The Pennsylvania State University

The Graduate School

Department of Computer Science and Engineering

DESIGN OF A SMART NON-VOLATILE MEMORY CONTROLLER: ARCHITECTURE MODELING, SYSTEMS ANALYSIS, PARALLEL I/O PROCESSING AND SCHEDULING ALGORITHMS

A Dissertation in

Computer Science and Engineering

by

Myoungsoo Jung

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Doctor of Philosophy

August 2013

The dissertation of Myoungsoo Jung was reviewed and approved[1] by the following:

Mahmut T. Kandemir
Professor of Computer Science and Engineering
Dissertation Adviser
Chair of Committee

Padma Raghavan
Professor of Computer Science and Engineering

Chita R. Das
Professor of Computer Science and Engineering

Long-Qing Chen
Professor of Material Science and Engineering

Raj Acharya
Head of the Department of Computer Science and Engineering

[1]Signatures are on file in the Graduate School.

Abstract

State-of-the-art Solid State Disks (SSDs) and Non-Volatile Memory (NVM) systems have undergone severe technology shift and architectural changes in the last couple of years, and, in parallel, SSD internal architecture has dramatically changed; modern SSDs now employ multiple internal resources such as NVM chips and I/O buses in an attempt to achieve high internal parallelism in processing I/O requests. In addition, to reduce intrinsic NVM system management overheads, SSD firmware employs advanced memory control strategies such as finer-granular address mapping algorithms and concurrency methods. As a result of complex interactions among these different mechanisms, modern SSDs can be plagued by enormous performance variations depending on whether the underlying architectural complexities and NVM management overheads can be hidden or not.

Designing a smart NVM controller is key hiding the architectural complexities and reducing the internal firmware overheads. To this end, we first model a multi-plane and multi-die NVM architecture, which is highly reconfigurable and aware of intrinsic latency variation imposed by diverse state-of-the-art NVM systems. This NVM model has been implemented as a high fidelity open-source simulator, capable of capturing cycle-level interactions between the many components in an SSD, which can be used for various high-level and low-level NVM performance analyses. Based on this architecture model, we then explore twenty four different concurrency methods implemented in NVM controllers, geared toward exploiting both system-level and NVM-level parallelism. Further, we quantitatively analyze the challenges, faced by

PCI Express-based (PCIe) SSDs in getting NVM closer to CPU and question popular assumptions and expectations regarding storage-class SSDs through an extensive experimental analysis.

Next, we present and discuss the significance of read performance degradations and write performance variations by performing comprehensive empirical experiments using a diverse set of commercial SSDs and propose two novel schedulers in order to address these read/write performance challenges that modern SSDs face: 1) Physical Address Queuing (PAQ) scheduler and 2) NVM garbage collection scheduling algorithm. PAQ is a novel I/O request scheduling method that avoids resource contention resultant from shared SSD resources. Our proposed PAQ significantly improves read performance by exposing the physical addresses of requests to the scheduler and selecting groups of operations that can be simultaneously executed without major resource conflict. In comparison, the novel garbage collection scheduler is an approach that removes garbage collection overheads of underlying flash firmware and provides stable write performance in SSDs during the I/O congestion periods. Our proposed garbage collection scheduler tries to secure free blocks and remove on-demand garbage collections from the critical path in advance or delay them to future idle periods, so that users do not experience garbage-collection-induced latencies during the I/O-intensive periods. Overall, this thesis (1) presents a simulation infrastructure to conduct SSD/NVM research, (2) characterizes both system-level and device-level challenges faced by state-of-the-art SSDs, (3) presents a set of novel storage optimizations including various concurrency methods and scheduling algorithms design, and (4) points out future research directions.

Table of Contents

List of Tables ix

List of Figures x

Acknowledgments xv

Chapter 1. Introduction 1

Chapter 2. NAND Flash Memory System Modeling and Simulation 6
- 2.1 Introduction 6
- 2.2 NAND Flash Microarchitecture 11
- 2.3 NAND Flash Operations 13
  - 2.3.1 Cache Mode Operation 14
  - 2.3.2 Internal Data Move Mode Operation 14
  - 2.3.3 Multi-plane Mode Operation 15
  - 2.3.4 Interleaved Die Mode Operation 15
- 2.4 Intrinsic Latency Variation of NAND Flash 18
- 2.5 Related Work 19
- 2.6 High Level View Of NANDFlashSim 22
  - 2.6.1 Software Architecture 22
  - 2.6.2 Clock Domains and Lifetime of Transaction 23
- 2.7 Implementation Details 25
  - 2.7.1 NAND Command Set Architecture 25
  - 2.7.2 Awareness of Latency Variation 26
  - 2.7.3 Enforcing Reliability Parameters 28
- 2.8 Evaluation 28
  - 2.8.1 Validation of NANDFlashSim 31
  - 2.8.2 Individual Cycle Analysis 34
  - 2.8.3 Performance and Power Consumption Comparison: Page Migration Test 37
  - 2.8.4 Breakdown of Read and Write Cycles 39
  - 2.8.5 Performance on Multi-plane and Multi-die Architectures 39
  - 2.8.6 Performance Sensitivity to Page Size 45
  - 2.8.7 Resource Contention 45
  - 2.8.8 Scheduling Strategy 46
- 2.9 Simulation Speed and Download 46
- 2.10 Conclusion 46

Chapter 3. Page Allocation Strategies for Parallelizing Data Accesses 48
3.1 Introduction 48
3.2 SSD Internals and Parallelisms 51
3.3 Page Allocation Strategies 53
3.4 Experimental Methodology 56
3.5 Results 56
3.5.1 Finding Overall Optimal Palloc scheme 57
3.5.2 Parallelism Interference 59
3.5.3 Resource Utilization 61
3.5.4 Optimization Potential for Parallelism 64
3.5.5 Discussion 65
3.6 Conclusion 66

Chapter 4. Challenges in Getting Flash Drives Closer to CPU 67
4.1 Introduction 67
4.2 Bringing SSDs Closer to CPU 70
4.2.1 PCIe Architecture 70
4.2.2 Flash Software Stack 72
4.3 Experimental Setup 74
4.4 Challenges in Resource Management 75
4.4.1 Memory Usage 75
4.4.2 CPU Usage 78
4.4.3 Challenges in System Performance 79
4.5 System Implication 84

Chapter 5. Revisiting Widely Held SSD Expectations and Rethinking System Level Implications 86
5.1 Introduction 86
5.1.1 Rethinking Read Performance 88
5.1.2 Examining Reliability on Reads 89
5.1.3 Reconsidering Write Performance 89
5.1.4 OS Support 90
5.2 Preliminaries 90
5.2.1 SSD and NAND Flash Internals 90
5.2.2 Flash Firmware 91
5.2.3 Reliability Challenges on Reads 93
5.2.4 Storage Interfaces, TRIM and SMART 95
5.3 Evaluation Setup 95
5.4 Testing Expectations on Reads 98
5.4.1 Are SSDs good for applications that exhibit mostly random reads? . 98
5.4.2 Can we achieve sustained read performance with sequential accesses? 101
5.4.3 What is the relationship between read performance and previous writes accesses? 103
5.4.4 Do program/erase (PE) cycles of SSDs increase during read only operations? 107

5.4.5 Is there any performance impact of the reliability management on reads? . . . 108
5.5 Testing Expectations on Writes . . . 110
5.5.1 How much impact does the worst-case latency have on modern SSDs? 111
5.5.2 What is the correlation between the worst-case latency and system throughput? . . . 112
5.5.3 Could DRAM buffer help the firmware to reduce garbage collection overheads? . . . 114
5.6 Testing Expectations on Advanced Schemes . . . 117
5.6.1 Can TRIM command reduce GC overheads? . . . 117
5.6.2 Does a TRIM command incur any overheads? . . . 119
5.7 Rethinking SSD Systems . . . 123
5.7.1 Reads . . . 125
5.7.2 Writes . . . 126
5.7.3 Advanced Schemes . . . 127
5.8 Related Work . . . 127
5.9 Conclusion . . . 128

Chapter 6. Memory Request Scheduling: Improving Parallelism in Solid State Drives 130
6.1 Introduction . . . 130
6.1.1 Contributions . . . 132
6.2 Background . . . 134
6.2.1 High-Level Architecture of SSD . . . 134
6.2.2 Multi-Plane Mode Operation . . . 136
6.3 Random Write vs. Random Read Performance . . . 137
6.4 Physically Addressed Queuing . . . 141
6.4.1 QBM Migration . . . 141
6.4.2 Conflict Classification . . . 142
6.4.3 Clump Composition . . . 144
6.4.4 Plane Packing . . . 147
6.4.5 Implementation of PAQ Scheduling . . . 148
6.5 Experimental Setup . . . 152
6.5.1 NAND Flash SSD Simulator . . . 152
6.5.2 SSD Configuration and Schedulers Tested . . . 152
6.5.3 Traces . . . 153
6.6 Experimental Results . . . 155
6.6.1 Aggregate Performance: Bandwidth, IOPs, and Latency . . . 156
6.6.2 Quantifying Parallelism: Idle Time and Contention . . . 159
6.6.3 Overheads of PAQ . . . 160
6.6.4 Time Series: In-Depth Analysis of a Database Trace . . . 160
6.6.5 Sensitivity Analysis . . . 161
6.7 Related Work . . . 162
6.8 Conclusion . . . 167

Chapter 7. Garbage Collection Scheduling 169
7.1 Introduction 169
7.2 Background and Related work 173
7.2.1 Flash Translation Layer 173
7.2.2 Garbage Collection 174
7.3 Impact of Garbage Collection in Commercial SSDs 175
7.4 High Level View of GC Scheduling 177
7.4.1 Idle Period Classification 177
7.4.2 Shifting Garbage Collection Overheads to Idle Periods 180
7.5 Implementation of our GC strategies 181
7.5.1 Details of Advanced GC Strategy (AGC) 181
7.5.1.1 Look-ahead Garbage Collection 181
7.5.1.2 Proactive Block Compaction 184
7.5.1.3 Incremental Garbage Collection 185
7.5.2 Details of Delayed GC Strategy (DGC) 186
7.5.2.1 Update Block Replacement 186
7.5.2.2 Retroactive Block Compaction 189
7.5.3 Putting the Two Schemes Together 190
7.6 Experimental Evaluation 190
7.6.1 Performance Comparison 194
7.6.2 Worst Case Response Time 196
7.6.3 Excess Waiting Time 198
7.6.4 Performance Compariosn of 3SSDs-RAID 200
7.6.5 Side-Effects of AGC and DGC 200
7.7 Conclusions 202

Chapter 8. Future Work 203
8.1 Near Term Future Research Directions 203
8.1.1 QoS-aware and GC-aware Host Interface Scheduler 203
8.1.2 Out-of-Order Non-Volatile Memory Execution 203
8.1.3 NVM Power Modeling 204
8.1.4 High-speed Non-Volatile Memory Interface 205
8.2 Long Term Future Research Directions 205
8.2.1 SSD Redesign 205
8.2.2 Exposing NVM to Computational Resources 206
8.2.3 On-Chip NVM Systems 206

References 207

List of Tables

2.1 NAND Flash Device Characterization 31
2.2 Workloads Characterization 31

3.1 Important characteristics of our traces. 58

4.1 Important characteristics of the tested PCIe SSDs. 73

5.1 Device characteristics of SSDs used in our study. 96

6.1 Trace decomposition into the number of writes and reads, and the percentage of random-reads and random-writes issued. 153

7.1 Important characteristics of our traces. The last column gives % of I/O requests containing sufficiently long idle (> 1 sec) periods. 192

List of Figures

1.1 Design and optimization goal of a smart NVM controller. Modern SSDs can be plagued by enormous performance variations depending on whether underlying architectural complexities and NVM firmware overheads can be hidden or not. Designing a smart NVM controller is key hiding the architectural complexities and reducing the internal firmware overheads. 2

2.1 Concept of a μarch-level NAND flash simulation model (NANDFlashSim). While existing SSD simulators are highly coupled to flash firmware emulation with simplified latency model, NANDFlashSim simulates NAND flash memory system itself with independently synchronous clock domains and detailed NAND operation timing models aware of latency variation. 7
2.2 Intrinsic Latency Variation Example. 9
2.3 NAND flash memory system internals. 12
2.4 A timing diagram of interleaved die with four legacy writes. While an existing simulation model simplifies bus activities and assumes that latencies are perfectly overlapped and interleaved with constant time, NANDFlashSim employs fine-grain bus activities and is aware of intrinsic latency variations. 16
2.5 NAND flash memory cell organization. MLC NAND flash memory has multiple states in a cell, which causes intrinsic latency variation [65]. 19
2.6 NANDFlashSim architecture. NANDFlashSim is a reconfigurable μ-level multi-plane and multi-die architecture. The number of registers, blocks, planes and dies can be reorganized, and each entity has independent synchronized clock domain. 21
2.7 The process of NAND flash memory transactions and examples of NAND command chains. NANDFlashSim handles fine-grain NAND transactions by NAND command set architecture 24
2.8 State machine for multiple NAND stages. Each state is identified by different type of stages and states of the machine are transited by different type of NAND commands. Since each die has their own state machine, NANDFlashSim provides an independent clock cycle domain per die. 25
2.9 Implemented evaluation hardware prototype (MSIS). MSIS is used to extract the LSB and MSB page address and to evaluate performance for NANDFlashSim validation. 29
2.10 Cumulative Distribution Function (CDF) of Latency. Latencies of NANDFlashSim are mostly overlapped with real NAND flash product latencies. 30
2.11 Performance comparison on SDP. Typical-case/worst-case time parameter based latency models show unreasonable performance gap from real product ones. . 33
2.12 Performance comparison on DDP. While typical-based latency model show more discrepant performance than that of multi-plane tests, the variation-aware NANDFlashSim provides performances close to the real product ones. 34

2.13 Cycle analysis for write operations (NANDFlashSim). Note that the command sequence is a chronological oder based on a set of NAND commands that host commits, and one cycle takes 1 nanosecond. 35
2.14 Cycle analysis for read operations (NANDFlashSim). Since handling 2x mode is fancy more than other operation modes, it requires more commands to read data. Note that system designers is able to get these microscopic cycle analysis for diverse NAND flash operations from outputs of NANDFlashSim. 36
2.15 Block migration performance comparison and energy consumption. While energy consumption are similar to different NAND operation modes, 2x cache mode and internal data move mode operations have great impact on enhancing performance in the page migration test. 38
2.16 Breakdown of cycles. Note that, in reads, TOR operations are the performance bottleneck while TIN operations are on the critical path in writes. 40
2.17 Cycle analysis for page migration. Internal data move modes removes the most NAND I/O bus activities thereby improving throughputs. 40
2.18 multi-plane architecture performance with varying page unit sizes. While write throughputs of many plane architecture are enhanced by 360.9% to single plane architecture, read throughputs are enhanced by 75.5%. 41
2.19 Multi-die architecture performance. While many dies architecture with the swr workload enjoys linear enhancement (ODP improves throughput 541.1% to the SDP ones), it saturates read throughputs with eight dies (76.3% enhancement compared to SDP ones). 42
2.20 Performance sensitivity to the number of plane and die with actual application workloads. the performance of many-die architecture is 54.5 % better than the performance of many-plane architecture in terms of IOPS. 43
2.21 Resource contention comparison between SLC NAND and MLC NAND devices. The resource contentions of MLC NAND flash have less impact on SLC NAND flash, but the contention problem is still problematic and become more serious as the number of die increases. 43
2.22 Sensitivity to page organization (2x, DDP). Most read performance are bounded because of TOR times and NAND flash I/O bus competition. 44
2.23 Effects of different NAND command scheduling policies. Based on different transfer sizes and scheduling policies, performance enhancement with multi-die and plane architecture show different performance. 44

3.1 Internals of SSD and NAND flash and illustration of different level data accessing methods. 50
3.2 Request-level parallelism. 53
3.3 Different page allocation strategies. 55
3.4 Throughput comparison. IOPS numbers are normalized with respect to corresponding CDPW IOPS. 57
3.5 Latency comparison. Latency values are normalized with respect to corresponding CDPW values. 57
3.6 The number of flash-transactions executed. 60
3.7 Waiting time required to resolve resource conflicts. 60

3.8 Parallelism breakdown for sql1. 62
3.9 Average channel utilization. 62
3.10 Execution breakdown (msnfs, write-intensive). 63
3.11 Execution breakdown (web, read-intensive). 63
3.12 Performance map with optimizations points. 64

4.1 Bandwidth trends over time for the thin interfaces versus SSDs (a), and flash storage integration into a place closer to CPU (b). 68
4.2 High-level views of our PCIe SSD architectures and their software stacks. . . . 71
4.3 Memory usage. 76
4.4 Time series comparison for host resources usage between FSSD (top) and BSSD (bottom) with the default 512B block-size access operation. 77
4.5 CPU usage. 78
4.6 Latency and throughput comparison. Note that all the latency and throughput of BSSD values are normalized to corresponding values of FSSD. 79
4.7 FSSD performance characteristics on the multi-core environment. 81
4.8 BSSD performance characteristics on the multi-core environment. 82
4.9 Queueing latency comparison observed by FSSD (a) and BSSD (b). 83

5.1 Modern SSD internal architecture. Note that an I/O request can be simultaneously served by many internal resources, which is one of the important characteristics of SSDs. 92
5.2 Read/write performance comparison under varying data transfer sizes and access patterns. In these comparisons, *RND* and *SEQ* denote the random access pattern and sequential access pattern respectively, and *RD* and *WR* stand for read and write. 99
5.3 Cumulative distribution function (CDF) of latency variance in sequential reads for “pristine” SSDs and “aged” SSDs (i.e., writing data with a random access pattern to the entire storage space of SSDs). Note that all the curves presented in the CDFs are shifted from left to right as SSDs get older. 102
5.4 Bandwidths with different physical data layouts. *SEQ* and *RND* denote sequential writes and random writes, used for the physical data (*PDT*) layout construction. Observe that throughput significantly varies based on the physical data layout, constructed by previous writes, even under same read request patterns. 104
5.5 Latencies with different physical data layouts. These latency comparison explains how the physical data layout is related to internal parallelism in two aspects. First, the read latency performed on RND-PDT is 2.3 times higher than that of SEQ-PDT that induces lower resource conflicts. Second, as the data movement size of reads increases, the magnitude of the latency improvement with SEQ-PDT is shorter than the improvement with RND-PDT that has many resource conflicts potential. 106
5.6 PE-cycle comparison between reads and writes. Note that PE cycles increase under read-only workloads. Further, with random accesses, the maximum PE cycles on reads are 12 times greater than that on writes. 109

5.7 Latency comparison between reads with reliability management (RMR) and ordinary reads (i.e., reads without RMR). When RMR is employed, the latency is at least 5 times higher than the latency of reads without RMR. 110
5.8 SSD and HDD latency comparison. While SSDs overall outperform HDDs, the worst-case latencies of SSDs are much higher than the worst-case latencies of HDDs. 111
5.9 Impact of write cliff. Initially, SSDs provide reasonable performance even though GCs are invoked. However, once write cliff begins, the performance significantly degrades and is not recovered later. 113
5.10 Worst-case latency correlation between the DRAM buffer cache and GC. The DRAM buffer provides excellent latency, but after the write cliff, it makes latencies even worse. 115
5.11 Bandwidth impact of TRIM. While SEQ-TRIM (the order of target addresses is ascending) can effectively remove GCs, RND-TRIM (the order of targets is random) has no impact on GC overheads. 116
5.12 Latency impact with TRIM. Similar to bandwidth impact, there is no latency gain with RND-TRIMs. 118
5.13 E-TRIM overheads. Since E-TRIM performs block erasure on demand and do not return control to the storage system, the host can be disabled until the TRIM process finishes. 120
5.14 I-TRIM overheads. I-TRIM is more efficient in controlling the TRIM commands, but the latency overheads are still about 6 $\sim$ 153 times worse than a 4KB write-latency. 121
5.15 Performance sustainability of the background tasks. Even though BFLUSH and BGC almost recover the performance on write cliff, they sustain the recovered performance for very short time (just a few seconds). 124

6.1 Physical internal architecture of SSD. 133
6.2 Software stack of an SSD. 135
6.3 Average response times and bandwidth results as transfer sizes varies under two commercial SSDs. 139
6.4 Firmware layers within a solid state disk. 142
6.5 An example that shows PAQ conflict classification. CH, P and D denote channel, package, and die, respectively. 143
6.6 An example that shows PAQ conflict classification and the associated clumps. . 145
6.7 Plane packing in PAQ. 148
6.8 Average bandwidth comparison between VAQ and PAQ. 154
6.9 Average I/Os per second (IOPS) comparison between VAQ and PAQ. 154
6.10 Average latency comparison between VAQ and PAQ. 154
6.11 Idle times for VAQ and PAQ. 154
6.12 Normalized total contention time comparison between VAQ and PAQ. 154
6.13 All latencies incurred for requests from trace sql3 for VAQ and PAQ. Note that average latencies are shown using a horizontal line in each case. 157
6.14 IOPs sensitivity to varying queue and channel sizes. 158
6.15 Waiting time sensitivity to varying queue and channel sizes. 163

7.1 Overview and comparison of SSD latencies with/without our proposed GC strategies (AGC and DGC), tested by random write pattern with 2048KB request size. Note that AGC and DGC shift GC overheads to idle periods (as shown in the real view), thereby providing stable I/O performance like a pristine state (as shown in the user view). 172
7.2 Latency comparison for a random write access pattern with 1MB request size using a real MLC-based SSD. 176
7.3 System throughput for four state-of-the-art SSDs (different vendors and NAND types). Note that all SSDs tested suffer from significant performance degradation once garbage collections begin. 177
7.4 A high-level view of our proposed GC strategy and idle time utilization. 179
7.5 An example of the look-ahead GC with a hybrid mapping scheme. By inquiring the mapping information, our AGC scheme figures out that the GC for LBN 1 will be invoked soon. 183
7.6 Job lists for AGC and DGC. 184
7.7 Checkpointing for incremental GC. At each checkpoint, by checking the device-level queue, the garbage collector can decide whether it can perform further collections or not. 185
7.8 Performance of AGC with relatively low I/O intensive workloads (SSD 0 of 6SSDs-RAID). 193
7.9 Performance comparison of different garbage collection strategies (SSD5 of 6SSDs-RAID with high I/O intensive workloads). 195
7.10 Worst-case response time (WCRT) analysis for 6SSDs-RAID. (a) With low I/O intensive workloads, P-FTL and AGC+DGC show deterministic behaviors while the performances of L-FTL and H-FTL fluctuate over time. (b) With high I/O intensive workloads, P-FTL experiences very high WCRT, whereas AGC+DGC continues to provide stable I/O performance. 196
7.11 Response times for a write intensive section (where the fraction of I/O executions with no idle time is account for about 90%). While H-FTL removes about 40% of the GC related overheads, AGC+DGC hides all on-demand GC latencies. 197
7.12 Excess waiting time (EWT). The x-axis represents the upper bound on EWT. (a) L-FTL and H-FTL experience I/O blocking problem stemming from GCs while P-FTL and AGC+DGC have no such problem. (b) With heavy writes, even though P-FTL results in fewer GC invocations, its GC latencies are much longer than others 199
7.13 Worst-case response time (WCRT) analysis for the 3SSDs-RAID. 199
7.14 Excess waiting time (EWT) analysis for the 3SSDs-RAID. 199
7.15 (a) Garbage collection type breakdown of total collection. (b) Block erase impact by free block threshold. 201

Acknowledgments

Chapters 1 and 5, in part, are a reprint of the introduction as it appears in “Revisiting Widely-held Expectations of SSD and Rethinking Implications for Systems,” Myoungsoo Jung and Mahmut Kandemir, in Proceedings of the ACM international Conference on Measurements and Modeling of Computer Systems (SIGMETRICS), 2013. The dissertation author was the primary investigator and the first author of this paper.

Chapters 1 and 2, in part, are a reprint of the material as it appears in “NANDFlashSim: Intrinsic Latency Variation Aware NAND Flash Memory System Modeling and Simulation at Microarchitecture level,” Myoungsoo Jung, Ellis Herbert Wilson III, David Donofrio, John Shalf and Mahmut Kandemir, in Proceedings of the 28th IEEE Conference on Massive Data Storage (MSST), 2012. The dissertation author was the primary investigator and the first author of this paper.

Chapters 1, 2, and 3, in part, are a reprint of the material as it appears in “An Evaluation of Different Page Allocation Strategies on High-Speed SSDs,” Myoungsoo Jung and Mahmut Kandemir, in Proceedings of the 4th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage), 2012. The dissertation author was the primary investigator and the first author of this paper.

Chapters 1, and 4, in part, are a reprint of the material as it appears in “Challenges in Getting Flash Drives Closer to CPU,” Myoungsoo Jung and Mahmut Kandemir, in Proceedings of the 5th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage), 2013. The dissertation author was the primary investigator and the first author of this paper.

Chapters 1, 2, and 6, in part, are a reprint of the material as it appears in “Physically Addressed Queueing (PAQ): Improving Parallelism in Solid State Disks,” Myoungsoo Jung, Ellis Herbert Wilson III and Mahmut Kandemir, in Proceedings of the 39th International Symposium on Computer Architecture (ISCA), 2012. The dissertation author was the primary investigator and the first author of this paper.

Chapters 1, and 7, in part, are a reprint of the material as it appears in “Taking Garbage Collection Overheads off the Critical Path in SSDs,” Myoungsoo Jung, Ramya Prabhakar and Mahmut Kandemir, in Proceedings of the 13th ACM/IFIP/USENIX 13th International Conference on Middleware (Middleware), 2012. The dissertation author was the primary investigator and the first author of this paper.

Chapter 1

Introduction

Non-Volatile Memory (NVM) based Solid State Disks (SSDs) have recently become immensely popular and been employed in different types of environments ranging from embedded systems to personal computers to high performance computing (HPC) systems. Moreover, various memory and storage systems have been proposed to take advantage of the performance benefits of SSDs over conventional block devices. For example, to reap up the benefits of high bandwidth on writes, prior HPC studies consider SSDs as a burst buffer [68], which can absorb heavy write traffic caused by check-pointing [83]. There also exist many applications developed under the expectation that NVM is biased toward reads in terms of performance and reliability. Enterprise servers, for example, consider employing SSDs for applications that exhibit many random reads [69, 92, 95] or use them as read caches [54, 7, 61, 82], sitting between main memory and hard disk drive (HDD). Similarly, SSDs are also introduced as a main memory replacement, memory extension, and a part of existing virtual memory systems [24, 23, 25, 92].

While many SSD applications and usage scenarios have been proposed and developed by prior research, modern SSDs and MVM systems have undergone severe technology shift and architectural changes. For instance, NAND flash cells have shrunk from 5x *n*m to 2x *n*m in the past four years, and now fewer electrons are stored per floating gate. These cell-level characteristics make NAND flash devices less reliable and introduce extra operations (e.g., multi-step I/O, verification, error correction processes) to successfully complete I/O requests, which in turn imposes longer latencies. State-of-the-art NAND flash packaging technologies employ an increased number of planes and dies within a single flash chip, a command queue, ECC engines, and faster data movement interfaces [17, 81]. In parallel, SSD internal architecture has dramatically changed; modern SSDs now employ multiple internal resources such as flash chips, I/O buses, and cores in an attempt to achieve high internal parallelism. In addition, to reduce garbage collection overheads, NVM firmware employs advanced strategies such as finer-granular address mapping algorithm. As a result of complex interactions among these different mechanisms, modern SSDs can be plagued by enormous performance variations depending on whether underlying architectural complexities and NVM firmware overheads can be hidden or not.

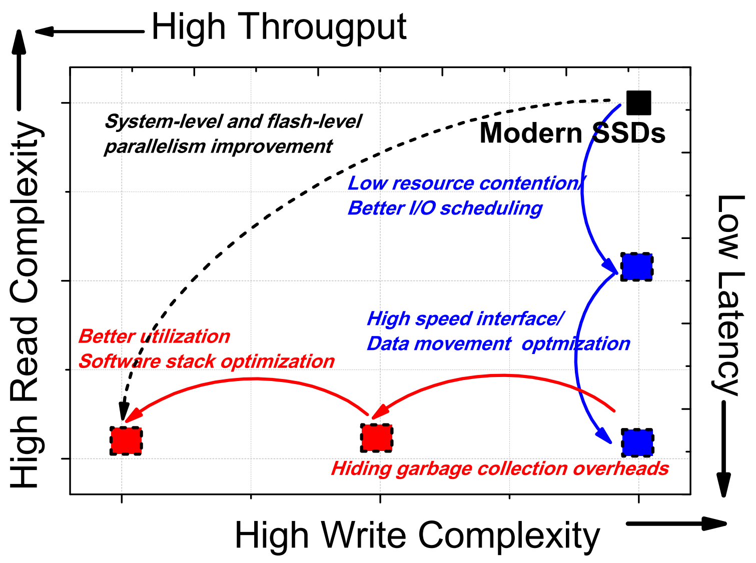

High Througput
High Read Complexity
Low Latency
High Write Complexity
Modern SSDs
System-level and flash-level parallelism improvement
Low resource contention/ Better I/O scheduling
High speed interface/ Data movement optmization
Better utilization
Software stack optimization
Hiding garbage collection overheads

Fig. 1.1 Design and optimization goal of a smart NVM controller. Modern SSDs can be plagued by enormous performance variations depending on whether underlying architectural complexities and NVM firmware overheads can be hidden or not. Designing a smart NVM controller is key hiding the architectural complexities and reducing the internal firmware overheads.

As shown in Figure 1.1, designing a smart NVM controller is key to hiding the architectural complexities and reducing the internal firmware overheads; hiding complexity of writes, such as garbage collection overheads and low resource utilization, is crucial for achieving higher throughputs, and alleviating complexity on reads such data movement overheads and resource contention is key to offering lower latencies of modern SSDs. To this end, we first model a multi-plane and multi-die NVM architecture, which is highly reconfigurable and aware of intrinsic latency variation imposed by various state-of-the-art NVM systems. This NVM model has been implemented as a high fidelity open-source simulator, capable of capturing cycle-level interactions between the many components in an SSD, which can be used for various NVM performance analyses. Based on this architecture model, we then explore twenty four different concurrency methods implemented in NVM controllers, geared toward exploiting both system-level and NVM-level parallelism with a variety of design parameters. Further, we quantitatively analyze the challenges faced by PCI Express-based (PCIe) SSDs in getting NVM closer to CPU, conduct a comprehensive data analysis and uncover critical storage-class SSD/flash characteristics, which are not reported, to the best of our knowledge, in the literature so far, and are opposite to the widely held expectations on SSDs.

In this dissertation, we also perform a comprehensive set of experiments using diverse commercial SSDs (e.g., drawn from different SSD makers, NVM types, SSD internal architectures) and analyze them in an attempt to show practical significance of read performance degradations and write performance variations. Specifically, in cases where read access patterns are mainly random, the read performance can be worse than that of writes due to architectural complexities; internal resource conflicts and contentions occur among multiple incoming I/O requests, which can in turn introduce long system-level I/O pending times. In contrast, write performance significantly varies based on patterns exhibited by garbage collections, which are an essential device-level activity, performed by an NVM firmware module in SSDs. The biggest challenge with existing garbage collectors is that their worst-case latency can be $64 \sim 128$ times higher than that of normal write operations. Our experiments show that garbage collectors introduce numerous blocking I/Os, and once a garbage collection operation begins, the response time of write operations on SSD increases substantially. Further, garbage collection overheads significantly reduce available bandwidth in most recent commercial SSDs.

To address these challenges behind reads and writes that state-of-the-art SSDs face, we propose two different types of schedulers, which can be implemented in NVM controllers: (1) Physical Address Queuing (PAQ) scheduler (tacking the read performance degradation issue) and (2) NVM garbage collection scheduling algorithm (geared toward addressing the write performance variation issue). Specifically, PAQ is a novel I/O request scheduler that avoids resource contention resulting from shared SSD resources. Our proposed PAQ dramatically improves read performance by exposing the physical addresses of requests to the SSD controllers/schedulers and selecting groups of I/O operations that can be simultaneously executed without major resource conflict. In parallel, our proposed NVM garbage collection scheduler that removes garbage collection overhead and provides stable I/O performance in SSDs during the I/O congestion periods. Our proposed garbage collection scheduler consists of two garbage collection strategies, namely, Advanced Garbage Collection (AGC) and Delayed Garbage Collection (DGC). AGC tries to secure free blocks and removes on-demand garbage collections from the critical path in advance, so that users do not experience garbage-collection-induced latencies during the I/O-intensive periods, whereas DGC handles the collections that AGC could not handle, by delaying them to future idle periods. While the PAQ scheduler mainly targets to improve read performance by avoiding resource conflicts, this garbage collection scheduler alleviates the write amplification factors posed by modern SSDs, thereby improving write performances and making them stable.

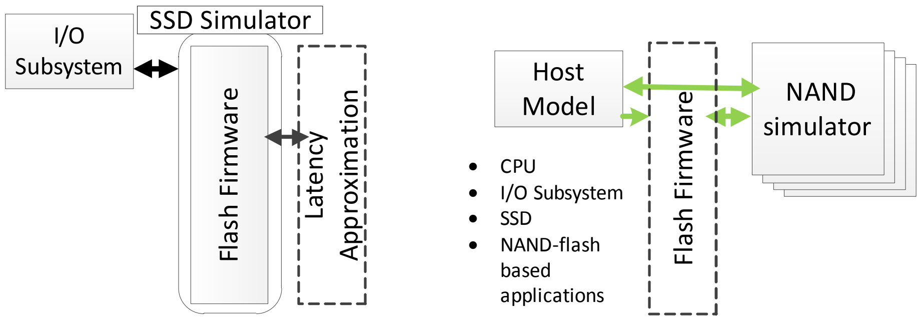
The rest of this dissertation proposal is organized as follows. In Chapter 2, we model a modern multi-die and multi-plane NAND flash architecture at a cycle-level. In Chapter 3, we study twenty four concurrency methods to understand details of internal parallelism. In Chapter 4, we quantitatively characterize the challenges faced by PCI Express-based (PCIe) SSDs in getting NVM closer to CPU, and in Chapter 5, we question popular assumptions and expectations regarding storage-class SSDs through an extensive experimental analysis. We then analyze the performance degradation in reads using different experiments and propose a novel I/O request scheduling strategy (PAQ) with the goal of removing this performance degradation in Chapter6. In addition, we show the significance of garbage collection overheads in modern SSDs and propose a garbage collection scheduler in order to make garbage collection overheads invisible to users by exploiting device-level idleness in Chapter 7. We finalize by discussing potential near-term and long-term future directions.

Chapter 2

NAND Flash Memory System Modeling and Simulation

2.1 Introduction

While processors have enjoyed doubled performance every 18 months, and main memory performance increases roughly 7% in the same time frame, non-volatile storage media has been stuck at a standstill for nearly a decade [86]. Many efforts have been made to remedy such a great gap, and NAND flash has positioned itself at the forefront of such efforts. Since NAND flash is hundreds to thousands of times faster than conventional storage media and has a small form factor, it has been employed in the construction of devices such as Solid State Disks (SSD), Compact Flash, and Flash Cards. NAND flash density is increasing by two to four times every two years [66], which is in turn decreasing its cost and enabling wide-spread deployment in arenas as diverse as embedded systems and high-performance computing. In addition, by introducing multiple planes and dies, the NAND flash memory is expected to continue in this trend as it experiences the same ease of scaling multiprocessors currently enjoy. As a result of such proliferation, performance, energy consumption, and reliability of NAND flash memory are becoming increasingly important [100]. Further, this proliferation also results in a diversification of target system configurations, such as additional Flash Translation Layer (FTL) logic positioned atop flash, which is often tailored to the demands of various NAND-flash based applications. However, because NAND flash is very sensitive to a large number of parameters, and some latency parameters fluctuate between best-case and worst-case [27], deciding on an optimal NAND flash memory system's configuration is non-trivial. Furthermore, NAND flash memory can have many different parameters depending on what memory system types are employed (e.g., single level cells (SLC), multi level cells (MLC), diverse node technologies (fabrication processes), page sizes, register management policy, memory density, and pin configurations). Consequently, this large parameter space and sensitivity of NAND flash to such parameters results in memory systems exhibiting significantly different behaviors.



(a) Existing SSD simulations (b) The proposed NAND simulation

Fig. 2.1 Concept of a μarch-level NAND flash simulation model (NANDFlashSim). While existing SSD simulators are highly coupled to flash firmware emulation with simplified latency model, NANDFlashSim simulates NAND flash memory system itself with independently synchronous clock domains and detailed NAND operation timing models aware of latency variation.

Unfortunately, a comparison between different types of NAND flash memory systems becomes even harder when multi-die and multi-plane architectures are considered [22]. In these architectures, scheduling methods and arbitration [85] among multiple dies and planes are important factors in determining I/O performance [22]. However, incorporation of these methods and arbiters results in a greatly increased complexity of flash firmware and controllers. Even though simulation-based prior research [32, 60, 70, 1] reveals performance tradeoffs in an application level, the main focus of such studies has been on SSD rather than the NAND flash memory system itself; this difference is pictorially shown in Figure 2.1(a). Such simulations make several assumptions that ignore, to varying extents, the fluctuating timing behaviors of the diverse I/O operations supported by state-of-the-art NAND flash memory. These assumptions range from extremely widespread, where the SSD is modelled as having constant time and energy per I/O request, to more confined but still overly simplified, where dies and planes are modelled but the interactions between various NAND commands and components are still represented with *constants*. This implies that the existing simulation models used in those prior studies are strongly coupled to particular flash firmware and policies – performing the exact same study using slightly different firmware or policy-set has the potential to result in wildly different performances and conclusions. Using such imprecise timing models of NAND flash and NAND operations, hardware and system designers may overlook opportunities to improve memory system performance. Furthermore, as shown in Figure 2.2, since such prior studies are oblivious of *intrinsic latency variation* of NAND flash, they are not be able to properly model diverse node technologies. Also, simplified latency models ignore the substantial contributions of the flash firmware to memory system performance. This may result in these designers overlooking research potential regarding new algorithms in/on NAND flash memory systems, such as those involved in internal parallelism handling, wear-leveling, garbage collection, the flash translation layer, flash-aware file systems, flash controllers, and so on.

To address heretofore mentioned drawbacks, the introduction of a microarchitecture (μarch) level NAND flash memory system simulation model that is decoupled from specific flash firmware and supports detailed NAND operations with cycle-accuracy is required. This

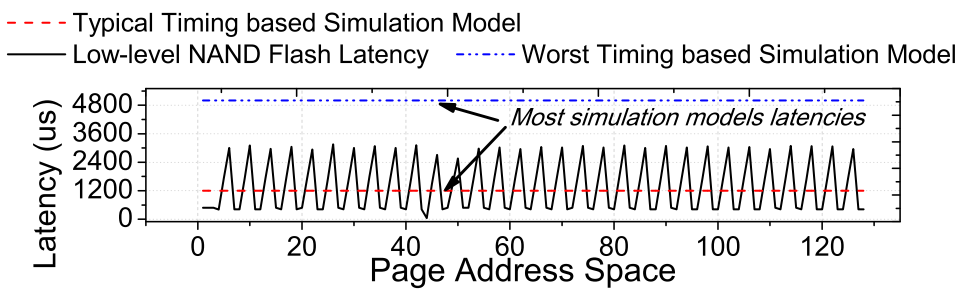


Fig. 2.2 Intrinsic Latency Variation Example.

low-level simulation model can enable research on the NAND flash memory system itself as well as many NAND flash-based devices, as illustrated in Figure 2.1(b). Specifically, in this Chapter, we propose *NANDFlashSim*; a latency variation-aware, detailed and highly reconfigurable μarch-level NAND flash memory system based on multi-die and multi-plane architectures. To the best of our knowledge, NANDFlashSim is the first simulation model to target the NAND flash memory system itself at μarch-level, and the first model to provide sixteen latency variation-aware NAND flash operations with NAND command set architecture.

From our comprehensive experiments using NANDFlashSim, we found that 1) most read cases were unable to leverage the highly-parallel internal architecture of NAND flash regardless of the NAND flash operation mode. Specifically, the read throughputs improvements between quad dies and octal dies, between four-plane and eight-plane, and between 4KB and 8KB page size are 10.9%, 10.8%, and 10.9%, respectively, while the write throughputs are improved by 91.2% on average. 2) the main contributor of performance bottleneck is I/O bus activity, not NAND flash activity itself. 50.5% cycles of total I/O execution cycles are consumed by operations related to such I/O bus activity. The bottleneck is more problematic when *advance NAND flash commands* (e.g., cache and multi-plane mode) are applied. 3) MLC NAND flash provides lower I/O bus resource contention than SLC NAND flash, but such resource contention becomes a serious problem as the number of dies increases, and 4) preference of employing many dies rather than employing many planes provides average 54.5% better performance in terms of throughput in disk-friendly workloads [88]. This Chapter makes the following main **contributions**:

- *Detailed Timing Model:* NANDFlashSim presents a μarch-level flash simulation model for many NAND flash-based applications. The memory system, controller and NAND flash memory cells have independent synchronous clock domains. In addition, by employing multi-stage operations and command chains for each die, NANDFlashSim provides a set of timing details for a large array of NAND flash operation modes including: legacy mode, cache mode, internal data move mode, multi-plane mode, multi-plane cache mode, interleaved-die mode, interleaved-die cache mode, interleaved-die multi-plane mode, and interleaved-die multi-plane cache mode. These detailed NAND operation modes and their associated timings expose performance optimization points to NAND flash-based application designers and developers.

- *Intrinsic Latency Variation-Aware Simulation Model:* NAND flash memory, especially MLC, suffers from intrinsic performance variations when accessing a block. In our observations, write latency of [75] and [34] varies between 250μs to 2,200μs and 440μs to 5,000μs, respectively (maximum latencies are 8.8 $\sim$ 11.3 times higher than minimum latencies). Therefore, NANDFlashSim, a cycle-accurate simulation model, is designed to be performance variation-aware and employs different page offsets in a physical block. To collect statistics related to the performance variation and validate our simulation model accuracy, we prototype a NAND flash hardware platform, called Memory Statistic Information System (MSIS). We present a comprehensive evaluation considering different types of NAND flash and NAND operation on both NANDFlashSim and MSIS.

• *Reconfigurable Micoarchitecture:* NANDFlashSim supports highly reconfigurable architectures in terms of multiple dies and planes. This architecture allows a researcher to explore true internal parallelism in such an architecture by exposing the intrinsic latency variations in NAND flash. In contrast to prior simulation models, NANDFlashSim removes the dependency on a particular flash firmware, which enables memory system designers and architects to develop and optimize diverse algorithms targeting NAND flash such as buffer replacement algorithms, wear-leveling algorithms, flash file systems, flash translation layers, and I/O schedulers.

2.2 NAND Flash Microarchitecture

Figure 2.3(a) illustrates the NAND flash microarchitecture [75], and Figure 2.3(b) depicts the physical NAND memory cell microphotograph [94]. Energy consumption and interface complexity are important factors in NAND flash memory system design. Therefore, interfaces for data, commands, and addresses are multiplexed onto the same I/O bus, which helps to reduce pin counts, interface complexity, and energy consumption [75]. Because of this, a host model must first inform the NAND flash package that it wishes to use the I/O bus through control logic before acquiring it. This information is provided via control signals like *command latch enable* (CLE) and *address latch enable* (ALE). Similarly, NAND commands are responsible for signalling usage of the I/O bus in addition to classifying following NAND operation types.

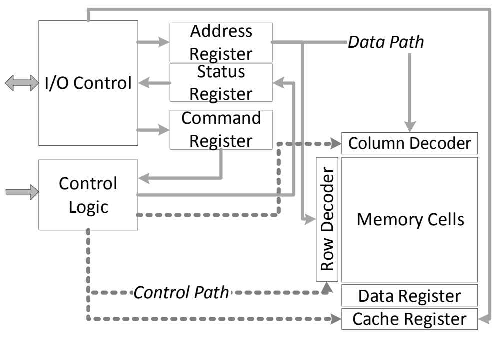
• *Page/Block.* A page is a set of NAND memory cells, and a block is a set of pages (typically 32$\sim$256 pages). A physical NAND block makes up a plane.

• *Register.* Registers are adopted to provide collection and buffering for delayed write-back of small writes and to fill the performance gap between the NAND flash interface and flash memory cells. Supporting multiple registers is a common trend to boost NAND flash memory performance. NAND flash is typically composed of a set of cache and data registers.

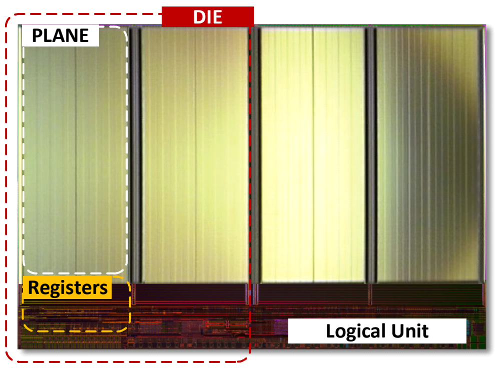
• *Plane.* A plane is the smallest unit that serves an I/O request in a parallel fashion. In practice, physical planes share one or more word-lines for accessing NAND flash cells, which enables the memory system to serve multiple I/O requests simultaneously [65].

• *Die.* A die contains an even number of planes and constitutes a NAND flash package. Depending on how many dies are placed into a package, a NAND flash memory is classified as a single die package (SDP), dual die package (DDP), quad die package (QDP), or octal die package (ODP).

• *Logical Unit.* A logical unit consists of multiple dies, and is the minimum unit that can independently execute commands and report its status. Multiple dies in a logical unit are interlaced by a chip enable (CE) pin, leading to a reduction in I/O bus complexity and total pin counts.



(a) NAND flash microarchitecture



(b) Die microphotograph [94]

Fig. 2.3 NAND flash memory system internals.

Since the number of dies sharing the I/O bus and CE pins is determined at packaging time, different numbers of logical units are used in DDP, QDP and ODP. Although state-of-the-art NAND flash provides at most four planes [94] and eight dies, our proposed simulation model can be configured to simulate a much larger number of planes and dies in a logical unit.

2.3 NAND Flash Operations

Legacy NAND operations can be classified into three types: *read*, *write* (also referred to as program), and *erase*. While reads and writes operate at a *page* granularity, erase operation executes on an entire block. To operate NAND flash memory, the first task is to load a command into the command register by raising the CLE signal, which informs what operation will be executed. After that, a start address for the operations is loaded into an internal address register by raising the ALE signal. Once the address is loaded, the NAND operation can be issued by loading the initiate command. Each of the NAND operations has different timings for data movement. For reads, a page of data is loaded from specific NAND memory cells into the data register. This data movement stage is called *transfer-out of NAND memory cells* (TON). Then, data are sequentially output from the register, byte by byte, which is a process termed *transfer-out of register* (TOR). In the case of a write operation, after the address is loaded, the data can be stored in the data register. This data movement stage, called *transfer-in of register* (TIR), should be processed before loading the initiate NAND command. Following TIR, the NAND flash memory system starts to write data from the register to NAND memory cells, called *transfer-in of NAND cell* (TIN) stage. In addition to these basic operations, state-of-the-art NAND flash memories support more complex operations to improve system performance [22]. Below, we explain different I/O modes, which are used in concert with these legacy commands.

2.3.1 Cache Mode Operation

In *cache mode* operation, data are first transferred to a cache register and then copied to a data register. After that the NAND flash memory system enters the TIN stage. In the meantime, the memory system is available again for TIR stage operations using the cache register because only the data register and memory cells are involved in writing. This cache mode operation overlaps the process of putting data into register and that of writing data into the NAND memory cells, thereby hiding the TIR time. Just like writes, read operations can also take advantage of the cache register. However, in our observations, cache mode operations demonstrate slightly different performances between reads and writes. This is due to the latency-dominating NAND operation differing between, which will be further discussed in Section 2.8.

