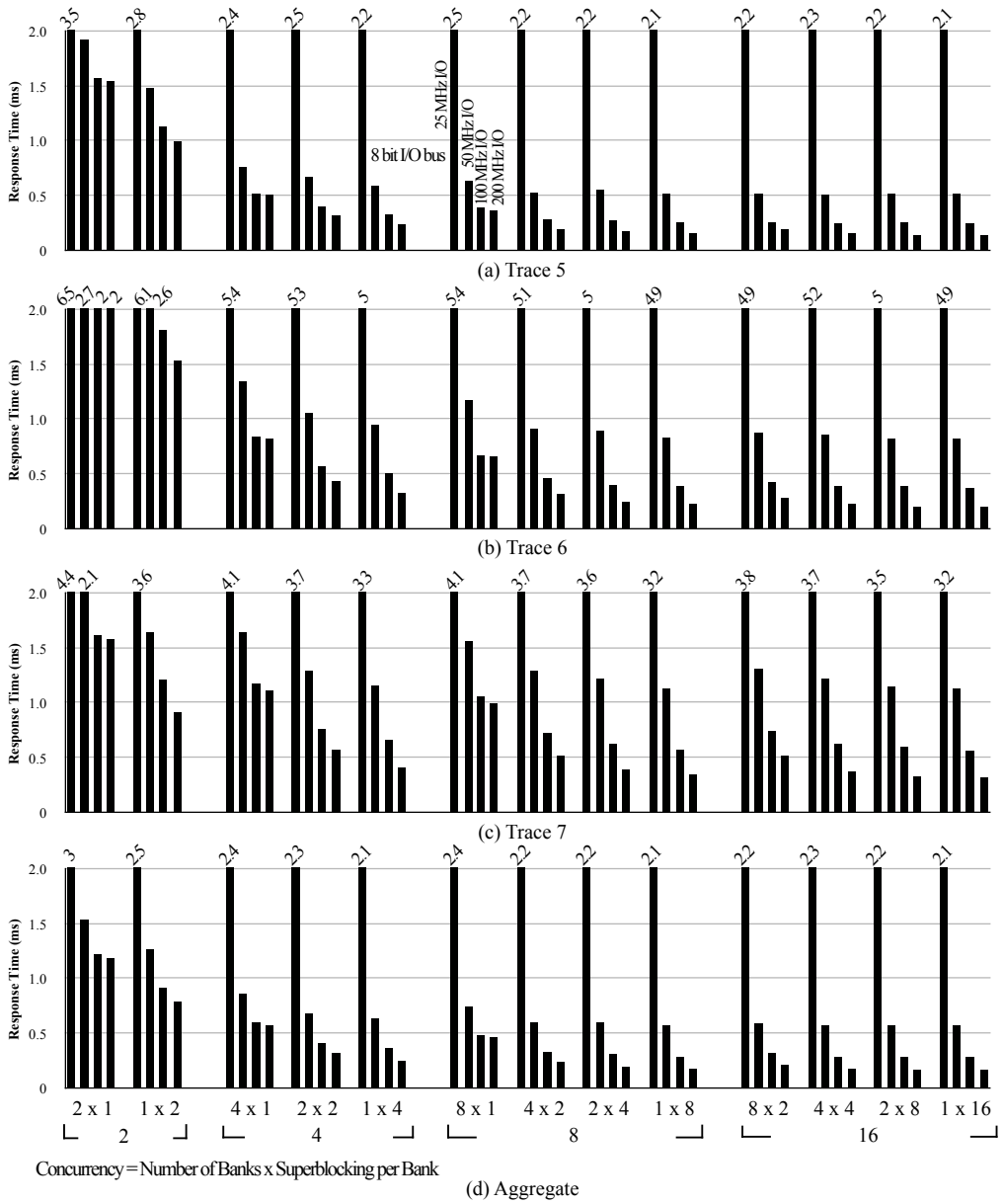


Figure 6.19: Reads with 2, 4, 8, and 16 level concurrency. Average read request service time in milliseconds using a shared 8-bit I/O bus for user workloads 5-7 is shown. Concurrency is defined as "Number of Banks x Level of Superblocking per Bank".

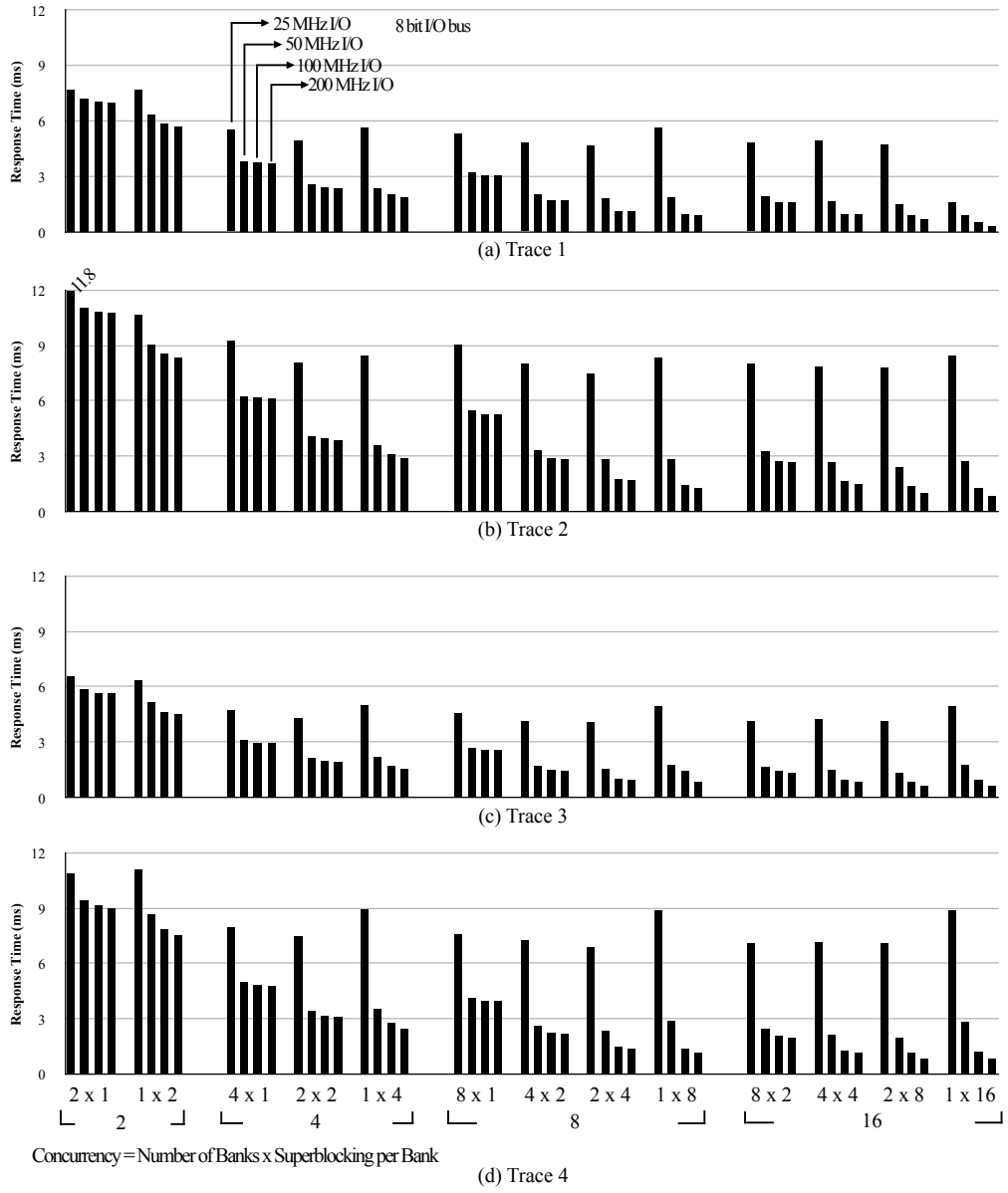


Figure 6.20: Writes with 2, 4, 8, and 16 level concurrency. Average write request service time in milliseconds using a shared 8-bit I/O bus for user workloads 1-4 is shown. Concurrency is defined as "Number of Banks x Level of Superblocking per Bank".

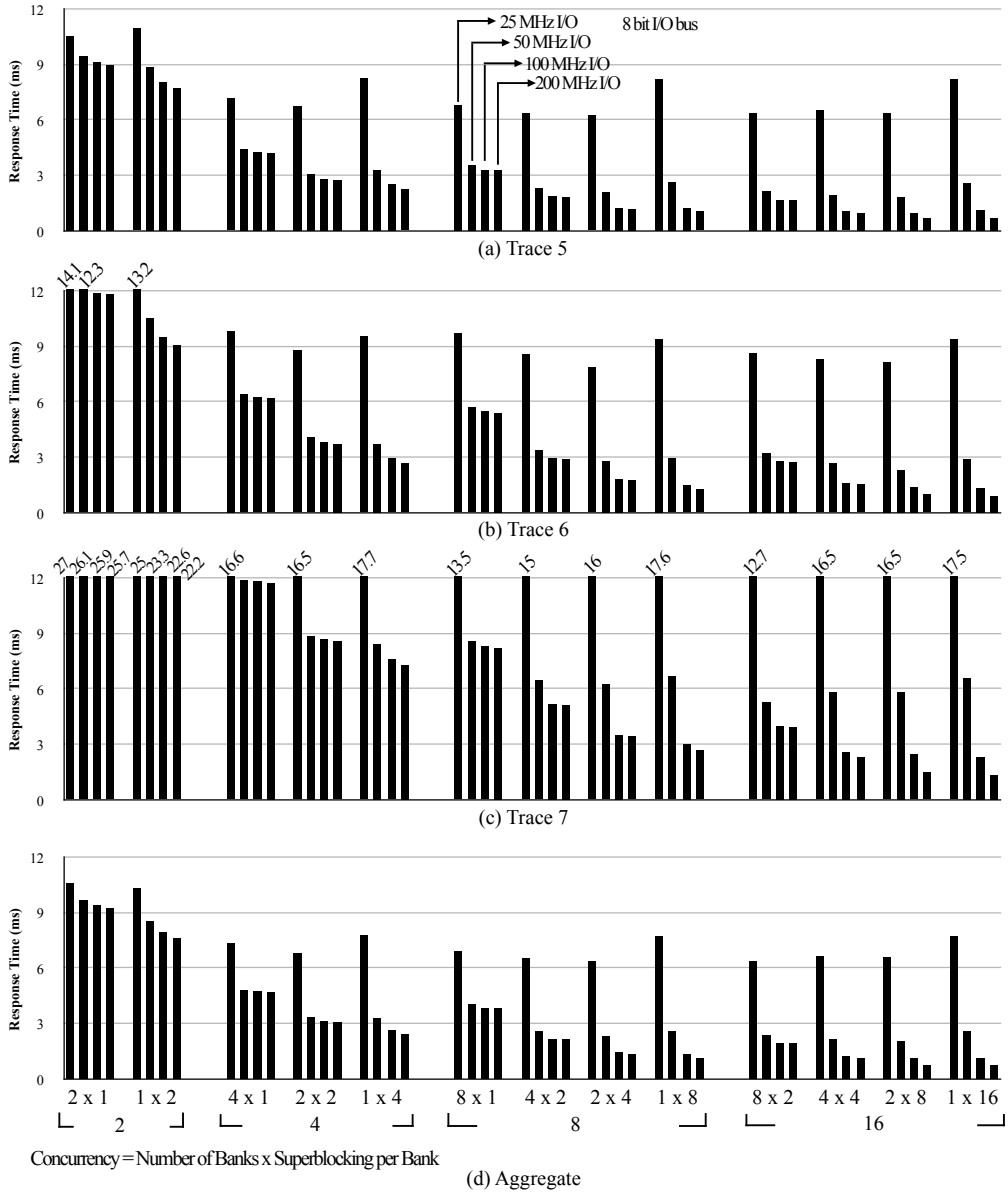


Figure 6.21: Writes with 2, 4, 8, and 16 level concurrency. Average write request service time in milliseconds using a shared 8-bit I/O bus for user workloads 5-7 is shown. Concurrency is defined as "Number of Banks x Level of Superblocking per Bank".

bandwidth. However, if one has to operate at low bandwidth either for cost cutting or power saving purposes, choosing a configuration heavily depends on the user workload. If overall request time is used as the performance metric, often times a configuration which uses a combination of banking and superblocking is ideal. For example; when the concurrency level is 8, in all traces except user trace 7, 8-way banking and 8-way superblocking perform slower than 2-way banking with 4 superblocks or 4-way banking with 2 superblocks. If read and write performance is evaluated individually, reads favor higher levels of superblocking. This is explained by the lesser probability of I/O bus contention. Since all memory blocks in a superblock operate in synch, there is no additional time spent in bus arbitration. Writes follow the same overall trend that combining banking and superblocking is more effective than only banking or only superblocking. This conclusion is counter intuitive. Superblocking especially improves write throughput by exploiting concurrency at device level. Therefore one would expect writes to benefit most from a configuration which uses a higher degree of superblocking. This is explained by the nature of user requests. As shown in chapter 5, in a typical user workload most writes are 4 or 8 Kbytes in size. With 2- and 4-way superblocking, data and cache registers are 4 and 8 Kbytes in size as well, thus fitting the request size. An 8 Kbytes write request will not benefit as much from 8-way or 16-way superblocking as it will underutilize data and cache registers. Such requests may be served faster when there are two memory banks each of which employing 4-way superblocks rather than when there is single memory bank with 8-way superblocking.

For high bandwidth systems (100 MBps and higher) increasing concurrency always improves performance. Also within a fixed level of concurrency, configuration with the highest degree of superblocking always performs better. This proves that the real limitation to NAND flash memory performance is its core interface. Exploiting concurrency at system level by banking and request living can workaround this limitation to an extent. Best performance is achieved when NAND interface performance is improved.

6.4. Media Transfer Rate

One of the factors limiting flash memory performance is considered to be its media transfer rate. In current flash memory devices, 8-bit 33 MHz I/O buses are common. As HDDs with 7200, 10K or 15K RPM are popular, and disk interface speeds are scaling up with serial interface and fiber channel, NAND flash SSD's performance is expected to be limited by its media transfer rate.

Open NAND Flash Interface (ONFI) is an industry workgroup dedicated to NAND flash memory integration into consumer electronic products by resolving compatibility and interoperability of NAND devices from different vendors. ONFI is made of more than 80 companies, including Hynix, Micron, Intel, Sony, Hitachi, and Kingston [67]. One of the new features of ONFI 2.1 is 166 and 200 MBps interface speed as an improvement over legacy NAND bus. Source synchronous DDR interface is used to achieve speeds up to 133 MBps with ONFI 2.0 and 166 MBps and higher with ONFI 2.1.

The impact of NAND I/O bus speed may be best observed in the context of sustainable read speed. Often times for storage systems, sustainable read and write

performance is considered as the performance metric. One of the popular file system benchmarks used to measure Unix file system performance is Bonnie [10]. Bonnie benchmark tests the performance of reading 100 Mbytes file. If we consider reading a file of 100 Mbytes in flash, this will correspond to reading 51200 pages of 2 Kbyte each. Considering read cache mode - as explained in chapter 3.3.2 - it will take 25 μs + 51200 * 3 μs + 2048 * 51200 * T(I/O bus). If 33 MBps I/O bus is used, the result is 3.3 seconds. If 133 MBps I/O bus is used, the result is 0.9 seconds, 73% faster. In both cases, more than 95% of the time is spent in data transfer through I/O bus.

In order to find the impact of the media transfer rate on performance, we have simulated different SSD organizations with different I/O bus speeds. We have simulated I/O rates of 25, 50, 100, 200 an 400 MBps with 2-, 4-, 8-way banking or superblocks. Average disk-request response time is reported, which is a sum of physical access time (time to read/write data from/to flash array) and queue wait time. Figures 6.22-24 illustrate average disk-request response time, average read time, and average write time for various user workloads.

In all storage system configurations, performance does not improve significantly beyond 100 MBps. This can be explained by the nature of user workloads. In a typical PC user workload, read and write requests are random, 4 to 8 KB in size with a read to write ration of approximately 50:50. With such workloads, random read and write performance dominates rather than sustainable throughput. For example, reading and writing of a large file such as 100 Mbytes file used in Bonnie benchmark is observed in less than 1% of the time.

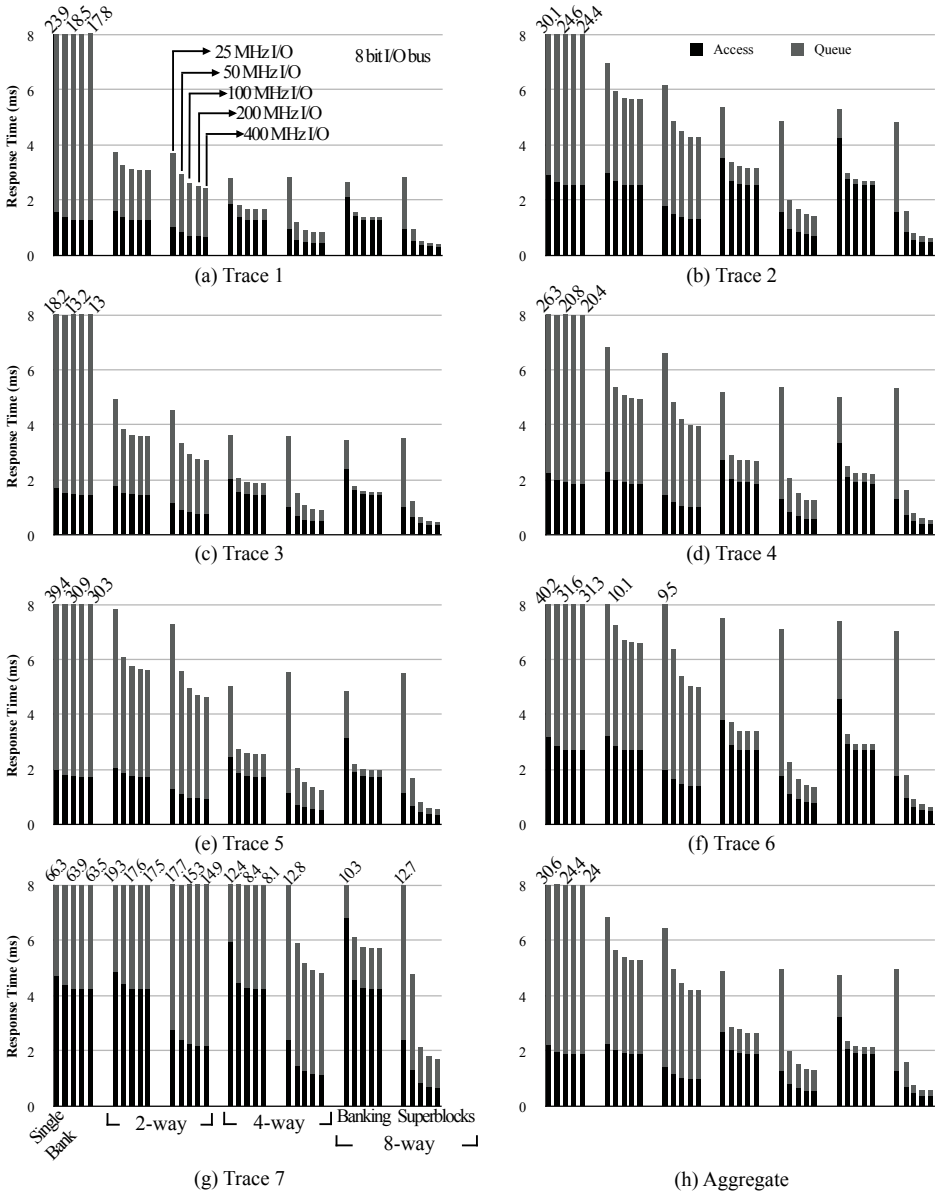


Figure 6.22: Media Xfer Rate. Average request service time in milliseconds using an 8-bit I/O bus for changing bus speeds and various user workloads is shown. Performance does not improve much beyond 100 MBps.

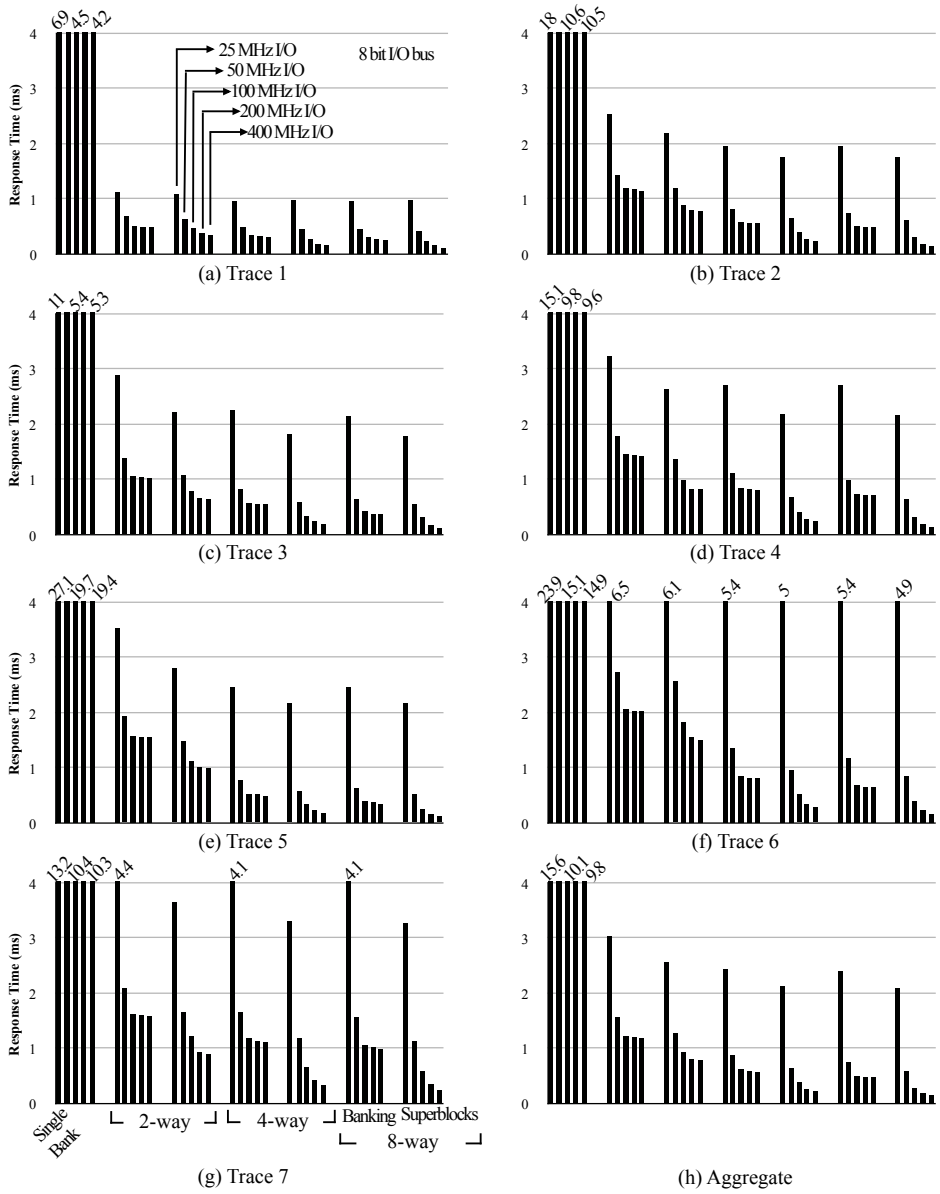


Figure 6.23: Reads - Media Xfer Rate. Average read request service time in milliseconds using an 8-bit I/O bus for changing bus speeds and various user workloads is shown. Performance does not improve much beyond 100 MBps.