2.3.2 Internal Data Move Mode Operation

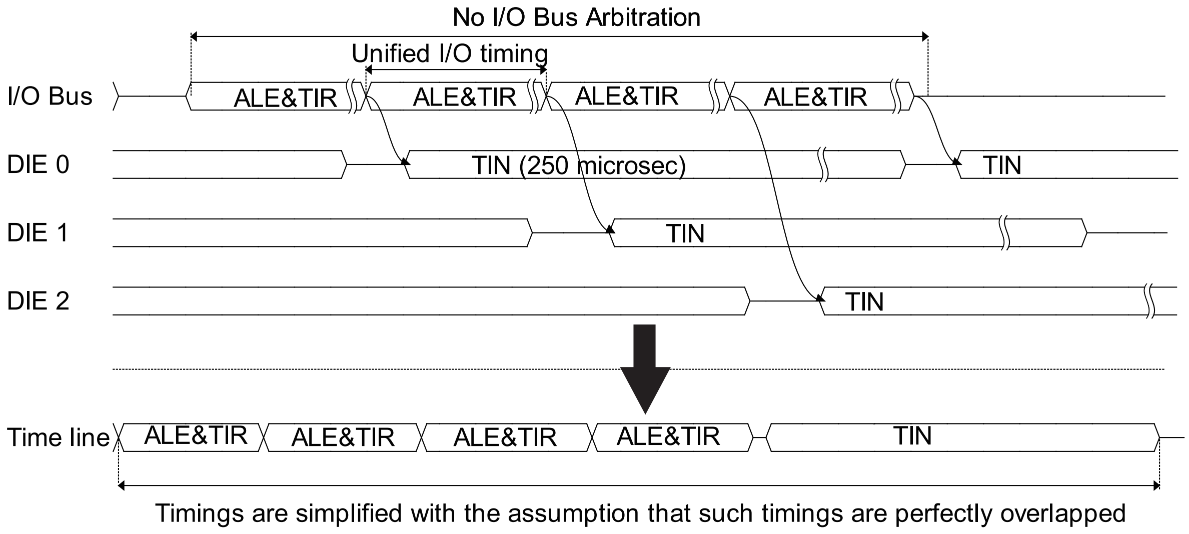
Flash applications may need to copy data from a source page to a destination page on the same NAND flash memory. Since data movement from one location to another within flash memory requires external memory space and cycles, a data copy is suprisingly expensive and time consuming. To reduce the time required to copy data, state-of-the-art NAND flash memory support *internal data move* operations. In these operations, a host is not involved in managing the data copy process. Instead, the host only has to load the source and destination address for copying data into the address registers, and commit the internal data move mode NAND command. Then, the NAND flash memory reads data from the source using the data register and directly writes it to its destination, without any data transfer involving the host model. That is, in internal data movement operation mode, data in one page of NAND memory destined for another page can be copied without any external memory cycles. This specialized operation alleviates the overheads of data copying, which notably results in greatly enhanced garbage collection performance [11], a critical task of flash firmware.

2.3.3 Multi-plane Mode Operation

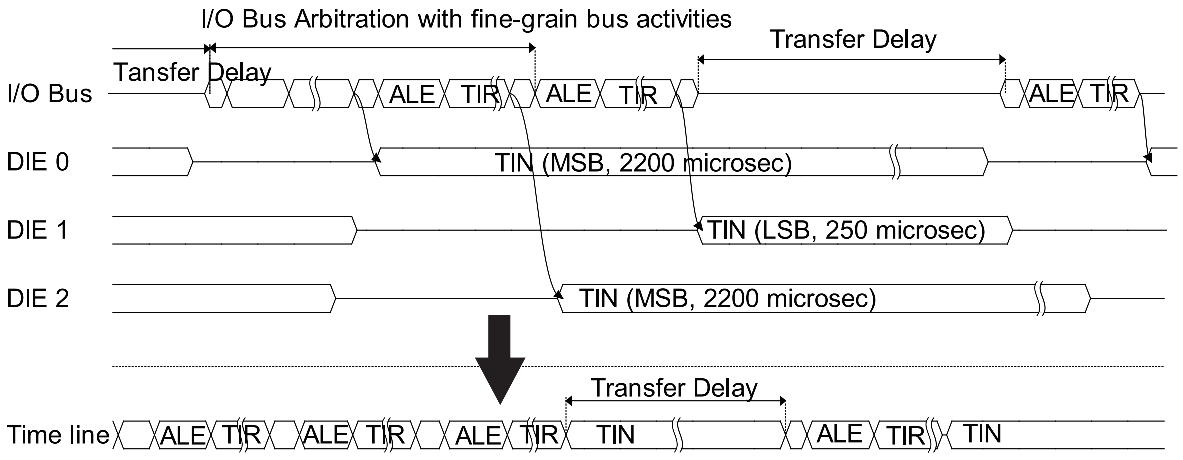
Multi-plane mode operations serve I/O requests using several planes at a time that are connected by word-line(s). Specifically, these operations can enhance performance up to *n* times, where *n* is the number of planes in a word-line. However, the multi-plane architecture carries with it limitations for addressing planes. Specifically, in multi-plane mode operations, I/O requests should indicate the same page offset in a block, same die address, and should have different plane addresses [75, 80]. These constraints are collectively referred to as the *plane addressing rule*. Therefore, performance enhancement using a multi-plane architecture may be limited based on user access patterns (we will discuss this issue in Section 2.8.5). Regulating plane addressing rules is required to obtain realistic performance with the multi-plane mode of operation. Using such rules, NANDFlashSim provides an accurate implementation of multi-plane mode operations, which may be used in any combination with other NAND flash operations.

2.3.4 Interleaved Die Mode Operation

State-of-the-art flash memory share between one and four I/O buses among multiple dies in order to reduce the number of pins. While sharing the I/O bus reduces energy consumption and complexity, I/O bandwidth of the system also reduces. This is because all NAND operations except those related to NAND memory cells (e.g., TON, TIN) should acquire permission to use the I/O bus before they start executing. Thus, efficient bus arbitration and NAND command



(a) A typical-case timing parameter based simulation model



(b) Latency-aware NANDFlashSim

Fig. 2.4 A timing diagram of interleaved die with four legacy writes. While an existing simulation model simplifies bus activities and assumes that latencies are perfectly overlapped and interleaved with constant time, NANDFlashSim employs fine-grain bus activities and is aware of intrinsic latency variations.

scheduling policies are critical determinants of memory system performance. *Interleaved die mode* operations provide a way to share the I/O bus and take advantage of internal parallelism by interleaving NAND operations among multiple dies. Unlike multi-plane mode operations, interleaved-die mode operations have no addressing restrictions.

It should be noted that all NAND operations discussed above can be used in any combination with interleaved-die operations. For example, a host model can issue an interleaved-die multi-plane mode operation, which stripes a set of multi-plane mode operations across multiple dies. Similarly, interleaved-die multi-plane cache mode operations are possible, which are operations that have the properties of operating in cache mode, being striped over multiple dies and being applied to multiple planes. A simplified and approximated latency circulation model with constant times is unable to capture the behavior of and interactions between these different types of operations. Furthermore, intrinsic latency variations exhibited by the NAND flash make it difficult for a latency model with constant time to mimic elaborate bus arbitration or scheduling NAND commands.

Consider the comparison, shown in Figure 2.4, between the existing simulation model (with constant time) and variation-aware NANDFlashSim. In the figure, four I/O requests are striped over three dies with interleaved-die legacy write mode. Existing simulation models will calculate the latency under the assumption that *timings are perfectly overlapped and interleaved*. Let t_{io} denote execution time for I/O activities, and t_{prog} denote programing (write) time. Suppose that n_{io} denotes the number of the write requests, and $t^{interleaved}_{resp\_legacy}$ denotes the response time for n_{io} requests, In existing simulation models, $t^{interleaved}_{resp\_legacy}$ is simply calculated by $n_{io} * t_{io} + t_{prog}$ as shown in the time line of Figure 2.4(a). However, in practice, $t^{interleaved}_{resp\_legacy}$ varies significantly based on system configurations. This is because t_{prog} fluctuates based on the access address and the transfer delay time is also varied by the service order. In contrast, NANDFlashSim is aware of latency variation and provides a method for scheduling NAND commands and activities with fine granularity.

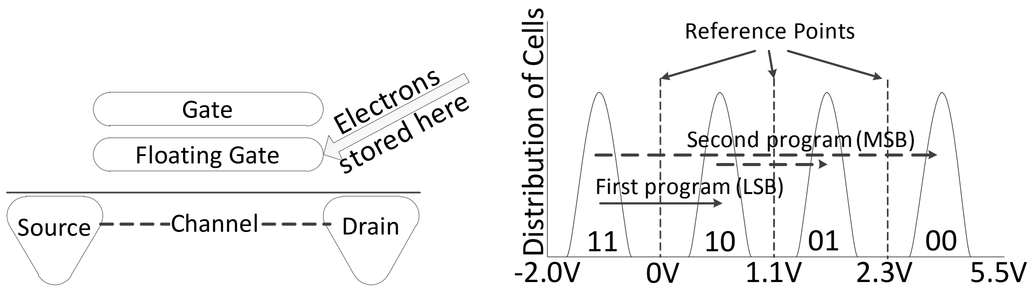
2.4 Intrinsic Latency Variation of NAND Flash

NAND flash memory has the interesting characteristic of performance variation [27, 63, 65], which results in the latencies of the NAND flash memory system to fluctuate significantly depending on the address of the pages in a block. Typically, this variation is not specified in the datasheets of NAND flash memory. NAND flash memory puts electrons, which represents cell states, into a NAND flash floating gate. To achieve this, NAND flash memory selects the NAND flash memory cells, and makes an electron channel between a source and drain (see Figure 2.5(a)). Once the channel is built and voltage is applied over a certain threshold voltage, electrons can be moved from the channel to the floating gate. This process is called *Fowler-Nordheim tunneling* (*FN-tunneling*) [65], which is a well-known programming (write) operation. As illustrated in Figure 2.5(b), based on differing cell distributions, a MLC NAND flash memory system can identify bit states like '11', '10', '00' and '01' in a cell[1]. According to the specific bit states for programming, therefore, a MLC NAND flash memory system will end up spending different amounts of time and power. Specifically, MLC NAND flash is able to store multiple bits on a cell using *incremental step pulse programming* (*ISPP)* [63, 65].

For example, in the first phase, MLC NAND flash programs a cell from '11' to '10' or '11' state. This phase represents the least significant bit (LSB) of an MLC cell. In the second phase, the NAND flash reprograms the cell from the '11' or '10' state to a '01'/'11' or

[1]The '0' bit in a NAND flash cell represents programed (written) state.

’00’/’10’ state, respectively, so that the memory cell represents the most significant bit (MSB). Since MLC devices utilize four states using this ISPP, FN-tunneling for MSB pages requires more energy and takes a longer time when compared to LSB pages [27, 64, 90]. Due to these NAND flash memory characteristics, one may observe performance variations between worst-case and typical-case programming time parameters. Since there is no need for ISPP to a specific cell in SLC flash, this latency variation characteristic is more pronounced in MLC NAND flash memory.



(a) NAND transistor cell (b) Example of cell distribution

Fig. 2.5 NAND flash memory cell organization. MLC NAND flash memory has multiple states in a cell, which causes intrinsic latency variation [65].

2.5 Related Work

There are several prior studies for simulating a NAND flash-based SSD. The SSD add-on [1] to DiskSim [5] is a popular simulator that models idealized flash firmware. FlashSim [60] is another simulator, implemented using object-oriented code for programmatic ease and extensibility. This simulator supports several types of flash software algorithms. While these simulation models compute performance by calculating latency for each of the basic NAND operations, SSDSim [32] accommodates latency calculation models for cache, multi-plane, interleaved-die operations of SLC devices at application-level.

Even though these simulation models can enable researchers to explore the design trade-offs of SSDs, they have limitations in simulating the μarch-level NAND flash memory since they highly simplify NAND flash characteristics, latencies, and energies from a flash firmware perspective. Also, *these studies are appropriate to simulate only SLC NAND flash type*.

•**Unaware of latency variations.** These existing simulation models are ignorant of NAND flash memory's latency variations; they implement the flash memory system based on constant times and energies of worst-case or typical-case time parameters. However, as mentioned in Section 2.3, the state-of-the-art memory systems are very complex and support diverse NAND I/O operations for high performance I/O, which results in latency varying immensely even between executions of operations of the same type. In contrast, our proposed NANDFlashSim is aware of the latency variations based on most significant bit (MSB) and least significant bit (LSB) page addresses in a block and provides legacy mode operations as well as a number of state-of-the-art modes for more complex operations at μarch-level. As consequence, NANDFlashSim is able to simulate both *MLC and SLC NAND flash*.

•**Coarse-grain NAND command handling.** Moreover, these past studies mimic multi-die and multi-plane architecture using coarse-grain I/O operations, which means that NAND operation and control are simplified by host-level I/O requests. Even though they consider basic I/O timing based on time parameter statistics and internal parallelism of NAND flash memory, the evaluation of accurate memory system latencies is non-trivial. Using multi-stage and command chains for each of the NAND flash operations , our proposed NANDFlashSim, reconfigurable for multi-die and plane architectures, provides detailed timing models for NAND operations and manages bus arbitration based on different latencies at μarch-level.

•**Weak model of NAND flash memory constraints.** The memory system's performance and energy consumption can exhibit a variety of patterns due to NAND flash memory constraints. For example, as mentioned in Section 2.3, multi-plane I/O operations should satisfy plane addressing rules. This constraint results in different performance characteristics depending on I/O patterns. Even though the past studies consider these kinds of constraints, their simulation is *tightly coupled with specific firmware*. This problem makes it very difficult to explore new memory systems that can be built using NAND flash memory. As opposed to these prior efforts, our NANDFlashSim regulates NAND flash memory constraints in μarch-level, and is not tied to any specific flash firmware, algorithm or NAND flash applications like SSDs.

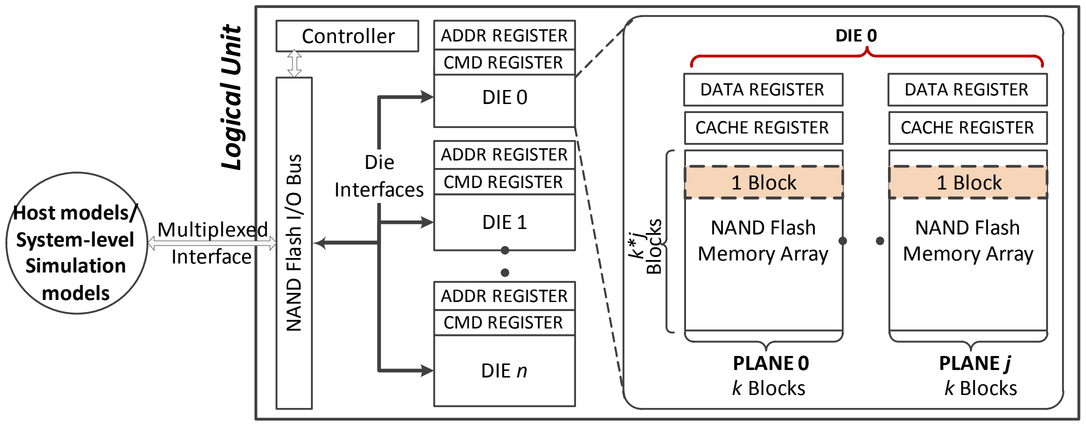


Fig. 2.6 NANDFlashSim architecture. NANDFlashSim is a reconfigurable μ-level multi-plane and multi-die architecture. The number of registers, blocks, planes and dies can be reorganized, and each entity has independent synchronized clock domain.

2.6 High Level View Of NANDFlashSim

NANDFlashSim employs a highly reconfigurable and detailed timing model for various state-of-the-art NAND flash memory systems. NANDFlashSim removes the specific flash firmware and algorithm from the NAND flash simulation model so that memory system designers and architects can employ NAND flash memory systems for various NAND flash-based applications and research/develop flash software for their specific purposes. To achieve its design goals, instead of employing underlying simplified latency calculation models, NANDFlashSim uses a NAND command set architecture and individual state machines associated to the command sets, which results in independent synchronous clock domains. These mechanisms enable designers and architects to closely study the NAND flash performance and optimization points at a cycle-level by exposing the details of NAND flash.

2.6.1 Software Architecture

Figure 2.6 illustrates the software architecture of our proposed simulation model. NANDFlashSim is comprised of a logical unit, NAND flash I/O bus model, several registers, a controller module, die modules, plane modules, and virtual NAND blocks. A host model can issue any type of NAND flash operations through the NAND I/O bus when the memory system is not busy. NANDFlashSim provides two interfaces to manage NAND flash memory. The first is a *low-level command interface*, which is compliant with Open NAND Flash Interface (ONFI) [80]. In this case, the host model fully handles the set of NAND commands for addressing, moving data, and operating NAND flash memory cells. Since a wrong command or inappropriate NAND command sequence can make the NAND memory system malfunction, NANDFlashSim verifies the correctness of command uses by checking the command/address registers and its own state machines every cycle. If it detects a wrong command sequence, it enforces a system fail and notifies the host model. The host model is able to identify the type of failure that occurred using read-status commands or return codes. The second is a *memory transaction based interface*. In this case, the host model is not required to manage the set of NAND commands, command sequences, or data movement. Figure 2.7 visualizes how NANDFlashSim supports such interface logic. When the logical unit of NANDFlashSim receives a request from the host model, it creates a *memory transaction* (discussed in the next subsection), which is a data structure that includes the command, address, and data. It then places the memory transaction into the internal NAND I/O bus. Once the controller module detects a memory transaction on the NAND flash I/O bus, it starts to handle the command sequence based on the command chain associated with the memory transaction. Note that this is handled by NANDFlashSim instead of the host model. In the meantime, the logical unit arbitrates NAND flash internal resources (e.g., the NAND I/O buses) and also manages I/O requests across multiple dies and planes. The set of NAND commands generated by the command chain handles the command/address latch and data movement processes such as TOR, TIR, TON, and TIN, called *stages* (we will discuss this shortly). It should be noted that using these two interfaces, other simulator models can be easily integrated into NANDFlashSim.

2.6.2 Clock Domains and Lifetime of Transaction

Our simulation model assumes that the logical unit, controller, die, and plane form a module working as an independently-clocked synchronous state machine. Many such state machines can be executed on separate clock domains. In general, there are two separate clock

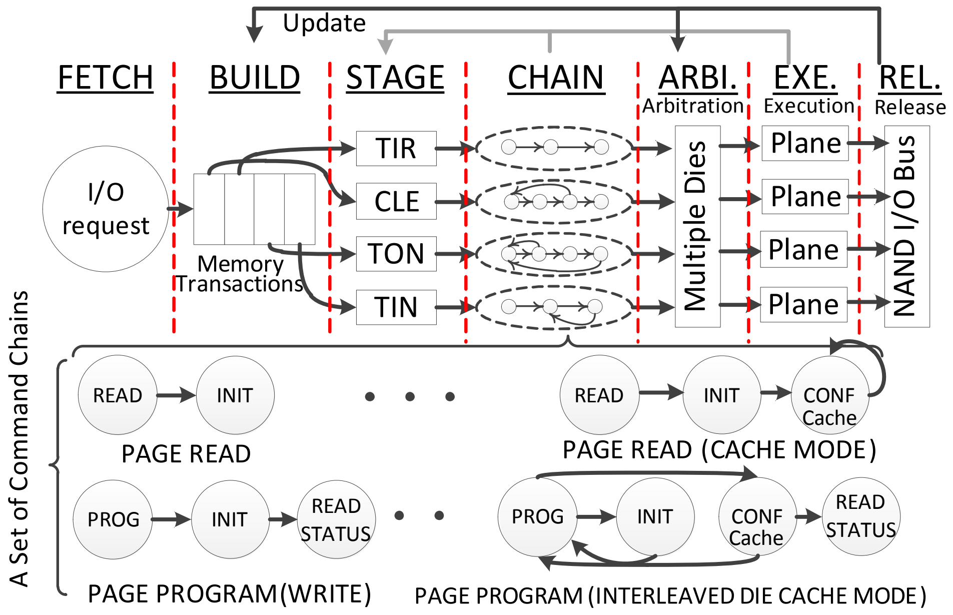


Fig. 2.7 The process of NAND flash memory transactions and examples of NAND command chains. NANDFlashSim handles fine-grain NAND transactions by NAND command set architecture

domains: 1) the host clock domain, and 2) NAND flash memory system's clock domain. The entities of NANDFlashSim are updated at every clock cycle, and the transaction lives until either getting an I/O completion notification or NAND flash memory requires a system reset due to an I/O failure. Since the time for a NAND operation can vary from a few cycles to a million cycles, updating all components (e.g., planes, dies, and I/O bus) of NANDFlashSim using the default update interface at every clock can be expensive and ineffective. Therefore, in addition to the default update interface NANDFlashSim also supports a mechanism to skip cycles not worth simulating. In this mechanism, NANDFlashSim looks over all modules in the logical unit, and then finds out the minimum clock cycles to reach the next state among them at a given point. NANDFlashSim updates system clocks for its own components based on the detected minimum clock cycles, thereby skipping the meaningless cycles in the update process.

2.7 Implementation Details

2.7.1 NAND Command Set Architecture

The number of combinations of operations possible with a state-of-the-art NAND flash memory is as high as sixteen, and each combination has varying timing behaviors. Therefore, NANDFlashSim divides a NAND I/O request into several multiple NAND command sets based on the information specified by ONFI and updates them at every cycle (as a default). To appropriately operate the NAND flash memory system, this NAND command set architecture is managed by *multi-stage operations* and *command chains*, as described next.

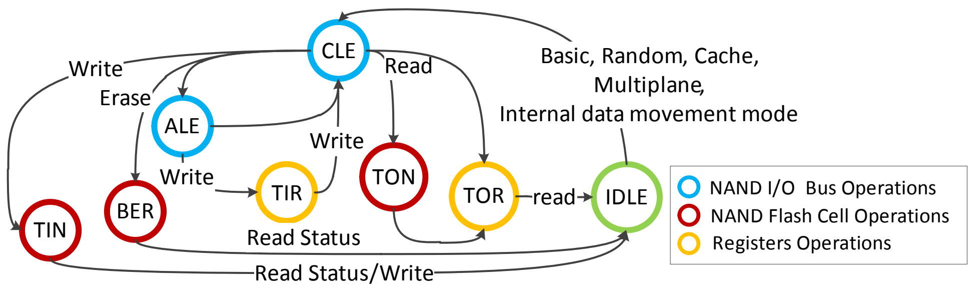


Fig. 2.8 State machine for multiple NAND stages. Each state is identified by different type of stages and states of the machine are transited by different type of NAND commands. Since each die has their own state machine, NANDFlashSim provides an independent clock cycle domain per die.

Multi-stage Operations. Stages are defined by common operations that NAND flash has to serve. Specifically, all types of μarch-level NAND operations should have at least one stage, which are classified by CLE, ALE, TIR, TOR, TIN, TON, and BER. CLE is a stage for a command by following command latch enable signal, and ALE a stage in which an address is loaded into an address register, which is triggered by address latch enable. BER is a stage for erasing block(s). Other stage names that NANDFlashSim employs are the same as the name described earlier in Section 2.2. These stages comprise an independently clocked synchronous state machine, as illustrated in Figure 2.8. This state machine describes different stages for each NAND I/O operations as visualized in the bottom of Figure 2.7. All dies have such state machines based on stage and regulate/validate correctness of NAND commands and multi-stage sequence.

Command Chains. A command chain is a series of NAND commands, and each combination of NAND operations has its own command chain. Even though the state machine with multi-stage is capable of handling diverse depths of NAND command sets, the introduction of a command chain is required, because many operations have different command requirements and sequences. Also, the process of transitioning from one stage to another stage varies based on what command is loaded into the command register. For example, as illustrated in Figure 2.7, the write operation has a different sequence for data movement and a different number of commands compared to the erase and read operations. When a combination of NAND operations with cache, multi-plane or die-interleaved mode is applied, the differences are more striking. Therefore, NANDFlashSim employs command chains, which are updated by the NANDFlashSim controller and logical unit. Also, the command chains are used to verify whether the host model manages NAND operation using a correct set of commands and command sequences or not. Using multi-stage operations and command chains, NANDFlashSim defines a NAND command set architecture and provides a cycle accurate NAND flash model.

2.7.2 Awareness of Latency Variation

NANDFlashSim is designed to be aware of intrinsic latency variations when it simulates MLC NAND flash. To extract real performance and introduce variation characteristics into

NANDFlashSim, we implemented a hardware prototype called MSIS, which stands for *Memory Statistics Information System*. MSIS is able to evaluate various types of NAND flash based on different memory sockets as illustrated in Figure 2.9. Suppose that n_{pages} is the number of page per block, and λ is constant value related to a page offset. n_{pages} and λ are device specific value. Typically, n_{pages} is powers of two, and λ is 2 or 4. We assume that a set of page addresses, which show relatively high latency, indicates the MSB pages referred as to $msb(n)$, where $\forall n, 0 \leq n \leq (n_{pages}/2)$. We also assume that another set of page addresses, which exhibit low latencies, are the LSB pages referred as to $lsb(n)$. With this assumption[2] in place, NANDFlashSim generates different programming timing based on these two sets of page addresses, which are extracted from MSIS. Even though these address sets of page addresses can be varied based on NAND flash manufacturers, we found that these address sets can be classified by two groups for diverse NAND flash devices (we tested eight devices from four manufacturers, and technology nodes of them range from 24 nanometer to 32 nanometer). These two groups of such page address sets indicated by different subscripts, α and β (e.g., $msb_{\alpha}(n)$, $lsb_{\alpha}(n)$, $msb_{\beta}(n)$, and $lsb_{\beta}(n)$). For $lsb_{\alpha}(n)$, if n is zero or equal to $n_{pages}-1$ then n and $n+1$ are LSB pages. Otherwise, the $lsb_{\alpha}(n)$ is generated by λn, and the $msb_{\alpha}(n)$ is generated by $\lambda n-(\lambda+1)$. On the other hand, for $lsb_{\beta}(n)$, if n is less than λ or n is grater than $n_{pages}-\lambda$ then n is LSB pages. Otherwise, λn and $\lambda n+1$ are elements of $lsb_{\beta}(n)$, and $\lambda n-(\lambda+2)$ and $\lambda n-(\lambda+1)$ are elements of $msb_{\beta}(n)$. It should be noted that NAND flash parameters related to these sets of addresses only affect NAND flash activity, especially transfer-in of NAND (TIN) stage. I/O bus activities such as CLE, ALE, TIR, and TOR are not affected by such sets of addresses.

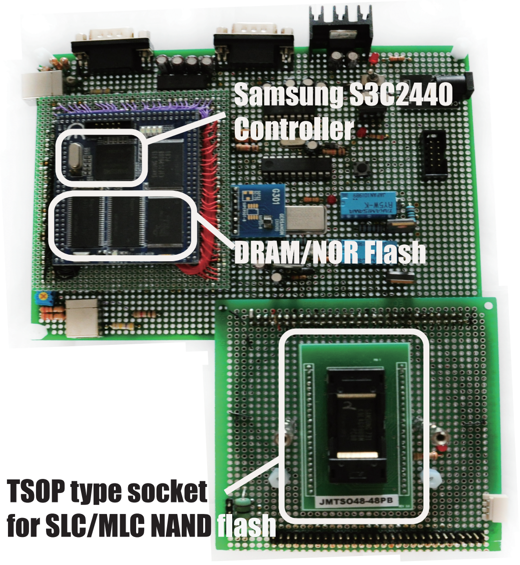
[2]This assumption is already widely used by both industry and academia [27, 64, 90].

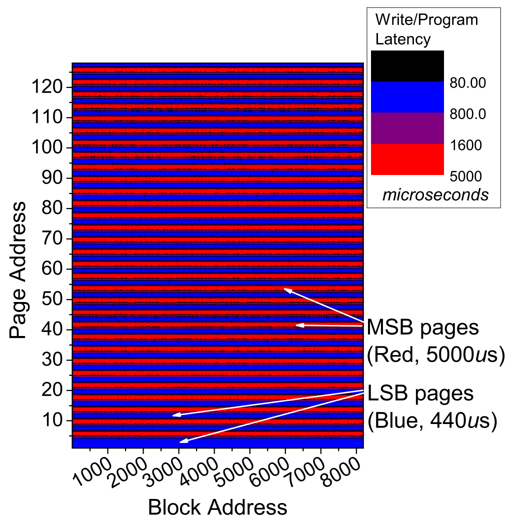
2.7.3 Enforcing Reliability Parameters

NANDFlashSim enforces three constraints to guarantee reliability: 1) the *Number-Of-Program* (*NOP*) constraint, 2) *In-order update* in a block, and 3) *endurance*. The NOP constraint refers to the total number of contiguous programmings that the NAND flash memory allows for a block before an erase is required. The plane model in NANDFlashSim records the number of programs for each page. If a request tries to program a page over the NOP limit, NANDFlashSim informs the host model of this violation. In addition, the plane model maintains the page address which was most recently written. When a host model requests to program a page that has a lower address than the most recently written page address, NANDFlashSim reports this as a violation of the in-order update constraint to the host. To enforce the endurance limitation, each block in the plane model tracks erase counts. When NANDFlashSim detects a request that would erase a block over the number of erases that the memory system guarantees, it informs the host model of this endurance violation. These reliability models provide an environment for system designers and architects to study NAND flash reliability and explore future research directions such as developing new wear-leveling, garbage collection and address mapping algorithms.

2.8 Evaluation

For the validation of NANDFlashSim compared to other real products, we utilize two different types of MLC NAND flash packages [75] (i.e., Single Die Package (SDP) and Dual Die Package (DDP), and two MLC devices came from two different manufacturers [75, 34]. In addition, for evaluating NANDFlashSim, we also use SLC and MLC type NAND flash. The


Samsung S3C2440 Controller
DRAM/NOR Flash
TSOP type socket for SLC/MLC NAND flash

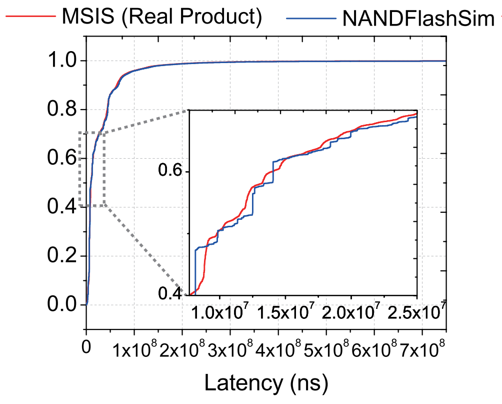


(a) Our Hardware Prototype (MSIS) (b) A Contour Map of Latency Variation

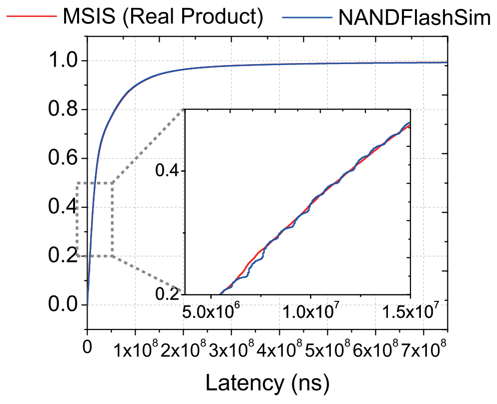
Fig. 2.9 Implemented evaluation hardware prototype (MSIS). MSIS is used to extract the LSB and MSB page address and to evaluate performance for NANDFlashSim validation.

main parameters for those devices such as block, page sizes and latency, are described in Table 2.1. Unless otherwise stated, we will use parameters of MLC1 as default.

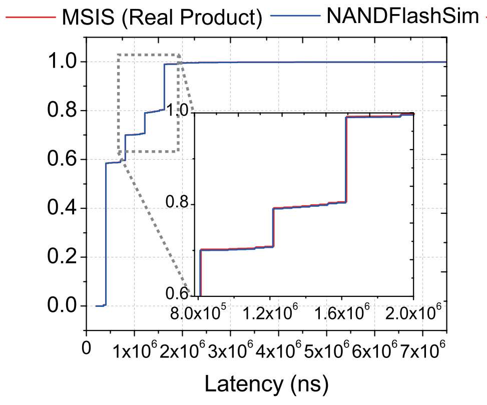
Table 2.2 analyzes workloads that we tested. In addition to a number of disk-friendly traces from actual enterprise applications (msnfs, fin, web, usr, and prn) [88], we also synthesized write and read intensive workloads of which access pattern are fully optimized to NAND flash. Specifically, in the swr and srd workloads, we perform reads and writes of data on different block boundaries, and make the access pattern of the workload sequential in the block boundary. With these synthesized flash-friendly workloads, the ideal performance of NAND flash can be evaluated with less restrictions. Access patterns of all workloads tested are used by both the hardware prototype (MSIS) and NANDFlashSim.



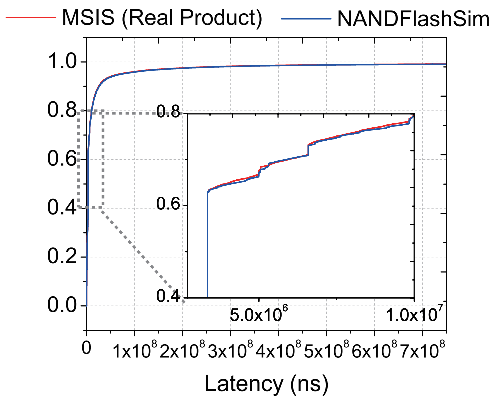
(a) msnfs



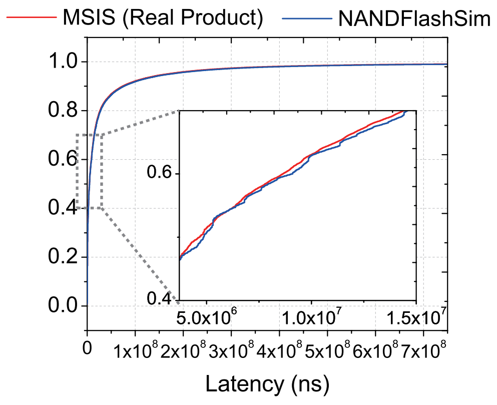
(b) fin



(c) web



(d) usr



(e) prn

Fig. 2.10 Cumulative Distribution Function (CDF) of Latency. Latencies of NANDFlashSim are mostly overlapped with real NAND flash product latencies.

Device Type	Feature	Value
Single Level Cell	Page Size(Byte)	2048
	# of Page Per Block	64
	# of Block	4096
	Write Latency(us)	250
	Read Latency(us)	25
	Erase Latency(us)	1500
Multi Leve Cell 1 (MLC1)	Page Size(Byte)	2048
	# of Page Per Block	128
	# of Block	8196
	Write Latency(us)	250∼2200
	Read Latency(us)	50
	Erase Latency(us)	2500
Multi Leve Cell 2 (MLC2)	Page Size(Byte)	8192
	# of Page Per Block	256
	# of Block	8196
	Write Latency(us)	440∼5000
	Read Latency(us)	200
	Erase Latency(us)	2500

Table 2.1 NAND Flash Device Characterization

Workloads	Write (%)	Write Req. Size (KB)	Read Req. Size (KB)
Synthesized Write Intensive (swr)	100	2	-
Synthesized Read Intensive (srd)	0	-	2
MSN File Storage Server (msnfs)	93.9	20.7	47.1
Online Transaction (fin)	84.6	3.7	0.4
Search Engine (web)	0.01	99.1	15.1
Shared Home Directories (usr)	2.6	96.2	52.6
Printing Serving (prn)	14.5	97.1	15.1

Table 2.2 Workloads Characterization

2.8.1 Validation of NANDFlashSim

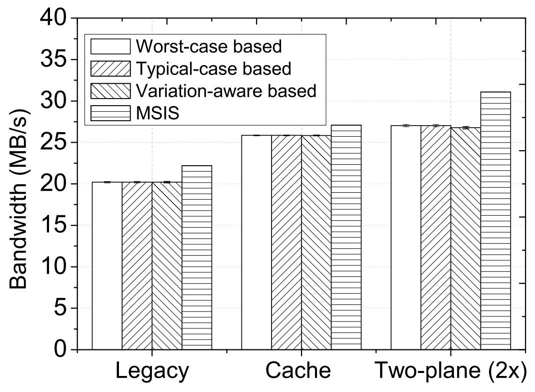
Latency Validation. Figure 2.10 pictorially illustrates cumulative distribution function (CDF) of latency for both NANDFlashSim and MSIS on enterprise application workloads. In this validation, interleaved die mode and multiplane mode NAND commands are interplayed with legacy mode NAND operations, and a queue (32 entries) [37] is applied for handling incoming I/O requests. The microscopic illustration of inflections for each CDF are also pictorially embedded. In the figures, the red line represents MSIS latency with MLC2 [34] and the blue line represents latency of variation-aware NANDFlashSim. As shown in the figures, latencies of NANDFlashSim are almost completely overlapped with the real product latencies. Since NANDFlashSim employs a variation-aware timing model as default, it exhibits very close performance to the real product of MSIS.

System Performance Validation. In these performance validation tests, we evaluate performance for our variation-aware based NANDFlashSim, worst-case timing based simulation model [51], typical-case timing based simulation model [1, 60, 32] and MSIS in terms of bandwidth. In this test, we scheduled NAND I/O commands in plane-first fashion, which means the requests are served with write/read two-plane mode first rather than striping them across multiple dies.

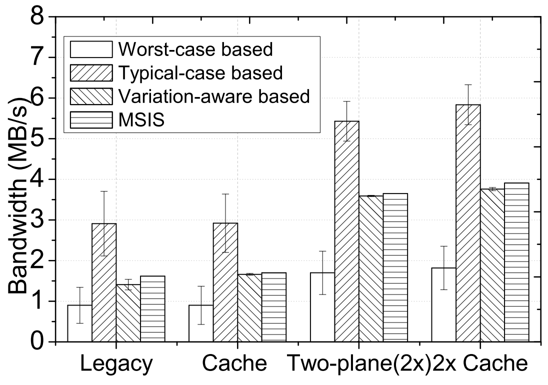
Figure 2.11 compares the SDP [75] read/write performance on NANDFlashSim and MSIS. The throughput values obtained using NANDFlashSim (with variance-aware latency model) are close to the real product performance of MSIS. In the read cases, NANDFlashSim is no more accurate relative to MSIS than the other timing models. This is because variation for reads of NAND flash memory is negligible. In contrast, write operations show different performances according to the type of latency models employed. Since write performances are seriously varied between the minimum to maximum latency, there is a performance gap between performance of MSIS and that of both worst-case timing parameter and typical-timing parameter based latency models.

These plots also depict bars of the percentage of deviation between MSIS performances and NANDFlashSim performances among different latency models. Specifically, variation-aware NANDFlashSim provides around 12.9%, 2.1%, 1.6% and 3.8% deviation in performance for the legacy, cache mode, 2x mode, and 2x cache mode operation, respectively. This is a significant improvement from the deviation range of 48.7% to 79.6% in typical-case timing parameter based simulation and from 44.4% to 53.5% in the worst-case timing parameter based simulation.

Figure 2.12 illustrates read/write performance results with DDP [75] on the same test set. Typical-timing parameter based latency model shows highly errant performance compared to MSIS ones (deviation range is 83.1% to 170.9%). Even though this latency model is the most popular one among SSD simulators, it mimics the ideal performance, which can be achieved *if and only if* the time spent in TIN can perfectly be overlapped with other operations stages, and the NAND bus I/O utilization is reasonably high across multiple dies. Although the worst-case parameter based latency model provide closer performance (deviation range is 7.3% $\sim$ 42.0%), it still shows unrealistic performance. In contrast, the performance deviation range of our current variation-aware NANDFlashSim is between 5.3% and 9.4%. This is because the detailed bus arbitrations across multiple dies, which are based on the intrinsic latency variations are captured by NANDFlashSim.

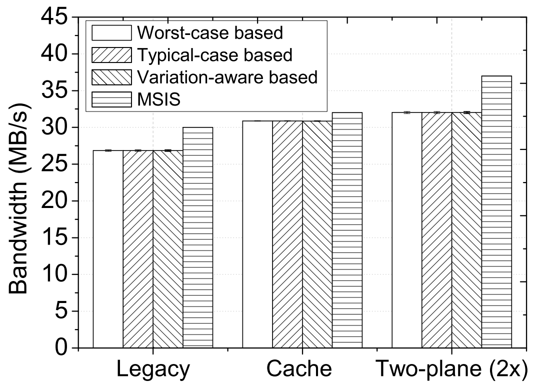


(a) Read performance (srd)

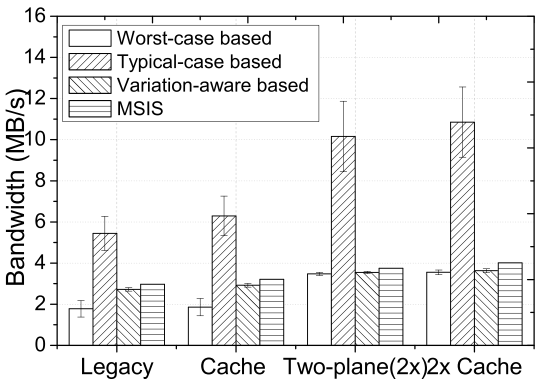


(b) Write performance (swr)

Fig. 2.11 Performance comparison on SDP. Typical-case/worst-case time parameter based latency models show unreasonable performance gap from real product ones.



(a) Read performance (srd)



(b) Write performance (swr)

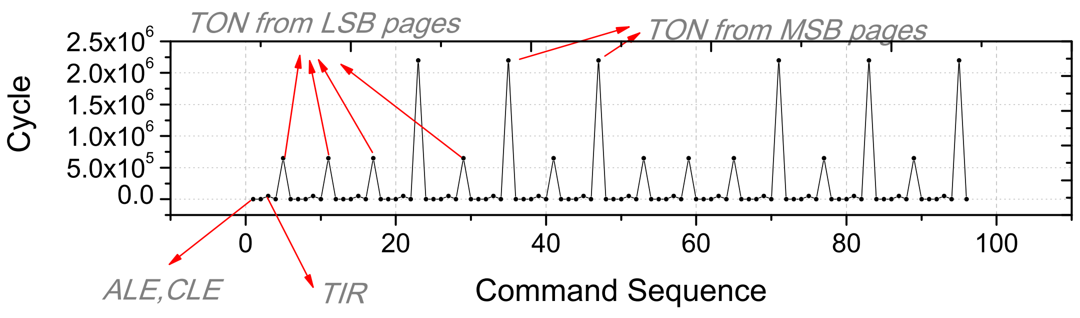
Fig. 2.12 Performance comparison on DDP. While typical-based latency model show more discrepant performance than that of multi-plane tests, the variation-aware NANDFlashSim provides performances close to the real product ones.

2.8.2 Individual Cycle Analysis

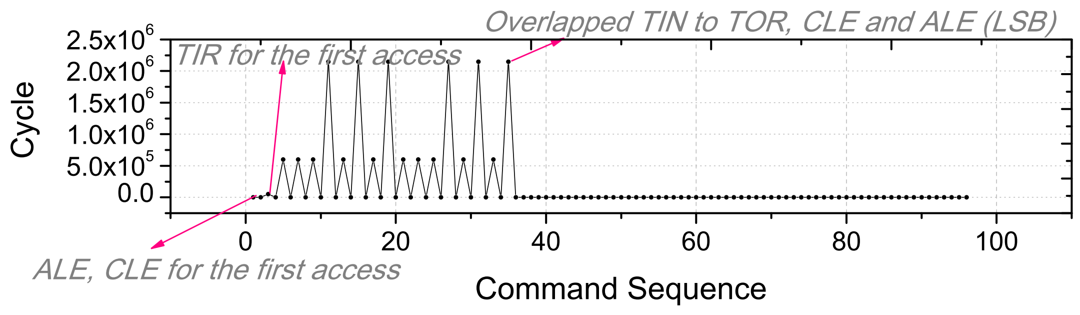
Figures 2.13 and 2.14 illustrate an individual cycle comparison among legacy, cache mode, and 2x mode operations. In this evaluation, we request 8 pages read or write for two blocks, and the size of the requests is 2KB.

Write Cycles. Cycle analysis for legacy mode writes is illustrated in Figure 2.13(a). In the write operations, the performances of TINs fluctuate from 650 thousand cycles to 5 million cycles. This intrinsic latency variation is one of the reasons why NANDFlashSim demonstrates performance closer to reality. Figure 2.13(b) illustrates write cycle analysis of cache mode operations. Since latencies for ALE, CLE, and TIR operations (related to operating the NAND flash I/O bus) can be overlapped with latency of the TIN operation, latencies for sixteen TIN stages consecutively occur without latencies spent to operate the I/O bus.

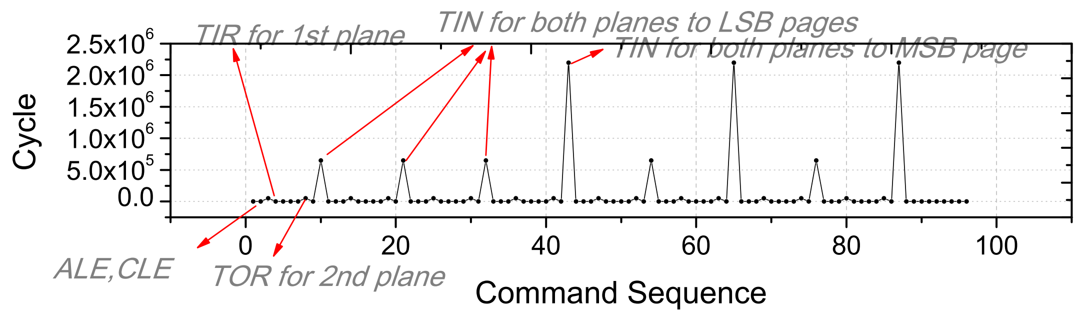
Figure 2.13(c) depicts cycle analysis for 2x mode write operations. While cache mode operations save cycles for the I/O bus, 2x mode operation reduces the number of TIN operations



(a) Legacy write operation

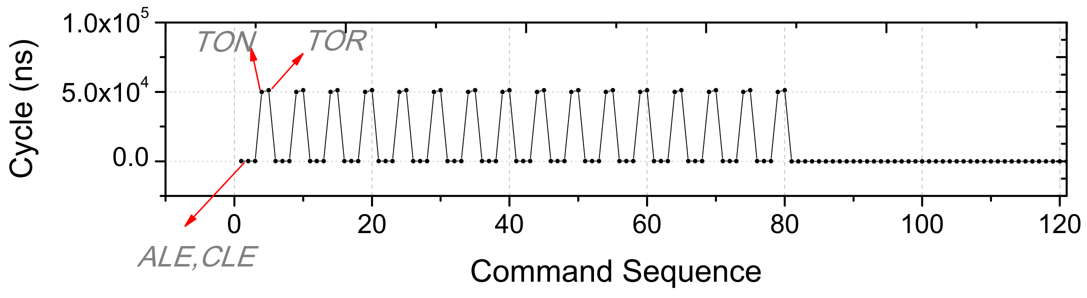


(b) Cache mode write operation

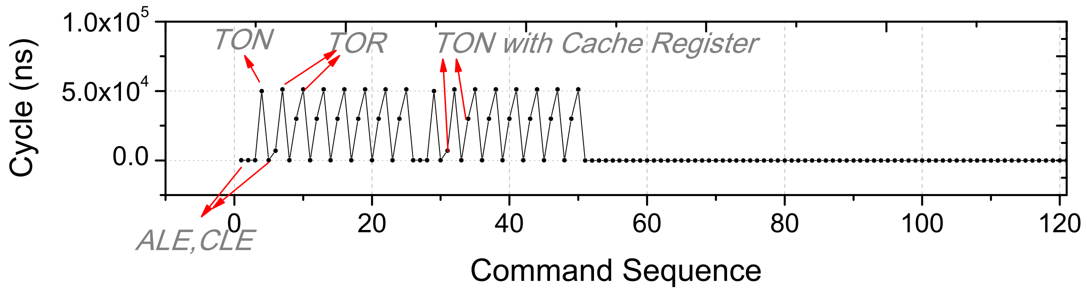


(c) Two-plane mode (2x) write operation

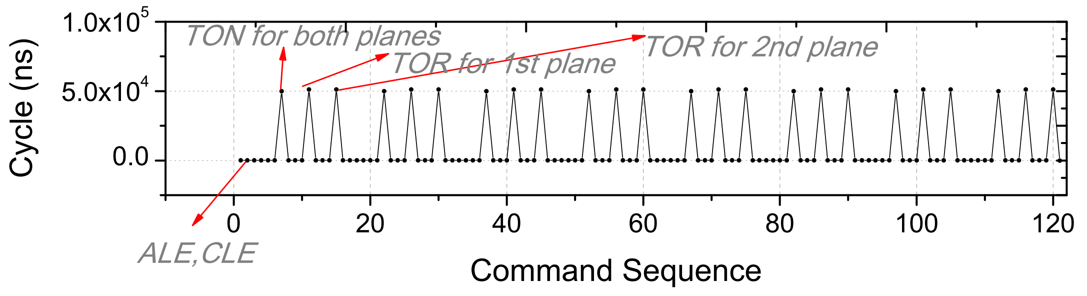
Fig. 2.13 Cycle analysis for write operations (NANDFlashSim). Note that the command sequence is a chronological oder based on a set of NAND commands that host commits, and one cycle takes 1 nanosecond.