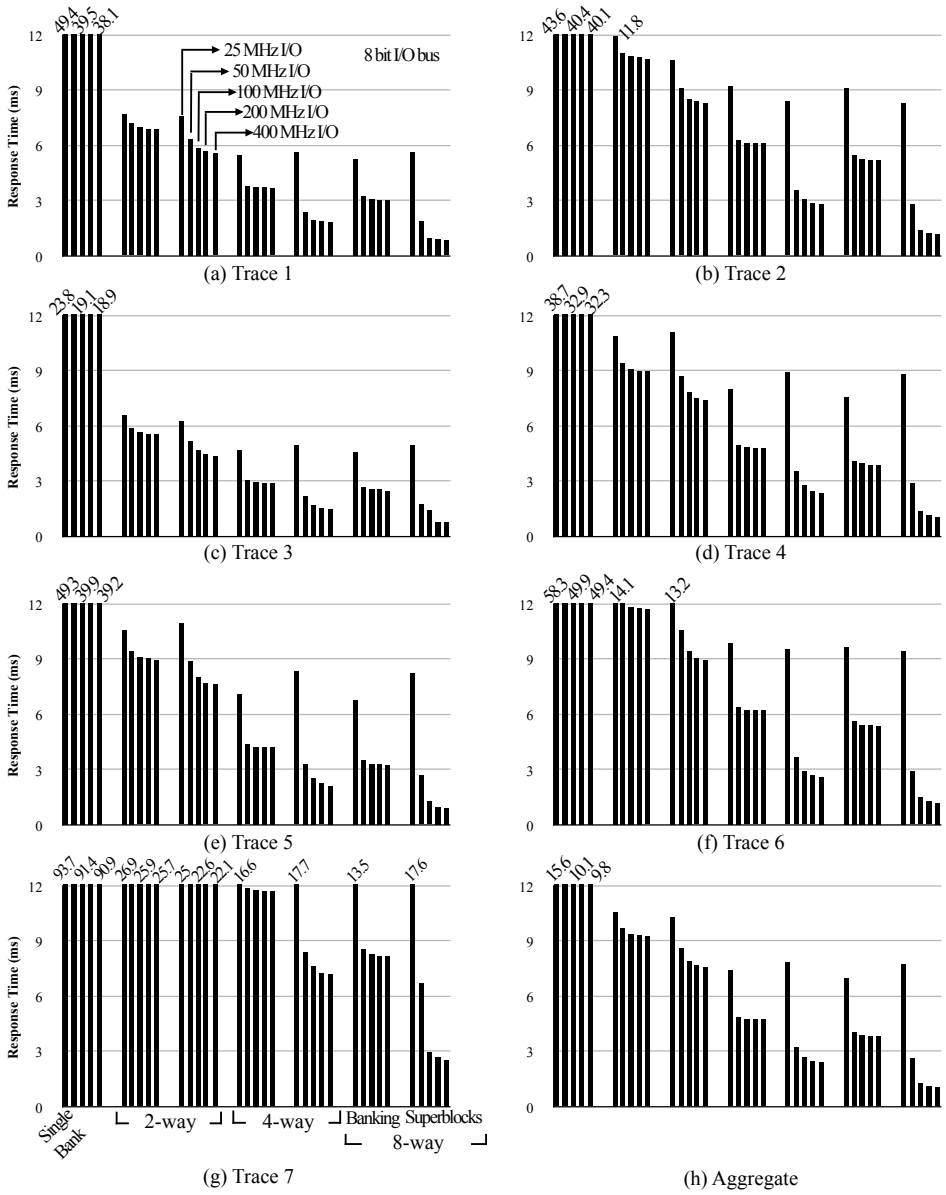
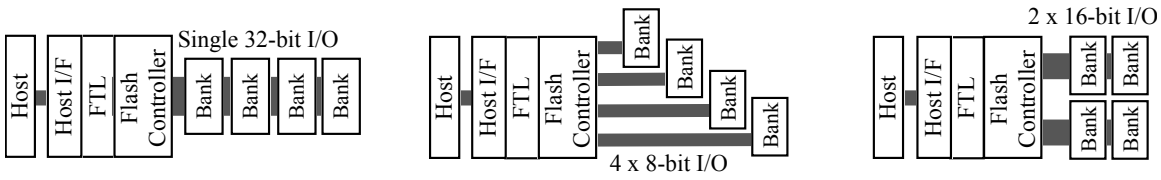


Figure 6.24: Writes - Media Xfer Rate. Average write request service time in milliseconds using an 8-bit I/O bus for changing bus speeds and various user workloads is shown. Performance does not improve much beyond 100 MBps.

For typical PC user workloads, the real limitation to NAND flash memory performance is not its low media transfer rate but its core interface. The random nature of workloads and small request sizes are the two main factors. If server workloads are considered, the impact of the media transfer rate can be of great importance. Server workloads often times read and write in large sizes. And sustainable throughput is one of the key metrics in reporting server performance. It is possible to observe slightly different results with different workload timings. If a workload is sped up by compressing inter-arrival times, one may see performance improvements beyond 100 MBps. For typical user workloads bus idle times dominate which in return underestimates performance improvements due to increased bus speed. As discussed, source synchronous DDR interface can be used to achieve speeds up to 200 MBps and possibly further. Moreover there are new flash memory architectures proposed which can achieve speeds considerably higher than 200 MBps by using ring style organizations. For example, HLNAND is one company whose NAND flash memory architecture uses up to 255 flash memory banks connected in a daisy-chain cascade. This architecture can run at speeds of up to 800 MBps [37].

6.5. System Bandwidth and Concurrency

As we have showed previously, flash memory performance can be improved significantly if request latency is reduced by dividing the flash array into independent banks and utilizing concurrency. The flash controller can support these concurrent requests through multiple flash memory banks via the same channel or through multiple independent channels to different banks, or a combination of the two. This is equivalent to saying, “I



(a) Single shared channel (b) Dedicated I/O channel for each bank (c) Multiple shared channels

Figure 6.25: Connecting 4 flash memory banks. Dedicated and shared I/O channels are used to connect 4 flash memory banks. Bus widths of 8, 16 and 32 bits are used. Bus speeds change from 25 MHz to 100 MHz. (a) Single 32-bit wide I/O bus is shared; (b) 32-bit I/O is divided into 4 8-bit dedicated I/O channels; (c) 32-bit I/O is divided into 2 16-bit channels and each channel is shared by 2 memory banks.

have 4 flash memory banks, how should I connect them? Should I use four 50 MHz 8-bit I/O channels, should I gang them together for a 32-bit I/O channel? Should I use faster or wider I/O? What should I do?".

To answer these questions, we have simulated a sample configuration with 4 flash memory banks and with various I/O channel configurations which provide total I/O bandwidths ranging from 25 MBps to 1.6 GBps. Figure 6.25 shows sample organizations used to connect 4 flash memory banks. While keeping the total number of memory banks constant at 4, we have simulated I/O channel widths of 8, 16, and 32 bits with I/O bus speeds of 25, 50, 100 MHz - the total I/O bandwidth ranging from 25 MBps (single 8-bit I/O bus at 25 MHz) to 1.6 GBps (4 dedicated 32-bit I/O buses at 100 MHz each). Total I/O bandwidth is calculated as "number of I/O channels x bus width x bus speed". The simulation results for various user workloads are shown in Figures 6.26-31. Average disk-request response time, which is a sum of physical access time (time to read/write data from/to flash array) and queue wait time, average read time, and average write time are reported.

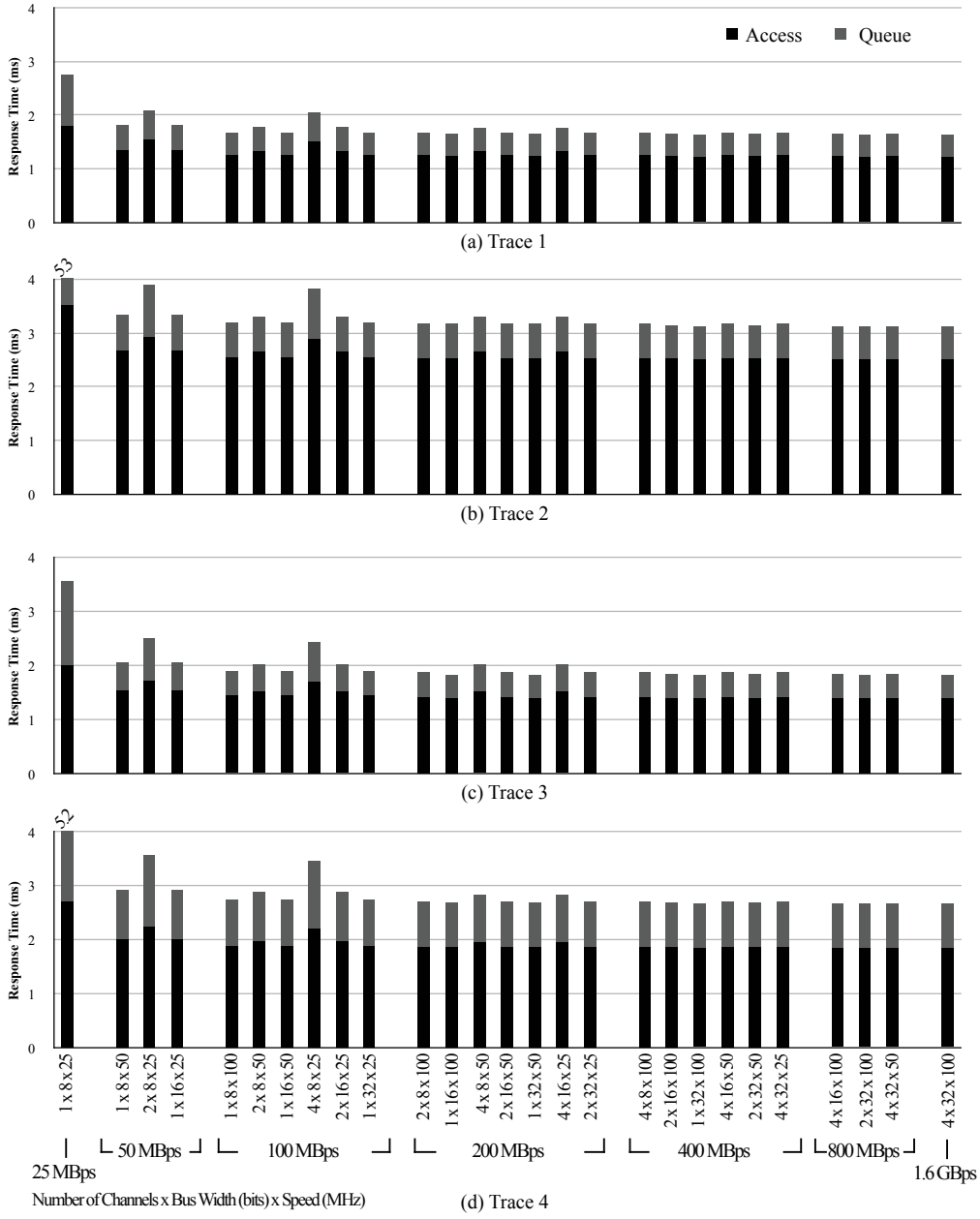


Figure 6.26: System bandwidth. Average request service time in milliseconds with changing system bandwidth for user workloads 1-4 is shown. Overall performance does not change much beyond 200 MBps.

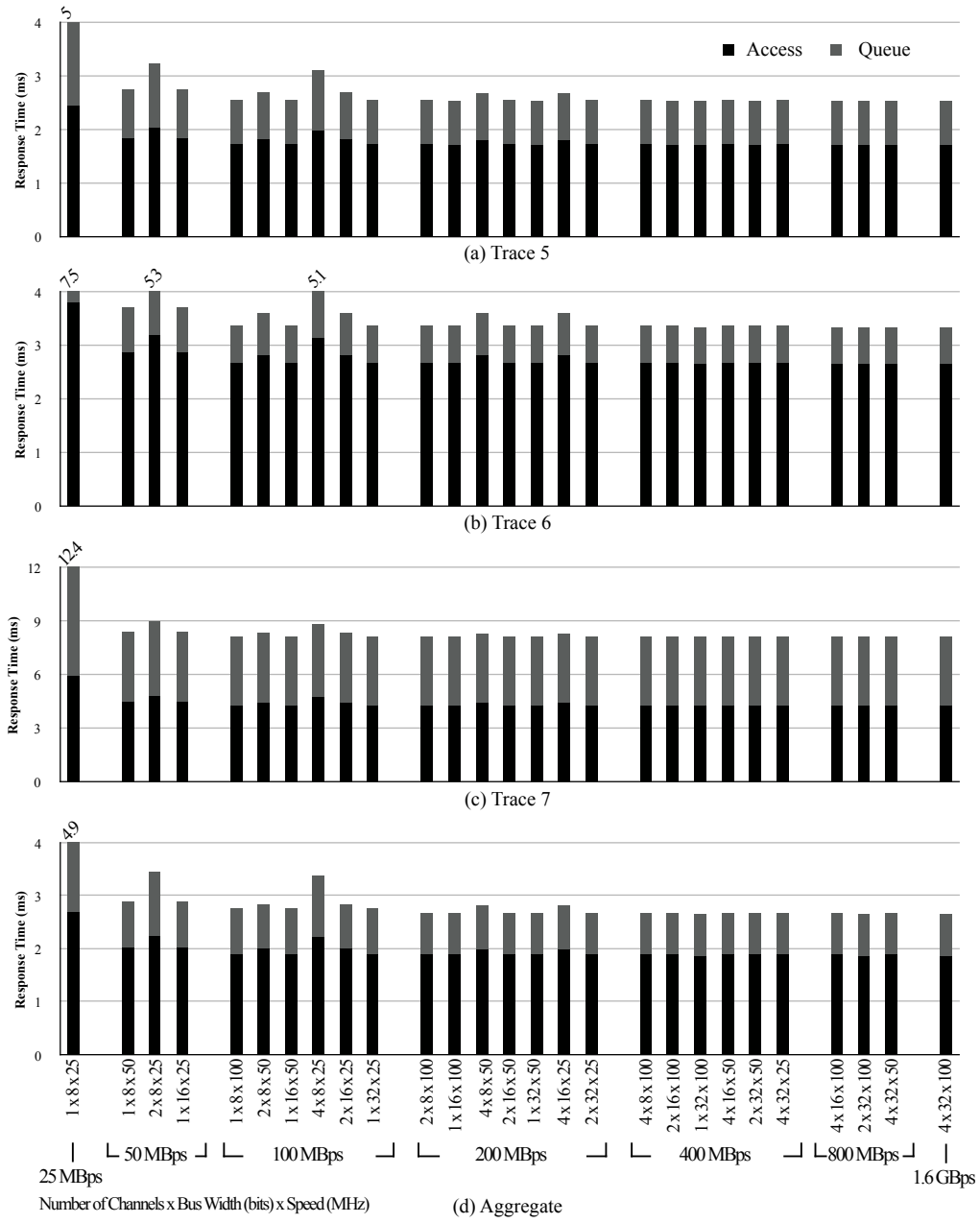


Figure 6.27: System bandwidth. Average request service time in milliseconds with changing system bandwidth for user workloads 5-7 is shown. Overall performance does not change much beyond 200 MBps.

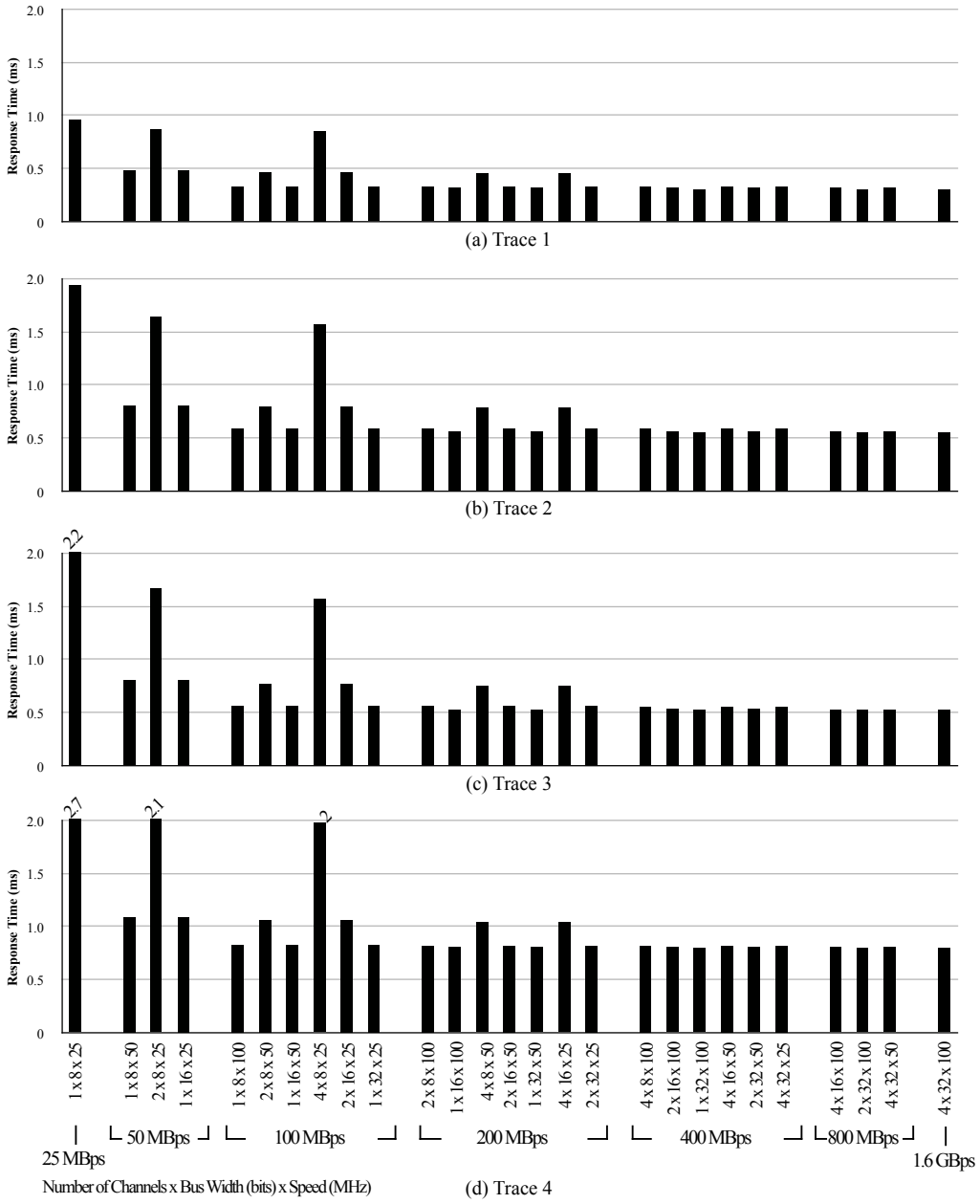


Figure 6.28: Reads - System Bandwidth. Average read request service time in milliseconds with changing system bandwidth for user workloads 1-4 is shown. Reads prefer faster I/O bus.

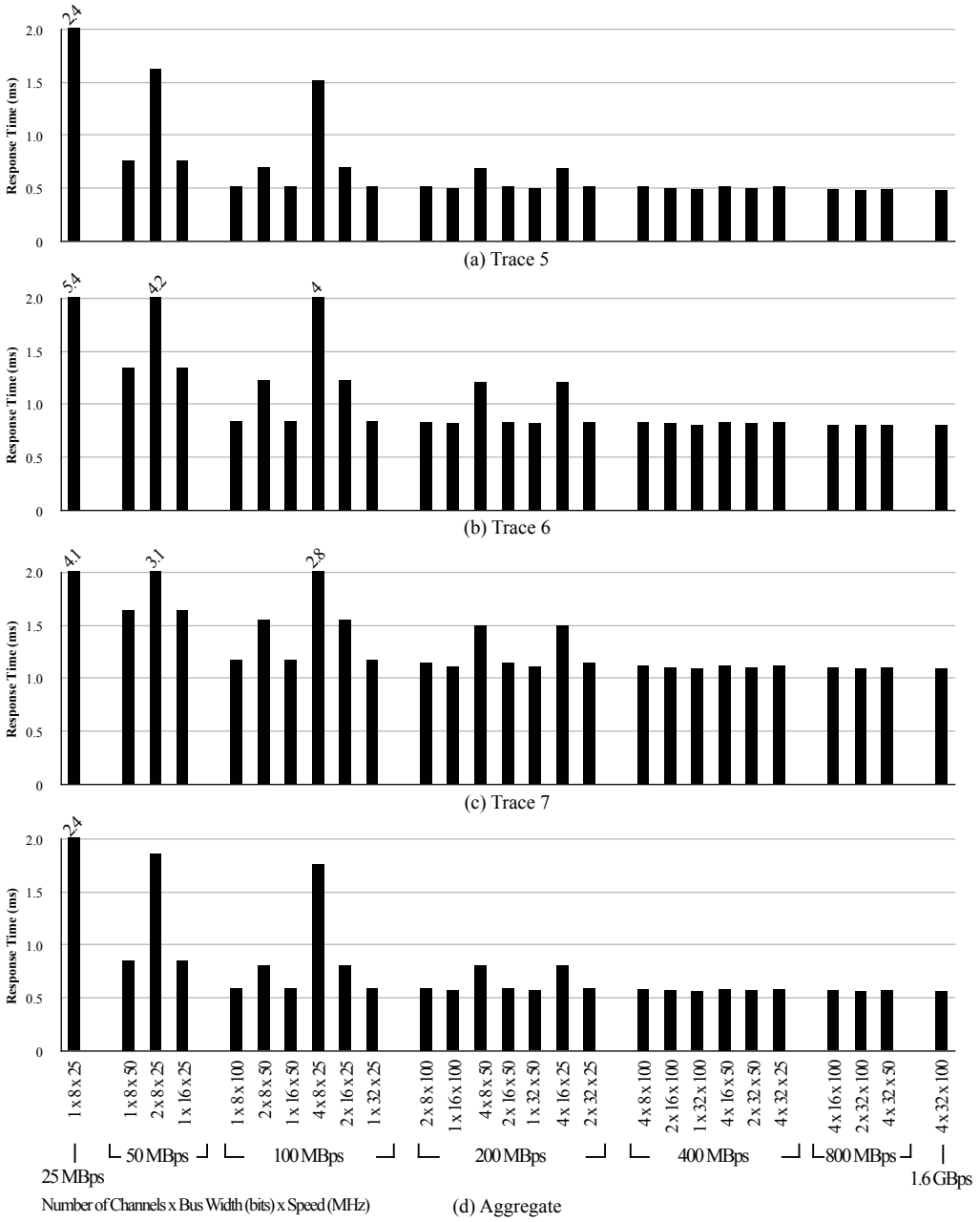


Figure 6.29: Reads - System Bandwidth. Average read request service time in milliseconds with changing system bandwidth for user workloads 5-7 is shown. Reads prefer faster I/O bus.

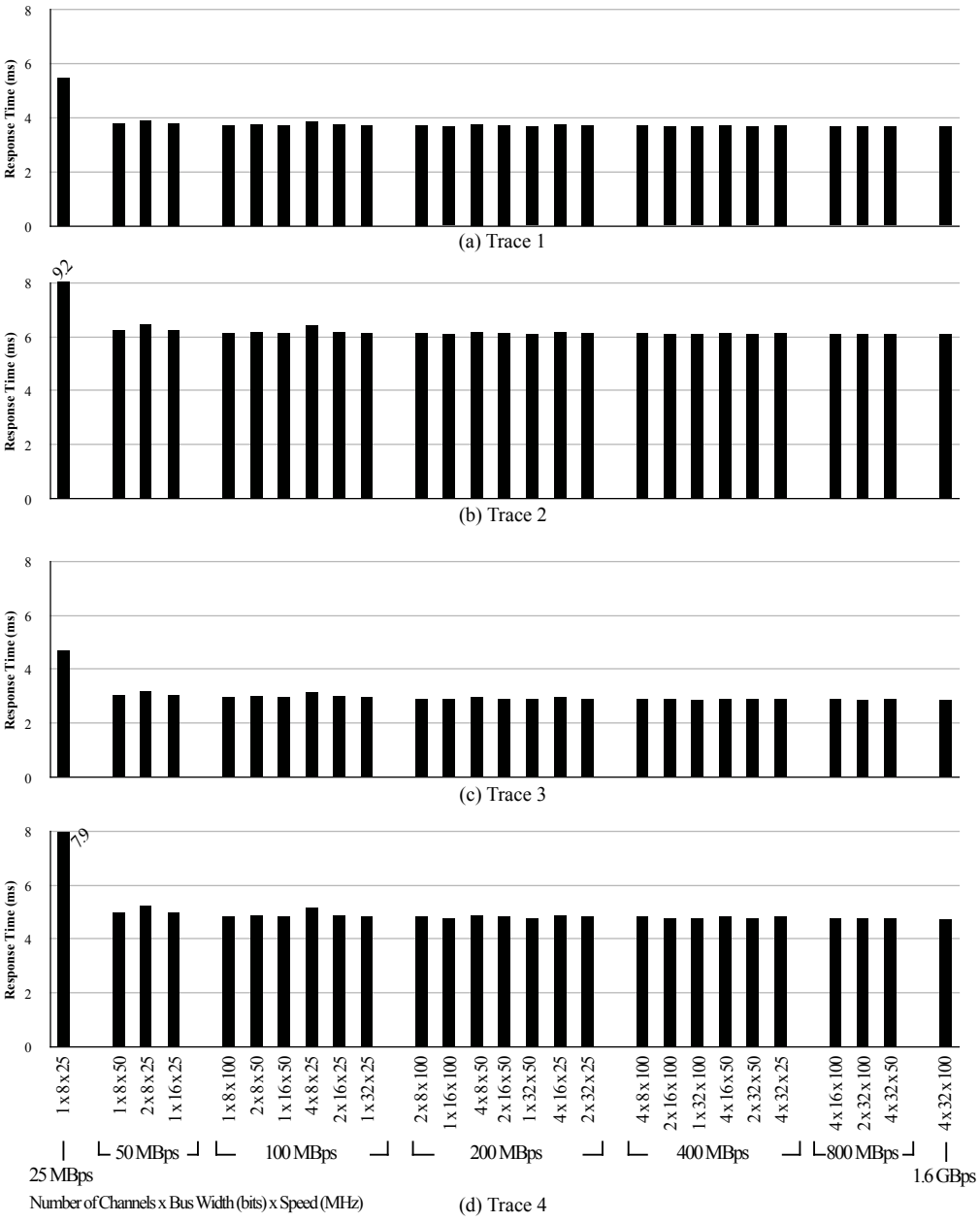


Figure 6.30: Writes - System Bandwidth. Average write request service time in milliseconds with changing system bandwidth for user workloads 1-4 is shown. So long as the total number of memory banks is constant, write performance does not change by connecting them differently.

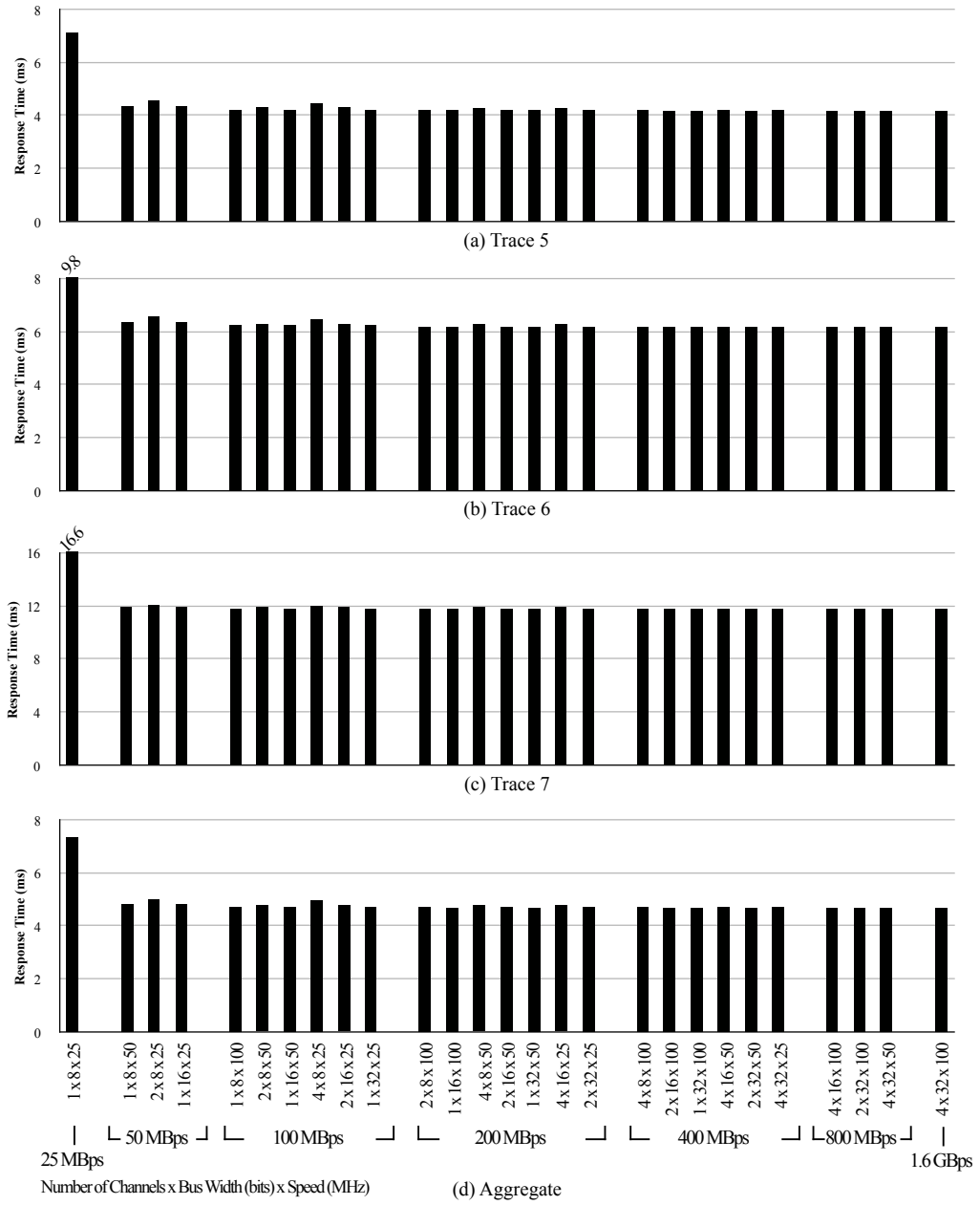


Figure 6.31: Writes - System Bandwidth. Average write request service time in milliseconds with changing system bandwidth for user workloads 5-7 is shown. So long as the total number of memory banks is constant, write performance does not change by connecting them differently.

No of Channels x Bus Width x Bus Speed	All requests	Reads	Writes
4 x 8 x 25 (100 MBps)	3.35	1.75	4.89
1 x 32 x 25 (100 MBps)	2.67	0.58	4.68

Table 6.5: 50 and 100 MBps Total I/O Bandwidth. Average request service time in milliseconds using a various interconnect configurations. Bus widths are in bits and bus speeds are in MHz.

One of the first observations is that overall performance does not change much beyond 200 MBps. In other words, if 4 flash memory banks are provided with the total bandwidth of more than 200 MBps, it does not matter how these memory banks are connected - single wide bus, dedicated bus, or multiple shared buses. This is expected as explained earlier. For typical PC user workloads, the real limitation to NAND flash memory performance is its core interface. As long as the system is provided with sufficient bandwidth, the performance will not improve any further unless more flash memory banks are added to the storage system.

For low bandwidth configurations - 25, 50 and 100 MBps - overall request service time changes within 15-20% if memory banks are connected differently. For some workloads, such as user trace 6, change in overall request times are even greater, 30-40%. For example; rather than connecting 4 memory banks with 4 dedicated channels (4 channels x 8-bit x 25 MHz), one can achieve better performance if 4 I/O channels are ganged together to form a single wide I/O bus (1 channel x 32 bit x 25 MHz) - summarized in table 6.5. The cost of doing so may not be very high since the total number of pins on the flash controller side will be the same and only the number of pins

on flash memory banks need to be increased. Load on the bus will be higher, which can slightly increase power consumption. Each memory bank now has to drive 32 pins rather than 8 pins. However, cost due to pin count increase may be eliminated if memory banks are tied together using superblocking as explained earlier. This way pin counts on each flash memory bank may be kept constant at 8. Another possibility would be using a narrower but faster bus - instead of using 32 bit bus at 25 MHz, using a 100 MHz 8-bit bus. Of course a faster bus will consume more power, but the options are available to choose between cost increase due to pin count and cost increase due to power consumption.

For read requests performance improvements are even higher. For example; rather than connecting 4 memory banks with 4 dedicated channels (4 channels x 8-bit x 25 MHz), one can cut read request times by 3 if 4 I/O channels are replaced by a single fast narrow I/O bus. Keep in mind that if more than 4 memory banks share a single I/O bus, read performance starts to deteriorate due to bus contention - as explained in section 6.1. This provides a challenge for system designers. On the one hand, overall system performance tracks average read response time and reads prefer faster channels. If the total system bandwidth is constant as a design requirement, then it is better if memory banks are connected using a shared, fast and narrow bus. But loading a single bus with more than 4 banks has a negative impact on read performance.

Write performance is heavily dependent on the number of available memory banks as explained earlier. As long as the total number of memory banks is constant, write performance is not expected to improve by connecting them differently. Figures

6.30-31 show that write performance does not change since the total number of memory banks is kept constant at 4.

6.6. Improving I/O Access

In chapter 6.1, we have showed that by increasing the level of banking and request interleaving one can improve performance significantly. However, one of the observations made was that as more banks are attached to a single I/O bus, read performance does not improve as much as write performance. This was due to an increase in the physical access times at high levels of banking. For low bandwidth shared I/O bus configurations, as more banks are connected I/O bus utilization increases to the point of traffic congestion. Delays in acquiring I/O bus in a shared bus configuration increases physical access times. Since read request timing mostly consists of time spent in reading data from flash memory via I/O bus; any congestion in the I/O bus impacts reads more than writes.

To improve I/O bus utilization, we suggest two device level optimizations. The optimizations target reducing delays in acquiring I/O bus for reads. As we mentioned earlier, performance of read requests is critical since overall system performance tracks the disk's average read response time [40].

When the timing of read and write requests is analyzed, shown in figure 3.7 in chapter 3, it is seen that data is transferred over I/O bus in bursts of 2 KB. This is due to data/cache register sizes being the same as page size in flash memory. Typical page size is 2 KB, as well as the size of data and cache registers. Therefore flash controller can only transfer data from/to a single flash memory bank in bursts of 2 KB. In a shared bus

configuration, flash memory bank has to acquire access to the bus for every 2 KB burst. Of course this assumes that bus is released any time it is not needed, which is typical in true resource sharing.

One way to reduce congestion in I/O bus is to reduce the number of bus access requests. If data can be transferred from/to flash memory array in bursts of larger than 2 KB, then there will be less requests for bus access. Once I/O bus is acquired, more data can be transferred before it is released. By increasing cache register size to more than 2 KB, one can increase data burst size. Figure 6.32 shows memory bank configurations with 2 KB cache register size and 8 KB cache register size. Timing of a sample 8 KB write request is also shown to display the change in I/O access. When cache size is 2 KB, data is written into the cache register in chunks of 2 KB and access to I/O bus is requested for each transfer - 4 I/O bus access requests for 8 KB write request. If the cache register size is increased to 8 KB - 4 pages of 2 KB each - then all data for the sample write request can be transferred in one burst. Access to I/O bus is requested only one time. Note than programming time does not change because data has to be written into flash memory one page at a time through the data register. The cache register only buffers data in advance as its name suggests. Moreover, read request timing will not change either. Even when the cache register size is increased, read requests will transfer data in bursts of 2 KB as flash memory reads data one page at a time. For read requests it is possible to buffer all pages of a request and transfer them over I/O bus in a single burst. However, holding data in the cache register until all pages are read from the memory

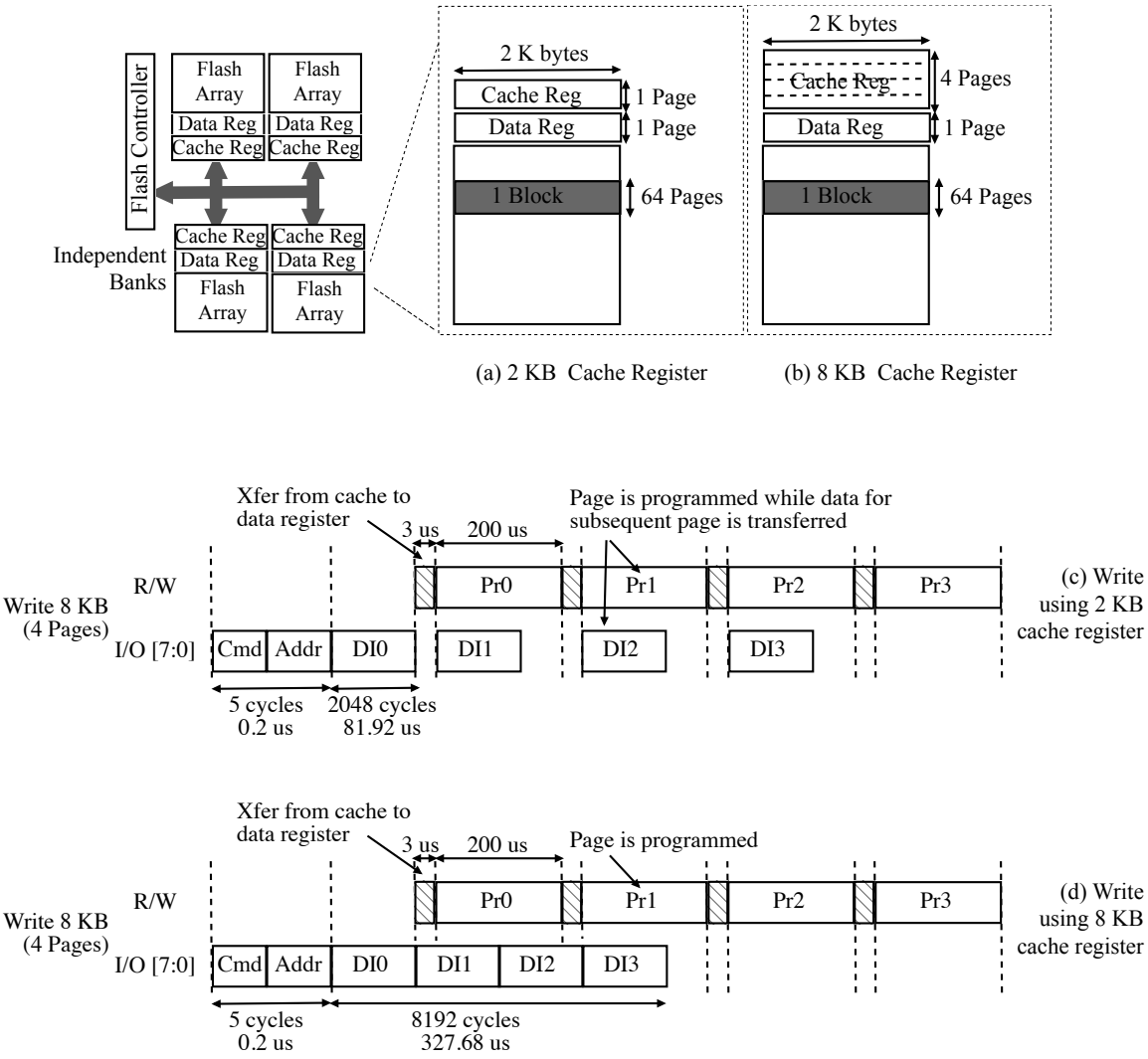


Figure 6.32: Increasing Cache Register Size. Memory bank configuration with 2KB and 8KB cache register sizes and timing of an 8 KB write request.

array will have a negative impact on the performance of reads - and on the overall system performance.

In order to gage the impact of increasing cache register size on performance, we have simulated a shared bus configuration with 2, 4, and 8 memory banks and cache register sizes of 2, 4, 8, and 16 KB. We have also increased the I/O bus speed from 25 MHz to 400 MHz for each configuration.

Increasing cache register and data burst size does not impact the performance for high speed I/O bus configurations. At high I/O bus speeds, time spent on transferring data over this channel is only a small portion. Most of the time spent for each access is physical access time - reading data from flash memory or writing data into flash memory.

For low speed I/O bus configurations, increasing cache register size does improve performance, especially for configurations when 8 memory banks share a single I/O bus. Figures 6.33-35 show an average read, write request response time, and overall response time for each configuration simulated. Depending on the workload, read performance improves 10-15% with increased cache size. The best performance is achieved when the cache register size is 4 KB. Since most of the requests in typical user workloads are 4 and 8 KB in size, a cache register matching typical request size provides the optimum design point. Although performance of write requests degrade with increasing cache register size, overall system performance will follow read request performance. The cost of increasing the cache register size should not be high either because no change in the flash interface is required.

Another optimization to improve I/O bus access is using different bus access heuristics for reads and writes. In a typical shared resource configuration, each component requests access for the resource when needed and releases it when it is not needed. However, considering the timing of a read or write request for NAND flash memory interface (figure 3.7 in chapter 3) ,data transfer is not continuous. Data is transferred from/to flash memory in bursts of 2 KB. Anytime 2 KB data is transferred from the cache register to the data register, access to I/O bus is not required and it can be released if another memory bank requests it. Only after a short period of time, memory bank releasing the bus will request it again. Figure 6.36a illustrates the timing of an 8 KB

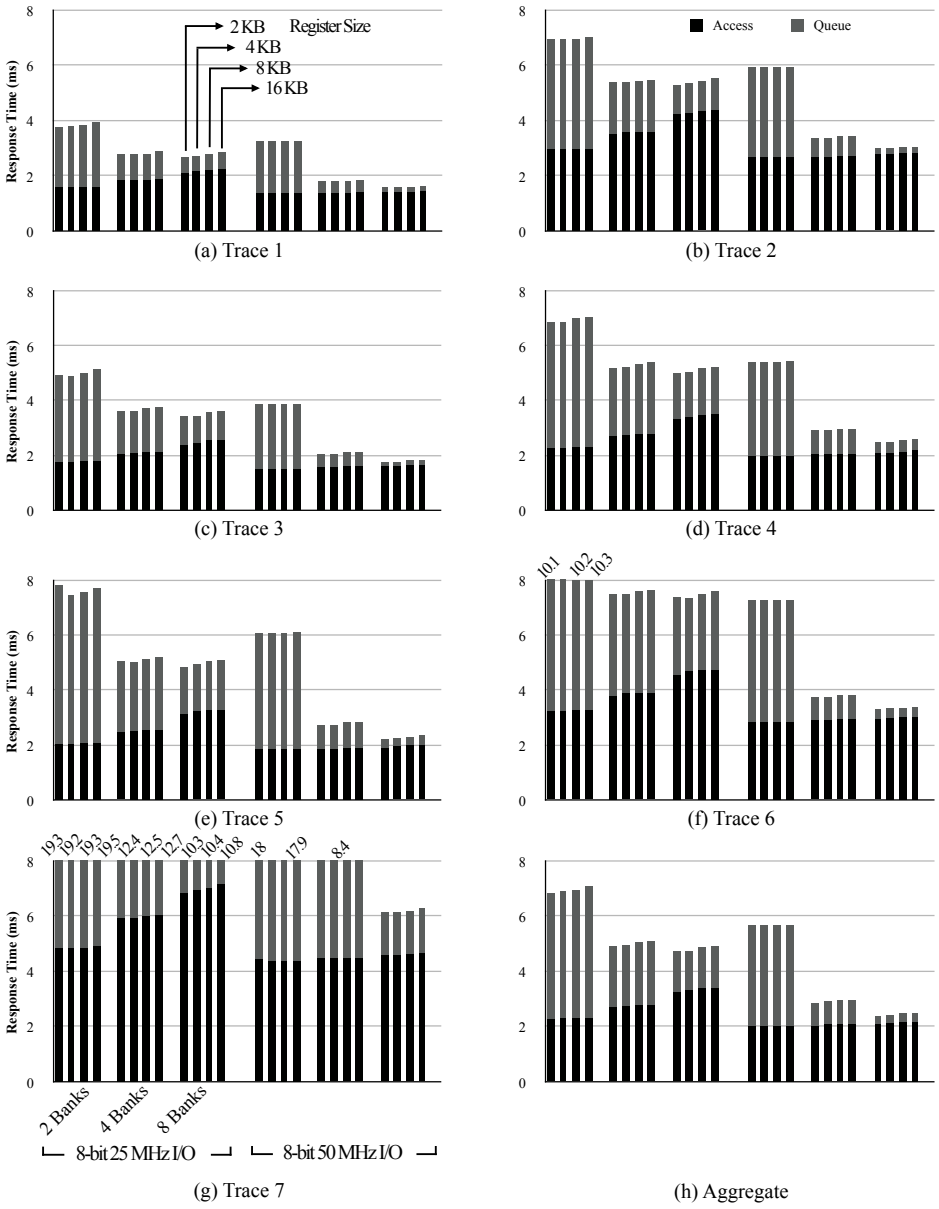


Figure 6.33: Increasing Cache Register Size. Average request service time in milliseconds with increasing cache register size.