(a) Legacy read operation



(b) Cache mode read



(c) Two-plane mode (2x) read operation

Fig. 2.14 Cycle analysis for read operations (NANDFlashSim). Since handling 2x mode is fancy more than other operation modes, it requires more commands to read data. Note that system designers is able to get these microscopic cycle analysis for diverse NAND flash operations from outputs of NANDFlashSim.

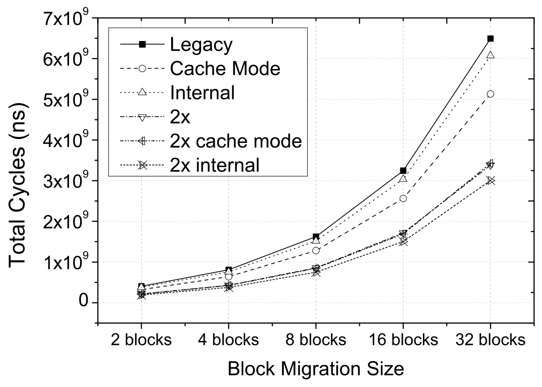
itself by writing data to both planes at a time. This is because those planes share one word-line. Since cycles spent in the TIN operation is much longer than the sum of cycles for ALE, CLE and TIR operations, it doubles throughput as shown in Figure 2.11(b).

Read Cycles. Figure 2.14 illustrates read cycle analysis executed by NANDFlashSim. Read operation behaviors for legacy, cache mode and 2x modes are similar to the writes, having only two main differences: 1) latencies for the TON operation do not fluctuate like TIN of write operations, and 2) the TOR cycle fraction of the total execution cycles (see Figure 2.14(a)) is close to the TON ones. In Figure 2.14(b), one can see that cycles for TOR, which are related to bus operations, are higher than TON related to operations for NAND memory cells, meaning that reads spend many cycles on the I/O bus operations (we will discuss more detail in Section 2.8.4).

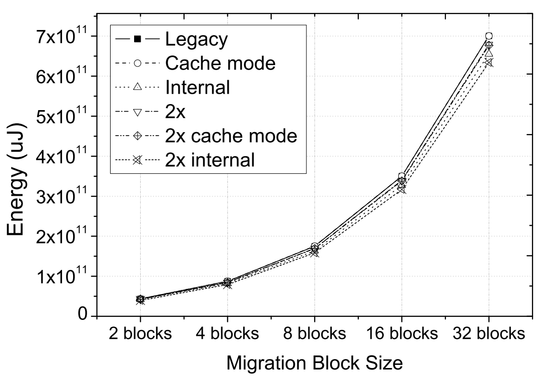
It should be noted that the reason one obtains accurate latency values from NANDFlash-Sim is that it works at cycle-level and executes NAND operations through multi-stage operations, which are defined by NAND command set architecture. In addition, NANDFlashSim reproduces intrinsic latency variations based on different addresses for LSB and MSB pages.

2.8.3 Performance and Power Consumption Comparison: Page Migration Test

We also evaluate a page migration test, which is a series of tasks copying pages from source block(s) to destination block(s) and erasing the block(s). This test mimics a very time consuming job of a flash firmware occuring frequently during garbage collection. To evaluate performance of page migration, we read whole pages in the source blocks and wrote them to the destination blocks on NANDFlashSim for various block sizes. In this process, we erased the



(a) Bandwidth



(b) Energy

Fig. 2.15 Block migration performance comparison and energy consumption. While energy consumption are similar to different NAND operation modes, 2x cache mode and internal data move mode operations have great impact on enhancing performance in the page migration test.

destination blocks before migrating pages, and erased the source blocks after the page migration tasks are done. These page migration tasks are performed by legacy, cache, internal, 2x, 2x cache, and 2x internal mode operations. As shown in Figure 2.15(a), there is little performance difference at page migrations of 2 blocks, but as the number of blocks for the migration increases, latencies for 2x cache mode, internal data move, and 2x internal data move mode operations are about two times smaller than legacy, cache mode and 2x mode operations. Importantly, the 2x internal data move mode operation outperforms all other operations. In contrast, energy consumptions for each NAND I/O operation are not much different between them as shown in Figure 2.15(b). This is because even though the latency benefits come from internal parallelism, the same amounts of power for operating I/O requests are required by all memory system components.

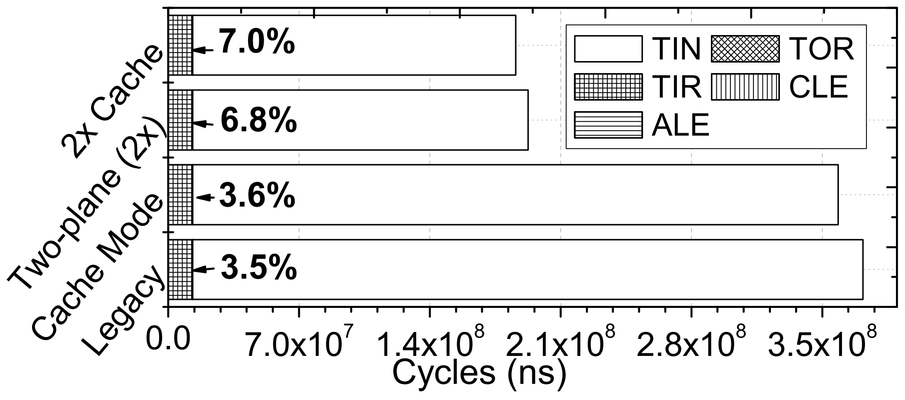
Figure 2.17 shows cycle analysis for each NAND operation type. One can see that internal data move mode operations (including 2x internal) eliminate operations associated with registers (TOR and TIR), thereby improving migration performance.

2.8.4 Breakdown of Read and Write Cycles

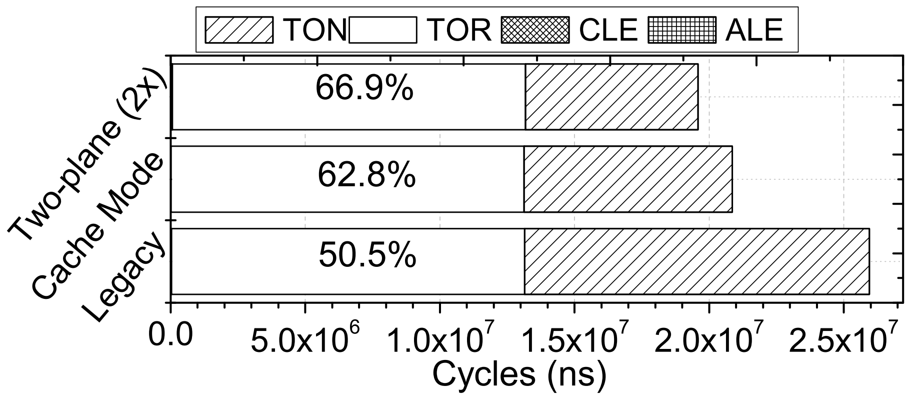
In the write cases shown in Figure 2.16(a), most cycles are consumed by operations related to NAND flash itself (93%$\sim$96.5%). While write operations spend at most 7.0% of the total time performing data movement (TOR/TIN), read operations spend at least 50.5% of the total time doing so. Therefore, even though 2x or cache mode is applied to read operations (see Figure 2.16(b), there is small benefit in terms of bandwidth. We believe that this is a reason why much research in industry is directed towards enhancing bus bandwidth. However, do note that the write performance cannot be enhanced by any kind of high speed interfaces because the speed is dominated by the latency of the TIN stage. It also should be noted that since NANDFlashSim allows us to count cycles dedicated to each NAND flash stage and command, it helps us determine which operations are the performance bottleneck, or which operation is the best operation for a specific access pattern to improve performance.

2.8.5 Performance on Multi-plane and Multi-die Architectures

Multi-plane. Figure 2.18 compares throughputs observed in NANDFlashSim for varying the numbers of planes and transfer sizes. Performance of write operations is significantly enhanced as the number of planes increases because most of TIN can be executed in parallel. In contrast, for the read operations, such benefits are much lower than for write operations. This is because cycles for data movement (TOR) are a dominant factor in determining bandwidth, and



(a) Writes (swr)



(b) Reads (srd)

Fig. 2.16 Breakdown of cycles. Note that, in reads, TOR operations are the performance bottleneck while TIN operations are on the critical path in writes.

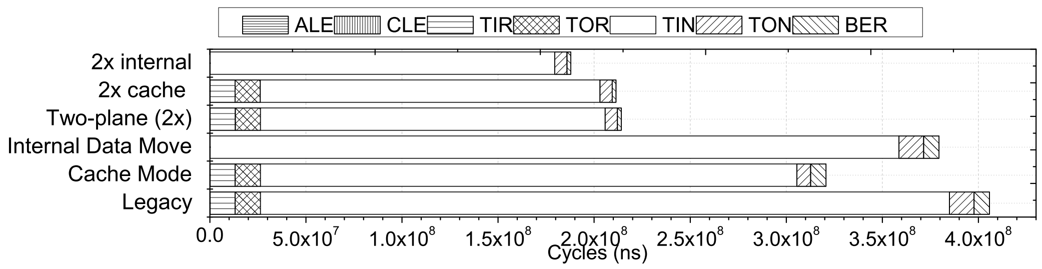
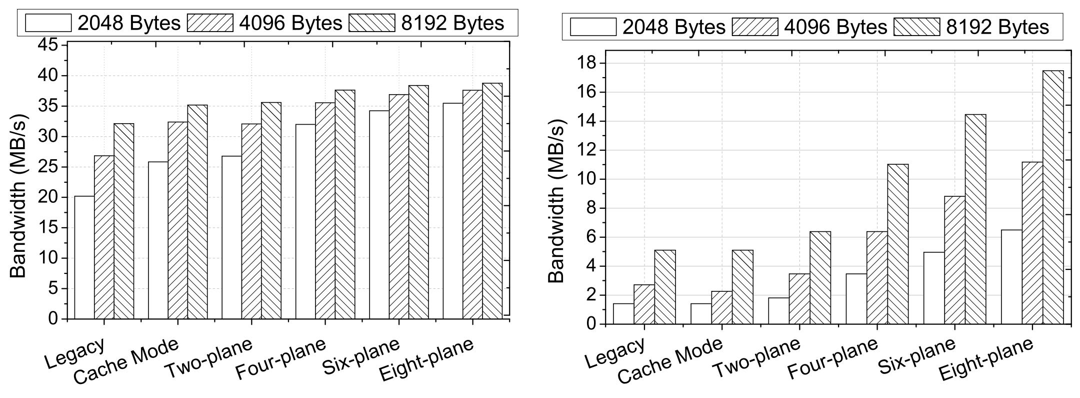


Fig. 2.17 Cycle analysis for page migration. Internal data move modes removes the most NAND I/O bus activities thereby improving throughputs.

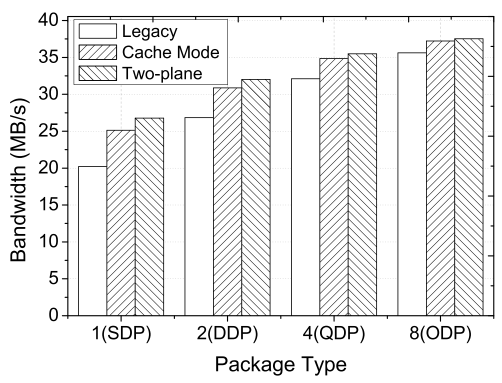
it is unaffected by the number of planes. As shown in Figure 2.20(a), this performance bottleneck of a many-plane architecture becomes more problematic under disk-friendly workloads. Specifically, the performance gains of the many-plane architecture become limited starting at a four-plane architecture. The main reason is that most workloads are optimized for traditional blocks without regard for the plane addressing rule. In other words, as the number of planes increases, it becomes hard to build multi-plane mode operations with existing disk-friendly I/O access patterns.



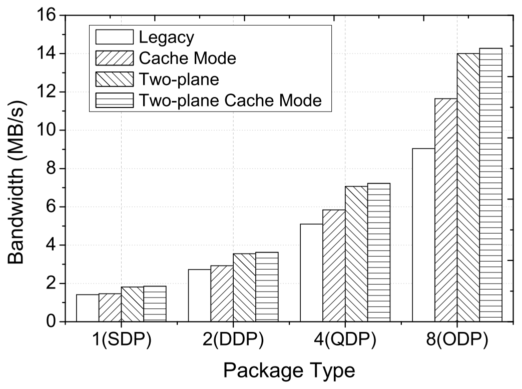
(a) Read performance (srd) (b) Write performance (swr)

Fig. 2.18 multi-plane architecture performance with varying page unit sizes. While write throughputs of many plane architecture are enhanced by 360.9% to single plane architecture, read throughputs are enhanced by 75.5%.

Multi-die. Figure 2.19 illustrates performance improvement as the number of dies increase. In this section, we tied multiple dies to one NAND flash I/O bus path and have them sharing one CE pin. Similar to multi-plane operations, reads performance enhancement (as the number of dies increases) is limited by latency of the TOR operation. Even though multiple dies are able to serve I/O requests in parallel, performance is bounded by data movement again. This is because



(a) Read performance (srd)

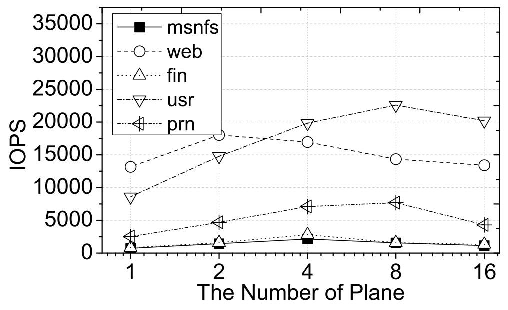


(b) Write performance (swr)

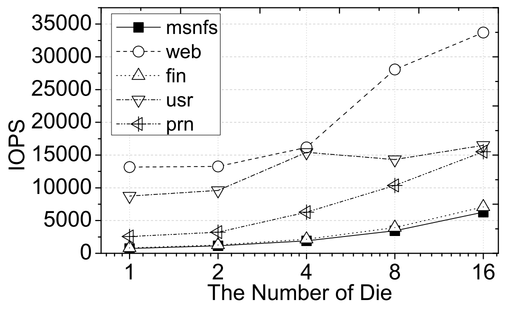
Fig. 2.19 Multi-die architecture performance. While many dies architecture with the swr workload enjoys linear enhancement (ODP improves throughput 541.1% to the SDP ones), it saturates read throughputs with eight dies (76.3% enhancement compared to SDP ones).

during execution of a TOR operation, the NAND flash I/O bus is not capable of handing another TOR one. Therefore, regardless of the fact that 2x or cache mode are applied, TOR operations are the performance bottleneck in read case.

In contrast, as shown in Figure 2.19(b), throughputs of write operations are significantly improved by increasing the number of dies. The reason behind this benefit is interleaving TIN, which is the dominant factor in determining write bandwidths with small bus resource conflicts. It should be noted that NANDFlashSim is able to reproduce/simulate resource (NAND flash I/O bus and dies) conflicts by employing multi-stage operations and being aware of intrinsic latency variations at μarch-level. As shown in Figure 2.20(b), unlike many-plane architecture, many-die architecture enjoys performance gains under even disk-friendly real workloads. This is because data can be parallelized across multiple dies with fewer restrictions.

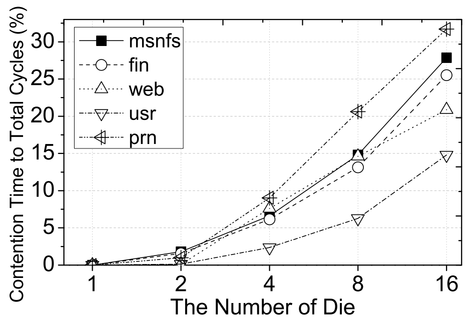


(a) Sensitivity to number of plane

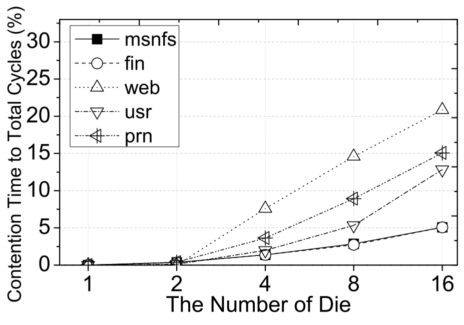


(b) Sensitivity to number of die

Fig. 2.20 Performance sensitivity to the number of plane and die with actual application workloads. the performance of many-die architecture is 54.5 % better than the performance of many-plane architecture in terms of IOPS.

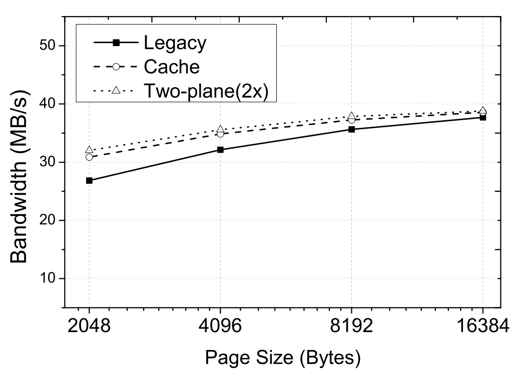


(a) Single level cell (SLC)

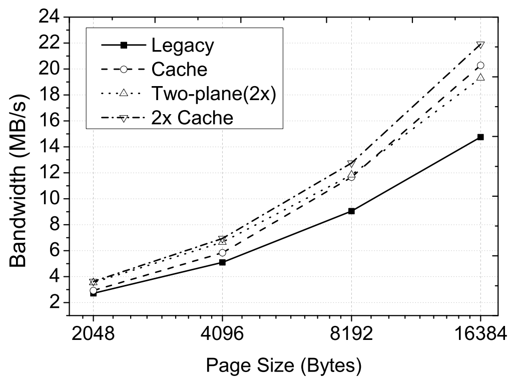


(b) Multi level cell (MLC)

Fig. 2.21 Resource contention comparison between SLC NAND and MLC NAND devices. The resource contentions of MLC NAND flash have less impact on SLC NAND flash, but the contention problem is still problematic and become more serious as the number of die increases.

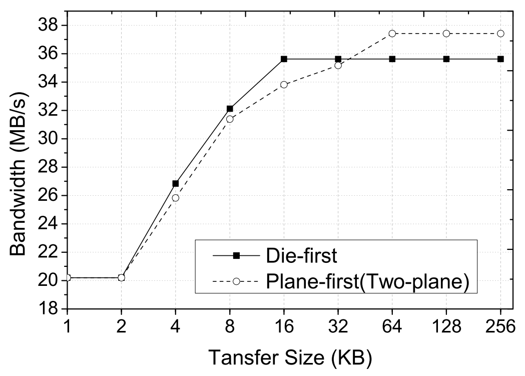


(a) Read (srd)

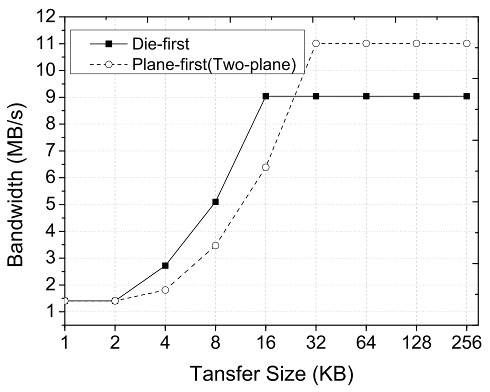


(b) Write (swr)

Fig. 2.22 Sensitivity to page organization (2x, DDP). Most read performance are bounded because of TOR times and NAND flash I/O bus competition.



(a) Read performance (srd)



(b) Write performance (swr)

Fig. 2.23 Effects of different NAND command scheduling policies. Based on different transfer sizes and scheduling policies, performance enhancement with multi-die and plane architecture show different performance.

2.8.6 Performance Sensitivity to Page Size

Intuitively, large page sizes can be a good choice to achieve high bandwidth because many bytes can be programmed or read within the same amount of cycles. However, this intuition is only true for writes. Figure 2.22 plots performance sensitivity to different page sizes on diverse read and write operations. While the bandwidth of writes for most operation modes increases as the page size increases, read performances saturate. As explained in Section 2.8.5, the small enhancements for read operations are due to bus resource conflicts and the large time spent in data movement.

2.8.7 Resource Contention

Since multiple dies share the flash interface, I/O bus activities such as ALE, CLE, TOR and TIR should be serialized, which means they cannot execute simultaneously. Instead, this I/O bus activities can be interleaved across multiple dies at μarch level. During the interleaving time, I/O requests related to such activities suffer from internal NAND I/O bus resource contention. Figure 6.12 visualizes the fraction of internal NAND I/O bus resource contention to total I/O execution time using disk-friendly workloads. As shown in the figure, interleaving I/O bus activities in SLC is 45.2% more competitive than MLC's ones. The reason is that since the latencies of MLC activities are much longer than the latencies of SLC activities, it has more chances to be executed with I/O bus activities at the same time. However, as the number of die increases, for both SLC and MLC throughput, the fraction of the I/O bus resource contention to total I/O execution time increases, which is a reason of performance limitation in many dies architecture.

2.8.8 Scheduling Strategy

To test the potential research on NAND command scheduling strategies, we implemented two simple command schedulers in the logical unit of NANDFlashSim: 1) Die-first and 2) Plane-first schedulers. The die-first scheduler simply stripes I/O requests as they arrive over multiple dies rather than planes. In the plane-first scheduler, I/O requests are collected into two pages upon arrival and served to multiple planes rather than striping them across dies. As illustrated in Figure 2.23, since multiple dies share one I/O bus, performances saturate faster than with the plane-first scheduler. Even though plane-first operation provides better performance, the die-first scheduler is more flexible in serving I/O requests of a smaller size. This is because multi-plane operation performance is limited by plane addressing rules (see Section 2.3.3), whereas multiple dies can be interleaved to serve I/O requests without any addressing constraints.

2.9 Simulation Speed and Download

The current version of NANDFlashSim is capable of executing 824 I/O requests (2KB) per second for DDP and 295 I/O requests per second for ODP with MLC1. The simulator performances were measured on a machine with virtualized dual core, 1GB memory, and 200GB disk. The source code can be downloaded from http://www.cse.psu.edu/∼mqj5086/nfs.

2.10 Conclusion

Since NAND flash memory is sensitive to a large number of parameters, and some performance parameters have significant latency variation, making decisions on how to configure NAND flash memory for optimal performance is non-trivial. A comparison of various NAND flash memory architectures become even harder when considering multi-die and multi-plane architectures, latency variations, energy consumption costs, reliability issues, and addressing restrictions. Therefore in this Chapter we propose NANDFlashSim, a detailed and highly configurable low-level NAND flash simulation model. NANDFlashSim supports detailed timing models for sixteen I/O operations by being aware of intrinsic latency variations. Our ongoing work includes incorporating a 400MHz high speed NAND interface (not published yet) and implementing a multiple logical unit on chip architecture. In addition, we plan to apply our simulation model to cycle accurate Green Flash [93] and Tensilica Xtensa simulation model [98] of hardware/software co-design platform for exascale computing [73].

Chapter Acknowledgements Chapter 2, in part, is a reprint of the material as it appears in ”NANDFlashSim: Intrinsic Latency Variation Aware NAND Flash Memory System Modeling and Simulation at Microarchitecture level,” Myoungsoo Jung, Ellis Herbert Wilson III, David Donofrio, John Shalf, Mahmut Kandemir, in Proceedings of the 28th IEEE Conference on Massive Data Storage (MSST), 2012. The dissertation proposal author was the primary investigator and the first author of this paper.

Chapter 3

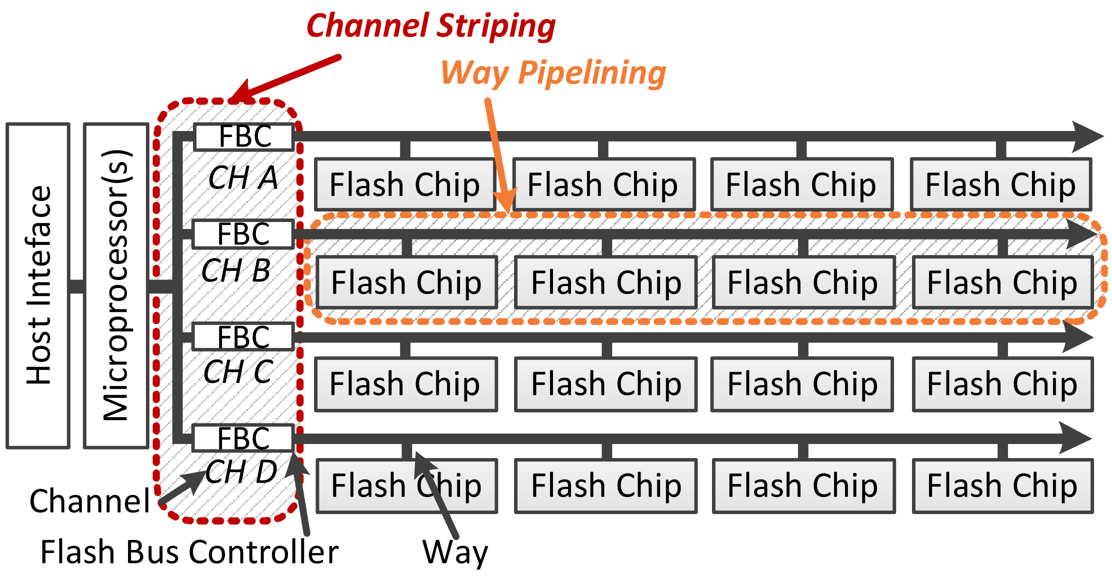
Page Allocation Strategies for Parallelizing Data Accesses

3.1 Introduction

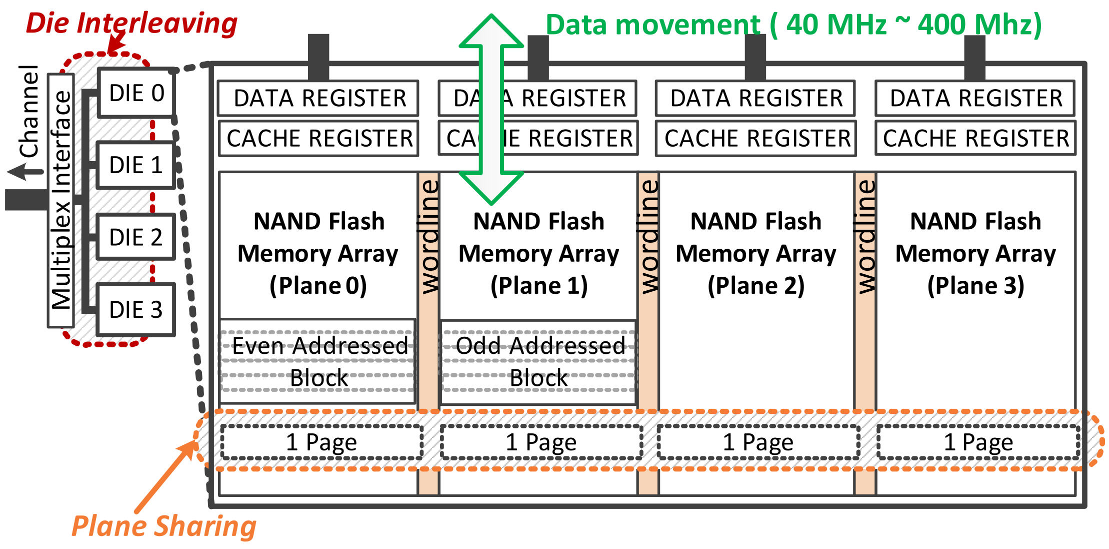
NAND flash-based Solid State Disks (SSDs) are being increasingly used in enterprise, personal and high performance computing systems, due to their performance advantage over spinning devices. While high-performance interfaces with transfer rates ranging from 6Gb/sec to 16GT/sec are being adopted by modern SSD architectures, the speed of NAND flash memory (i.e., flash) is still limited by about 40MB/sec [20]. This performance gap between SSD interfaces and flash chips has driven the research that target internal parallelism in SSDs, which can have a great impact on improving system performance. From an architecture perspective, SSD systems and flash devices expose parallelism at different levels. More specifically, SSD systems employ multiple flash chips over multiple channel I/O buses and multiplexed flash interfaces, which means that multiple SSD components can be simultaneously activated to serve incoming I/O requests. In parallel, flash technologies are being developed to extract maximum parallelism. A single flash chip consists of multiple dies, each of which accommodating multiple planes. Obviously, performance characteristics of modern SSDs are varied based on what strategies are employed for parallelizing data accesses across hundreds or thousands of flash dies and planes. A key design issue behind exploiting parallel data accesses is how to efficiently exploit internal parallelism and how to organize parallelism-friendly physical data layout for both the SSD system and flash levels.

As exploiting internal parallelism is key to improving performance and filling the performance gap between flash and high-speed interfaces, parallel data access methods are getting attention from both academia and industry [10, 13, 84, 101]. Architectural approaches to system-level parallelism using multiple I/O buses and flash chips such as ganging/superblock have been explored, and flash-level concurrency mechanisms utilizing multiple dies and planes within a flash chip like interleaving/banking have been already studied [1, 20]. However, *page allocation (palloc) strategies* enabling both system- and flash-level parallelism by determining physical data layout, have received little attention so far. A few palloc schemes in favor of the channel striping based data access method have been investigated [42], and the interplay between flash-level parallelism and these *channel-first* palloc schemes have been studied [31]. In addition, very little has been published on understanding the interactions between system-level and flash-level parallelism.

In this Chapter, we explore different page allocation strategies, geared toward exploiting both system-level and flash-level parallelism – we study a full design space siting on system and flash-level organizations with a variety of parameters such as a standard queue, multiple buses, chips, and diverse advance flash operations. Specifically, we evaluate twenty four palloc strategies including the *flash-level resource-first* and *way-first* strategies we defend. The questions we are interested in answering include 1) *which palloc scheme would be globally optimal for parallelizing data accesses when both system- and flash-level parallelisms are considered?*, 2) *what are the relationship between different level concurrency methods?*, and 3) *what are the resource utilizations of different palloc schemes?* To the best of our knowledge, this is the first report that explores all possible combinations of palloc strategies considering all levels of parallelism in SSDs. Our main **contributions** can be summarized as follows:



(a) SSD Internals (system-level)


Die Interleaving
Data movement (40 MHz ~ 400 Mhz)
Channel
Multiplex Interface
DIE 0
DIE 1
DIE 2
DIE 3
DATA REGISTER
CACHE REGISTER
NAND Flash Memory Array (Plane 0)
NAND Flash Memory Array (Plane 1)
NAND Flash Memory Array (Plane 2)
NAND Flash Memory Array (Plane 3)
wordline
Even Addressed Block
Odd Addressed Block
1 Page
Plane Sharing

(b) Flash Internals

Fig. 3.1 Internals of SSD and NAND flash and illustration of different level data accessing methods.

• *Determining good page allocation schemes.* We observe from our experiments that the channel-and-way striping based palloc is *not* the best strategy from a performance perspective, despite recent works [31, 42] claiming that. Our experiments in contrast reveal that, when advance flash operations are considered, a flash-level resource-first palloc scheme results in better throughput than the approach in [31, 42] (as much as 84.8% and on average 40.1% with very similar response times).

• *Addressing parallelism interference.* As opposed to the common perception that system and flash-level concurrency mechanisms are largely orthogonal, channel striping method in system-level makes it hard to exploit flash-level parallelism under disk-friendly workloads. In fact, in the worst case, flash-level parallelism extracted by the channel striping method shrinks as much as 99% and on average 44% since it exhibits poor locality in flash chips.

• *Addressing resource utilization.* We observe from our experimental analysis that most parallel data access methods and palloc schemes have room for performance improvement since many internal resources with them are still underutilized. When considering all the cases tested, channel resources are 57.9% underutilized and the activate time for buses and flash memory cells accounts for only 22.1% of the total execution time.

3.2 SSD Internals and Parallelisms

Since multiple flash chips are packaged in the form of an SSD, there are numerous hardware components and buses that work in tandem to provide access to the internals of flash. In this context, *channels* are I/O buses that are independently operated by microprocessors, and *ways* are data paths, connected to flash chips in each channel. Within each flash chip are one or more *dies*, sharing the single multiplexed interface. Lastly, the dies accommodate multiple *planes*, the smallest unit to serve a request in parallel. Figures 3.1(a) and 3.1(b) depict SSD and flash internals with corresponding *parallel data access methods*, respectively.

System-Level Parallelism. At a system-level, in the beginning of a data access process, an I/O request can be striped over multiple channels, and this process is termed as *channel striping*. Unlike channels, way-level activities should be serialized because the multiplexed interfaces of each flash chip are shared within a channel. Individual chips can however work in parallel, and a flash memory transaction consists of multiple phases; consequently, I/O requests can be pipelined. Therefore, using *way pipelining*, multiple I/O requests can be simultaneously served by multiple flash chips in a channel.

Flash-Level Parallelism. After I/O requests are striped over multiple flash chips, they can be further interleaved across multiple dies in a flash chip. Similar to way pipelining, the data movements and flash command controls in this *die interleaving* need to be serialized. Still, in an ideal case, performance increases by about n times, where n is the number of dies. *Plane sharing* concurrently activates flash operations on multiple planes, which can improve performance by about m times, where m is the number of planes. Finally, these two parallel data access methods can be combined when incoming I/O requests span all of the flash internal components. This method is referred to as *die interleaving with multiplane*, and it can improve performance by about $n * m$ times. It should be noted however that, unlike system-level parallelism, data accessing mechanisms in this level are only available via *advance flash commands* provided by flash chip makers.

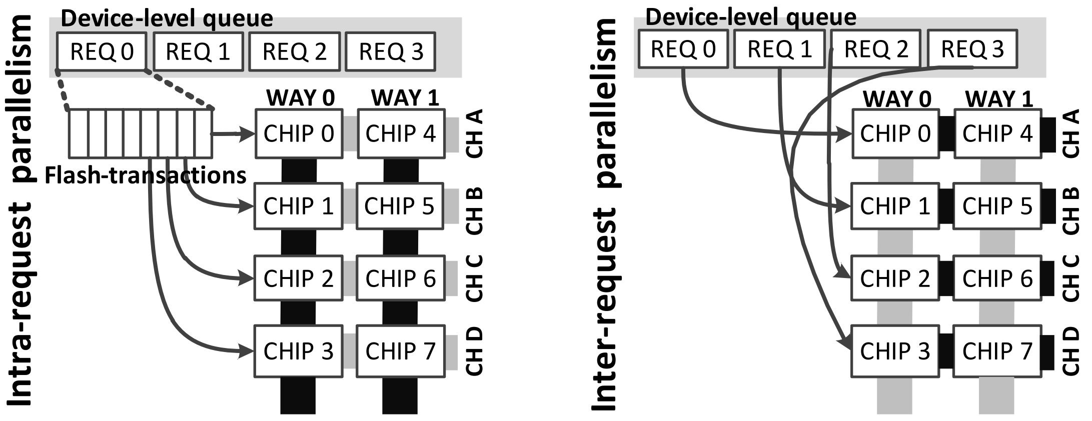


Fig. 3.2 Request-level parallelism.

Request-Level Parallelism. Parallel data access methods can serve flash-transactions within an I/O request or between the I/O requests sitting in a device-level queue (Figure 3.2). Generally speaking, *Intra-request parallelism*, referring the former reduces latency, and *inter-request parallelism* indicating the latter, improves storage throughput.

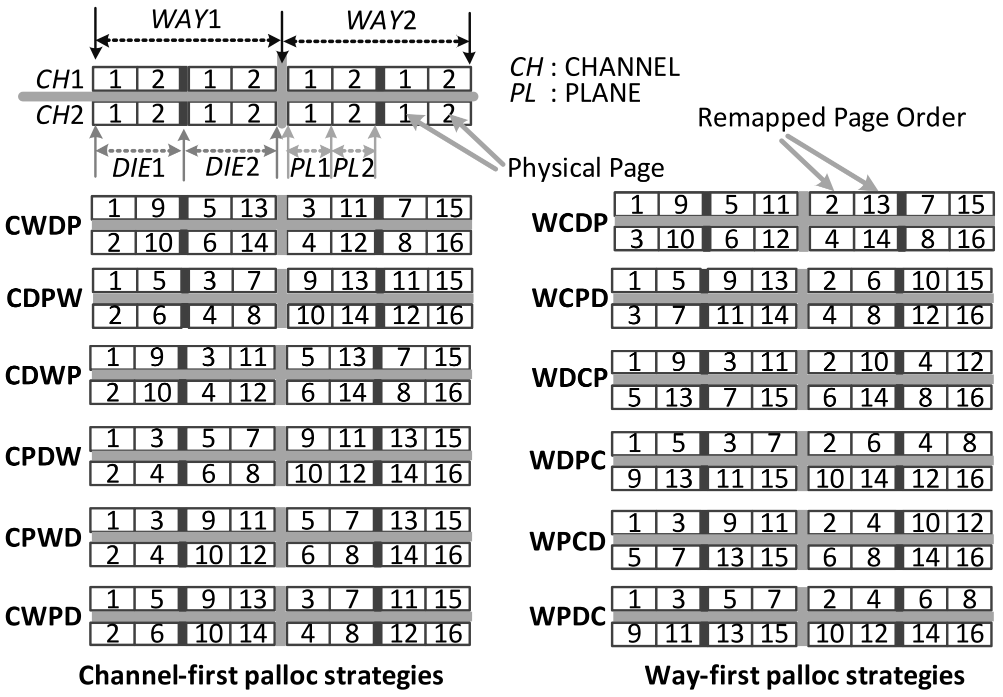
3.3 Page Allocation Strategies

SSDs decide physical data layout by remapping logical and physical addresses. This data layout within and between flash chips should be carefully determined so that one can exploit all levels of parallelism mentioned in Section 3.2. Since page allocation (*palloc*) strategies are directly related to the physical layout of data, the performance of an SSD can vary based on which palloc scheme is employed.

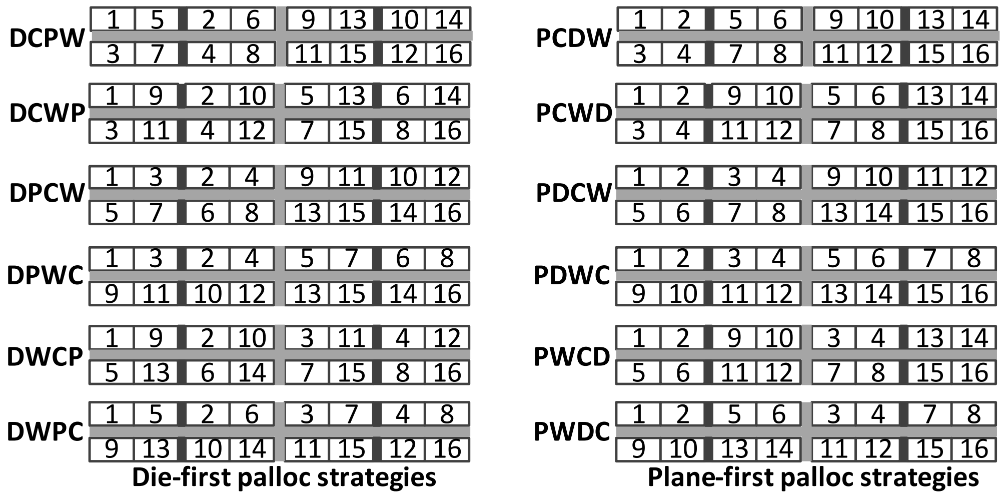
Figure 3.3 illustrates twenty four different palloc strategies oriented toward exploiting system-level and flash-level resources. At the top-left corner of Figure 3.3(a), we show how to identify the internal resources in these different palloc strategies. In order to distinguish among different palloc strategies, we use abbreviations composed of the initial letters of internal resources based on their priority. The order of numbers in the figure indicates how each palloc scheme allocates internal resources. For example, in the CWDP (Channel-Way-Die-Plane) palloc scheme, requests are first striped across multiple channels and ways. Flash-transactions corresponding to these requests are then assigned to multiple dies and planes.

Channel-first palloc strategies allocate internal resources in favor of the channel striping method, which can maximize the benefits coming from intra-request parallelism. Therefore, latencies experienced by these palloc strategies are expected to be lower when the requests span all of channels. In comparison, *way-first palloc strategies* are oriented toward taking advantage of the way pipelining, and can improve throughput by maximizing inter-request parallelism. In contrast, *die-first* and *plane-first palloc strategies* allocate flash-level resources rather than channels or ways in an attempt to reap up the benefits of die interleaving, plane sharing, or die interleaving with multiplane methods.

These palloc strategies can be incorporated into an existing flash translation layer (FTL), which is the internal software to perform mapping between logical and physical addresses. Even under the situation that the FTL remaps addresses, page ordering performed by pallocs will still have a great performance impact because pallocs determine the order in which coalesced data pages are written to physical pages, which in turn influences the order in which the pages are read.



(a) System-level resource-first allocation



(b) Flash-level resource-first allocation

Fig. 3.3 Different page allocation strategies.

3.4 Experimental Methodology

To evaluate each of the palloc schemes shown in Figure 3.3, we needed a high-fidelity simulator that can capture cycle-level accuracy and interaction between internal resources. Motivated by this, we developed a *cycle-accurate* NAND flash simulator[1], which is hardware-validated, aware of intrinsic flash latency variation and support advance flash operations. Micron multi-level cell (MLC) NAND flash[2] is used for the NAND flash simulator. The package type of this MLC flash is dual die, and it employs a two-plane architecture. We built a simulation framework that combines multiple NAND flash simulator instances under a page level address mapping flash translation layer and a garbage collector similar to the one employed in [1]. Eight channels and eight ways are simulated with a FIFO-style NCQ (32 entries on virtual addresses that the FTL provides) [37]. Each channel works at 50 MHz and the frequency of microprocessor used to parallelize data accesses is 800 MHz. For evaluating the effectiveness of our palloc strategies, we chose real enterprise-scale workloads including MSN file storage server (msnfs), shared home folder (usr), financial transaction processing (fin), database management system (sql) [3, 78]. The important characteristics of these traces are given in Table 3.1.

3.5 Results

In order to quantify the performance of our palloc strategies, we used IOPS (as our throughput metric) and average latency. In addition, to better understand the relationship between palloc performances and internal resource usages, we also measured the contribution of

[1]The source code of this simulator [49] can be downloaded from http://www.cse.psu.edu/~mqj5086/nfs.

[2]2 KB page size, page read latency is 50 μsec, page write latencies are varied from 250 μsec to 2.2 *m*sec, and erase time is 2.5 *m*sec [75].

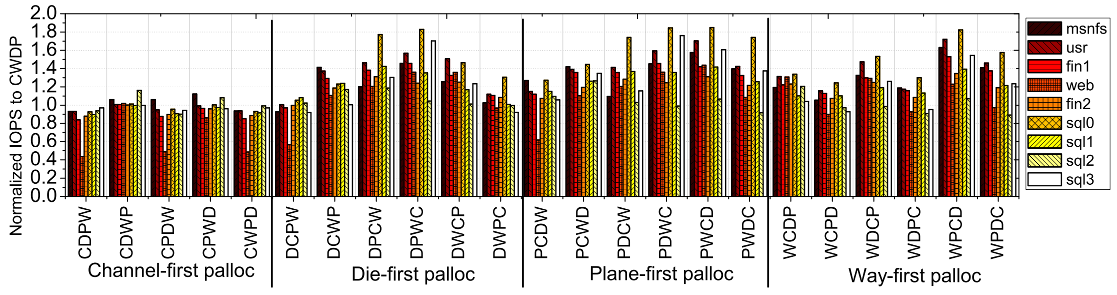


Fig. 3.4 Throughput comparison. IOPS numbers are normalized with respect to corresponding CDPW IOPS.

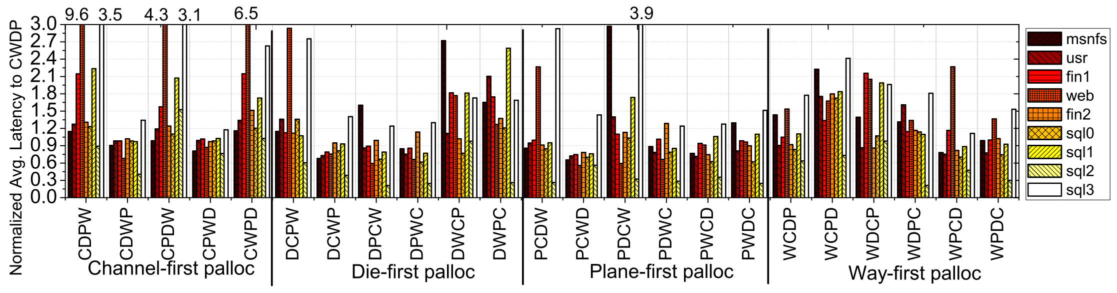


Fig. 3.5 Latency comparison. Latency values are normalized with respect to corresponding CDPW values.

channel, way, die, and plane level parallelism to data accesses and the total number of transactions for all palloc schemes. Finally, we studied the utilization of channels and the fraction of the time spent on different internal resource activities.

3.5.1 Finding Overall Optimal Palloc scheme

Figures 3.4 and 3.5 plot, respectively, IOPS and average latency values for each palloc scheme tested. To enable better comparisons, all IOPS and latency numbers are *normalized* with respect to the corresponding CWDP value, which is reported as being the "optimal palloc scheme" by prior research [31, 42]. We observe that CWDP, DPWC, PWCD, and WDCP exhibit the best latency and throughput number among all channel-first, die-first, plane-first, and way-first palloc strategies, respectively. When all the test cases are considered, one can conclude that **PWCD** is the globally optimal palloc strategy from the performance angle.