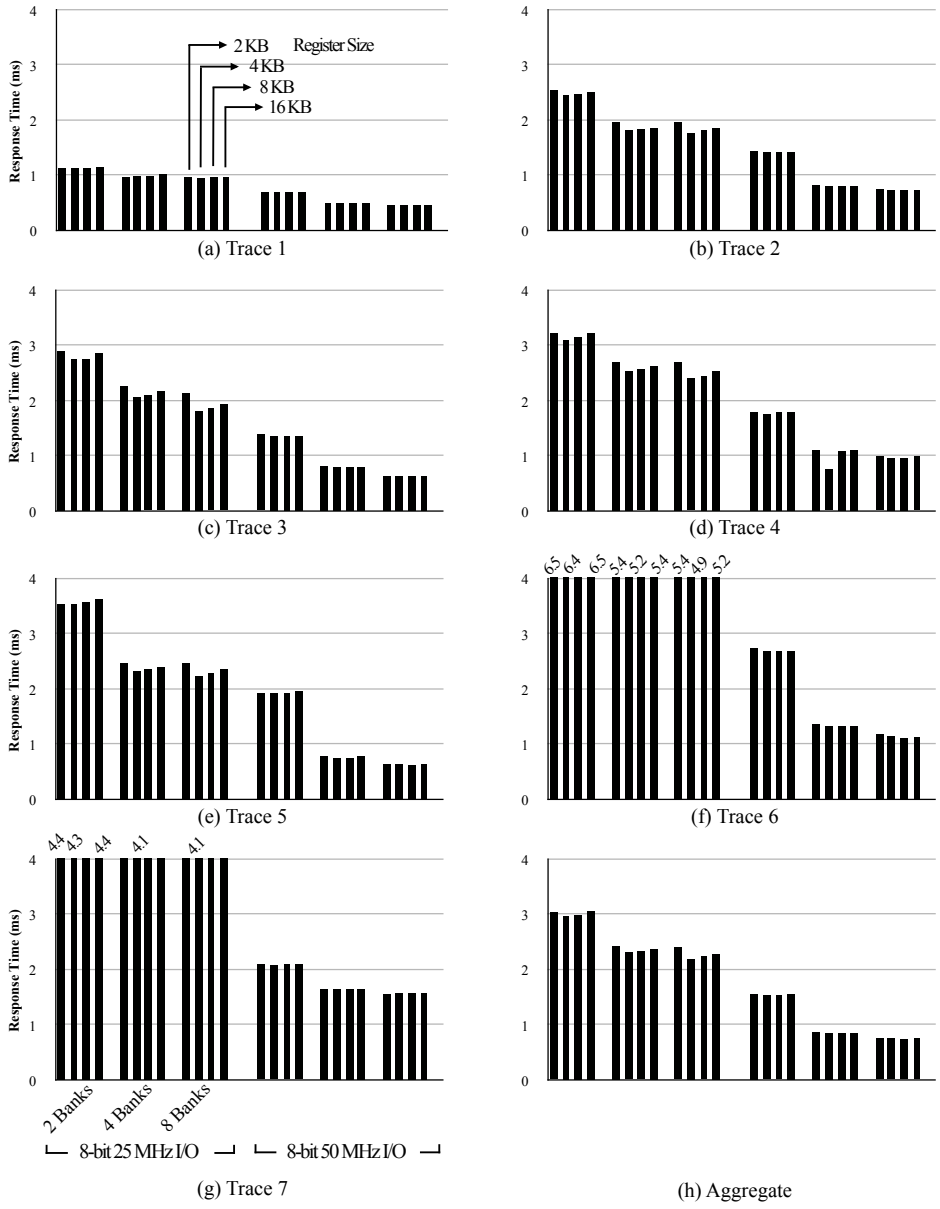


Figure 6.34: Reads with Increasing Cache Register Size. Average read request service time in milliseconds with increasing cache register size. Best performance is achieved when cache register size is 4 KB.

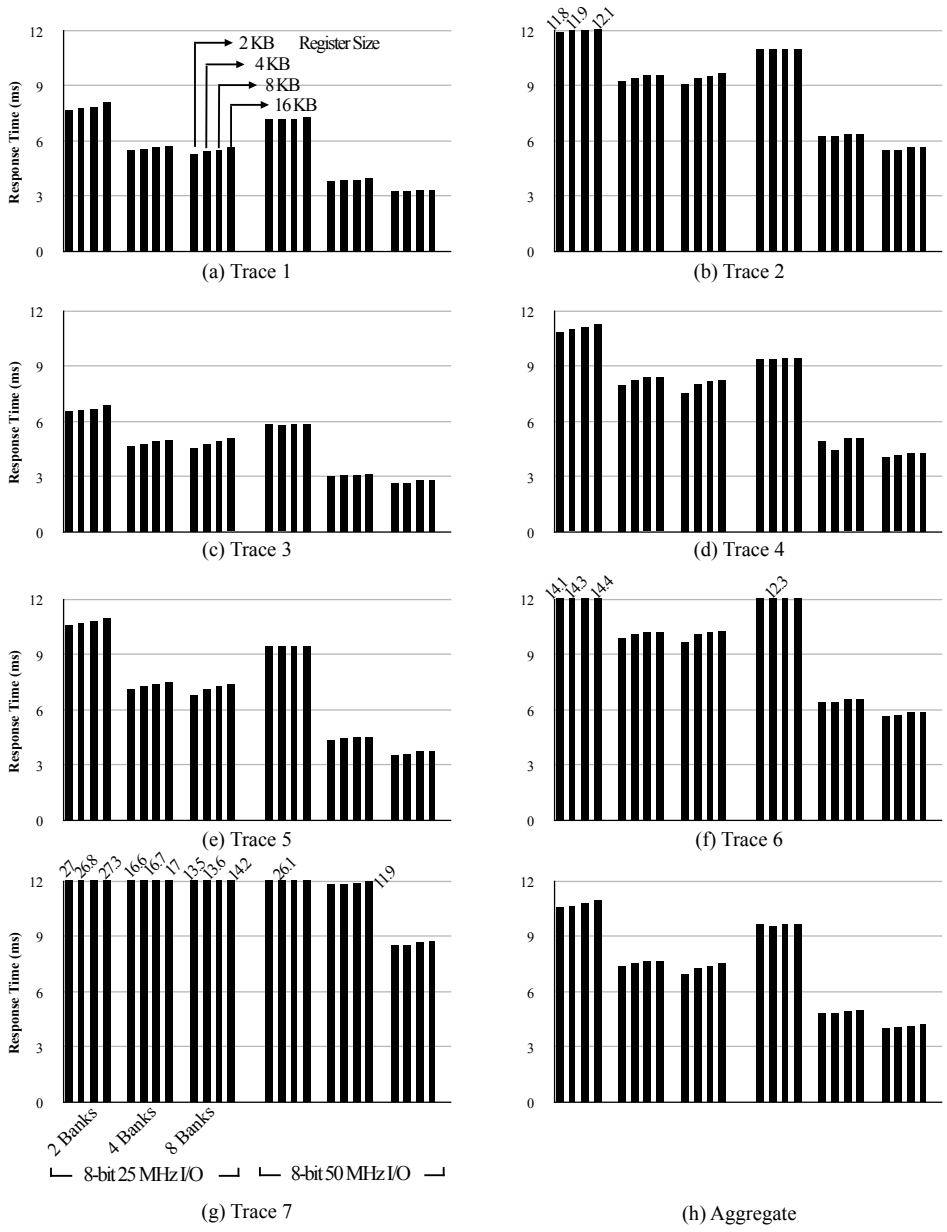
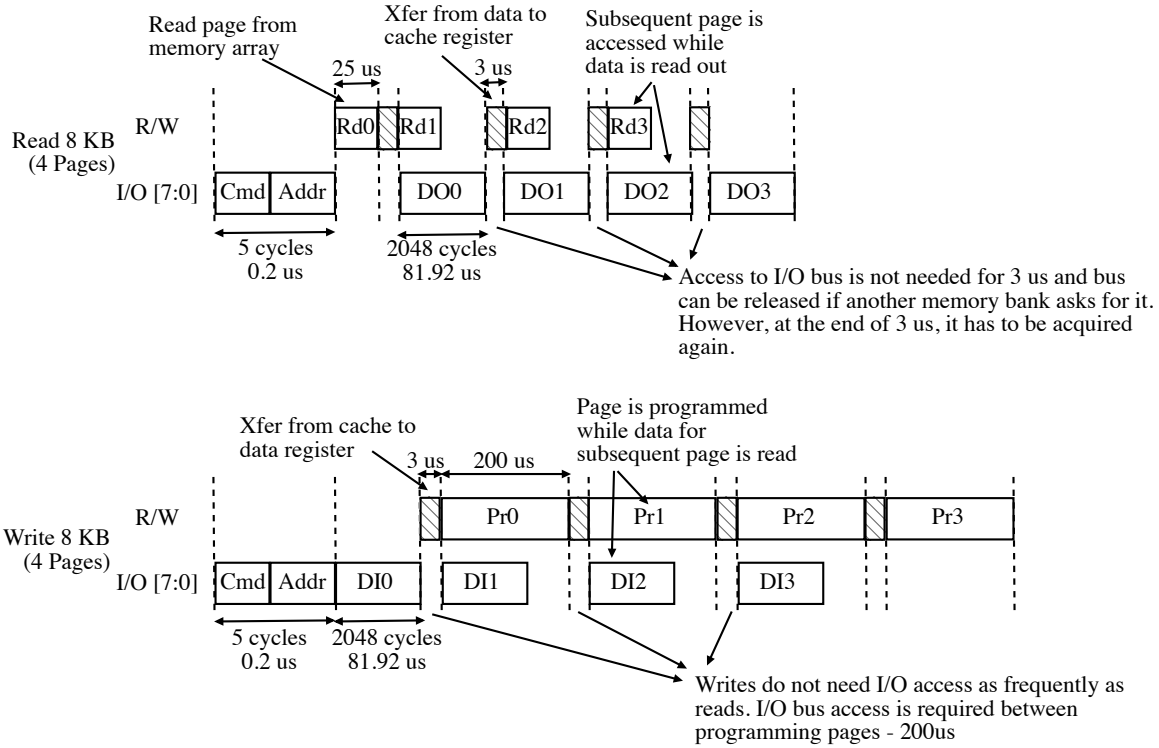
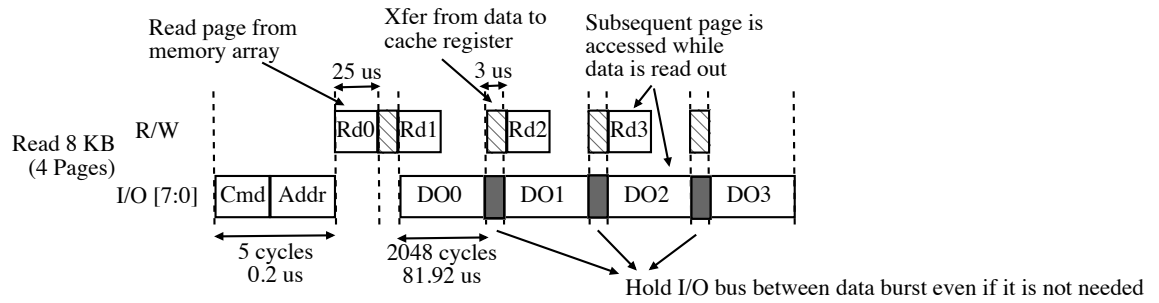


Figure 6.35: Write with Increasing Cache Register Size. Average write request service time in milliseconds with increasing cache register size.



(a) Release bus when it is not needed



(b) Reads hold bus during entire data transfer

Figure 6.36: I/O Bus Access Timing. (a) Timing of read and write requests, access to I/O bus is released when it is not needed. (b) Timing for read requests with hold bus during entire data transfer access policy.

read request. If after page 0 is transferred, another memory bank requests I/O access, bus will be released as it is not used while page 1 is copied from data to cache register. After 3 μs, access to I/O will be needed to transfer page 1. In a shared I/O bus configuration, the timing of a read request may be interrupted several times. The same is true for writes. However, as the timing of writes is dominated by page programming, access to I/O bus is not required as frequently as reads. For read requests it is better to guarantee uninterrupted I/O bus access during the entire data transfer.

We have simulated a shared bus configuration with 2, 4, and 8 memory banks and with I/O bus speeds of 25 MHz to 400 MHz. For each configuration we have tested two different I/O access policies. The yypical case simulates a true shared resource policy. I/O bus is released whenever it is not needed. In our proposed policy, I/O bus is held during the entire read data transfer time; and when servicing writes, I/O bus is released whenever it is not needed. Figures 6.37-39 show average read, write request response time and overall response time for each policy simulated.

For high speed I/O bus configurations the difference between two policies is negligible. However for slow I/O buses - 25 MHz - read performance improves 5-10% on average when the I/O access policy is “hold bus during entire read request”. Depending on user workloads and the number of memory banks sharing I/O bus, read performance can be improved up to 20% if read requests hold I/O bus during the entire data transfer. Although average disk request times and average write request times show 2-5% degradation, overall system performance will trace read request performance. The cost of implementing this policy change will not be high since no changes to flash interface is

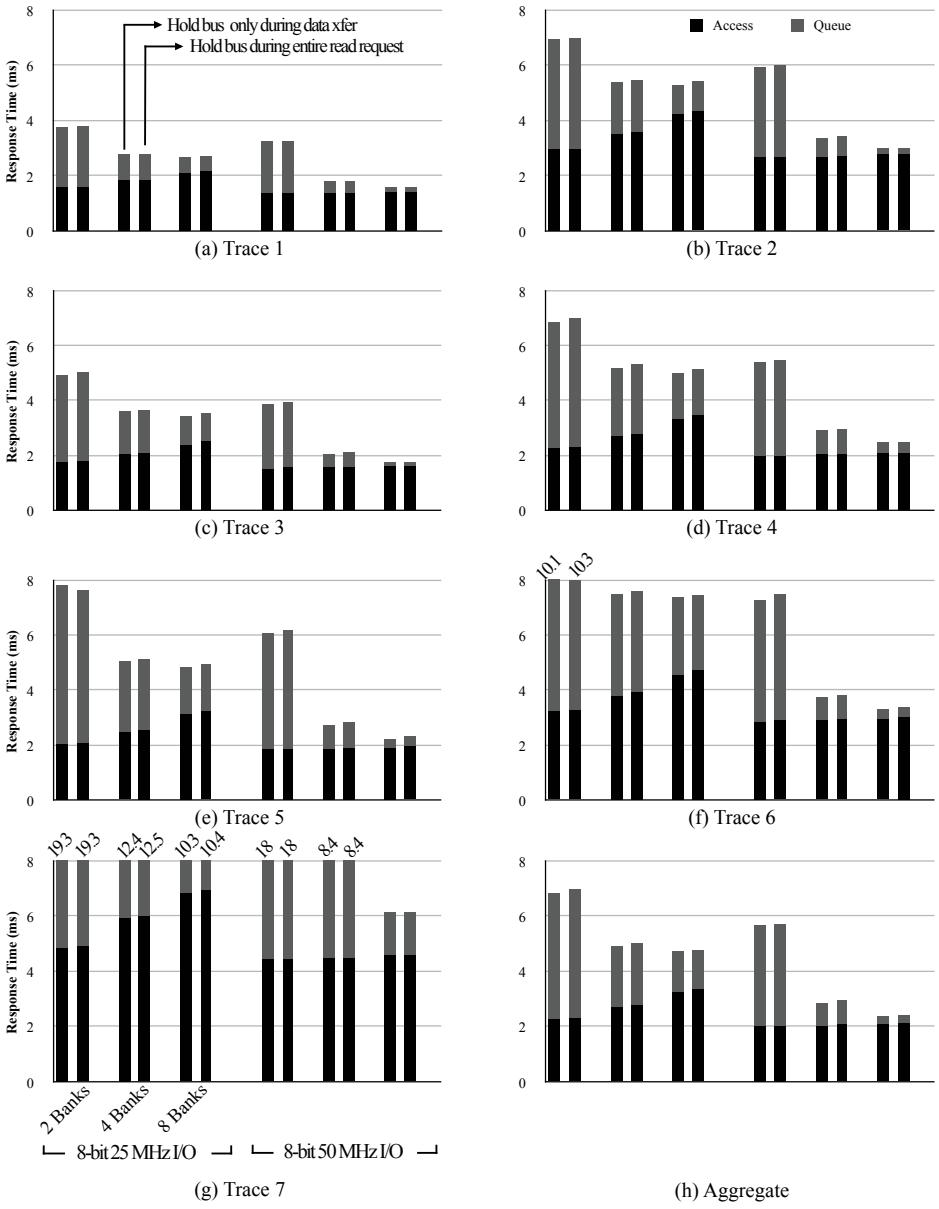


Figure6.37: Hold I/O Bus for Reads. Average request service time in milliseconds when the I/O bus access policy is "hold bus for entire transfer" for reads.

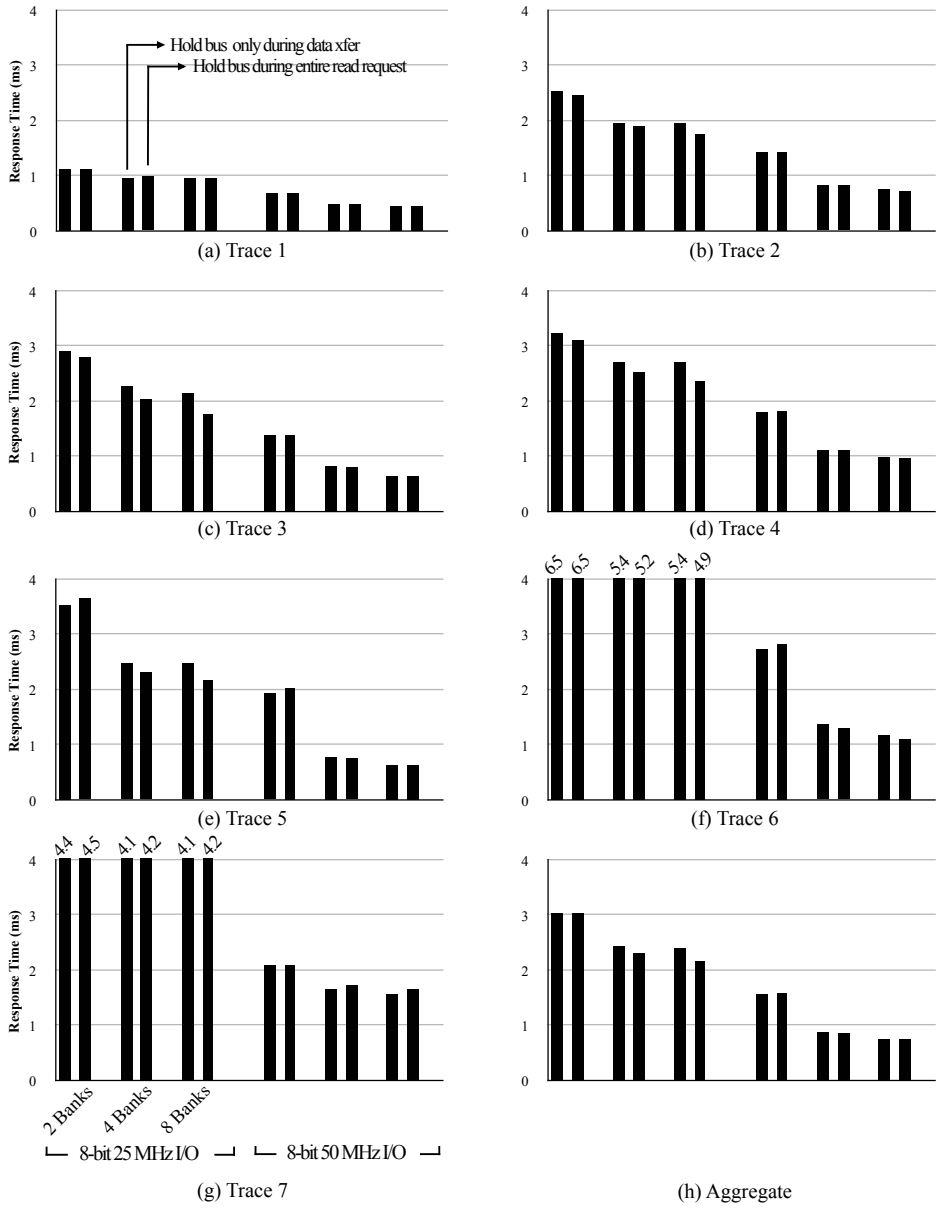


Figure6.38: Reads with Hold I/O Bus for Reads. Average read request service time in milliseconds when the I/O bus access policy is "hold bus for entire transfer" for reads. Read performance can be improved up to 20% if read requests hold I/O bus during data transfer.

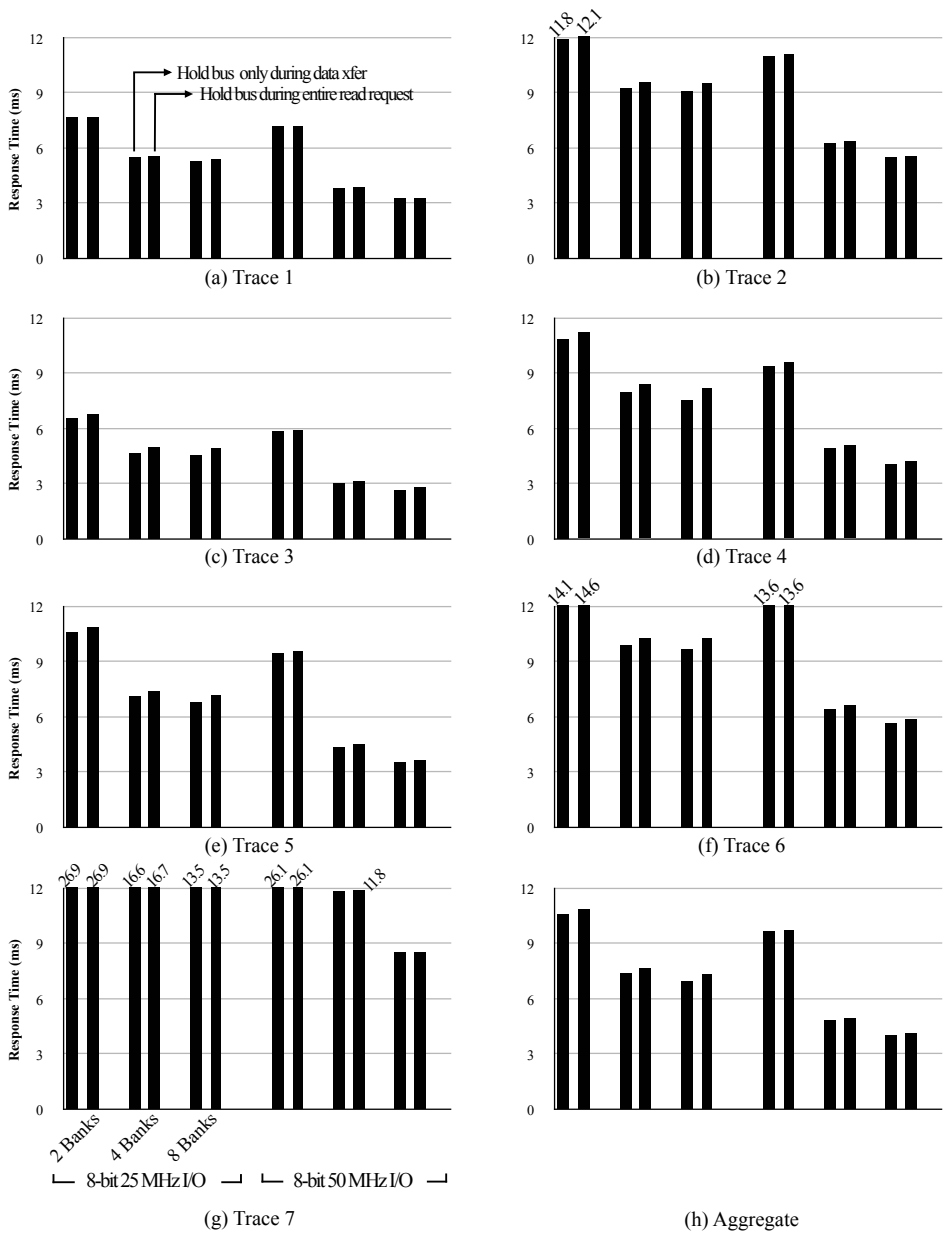


Figure6.39: Writes with Hold I/O Bus for Reads. Average write request service time in milliseconds when the I/O bus access policy is "hold bus for entire transfer" for reads.

required.