	Data Size (MB)	**Write Fraction (%)**	**Avg. Write Size (KB)**	**Avg. Read Size(KB)**	**Randomness (%)**
fin1	18057	84.6	1.5	1	96.9
fin2	8846	21.5	1	1	97.4
msnfs	32490	93.9	10	23.5	87.2
usr	50727	27.0	5	20	92.2
web	15985	0.1	4	7.5	93.5
sql0	30433	40.2	4	14.5	89.9
sql1	4676	14.55	4.5	22.5	73.6
sql2	276407	0.3	10.5	26.5	71.9
sql3	1196	37.12	10	37	56.8

Table 3.1 Important characteristics of our traces.

From a throughput perspective, most die and plane-first palloc strategies provide about 29% better IOPS, compared to channel-first palloc schemes. One of the main reasons behind the better throughput of such strategies is that they exhibit high levels of die and plane locality, helping to build flash-transactions exploiting flash-level parallelism at on-line. Figure 3.6 pictorially shows the total number of flash-transactions measured at the flash chip level. Plane-first and die-first palloc strategies dramatically reduce the number of flash-transactions compared to the palloc strategies that target system-level parallelism. This is mainly because the flash-level parallelism is achieved via advance flash operations, constructed by aggregating multiple incoming requests at runtime. We observe from our experimental results that PWDC and DPWC are able to achieve 82.7% and 81.6% more flash-level parallelism, respectively, than CWDP.

The flash-level resource-first palloc schemes may introduce more bus contention in a channel when the lengths of I/O requests are not enough to span all the elements. Therefore, their latency can be worse than that of the channel-first palloc schemes. As shown in Figure 3.5, most die-first and plane-first palloc schemes provide 11.1% worse latency (as compared to CWDP), which are reasonable considering the significant throughput improvements they bring. Interestingly, PDCW and PDWC show even slightly lower latency compared to CWDP. This is because channel striping in some cases suffers from resource conflicts, between the committed flash operations and the current flash operations. Figure 3.7 presents the waiting times taken to resolve the resource conflicts. As shown in this graph, latencies for palloc schemes are as higher as the waiting time due to longest flash-transactions time. In contrast, high die-interleaving-with-multiplane operation rates (Figure 3.8) of PDCW and PDWC (12.9% $\sim$ 21.4%) result in reduced the overall waiting times.

A comparison of writes vs. reads. Consider a write-intensive workload (msnfs) and a read-intensive workload (fin2). For the write-intensive workload, the channel-first pallocs outperform flash-level resource-first pallocs by 23% (on average), in terms of latency, while their throughputs are on average 34.4% worse than that of the flash-level resource-first pallocs. In contrast, for the read-intensive workload, both the latency and IOPS of the channel-first pallocs are on average 12.5% and 27.9% worse than that of the flash-level resource-first pallocs, respectively. Although not presented here in detial, we believe that one of the reasons why the channel-first pallocs show worse performance than that of a flash-level resource-first palloc in most read cases is that the bus activity fraction of the total execution time for reads, which causes high system-level resource contention, accounts for at least 50.5%, whereas that for writes is as much as 7%.

3.5.2 Parallelism Interference

In order to better understand the cross-interactions among parallelisms at different levels and performance, we categorize all flash-transactions executed based on their operation types. As opposed to the common perception that the system-level and flash-level concurrency mechanisms are largely orthogonal, we observe that channel striping method in system-level makes it hard to exploit flash-level parallelism. Specifically, as shown in Figure 3.8, the percentage of

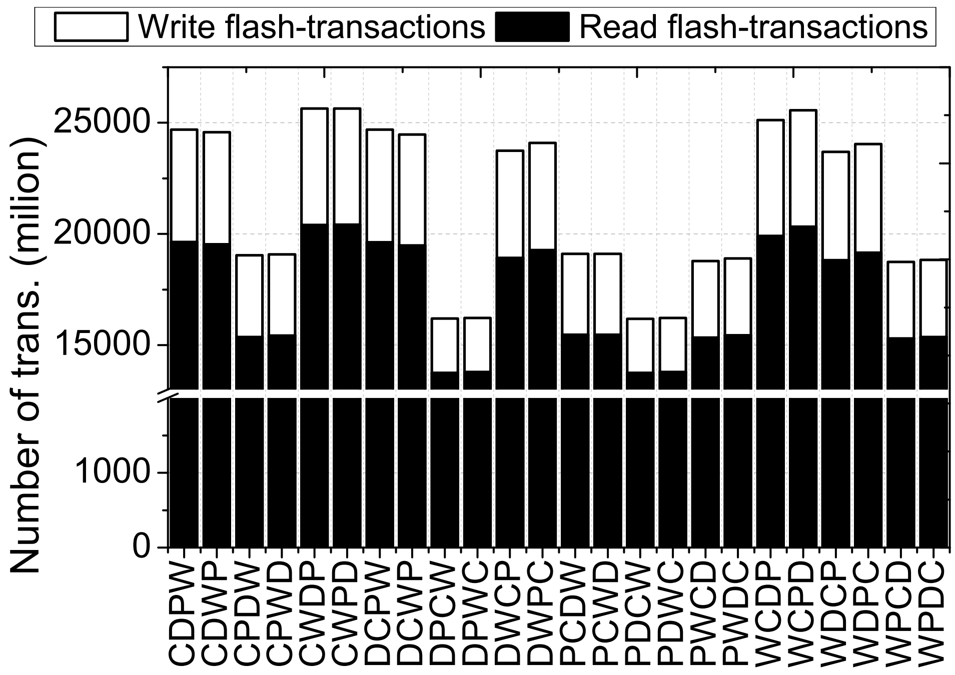


Fig. 3.6 The number of flash-transactions executed.

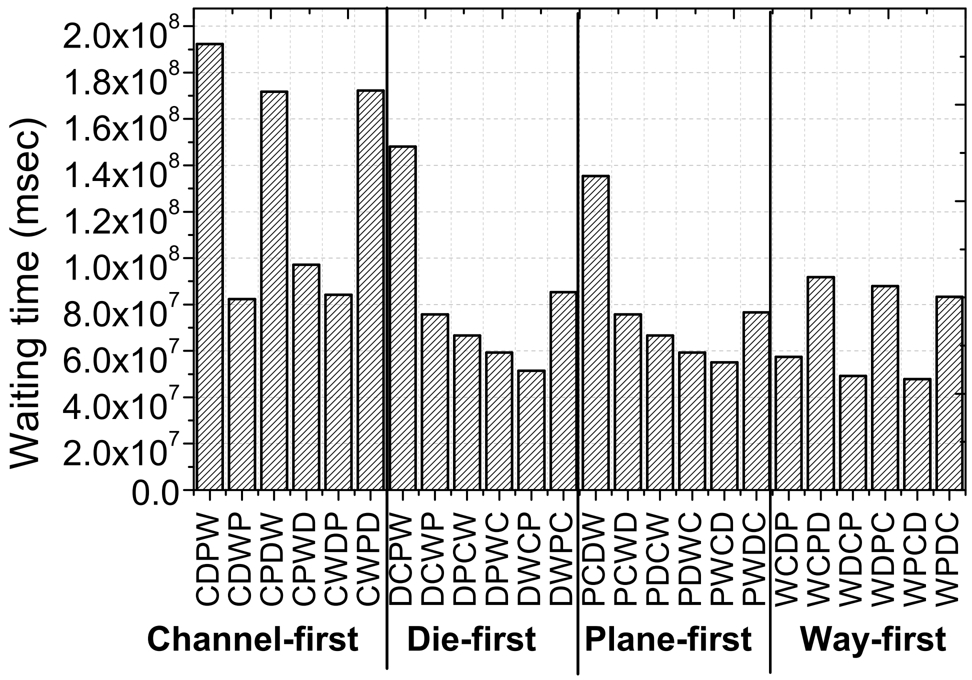


Fig. 3.7 Waiting time required to resolve resource conflicts.

flash-level parallelism exploited by the channel-first palloc schemes shrink as much as 99.8% and on average 44.9%, compared to the plane-first palloc schemes. Even when the way-first palloc schemes are employed, the percentage of flash-level parallelism still shrinks 40.7%. The main reason behind this low flash-level parallelism is that channel and way-first palloc strategies induce poor flash-level locality. With these palloc strategies, the transfer sizes of I/Os are insufficient to span all of dies and planes, and consequently, flash-level parallel data accesses cannot be made at runtime.

3.5.3 Resource Utilization

Increasing resource utilization is another big concern for parallel data accesses. We observe that many internal resources are significantly *underutilized*. Figure 3.9 plots the average channel utilizations under each palloc scheme tested. The channel resource utilization accounts for 43.1% on average with most parallel data access methods. Especially, channel-first palloc schemes exhibit poor channel utilization under disk-friendly workloads even though such schemes are oriented toward taking advantage of channel-level parallelism. Since they cannot commit flash-transactions until the previous requests are completed, when there is a conflict in a flash or channel, these schemes would not be able to achieve high levels of channel utilization.

Figures 3.10 and 3.11 plot the execution time breakdown for the write and read intensive workloads, respectively. One can observe from these results that about 80% of the total execution time are spent idle.

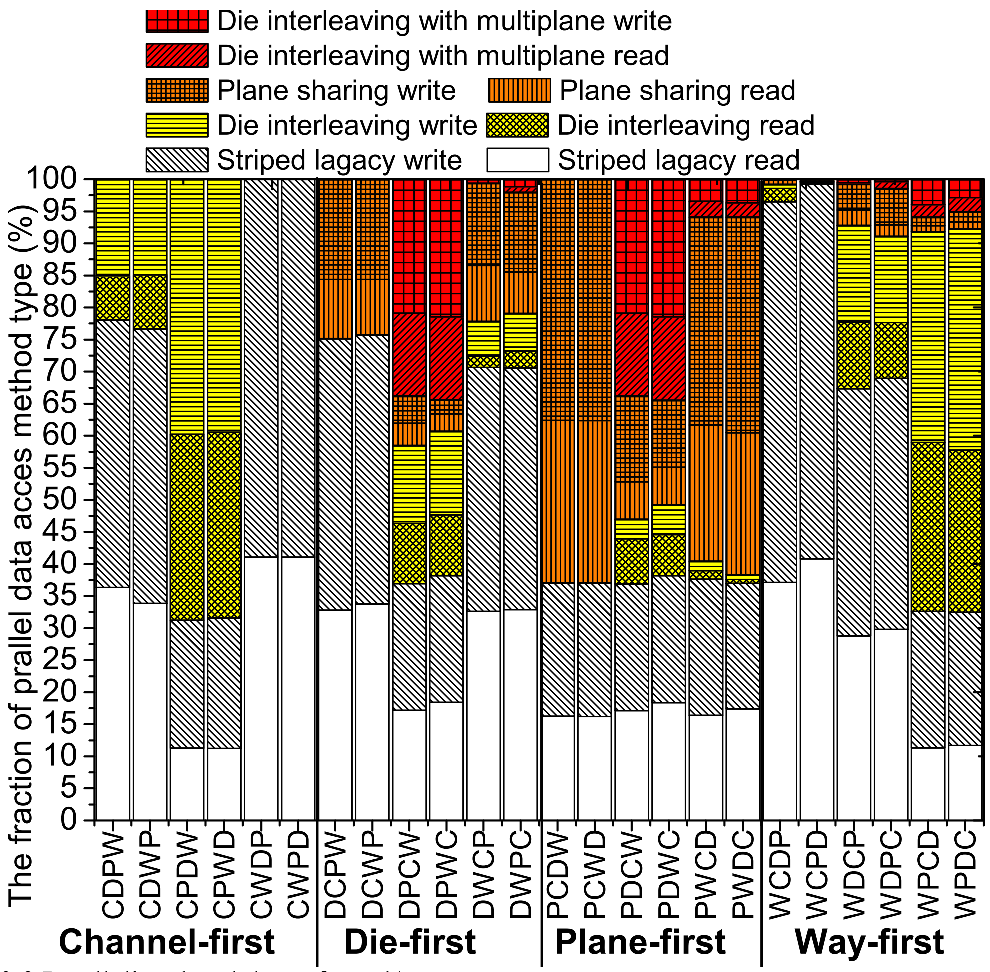


Fig. 3.8 Parallelism breakdown for sql1.

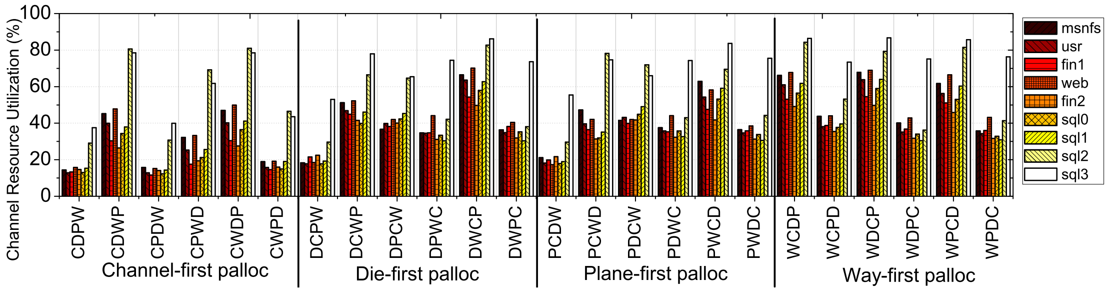


Fig. 3.9 Average channel utilization.

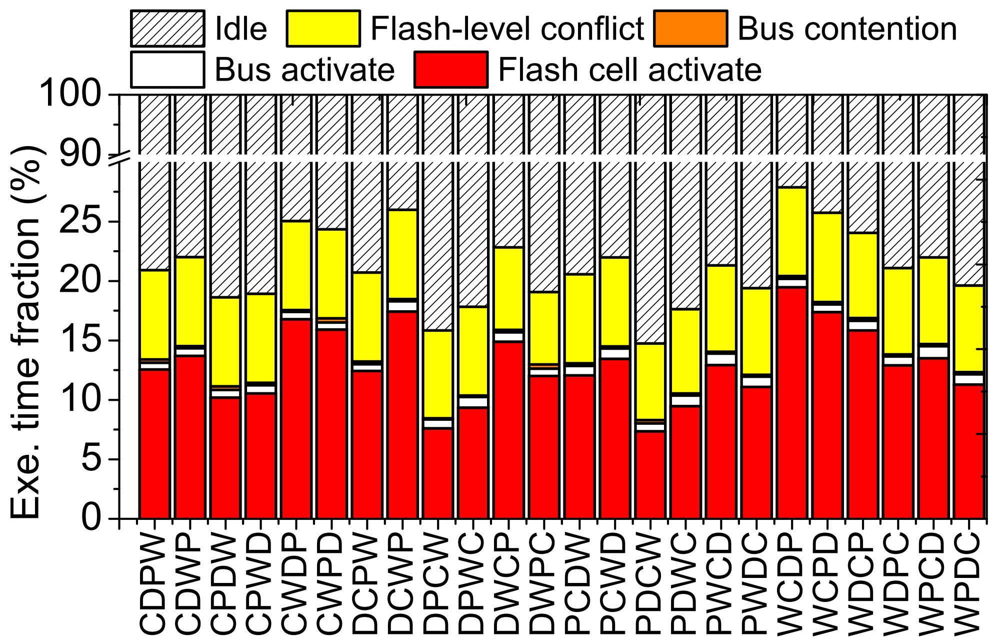


Fig. 3.10 Execution breakdown (msnfs, write-intensive).

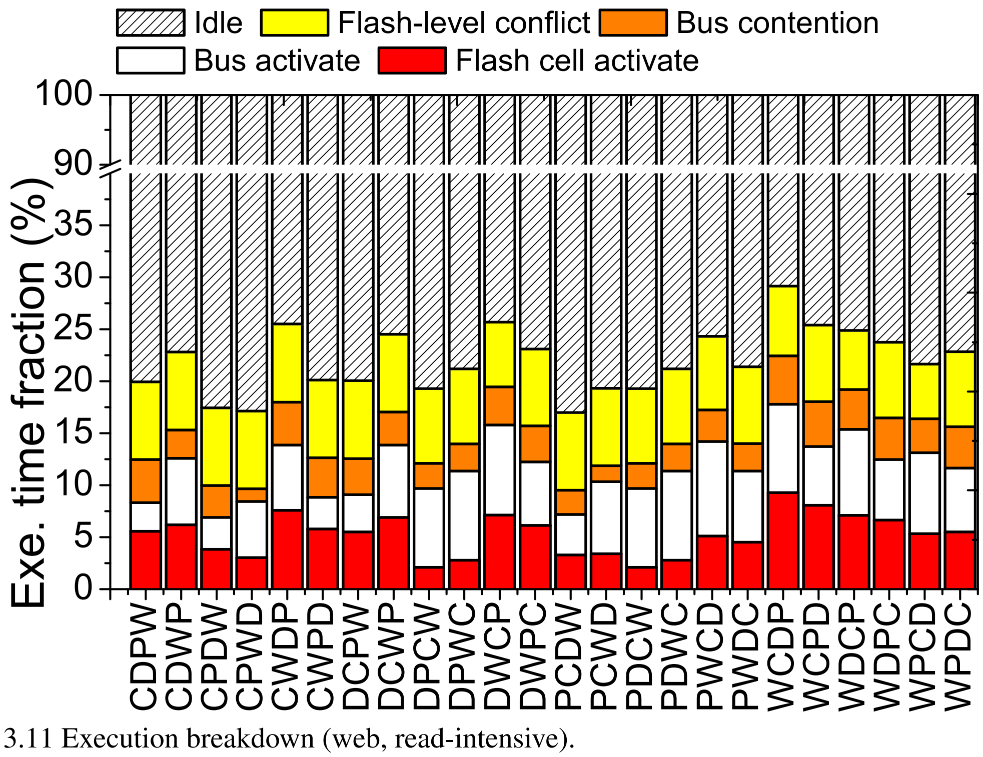


Fig. 3.11 Execution breakdown (web, read-intensive).

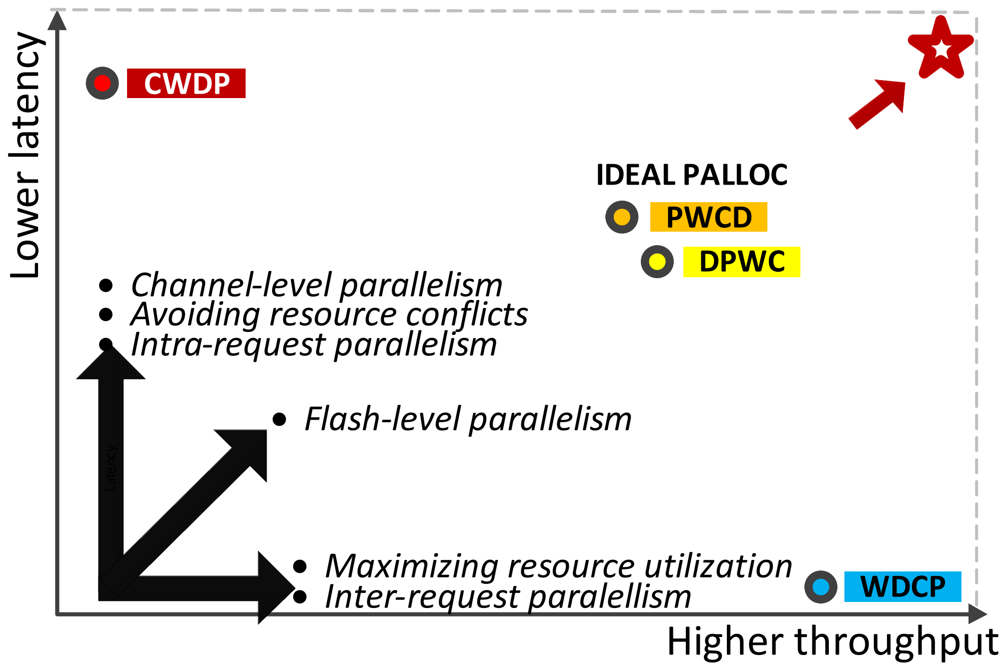

Lower latency
CWDP
IDEAL PALLOC
PWCD
DPWC
• Channel-level parallelism
• Avoiding resource conflicts
• Intra-request parallelism
• Flash-level parallelism
• Maximizing resource utilization
• Inter-request paralellism
WDCP
Higher throughput

Fig. 3.12 Performance map with optimizations points.

3.5.4 Optimization Potential for Parallelism

Based on the I/O access patterns we studied, it can be observed that each palloc scheme exhibits different performance characteristics and optimization points. Figure 3.12 pictorially summarizes the potential of parallelism optimization from both the latency and throughput perspectives. For the latency sensitive applications, channel striping and channel-first palloc give much better position to leverage architectural parallelism. Alleviating resource conflicts is a key to reduce latency and improve inter-request parallelism. In comparison, way- and flash-level parallelism are more suitable for throughput sensitive applications. Maximizing resource utilization is a major factor in exploiting these different levels of parallelisms.

3.5.5 Discussion

As our experimental results demonstrate, although flash-level resource-first palloc schemes generally perform better, their relative performances can vary based on how system-level resources are combined with flash-level resources and how well access patterns are suited to their combination. For example, PDWC favors flash-level resources more than PWCD, but the average throughput of PDWC is slightly worse than that of PWCD. We believe that this is because long data movement time of die-interleaving-with-multiplane of PDWC makes system-level resource contention a little bit more pronounced under certain workloads like usr. Similarly, the performance of WPCD is as good as DPWC. Although WPCD favors the a system-level resource (way), it allocates plane resources first among different ways within a channel, thereby achieving high flash-level parallelism with the plane sharing. Note that, in most cases that we tested, the way-first pallocs are better than the channel-first pallocs in terms of throughput. As mentioned in Section 3.2, the way-first pallocs reap the benefits of inter-request parallelism, which has an impact on improving bandwidth. One of the reasons behind this behavior is that flash-transactions of each I/O request are served within a channel so that several requests in the device queue can be issued over multiple channels in parallel. We however believe that the performance of the way-first pallocs would be degraded when the sizes for each request are larger than the total amount of contiguous physical pages in the channel.

We observe that, when access patterns are fully sequential, the I/O requests span all internal resources, so there is no performance difference between different pallocs.

3.6 Conclusion

This Chapter evaluates all possible page allocation (palloc) strategies using a cycle-accurate SSD simulator. Our experimental results reveal that the channel-first palloc strategies are not the best from a performance perspective, when all levels of parallelism are considered. Further, our results show that flash-level parallelism can be interfered by channel-first palloc schemes, and internal resources are significantly underutilized with most data access methods. We believe our results and observations can be used for selecting the ideal palloc schemes, given a target workload.

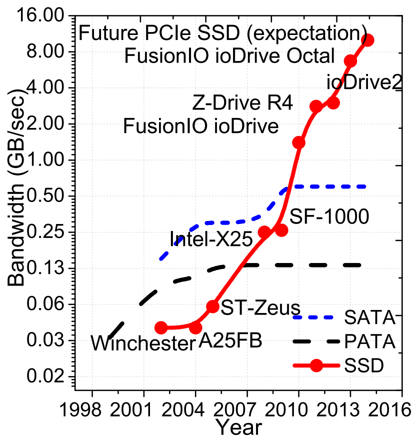
Chapter Acknowledgements Chapter 3, in part, is a reprint of the material as it appears in ”An Evaluation of Different Page Allocation Strategies on High-Speed SSDs,” Myoungsoo Jung, Mahmut Kandemir, in Proceedings of the 4th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage), 2012. The dissertation proposal author was the primary investigator and the first author of this paper.

Chapter 4

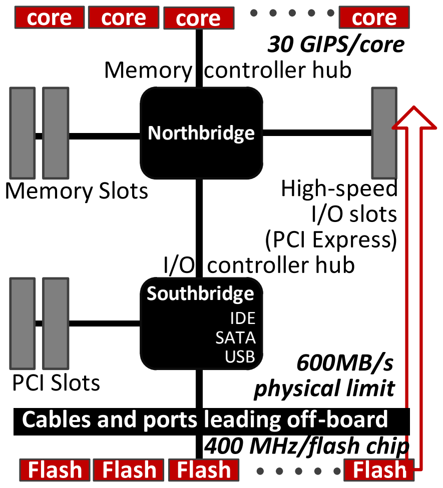
Challenges in Getting Flash Drives Closer to CPU

4.1 Introduction

Over the past few years, NAND Flash-based Solid State Disks (SSDs) are widely employed in various computing systems ranging from embedded systems to enterprise-scale servers to high-performance computing systems, thanks to their high performance and low power consumption. Even though SSDs were originally meant to be a block device replacement or a storage cache that works along with slow spinning disks, their performance has bumped to standard thin storage interfaces such as SATA 6Gpbs, which, at this point, blurs the distinction between block and memory access semantic devices. Figure 4.1(a) plots the bandwidth trends for the thin interfaces versus various SSDs in real world. While the bandwidth of SATA interface has increased from 150MB/s to 600MB/s over a decade, SSDs have improved their bandwidth by four times at the same period. As a result of this remarkable performance improvement, as shown in Figure 4.1(b), both industry and academia started to consider taking SSDs out from the I/O controller hub (i.e., Southbridge) and locate them as close to the CPU side as possible. Obviously, PCIe SSDs are by far one of the easiest ways to integrate flash memory into the processor-memory complex (i.e., Northbridge), which requires no cabling or connections to other I/O devices involved in handling flash memory. By exploiting the benefits of the PCIe interface, latencies are expected to be kept as close to DRAM levels as possible. However, since these SSD technologies consider the PCIe bus as a storage interface, their interfaces are different from conventional memory interconnections at Northbridge as well as thin storage interfaces at Southbridge. The data movement management and underlying flash storage management behind this new SSD interface makes PCIe SSDs a pivotal milestone in the evolution of SSD architecture and software stack designs. Unfortunately, the system characteristics of these new emerging PCIe SSD platforms has received, so far, little attention in the literature, and challenges behind these SSDs and software technologies remain largely unexplored. Further, the public datasheets of SSDs give very little information.



(a)



(b)

Fig. 4.1 Bandwidth trends over time for the thin interfaces versus SSDs (a), and flash storage integration into a place closer to CPU (b).

In this chapter, we quantitatively analyze the challenges PCIe SSDs face in getting flash memory closer to the CPU side and study two representative PCIe SSD architectures and flash software stacks therein: 1) *from-scratch PCIe SSD architecture* and 2) *bridge-based PCIe SSD architecture*. The from-scratch PCIe SSD is built from bottom to top by employing FPGA- or ASIC-based native PCIe controller(s). In contrast, the bridge-based PCIe SSD leverages the conventional high-performance SSD controller(s) by employing an on-board PCIe-to-SAS (or -SATA) bridge controller. Unlike the latter, the from-scratch SSD further optimizes the flash software stack in order to maximize the storage and data processing efficiency. To characterize these two different architectures, we performed a comprehensive set of experiments using two state-of-the-art PCIe SSDs from two different vendors. To the best of our knowledge, our data analysis and presented resource management characteristics on PCIe SSDs are not reported in the literature so far and not studied well in the past. Our main **contributions** can be summarized as follows:

• *Characterizing the performance of the emerging PCIe SSD architectures.* We observe that the latency and throughput of the from-scratch PCIe SSD outperforms the bridge-based PCIe SSD, which is opposite to the information one can get from the datasheets of these SSDs. Specifically, the from-scratch SSD offers on average 29% and 39% shorter latency, and provides 21% and 81% better throughput on reads and writes, respectively. In addition, the from-scratch SSD offers stable write performance in terms of both latency and throughput under a heavy write-intensive workload, while the bridge-based SSD exhibits some sort of write cliff [47], which is a significant performance drop caused by garbage collections.

• *Analyzing host-side resource usages on different flash memory storage stacks.* Even though the from-scratch SSD offers better and sustained performance, it overly consumes host-side resources in terms of memory and computation power, which might be unacceptable in many applications. Specifically, the from-scratch SSD needs about 10 GB host-side memory space for I/O services, whereas the bridge-based PCIe SSD requires only 0.6 GB at most. In addition, it consumes 80% of CPU cycles in completing I/O requests, whereas the latter only needs 23% computation power for the same I/O services.

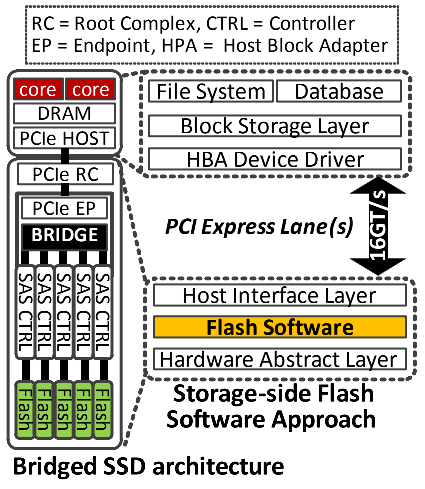
• *Addressing the challenges brought by PCIe SSDs as shared resources.* We observed that the performance of both bridged-based SSD and from-scratch SSD significantly degrades as we increase the number of I/O processing workers. While host-side resources consumption of the bridge-based SSD is not impacted by the number of workers, the from-scratch SSD requires more host-side memory space and more CPU cycles (32% and 160%, respectively). We also found that these emerging SSDs exhibit about 100 times longer latency with device-level queue method compared to the one with the legacy mode.

4.2 Bringing SSDs Closer to CPU

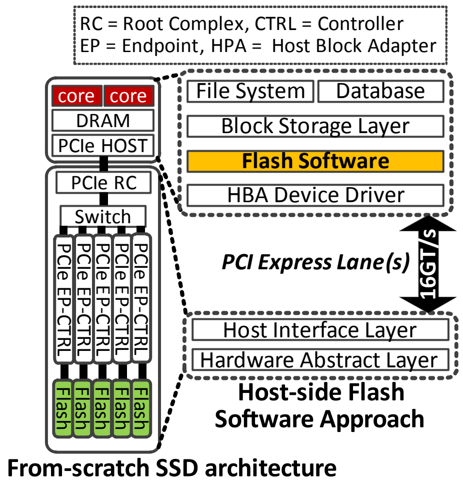
To bring flash drivers closer to the process-memory complex, one needs to achieve shorter latency values with higher throughput than conventional SSD devices. Because of the adapter form-factor of PCIe SSD platforms, which allows them to allot more space in employing multiple flash packages and SSD controllers, PCIe SSDs are in the much better position to reap the benefits of higher parallelism compared to conventional SAS/SATA SSDs. In addition to high PCIe bus capacity, this advantage of parallelism enables PCIe SSDs reduce latency while increasing throughput, as compared to single SAS/SATA SSDs. Further, they also improve the performance of the flash software stack. In this section, we explain two representative PCIe SSD architectures and corresponding flash software stacks.

4.2.1 PCIe Architecture

Bridge-based PCIe SSD. As shown on the left side of Figure 4.2(a), the bridge-based SSD employs multiple traditional SSD controllers, each of which handling the underlying flash packages like a single SAS/SATA SSD. These SSD controllers are also connected to a PCIe-to-SAS (and



(a) Bridge-based SSD



(b) From-scratch SSD

Fig. 4.2 High-level views of our PCIe SSD architectures and their software stacks.

-SATA) bridge controller, which interconnects upper external PCIe link and under internal SAS link. The bridge controller internally converts the PCIe protocol to the SAS protocol (or vice versa) so that it can leverage existing SSD technologies and offer high compatibility. In addition, the bridge controller stripes the incoming I/O requests over multiple SSD controllers, which is similar to what RAID controllers do to improve storage-level parallelism. Consequently, the bridge-based SSD architecture can expose an aggregated SAS/SATA SSD performance to the PCIe root complex (RC) device, which connects the internal PCIe fabric, composed of one or multiple bridge controllers, to the processor-memory complex.

From-scratch PCIe SSD. One of challenges behind the bridge-based SSD architecture is the high performance overheads in internally converting different protocols and in processing I/Os, using the indirect control logic, from CPU to flash memory. Motivated by this, the from-scratch PCIe SSDs have been built from bottom to top by directly interconnecting the NAND flash interface and the external PCIe link, as shown in Figure 4.2(b). Since PCIe is a set of point-to-point links, the connection between the PCIe RC and the flash interface is implemented by one or more switch devices, each internally handling multiple PCIe endpoints (EPs). The PCIe EP has independent upstream and downstream buffers, which control the in-bound or out-bound I/O requests in front of the flash memory. This scalable architecture can easily expand the storage capacity by putting more flash chips into its PCIe network topology and straightforwardly expose true NAND flash memory performance to the upper processor-memory subsystem. These PCIe EPs and switches are typically implemented by FPGA or ASIC as a form of native PCIe controller, and the flash software can be optimized to reduce latency and offer better throughput, as discussed below.

4.2.2 Flash Software Stack

Storage-side Flash Firmware. Typically, the flash control modules are implemented in the storage side for most conventional SSDs and bridge-based PCIe SSDs as “flash firmware”. In this storage-side flash software stack, a hardware abstract layer (HAL) handles low-level NAND flash commands and manages the I/O bus for moving data between SSD controller and internal registers of individual flash memories, as depicted on the right side of Figure 4.2(a). On top of the HAL, the main flash software modules are built, which include the flash translation layer (FTL), buffer cache, wear-leveler and garbage collector. Among the flash software modules, the FTL is the core logic in managing flash memory, and translates addresses from virtual to physical. Finally, a host interface logic atop the flash software is mainly responsible for the protocol conversion, parsing requests, and scheduling them. This conventional flash software

	From-scratch SSD	**Bridge-based SSD**
Code-name	**(FSSD)**	**(BSSD)**
Interface	PCIe 2.0 x8	PCIe 2.0 x8
Flash Software Module	Host-side kernel driver	Storage-side firmware
Price	\$2490	\$2152
Controller Type	Xilinx FPGA	SAS-to-PCIe Bridge
Storage capacity	430GB	400GB
Write Bandwidth	700MB/sec	750MB/sec
Read Bandwidth	1GB/sec	1.4GB/sec
512B I/O Latency	45μsec	65μsec
Flash Type	QDP MLC	eMMC
Internal DRAM	Publicly N/A	2GB
Debut	2012 Q2	2012 Q3

Table 4.1 Important characteristics of the tested PCIe SSDs.

stack lets SSDs expose the underlying flash memory to the processor-memory complex without any host-side storage stack modification.

Host-side Flash Software Module. Flash software could manage the underlying flash memory more efficiently if it is possible to access the host-level resources such as file system and incoming I/O request information. Consequently, there exist several prior proposals to migrate the flash software to the host-side, as illustrated on the right side of Figure 4.2(b). In addition, by implementing the flash software modules on the host side, we can 1) unify indirect flash software logic [2, 103, 41] and 2) overlap storage and data processing times by exploiting abundant host-side computation and memory resources [95, 46]. Specifically, [41] proposes virtual storage layer (VSL) and direct file system (DFS) by migrating the flash software module from the storage side to the host side (especially FTL), so that it can optimize data accesses as well as offer extensive OS support. [103] unified FTL and the host-side file system to remove indirect address mapping, and [95] moved the internal buffer cache to the host-side to improve performance when targeting write-intensive workloads. [46] migrated garbage collector and page allocator [45] from SSD to the host-side software stack. Thanks to this flash software module migration, a from-scratch SSD can maximize throughput while reducing latency.

4.3 Experimental Setup

PCIe SSDs. We chose two most-recently-released, cutting edge PCIe SSDs from two different vendors. Since our goal is not to perform reverse engineering of these commercial products, we refer to each of them using a code-name – *FSSD* refers the from-scratch SSD, and *BSSD* refers to the bridge-based SSD. Our SSDs and their important characteristics are listed in Table 4.1. It should be noted that, even though these two architectures have been built based on very different deign concepts, both PCIe SSDs are geared toward offering shorter latency and better throughput, and designed for workstations.

System Configuration. Our experimental system is equipped with an Intel Quad Core i7 Sandy Bridge 2600 3.4 GHz processor and "16GB" memory (four 4GB DDR3-1333Mhz memory). In this system, all of the functions of the Northbridge reside on the CPU, and all SSDs we tested are connected to Sandy Bridge through the PCIe 2.0 interface. We executed all our tests in NTFS, and stored logs and output results into separate block devices in a full asynchronous fashion; *neither a system partition nor a file system is created on our SSD test-beds*. Note that this configuration allows each SSD test-bed to be completely separated from the evaluation scenarios and tools.

Measurement Tool. We modified an Intel open source storage tool, called *Iometer* [35], to capture time series of performance characteristics and host-side memory usage. To measure accurate memory usage at a given time, we added a module in calling *GlobalMemoryStatusEx* into Iomoter, which is an Window system function that allows users to retrieve the current state of both physical and virtual memory. In addition, to minimize interference between successive evaluations, our modified Iometer physically erased whole region of underlying device though *secure erase command* in SMART, in every evaluation step.

4.4 Challenges in Resource Management

4.4.1 Memory Usage

Overall memory usage evaluation. One can see from Figure 4.3 that FSSD needs at least 2GB memory for writes and 1.5GB memory space for reads while BSSD requires only 0.6GB memory space regardless of the I/O type and size. As a result of flash software migration, the host side kernel drivers require memory space in loading their image and containing their in-memory structures. In addition, we believe that there are two main reasons why FSSD consumes on average sixteen times more memory space than BSSD. First, the unified file system [41] and migrated flash software [2, 103, 46] require host-side memory to maintain huge mapping tables. Second, the host-side write buffer cache consumes memory space in hiding the underlying flash memory complexity, such as garbage collections, endurance [4] and intrinsic latency variation [47]. For instance, as shown in Figure 4.3, FSSD's memory usage varies based on access granularity and pattern, whereas the BSSD's memory usage is not different by them. This is because, in the bridge-based SSD architecture, the table is implemented in the SSD, and data processing is only performed at the storage side.

Time series analysis. Figure 4.4(a) plots memory usage of FSSD (top) and BSSD (bottom) over time. In this test, we evaluated them with a 512B block access granularity since all the system-level operations are block-based, and the default block-size is 512B. In addition, we performed the memory usage test based on two different I/O access scenarios: 1) queue mode

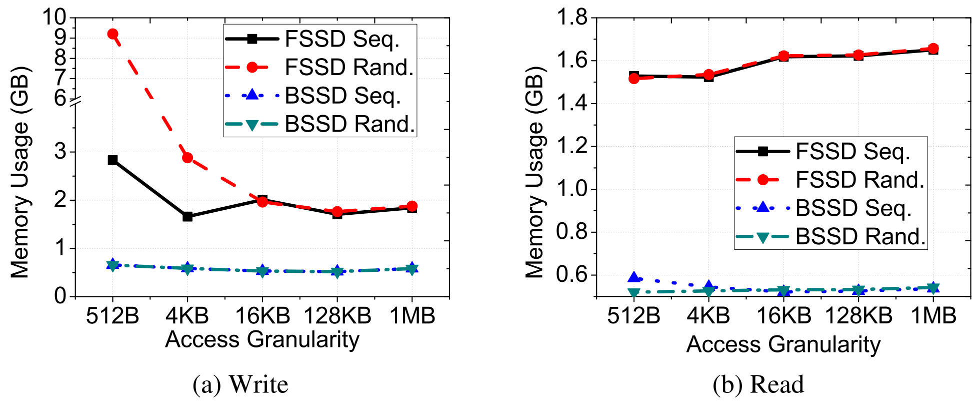
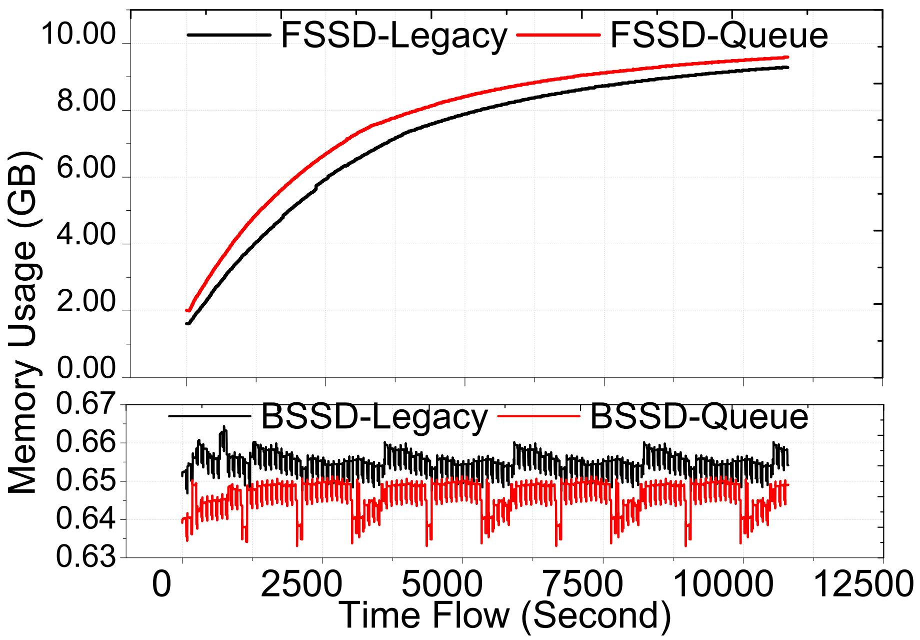
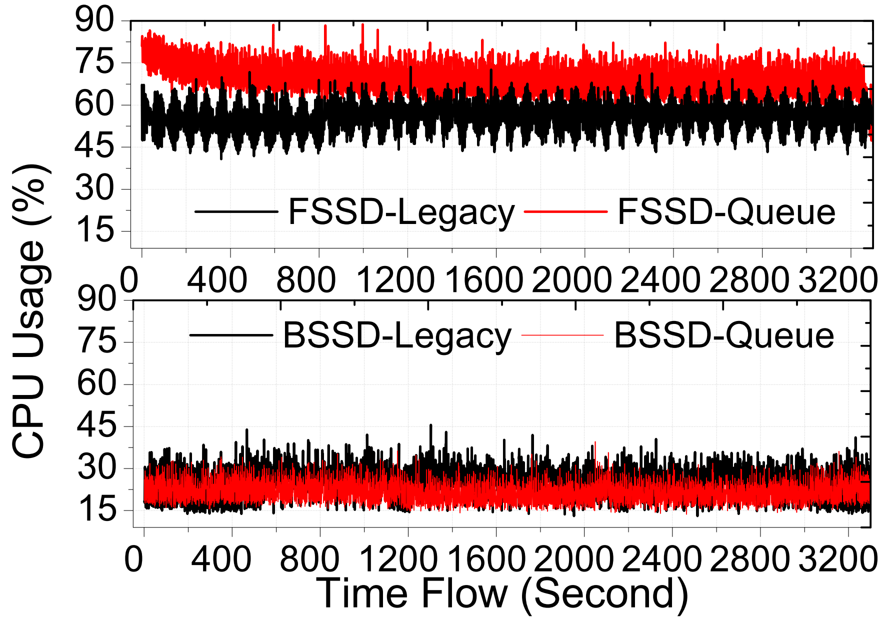


Fig. 4.3 Memory usage.

operation (using 128 queue entries) and legacy mode operation (submitting the request whenever the device is available to serve an I/O request). One can see from the figure that FSSD requires about 2GB memory space at the very beginning of the I/O process for both queue and legacy mode operations. Interestingly, as the I/O process progresses, the amount of memory usage keeps increasing in a logarithmic fashion and reaches about 10GB. It should be noted that, considering that the target system is a workstation, we believe that 10GB memory usage to manage only the underlying SSDs may not be acceptable in many applications. In contrast, as shown in the bottom part of the figure, BSSD keeps memory usage around 0.6GB over time. We believe that the reason why the memory usage of FSSD keep increasing over time is because of the host-side address mapping and caching. In particular, DFS/VFS [41] uses a B-tree structure to map addresses between the physical and virtual spaces, which tends to increase the memory requirements of the mapping information by adding more node entries to serve incoming I/O requests. We also believe that the huge memory usage is primarily caused by host-side buffer caching.

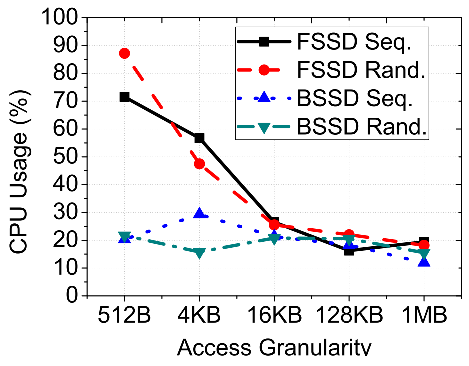


(a) Memory usage comparison

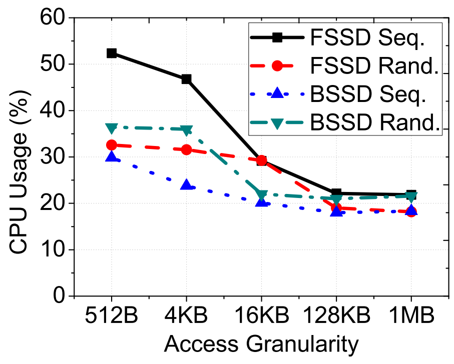


(b) CPU usage comparison

Fig. 4.4 Time series comparison for host resources usage between FSSD (top) and BSSD (bottom) with the default 512B block-size access operation.



(a) Write



(b) Read

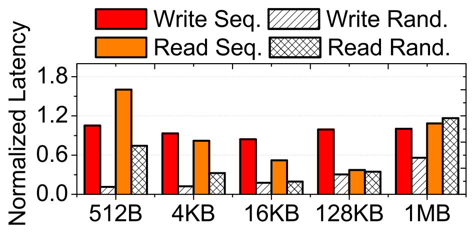
Fig. 4.5 CPU usage.

4.4.2 CPU Usage

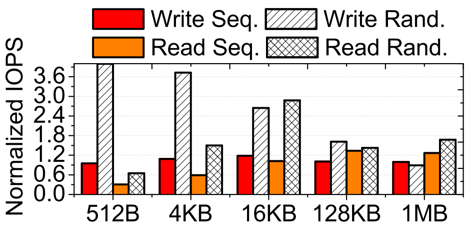
Overall CPU usage evaluation. Figures 4.5(a) and 4.5(b) give CPU usage on the host-side in serving reads and writes, respectively. Similar to memory usage analysis, FSSD requires computation power about three times more than BSSD, except for cases where access granularity is larger than 16KB. We conjecture that one of main reasons why FSSD requires higher CPU usage (52%∼87%) for finer granular I/O accesses is that smaller size I/O requests leads to an increase in the size of the address mapping table lookup and update (or cache lines of host-side buffer). In contrast, BSSD only consumes 20%∼30% for the same I/O services. This is because the mapping table lookup and update processing are performed on the storage-side.

Time series analysis. Figure 4.4(b) compares the CPU usage of FSSD (top) and BSSD (bottom) under the workloads that exhibit high number of default block size accesses. As before, we evaluated our PCIe SSD test-beds with both legacy and queue mode operations. FSSD consistently consumes 60% of the cycles on the host-side CPU with legacy mode operations, and I/O service with queue mode operation requires 50% more CPU cycles than the legacy mode. We believe that a CPU usage over 60% for just I/O processing can degrade overall system performance. In contrast, BSSD only uses about 20% CPU cycles irrespective of the I/O operation mode.

4.4.3 Challenges in System Performance



(a) Latency



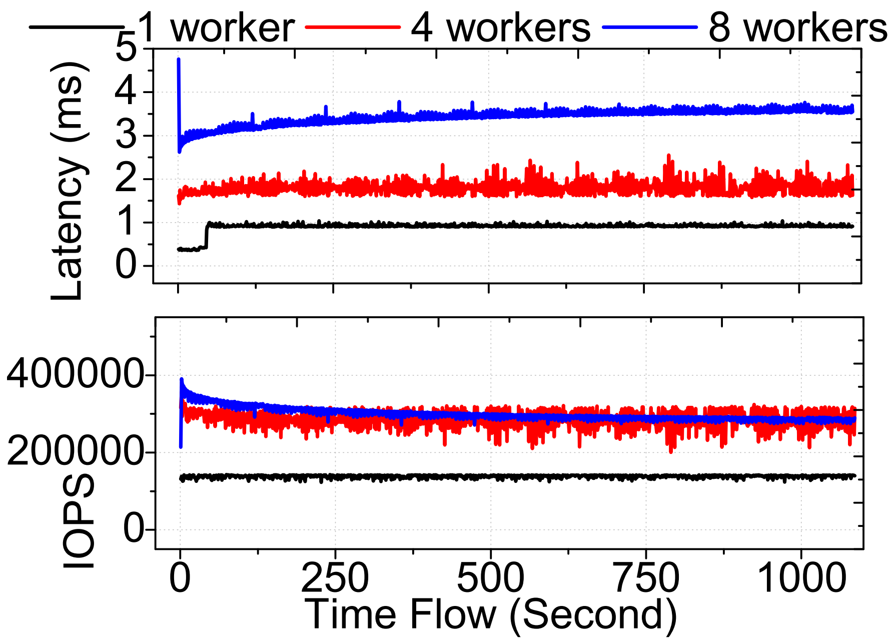
(b) IOPS

Fig. 4.6 Latency and throughput comparison. Note that all the latency and throughput of BSSD values are normalized to corresponding values of FSSD.

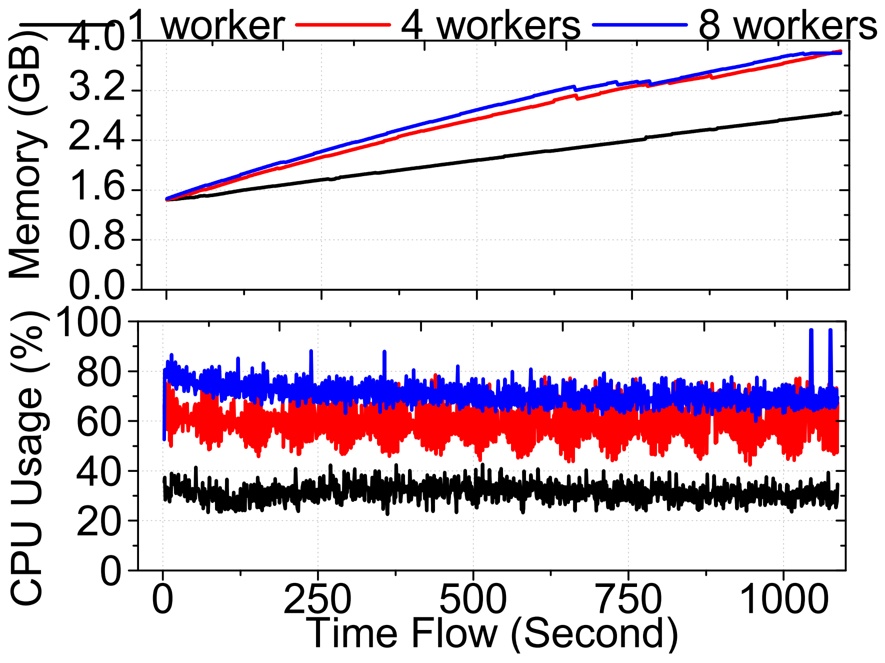
Overall performance comparison. It is hard to directly compare the microscopic performance characteristics on the two different SSD architectures since their flash software and platforms have different optimization techniques. For example, we observed that BSSD's random writes with default block-size accesses exhibit 7.2 times better performance compared to sequential writes, which is opposite to common expectation on most modern SSDs. We believe that this is because BSSD puts incoming default-block size I/O requests into its internal 2GB DRAM buffer and additional non-volatile SRAM [18], but forwards the large sized I/O requests to the underlying flash memory, which in turn shows the unexpected performance. Consequently, we compare the overall performances of BSSD and FSSD. Figures 4.6(a) and 4.6(b) compare the latency and IOPS between FSSD and BSSD. We see that most latency values observed with BSSD are on average 39% worse than FSSD, which is opposite to the information one could obtain from the datasheets of these SSDs (see Table 4.1). We think that multiple controllers, indirect address mapping modules, and protocol conversion overheads of BSSD on data path from CPU to flash memory contribute to this longer latency.

Multi-core system environment. To evaluate the performance impact in a multi-core system environment, we executed one to eight I/O processing workers on FSSD and BSSD in parallel. The results considering FSSD and BSSD as a shared resource are plotted in Figures 4.7 and 4.8. The latency of both FSSD and BSSD increases as we increase the number of workers. Specifically, latency values with eight workers on FSSD and BSSD are worse than four workers by 118 % and 108%, respectively and worse than single worker by 289% and 704%, respectively. Throughput trends are a bit different compared to latency trends. While BSSD has no IOPS benefits by increasing the number of workers, the IOPS of FSSD increases. The IOPS of four workers are 2.2 times better than single worker evaluation. However, the advantage of many workers decreases because of the higher memory and CPU usages. In contrast, BSSD shows similar IOPS and host-side resource usages irrespective of the number of workers employed.

Queuing latency. Device-level queueing mechanisms are one of the crucial components, which can improve storage throughput. For example, NVMe offers 64K queue entries [36]. SAS/SATA [96, 37] also provides a device-driven queue mechanism, which allows the storage devices to determine the order of I/O request executions without any host-side software interrupts. However, we observed that the latency values with a queuing method significantly drop irrespective of the SSD architecture. As shown in Figure 4.9, the random and sequential write latencies of FSSD are longer than legacy mode latencies by about 106 times. Similarly, BSSD resulted in 99 times worse latency with the queue mode operation than the one with legacy mode. We believe that these significant latency drops with the queue mode operation would be a problematic obstacle to bring flash memory closer to the memory-processor subsystem.

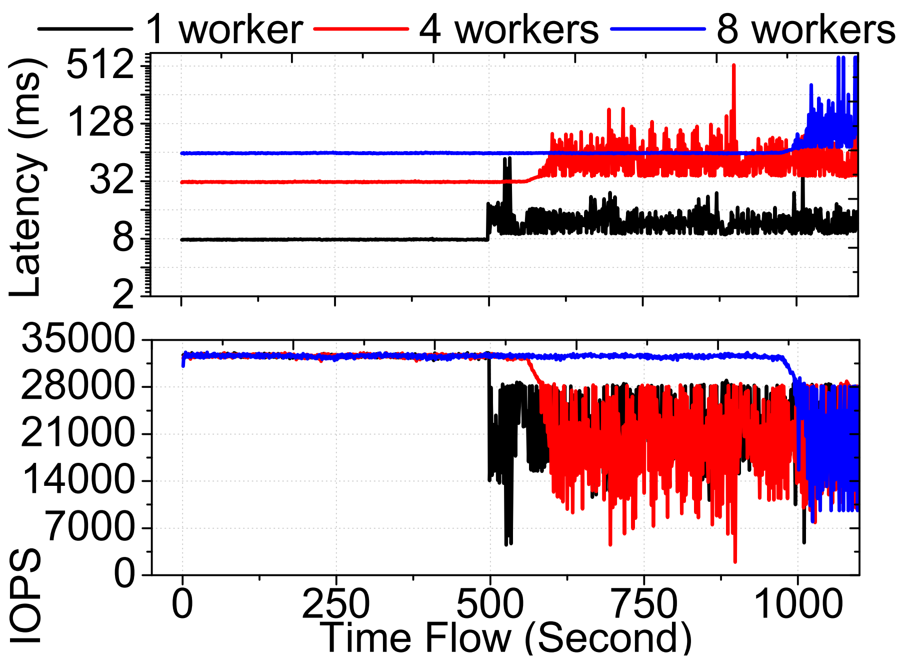


(a) Performance

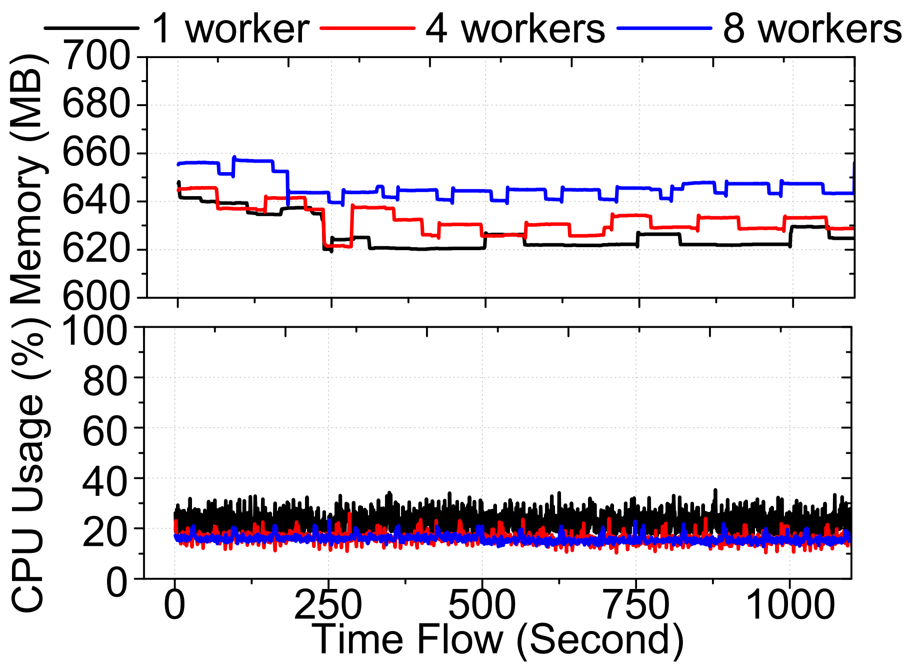


(b) Resource

Fig. 4.7 FSSD performance characteristics on the multi-core environment.

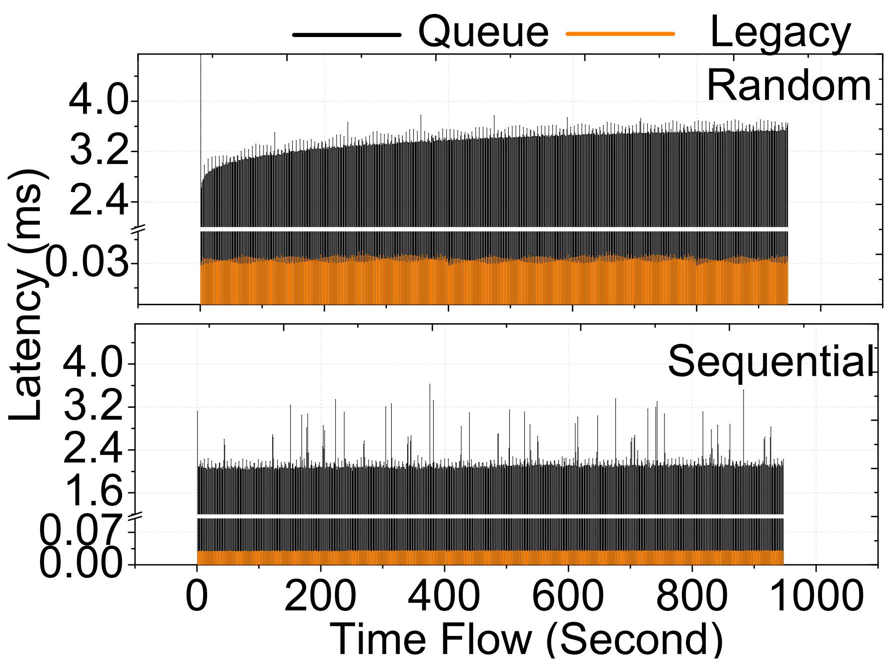


(a) Performance

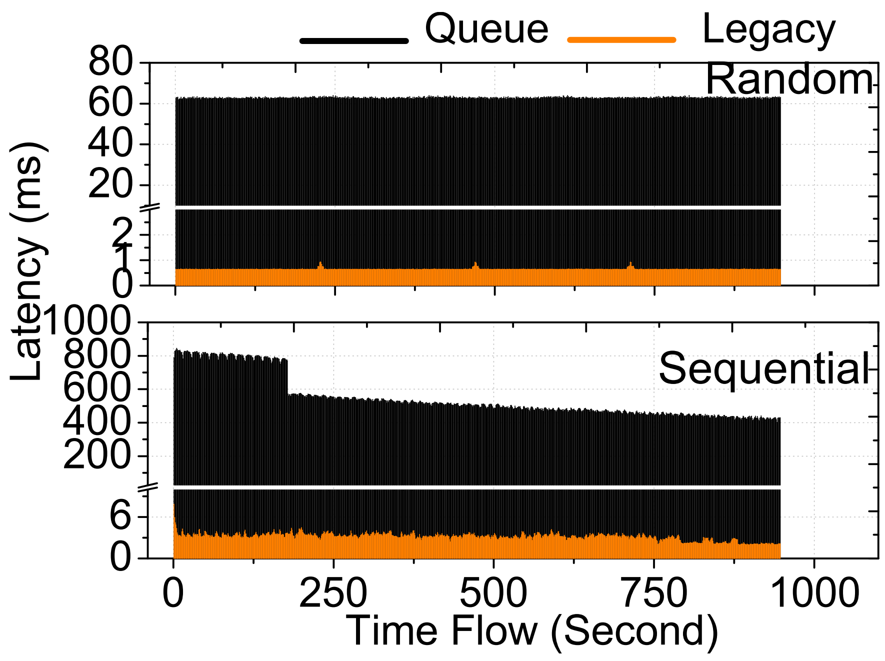


(b) Resource

Fig. 4.8 BSSD performance characteristics on the multi-core environment.



(a) FSSD



(b) BSSD

Fig. 4.9 Queueing latency comparison observed by FSSD (a) and BSSD (b).

Garbage collections. Figures 4.7(a), and 4.8(a) also tell us the difference between FSSD and BSSD in managing underlying garbage collections (GCs). While FSSD offers very sustained performance, the latency and throughput values of BSSD drop starting with the half of I/O execution. This is mainly because of GCs, which are a series of SSD internal tasks reading data from old flash block(s), writing them to new block(s), and erasing the old block(s). This performance drop caused by GCs is also referred to as *write cliff* [47]. One of the reasons behind the sustained performance of FSSD is the ample host-side buffer and the optimized flash software stack. It should be noted that BSSD also employs a 2GB internal memory as buffer, but it cannot hide the GC overheads, which means that the sustained performance of FSSD does not solely come from the available buffer.

4.5 System Implication

Co-operative approach. In summary, we observed that, while the performance of from-scratch SSD is better than the bridge-based SSD, the former requires huge host-side resources, which may not be acceptable in many cases. We believe that an approach that partially migrates flash software functionalities from SSD to the host-side can be a promising mid-way option in achieving higher performance and lower host-side resource consumption. For example, FTL partitioning [43] moves only the address mapping module rather than moving the whole FTL cores (e.g., buffer cache, wear-leveler). Similarly the middleware and firmwmare cooperative approaches [46] only move the garbage collector, and I/O scheduler [50] is aware of internal parallelism from the storage-side to the host-side, which requires less system memory resources.

All-flash storage arrays. PCIe SSD based all-flash arrays or SSD RAID systems can directly experience the host-side resource challenges we demonstrated so far. For example, if system designers build a 5-RAID system based on FSSD, it requires approximately 50GB memory space, and they have to carefully design the processors to manage the underlying five FSSDs. If the designers build the RAID with BSSD and intend to improve performance by striping all incoming I/O requests over the five BSSDs, they need to carefully manage GCs globally because the probability that the system could suffer from a straggler performing GCs significantly increases.

Chapter Acknowledgements Chapter 4, in part, is a reprint of the material as it appears in ”Challenges in Getting Flash Drives Closer to CPU,” Myoungsoo Jung, Mahmut Kandemir, in Proceedings of the 5th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage), 2013. The dissertation proposal author was the primary investigator and the first author of this paper.

Chapter 5

Revisiting Widely Held SSD Expectations and Rethinking System Level Implications

5.1 Introduction

NAND Flash-based Solid State Disks (SSDs) have recently become immensely popular and been employed in different types of environments ranging from embedded systems to personal computers to high performance computing (HPC) systems. Moreover, various memory and storage systems have been proposed to take advantage of the performance benefits of SSDs over conventional block devices. For example, to reap the benefits of high bandwidth on writes, prior HPC studies consider SSDs as a burst buffer [68], which can absorb heavy write traffic caused by check-pointing [83]. There also exist many applications developed under the expectation that NAND flash is biased toward reads in terms of performance and reliability. Enterprise servers, for example, consider employing SSDs for applications that exhibit many random reads [69, 92, 95] or use them as read caches [54, 7, 61, 82], sitting between main memory and hard disk drive (HDD). Similarly, SSDs are also introduced as a main memory replacement, memory extension, and a part of existing virtual memory systems [24, 23, 25, 92].

While many of these SSD applications and usage scenarios are proposed and developed based on common expectations from SSDs, modern SSDs and NAND flash systems have undergone severe technology shift and architectural changes in the last couple of years. Specifically, NAND flash cells have shrunk from 5x *n*m to 2x *n*m in the past four years, and now fewer electrons are stored per floating gate. These cell-level characteristics make flash devices less reliable and introduce extra operations (e.g., multi-step I/O, verification, error correction processes) to successfully complete I/O requests, which in turn imposes longer latencies. State-of-the-art NAND flash packaging technologies employ an increased number of planes and dies within a single flash chip, a command queue, ECC engines, and faster data movement interfaces [17, 81]. These technological changes led in turn to modulations in SSD behavior and performance characteristics. In parallel, SSD internal architecture has dramatically changed; modern SSDs now employ multiple internal resources such as flash chips, I/O buses, controllers and cores in an attempt to achieve high internal parallelism. In addition, to reduce performance variations and garbage collection overheads, flash firmware employs advanced strategies such as finer-granular address mappings, DRAM buffer and background tasks. Finally, thin storage interfaces of modern SSDs define command feature sets, which provide a way to efficiently expose underlying SSD characteristics to operating systems (OS). Consequently, OS can manage SSD internal resources more efficiently by utilizing system level information.

Unfortunately, most prior works study SSD behavior and performance characteristics based on limited information, or evaluate them based on select I/O access patterns to understand SSD-level parallelism and performance implications. In our opinion, these studies do not help OS and system designers in understanding critical SSD features, and integrating SSDs into existing storage stacks and efficiently optimizing them. Further, a more problematic issue is that, even though SSD NAND flash technology has changed dramatically over the last couple of years, many research groups still employ SSDs based on assumptions that do no hold anymore.

In this chapter, we conduct an extensive experimental evaluation with six state-of-the-art SSDs carefully selected by considering different types of flash fabric technologies, manufacturers, cores, chips, and over-provisioning strategies. Based on our empirical evaluations, we next perform a comprehensive data analysis and uncover critical SSD/flash characteristics, which are not reported, to the best of our knowledge, in the literature so far, and are opposite to the widely held expectations on SSDs. Our main goal is to correct common misconceptions on SSDs using new data, which greatly effect performance as well as reliability of modern SSDs, but have not been studied well in the past. We hope to motivate both academia and industry to rethink SSD system design, management and optimization based on our evaluations and data analysis. In this chapter, we answer, either directly or indirectly, several questions described in the following subsections, and reveal some critical data regarding state-of-the-art SSDs, which should be, in our opinion, taken into account by both OS and SSD designers. The questions we want to address can be categorized into five groups.

5.1.1 Rethinking Read Performance

A well-known intrinsic characteristic of NAND flash is that their read performance is tens to hundreds times better than their write performance [1]. In addition, since SSDs have no moving parts, they are expected to provide fast random read accesses [92]. Motivated by these, many platform designers consider SSDs for the applications that contain mostly random reads.

1. Are SSDs biased toward reads at a system level? Why do random reads constitute a performance bottleneck in modern SSDs?
2. For the sequential read accesses, could SSDs support sustained performance? Is there any performance degradation on reads? How could a system achieve a sustained read performance?
3. What is the relationship between read performance and internal SSD parallelism? Can users characterize read performance by examining different I/O access patterns?

5.1.2 Examining Reliability on Reads

Unlike writes, reads require no erase operation or content-update on NAND flash. Consequently, many computing domains exploit SSDs as a read cache or an intermediate layer when targeting read-intensive workloads to extend SSD lifetime and avoid heavy write penalties.

1. Do program/erase (PE) cycles of SSDs increase during read-only access periods? If it is, why do reads need a block erasure?
2. Are there performance impacts caused by internal I/O operations on reads?
3. What parameters do system-level designers need to control in order to extend SSD lifetime?

5.1.3 Reconsidering Write Performance

Many schemes have been proposed by prior SSD research to reduce garbage collection (GC) overheads such as over-provisioning [30], DRAM buffer [56, 40, 44], and finer-granular address mappings [52, 28, 1]. Based on this, it is expected that long GC operations can be reordered and deferred, and therefore, they do not cause severe throughput degradation at a system level.

1. How much impact do GC latencies have on system performance in practice?
2. Is there any relationship between the worst-case latency of GCs and system throughput? Could we quantify this relationship?
3. Could DRAM buffer help firmware in reducing the GC overheads? If not, why?

5.1.4 OS Support

TRIM commands enable OS to invalidate deleted system-level data contents, which can in turn reduce GC overheads significantly. Motivated by this, many emerging SSD platforms (e.g., flash virtual memory, file system, database) are expected to send TRIM commands to underlying SSDs as much as they can.

1. In theory, OS and users can eliminate unnecessary GC operations through TRIMs. How much of GC overheads can be eliminated using TRIM commands?
2. Does TRIM command request pattern matter?
3. Do TRIM commands themselves impose any overheads?

5.2 Preliminaries

State-of-the-art SSDs are composed of multiple cores, memory modules, data buses, and storage media. In the following, we provide a quick overview of SSDs and NAND flash, basic flash firmware features, storage interfaces, and reliability issues.

5.2.1 SSD and NAND Flash Internals

Modern SSDs and NAND flash chips employ several components to scale their performance under a given technology. As shown in Figure 5.1, an SSD employs multiple internal resources described below.

Controllers. For physical layer (PHY) management, SSDs have two different controllers: 1) non-volatile memory host controller (NVMHC) and 2) flash channel controller (FCC). NVMHC manages the front-end PHY layer to communicate with outside through a conventional thin interface/bus. FCC, in contrast, handles the back-end PHY layer to control underlying flash packages and corresponding interfaces.

Multicore. While the controllers are responsible for handling the PHY layer, the embedded processor is dedicated to running the flash firmware, which is composed of multiple software layers. Since each layer of the firmware has a different goal and some of their functionalities can be parallelized, modern SSDs employ multiple cores (or processors) in an attempt to minimize computation overheads [50, 45].

Multiple Channels. The flash package interconnection design is very important from a parallelism angle. In general, considering hardware complexity and signal integrity, several flash packages are connected to a single data bus, referred to as *channel*, and multiple channel are used in SSDs.

Flash Package. A flash die is composed of multiple planes, and multiple dies are stacked in a single flash package, which helps one improve storage capacity and flash-level parallelism. Multiple requests can be interleaved through a limited number of CE (chip enable) and I/O pins. In addition, modern flash packages employ a queue and an ECC engine to offload system level overheads imposed by flash commands management and error code checking, respectively.

5.2.2 Flash Firmware

Depending on the specifics of the underlying hardware configuration, the role of flash firmware can be quite diverse. We now explain the common tasks of the firmware, which have an impact on system performance.

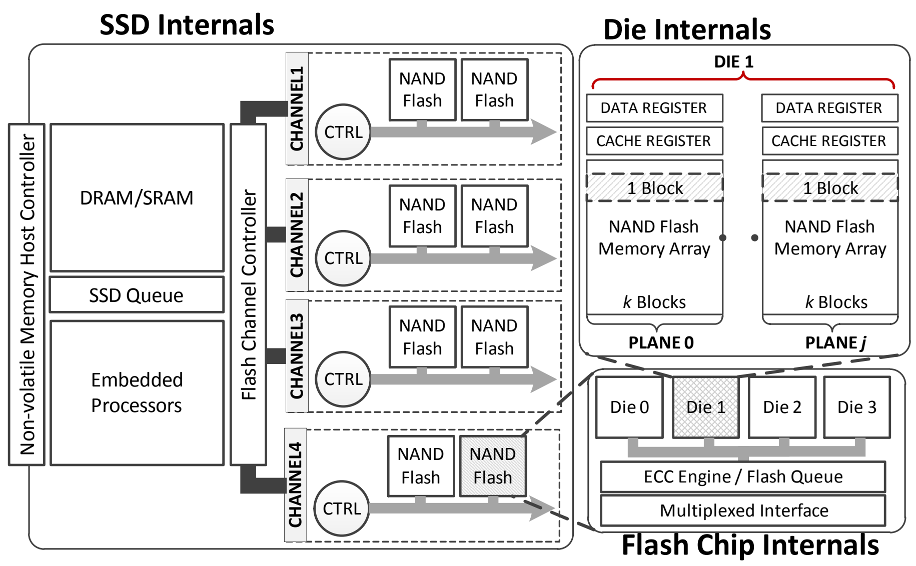


Fig. 5.1 Modern SSD internal architecture. Note that an I/O request can be simultaneously served by many internal resources, which is one of the important characteristics of SSDs.

Parallelism. Flash firmware can strip an I/O request over multiple channels, flash packages, and dies therein, in order to improve system performance in terms of both latency and throughput [50, 45, 1, 31]. Since internal parallelism is key to boosting SSD performance, efficient parallelization of data accesses is one of the crucial tasks of the firmware.

Address Mapping. Since NAND flash allows no in-place updates in a block, when a write arrives, flash firmware stores data in a temporal block (which is prepared in advance), and remaps the original address of the request (*virtual address*) and the actual location (*physical address*) of the corresponding data for future reads. In some cases, flash firmware can also remap the addresses in order to improve internal parallelism on writes [45, 20].

Garbage Collection. When the prepared blocks run out, flash firmware needs to reclaim physical block(s) so that it can serve an incoming update request. Since this block reclaiming task, called *garbage collection* (GC), is basically a series of extra internal operations, which include reading/writing live data from the target blocks to new block, erasing the old blocks and updating the mapping information, it can introduce long latencies and degrade performance. To reduce GC overheads as much as possible, modern SSDs employ more elaborate address mapping schemes, including some proposals that perform GC in the background.

Endurance. Flash blocks have limits in terms of the number of program (write) and erase cycles, referred to as *PE cycle*. Typically, a block that experiences higher PE cycles also experiences more errors and worse memory characteristics. Further, once a block reaches its PE cycle limit, it is not available anymore for storage. Since guaranteed PE cycles get smaller as technology shrinks, flash firmware needs to consider endurance related issues.

Wear Leveling. Since not all the information stored within the same location changes with the same frequency, it is important to keep the aging of each block as minimum and as uniform as possible. Flash firmware is also responsible for ensuring that all physical blocks are evenly used (to the maximum extent possible) and keeping the aging under a reasonable value. These tasks are collectively referred to as *wear leveling*.

Disturbance. Since a flash block is composed of multiple *NAND strings* to which the memory cells are connected in series (in groups of 64, 128, or 256), a memory operation on a specific flash cell may influence the charge contents on a different cell. This is referred to as *disturbance*, which can occur on any flash operation and lead to errors in undesignated memory cells.

5.2.3 Reliability Challenges on Reads

A read operation may fail because of 1) *read disturbance*, 2) *retention error* (leakage problem), and 3) *noise* (e.g., at the power rails). We now explain what SSDs do to address read failures.

ECC Recovery. To avoid failure on reads, *error-correcting codes* (ECC) are widely employed [14]. ECC can correct certain bit errors but typically introduces extra computation cycles on both reads and writes. More specifically, while the encoding takes a few cycles (on writes), the decoding requires lots of cycles. This cycle disparity between encoding and decoding imposes extra overheads on reads, which in turn degrades system performance. Since wider ECCs are required as flash technology shrinks, ECC overheads become more pronounced in modern SSDs.

Read Disturbance Management. Read disturbance can occur when reading the same target cell multiple times without any erase operation. When reading data from a specific cell, V_{read} (0 V) is applied to that cell, and all other cells are biased at V_{pass} (4$\sim$5 V), which makes them behave as pass-transistors. As a result, the cells on successive read operations can gain charge, which has similar impact on unintended writes. Since read disturbance can be corrected if the corresponding block is physically erased, it is necessary to erase the block associated with target page address causing the read disturbance. In general, to preserve data consistency, flash firmware reads all live data pages, erases the block, and writes down live pages to the erased block. Some flash firmware migrates live data to new block and remaps the address information between the old and new blocks [6]. This process, called *read block reclaiming*, introduces long latencies and degrades performance.

Runtime Bad Block Management. Even though ECC can correct certain bit errors and flash firmware keeps aging under control, endurance characteristics of flash storage get worse over time. In particular, raw bit error rate increases exponentially with PE cycles [17, 6], which leads to *uncorrectable ECC* (UECC) errors. To avoid UECC errors, flash firmware marks the blocks whose raw error rates have reached the error recovery coverage limit as “bad blocks”. It then replaces each bad block with a new block by remapping addresses, in an attempt to avoid future UECC errors. Similar to read block reclaiming and GCs, this *bad block management* also degrades system performance.

5.2.4 Storage Interfaces, TRIM and SMART

Conventional storage interfaces hinder the scalability of modern SSDs and make efficient SSD management difficult. To help with this, high-speed interfaces such as SATA 6.0Gbps and PCI Express are employed. Further, the most recent version of SATA provides SSD-specific command feature sets, which enable underlying SSDs to expose their internal characteristics to the OS. Specifically, *TRIM*, one of these command feature sets, allows the OS to invalidate data blocks that are no longer considered in use and delete the obsolete data at a system level. TRIM commands are expected to significantly reduce GC overheads and alleviate potential write degradation in many SSD applications. *SMART* is another command feature set, which enables self-monitoring, analysis, and state-reporting. Using it, OS designers can effectively manage SSDs by retrieving internal SSD information such as PE cycles and the number of channels and physical blocks.

5.3 Evaluation Setup

Solid State Disks. Today, there exist many different SSDs on the market, with quite different performance characteristics based on the vendor and system configurations, in terms of the DRAM buffer size, the number of cores, and the number of flash chips. For our experiments, we chose six representative products shipped by five different SSD-makers. All these SSDs are manufactured between 2011 and 2012, and their firmware are updated with the latest available version for our evaluations. Since our goal is not to perform reverse engineering or make performance comparison across these commercial products, we refer to each of them using a different postfix character, instead of giving its full name. Our SSDs and their important characteristics are listed in Table 5.1. Since the runtime information provided by different vendors varies a lot, in each of our evaluations, we select an appropriate subset of our SSDs and use them, and also mention the reason behind our selection. In general, SSD-L, -C, and -Z are evaluated for all basic tests, and SSD-A, -X, -P are used for more specific evaluations.

	Basic Test			**Specific Test**		
Postfix Name $\rightarrow$	-L	-C	-Z	-A	-X	-P
Storage (GB)	120	256	256	256	240	240
DRAM (MB)	128	256	0	256	0	0
Numbers of Chips	16	16	8	8	16	16
Technology (*n*m)	34	25	25-32	32	25	25
Numbers of Cores	2	3	1	3	1	1
Over-provision (%)	15	7.3	14.5	9.5	14.4	14.4

Table 5.1 Device characteristics of SSDs used in our study.

Measurement and Characterization Tools. In order to uncover hidden performance characteristics and examine widely held expectations on our SSDs, we need well-defined I/O access patterns, which can be controlled and reproduced irrespective of the underlying test platform. Consequently, we use Intel open source based storage tool, *Iometer* [35], as our default measurement and characterization tool. Iometer can generate various I/O workloads parameterized in terms of read/write ratio, sequential access/random access ratio, request sizes and the number of queue entries. However, Iometer reports performance results in terms of only average/min/max values at the end of the entire evaluation process. Therefore, for some of our evaluations that require a more microscopic view with finer resolution than what Iometer provides, we use a *modified*

Iometer, which captures the latency per individual I/O requests and performance characteristics on a second-basis without any underlying software intervention. Lastly, to evaluate the PHY level latencies, especially for the TRIM command overhead characterization, we use the commercial LeCroy SATA protocol analyzer (*Sierra M6-1*) [62] and double check the protocol status with this analyzer.

Protocol Controls. To accurately evaluate different technologies employed by modern SSDs, we also need a clean evaluation chamber under our control. For example, even though an application tool can mimic system idleness to evaluate background tasks by injecting artificial idle periods, the advanced host controller interface (AHCI) driver/controller can periodically send commands like SMART to examine the underlying system, which can make SSDs continuously busy. To the best of our knowledge, there exist no public tool, which can generate a specific ATA command, check its PHY level latency, and directly handle the AHCI. This is why, for some of our investigations that require the management of ATA commands (e.g., TRIM, SMART) and the control of the AHCI, we needed an in-house driver. Therefore, we also developed an *AHCI miniport driver*, as a part of WDM (windows driver model, which can generate TRIM commands by filling target addresses for the deleted contents with different access patterns (random/sequential), handle SMART commands to check the PE cycles of the SSDs, and manually control specific power modes to examine background tasks.

Experimental System. Our experimental system is equipped with an Intel Quad Core i7 Sandy Bridge 2600 3.4 GHz processor and 4GB DDR3-1333Mhz memory. Intel Z64 chipset is employed as the I/O controller hub in southbridge, and all SSDs we tested are connected to Z64 through the SATA 6.0Gbps interface. We execute all our scenarios in Microsoft NTFS, store logs and output results into separate block devices in a full asynchronous fashion; and neither a system partition nor a file system is created on our SSD test-beds. This configuration allows each SSD test-bed to be completely separated from the evaluation scenarios and tools.

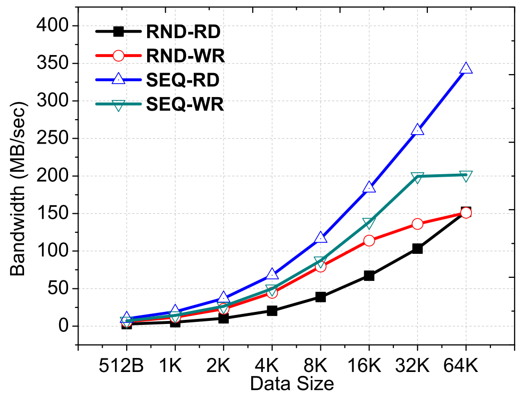
5.4 Testing Expectations on Reads

The most remarkable performance characteristic shifts on modern SSDs are observed in reads, since they are vulnerable to changes in internal SSD architecture. In this section, we first examine overall performance, comparing reads with other types of operations, and then analyze the challenges on reads in terms of performance sustainability and performance dependency on internal parallelism. Lastly, we uncover reliability problems on read-intensive workloads, which is one of the most critical issues, in our opinion, for both the OS and SSD designers.

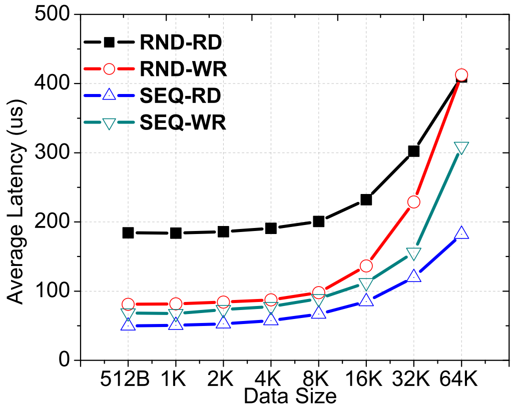
5.4.1 Are SSDs good for applications that exhibit mostly random reads?

To compare read performance with the performance of other types of operations, we executed different workloads composed of *sequential accesses* and *random accesses* with varying data transfer sizes (ranging from 1 sector to 128 sectors) on SSD-L, -C and -Z; we observed that SSD-A exhibits similar performance characteristics to SSD-C, and SSD-P and -X achieve similar performance results to SSD-Z in this test. Performance comparison across our three SSDs is plotted in Figure 5.2. Specifically, Figures 5.2(a), 5.2(c) and 5.2(e) show variance in overall bandwidth, and Figures 5.2(b), 5.2(d), and 5.2(f) plot variance in latency for SSD-L, -C and -Z, respectively.

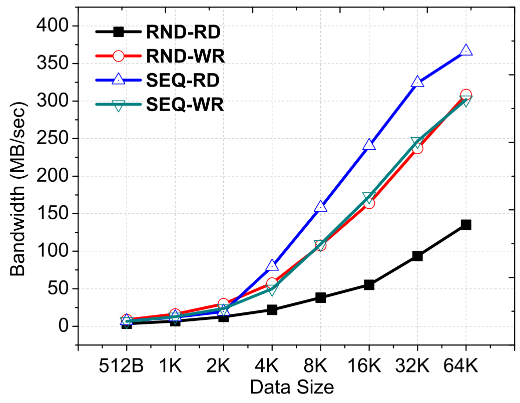
Our first observation is that the *performance values with random read accesses (denoted using RND-RD) are worse than other types of access patterns and operations, including even*



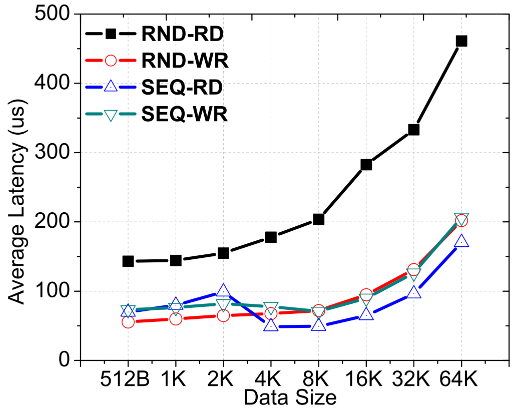
(a) SSD-L Bandwidth.



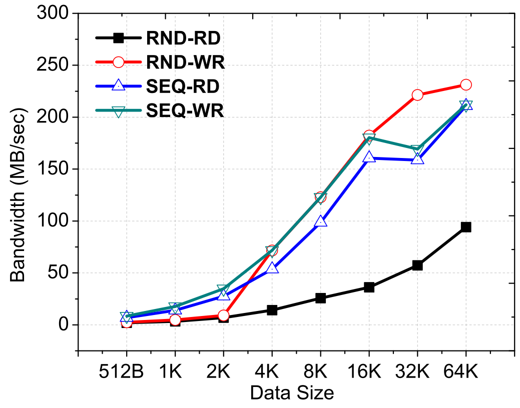
(b) SSD-L Latency.



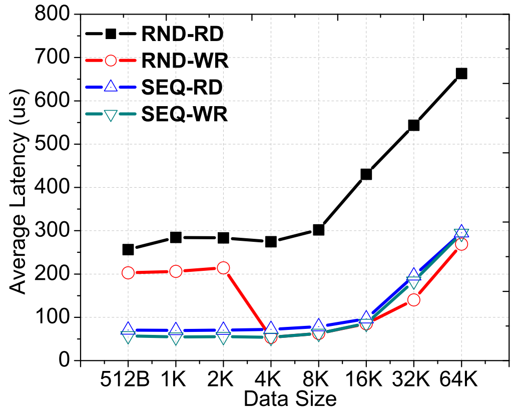
(c) SSD-C Bandwidth.



(d) SSD-C Latency.



(e) SSD-Z Bandwidth.



(f) SSD-Z Latency.

Fig. 5.2 Read/write performance comparison under varying data transfer sizes and access patterns. In these comparisons, *RND* and *SEQ* denote the random access pattern and sequential access pattern respectively, and *RD* and *WR* stand for read and write.

random write accesses, which is in *direct contrast* with the widely held expectation on read performance of SSDs in the literature. Specifically, the bandwidth values with random read accesses of SSD-L, -C, and -Z account for 59.7%, 39.4% and 23.7% of the corresponding values with random writes, respectively. Read latency characteristics are not much different from bandwidth; the latency values observed with sequential writes, random writes, and sequential reads only account for 41.3%, 35.2%, and 35.9% of the latencies observed with random reads, respectively (on average).

We believe that the main reason why SSDs can experience opposite performance characteristics at a flash level (reads are much faster than writes at a memory cell level) is the lack of internal parallelism on random reads. Note that, sequential accesses can be striped over multiple channels in a round-robin fashion, and the striped sub-requests can be interleaved across multiple flash dies in each channel. In contrast, random read accesses can potentially create a scenario where multiple requests end up contending for the same internal resources (e.g., channel, package, die, plane), referred to as *resource conflicts*. A request experiencing resource conflict has to wait for the completion of the other request(s) heading to the same resources. Therefore, the resource conflict on random reads causes low parallelism and thus degrades both bandwidth and latency. Unlike reads, flash firmware can easily forward the incoming write requests to a target sitting in idle by remapping addresses, which leads to low resource conflicts and high levels of parallelism.

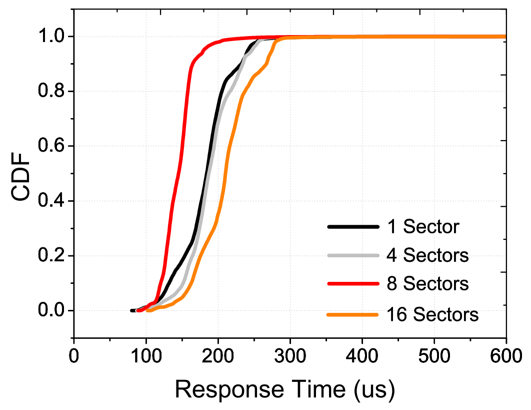
One potential concern on this read-write comparison would be the impact of the internal DRAM buffer. Since writes can be buffered in DRAM, if the internal DRAM does not flush the in-memory data to the flash medium, write performance would be much better than reads. However, as shown in Figures 5.2(e) and 5.2(f), we observed that DRAM-less SSDs, namely SSD-P,

-X, and -Z, exhibit very similar performance characteristics to 128MB and 256MB DRAM-equipped SSDs. Further, the amount of data written into those SSDs is over 200GB, which cannot be buffered by a small size DRAM; in fact, the DRAM capacity accounts for under 1% of the total amount of data we wrote in these experiments.

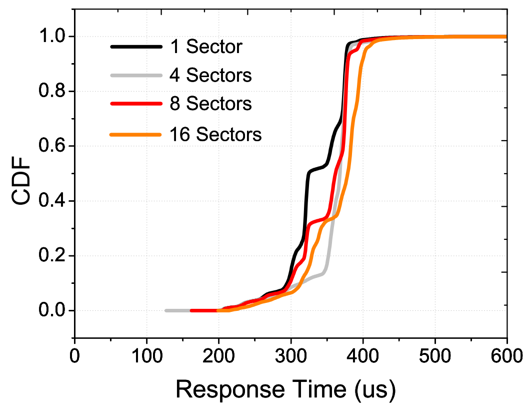
5.4.2 Can we achieve sustained read performance with sequential accesses?

As demonstrated in the previous section, all the SSDs tested generate their best performance on sequential read accesses. In this section, we further examine whether the sequential read performance can be sustained over time or not, which might have a significant impact on read-intensive SSD applications. For this set of experiments, we executed sequential read accesses with transfer sizes ranging from 1 to 16 sectors on two different SSD sets; "pristine" SSDs and "aged" SSDs, and measured their latencies per request with the modified Iometer. The results are presented in Figure 5.3. Specifically, Figures 5.3(a), 5.3(c) and 5.3(e) plot cumulative distributed function (CDF) of latency for pristine SSD-L, -C and -Z, and Figures 5.3(b), 5.3(d) and 5.3(f) plot the same for aged SSD-L, -C and -Z, respectively.

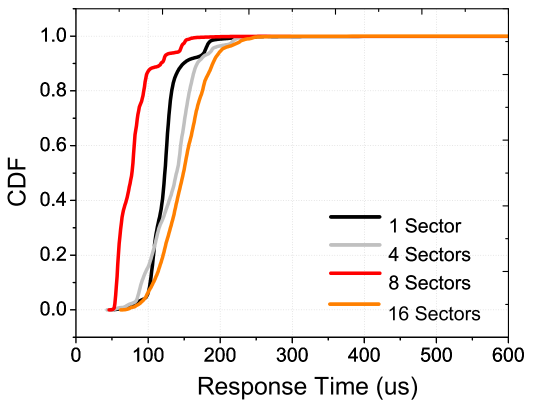
One can observe from these results that most of the read requests on all pristine SSDs are served within 300 $\sim$ 400 μsec. However, when SSDs get order, CDF curves shift from left to right exhibiting worse performance characteristics. The aged SSDs take over 600 μsec for serving all the I/O requests, which is two to three times worse than our pristine SSDs. One can conclude from this analysis that, *sequential read performance characteristics get worse with aging and as I/O requests are being processed*, which are unfortunately captured neither by NAND flash data sheets or nor by SSD specifications. We believe that this performance degradation on reads is mainly caused by fragmented physical data layout and reliability management overheads



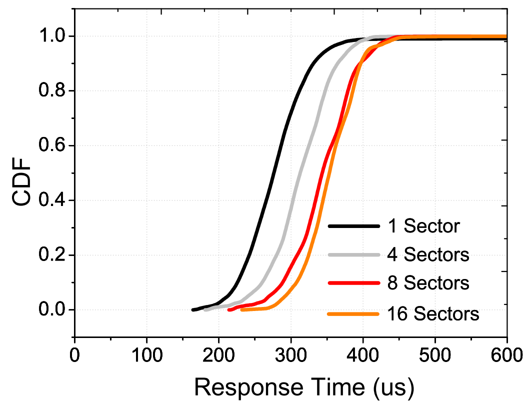
(a) Reads on pristine SSD-L.



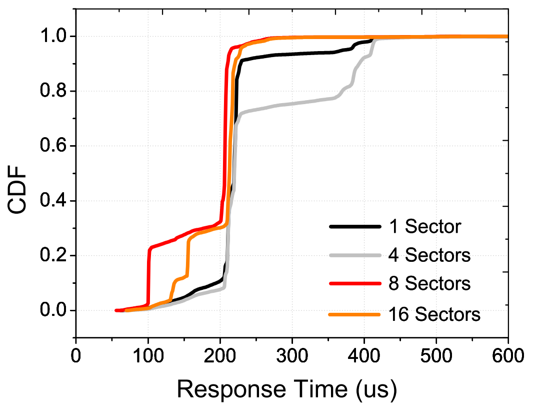
(b) Reads on aged SSD-L.



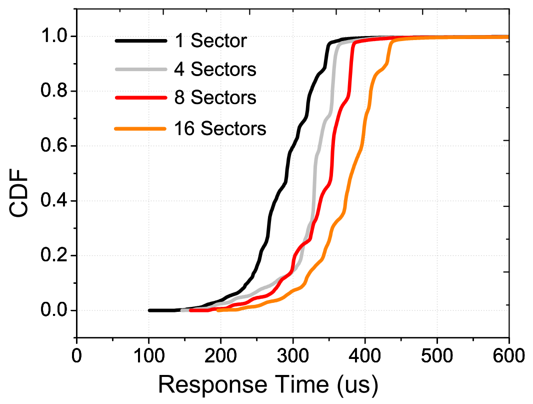
(c) Reads on pristine SSD-C.



(d) Reads on aged SSD-C.



(e) Reads on pristine SSD-Z.



(f) Reads on aged SSD-Z.