6.7. Request Scheduling

One of the classic problems with disk drives have been command queuing and scheduling. In a typical storage system, the host sends a batch of I/O requests to the disk drive and waits for them to be serviced. One way to service these requests is using a first come first serve policy (FCFS). However, if the disk drive is given flexibility to decide a more efficient order to serve these requests, performance of the storage system can be improved. This has been the rationale behind various queuing and request reordering policies implemented in today's conventional hard disk drives.

In conventional hard disks, majority of the time spent in serving an I/O request is seek time - time to move read/write head from its current location to the target location. Therefore, majority of disk scheduling algorithms are based on reducing seek time. Among I/O requests in the queue, ones whose target location is closer to read/write heads current location are served first. One of the most common disk algorithms based on this principle is Shortest Seek Time First (SSTF). Since SSTF algorithm is a greedy algorithm, it has a tendency of starvation. Another algorithm that is based on minimizing seek time is LOOK (aka Elevator Seek, SCAN) algorithm. In this algorithm, read/write head moves from one end of the disk to the other end and serves requests along the way. Both of these algorithms were developed when seek time was the dominant component of the total I/O request service time. Modern disk drives have improved significantly in their seek times. Although seek time is still a big component of the total request service time, physical access time is currently considered as the main component of servicing an I/O

request. Physical access time includes both seek time and rotational latency. Thus scheduling algorithms based on shortest access time are common among modern disk drives. Shortest Access Time First (SATF) is one such algorithm where the request with the minimum access time is serviced first.

Different than conventional hard disk drives, flash memory solid-state disks do not have any mechanical components as mentioned before. Therefore, there is no concept of seek time or rotation latency for flash memory disks. For solid-state disks, one cannot differentiate between different I/O requests in queue by comparing their address information. Accessing any location within flash memory has the same cost of loading a page. On the other hand, for flash solid-state disks the dominant factor in overall request service time is time spent in NAND flash interface as mentioned before. Moreover, flash interface timing is very different between reads and writes. There is a scale difference between reading a page from the flash memory array, 25 μs, and writing a page to the flash memory, 200 μs. We have also observed in all our previous results that; there is a big difference between average read times and average write times. For NAND flash solid-state disks, two factors are dominant in the overall request service time: request type (read or write) and request size. If a request requires reading or writing several pages, its service time will be proportional to the number of pages accessed.

In order to serve I/O requests more efficiently and improve solid-state disk performance, we suggest 4 different request scheduling policies designed for flash memory. All 4 of these request schedules are based on the fundamental idea of “servicing the request which will take the shortest time first”.

Schedule 1 - Read Priority (RP): This request scheduling policy is based on two facts. First; there is a big scale difference between read times and writes times for flash memory. Typically reads will almost always take a shorter time to serve than writes - for a typical 4K and 8K request. Second; overall system performance tracks disk's average read response time [40]. In this policy, read requests are always serviced before any write request. In other words, reads have a higher priority over writes.

Schedule 2 - Shortest Request First, First Come First Serve on Ties (SRF - FCT): This request scheduling policy is based on the fact that NAND flash interface is the dominant factor in request response time. A request which reads or writes the least number of pages (shortest request) will be serviced first. On the other hand, typical user workloads consist of 4K or 8K requests. Request sizes do not show high variation. Therefore, there will be plenty of requests in queue which are of identical size. In such tie situations, FCFS policy is observed.

Schedule 3 - Shortest Request First, Read Priority on Ties (SRF - RPT): This request scheduling policy is same as schedule 2 but uses read priority algorithm on ties. If there are two requests of equal size but different types, reads have a higher priority over writes.

Schedule 4 - Weighted Shortest Request First (WSRF): This scheduling policy is a variation of schedule 2. In addition to request size, it also factors in the flash interface timing difference between reads and writes. For flash memory, typical page read time is 25 μs and typical page program time is 200 μs. There is an 8x scale difference between reads and writes. In this schedule, request sizes are weighted according to request type.

The sizes of write requests are multiplied by 8 and then compared. For example, between 8K read and 4K write request read request will be serviced first. A 4K write request will be considered equivalent to a 32K read request.

In order to gage the performance impact of these request scheduling algorithms, we have simulated these algorithms using a single bank flash memory configuration. Although multiple bank configurations improve performance significantly, using a single bank will generate long enough request queues, which will in turn make it easier to measure the performance impact of these request scheduling algorithms. Also I/O bus speed has a limited impact on read performance as explained earlier. Therefore, we have also increased I/O bus speed from 25 MHz to 400 MHz to incorporate I/O bus speed as a factor. Figures 6.40-42 show average read, write request response time and overall response time for each policy simulated. We have assumed FCFS scheduling policy as the baseline.

When reads are given priority over writes, their performance improves significantly, by 60-80%. This performance improvement on reads only comes at a small cost to writes - a write performance degradation of roughly 5%. Although overall request time improves by 15-20%, user perceived performance will track bigger improvements on read time. Another observation is that one can achieve read performance of 2 or 4 banks by using a single bank with read priority as a scheduling algorithm. In designs where increasing the numbers memory banks is costly either due to increased pin count or power consumption, implementing read priority at the driver level can be a cost effective solution to providing similar read performance.

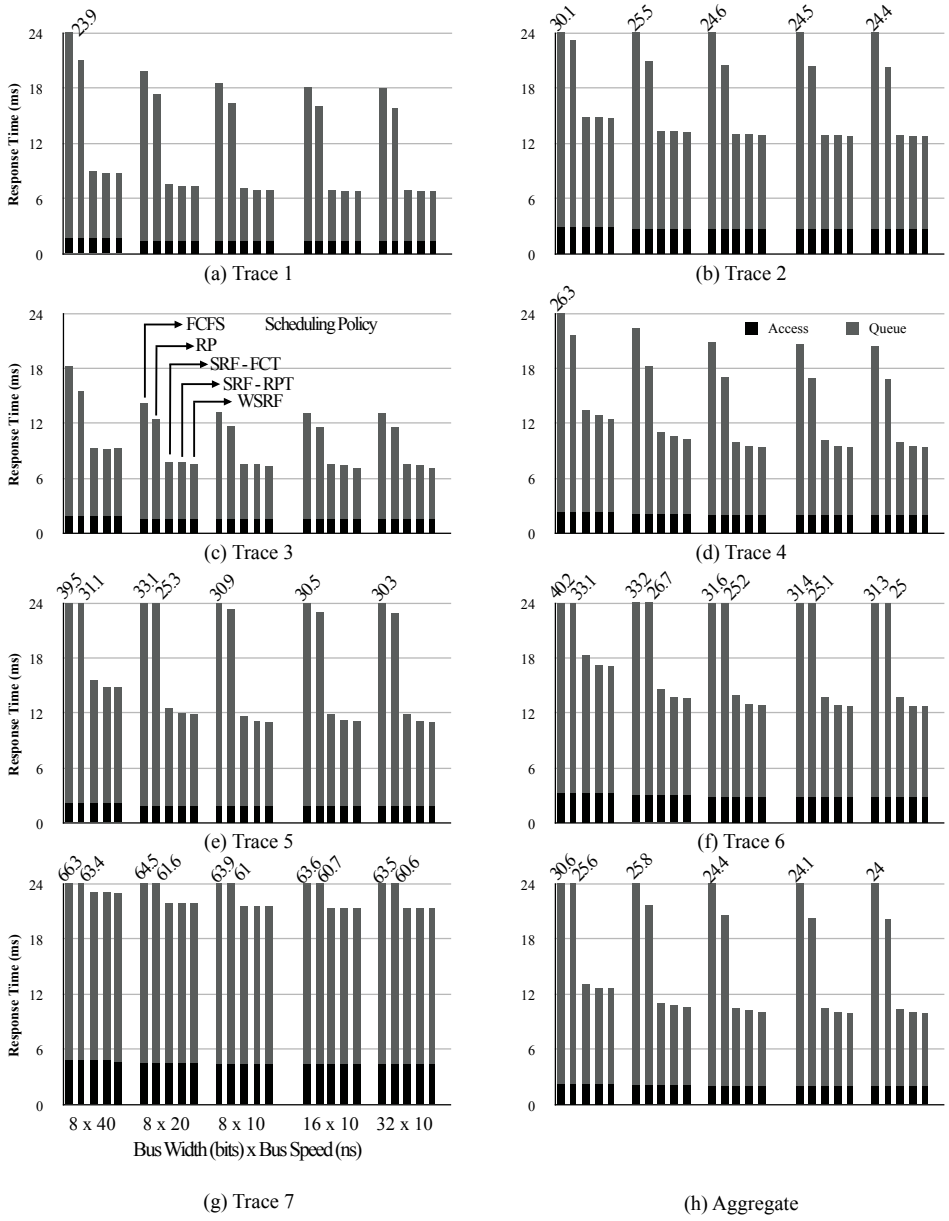


Figure 6.40: Request Scheduling. Average request service time in milliseconds with different request scheduling policies.

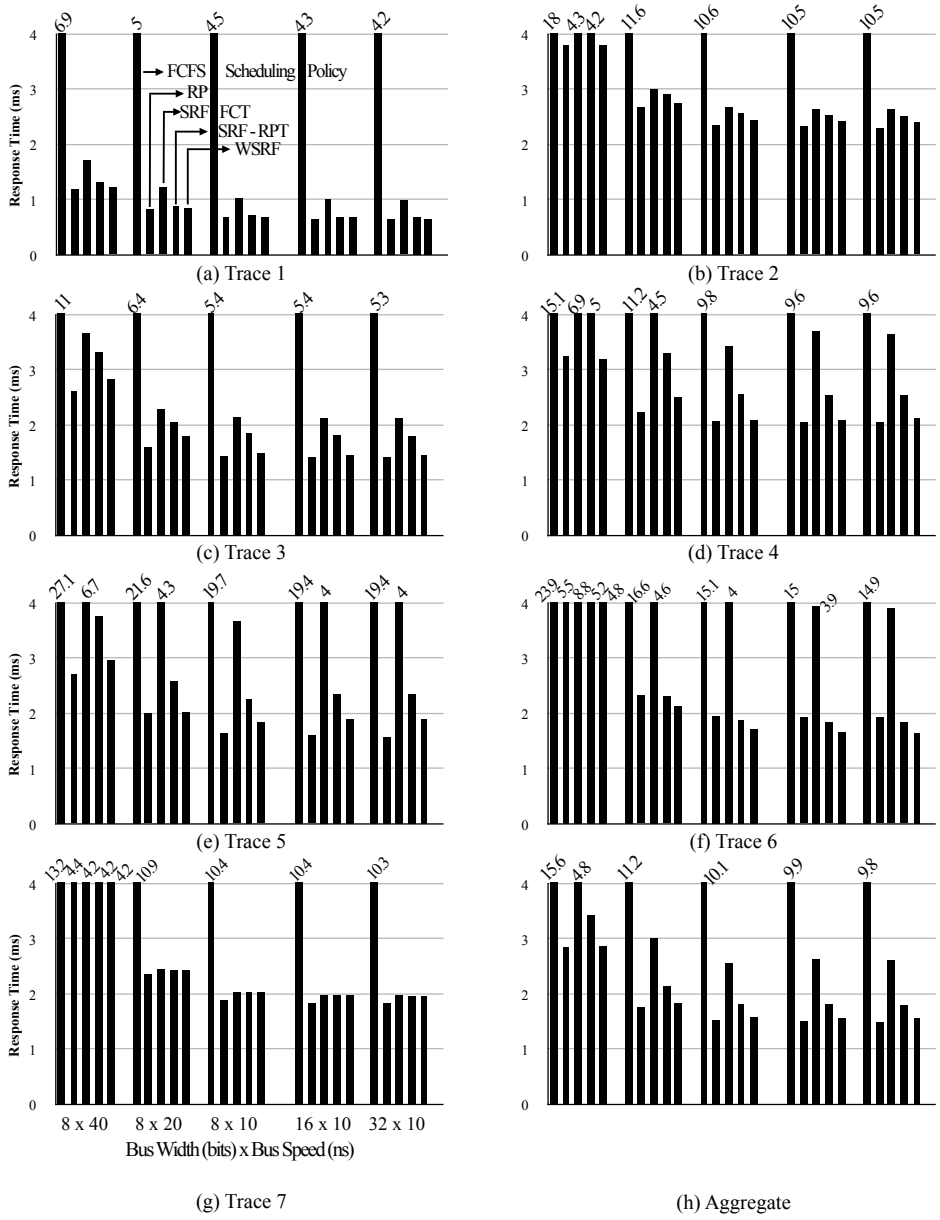


Figure 6.41: Reads - Request Scheduling. Average read request service time in milliseconds with different request scheduling policies. Performance with WSRF scheduling algorithm is very comparable to performance with RP.

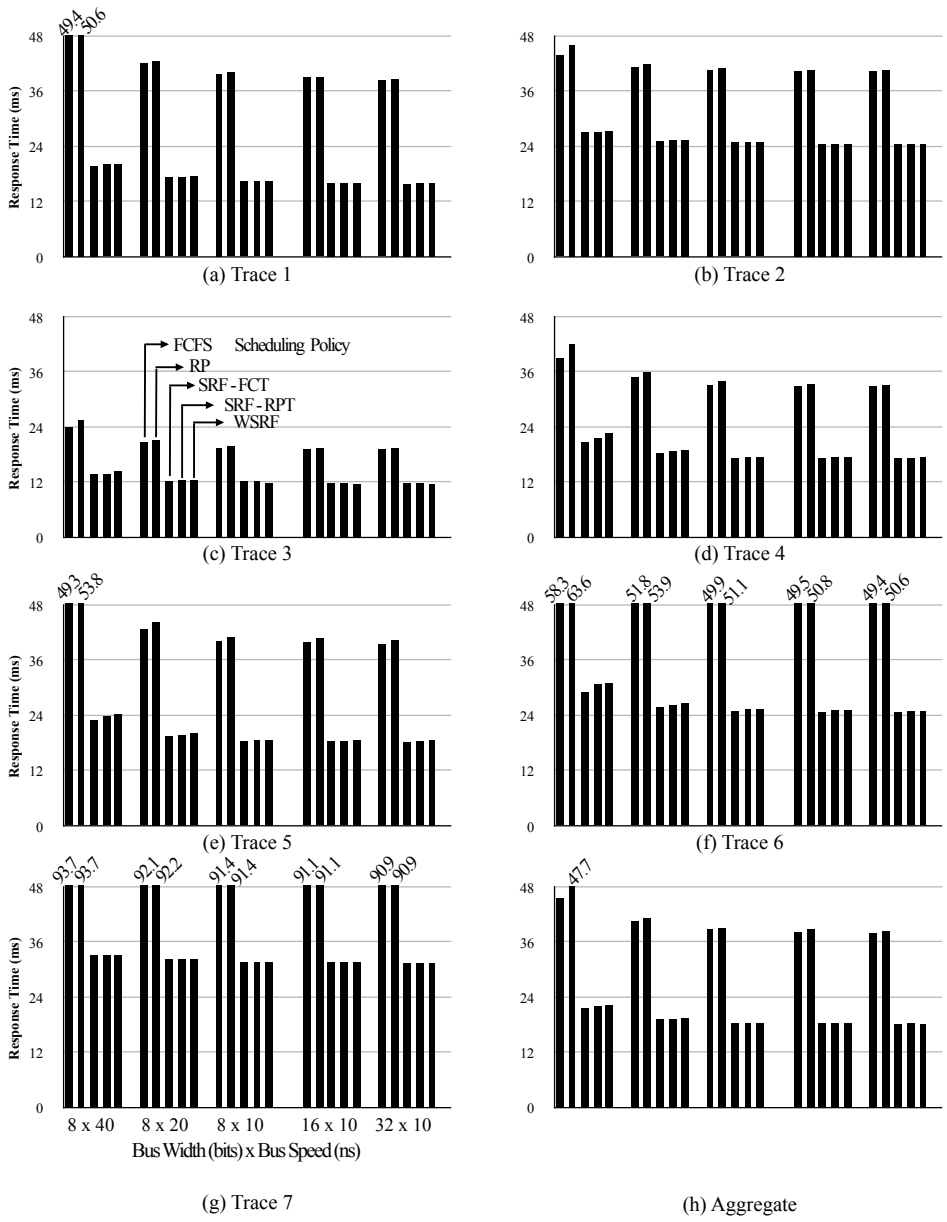


Figure 6.42: Writes - Request Scheduling for Writes. Average write request service time in milliseconds with different request scheduling policies. Writes benefit from using some form of the shortest request first algorithm.

When some form of shortest request first scheduling algorithm is used, performance improvements can be even bigger. Average request time improves 50-60% when the shortest requests are serviced first. Moreover, both reads and writes benefit from using these scheduling policies. Although SRF-FCT and SRF-RPT algorithms do not perform as well as RP if reads are considered, WSRF algorithm provides a great balance between reads and writes. Its read performance is almost as good as RP, at the same time it also improves writes significantly.

In addition to a single bank flash memory configuration, we have also simulated a 4 channel, 16 banks (4 banks per I/O channel) flash memory configuration. Due to higher concurrency, queue times in this configuration were less than 1 ms and not one of these scheduling algorithms had an impact. It is observed that, for any configuration and user workload where queue times are less than 1-2 ms, scheduling policies become obsolete. On the contrary, if server workloads are used rather than PC user workloads, one can expect even higher improvements using these algorithms. Server workloads will be much heavier than typical user workloads and will result in a longer queue length. Thus a greater possibility of request re-ordering.

One of the side effects of any greedy request scheduling algorithm is that some I/O requests may end up waiting in queue for an unduly long time. This is often referred to as starvation. For the request scheduling algorithms we have proposed here, very large write requests are prone to starvation.

Table 6.6 shows queue time distribution for user workload 1. This workload can be considered as read heavy, as its read to write ratio is 60:40. It is slightly biased

Queue Statistics	FCFS	RP	SRF-FCT	SRF-RPT	WSRF
Avg. Queue Time (ms)	22.39	19.49	7.29	7.15	7.13
Max. Queue Length	544	541	120	120	118
Queue Time Distribution					
(0, 1)	24709	25342	29504	29960	29571
[1, 10)	13016	13380	14626	14054	14620
[10, 100)	9491	9116	6393	6332	6334
[100, 1000)	4731	3875	1001	966	954
[1000, Max]	33	179	104	118	121
Max. Queue Time (ms)	1086.77	1219.58	3322.02	3322.02	3333.36

Table 6.6: User Workload 1 (Read Biased) Queue Statistics. Queue wait time distribution for user workload 1 (read biased workload).

towards reads as requests in user workloads generally are partitioned 50:50 between reads and writes. If user workload 1 is run with a single bank, 8 bit, 25 MHz I/O configuration with FCFS scheduling policy, the average queue time for any request is 22.39 ms and the maximum queue length is 544 requests. If we exclude requests which did not experience any queue time, 47.5% of the requests spent less than 1 ms in the queue. The maximum time spent in queue is 1086.77 ms. 33 requests spent more than 1000 ms waiting in the queue. One observation is that, when proposed scheduling algorithms are used number of requests in [10, 100) and [100, 1000) buckets decreased considerably. Bucket [a, b) represents the number of requests which spent larger than a, less than b time waiting in the queue. On the other hand, the impacts of starvation can be observed as well. With any shortest request first type algorithm, the maximum queue time is increased by 3 times.

Queue Statistics	FCFS	RP	SRF-FCT	SRF-RPT	WSRF
Avg. Queue Time (ms)	61.66	58.71	18.31	18.31	18.31
Max. Queue Length	110	110	82	82	82
Queue Time Distribution					
(0, 1)	3678	3716	4669	4657	4676
[1, 10)	4547	4554	4662	4621	4658
[10, 100)	3518	3621	5090	5083	5088
[100, 1000)	3514	3374	774	778	773
[1000, Max]	18	7	7	7	7
Max. Queue Time (ms)	1320.86	1309.28	1237.68	1237.31	1237.31

Table 6.7: User Workload 7 (Write Biased) Queue Statistics. Queue wait time distribution for user workload 7 (write biased workload).

Also the number of requests in the [1000, Max] bucket also increased substantially. For example, there were only 33 requests which spent between [1000, 1086.77] ms waiting in the queue with FCFS. However, there are 121 requests which spent between [1000, 3333,36] ms in the queue with WSRF. It is important to note that these 121 requests only represent 0.12% of all requests in the workload and they are almost always are write requests.

Let's look at queue time distribution for a different workload. Table 6.7 shows queue time distribution for user workload 7. This workload can be considered as write heavy, as its read to write ratio is 35:65. When FCFS scheduling policy is used, the maximum queue time is 1320.86 ms and there are 18 requests which spent between [1000, 1320.86] ms in the queue. However, if the scheduling policy is changed, the

maximum queue time decreases and there are less requests whose queue time is larger than 1000 ms. Our simulations did not observe starvation in a write heavy workload.

With our scheduling algorithms, some kind of aging algorithm may be put in place to prevent possible starvation. If a request spends more than a pre-determined time in queue, it's priority may be increased and it may be moved to the front of the queue. As mentioned mostly writes are prone to possible starvation. From an end user perspective, a write request waiting in the queue for an extended period of time is not of much importance. Already all modern disk drives implement read hit on write requests while they are waiting in the queue. Unless there is a sudden power loss these write requests will not be lost and it does not matter from a caching perspective if these writes are indeed written to physical media or waiting on the queue. And user perceived performance is dependent on read requests. As long as the possibility of starvation on a read request is near zero, time limits on aging can be relaxed as much as possible. Or may be not implemented at all. When flash memory is used in a USB drive, the possibility of sudden power loss is a reality and the amount of disk cache is very limited or does not exists at all for cost reasons. However, flash memory solid-state disks operate within the host system and sudden power loss is not much of a concern. They are equipped with a disk cache component much like conventional hard disk drives.

In addition to these various scheduling algorithms, we have also implemented modular striping as default write scheduling algorithm in our simulations. If there are a total of x memory banks, the N^{th} write request is assigned to bank number $N \pmod{x}$ by default. Flash memory has a relatively long write (program) latency but flash memory

array also consists of multiple banks. Since we allocate a newly erased page for each write request, choosing a memory bank for each write and an empty page within that bank becomes a run-time decision of resource allocation. By distributing sequential writes among flash memory banks, we can hide write (program) latency. Striping is a typical RAID 0 based scheduling policy used extensively in the disk community to distribute data across several disks for better performance. Therefore, we have assumed modular striping for write requests as a logical choice to utilize available parallelism within flash memory array. The simplicity of this approach also provides a proficient design choice as it adds almost no additional complexity to flash controller.