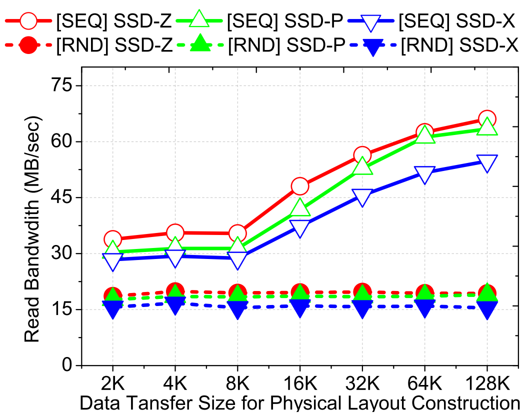
Fig. 5.3 Cumulative distribution function (CDF) of latency variance in sequential reads for "pristine" SSDs and "aged" SSDs (i.e., writing data with a random access pattern to the entire storage space of SSDs). Note that all the curves presented in the CDFs are shifted from left to right as SSDs get older.

on reads. This read performance degradation also implies that the read behavior of an SSD cannot be easily characterized by the OS by examining only the current I/O request patterns despite recent works [13, 12] claiming that. We will provide a deeper evaluation and more evidence on this read performance characteristic in the following sections.

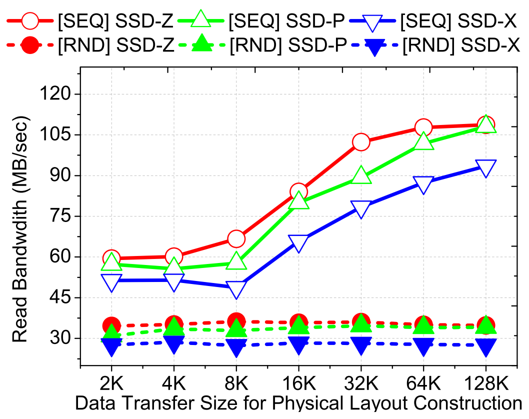
5.4.3 What is the relationship between read performance and previous writes accesses?

In this section, we analyze read performance of SSDs by building 14 different physical layouts, in an attempt to reveal the relationship between read performance and previous writes accesses as well as internal SSD parallelism. The insight behind this evaluation is that the order of random writes can be transformed into a kind of sequential pattern by the underlying flash firmware, which may have performance impact on current writes as well as future reads. Specifically, the address remapping process allows the flash firmware to easily strip the write requests over multiple internal resources irrespective of the access pattern, which leads to high levels of parallelism as well as improved performance on random writes. However, this physical data layout construction on writes may also introduce different performance behavior for future reads. Unlike writes, for a read to occur, the data to be read should exist and it must reside in a particular location. As a result, the degree of parallelism and performance on reads depends highly on the underlying data layout, *which is constructed during the previous writes*.

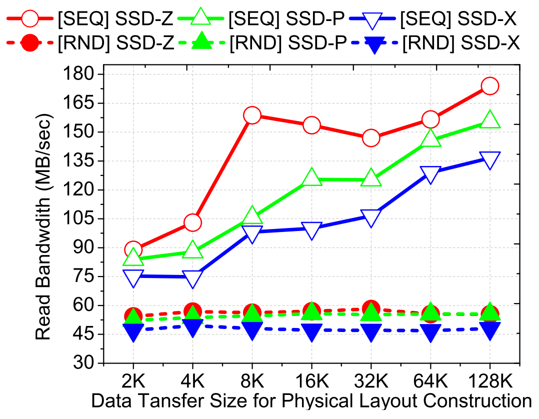
To quantify this impact, we chose SSDs that have no DRAM, to minimize any potential side effects of buffering on both read and writes. We then randomly wrote data into the entire address space of the DRAM-less SSDs, SSD-P, -X and Z, with seven different data transfer sizes ranging from 4 sectors to 256 sectors. As a control group, we also wrote data into the same type of SSDs but different devices, using sequential access patterns composed of the same data



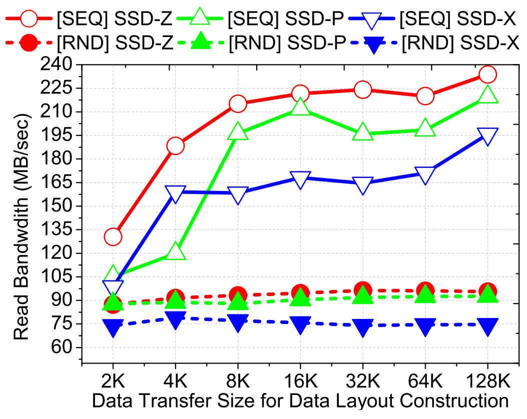
(a) 4KB Sequential Reads.



(b) 8KB Sequential Reads.



(c) 16KB Sequential Reads.



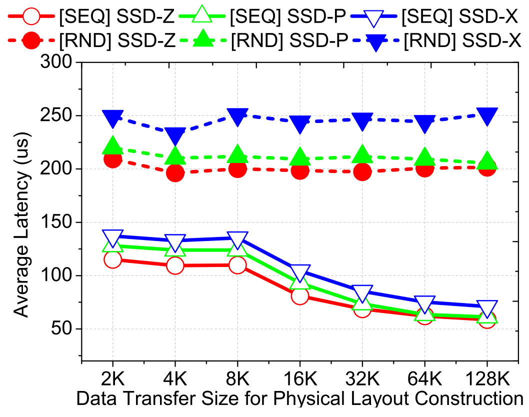
(d) 32KB Sequential Reads.

Fig. 5.4 Bandwidths with different physical data layouts. *SEQ* and *RND* denote sequential writes and random writes, used for the physical data (*PDT*) layout construction. Observe that throughput significantly varies based on the physical data layout, constructed by previous writes, even under same read request patterns.

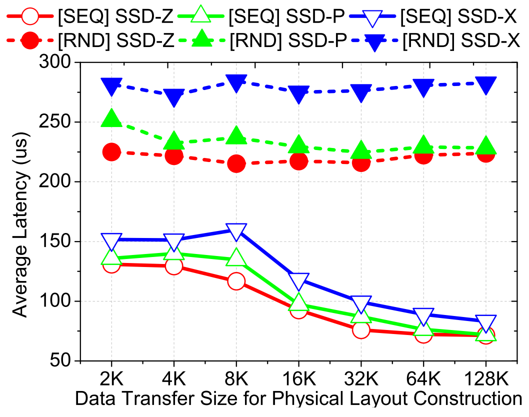
transfer sizes used in the random writes. We then read the entire space of those SSDs with varying data sizes ranging from 4KB to 32KB, and measured the bandwidth and latency. The results are plotted in Figures 5.4 and 5.5.

To make our discussion easier to follow, let *RND-PDT* denote the physical data layout resulting from random writes, and *SEQ-PDT* denote the physical data layout resulting from sequential writes. In Figures 5.4 and 5.5, the dashed-lines and solid-lines indicate the read performance on RND-PDT and SEQ-PDT, respectively. One can observe that, *read performance significantly varies based on the physical data layout organization even though current I/O request access patterns are exactly the same*. More specifically, bandwidth values for all the evaluations on RND-PDT (Figure 5.4) are under 80 MB/s, whereas bandwidth values on SEQ-PDT reach up to 220 MB/s. As the data transfer size used during the physical data layout construction increases, the performance gains are more pronounced since this allows the flash firmware to more easily build a physical data layout by sequentially writing data back-to-back. This performance impact is also observed in our latency characterization plotted in Figure 5.5. While the minimum latency with RND-PDT is 210 μsec, the latency with SEQ-PDT is around 80 μsec.

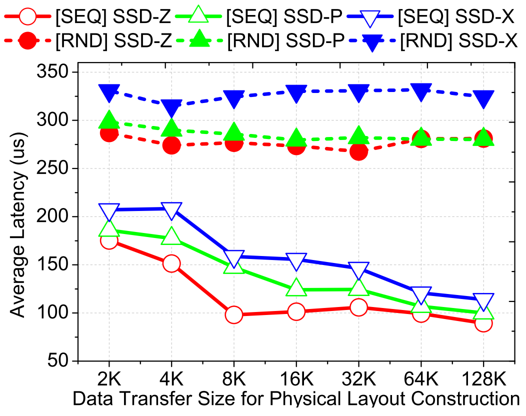
Based on these evaluations, one can conclude that, since the virtual address space that the flash firmware provides on RND-PDT has been constructed during previous writes, the order of sequential read accesses on the virtual address space are jumbled. Consequently, the read accesses can suffer from multiple resource conflicts at a specific channel, chip and die, even though the access pattern itself is sequential. This in turn can degrade read performance due to low internal SSD parallelism. In contrast, sequential accesses on SEQ-PDT can be simultaneously served from multiple internal resources in different locations without any major resource conflict since there are no changes in the order of virtual addresses.



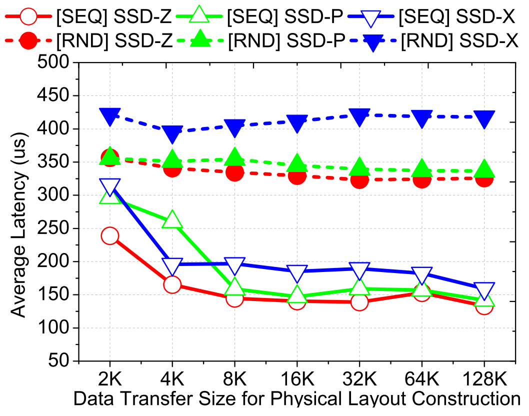
(a) 4KB Sequential Reads.



(b) 8KB Sequential Reads.



(c) 16KB Sequential Reads.



(d) 32KB Sequential Reads.

Fig. 5.5 Latencies with different physical data layouts. These latency comparison explains how the physical data layout is related to internal parallelism in two aspects. First, the read latency performed on RND-PDT is 2.3 times higher than that of SEQ-PDT that induces lower resource conflicts. Second, as the data movement size of reads increases, the magnitude of the latency improvement with SEQ-PDT is shorter than the improvement with RND-PDT that has many resource conflicts potential.

5.4.4 Do program/erase (PE) cycles of SSDs increase during read only operations?

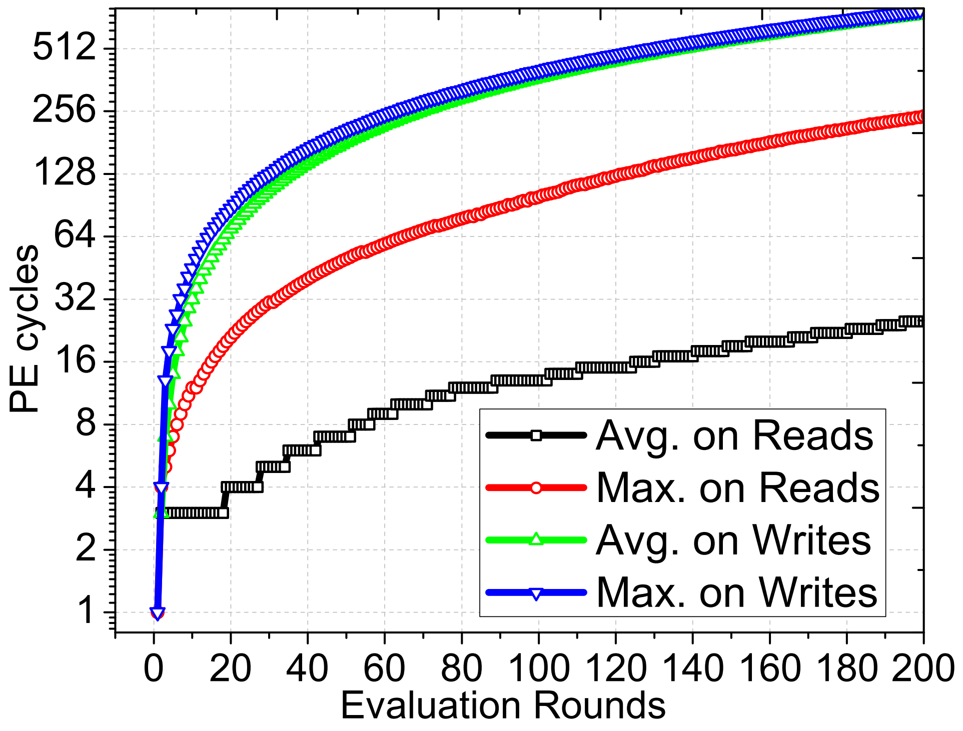
To study the PE cycle characteristics on reads, we executed Iometer with two different *read-only* workloads, composed of sequential and random access patterns, about 200 rounds, each with a running time of 1 hour (total 200 hours). In each round, we sent a SMART command using our in-house AHCI minport driver and measured PE cycles by decoding return codes based on the SMART attribute table [97]. To compare the PE cycles between reads and writes, we also measured the PE cycles on write-only workloads with the same access patterns and measurement method used in the read-only workload evaluations. We observed that unfortunately all the SSDs tested provide insufficient information to understand SSD internal characteristics; in particular, all the data are normalized or provided as percentage based on their lifespan expectations, and some SSDs do not even report their PE cycles on reads. Therefore, we present the reliability evaluation results of a specific version of SSD-A, which is used in Apple MacBook Air; unlike other SSDs we tested, SSD-A provides absolute maximum/average PE cycles on both reads and writes.

Figures 5.6(a) and 5.6(b) give the variance in PE cycles on two different SSD-A instances under sequential and random access patterns, respectively. One can see from these plots that *PE cycles increase in every evaluation round, in a direct contrast to what the current literature on systems exploiting SSDs would lead one to believe*. In sequential reads, the maximum PE cycles reach the half of PE cycles on writes, as shown in Figure 5.6(a). Ironically, the maximum PE cycles with the random read-only workload are higher than that of writes by about 12x (Figure 5.6(b)). We believe that the reason behind this PE cycle increase on read-only workloads is the read disturbance and runtime bad block management. Since these activities require erasing block(s) and live-data migration to the target block(s), *read requests can shorten the SSD lifespan and significantly degrade overall performance*. Further, the disparity between the maximum and average PE cycles tell us another story; wear leveling strategies employed by current flash firmware mainly focus on writes, *not* on reads. While SSD-A firmware keeps reducing the gap between the maximum and average PE cycles on writes, the maximum PE cycles on reads is 247 times higher than the average PE cycles in each round, which makes certain blocks wear out faster and worsen SSD endurance characteristics.

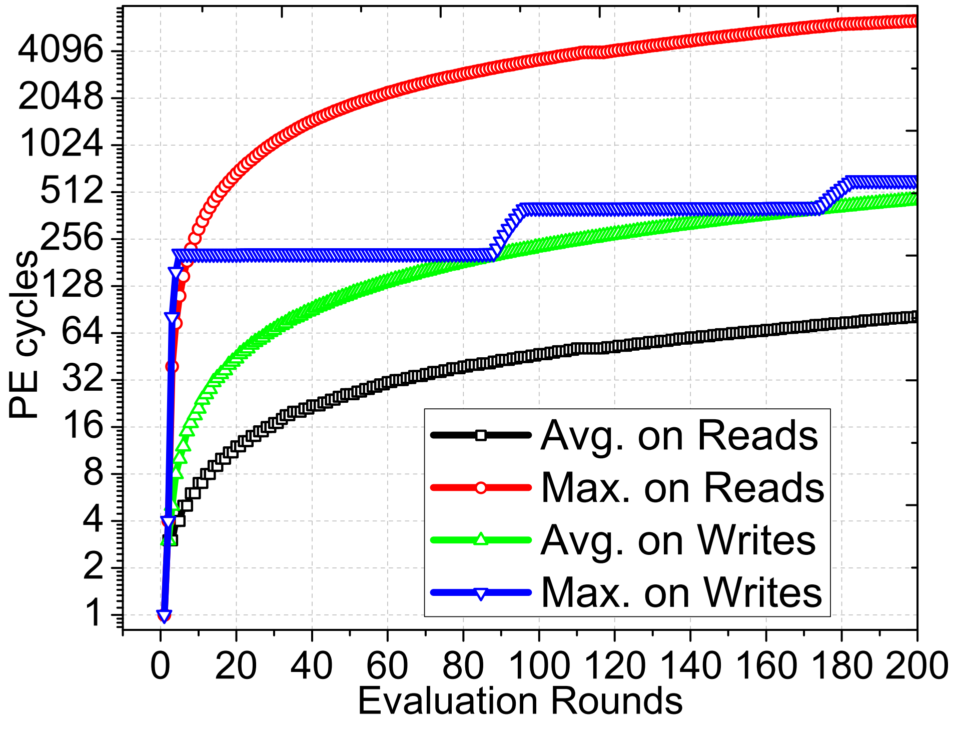
5.4.5 Is there any performance impact of the reliability management on reads?

All the activities for handling read disturbance management, runtime bad block management, and ECC, referred collectively to as *reliability management on reads* (RMR), require additional I/O operations and compute cycles. These overheads are not revealed to users, but can contribute to long latencies on normal operations. In this section, we examine the latency variation between reads *with RMR* and reads *without RMR*, which is veiled by most SSD manufactures. For our evaluation, we executed the random read-only workload, used in Section 5.4.4, on three devices, SSD-L, -C, and -Z, and the results are given in Figure 5.7.

As indicated by these plots, *the read latency with RMR is at least 5 times higher than the latency of reads without RMR*, which would be unacceptable for latency-sensitive SSD applications. Further, RMR overheads are more pronounced with small size random access patterns (1 sector $\sim$ 64 sectors), which is the dominant request size in many file systems. Considering 8 sector (4KB) requests as an example, while the read latencies without RMR on SSD-L, -C, and -Z are 75, 60, and 47 μsec, respectively, the increased read latencies with RMR are 685, 1787, and 4944 μsec, in the same order. Even though the latency disparity between ordinary reads and



(a) Sequential Patterns.



(b) Random Patterns.

Fig. 5.6 PE-cycle comparison between reads and writes. Note that PE cycles increase under read-only workloads. Further, with random accesses, the maximum PE cycles on reads are 12 times greater than that on writes.

reads with RMR tends to decrease as the transfer size increases, high RMR-induced latencies with large data sizes (1MB $\sim$ 32MB) still seem to be problematic.

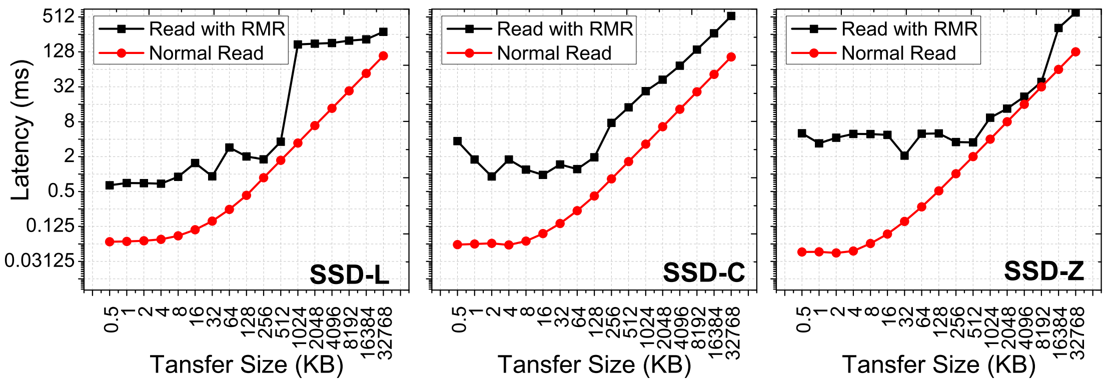
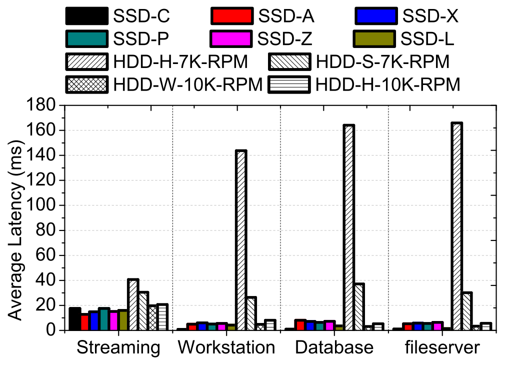


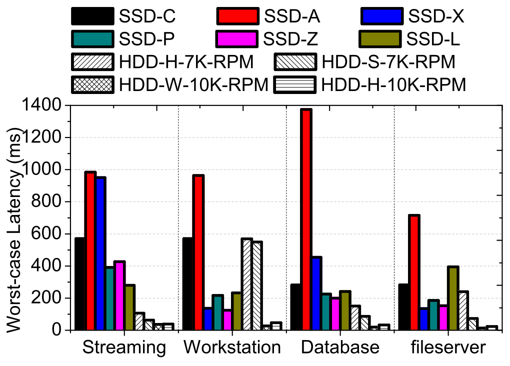
Fig. 5.7 Latency comparison between reads with reliability management (RMR) and ordinary reads (i.e., reads without RMR). When RMR is employed, the latency is at least 5 times higher than the latency of reads without RMR.

5.5 Testing Expectations on Writes

Modern SSD firmware and architecture are well optimized to improve write performance, but the worst-case performance on writes is still a problematic challenge. In this section, we examine the worst-case write latencies, analyze their correlation with system throughput, and investigate the impact of the internal DRAM buffer on latency. To evaluate the worst-case write latency, we prepared a set of *fully-utilized devices* by writing data (with sequential access pattern) that covers the entire storage space of SSDs. This makes SSDs reclaim block(s) for new incoming requests so that we can easily capture the worst-case latency and system throughput imposed by GCs.



(a) Average Latency.



(b) Worst-case Latency.

Fig. 5.8 SSD and HDD latency comparison. While SSDs overall outperform HDDs, the worst-case latencies of SSDs are much higher than the worst-case latencies of HDDs.

5.5.1 How much impact does the worst-case latency have on modern SSDs?

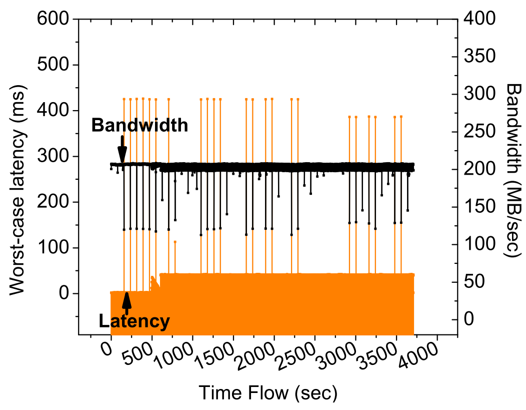
We compare all the SSDs tested and two types of enterprise-scale HDDs (7K RPM and 10K RPM) in terms of both the average latency and the worst-case latency. To quantify the average latency, we use pristine devices, and for the worst-case latency evaluation, we employ the fully-utilized devices for SSD and HDD. We run Iometer with its enterprise open-workloads including streaming, workstation, database and filesever applications, with the fraction of writes being 99%, 20%, 33% and 20% (of total I/Os), respectively.

Figures 5.8(a) and 5.8(b) plot the average latency and worst-case latency of our SSDs and HDDs. We see that, the average latencies of all the SSDs are better than HDDs, irrespective of the workload type used. Especially, compared to the 7K RPM HDDs, SSDs provide 2 $\sim$ 173 times shorter latency. However, *the worst-case latencies on fully-utilized SSDs are much worse than that of HDDs*, which is problematic for many write-intensive SSD applications. Specifically, the worst-case latencies of all the SSDs tested are 12 and 17 times worse than that of 10K RPM HDD-H and HDD-W, respectively, on an average. This is mainly because NAND flash in SSDs allows no in-place update with overwrites, which leads to GC invocations that contribute to overall latency.

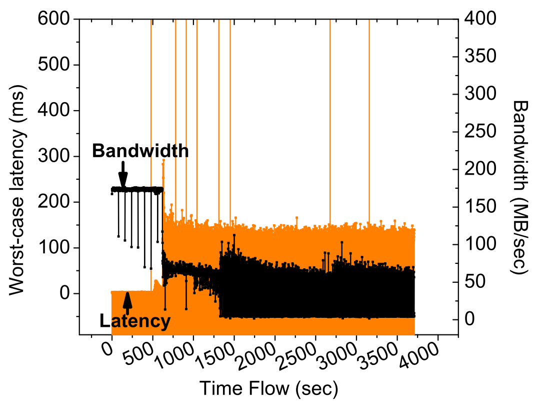
5.5.2 What is the correlation between the worst-case latency and system throughput?

To study the correlation between GC and the worst-case latencies, we prepared two sets of fully-utilized devices, each consisting of SSD-L and -C. We then executed our modified Iometer with write-intensive workloads composed of 100% random accesses and sequential accesses for an hour on these SSDs, and measured the latency and throughput values.

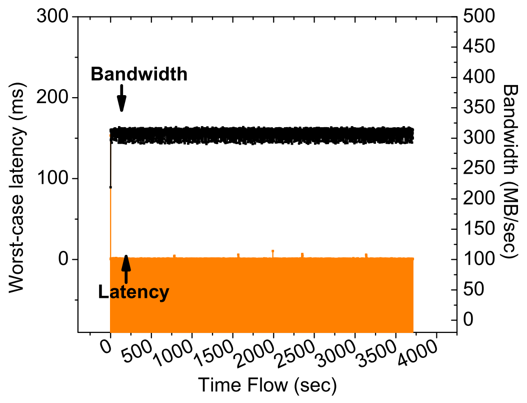
Figures 5.9(a) (SSD-L) and 5.9(c) (SSD-C) plot the time series for both latency and system throughput along with GCs under the sequential access pattern. For both SSD-L and SSD-C, GCs are infrequently invoked, occasionally imposing long latencies but not impacting system throughput much. The execution in this case recovers performance immediately after the GC. In contrast, with the random write workloads, the worst-case latencies imposed by GCs significantly increase, which in turn dramatically drop the system throughput as shown in Figures 5.9(b) and 5.9(d). This performance characteristic caused by GCs under random write workloads is referred to as *write cliff*. Specifically, *once the write cliff begins, SSD latencies (bandwidth) become 11x (3x) worse than the normal case*. Further, *more problematic challenge of modern SSDs is that the performance degradation on the write cliff is not recovered even after many GCs are executed*. We believe that this is because the range of random access addresses is not covered by the reclaimed block(s). Consequently, block reclaims performed by GC are required for each access, which in turn leads to successive GC invocations. Even though we focused on two SSDs in this section (due to space concerns), we observed write cliffs in all the SSDs tested.



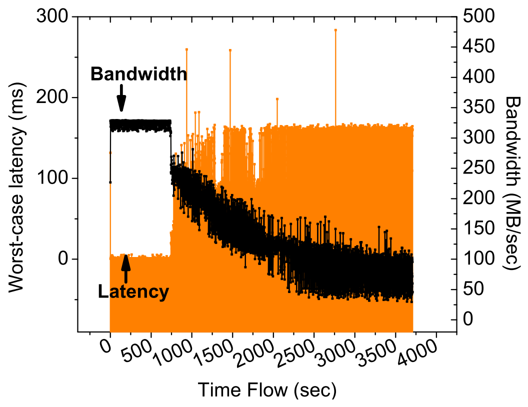
(a) SSD-L non write cliff.



(b) SSD-L write cliff.



(c) SSD-C non write cliff.



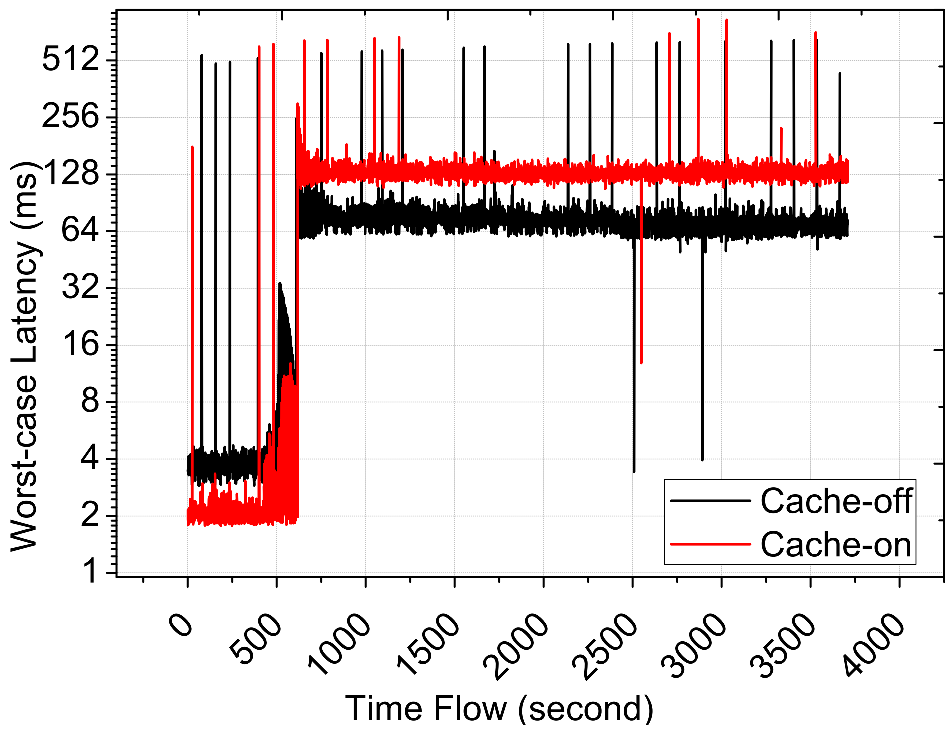
(d) SSD-C write cliff.

Fig. 5.9 Impact of write cliff. Initially, SSDs provide reasonable performance even though GCs are invoked. However, once write cliff begins, the performance significantly degrades and is not recovered later.

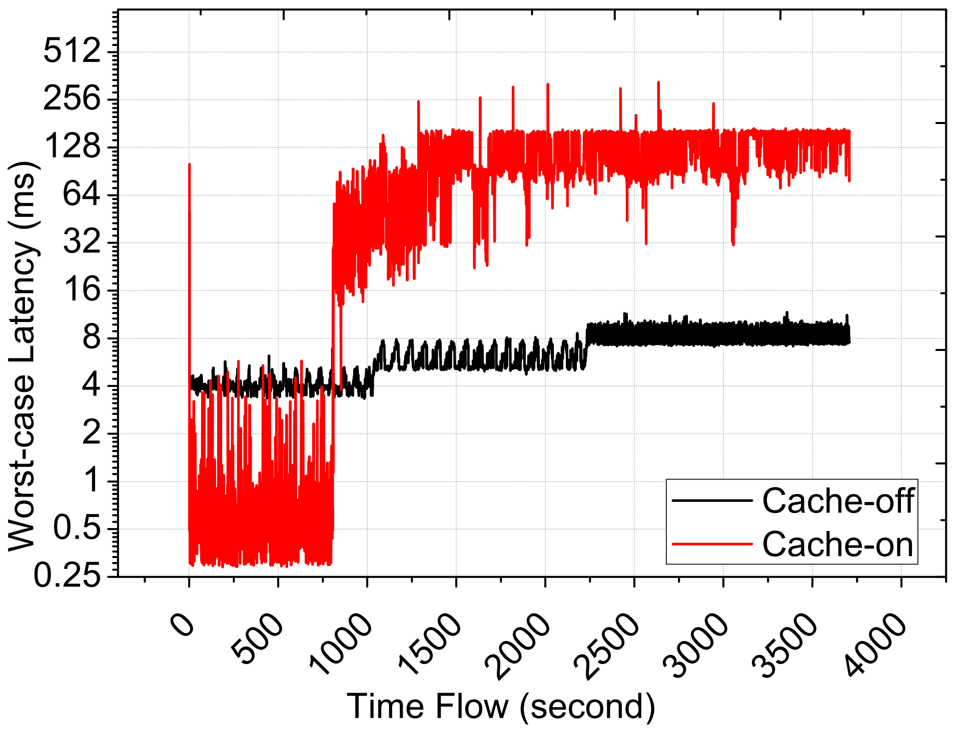
5.5.3 Could DRAM buffer help the firmware to reduce garbage collection overheads?

Since writes can be buffered using internal the DRAM, modern SSDs are somewhat expected to hide GC overheads. In this section, to examine the DRAM impact on GCs, we setup two fully-utilized device sets and write data with a sequential access pattern. In these experiments, one of these two sets are evaluated under the disabled (DRAM) cache (cache-off), and the other set is evaluated under the cache (cache-on). To make device status cache-off, we submit cache-disabled command, which brings the force access unit (FAU) tag of SATA 3.0 [97] for every I/O requests.

Figures 5.10(a) and 5.10(b) illustrate the time series comparison between the cache-on and cache-off status devices, SSD-L and -C, respectively. The worst-case latencies of SSD-L are hidden by the DRAM buffer before the write cliff begins. However, once GCs start to be invoked in a series, the latency of the cache-on SSD-L becomes two times worse than that of the cache-off SSD-L. In the case of SSD-C, the performance disparity between the cache-on and cache-off status are more pronounced. *While the DRAM buffer provides four times shorter latency compared to cache-off SSD-C before the write cliff begins, it introduces four times worse latencies when the write cliff kicks in.* Even though a system can react before the data is written into the actual flash device, the DRAM buffer needs to flush the in-memory data to the flash medium periodically. Since target addresses of the buffered data are fully random, this flushing of data introduces a large number of random accesses, which can in turn accelerate GC invocations of SSDs and introduce write cliffs.

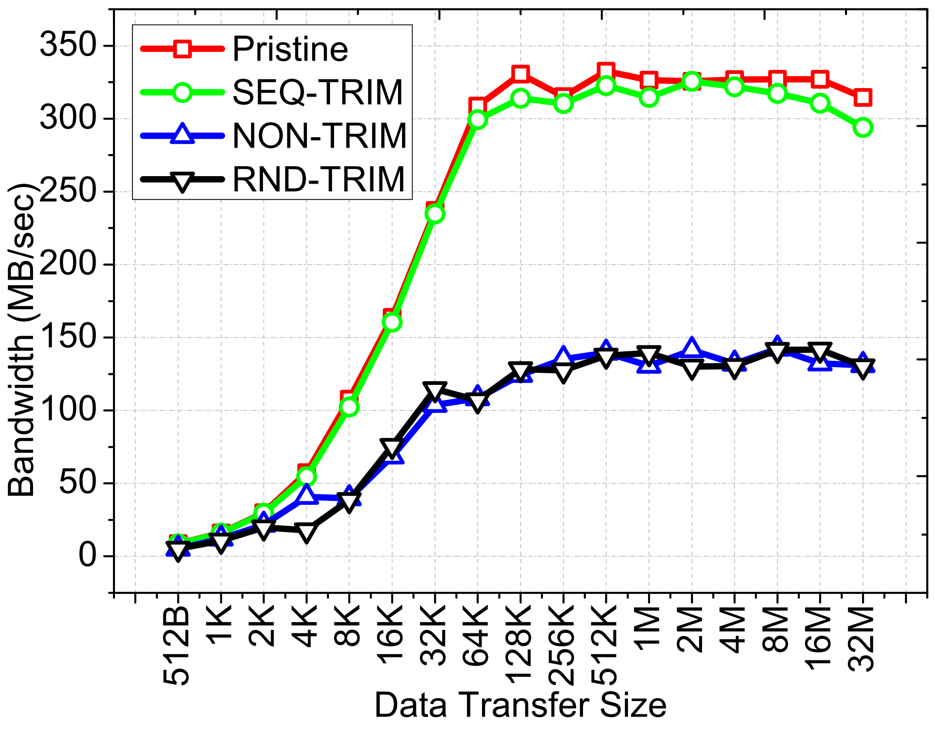


(a) SSD-L.

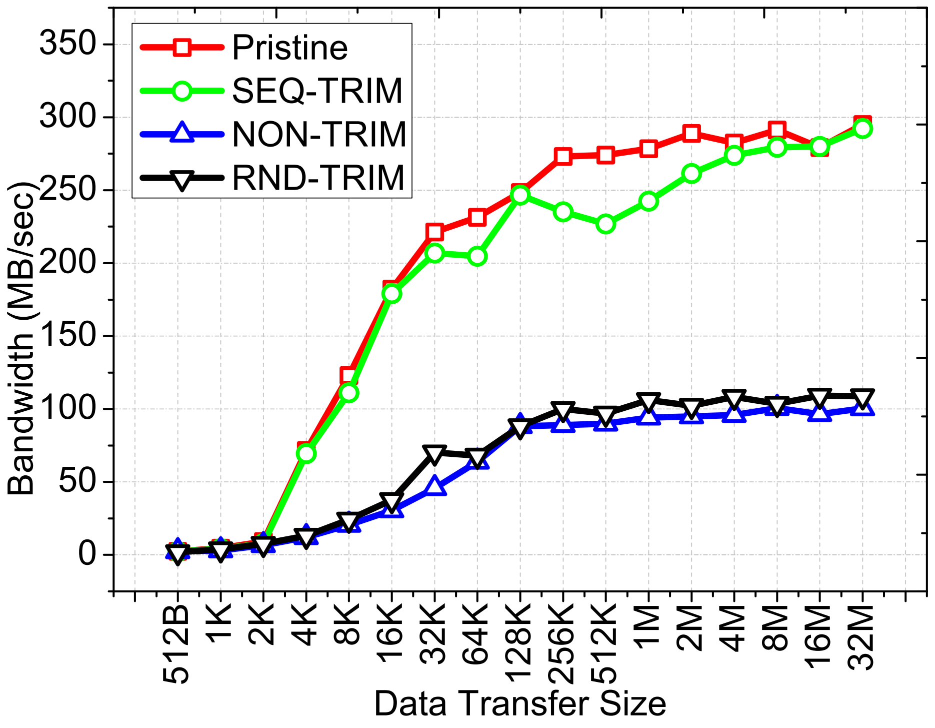


(b) SSD-C.

Fig. 5.10 Worst-case latency correlation between the DRAM buffer cache and GC. The DRAM buffer provides excellent latency, but after the write cliff, it makes latencies even worse.



(a) SSD-C.



(b) SSD-Z.

Fig. 5.11 Bandwidth impact of TRIM. While SEQ-TRIM (the order of target addresses is ascending) can effectively remove GCs, RND-TRIM (the order of targets is random) has no impact on GC overheads.

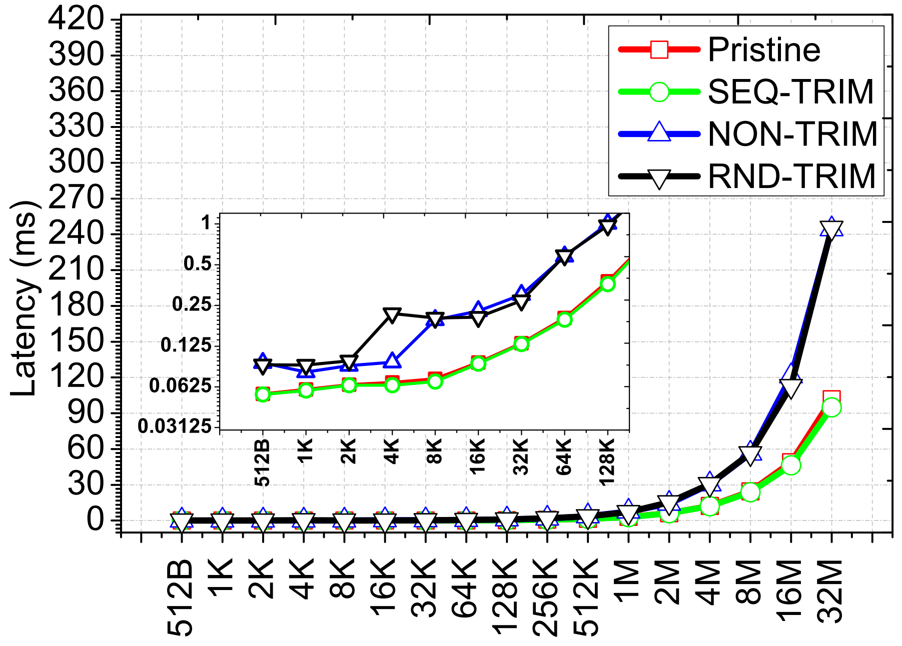
5.6 Testing Expectations on Advanced Schemes

In this section, we evaluate two advanced SSD schemes, TRIM OS support and background tasks, which recently received a lot of attention.

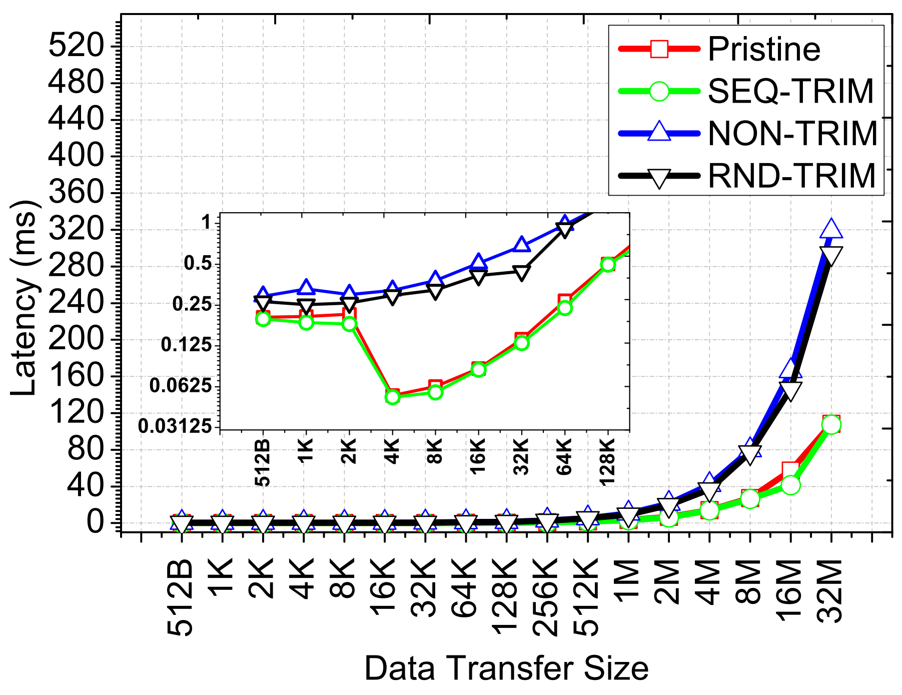
5.6.1 Can TRIM command reduce GC overheads?

To quantify the performance impact of TRIM commands, we first wrote data over the entire storage space of SSD-C and -Z using sequential and random access patterns, respectively. Then, at a system level, we deleted all the data written using TRIM commands, which consists of two command-composition steps. The first step is to setup the TRIM field of the DATA-SET-MANAGEMENT command and send it to the SSD to let it know that the host wants to delete data, which is specified target addresses by following the TRIM request data (TRD) frames. Next, we need to configure the logical block address (LBA) range entries in the TRD frames and submit them to the SSD. Using multiple TRIM commands and TRD frames that cover the entire SSD address space, we wiped out all the written data in the previous step. If the order of the target LBAs in the TRD frames is ascending, we refer to the corresponding TRIM command pattern as *SEQ-TRIM*. On the other hand, if the order of the target addresses in the TRD frames form a random access pattern, which has been used for writing data in the previous step, we denote this TRIM pattern as *RND-TRIM*.

We measured performance by writing data of different sizes trimmed using SEQ-TRIM and RND-TRIM. In addition, we also evaluated the performance of our pristine-state SSDs, denoted by *Pristine*, and SSDs that have no TRIM command management, called *NON-TRIM*. The



(a) SSD-C.



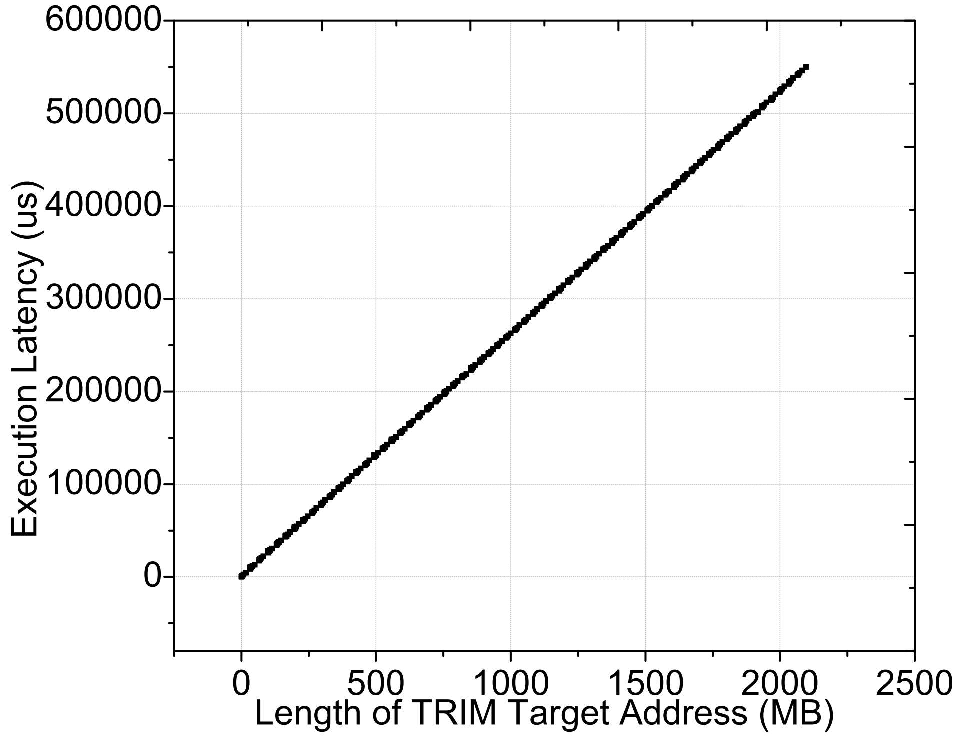
(b) SSD-Z.

Fig. 5.12 Latency impact with TRIM. Similar to bandwidth impact, there is no latency gain with RND-TRIMs.

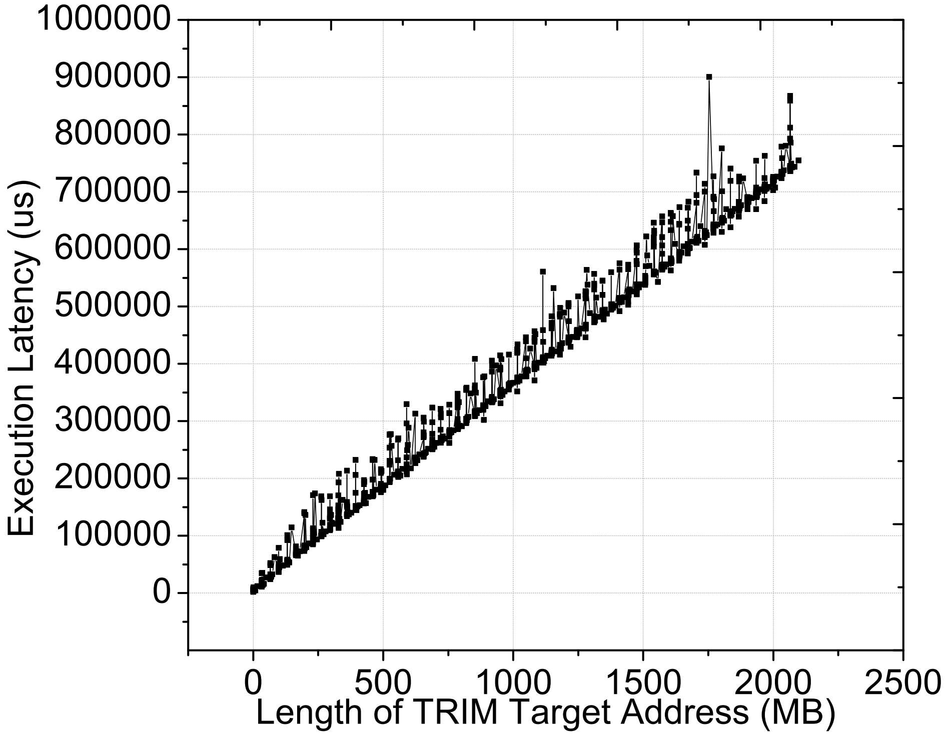
main insight from this evaluation is that, if SSDs tested handle the TRIM commands appropriately, all the written data should be successfully deleted, irrespective of which TRIM pattern is employed. As a result, the trimmed SSDs are expected to exhibit the same performance as our pristine state SSDs. As shown in Figure 5.11, SEQ-TRIM works very well for deleting data in both SSD-C and -Z. As expected, the bandwidth of the SSDs trimmed by SEQ-TRIM is similar to that of Pristine. In contrast, RND-TRIM shows no success in alleviating the GC overheads in both SSD-C and -Z. Our latency evaluation results also show similar performance characteristics. To better understand the execution-time impact of TRIM, we also studied performance at a finer-level, focusing on small data transfer sizes, ranging from 1 sectors to 256 sectors, in Figures 5.12(a) and 5.12(b). As can be observed, the latencies of the SSDs trimmed by RND-TRIM are longer than those of the SSDs trimmed by SEQ-TRIM by about 3x on average. One can conclude from this analysis that, *SSDs do not trim all the data, and their behavior is strongly related to the TRIM command submission patterns*. In our evaluation, only SEQ-TRIM could successfully delete data, providing a similar latency to pristine SSDs.

5.6.2 Does a TRIM command incur any overheads?

One concern that OS designers might have is the potential overheads that can be experienced by an SSD in processing the TRIM command itself. In this section, we measure the latency incurred when processing a TRIM command, using our in-house AHCI miniport driver and the LeCroy protocol analyzer, Sierra M6-1. To do this, we wrote data varying from 512B to 2GB, which has the same address-range coverage as a TRD frame, and sent a TRIM command by filling the corresponding LBAs into the TRD frame. We then captured the time duration from

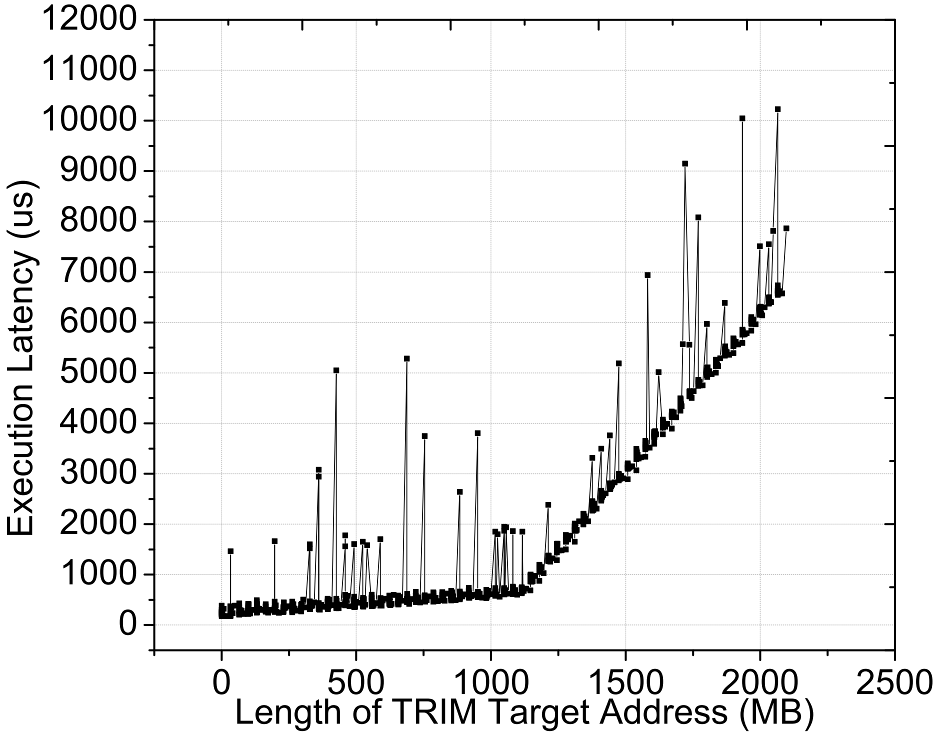


(a) SSD-L.

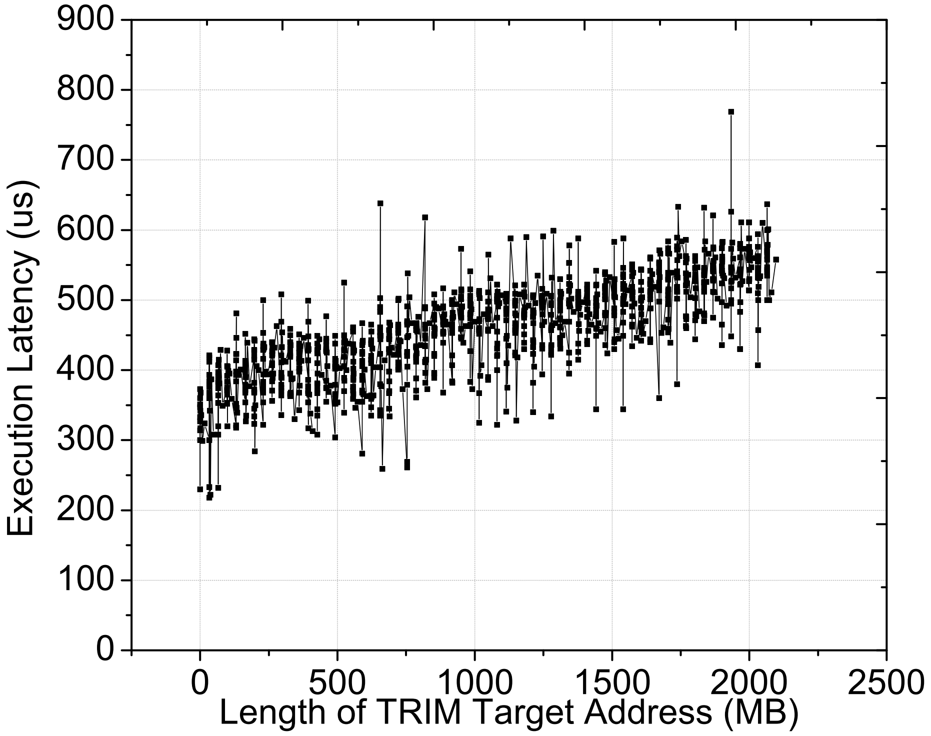


(b) SSD-Z.

Fig. 5.13 E-TRIM overheads. Since E-TRIM performs block erasure on demand and do not return control to the storage system, the host can be disabled until the TRIM process finishes.



(a) SSD-A.



(b) SSD-C.

Fig. 5.14 I-TRIM overheads. I-TRIM is more efficient in controlling the TRIM commands, but the latency overheads are still about 6 $\sim$ 153 times worse than a 4KB write-latency.

the DATA-SET-MANAGEMENT command submission to the end of the response of the following TRD frames as *TRIM-latency*. Since Sierra M6-1 [62] provides a detailed protocol-level timing model for the command issue and completion, we are able to capture TRIM-latency for each TRIM process. As shown in the previous section, since SSDs do not get any benefit from RND-TRIM, in this test, we focus on measuring TRIM-latency on SEQ-TRIM.

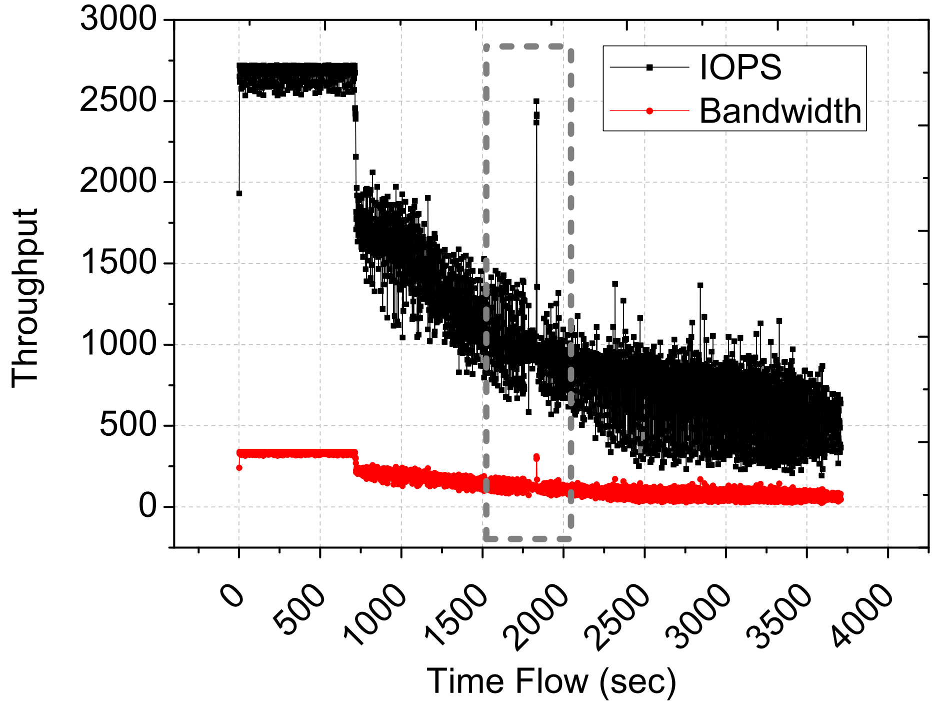
Based on our observations and experience, we can classify existing TRIM command management strategies into two types: 1) *block erasure in real-time*, called E-TRIM, and 2) *data invalidation based on address and prompt response*, referred to as I-TRIM. As shown in Figure 5.13, E-TRIM requires long processing times to erase physical blocks based on the target addresses specified by TRIM. For example, to process a single TRIM command covering a storage space of 2GB, SSD-L and SSD-Z take 754 *m*sec and 550 *m*sec, respectively. Note that the TRIM-latency increases linearly with the amount data that LBAs in the TRD frames aim to delete due to physical block erasures.

In contrast, SSDs with I-TRIM just mark flags into a mapping table indicating that the contents are not valid anymore and return response immediately. This TRIM strategy can alleviate the time consuming activities of GCs such as the live-data lookup and relocation, even though it does not erase actual blocks. Although I-TRIM has some downsides (e.g., extra DRAM requirement, the possibility of losing in-memory TRIM data in the case of power failure), it is expected to improve SSD reliability to some extent. Specifically, 2x *n*m technology flash chips are much less reliable than larger feature size (3x$\sim$5x *n*m) flash chips, mainly because their memory array suffers considerably more from disturbance and has lower endurance characteristics. Further, the industry observed that such disturbances occur even when erasing a block and the endurance gets worse as PE cycles increase [17, 6]. Consequently, lower technology flash-equipped SSDs employ I-TRIM rather than E-TRIM in an attempt to reduce the number of block erasures as much as possible and improve reliability. From a performance perspective, the latency of I-TRIM is much shorter than the latency of E-TRIM since the former requires only a few cycles to update the underlying mapping table, as shown in Figure 5.14. However, I-TRIM still takes 400 μs $\sim$10000 μs to handle individual TRIM commands. This I-TRIM latency is longer than an 8 sector write latency by 6 times $\sim$ 153 times. We believe that this is because the flash firmware cannot maintain all the mapping information in the internal DRAM, which leads to extra I/Os on the flash medium to load/store the mapping table on demand. As a result, it takes some cycles to manage the mapping table and update the TRIM information in the appropriate entries of the table. One can conclude from this analysis that *modern SSDs require much longer latencies to trim data than normal I/Os would take*, which may put extra pressure on host systems.

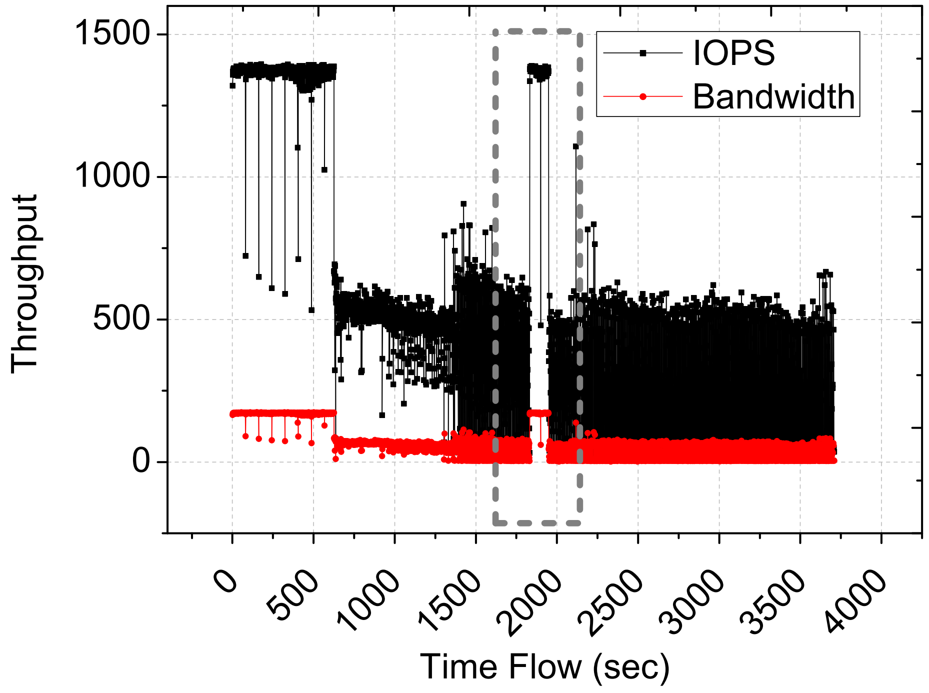
Note that, unlike the other devices tested, SSD-L employs more reliable flash chips with 35 *n*m technology and the highest degree of over-provisioning. These two factors make the write amplification factor of SSD-L lower and thus make the device reliable [30] even though BGC introduces more block erasures. Also, SSD-L needs more power to perform BGC more than SSD-C and SSD-Z, by about 270% and 78%, respectively, in idle periods.

5.7 Rethinking SSD Systems

Based on our experimental results and observations, we now provide a summary of our answers to the questions we raised at the beginning of this chapter.



(a) SSD-C BFLUSH.



(b) SSD-L BGC.

Fig. 5.15 Performance sustainability of the background tasks. Even though BFLUSH and BGC almost recover the performance on write cliff, they sustain the recovered performance for very short time (just a few seconds).

5.7.1 Reads

As against the common expectation, the random read performance of SSDs, in terms of both latency and bandwidth, is worse than the other types of operations even including writes. Further, sequential read performance is degraded over time because of two factors: 1) physical data layout changes on writes, which lead to modulations in internal parallelism, and 2) reliability management overheads on reads.

Read Request Reordering. A flash-aware I/O scheduler can transform the random order of addresses on reads to a sequential order, or schedule them by being aware of the internal parallelism, in order to avoid the poor random read performance. There are several studies in the literature that schedule write addresses [56, 40, 44]; in comparison, reordering read requests has received considerably less attention.

Read Frequency Control. Since block erasures are also involved in reads, reads may shorten the SSD lifespan. This means SSD applications need to be aware of the underlying read disturbances and runtime bad block management characteristics. Read frequency control can potentially improve SSD lifespan as well as the read performance. For example, it can remove the hot read spot regions, or reorganize file system blocks to ensure that the underlying physical blocks are accessed as evenly as possible.

De-indirection Interface. De-indirection interface [2, 103] is a promising approach to efficiently manage an SSD by removing the firmware level indirection (address remapping) [41]. Through the de-indirection interface (nameless write), a file system manages the returned physical address as a result of write, and keep updating the physical space changed by the weal-leveling scheme through a callback. In practice, a nameless write interface has to be aware of the underlying read reliability issues like read disturbances and UECC. This is because the address space can be reconstructed even on reads due to such reliability issues, which can in turn corrupt the data consistency of the file system using nameless-writes.

5.7.2 Writes

Modern SSDs are well optimized to hide GCs, but the throughput of writes significantly drops and the worse-case latency sharply increases when the write cliff is reached. Further, the worst-case latency of SSDs is much higher than HDDs, which implies that it needs to be paid much more attention especially in the context of latency-sensitive applications. Interestingly, the internal DRAM buffer would make the latency imposed by GCs on the write cliff much longer. This implies that the DRAM buffer management is in need of being aware of GCs in order to avoid making the worst-case latency even worse.

Background Task Scheduling. To alleviate the performance overheads on the write cliff, systems may utilize the background tasks by artificially injecting idle times. Since the recovered performance by background tasks is not sustained, the scheduler needs to periodically inject idle periods even under the I/O congestion periods [46]. In addition, considering the fact that the background tasks require long idle periods to fully recover the performance loss on the write cliff, the scheduler can inject idle times in an interleaving fashion and hide potential GC overheads over multiple SSD resources (e.g., flash array storage systems, and SSD RAID systems).

Exposing SSD Firmware API. A more promising approach to handle the background tasks would be exposing APIs that allow a host explicitly to handle flash firmware tasks. For example, similar to the TRIM mechanism, a host can explicitly call GCs or flush data through the DATASET MANAGEMENT command on idle times so that the host CPU-burst time can be overlapped with the SSD internal tasks. Based on our experiments, we believe that directly handling SSD internal tasks is much better approach to handle the write cliff than implicitly scheduling the background tasks.

5.7.3 Advanced Schemes

The magnitude of performance gains with the TRIM commands significantly varies depending on the TRIM request pattern. While SEQ-TRIM can effectively eliminate GC overheads, RND-TRIM has no positive impact on performance. In addition, SSDs require quite long execution times to process TRIMs, which can lead to unexpected performance degradation. To address this, a file system can consider new TRIM management strategies that are aware of the TRIM-process characteristics.

TRIM Buffer and Scheduler. From the beginning of the TRIM process, the host can send TRIM commands by composing target addresses in an increasing order. Similarly, a host module can buffer TRIMs and merge the delete information under the file system in order to transform RND-TRIM to SEQ-TRIM. In addition, it is better to utilize idle times to submit TRIM commands at a system level to avoid potentiol TRIM-latency overheads. A TRIM scheduler can blend legacy I/Os with TRIMs by utilizing system idle periods, thereby hiding the long TRIM execution times.

5.8 Related Work

A lot of prior work focused on improving SSD performance and overcoming flash-intrinsic limitations such as the erase-before-write problem. Flash translation layers (FTL) have been developed to alleviate the write performance degradation by employing different granular address mappings [15, 52, 28, 1]. In addition to FTL, various buffer management schemes [56, 40, 44] have been investigated to improve write performance. Recently, GC schedulers utilizing idle periods have been proposed to avoid heavy performance penalties on write cliff [48, 51]. There also exist several efforts on SSD architecture. For example,[20, 1, 13, 12, 9, 10] revealed internal SSD architecture in detail. In addition, [45, 31] proposed different page allocation strategies to take advantage of the internal parallelism on writes. There exists a scheduler [50] that explicitly handles I/Os by avoiding resource conflicts, thereby improving the degree of parallelism. FlashVM [92] is a flash virtual memory to reap the benefits of random read performance superiority of SSDs, and Facebook flashcache [54] is a read cache leveraging read performance superiority of SSDs to improve MySQL. [7, 61, 82] also proposed SSD cache, filling the I/O gap between main memory and disks in data centers. In general, these SSD-oriented prior studies have been performed based on common expectations. In our experiments and data analyses, we observed many unexpected performance characteristics and reliability issues, which should be addressed by both academia and industry.

5.9 Conclusion

In this chapter, we examined widely held expectations and conceptions on modern SSDs using six different commercial SSDs and a series of experiments. Our experimental results revealed many previously-unreported SSD characteristics from both performance and reliability angles. We also discussed what these characteristics mean to both SSD designers and system designers. Our ongoing work includes designing and implementing system support that can take into account our newly-discovered facts on SSDs, and evaluate this support using a diverse set of workloads drawn from embedded computing, enterprise computing and high-performance computing domains.

Chapter Acknowledgements Chapter 4, in part, is a reprint of the material as it appears in ”Revisiting Widely-held Expectations of SSD and Rethinking Implications for Systems,” Myoungsoo Jung, Mahmut Kandemir, in Proceedings of SIGMETRICS, 2013. The dissertation proposal author was the primary investigator and the first author of this paper.

Chapter 6

Memory Request Scheduling: Improving Parallelism in Solid State Drives

6.1 Introduction

NAND flash-based devices such as Solid-State Disks (SSDs) are becoming increasingly popular in a number of markets. Flash has already become the dominant storage technology in mobile devices for its low-power, density, and resilience to shock. Moreover, flash-based SSDs are making significant inroads into consumer computers such as laptops as well as enterprise applications such as Storage-Area-Networks (SANs) and even high performance computing (HPC). Their lack of moving parts – perhaps the biggest problem with traditional spinning magnetic disk – allows them to serve random requests at a far higher rate than disk. However, care needs to be used when writing to these devices, as flash memory cells wear out with overuse. Therefore, write-heavy workloads are not well-suited for these devices. For such reasons, enterprise and HPC areas have strongly considered SSDs for workloads rife with reads, especially for applications that demonstrate mainly random access patterns.

Such use-cases make sense when considering individual flash memory cells, which are biased towards reading, showing typically between ten and forty times better performance for reads than writes. This is due to a significant duration disparity for the three basic operations in flash: read, write, and erase. Perhaps more importantly, this disparity is exacerbated by the requirement that if a block to be written upon is not already free, an erase must precede the write. Specifically, while reads operate on the tens of microseconds, a write takes hundreds of microseconds, and an erase requires thousands of microseconds. Therefore, if a write occurs to an occupied block, an erase latency plus the write latency is incurred. Moreover, this problem becomes worse for writing to a subset of a block such that existing data therein will be kept. These situations require all three operations: first perform a read, then an erase, and finally write the block down with a mix of new and old data. For the latter two situations – erase/write and read/erase/write – the latency of writing to an SSD approaches that of spinning disk.

Due to this vast disparity in latencies, internal research on SSDs (mostly on the flash translation layer such as in [1]) has been concentrated on avoiding the costs of doing such writes. However, since flash cannot yet match the density-to-cost ratio spinning disk excels at, external research considering the *use* of SSDs rather than the *development* of them has focused on their use as a cache to alleviate the I/O-latency gap between memory and disk [69][7][54]. As a cache, researchers capitalize on the presumed efficacy in serving random reads when compared to spinning disk, and often the caching algorithms employed avoid performing large numbers of writes in order to extend the SSDs lifetime and to avoid the high penalties of writes. These two divergent research directions – internal research working to improve writes and external research developing mechanisms to capitalize on improved performance for reads versus writes – have resulted in an under-performing research landscape.

Ironically, when one examines the *random-read* versus *random-write* performance for SSDs, what is witnessed is often a performance benefit for random writes instead of random reads, due to great difficulty experienced in achieving parallelism for reads, but ease in doing so for writes. This particularly strange *reversal of expected performance* tends to be glossed over or completely ignored in many works. However, it is a critical issue that deserves attention especially for enterprise applications seeking to utilize these devices as caches for their presumed efficacy for random-reads.

6.1.1 Contributions

First, we identify areas where resource contention may exist in an SSD that cause degraded random-read performance. We then develop and present a new request scheduling algorithm that maximizes performance by avoiding internal resource conflicts. We propose *Physically Addressed Queuing (PAQ)*, a request scheduler that is in part inspired by physical address-based memory scheduling techniques [29, 58, 59, 71, 76] and improves random-read performance by identifying and avoiding conflicts for I/O requests. To our knowledge, there is no published work that proposes a scheduler that utilizes physical addresses to optimize random-read performance for SSDs. We summarize our major contributions below:

• *QBM Relocation:* In order to identify and avoid contention for shared SSD resources, we propose to move the queue and buffer management (QBM) functionalities, typically located within the host interface logic, beneath the flash translation layer (FTL). This exposes physical addresses to PAQ, instead of relegating it to work solely with logical block addresses (LBAs).

• *I/O Clumping:* We classify conflicts into groups based on the physical SSD component(s) they share, which provides a framework we use when performing conflict avoidance and improving parallelism within the SSD.

• *I/O Request Rescheduling:* With the ability to identify a request's physical addresses and a framework to "clump" groups of sub-requests together that do not conflict, we present our new queuing algorithm, PAQ, that reschedules requests such that conflicts are avoided to the fullest extent possible. This results in greatly improved random read latency and bandwidth without imposing significant performance degradation for other types of requests.

• *Plane Packing:* The last level of parallelism within the SSD – plane-level parallelism (see Section 6.2.1) – requires a number of constraints to be satisfied. Specifically, we show that given access to the physical addresses of a request's accesses, PAQ can identify more accesses between requests that can benefit from multi-plane mode to improve parallelism.

Using our modified SSD architecture and our proposed PAQ algorithm, *we seek to demonstrate greatly improved read latency for random accesses.* Our experimental analyses indicate improvements over traditional scheduling in bandwidth, IOPS, and average latency. Specifically, for bandwidth, we see as high as 62.7% and in the average case 32.6% improvements. For IOPS PAQ demonstrates as high as 62.6% and in the average case 32.7% improvement. And for latency, we witness as high as 41.6% and in the average case 25.1% improvement. Further, in all cases tested, PAQ results in performances at least as good as those for traditional scheduling.

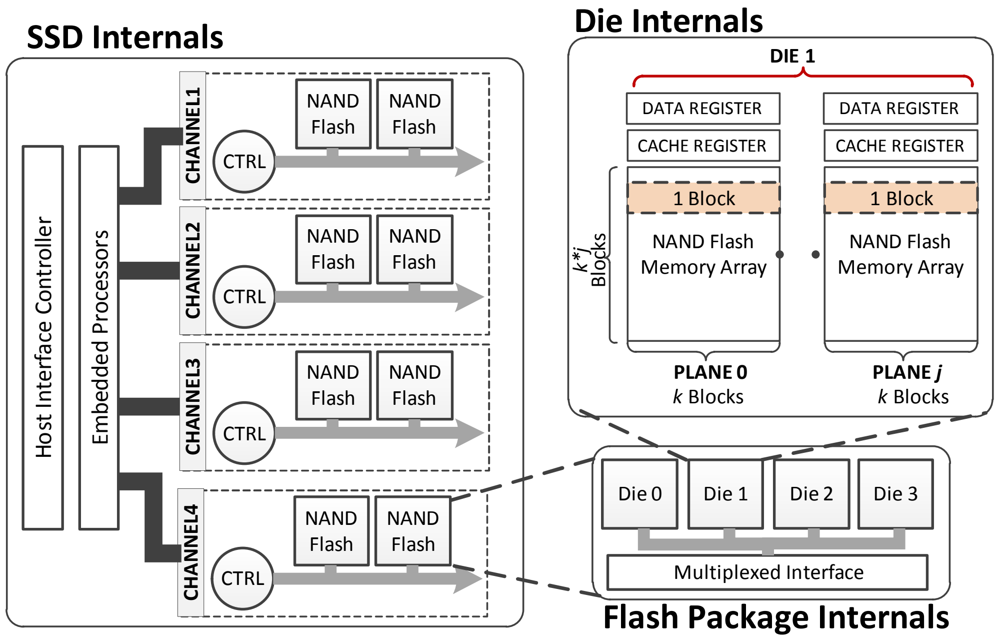


Fig. 6.1 Physical internal architecture of SSD.

6.2 Background

Flash storage presents the first serious departure from the magnetic storage characteristics that were studied for decades. With it come a host of new characteristics and nuances; considerably more than were present in rotational magnetic disk. While many prior studies have already explored the finer details of flash characteristics, it is critical to at least have a basic background on the medium to appreciate the benefits of our proposed Physically Addressed Queuing. To that end, in the following subsections we present a cursory overview of NAND flash architecture and details of one particularly high-potential but underutilized access mode available in SSDs.

6.2.1 High-Level Architecture of SSD

While flash is a powerful storage medium for its fast random access speeds, low power consumption, and lack of moving parts, it requires a number of mechanisms to enable it to work efficiently. Being packaged most commonly in the form of Solid State Disk (SSD), there are numerous hardware components and software layers within an SSD that work in tandem to provide access to the flash medium within.

A subset of the physical internals relevant to our work are shown in Figure 6.1, and represent typical hardware used in commercial SSDs [75, 1, 20]. There are four main levels that *increase parallelism* in an SSD:

• *Channels.* At the highest level, there exist multiple channels that are operated by embedded processors; each can be operated in a completely independent fashion.

• *Flash Packages.* Each channel is shared by NAND flash packages for transmitting data and operation messages.

• *Dies.* Within each NAND flash package are one or more dies, each sharing one or more buses upon which their communication is interlaced. Communication is interlaced using a chip enable (CE) pin, leading to a reduction in I/O bus complexity but also adding potential for contention over the CE pin by requests.

• *Planes.* Finally, within each die exist one or more planes, the smallest unit to serve an I/O request in a parallel manner. Each plane shares a wordline for accessing the flash memory cells, which leads to the important consequence that multiple requests can be served simultaneously in a single wordline access. However, in order to do so, the requests must adhere to the *plane-addressing rule*, a constraint we expand upon in the following section. Resource sharing occurs at each level, which was intended to decrease the complexity and therefore cost to design and manufacture SSDs.

A subset of the relevant software layers are shown in Figure 6.2, and are described below:

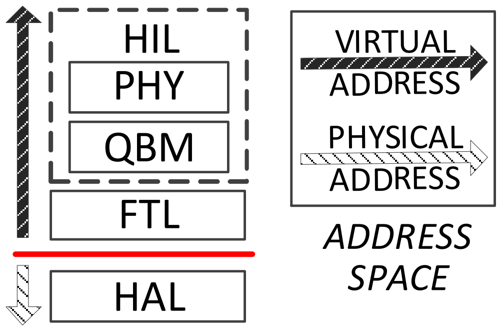


Fig. 6.2 Software stack of an SSD.

• *Host Interface Layer (HIL)*: The HIL is responsible for communication between the host system and underlying layers within the SSD. Specifically, the HIL performs parsing of I/O commands, hand-shaking based on the interface protocols, initiating data transfer, and committing NAND transactions to underlying layers. The raw communication protocol portion of the responsibilities are handled by the Physical layer (PHY), whereas responsibilities related to I/O scheduling and buffering are dealt with by the Queue and Buffer Management layer (QBM).

• *Flash Translation Layer (FTL)*: The FTL is responsible for address translation between the host address space, which contains virtual or logical addresses, and the physical address space, which fully specifies the channel, flash package, and flash die where the data is located. The FTL separates the logical from the physical address space to allow for higher levels to treat the logical addresses as they have historically been treated for conventional block devices, and to allow the lower levels to handle the complicated nuances of NAND flash memory (e.g., in-place updates are not possible without preceding read and erase operations).

• *Hardware Abstraction Layer (HAL)*: The HAL is a device driver, which manages physical NAND flash memory. Specifically, it is charged with committing NAND transactions using the physical address provided to it from the FTL to underlying flash memory, and periodically checking each NAND flash package to monitor transaction statuses (using the ready/busy pins).

6.2.2 Multi-Plane Mode Operation

Besides the three basic NAND operations we mention earlier – read, write and erase – there exist a number of advanced modes and operations that seek to improve NAND parallelism but come with constraints that must be adhered to in order to achieve such performance. In this work one mode is particularly amenable to our proposed clumping strategy where we examine accesses between distinct requests in the hope of improving performance. In that mode, *multi-plane mode*, operations serve multiple requests simultaneously by sending them together down the same wordline. This mode has the potential to improve performance *n* times the number of planes attached to a particular wordline, but comes with the caveat that these requests *must* target the exact same page offset in a block, the exact same die address, and indicate different plane addresses.

Therefore, there is a very limited field of requests that can be served in this mode when one considers requests individually. However, by examining a number of transactions in separate requests, which are all waiting in the queue, and intelligently selecting those which can be *packed* together for expedited service down a single wordline, the likelihood of executing transactions that fulfill multi-plane mode constraints is increased. This is the foundational idea behind our plane-packing optimization, which we show can improve performance immensely, especially for larger queues where the pool of available requests is increased.

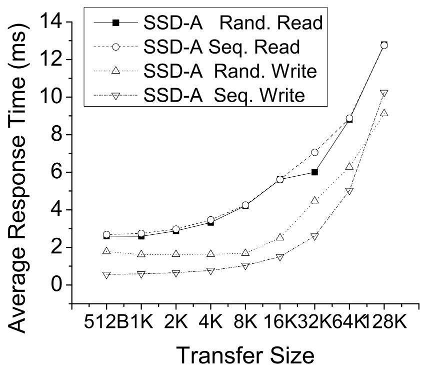
6.3 Random Write vs. Random Read Performance

While the objective to simplify SSD design is achieved via resource sharing, what is not readily expressible through the physical diagram is the potential for *conflicts* between requests that end up contending for such a shared resource. However, these conflicts are relatively small for sequential and random writes. At first blush, this appears contradictory since flash memory cells perform poorly for writes, but upon further inspection the independence from data layout that writes enjoy leads to massive increases in internal parallelism. For instance, when a write request is issued, whether it is random or sequential, the FTL of the SSD will work to split up and stripe that sequential request or simply will send the random requests down separate channels. These requests are destined for packages and dies that are the least busy, which results in resources in the SSD being utilized in a fairly balanced manner without “hotspots” arising in particular areas that bottleneck performance. Moreover, writes can easily utilize multi-plane mode since there is no data layout that must be adhered to; the writes can simply be packed together in any manner.

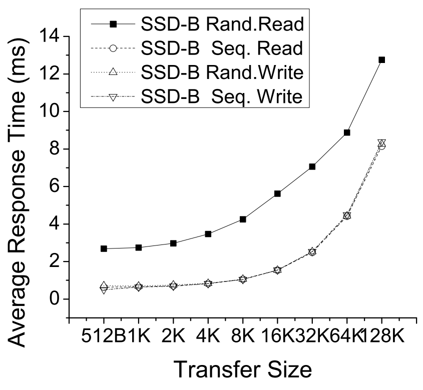
Such parallelism is not as easily achieved for reads. Specifically, in order for a read to occur, there must be some data to be read, and that data must reside in a particular block, on a particular die, within a particular flash package, and along a particular channel within the SSD. These particularities result in a very rigid performance dependence upon data layout and access sequence, which in turn results in requests queuing up behind each other and sequentiality sprouting up that strangles parallelism. *In addition, the QBM, in its current position above the FTL, is helpless to do anything about such performance dependence because it solely has knowledge of the virtual address – there is no way for it to intelligently reorder accesses to decrease conflicts.*

To validate our concerns regarding random read performance, we performed tests on two commercial Samsung 470 series SSDs manufactured in 2010, which we hereafter refer to simply as "SSD-A" and "SSD-B." In these tests, we used the IOmeter characterization tool [35] to perform sequential writes, sequential reads, random writes and random reads, each for nine separate transfer sizes ranging from 512 bytes to 128 kilobytes for both drives, and allow at most 16 outstanding I/Os to exist at any given time. The amount of data moved by the tool fills about 80% of both SSDs.

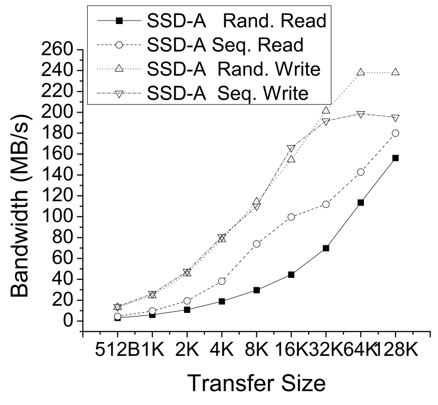
Both SSD-A and SSD-B utilize 32 *n*m fabrication process NAND flash memory packages, employ a DDR2 flash interface (144Mbs), and utilize Samsung's second generation S3C29MAX01 controller. ARM-based multi-core processors with dual-cache chips manage the SSD internals, and the host interface is connected via Serial ATA Generation 2 (3Gbps). The size of DRAM



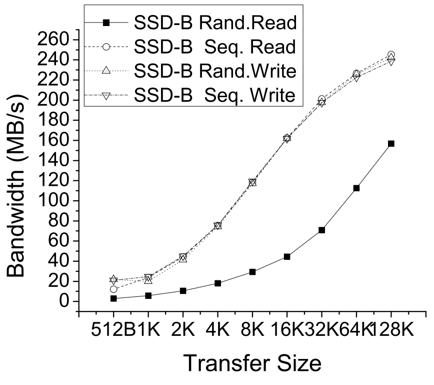
(a) Latency (SSD A)



(b) Latency (SSD B)



(c) Bandwidth (SSD A)



(d) Bandwidth (SSD B)

Fig. 6.3 Average response times and bandwidth results as transfer sizes varies under two commercial SSDs.

cache employed in both devices is 256MB, comprised of two DRAM chips having 667Mbps data rate. The first drive, marked “SSD A,” has a capacity of 64GB and operates using 4 channels and 16 packages. The second drive, marked “SSD B,” has a capacity of 256GB and operates using 8 channels and 64 packages. While manufacturers do not publicize exactly how many dies are in each package, single-, dual-, quad-, and octal-die package are all possibilities in production. We suggest that for these devices dual-die package is most likely based on the price point and the date manufactured. The results of our tests are shown in Figure 6.3. Specifically, Figures 6.3(a) and 6.3(b) plot variances in latency as transfer size is increased for SSD A and SSD B respectively, and Figures 6.3(c) and 6.3(d) show variance as transfer size increases in terms of overall bandwidth.

In the latency results, for SSD A, random reads and sequential reads perform similarly, but both are still far worse than either types of the writes by at least 25% and as high as 361%. For SSD B, there exists a clear case that while all other operations incur approximately the same latencies, random reads suffer by at least 56% to at most 319% in comparison. The bandwidth results tell a very similar story, but bring to light the increasing gap as transfer size increases. In all cases tested, random writes outperform random reads, in direct contrast to what a majority of the literature on flash memory would lead one to believe. It should be noted that, read operations tested are not affected by any transactions related to garbage collection activities because read and write tests for both latency and throughput are performed in an entirely separate fashion. Also, note that the fraction of DRAM that might be used for buffering I/Os is about 0.48% (SSD-A) and 0.12% (SSD-B) in our tests. In practice, the fraction of DRAM for buffering I/Os is even smaller because it is concurrently used for maintaining metadata and in-memory data structures of the FTL.

6.4 Physically Addressed Queuing

In order to improve the poor random-read performance we demonstrated above, we propose a new request scheduling scheme named *Physically Addressed Queueing (PAQ)*. Unlike previous schedulers who do not have access to the physical addresses of requests, with PAQ we propose moving the QBM layer out from the HIL and beneath the FTL to provide such crucial functionality. With exposure of the physical layout to our PAQ scheduler, we are able to positively identify requests that will cause conflicts as they concurrently contend for the same resources. Using these physical addresses, we present a classification system for conflicts, and describe how PAQ can aggregate groups of requests together that do not share conflicts. Such groups of requests without interdependence we term a *clump*[1], and we show that PAQ works to build clumps in a conflict-first bottom-up fashion such that total contention is reduced and SSD performance is improved. Last, we discuss how PAQ can improve multi-plane mode performance immensely given the physical layout of requests, which is an optimization that can orthogonally improve overall performance.

6.4.1 QBM Migration

Since the QBM layer must be exposed to physical addresses of requests in order to make intelligent decisions that decrease conflict, we propose migrating the QBM out of the HIL and positioning it directly beneath the FTL. This migration is visually depicted in Figure 6.4. As is indicated by the virtual and physical address spaces and the horizontal line separating the two,

[1]While related, clumping differs from memory request coalescing in that clumping does not combine multiple requests into one, nor does it remove duplicate requests [19]. The main purpose of clumping is to avoid conflicts between requests.

moving the QBM out of the HIL and down below the FTL provides physical address exposure to the QBM layer and therefore PAQ.

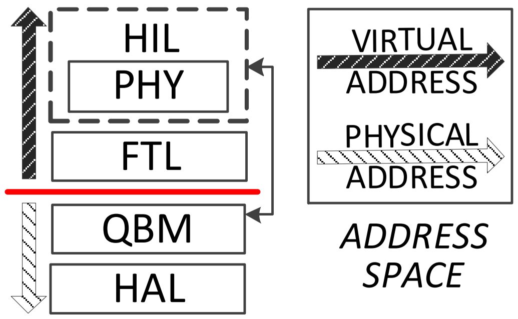


Fig. 6.4 Firmware layers within a solid state disk.

This change is achievable in practice because the PHY and QBM layers operate within distinct protocols. Specifically, in the SATA interface, while the PHY layer resides in the physical, link and transport layers defined in the SATA protocol, the QBM resides in the application layer, allowing for migration of the QBM without necessitating changes to existing manufacturing processes or established protocols.

6.4.2 Conflict Classification

To build a queuing mechanism that can identify and properly schedule around contention, we first propose the following conflict classification framework:

• *Domain:* a set of requests that require access to the same channel. Concurrently scheduling requests in a domain increases the potential to cause channel I/O bus contention, but also has the potential to exploit parallelism by interleaving requests across multiple flash packages and dies.

• *Cluster:* a set of requests requiring access to the same flash package. A cluster may contend for the same NAND I/O bus in accessing a flash package, which incurs domain-level conflicts. However, the requests in a cluster enjoy die-level interleaving parallelism if no node-level conflict exists.

• *Node:* a set of requests that require access to the same flash die. A node always incurs resource conflicts, and it also has the potential for a number of resource contentions among the requests in same domain and cluster, including NAND flash register, NAND I/O bus, and channel I/O bus.

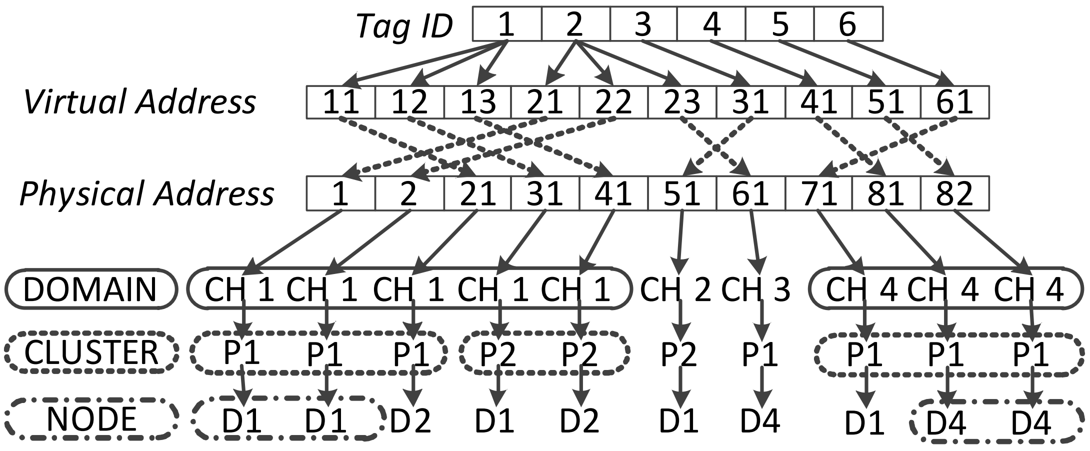


Fig. 6.5 An example that shows PAQ conflict classification. CH, P and D denote channel, package, and die, respectively.

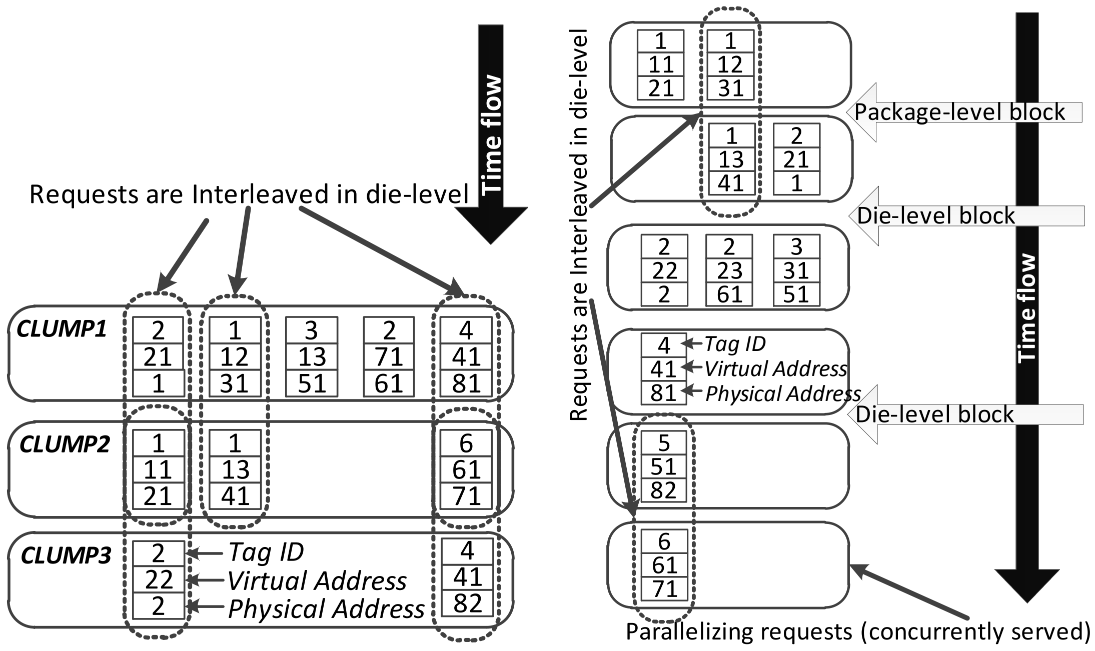
The fact that each level incurs conflicts at its level and *all levels above it* is a critical part of conflict classification. To visualize this inclusive hierarchy, Figure 6.5 provides an example of requests in the command queue, along with the virtual and physical addresses of the transactions within those requests. Beneath the transactions, we show what channels, packages and dies the transactions require access to. We also demarcate potential conflicts using rounded rectangles surrounding all transaction targets that are the same physical component. Domains are indicated using a solid-rounded rectangle, clusters using a dotted-rounded rectangle, and nodes using a dot-dashed-rounded rectangle. The resulting set of clumps PAQ constructs is shown in 6.6(a).

In the simplest case, for request ID #3, it requires access to LBA 31, physical address 51, channel 2, package 2 and die 1. As it is the only request in the queue seeking to access channel 2 (no node-, cluster- or domain-level conflicts exist), it is free from conflict and may be executed immediately. However, if we look at a more complicated scenario resultant from request ID #2, we see that it is attempting to access physical addresses that exist along two channels, two packages, and two dies. This request causes domain- and cluster-level conflicts with request ID #1, and also results in domain-, cluster- and node-level conflicts with other transactions in the same request. Therefore, even for a command queue with solely a single request, there may exist conflicts between multiple transactions in that request.

While conflicts occur for accesses to the same channel, package, or die, it is important to note that the window of conflict is wider for lower levels due to a pipelining of each transaction through the architecture, resulting in a slightly more complex landscape for conflicts. For instance, it would be highly inefficient to conclude that a channel and package is unavailable just because a single die on that path is busy. Luckily, two simple intuitions prove effective in solving this complexity and are the foundation of our scheduling algorithm to build low-contention clumps.