Table 6.8 and 6.9 illustrate total number of pages read from and written into each memory bank in a 16 bank flash memory array configuration for each user workload. Modular striping generates an equal number of write requests to each bank, although request sizes can be different. For example, the first request can be an 8 KB write request and will be routed to bank 1. The second request can be 4 KB and will be sent to bank 2. Even though each bank serves 1 write request in this case, write latencies will be different and the total amount of sectors written will not distribute evenly.

In our simulations with real user workloads we have observed that modular striping of writes not only generate equal number of write requests per bank, it also distributes write sectors almost evenly among flash memory banks. For a flash array of 16 banks, one would expect each bank to serve 6.25% of the overall sectors written for perfect uniform distribution. As seen in Table 6.8, the number of sectors written into each bank is within 5% of each other - meaning the busiest bank serves 6.56% of the total

	User Workloads							
Bank #	1	2	3	4	5	6	7	Aggregate
1	5.65%	6.39%	6.83%	6.40%	6.30%	6.51%	6.28%	6.34%
2	6.09%	6.37%	6.96%	6.79%	6.70%	6.30%	6.34%	6.52%
3	6.59%	5.77%	6.78%	6.56%	6.26%	6.06%	6.41%	6.36%
4	6.79%	6.47%	6.71%	5.95%	6.58%	6.75%	6.22%	6.53%
5	6.32%	6.72%	6.06%	6.13%	6.36%	6.21%	6.97%	6.37%
6	6.64%	6.50%	6.46%	5.74%	5.81%	5.96%	7%	6.29%
7	6.61%	6.78%	6.42%	5.71%	6.28%	6.00%	6.43%	6.33%
8	6.01%	7.09%	6.01%	6.08%	6.18%	6.31%	7.05%	6.35%
9	5.94%	6.10%	5.92%	6.01%	5.86%	6.53%	7.55%	6.22%
10	6.31%	5.63%	6.17%	6.62%	6.13%	6.55%	6.68%	6.28%
11	6.57%	6.24%	6.01%	6.72%	6.31%	5.33%	5.91%	6.16%
12	6.12%	5.74%	5.58%	6.14%	5.83%	6.15%	5.44%	5.86%
13	6.06%	5.25%	6.18%	5.99%	5.73%	6.39%	5.63%	5.90%
14	5.87%	6.09%	5.82%	6.31%	6.43%	6.35%	5.48%	6.06%
15	6.02%	6.64%	6.32%	6.35%	6.59%	6.38%	4.98%	6.22%
16	6.41%	6.20%	5.77%	6.48%	6.64%	6.38%	5.64%	6.23%

Table 6.8: Percentage of Write Sectors per Bank. Ideally each memory bank should serve equal number of write requests. The percentage of write sectors should be 6.25% for a 16 bank configuration.

write sectors and the least busy bank serves 5.95% of the total write sectors. These values slightly vary among workloads with largest variation in user workload 7. In this workload, the number of sectors written into each bank is within 20% of each other. This shows that modular striping of write requests not only generates an equal number of write requests per bank, but it also generates an almost equal number of total sector writes for

	User Workloads							
Bank #	1	2	3	4	5	6	7	Aggregate
1	6.19%	5.24%	10.64%	11.04%	15.78%	6.68%	5.92%	9.01%
2	2.45%	4.34%	5.66%	3.60%	4.34%	11.16%	3.87%	5.14%
3	25.13%	9.41%	8.42%	7.64%	7.63%	17.64%	3.95%	12.71%
4	6.01%	8.61%	8.13%	3.28%	7.58%	7.14%	3.10%	6.50%
5	3.98%	1.89%	4.40%	3.93%	2.81%	6.62%	3.46%	4.01%
6	3.25%	1.31%	1.93%	1.15%	1.64%	1.30%	2.94%	1.95%
7	6.75%	2.59%	3.88%	1.73%	3.57%	1.40%	3.12%	3.52%
8	5.95%	2.27%	3.61%	8.75%	5.20%	2.23%	7.30%	4.91%
9	5.85%	5.74%	4.91%	8.83%	5.91%	5.29%	31.92%	7.99%
10	2.91%	28.02%	15.94%	9.65%	7.38%	5.33%	8.44%	10.34%
11	12.03%	16.57%	21.93%	31.51%	25.15%	14.64%	7.66%	18.90%
12	6.83%	4.32%	2.01%	1.05%	1.89%	13.35%	3.40%	4.96%
13	1.82%	1.76%	1.39%	0.91%	1.79%	1.12%	4.45%	1.70%
14	1.66%	1.40%	1.29%	1.17%	2.16%	1.02%	4.13%	1.66%
15	4.55%	4.02%	3.87%	4.39%	4.87%	3.72%	2.47%	4.11%
16	4.66%	2.53%	1.99%	1.38%	2.29%	1.37%	3.88%	2.59%

Table 6.9: Percentage of Read Sectors per Bank. Uniform distribution of read requests would be the best case scenario as it will utilize available parallelism, however storage systems do not have any control over what the user would request for reads.

each bank. The main reason for this is that fact that typical PC user workloads are bursty and request sizes do not vary much within each burst. Although more sophisticated write scheduling algorithms can be implemented, which aim at uniform distribution of write requests, their impact will not be very different. Moreover, they will add more complexity into the flash controller logic, whereas modular striping is a very straight forward algorithm to implement at almost no cost.

Although write requests are distributed equally among banks, the same can not be said of reads. Table 6.9 shows the total number of sectors read from each bank for all user workloads. The distribution of read sectors among banks is not as uniform as writes and in some cases shows high variations. There are two factors which affect how read sectors are distributed among banks. One is temporal locality of requests. If data is written into disk, it is likely that it will be read in the near future. Then the distribution of writes among banks increases the possibility of future read requests being distributed as well. In this case, modular striping also helps in distributing reads - although its impact is limited. This may be controlled better by using a different write scheduling algorithm. Much like modular striping which optimizes write latency (and improves read distribution to some extent), another scheduling algorithm may try to achieve uniform distribution of reads. Note that if write requests are not evenly distributed, some banks might become too busy serving writes. This will degrade performance of reads scheduled to these busy banks. Fortunately this can be worked around by utilizing a RP (read priority) or WSRF scheduling algorithm. Some heuristics that come to mind as an alternative to modular striping are: N^{th} write request assigned to the bank with the least number of write sectors or to the bank with shortest request/read/write queue or to the bank whose blocks have worn out the least (wear leveling).

A second factor which controls how read requests are distributed is simply user workload characteristics. One user workload might generate reads whose addresses are unevenly distributed whereas another user workload might generate read requests whose addresses are within close proximity of each other. Unfortunately, this second factor is

outside of our control. Although it is possible to design a scheduling algorithm which attempts to predict locality of user read requests, it will very likely increase the complexity of flash controller substantially.

6.8. Block Cleaning

As mentioned earlier, flash memory technology does not allow overwriting of data (in-place update of data is not permitted) since a write operation can only change bits from 1 to 0. To change a memory cell's value from 0 to 1, one has to erase a group of cells first by setting all of them to 1.

Typically each flash memory design maintains a list of recently erased blocks (free blocks). As write requests are received, free pages from these free blocks are allocated in consecutive order for these write requests. Over time, the number of free blocks will diminish. In this case flash memory will have to perform some form of block cleaning. Flash memory array will be scanned for blocks with invalid pages as potential erase blocks and block cleaning will be performed. During block cleaning, the flash memory device will be in busy mode and all read and write requests to the device will be stalled in queue.

One of the important factors during this cleaning process is "block cleaning efficiency". Block cleaning efficiency is defined as the percentage of invalid pages to the total number of pages during block cleaning. Efficiency can heavily impact the latency of block cleaning. When a block is cleaned with 100% efficiency, all pages in this block are invalid pages and block cleaning only involves erasing all pages within this block. The typical block erase times are 1.5 or 3 ms. If a block is cleaned with 50% efficiency, half

of the pages within the block has valid user data and has to moved first. 32 pages of valid user data will be read out, written into some other free block and then erase operation will be performed. This will substantially increase cleaning latency. If the movement of valid user data can be limited within the same memory bank or within the same die in a memory bank, copying of valid pages can be performed via an internal move operation. With internal move operation, a page will be read into the cache register and then moved into the data register. While data from the data register is written to a different location, the next page can be read into the cache register. In our example of block cleaning with 50% efficiency, copying of 32 pages of valid user data will take 1 page read (read first page), 31 interleaved page read and write operations and 1 page write (write last page). Assuming 25 μs for page read, 200 μs for page write and 3 μs for cache-to-data register transfer time, it will take 6.521 ms just to move valid data. This will add to the already long latency of the block erase process. Moreover, if copying the valid data is not performed within the same memory bank or within the same die then the data has to be read and written via 8-bit I/O interface and will take even longer.

In order to measure the impact of block cleaning on flash memory performance, we have simulated a sample configuration with a very limited number of free memory blocks. We have used 32 GB flash solid-state disk with 4 memory banks sharing a single 8-bit 25 MHz I/O channel. We have assumed that each 8 GB memory bank has only 2048 free blocks - 256 MB free space. When the number of free blocks falls below a pre-determined level, block cleaning is triggered automatically. A single block is selected as a candidate for block cleaning and all read and write requests are blocked until the block is

cleaned and added into the free block pool. Each flash memory bank has its own 256 MB free block space and implements block cleaning independently. While one bank is busy with block cleaning, other banks may continue to serve their requests if they have free blocks above the threshold. Depending on how low this threshold is set, flash memory banks will start block cleaning much earlier or much later. In order to gage the impact of this threshold, we have simulated 3 different levels - 64 MB, 128 MB and 192 MB. When this threshold is low, flash memory banks start block cleaning later and when it is higher, flash memory banks start block cleaning much earlier.

As we mentioned above, another factor which determines block cleaning latency is block cleaning efficiency. We have simulated block cleaning efficiency levels of 100%, 70% and 30%. A higher cleaning efficiency translates into lower block cleaning latency - thus read and write requests in the queue are blocked for a shorter amount of time. We have also assumed that block cleaning is performed within each bank. This allowed us to perform an internal data move operation during block cleaning if valid user pages needed to be copied.

Figures 6.43-45 show the impact of block cleaning on average disk-request response time, average read response time, and average write response time for various user workloads.

As expected, block cleaning has a negative impact on request response times. However, identifying block cleaning as a performance bottleneck for flash memory solid-state disks would be inaccurate. A closer look at simulation results show that block cleaning efficiency is the parameter that defines the level of performance degradation

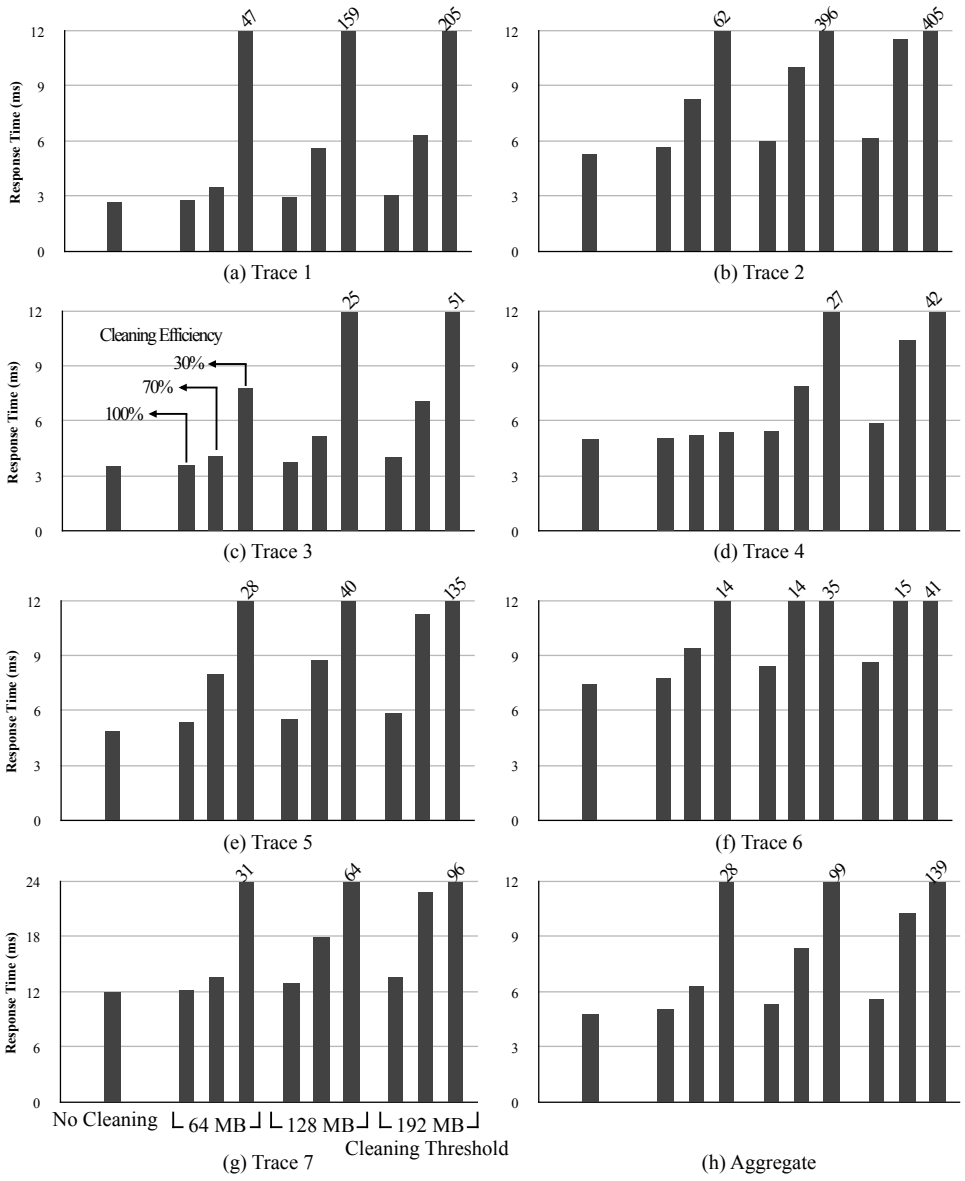


Figure 6.43: Block Cleaning. Average request service time in milliseconds when block cleaning is triggered at different thresholds with varying cleaning efficiency. Block cleaning efficiency is the parameter that defines the level of performance degradation.

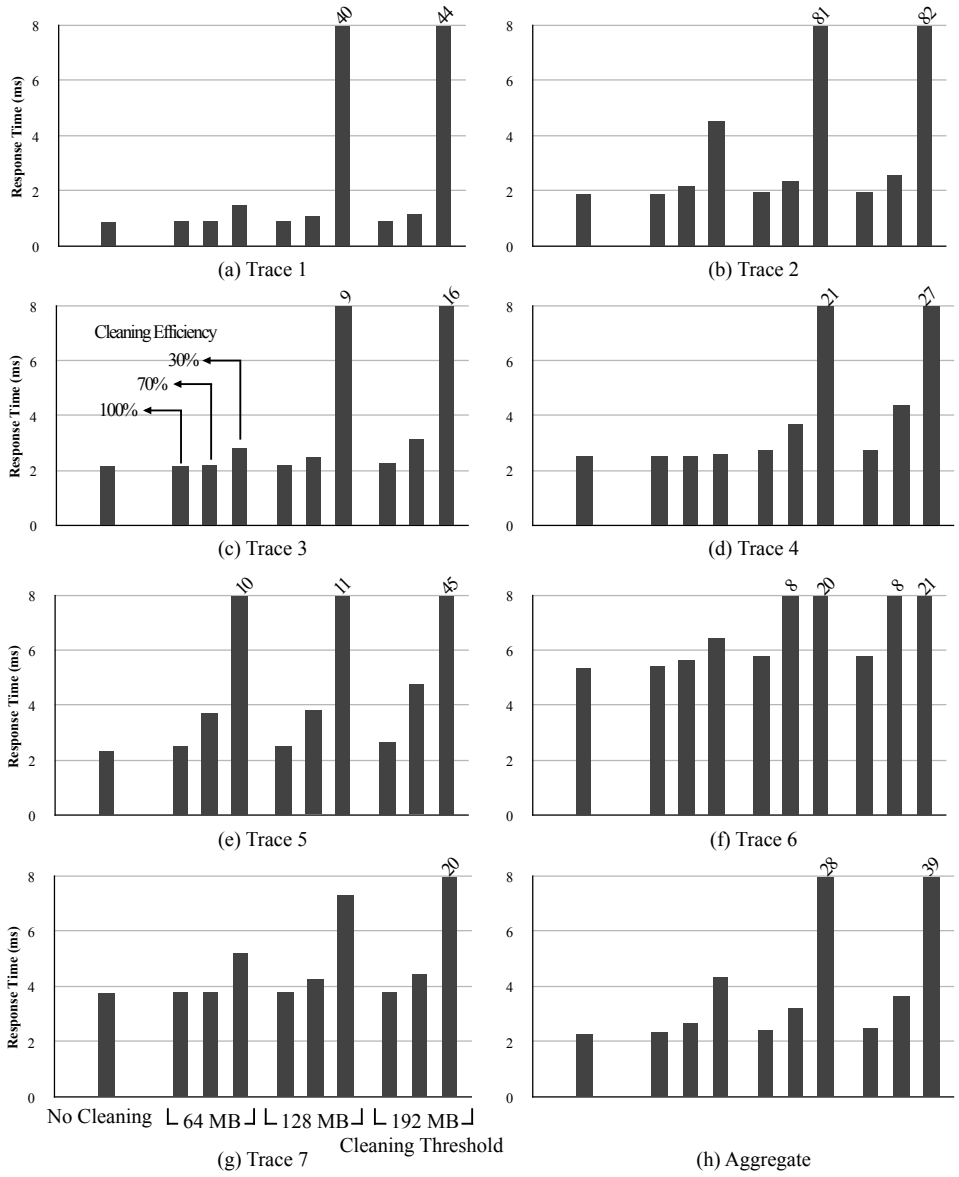


Figure 6.44: Reads with Block Cleaning. Average read request service time in milliseconds when block cleaning is triggered at different thresholds with varying cleaning efficiency. Block cleaning efficiency is the parameter that defines the level of performance degradation.

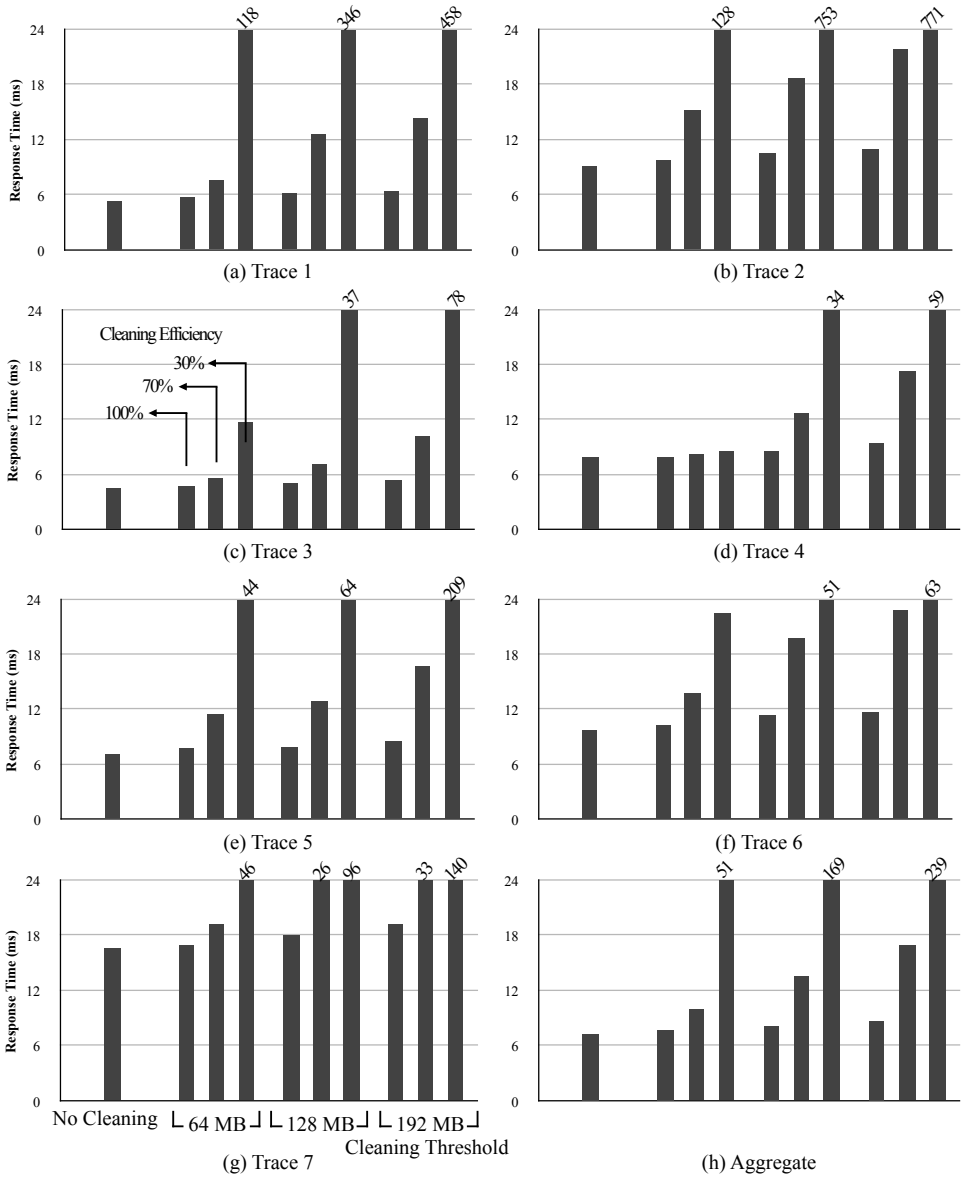


Figure 6.45: Writes with Block Cleaning. Average write request service time in milliseconds when block cleaning is triggered at different thresholds with varying cleaning efficiency. Block cleaning efficiency is the parameter that defines the level of performance degradation.

induced by block cleaning. If we consider results with 100% block cleaning efficiency, the average disk request response time increased by 5-20% depending on the cleaning threshold. For read requests, this was even lower. When cleaning threshold is set to 192 MB - higher threshold triggers block cleaning much earlier - read requests performance only decreased by 7% on average. When block cleaning efficiency is reduced to 70%, latency increase induced by block cleaning starts to show. The impact of the cleaning threshold is better observed when the cleaning efficiency is reduced from 100%.

When block cleaning efficiency is reduced to 30%, the performance of the storage system degrades significantly. Request response times increase 10 times, in some cases 100 times. If we draw on an analogy to main memory, block cleaning for flash memory is what swapping is to RAM. When the amount of swapping is low, the performance degrades but O/S would still operate. If more memory is swapped, the performance would degrade further until the entire virtual memory system starts trashing. Similarly, performance of the flash memory system degrades as block cleaning efficiency decreases until to a point where the flash memory system starts trashing. At this point the storage system becomes so busy with internal data movement and block cleaning that the user requests are held up in the queue almost indefinitely.

One of the performance parameters that affect the cost of block cleaning is identified as uniformity [2]. Uniformity is defined as the fraction of blocks that are uniform in flash memory. All pages in a uniform block are either valid or invalid. In other words, a uniform block does not contain both valid and invalid pages together. The main conclusion of this study is that; the cost of block cleaning increases dramatically when

uniformity is low. The results of our simulations support this conclusion. As uniformity in flash memory decreases, the number of blocks with all invalid pages also decrease. If the number of blocks with all invalid pages is low, the probability of block cleaning with 100% efficiency is also low. When block cleaning is triggered, most of the candidate blocks have both valid and invalid pages within them , decreasing block cleaning efficiency. This dramatically increases request response times as our results show.

It is important to note that block cleaning efficiency not only determines block cleaning latency due to internal data movement (copying of valid pages), but it also determines how many free pages can be added to the free pool. For example, if a block is cleaned with 100% efficiency, the number of free pages increases by 64. Flash memory can serve 128 KB write requests before block cleaning is triggered again. However, if cleaning efficiency is 30%, only 19 pages are added to the pool. The other 45 pages out of 64 available are used in storing valid user data in the cleaned block. Shortly after 38 KB writes are served, block cleaning needs to be performed again. Therefore block cleaning efficiency also contributes to the frequency of block cleaning.

There are several algorithms which can be implemented at FTL to reduce the impact of block cleaning on performance by either ensuring enough free pages when a burst of user requests arrive or by ensuring that block cleaning is implemented at 100% efficiency (minimum block cleaning latency).

One way to reduce the impact of block cleaning is to postpone it as much as possible by employing a variable cleaning threshold. All studies on flash memory performance in literate assumes a fixed block cleaning threshold. With a fixed threshold,

block cleaning will be triggered as soon as this threshold is crossed. On the other hand, we know that typical PC user workloads are bursty. Instead of a fixed threshold, a variable threshold can perform better. A variable threshold can postpone block cleaning until all user requests within a batch are served.

Another way to limit block cleaning is harvesting disk idle times as efficiently as possible. As outlined in [38], in I/O systems there is a lot of idle time for performing background tasks. There has been significant research in the HDD community to harness these idle times to reduce overall power consumption of the I/O system. Since SSDs have an advantage over HDDs in power consumption, the same algorithms can be used for SSDs to harness disk idle times for proactive block cleaning. Figure 6.46 displays disk idle time distribution for a single I/O channel, 4 memory banks configuration. As shown, each memory bank has substantial number of idle periods of larger than 10 ms. When all our user workloads and all 4 memory banks are considered, there are more than 100K idle periods of 10 to 100 ms, more than 30K idle periods of 100 ms to 1 s, and approximately 18K idle periods of 1 to 10 s. If only 60% of idle times which are larger than 1 s can be detected and used for block cleaning, it will eliminate all of the block cleaning operations triggered during normal operation (figure 6.47 illustrates the number of times block cleaning is triggered for all threshold and efficiency levels we have simulated). In an idle period of 1 s, 666 blocks can be cleaned with 100% efficiency or 94 blocks can be cleaned with 30% efficiency. It is possible to detect these idle times using the characteristics of typical user workloads and perform block cleaning operation in the background unobtrusively. If an idle period is not long enough to perform several erase

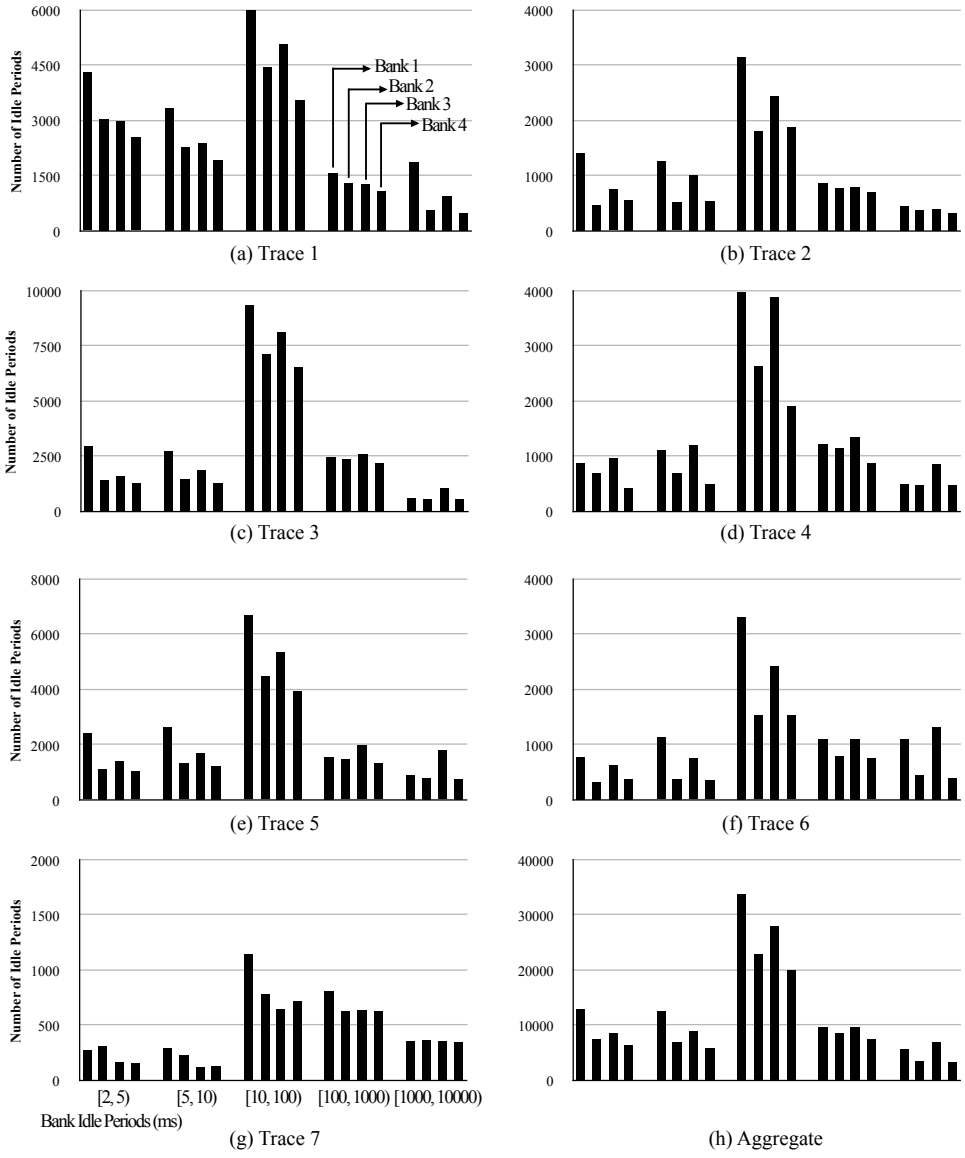


Figure 6.46: Idle Time Distribution. Distribution of idle times for memory banks in a single shared I/O bus, 4 memory banks configuration. Each memory bank has a substantial number of idle periods.

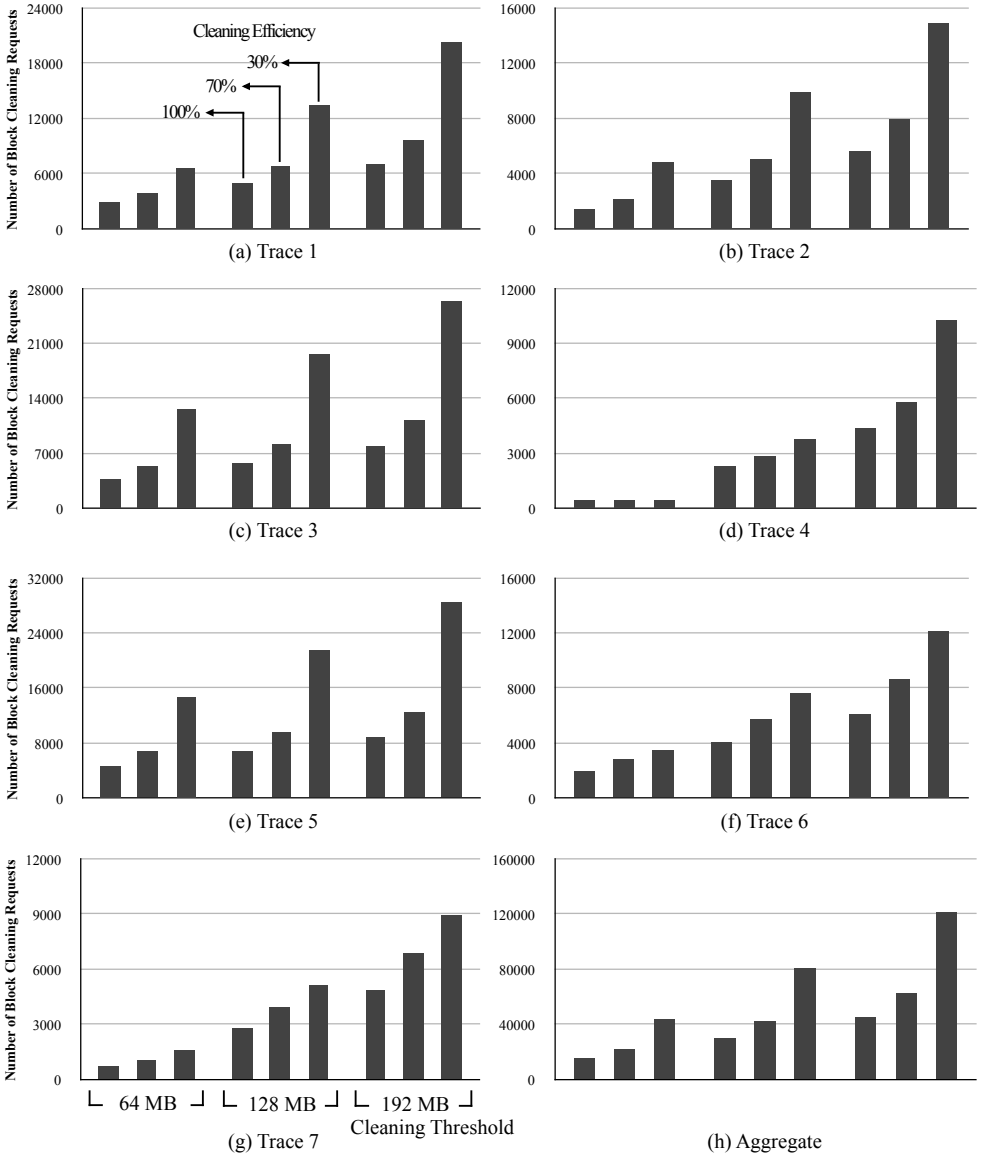


Figure6.47: Block Cleaning Requests. The number of times block cleaning is triggered for all simulated threshold and efficiency levels. If only 60% of idle times which are greater than 1 s can be utilized for block cleaning, it will eliminate all block cleaning operations triggered during normal operation.

operations, it can still be used to move several pages around so that block cleaning efficiency is always kept high. This way, the state of flash memory is optimized so that future block cleaning requests are executed with the least possible latency. On the other hand, all this background activity will increase power consumption of the flash memory system. Even if there are no user requests, flash memory will be performing read, write, and erase operations in the background. Performance impacts of block cleaning is so high that the increase in power consumption is justified.

Baek et. al. [2] suggests a different page allocation scheme which maximizes uniformity in flash storage - thus increasing block cleaning efficiency. As mentioned before, page allocation in flash memory is a run-time resource allocation decision. In our tests we have employed a write striping algorithm to utilize available concurrency. A different resource allocation policy suggested by [2] is modification-aware page allocation, which differentiates between hot data and cold data. Hot data is user data that is updated frequently and allocating hot data to the same block increases uniformity.

In addition to its performance impact, block cleaning also has implications on power consumption of the solid-state disk. Device manufacturers usually report power consumption during typical read and write performance tests. Furthermore, these tests are performed when flash memory is in perfect condition (i.e. blocks either have 100% valid data or recently erased). Typically flash memory consumes same amount of power for read, program and erase operations. For example, sample 1 Gbit memory from Micron draws 25 to 35 mA current during page read, program and block erase operations [59]. When power supply voltage of 2.7 to 3.3 V is considered, this corresponds to 70 to 120

mW per operation. An erase operation will not add too much to power consumption by itself, however when block cleaning efficiency and corresponding internal data movement is concerned power consumption will increase considerably during block cleaning. For example, cleaning a block with 50% efficiency translates into moving 32 pages of valid data - 1 page read, 31 interleaved page read and write and 1 page write - followed by an erase operation. Flash memory has to perform 64 additional operations (64x power consumption) before the desired erase operation. If we consider a sample scenario where I/O requests are 8KB writes (typical average request size) and 50% block cleaning efficiency; block cleaning will be triggered at every 8 requests and system power consumption will increase by a factor of 3. If block cleaning efficiency is 30%, increase in power consumption will be by a factor of 2.6 when there is an incoming stream of average write requests.

6.9. Random Writes and Mapping Granularity

One of the main concerns with flash memory performance has been with random writes. There have been studies documenting poor performance when requests are writes and they are not sequentially ordered. Birrell et. al. looked into commodity USB flash disks and found that the latency of random access writes is consistently higher [7].

The reason behind poor performance of random writes is LBA-PBA mapping performed at the flash translation layer. As we explained in section 3.3.3, flash memory solid-state disks do not support in-place update of data. Rather, every write request for a specific logical address results in data to be written to a different physical address. Therefore, NAND flash memory uses dynamically updated address tables and employes

various mapping techniques to match a logical block address requested by the host system to a physical page or block within flash memory. Most of the typical address mapping algorithms used for flash memory use two map tables; direct map tables and inverse map tables. Direct map tables are stored fully or partially in SRAM. The cost associated with SRAM has been one of the biggest concerns with commodity USB flash memory. If mapping is implemented at block granularity, the size of the map table in SRAM would be small. Its size and cost increases as mapping is performed at a finer granularity.

Mapping granularity not only determines SRAM size and overall cost of the flash memory system, but it also plays an important role in the latency of write requests. Figure 3.8 in section 3.3.3 shows a sequence of events for a sample write request with mapping at block and page granularity. With page size is 2 KB and block size is 128 KB, 4 KB write request will incur drastically different latencies with different mapping schemes. If mapping is performed at block granularity, all valid user data within the target block will be read, 4KB of it will be updated and all valid and updated user data will be written to a free block. In the worst case, if all other pages within the target block are valid, 4 KB (2 pages) write request will trigger the reading of 62 pages and writing of an additional 62 pages. These additional read and writes performed in the background will substantially increase write latency. This is indeed what is observed as “poor performance with random writes”. Figure 6.48 shows a sequence of events for two 4KB write requests with block mapping. When these write requests are random - not sequentially ordered - each 4KB write request not only updates 8 sectors but also moves 16 other sectors. Overall write

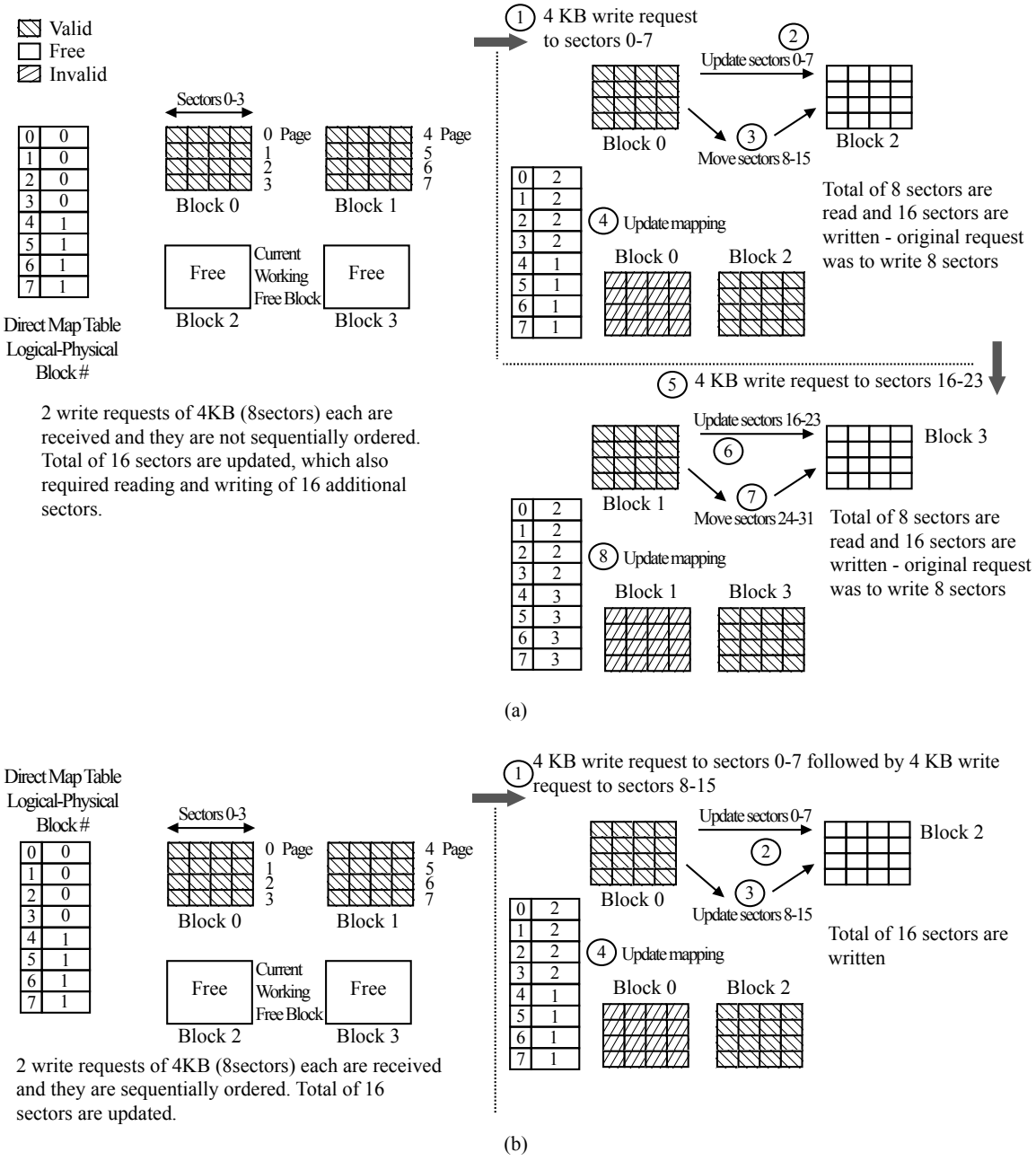


Figure 6.48: Non-sequential Writes. Sequence of events when two 4 KB write requests are received. (a) Write requests are random - non-sequential. (b) Write requests are sequential.

latency is (assuming 25 µs page read and 200 µs page program time) 850 µs. If these write requests were sequentially ordered, there would not be any need for additional data moves. Write latency would be 400 µs. This example illustrates how mapping granularity impacts write latency. If blocks of larger sizes are used, the impact would be greater.

In all of our simulations we have assumed mapping at page granularity. We have also analyzed our user PC workloads to understand the impact of mapping granularity on write requests. Figure 6.49 shows the number of additional write requests in each of our user workloads for various levels of address mapping. When mapping is performed at block level - block sizes of 64 pages - the number of sectors written increases by 3-6x. Although this is a worst case scenario since it assumes all of the pages contain valid user data, it only shows additional write requests. For each additional sector written, there is an additional read request performed in the background. It is important to note that typical PC user workloads include a fair amount of sequential access - spatial locality. With a hypothetical workload which generates truly non-sequential write requests, the additional data copied will be even larger.

In our simulations we have used mapping at page level. This is still higher than conventional hard disk drives where access granularity is a sector. Fortunately, requests in a typical PC user workload are aligned with virtual memory page size of 4 KB. In all our workloads, random writes only increased the total number of sectors written by 0.4%. Our results also show that mapping granularity can be increased to 8 KB, thus matching average workload request size without generating any significant additional reads and writes. Furthermore, mapping granularity can even be increased to 16 KB without

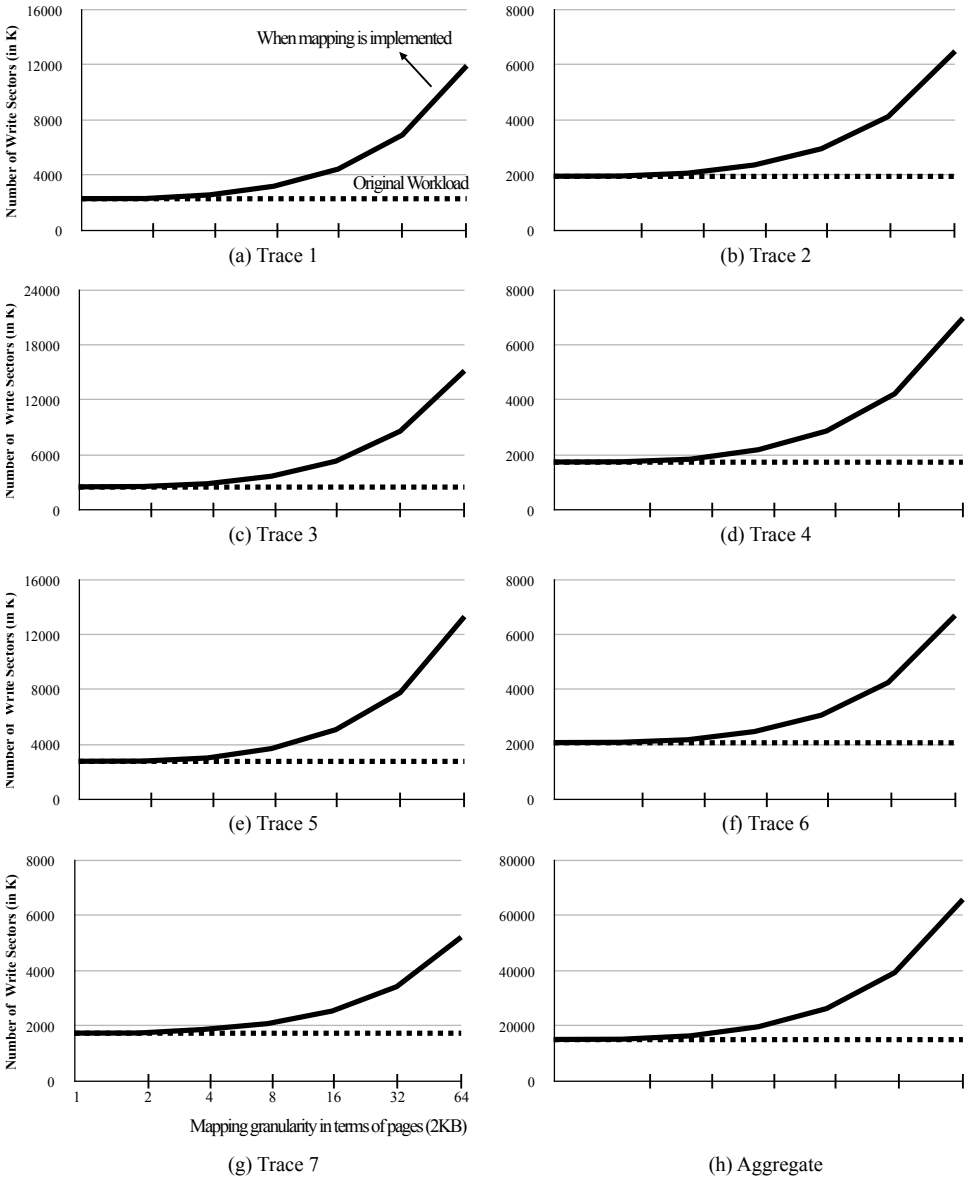


Figure 6.49: Additional number of sectors written. Number of additional sectors written for different levels of mapping granularity. Mapping granularity can be increased to 16 KB with less than 20% increased in the total number of sectors written.

generating more than a 20% increase in the total number of sectors written. Compared to page (2 KB) mapping, mapping at 16 KB will cut SRAM size by 4 with a nominal impact on performance. For designs with only a limited budget for SRAM this can provide a good cost-performance trade-off.

Chapter 7: Conclusions and Future Work

Today's typical solid-state disk drive is a complex storage system. Although it provides a simpler face and lacks the complexities of mechanical parts, it has its own system problems. NAND flash memory employs multiple memory banks in parallel to increase storage system bandwidth and performance. When multiple memory banks are available, data placement and resource allocation becomes a critical for performance and load balancing. The asymmetric nature of read and write requests in flash memory pose additional challenges and increases dependency on user workloads. Effective wear leveling and block cleaning are two other issues unique to flash memory systems which may effect performance.

The relationship between the flash memory system organization and its performance is both very complex and significant. Issues arise in the design of solid-state disk architectures mirror complex system problems. Therefore it is important to study the internals of solid-state disk drives, provide an in-depth analysis of system-level organization choices for solid-state disks, investigate device-level design trade-offs, and provide a model on how solid-state disks work.

We have developed a solid-state disk simulator to measure the performance of various flash memory architectures. Our SSD simulator models a generalized NAND flash memory solid-state disk by implementing flash specific commands and algorithms, all while providing the illusion of an HDD. We have collected our own disk traces from portable computers and PCs running real user workloads to drive our simulator. Our workloads represent typical multi-tasking user activity and consist of not only I/O traffic

generated by user applications, but also I/O requests generated by system and admin processes.

With our SSD simulator, we explored the full design space of system-level organization choices for solid-state disks. We also investigated device level design trade-offs as well, including pin bandwidth, and I/O width. Moreover, we explored the potential for improvements to solid-state disk organizations by flash oriented queueing algorithms and bus access policies. We found the following:

- The flash memory bus does not need to scale up to HDD I/O speeds for good performance. The real limitation to flash memory performance is not its bus speed but its core interface: the movement of data between the flash device's internal storage array and internal data and cache registers.
- SSD organizations that exploit concurrency at both the device- and system-level (e.g. RAID-like organizations) improve performance significantly. These device- and system-level concurrency mechanisms are, to a degree, orthogonal.
- Given a storage system with a fixed media transfer bandwidth, there are always several near-optimal configurations that are within several percent of each other. It is imperative to study full design space of flash memory organizations including performance, cost, and power models.
- NAND flash interface provides drastically different read and write timing which results in large performance disparities between reads and writes. Structural mechanisms and physical organizations outlined in this dissertation mainly target improving the throughput of write requests by reducing flash memory

programming time (ganging, striping, etc.) or by hiding programming latency (request interleaving using multiple banks and channels). However, asymmetry between reads and writes and the scale factor between their performance persists.

- This scale factor between read and write rates make solid-state disk performance more dependent on the user workload.
- The inherent parallelism used in existing solid-state disk systems to amortize write overhead can come at the expense of read performance if not handled carefully.
- When distinctive differences between reading from and writing to flash memory and the impact of system- and device-level concurrency techniques are taken into account, there is potential for further improvements to solid-state disk organizations by flash oriented heuristics and policies. Heuristics and policies suggested in this dissertation accommodate the asymmetry between reads and writes to optimize the internal I/O access to solid-state disk storage system without significant changes to its physical organization.
- Read performance is overlooked in existing flash memory systems. Flash oriented heuristics and policies presented in this dissertation favor read requests over write requests whenever possible; because, as shown in Jacob, Ng, and Wang [40], overall computer system performance (i.e., CPI) tracks disk's average read-response time.

The scope of this dissertation can be further extended to provide better understanding between solid-state disk architectures and their system performance. Following are possible areas for future work:

- Although NAND flash interface modeled in this dissertation provides an accurate timing of read, write (program) and erase operations for SLC flash memory, more and more MLC flash memory chips are becoming commercially available as technology scales down. Therefore our timing models can be enhanced by introducing MLC support into our solid-state disk simulator.
- Our solid-state disk simulator can also be enhanced by incorporating power consumption models into it. Some of the organizational trade-offs investigated may have power constraints, such as maximum level of concurrency allowed.
- In this dissertation, we have followed a trace-driven approach since our primary metric was request response time. An execution driven approach is also possible to model a closed storage subsystem. If execution time is considered as the performance metric, it is important to enable the feedback between the storage system performance and the timing of the subsequent I/O requests.
- NAND flash solid-state disks assume a block device interface and hide their peculiarities from the host system. FTL layer implements various algorithms and data structures to support the block device interface and to extract maximum performance from the flash memory array. However, if solid-state disks identify themselves to the host system different then hard disk drives, better performance and cost trade-offs can be achieved at the system level. One such example is the

support of a trim command being added to the ATA interface standard. With trim command host OS is aware of the underlying solid-state drive and enhances its performance by specifying which files are deleted. When a file is deleted, file system marks it accordingly but does not necessary notify the storage subsystem. To solid-state disks, a deleted user file still appears as valid data and special care must be taken during write operations and block cleaning process. With trim command, file delete information is propagated to the solid-state disk, which in return can mark the data as invalid and avoid costly internal data movement operations. Further research is imperative in flash specific file systems and expanding operating system's support for solid-state disks.

- Its high performance and low power not only enables solid-state disk as an alternative to hard disk drive, also provides another layer in the memory hierarchy after main memory. Furthermore, with the availability of SLC and MLC technologies solid-state disks can be utilized at a finer granularity within the

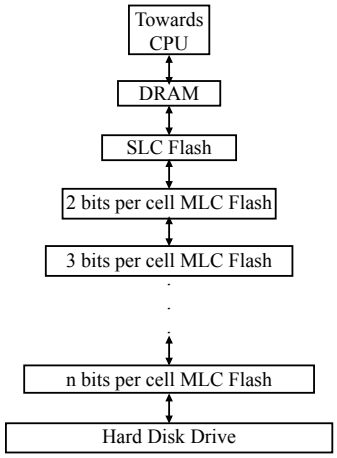


Figure 7.1: Memory Hierarcy. System memory hierarchy can be redesigned to utilize performance and cost trade-offs available through SLC and MLC flash memory technology.

memory hierarchy. A redesign of memory system hierarchy is imperative using various flash memory technologies for better performance and cost trade-offs as summarized in Figure 7.1.

References

[1] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigraphy. "Design Tradeoffs for SSD Performance." In *Proc. USENIX Annual Technical Conference (USENIX 2008)*, Boston, MA, June 2008.

[2] S. Baek, J. Choi, D. Lee, and S. H. Noh. "Performance Characteristics of Flash Memory: Model and Implications." In *Proc. 3rd International Conference on Embedded Software and Systems,* Daegu, Korea*,* pp. 162-173*,* 2007.

[3] S. Baek, S. Ahn, J. Choi, D. Lee, and S. H. Noh. "Uniformity Improving Page Allocation for Flash Memory File Systems." In *Proc. 7th ACM & IEEE International Conference On Embedded Software*, pp. 154-163, 2007..

[4] D. Barnetson. "Solid State Drives: The MLC Challenge." *MEMCO 08*, http:// www.forward-insights.com/present/SandiskMemcon.pdf, 2008.

[5] R. Bez, E. Camerlenghi, A. Modelli, and A. Visconti. "Introduction to Flash Memory." *Proceedings of the IEEE,* vol. 91, no. 4, pp. 489-502, April 2003.

[6] R. Bez and P. Cappelletti. "Flash Memory and Beyond." In *2005 International Symposium on VLSI Technology (IEEE VLSI-TSA)*, pp. 84-87, April 2005.

[7] A. Birrell, M. Isard, C. Thacker, and T. Wobber. "A Design for High-Performance Flash Disks." *ACM SIGOPS Operating Systems Review*, vol. 41, no. 2, pp. 88-93, 2007.

[8] T. Bisson, S. A. Brandt, and D. D. E. Long. "NVCache: Increasing the Effectiveness of Disk Spin-Down Algorithms with Caching." In *Proc. 14th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems,* 2006.

[9] T. Bisson and S. A. Brandt. “Reducing Hybrid Disk Write Latency with Flash-Backed I/O Requests.” In *Proc. 15th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS’07)*, 2007.

[10] Bonnie, Unix File System Benchmark, http://www.textuality.com/bonnie.

[11] J. E. Brewer, and M. Gill. Nonvolatile Memory Technologies with Emphasis on Flash: A Comprehensive Guide to Understanding and Using NVM Devices. Wiles-IEEE Press, 2007.

[12] G. Campardo, R. Micheloni, and D. Novosel. VLSI-design of Non-volatile Memories. Springer, 2005.

[13] Y. B. Chang and L. P. Chang. “A Self-Balancing Striping Scheme for NAND-Flash Storage System.” In *Proc. 2008 ACM symposium on Applied computing,* 2008.

[14] L. P. Chang and T. W. Kuo. “An adaptive striping architecture for flash memory storage systems of embedded systems.” In *Proc. 8th IEEE Real-Time and Embedded Technology and Applications Symposium*, 2002.

[15] L. P. Chang and T. W. Kuo. “Efficient management for large scale memory storage systems.” In *Proc. ACM Symposium On Applied Computing,* 2004.

[16] L. P. Chang and T. W. Kuo. “Efficient Management for Large-Scale Flash-Memory Storage Systems with Resource Conservation.” *ACM Transactions on Storage (TOS),* pp. 381-418, 2005.

[17] L. P. Chang. “On Efficient Wear Leveling for Large-Scale Flash-Memory Storage Systems.” In *Proc. 2007 ACM Symposium on Applied Computing,* Seoul, Korea, 2007.

[18] F. Chen, S. Jiang, and X. Zhang. “SmartSaver: Turning flash memory into a disk energy saver for mobile computers.” In *Proc. 2006 International Symposium on Low Power Electronics and Design (ISLPEDÕ06)*, pp. 412-417, 2006.

[19] M. L. Chiang, P. C. H. Lee, and R. C. Chang. “Using data clustering to improve cleaning performance for flash memory.” *Software-Practice and Experience,* vol. 29, pp. 267-290, 1999.

[20] Y. Choi. “16-Gbit MLC NAND flash weighs in.” *EE Times,* http://www.eetimes.com/showArticle.jhtml?articleID=201200825, July 2007.

[21] T. S. Chung, D. J. Park, S. W. Park, D. H. Lee, S. W. Lee, and H. J. Song. “System Software for Flash Memory: A Survey.” *International Conference of Embedded and Ubiquitous Computing (EUC),* 2006.

[22] B. Cormier. “Hynix is the NAND flash memory engraved in 48 nm memory.” *PC-Inpact,* http://translate.google.com/translate?hl=en&sl=fr&u=http://www.pcinpact.com/actu/news/40473-Hynix-memoire-flash-NAND-48-nm.htm&sa=X&oi=translate&resnum=3&ct=result&prev=/search%3Fq%3DHynix%2B48%2Bnm%2BNAND%26hl%3Den, 2007.

[23] V. Cuppu and B. Jacob. “Concurrency, Latency, or System Overhead: Which Has the Largest Impact on Uniprocessor DRAM-System Performance?” In *Proc. 28th Annual International Symposium on Computer Architecture (ISCA'01),* pp. 62-71, 2001.

[24] C. Dirik and B. Jacob. “The Performance of PC Solid-State Disks (SSDs) as a Function of Bandwidth, Concurrency, Device Architecture, and System Organization.”

Proc. 36th Annual International Symposium on Computer Architecture (ISCA'09), Austin Texas, 2009.

[25] F. Doughs, F. Kaashoek, B. Marsh, R. Caceres, and J. A. Tauber. "Storage alternatives for mobile computers." In *Proc. 1st USENIX Symposium on Operating Systems Design and Implementation (OSDI),* Monterey, California, November 1994.

[26] D. Dumitru. "Understanding Flash SSD Performance." http://managedflash.com/news/papers/easyco-flashperformance-art.pdf, August 2007.

[27] G. Duncan. "Samsung Reveals 30nm, 64 Gb Flash." *Digital Trends,* http://news.digitaltrends.com/news-article/14582/samsung-reveals-30nm-64gb-flash-printer-friendly, 2007.

[28] J. Elliott. "NAND Flash: Becoming the Ubiquitous Storage Platform." *Flash Memory Summit*, http://www.flashmemorysummit.com/English/Collaterals/Presentations/2008/20080813_Keynote5_Elliott.pdf, *August* 2008.

[29] E. Gal and S. Toledo. "Mapping Structures for Flash Memories: Techniques and Open Problems." In *Proc. IEEE International Conference on Software - Science, Technology & Engineering,* 2005.

[30] E. Gal and S. Toledo. "Transactional flash file system." In *Proc. USENIX 2005 Technical Conference,* 2005.

[31] E. Gal and S. Toledo. "Algorithms and Data Structures for Flash Memories." *ACM Computing Surveys*, vol. 37, no. 2, pp. 138-163, 2005.

[32] G. R. Ganger, B. L. Worthington, and Y. N. Patt. “The DiskSim Simulation Environment Version 2.0 Reference Manual.” http://www.pdl.cmu.edu/DiskSim/disksim2.0.html.

[33] G. R. Ganger. “System-oriented evaluation of I/O subsystem performance.” Doctoral Thesis, University of Michigan, 1995.

[34] J. Gray and B. Fitzgerald. "Flash Disk Opportunity for Server-Applications." http://research.microsoft.com/~gray/papers/FlashDiskPublic.doc, January 2007.

[35] Hard Disk Drive Specification Deskstar 7K500. Hitachi, http://www.hitachigst.com/tech/techlib.nsf/techdocs/CE3F5756C827F35A86256F4F006B8AD4/$file/7K500v1.5.pdf, 2006.

[36] Hitachi Deskstar 7K500. Hitachi, http://www.hitachigst.com/tech/techlib.nsf/techdocs/242718EDA9762C0386256F4E006B2F86/$file/7K500ud_final.pdf, 2005.

[37] HLNAND. HyperLink NAND Flash. MOSAID Technologies Inc., http://hlnand.com/852572C9004980E9/ID/Next-Gen-Memory-WP1, May 2007.

[38] W. Hsu and A. J. Smith. “Characteristics of I/O Traffic in Personal Computer and Server Workloads.” *IBM Systems Journal*, vol. 2, no. 2, pp. 347-372, April 2003.

[39] C. Hwang. “Nanotechnology Enables a New Memory Growth Model.” *Proceedings of the IEEE*, vol. 91, no. 11, pp. 1765-1771, November 2003.

[40] B. Jacob, S. Ng, and D. Wang. Memory Systems: Cache, DRAM, Disk. Morgan Kaufmann, 2007.

[41] H. Jeong. “Trends and Vision for the future memory.” *9th Leti Review,* http://www-leti.cea.fr/home/liblocal/docs/Annual%20Review/PRESENTATIONS/Leti_AR07_session2_2.2_Hongsik%20Jeong.pdf, 2007.

[42] JFFS2: The Journalling Flash File System. Red Hat Corporation. http://sources.redhat.com/jffs2/jffs2.pdf, 2001.

[43] T. Kgil, T. Mudge. “FlashCache: a NAND flash memory file cache for low power web servers.” In *Proc. 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems,* Seoul, Korea, pp. 103 -112, 2006.

[44] H. Kim and S. Ahn. “A Buffer Management Scheme for Improving Random Writes in Flash Storage.” In *Proc. 6th USENIX Symposium on File and Storage Technologies (FAST’08)*, pp. 239-252, 2008.

[45] B. Kim, S. Cho, and Y. Choi. “OneNAND (TM): A High Performance and Low Power Memory Solution for Code and Data Storage.” In *Proc. 20th Non-Volatile Semiconductor Workshop,* 2004.

[46] J. Kim, J. M. Kim, S. H. Noh, S. L. Min, and Y. Cho. “A Space-Efficient Flash Translation Layer for CompactFlash Systems.” *IEEE Transactions on Consumer Electronics,* vol 48, pp. 366-375, 2002.

[47] Y. J. Kim, K. Y. Kwon, and J. Kim. “Energy Efficient File Placement Techniques for Heterogeneous Mobile Storage Systems.” In *Proc. 6th ACM & IEEE International conference on Embedded software,* 2006.

[48] Y. Kim, S. Lee, K. Zhang, and J. Kim. “I/O Performance Optimization Techniques for Hybrid Hard Disk-Based Mobile Consumer Devices.” *IEEE Transactions on Consumer Electronics*, vol. 53, no. 4, pp. 1469-1476, November 2007.

[49] T. Krazit. “Intel flashes ahead to 1 Gb memory.” *CNET News,* http://news.cnet.com/Intel-flashes-ahead-to-1Gb-memory/2100-1006_3-6057216.html?tag=nw.11, 2006.

[50] T. W. Kuo, J. W. Hsieh, L. P. Chang, and Y. H. Chang. “Configurability of performance and overheads in flash management.” In *Proc. 2006 Conference on Asia South Pacific Design Automation,* 2006.

[51] M. LaPedus. “Intel, Micron roll 34-nm NAND device.” *EE Times,* http://www.eetimes.com/showArticle.jhtml?articleID=208400713, 2008.

[52] M. LaPedus. “SanDisk, Toshiba to ship 32-nm NAND in '09.” *EE Times,* http://www.eetimes.com/news/latest/showArticle.jhtml?articleID=212800210&printable=true&printable=true, 2009.

[53] S. H. Lim and K. H. Park. “An efficient NAND flash file system for flash memory storage.” *IEEE Transactions on Computers,* pp. 906-912, 2006.

[54] G. MacGillivray. “Inside Intel's 65-nm NOR flash.” *Techonline,* http://www.techonline.com/article/printArticle.jhtml?articleID=196600919, December 2006.

[55] C. Manning. “YAFFS: Yet Another Flash File System.” http://aleph1.co.uk/yaffs, 2004.

[56] R. McKee. “Lecture 26: Embedded Memory - Flash.” *EE241 - Spring 2005, University of California, Berkeley*, http://bwrc.eecs.berkeley.edu/Classes/icdesign/ee241%5Fs05/Lectures/Lecture26-Flash.pdf, 2005.

[57] Memory Management in NAND Flash Arrays. Micron Technology, Inc. Technical Note TN-29-28. http://download.micron.com/pdf/technotes/nand/tn2928.pdf, 2005.

[58] S. L. Min and E. H. Nam. "Current Trends in Flash Memory Technology." In *Proc. 2006 Asia South Pacific Design Automation (ASP-DAC '06)*, pp. 332-333, January 2006.

[59] MT29F1GxxABB 1 Gb NAND Flash Memory. Micron Technology, Inc. http://download.micron.com/pdf/datasheets/flash/nand/1gb_nand_m48a.pdf, 2006.

[60] MT29FXX08XXX 4Gb NAND Flash Memory. Micron Technology, Inc. http://download.micron.com/pdf/datasheets/flash/nand/4gb_nand_m40a.pdf, 2006.

[61] A. S. Mutschler. "Toshiba touts 43-nm CMOS 16-Gb NAND flash." *EDN: Electronics Design, Strategy, News,* http://www.edn.com/index.asp?layout=articlePrint&articleID=CA6529998, 2008.

[62] D. Myers. "On the Use of NAND Flash Memory in High-Performance Relational Databases." Master's thesis, MIT, 2007.

[63] NAND Flash Applications Design Guide. Toshiba America Electronic Components, Inc., http://www.dataio.com/pdf/NAND/Toshiba/NandDesignGuide.pdf.pdf, April 2003.

[64] NAND Flash-based Solid State Disk Module Type Product Data Sheet. Samsung Electronics Co., Ltd., http://www.bigboytech.com/new/v1.5/ssd/docs/ssd_module_type_spec_rev121.pdf, January 2007.

[65] NAND vs. NOR Flash Memory Technology Overview. Toshiba America Electronic Components, Inc., http://www.toshiba.com/taec/components/Generic/Memory_Resources/NANDvsNOR.pdf, 2006.

[66] NSSD (NAND Flash-based Solid State Drive) Standard Type Product Data Sheet. Samsung Electronics Co., Ltd., http://www.samsung.com/global/business/semiconductor/products/flash/Products_StandardType_25inch.html, 2007.

[67] Onfi, Open NAND Flash Interface, http://onfi.org

[68] C. Park, W. Cheon, Y. Lee, M. S. Jung, W. Cho, and H. Yoon. “A Re-configurable FTL (Flash Translation Layer) Architecture for NAND Flash based Applications.” *18th IEEE/IFIP International Workshop on Rapid System Prototyping (RSP),* vol. 28, pp. 202-208, 2007.

[69] C. Park, P. Talawar, D. Won, M. Jung, J. Im, S. Kim, and Y. Choi. "A High Performance Controller for NAND Flash-based Solid State Disk (NSSD)." In *Proc. 21st IEEE Non-Volatile Semiconductor Memory Workshop (NVSMW)*, pp. 17-20, 2006.

[70] D. Parthey and R. Baumgartl. “Timing Properties of Removable Flash Media.” *Junior Researcher Workshop on Real-Time Computing*, 2007.

[71] V. Prabhakaran and T. Wobber. “SSD Extension for DiskSim Simulation Environment.” *Microsoft Reseach*, http://research.microsoft.com/en-us/downloads/b41019e2-1d2b-44d8-b512-ba35ab814cd4/default.aspx, March 2009.

[72] M. Rosenblum and J. K. Ousterhout. “The Design and Implementation of a Log-Structured File System.” *Proceedings of the 13th ACM Symposium on Operating Systems Principles* 1-15, 1991.

[73] S. Seguin. “Toshiba Launches First 512 GB SSD.” *Tom's Hardware,* http://www.tomshardware.com/news/Toshiba-512GB-SSD,6716.html*,* 2008.

[74] Y. Shin. "Non-volatile Memory Technologies for Beyond 2010." *2005 Symposium on VLSI Circuits*, pp. 156-159, June 2005.

[75] STMicroelectronics Offers Automotive-Grade 32 Mbit NOR Flash Memory. *IHS,* http://parts.ihs.com/news/stmicroelectronics-32mbit-nor.htm, 2007.

[76] Two Technologies Compared: NOR vs. NAND. M-Systems, http://www.dataio.com/pdf/NAND/MSystems/MSystems_NOR_vs_NAND.pdf, July 2003.

[77] J. Walko. "NOR flash parts move to 65 nm processing." *EE Times Asia,* http://www.eetasia.com/ART_8800556954_480200_NP_509eba15.HTM#, 2008.

[78] G. Wong. "Flash Memory Trends." *Flash Memory Summit*, http://web.njit.edu/~rlopes/5.2%20-%20Flash%20Memory%20Trends_FMS.pdf, August 2008.

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

[80] J. H. Yoon, E. H. Nam, Y. J. Seong, H. Kim, B. Kim, S. L. Min, and Y. Cho. "Chameleon: A High Performance Flash/FRAM Hybrid Solid State Disk Architecture." *IEEE Computer Architecture Letters*, vol. 7, no. 1, pp. 17-20, January 2008.

[81] C. Yuanhao, J. W. Hsieh, and T. W. Kuo. "Endurance enhancement of flash-memory storage systems: an efficient static wear leveling design." In *Proc. 2007 Design Automation Conference,* pp. 212-217, 2007.