6.4.3 Clump Composition

With the physical addresses available to the QBM layer, and a classification scheme that allows us to identify conflicts and their location, we construct our PAQ scheduling algorithm around the following two intuitions: (i) Lower-level conflicts are most costly – PAQ should avoid them if possible, and (ii) if PAQ can schedule a single transaction from an area of conflict while doing other work and only later execute the other contentious operation, it can avoid contention



(a) Constructed PAQ clumps (b) VAQ-style scheduling

Fig. 6.6 An example that shows PAQ conflict classification and the associated clumps.

while achieving parallelism. These intuitions lead us to establish the following goals on how PAQ should perform clump composition, the most critical contribution of our work: (Goal 1) Add transactions incurring conflicts in the lowest levels first. (Goal 2) For node- and cluster-level conflicts, never schedule a clump such that either would have greater than one transaction issued concurrently from it. (Goal 3) Continue adding transactions to the clump, prioritizing for low-level conflicts, until no more can be added without breaking Goal #2. Succinctly, *PAQ attempts to build clumps in a bottom-up, conflict-first fashion such that the lowest level with contention does not have conflicting transactions in the clump.* To properly compare our PAQ clumping strategy, let us consider how the requests in Figure 6.5 would be handled by the *default* (traditional) scheduling scheme, which we name *Virtually Addressed Queuing* (VAQ). As shown in Figure 6.6(b), request #1 would be sent to the FTL for execution first, since VAQ performs FIFO scheduling from the command queue.

There would be brief intra-request contention for channel #1 and package #2, but this time is relatively small compared to the time spent at the die. Since there was no intra-request contention for any dies, the FTL would return stating that the request was being handled, and VAQ could then move on to attempt to issue request #2. Request #2 will experience intra-request contention for its first two requests, whom both attempt to access die 1 in package 1 on channel 1, which stall the FTL and prevent VAQ from proceeding to request #3 despite the fact that it has no contention with any transactions. Fast-forwarding to requests #4 and #5, we can also see that these two requests compete for die 4 on package 1 on channel 4, stalling request #6 for little good reason since it is targeting a separate die. This greedy, FIFO nature of request serving is a necessity for VAQ; because it has *no* idea where the actual transactions are destined (since it can only see virtual addresses, which tell it nothing), FIFO is the best it can hope to do.

PAQ, on the other hand, attempts to leverage its knowledge of transaction destinations by constructing groups of transactions that can run concurrently without contending for resources in a significant way. As mentioned previously, we term such groups of low-contention transactions "clumps." To do so, PAQ examines the physical addresses of transactions in the command queue and schedules such transactions in a "conflict-first, bottom-to-top" manner. Specifically, the lowest level in the hierarchy contains node-level conflicts, of which we only have two (indicated by the dash-dot-rounded rectangles): between physical address 1 and 2 and between physical address 81 and 82. Therefore, PAQ chooses to place transactions on physical address 1 and 81 in clump 1. Then, examining one level up, PAQ again identifies and prioritizes requests from areas of conflict (i.e., clusters), *which are not already serving a transaction* (e.g., PAQ would never add addresses 2 or 21 since they both share package 1, whom is already serving a transaction for this clump from physical address 1). Specifically, PAQ then adds the transaction on physical address 31 to the clump as it is the only transaction with a cluster-level conflict but who is not already serving a request for the previously selected transactions in clump 1. Third, PAQ performs the same process for the channel level: find all domain-level conflicts not already serving requests for previously selected transactions in the clump and add them. However, no transactions meet this criteria. Last, PAQ selects transactions having *no conflicts at any level* (it is always safe to schedule them), and adds them to the clump. This will add physical address 51 and 61, completing clump 1 to include all of: 1, 31, 51, 61, and 81.

While there may be very brief pauses (on the order of tens of nanoseconds) for domain-level conflict between physical addresses 1 and 31, once the request is passed to the package and then die, *PAQ will enable all dies in the example to operate concurrently*. This concurrency allows for interleaving between transactions that use the same package but separate dies, which we demarcate using dotted-vertical-rounded rectangles in Figure 6.6(a). As can be seen in that figure, all requests can be served in just three clumps whom all enable interleaving of dies in the same package, whereas VAQ scheduling (see Figure 6.6(b)) requires six request sets with only two requests interleaved.

6.4.4 Plane Packing

In theory, multi-plane mode operation should allow SSDs to achieve *n*x speedup (where *n* is the number of planes in a NAND flash), because *n* pages can be served simultaneously. However, such speedup is generally not reached because traditional VAQ is ignorant of physical addresses, which is a prerequisite to intelligently constructing multi-plane-mode operations. Moreover, since traditionally the underlying FTL is oblivious of the device-level queue and requests therein, it is not possible for the FTL and HIL to collaborate to construct multi-plane mode

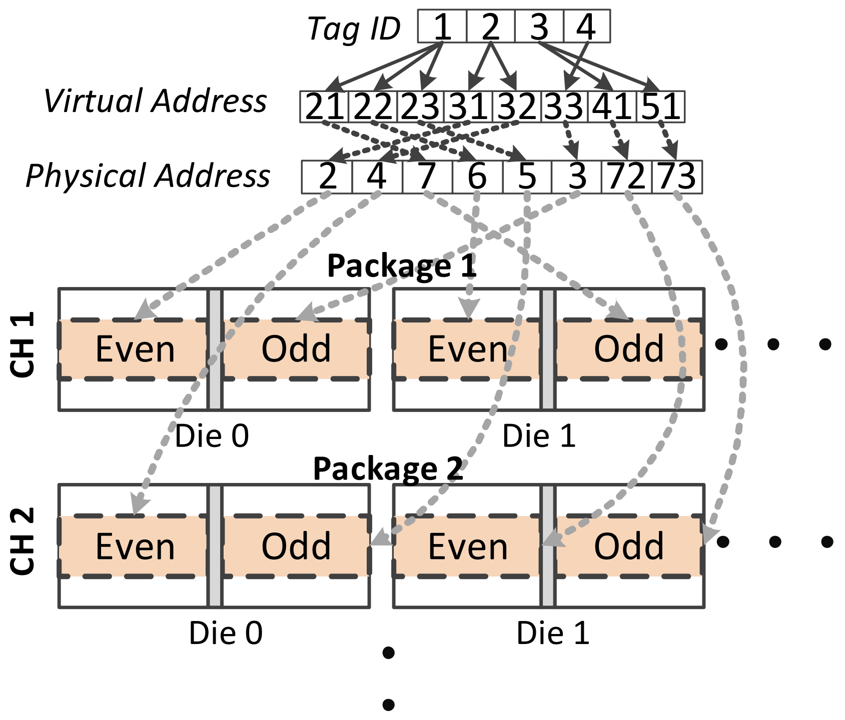


Fig. 6.7 Plane packing in PAQ.

requests. For example, in Figure 6.7, even though a VAQ scheduler will reorder commands in attempt to satisfy the plane-addressing rule, the order of transactions associated with the reordered commands is still not sufficient to build multi-plane-mode operations; without knowledge of the physical addresses, it is purely luck for the FTL to be able to execute multi-plane mode transactions. As a result, in our example in Figure 6.7, VAQ would only permit a multi-plane mode operation to be built by the FTL: (72, 73). Overall, to complete the requests of the VAQ scheduler, the FTL would commit and the HAL should handle six separate transactions. In contrast, since the physical addresses are visible to our PAQ scheduler, the transactions in the queue are built with respect to plane addresses by packing them. In the example, PAQ commits solely four transaction pairs to the HAL, all executing in multi-plane mode: (2, 3), (6,7), (4,5), and (72, 73).

6.4.5 Implementation of PAQ Scheduling

Algorithm 1 adding_io_request(tag). Note that address translation for read requests occurs before the actual data transfer starts in PHY.

```
 1: head_lpn := tag.lsn % the size of page
 2: tail_lpn := (tag.lsn + tag.length) % the size of page
 3: {get physical address info and record}
 4: while head_lpn != tail_lpn do
 5:   if tag.req_type = read then
 6:     ppn := ftl.translate_physical_address(head_lpn)
 7:     pair(ch_id, flash_id) := ftl.parse_channel_and_way(ppn)  pair(die_id, plane_addr) :=
        ftl.parse_die_and_plane(ppn)
 8:     nand_trans := build_trans(ppn)
 9:     add(nand_trans, clump_table[ch_id][flash_id])
10:     device_queue.push_back(tag)
11:     send_ack(tag)
12:   else
13:     device_queue.push_back(tag)
14:     send_ack(tag)
15:     ftl.page_basis_commit(head_lpn)
16:   end if
17:   head_lpn += 1
18: end while
```

Algorithms 1 and 2 describe how our approach is implemented and manages PHY and QBM, respectively, of the HIL. First, for the PHY management, address translation occurs between the time pre-information (called a *tag*, which includes information like logical block address and request size) is received, and the time acknowledgment is sent (if the request type is read). For requests that are not a read, the PHY sends acknowledgment first and bypasses the request to the underlying FTL. The reason behind these two different strategies is that the addresses of read are decided at write time, for quick translation. In contrast, in the write case, the decision of the physical target has not yet been determined by the FTL.

Once the PHY translates the physical address for the read request, it adds information along with the physical address into a table, called the *clump table*. The clump table is a tool for communication between the PHY and the QBM, which tracks the physical addresses (plane and die) per flash package. It should be noted that the latency incurred from the computation overhead in address translation can be hidden by overlapping it with the process of receiving the tag and sending an acknowledgment. Once a tag is added to the device-level queue, the physical address information associated with it is preserved in the clump table until the request is served by the QBM. When the device-level queue is not empty, from the front of the queue, the QBM commits NAND transactions to the underlying HAL by visiting each entry of the clump table associated with the target tag. As described in the previous section, the QBM first checks whether a node-level conflict related to the tag exists or not. If no request is found in the node level, it then checks the cluster level. The QBM commits the transaction if it is found in the cluster level. If not, the QBM checks the domain level in a similar fashion as it did for the cluster-level. As previously alluded to, the QBM must maintain information regarding distinct transactions that have previously been issued to a particular cluster. Otherwise, the possibility exists that two transactions in a cluster might get simultaneously issued (and cause conflict). In the case that there are no requests found having node- or cluster- or domain-level conflicts, the QBM will simply issue any transaction headed for the currently selected NAND flash by identifying one in the clump table. Once the QBM commits the transaction, it moves on to perform the same process for the next NAND flash and its associated entries. It should be noted that even though the QBM does not physically reorder transactions in the device-level queue, the order in which these transactions are actually issued is changed.

Algorithm 2 read_data_transfer(tag). Serving I/O request and managing QBM.

```
 1: for ch_id := 0...n do
 2:    for flash_id := 0...m do
 3:       {build clump}
 4:       prev_tag := check_and_wait(ch_id, flash_id)
 5:       if prev_tag.committed_trans = 0 then
 6:          device_queue.release(prev_tag)
 7:       end if
 8:       trans := get_trans_from_node(tag, clump_table[ch_id][flash_id])
 9:       if trans is not assigned then
10:          trans := get_trans_from_cluster(tag, clump_table[ch_id][flash_id])
11:       end if
12:       if trans is not assigned then
13:          trans := get_trans_from_domain(tag, clump_table[ch_id][flash_id])
14:       end if
15:       if trans is not assigned then
16:          pair(tag, trans) := get_another_req(clump_table[ch_id][flash_id])
17:       end if
18:       {packing}
19:       assoc_trans := get_associate_plane_trans(tag, clump_table[ch_id][flash_id])
20:       trans := packing(trans, assoc_trans)
21:       hal.commit(ch_id, flash_id, trans)
22:       tag.committed_trans += 1
23:    end for
24: end for
```

6.5 Experimental Setup

6.5.1 NAND Flash SSD Simulator

To implement and evaluate PAQ, we required a high-fidelity simulator that was capable of capturing cycle-level interactions between the many components in an SSD. While there exist a few well-known SSD simulators such as Microsoft Research Lab's SSD extension [1] to DiskSim [5] and FlashSim [60], neither of these nor most others deliver the high-fidelity results we require. In order to experimentally evaluate PAQ, we developed a cycle-accurate NAND flash simulator [2] that provides: (i) Fine-grained NAND command handling, so that conflicts between competing commands are made evident (ii) Advanced command implementation with maintenance of strong constraints (i.e., to properly evaluate our multi-plane mode optimization) (iii) Awareness of intrinsic latency variations for diverse NAND I/O operations based on the current state of the memory cells. In addition, we built a simulation framework that performs all of the high-level tasks of an SSD, which builds and issues requests to many concurrent instances of the NAND flash simulator. This provides us with a cycle-accurate SSD simulator.

6.5.2 SSD Configuration and Schedulers Tested

In this work we configure our simulation environment as having 8 channels, 8 flash packages per channel (64 total), double-die package format (128 total) and a queue size of 32, which is the standard for SATA-based SSDs, with a page-level mapping FTL similar to the one explained in [1, 10]. We believe this represents the typical modern SATA SSD, but we experiment with varying queue and channel counts in the sensitivity section to provide further insight into

[2]The source code of this simulator [49] can be downloaded from http://www.cse.psu.edu/∼mqj5086/nfs.

	Write instructions	Read instructions	Percent of random-writes	Percent of random-reads
fin1	4,099,354	1,235,633	97.86	96.98
fin2	653,082	3,046,112	99.2	97.49
web1	212	1,055,236	99.06	93.53
web2	990	4,578,819	100	93.33
web3	1,260	4,260,449	99.84	91.25
usr1	1,333,406	904,483	94.23	92.2
usr2	3,857,714	41,426,266	96.24	88.97
usr3	1,994,612	8,575,434	97.02	82.77
prn1	4,983,406	602,480	76.4	88.6
prn2	2,769,610	8,463,801	97.16	90.5
sql1	1,423,458	606,487	93.5	89.91
sql2	73,833	87,058	16.07	73.66
sql3	38,963	5,136,405	92.95	71.96
sql4	21,330	10,050	46.89	86.95
msnfs1	1,467,625	41,772	87.23	99.79
msnfs2	2,100,032	121,697	66.71	88.8
msnfs3	500	0	0	99
msnfs4	4,014	338	22.52	64.79
msnfs5	3,003,205	9,624,191	97.86	96.98
msnfs6	3,040,098	9,941,612	100	97.51

Table 6.1 Trace decomposition into the number of writes and reads, and the percentage of random-reads and random-writes issued.

how our scheme would behave under varying protocols and future SSDs. We evaluate the following queuing strategies:

- **VAQ**: The default virtually addressed queuing scheme.
- **PAQ0**: Physically addressed queuing, only using our plane-packing optimization.
- **PAQ1**: Physically addressed queuing, only using our clumping optimization.
- **PAQ2**: Physically addressed queuing, using both plane-packing and clumping optimizations.

6.5.3 Traces

We wanted to validate performance across a number of traces from *actual enterprise applications*. To that end, we collected traces of workloads representative of the following enterprise areas (with the corresponding abbreviation we use afterwards in parentheses): online

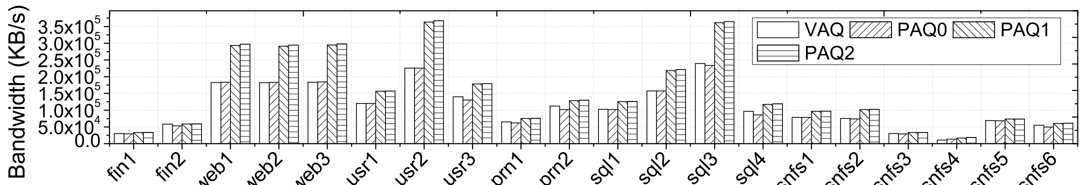


Fig. 6.8 Average bandwidth comparison between VAQ and PAQ.

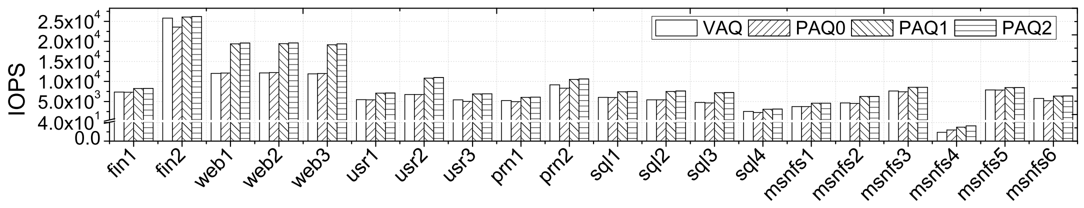


Fig. 6.9 Average I/Os per second (IOPS) comparison between VAQ and PAQ.

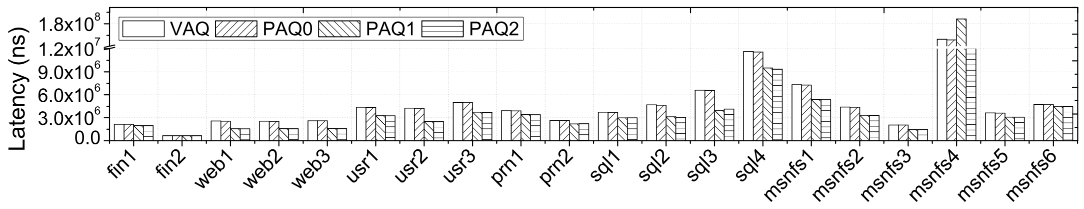


Fig. 6.10 Average latency comparison between VAQ and PAQ.

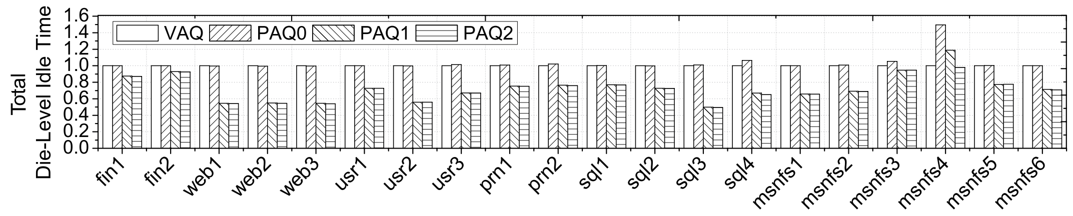


Fig. 6.11 Idle times for VAQ and PAQ.

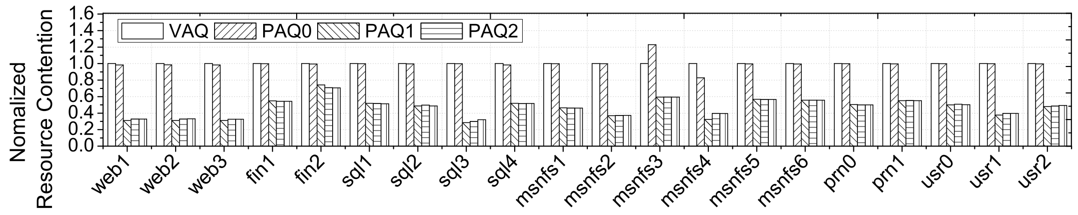


Fig. 6.12 Normalized total contention time comparison between VAQ and PAQ.

transaction processing (fin), search engines (web), shared home directories (usr), print serving (prn), relational database management systems (sql), and remote file storage servers (msnfs). There exist multiple traces with the same prefix but varying numeric extensions; some of these are traces of different points in the lifetime of the application, and others are from distinct applications that happen to fall into the same category. These traces are available at [3] and [88], and the latter was originally detailed and presented in [78]. In order to provide the reader insight on the high-level nature of the traces and the overall landscape of our trace selection, we have characterized each traces total number of I/O instructions and proportion of access type and pattern in Table 6.1.

6.6 Experimental Results

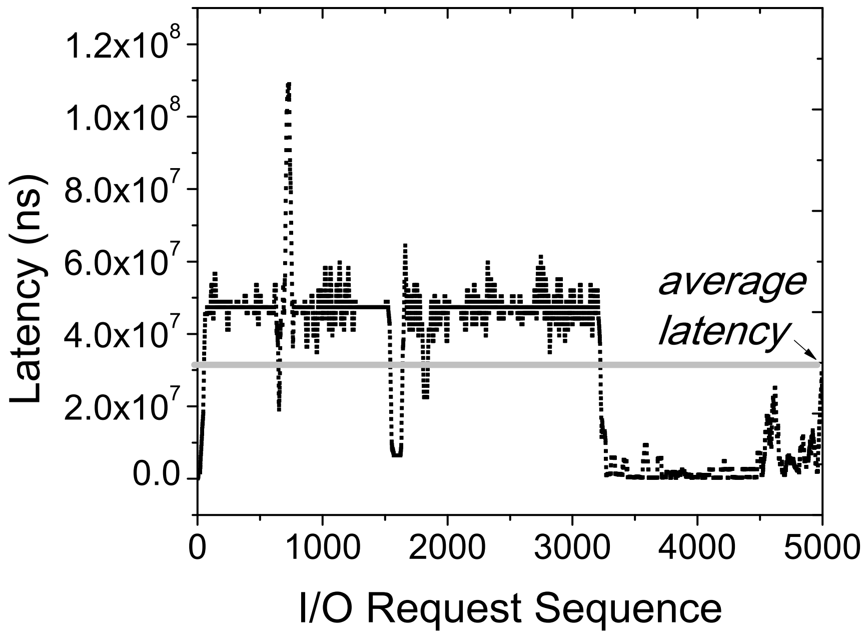
In evaluating PAQ we quantify its impact on overall performance relative to VAQ. Specifically, for performance, we measure total bandwidth, I/Os per second (IOPS), and average latency for all of the traces we discussed earlier. To connect those performance improvements to our original goal of improving parallelism and utilization, we also measure and report contention time and idle time for each trace for the default scheme versus PAQ. Moreover, to give a low-level idea on the impact of different queuing schemes, we show the exact latencies of each I/O request for a series of requests in one of the traces for PAQ versus VAQ. Finally, we present sensitivity analysis along the axes of queue size and channel count to demonstrate PAQ's impact on flash storage devices having various interconnection protocols (e.g., PCI-E, SATA, SAS, etc) and how it will impact future devices, presumably having higher channel counts and larger queues.

6.6.1 Aggregate Performance: Bandwidth, IOPs, and Latency

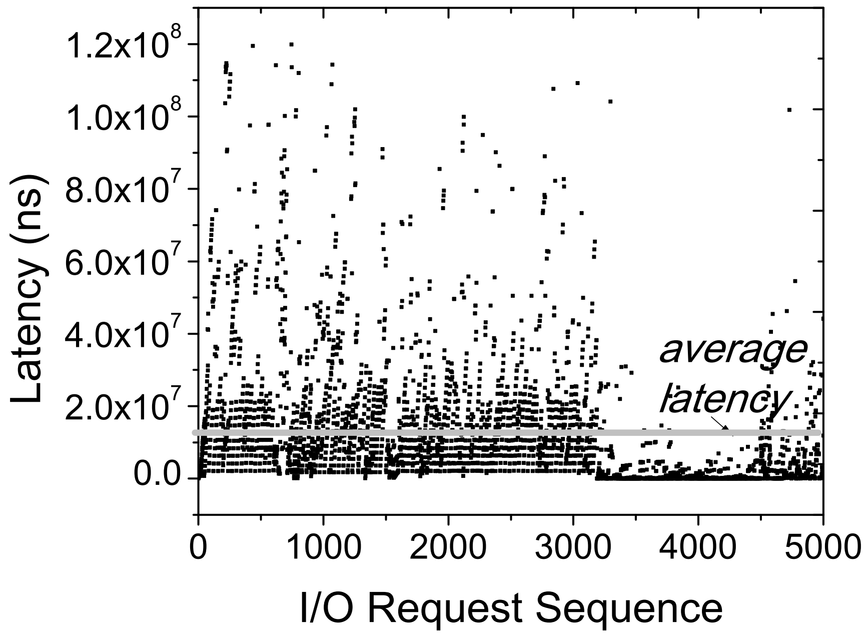
As can be seen in Figure 6.8, PAQ improves bandwidth for read-intensive workloads immensely; five of the twenty workloads exceeding a 100MB/s improvement. Furthermore, for all of the web workloads the improvement is greater than 100MB/s and *such workloads are comprised of greater than 90% of random reads* (see Table 6.1).

Lastly, PAQ2 never hurts the performance for any workload, regardless of whether it is read- or write-oriented or has mostly random or sequential requests. PAQ0 occasionally hurts performance because with solely the default SATA queue size of 32 there is a limited window of requests with which to consider packing. Later, in performance sensitivity analysis we demonstrate that with increased queue sizes this degradation disappears. The IOPS measurements shown in Figure 6.9 tell the other side of our story; those traces that showed particularly low bandwidth were generally dominated by much larger numbers of requests whom had smaller sizes than those with high bandwidth (a good example is fin2). Just as with bandwidth, for random-read intensive workloads, PAQ2 does a great job improving performance and never performs worse than VAQ. However, for our worst performing trace, msnfs4, the IOPS are so few that they appear to be missing from the figure. In that case, despite Table 6.1 describing it as mainly issuing sequential-writes, we find that these sequential writes are intermixed with very small random-read requests.

This write-performance degradation occurs for all schedulers as a result of the great disparity between read and write latency and these random reads undermining the parallelism of the sequential-write requests. Even in the face of such random-read interference, PAQ delivers 1.41 times the IOPS and bandwidth of VAQ.

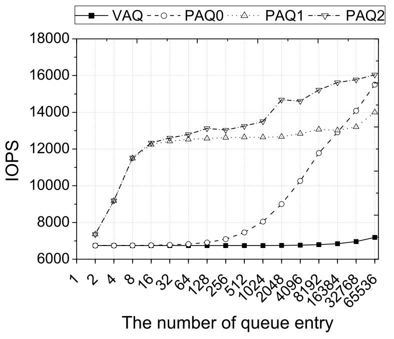


(a) Latency (VAQ)

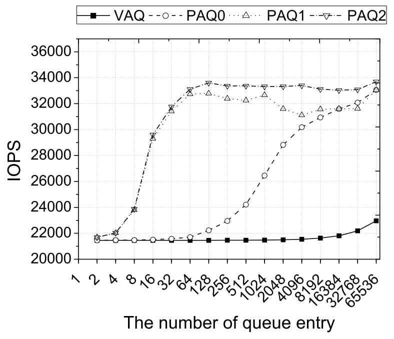


(b) Latency (PAQ)

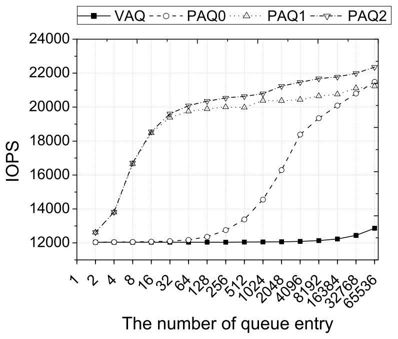
Fig. 6.13 All latencies incurred for requests from trace sql3 for VAQ and PAQ. Note that average latencies are shown using a horizontal line in each case.



(a) 4 Channels



(b) 8 Channels



(c) 16 Channels

Fig. 6.14 IOPs sensitivity to varying queue and channel sizes.

6.6.2 Quantifying Parallelism: Idle Time and Contention

While raw throughput and total I/Os completed per second are important metrics, it is still critical that individual transactions from operations are not delayed so long that average latency increases a great deal. Interestingly, what we find as shown in Figure 6.10 is that, *in the average case*, PAQ2 actually decreases latency fairly significantly. Moreover, PAQ2 improves performance in a similar proportion *across nearly all workloads, regardless of their access type breakdown*. Last, we see that while in bandwidth and IOPS PAQ1 and PAQ2 performed very similarly, in the case of msnfs4, using just plane-packing or clumping alone did not improve performance as much as together, giving credence to our belief that both are needed for best performance.

We originally conjectured that the poor performance we observed in real SSDs in Section 6.3 for random reads was a direct result of difficulty achieving parallelism in the device. While we have demonstrated aggregate performance improvements using PAQ2, we further seek to directly demonstrate that these improvements were correlated to increasing utilization of the individual dies (thereby reducing idle time) and reducing total contention time. Idle time is measured as the total time each die spent without serving any transaction, and is shown again for each trace in Figure 6.11. It has been normalized to 1 so that all improvements can be clearly seen. Decreases in total idle time for PAQ2 around 20% are witnessed on average, with a few read-oriented traces reaching improvements as high as approximately 60% and a few write-oriented traces experiencing almost no improvements. Contention time is somewhat harder to measure than idle time, but we formalize it as *the amount of time a NAND transaction spends waiting on the I/O bus within a flash package to get to a specific die.* As the results in Figure

6.12 reflect, contention time does not directly reflect idle time. While PAQ2 does not always result in a great reduction of idle time, it does result in a very significant reduction in contention time across *all* traces.

6.6.3 Overheads of PAQ

As PAQ involves advanced scheduling to improve performance, it is worthwhile to consider the overheads such scheduling might cause. The process of checking the clump table described in Section 6.4.5 is theoretically bounded by O($n * m$) for each read transfer, where n is channel count (state-of-the-art is 4$\sim$16) and m is flash package count (state-of-the-art is 4$\sim$16). This results in a search space of approximately a few thousand choices, which is inexpensive to iterate through. Our estimations show this overhead to be approximately 1% (on an SSD with 72MHz microprocessor and 64 dual-die flash packages), which does not affect our conclusions. Specifically, while a read operation takes 180 μs (including NAND flash I/O bus activities), we estimate iterating through the search space to take 1.77 μs. Furthermore, modern SSDs are increasingly using multi-core processors, which are often underutilized and could be employed for queue management with even less impact on performance.

6.6.4 Time Series: In-Depth Analysis of a Database Trace

Next, we examine the differences in latencies between VAQ and PAQ2 in greater detail since that was the area of most uncertainty. For this, we focus on the sql3 trace, and present latencies incurred for its first 5000 I/O requests (about the first 10% of its execution) in Figure 6.13 for both VAQ and PAQ. It can be observed that, while VAQ demonstrates very consistent but moderately high latencies, PAQ is able to decrease latencies a great deal for a majority of the requests. However, there do exist some requests PAQ has delayed due to their conflicting nature; in a small subset of cases their response times exceed twice the latency of VAQ. We believe this characteristic is a natural side-effect of PAQ's performance-enhancing optimizations, but suggest that a balance can be struck and a bound on such spikes set by giving priority to those requests approaching a defined Quality-of-Service (QoS) threshold. Such examination is deferred for future work.

6.6.5 Sensitivity Analysis

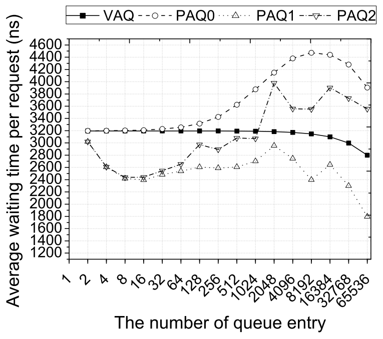
We wanted to expose how PAQ's performance might be impacted by varying current protocols and flash devices such as flash connected via not only our examined SATA protocol, but also SAS, PCI-Express and other emerging connecting protocols. Moreover, we wanted to determine whether PAQ would still offer benefits for future drives that will undoubtedly have higher component counts and queue sizes. Therefore, we performed an IOPS and waiting time sensitivity analysis for VAQ and our PAQ varieties on varying channel counts and queue sizes, two parameters we believe will fluctuate the most in upcoming devices and between different protocols respectively.

While the IOPS sensitivity analyses shown in Figure 6.14 demonstrates a predictable increase in IOPS as the numbers of components increases and as queue size increases, there are two less obvious take-aways: First, VAQ shows almost no improvements in performance as queue size increases, whereas all versions of PAQ demonstrate significant gains. Since VAQ does not utilize the queue for anything but FIFO storage, whereas PAQ utilizes the entire queue in performing its optimizations, this observation is expected. Second, while we saw little evidence that plane-packing by itself in the earlier results gave benefits, as we increase queue size in this analysis we see rapidly increasing performance for greater queue sizes. Such improvements are a result of an increased space for the optimization to search through and select packable blocks from.

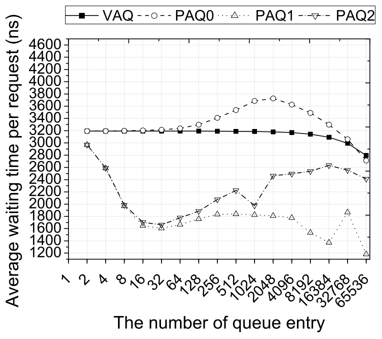
Figure 6.15 illustrates an *Average Waiting Time* (AWT) sensitivity test. AWT means the average time an incoming request must wait for a spot in the device-level queue, when that queue is full. Interestingly, in contrast to the IOPS sensitivity test, as we increase the number of queue entries, AWT of PAQ0 slightly increases, worse than VAQ at some points. This phenomenon occurs as a result of a delay experienced prior to PAQ0 taking advantage of multi-plane-mode operations. Even though multi-plane-mode operations can boost system throughput 1.53 times when compared to VAQ, it requires data movement between the HIL and the target NAND flash, which preempts channel I/O bus while transferring data and therefore imposes a delay. PAQ2 also employs the plane packing scheme, but, AWT of PAQ2 is better than VAQ's. This is because the benefits of I/O clumping covers the channel source preemption time of plane packing. As a result, PAQ2 improves both system throughput and average access waiting time.

6.7 Related Work

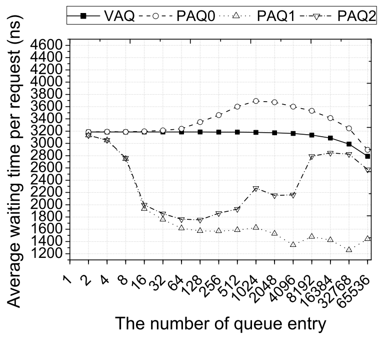
In [102], the authors attempt to uncover the specific resource contention that occurs in SSDs, the areas where parallelism is far below optimal, and present a dynamic request rescheduling scheme to supposedly improve results. However, in their work they fail to account for the reality that the addresses which correspond to the requests they "reschedule" are virtual. That is, while they propose that their scheme increases parallelism by avoiding conflict and capitalizing



(a) 4 Channels



(b) 8 Channels



(c) 16 Channels

Fig. 6.15 Waiting time sensitivity to varying queue and channel sizes.

on multi-plane mode in flash dies, they actually do not have knowledge of where those addresses point to since physical addresses are not available except beneath the FTL.

[102] actually is one paper in a larger body of works that have attempted to improve the read performance of SSDs without grappling with the key issue we point out in our work: the disparity between virtual and physical addresses. Another work suffering from similar oversimplification is [10], where the researchers observe suboptimal access trends in their traces and claim improvements are possible via the coalescing of read requests. The problem here is again that these traces are of LBAs, not of the physical addresses, and therefore coalescing may or may not end up improving performance internally within the SSD. Any improvements observed are likely the result of a clean-room testing environment leading to artificial alignment of LBAs and physical addresses; in a real installation performance improvements may or may not result from such coalescing.

It is also critical to emphasize major differences between flash-based SSDs and other flash mediums. As we discussed, SSDs are subjected to the very distinct nuances of operating through traditional storage interfaces such as SATA and SAS. Because of this, work aimed at optimizing flash-based storage connected via PCI buses or otherwise operating in a byte-addressable manner will likely not work when such optimizations are applied to flash-based SSDs. Therefore, while works such as [9] may appear to tackle a similar problem to what we target, assumptions they make, such as “the memory technology has performance…to that of DRAM and that it presents a DRAM-like interface,” pour a foundation that is entirely distinct from the base of assumptions we work from to improve flash-based SSDs.

In [13], the authors demonstrate that modern SSD performance is less dependent upon access patterns than interference between and within accesses and how the data is physically laid out on the dies. They also demonstrate that write access is faster than for reads in some cases and is far less dependent on access pattern and data layout than reads are. These realizations serve as an important foundation for our work. Using our PAQ scheme, we are able to decrease the impacts of the two major performance limiting factors they identify: interference and data layout.

In [53] and [20], the authors recognize the dire need to exploit parallelism in flash-based SSDs and propose three techniques for doing so over multiple independent channels: striping, interleaving and pipelining. Other, similar strategies have been presented to increase the parallelism of accesses over multiple NAND-flash packages, such as ganging [1], superpaging [20], and multi-plane mode, among others. While these schemes are capable of achieving significant performance improvements for higher parallelism than more serial alternatives, it is important to recognize that all such strategies only achieve parallelism for certain types of accesses. As such, while they work well for writes, since these may go anywhere and therefore achieving parallelism is trivial, these schemes often fail to improve the performance for reads since many reads do not perform accesses to strictly sequential physical addresses. We present and quantify the shortcomings of these advanced operations in our VAQ results, which utilizes 8 channels (superpage-based striping), 8 flash packages-per-channel (ganging), and 2 dies-per-package utilizing interleaved-die and multi-plane-mode operations, and yet fails to achieve peak performance. Therefore, the novelty of our work is not akin to the previously discussed strategies, which provide complex mechanisms to improve performance *only for certain access patterns*. PAQ's novelty lies mainly in reordering and reorganizing the accesses to enable mechanisms such as striping and ganging to perform their best.

While scheduling using physical addresses is novel in the context of SSDs, DRAM controllers already employ physical-address-based scheduling. Static as well as hardware-assisted dynamic-memory-access-reordering strategies [29, 71], and various other DRAM scheduling algorithms [33, 89] have been investigated in order to maximize DRAM bandwidth. Zuravleff and Robinson [104] also proposed a DRAM controller that improves data throughput in DRAM by reordering requests coming to a memory controller without changing the actual outcomes. Examples of QoS-aware controllers are fair-queuing memory systems [79], stall-time fair queuing [76], and start-time fair queuing [87]. These controllers were designed to be fair to applications that share a limited memory bandwidth and provide QoS guarantees if needed. With the increase in on-chip memory controllers, the idea of coordinated control of memory channels emerged. In [58], a method that achieves fairness by keeping track of attained service information for applications running simultaneously is proposed. By prioritizing threads with the least attained service, a fair memory scheduling scheme is obtained. In [59], fairness and throughput are considered simultaneously. The proposed thread cluster memory scheduling scheme isolates latency sensitive and bandwidth sensitive applications.

These DRAM scheduling works are motivated by parallel memory architecture, which has similarities to modern SSDs. However, there are three important differences: First, unlike DRAM, SSD schedulers must adhere to idiosyncrasies of NAND flash such as erase-before-write, endurance, asymmetric I/O speeds, and diverse NAND flash command protocols. Second, SSDs are connected through thin interfaces, which introduce more limitations in I/O scheduling than DRAM controllers, including different levels of queue management, I/O handshaking, host-device data movement for various I/O lengths, and I/O completion protocol. Lastly, an SSD scheduler should also be aware of underlying flash firmware features such as page- or block-level address remapping and garbage collection. These *differences in medium characteristics, interface nuances, and the responsibilities schedulers must carry out* drive research on scheduling mechanisms in each domain in separate directions.

6.8 Conclusion

As NAND flash-based storage devices, such as SSDs, becomes increasingly considered as caching mediums for their ability to serve random-reads at a high rate, extracting full performance for such workloads is only possible if barriers to parallelism are overcome. Our presented scheme to improve performance on random reads, PAQ, demonstrates the need for the queuing scheduler to have access to physical addresses of requests, and we discuss how *moving the QBM beneath the FTL* would achieve such. Further, we design and present a conflict classification methodology, *I/O clumping*, which PAQ utilizes to effect *efficient and low-contention I/O request scheduling.* Lastly, PAQ uses knowledge of the physical addresses of I/O requests to better enable multi-plane mode commands via our *plane-packing* optimization. In this work we implement PAQ in a cycle-accurate SSD simulator and evaluate its performance on a diverse set of traces taken from actual enterprise workloads. On those traces we demonstrate bandwidth and IOPS improvements exceeding 62% and decreases in latency as far as 41.6% on random reads when compared to the traditional queue scheduler, without slowing writes or sequential accesses.

Chapter Acknowledgements Chapter 6, in part, is a reprint of the material as it appears in ”Physically Addressed Queueing (PAQ): Improving Parallelism in Solid State Disks,” Myoungsoo Jung, Ellis Herbert Wilson III, Mahmut Kandemir, in Proceedings of the 39th International

Symposium on Computer Architecture (ISCA), 2012. The dissertation proposal author was the primary investigator and the first author of this paper.

Chapter 7

Garbage Collection Scheduling

7.1 Introduction

Over the past decade, different computing domains, ranging from high performance computing to enterprise server platforms to embedded systems, are adopting SSDs [8] [55], due to their technical merits such as good random access performance, low power consumption, higher robustness to vibrations and temperature, and higher read/write bandwidth than hard disks [10]. NAND flash capacity is increasing by two to four times every two years [66] and SSD prices are expected to continue to fall to the extent of becoming cheaper than high-speed hard disk [21], which can in turn enable widespread deployment in diverse computing domains.

Modern SSDs internally employ a flash translation layer (FTL), managing two intrinsic properties of NAND flash memory to emulate it as a block device: first, no write is allowed before erasing a block, called the *erase-before-write* property. Second, NAND flash makers adopt a write sequence in a block due to the page-level program disturbance behavior [74] [27], which has a deep relationship with modern NAND flash memory reliability and data integrity. In addition to the erase-before-write property, this *in-order-update* property in a block necessitates *out-of-place updates* for write operations. To enable such out-of-place updates in the SSD, FTL remaps the logical addresses that conventional block devices provide to the physical addresses presented by the NAND flash memory. In addition, the FTL employs a garbage collector, which reclaims the invalid pages, incurred during the out-of-place update process. At a high-level, the garbage collector relocates valid pages in certain blocks to new blocks, which are prepared in advance, and erases them in order to make rooms for new writes. This operation is referred to as *garbage collection* (GC).

The biggest problem with existing garbage collectors is that their worst-case latency can be as high as 64$\sim$128 times than that of normal write operations [52] [67]. Our own experiments show that GCs introduce numerous blocking I/Os, and once a GC operation begins, the response time of write operations on SSD increases substantially. Further, GC overheads significantly reduce available bandwidth in most recent commercial SSDs. Unfortunately, this interaction between the GC and writes introduces significant performance variations/degradations during I/O, which may not be acceptable in many I/O-intensive computing environments.

Motivated by this, most current FTLs optimize mapping policies to minimize the number of GC invocations and hide their undesired latency. For example, existing buffer management schemes are specialized to reduce the number of writes to NAND flash. Also, some SSDs employ partial block cleaning techniques [11] [15] that attempt to provide stable GC performance by balancing the number pages/blocks between production and consumption of them using an extra non-volatile buffer. However, there is yet another dimension to avoiding GC overheads. Specifically, *the presence of idle I/O times in workloads can be exploited by shifting garbage collections from busy periods to other periods where they can be accommodated with minimum performance penalty*.

In this Chapter, we propose a novel GC strategy, an approach that removes GC overheads and provides stable I/O performance in SSDs during the I/O congestion periods. Our proposed GC strategy consists of two components, called *Advanced Garbage Collection* (AGC) and *Delayed Garbage Collection* (DGC). More specifically, AGC tries to secure free blocks and remove on-demand GCs from the critical path *in advance*, so that users do not experience GC-induced latencies during the I/O-intensive periods, whereas DGC handles the collections that AGC could not handle, by delaying them to future idle periods. Since our approach mainly reschedules garbage collections, it can work with any existing FTL.

Shifting GC operations however can increase program/erase (PE) cycles, which makes the life time of SSDs shorter. For example, if a garbage collector heedlessly reclaim blocks, which have the potential to be further utilized or used for new writes, it can introduce unnecessary PE cycles in relocating valid pages within them. To prevent this problem, we propose two different implementations for AGC, called *look-ahead garbage collection* and *proactive block compaction*, based on the duration of the idle period under consideration and the style of GC detection. Specifically, the look-ahead GC utilizes short idle periods and reclaims block based on the online information extracted from a device-level queue, whereas the proactive block compaction targets long idle periods and perform GCs only related to fully utilized blocks.

As shown in Figure 7.1, the main **goal** behind our strategies is to perform as many GCs as possible in the *idle periods*. Our **contributions** in this Chapter can be summarized as follows:

• *Eliminating GC overheads:* When using our garbage collection strategies, applications do not experience GC overheads. This is because our strategies successfully migrate on-demand GCs from busy periods to idle periods. Experimental results show that our proposed GC strategies result in stable I/O performance under various types of workloads.

• *Avoiding additional GC operations:* The proposed schemes (AGC and DGC), when applied together, do not increase the original number of GC operations. They only reschedule the GC operations that would be invoked soon by speculating their GC activities and identifying appropriate idle periods based on their durations (short or long). If the frequency with which

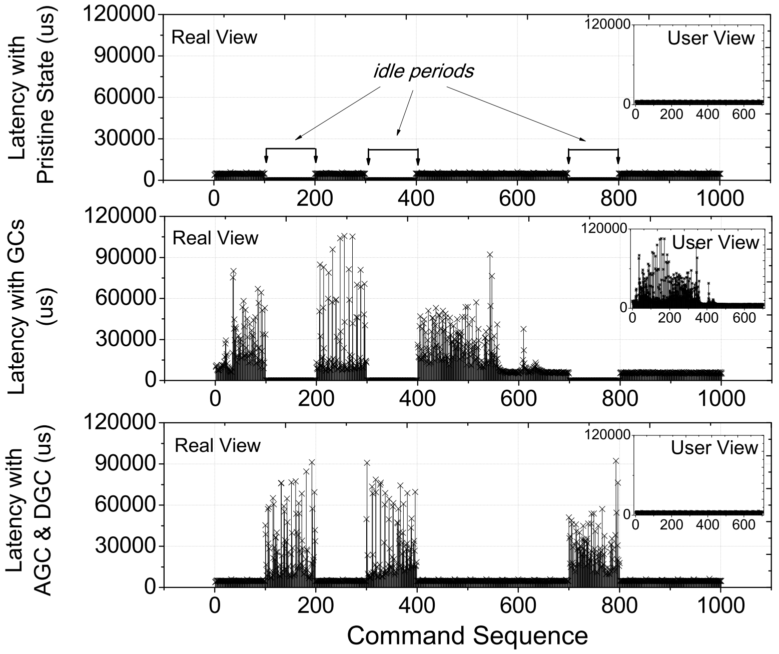


Fig. 7.1 Overview and comparison of SSD latencies with/without our proposed GC strategies (AGC and DGC), tested by random write pattern with 2048KB request size. Note that AGC and DGC shift GC overheads to idle periods (as shown in the real view), thereby providing stable I/O performance like a pristine state (as shown in the user view).

idleness occurs is not high enough, then the GC invocations are postponed to future idle periods without affecting latency of I/O operations. As a result, we do not incur any additional GC operations.

• *Compatibility with underlying FTL schemes:* Most optimized garbage collectors proposed in the literature need additional non-volatile (NV) blocks on the SSD and/or require customized FTLs for successful execution. In contrast, our proposed schemes do *not* require any extra NV blocks or modifications to the existing data structures, and can therefore work with any FTL.

To test the effectiveness of our GC schemes, we implemented them in a simulator that models bus-level transactions and collected statistics using a variety of workloads. Our experimental results show that the proposed schemes reduce GC overheads (without causing additional write/erase operations) between 66.7% and 98.2%, in terms of the worst-case response time. Further, our schemes prepare free blocks in advance to help prevent the block thrashing problem. Consequently, they reduce the number of block erase operations by 16.6%, compared to a conventional FTL.

7.2 Background and Related work

7.2.1 Flash Translation Layer

The NAND-based flash consists of physical *blocks*; a physical block is the erase unit and is composed of several *pages*, which are the read/write units in the NAND flash. One of the drawbacks of the NAND flash is that a page needs to be updated in-place within the block. In addition, writes to a formerly written page are not allowed before erasing the entire block corresponding to it. Since a block is much larger than a page and an erase operation is more expensive than a write operation, the NAND flash alone is not sufficient to build an SSD. Therefore, a *Flash Translation Layer* (FTL) is required within the SSD to prepare physical blocks ahead of time. Whenever an SSD receives write requests, it forwards them to a temporal block called an *update block*. The FTL then serves requests by physically (in-place) writing them into a block. This allows logical out-of-order update by mapping addresses between the in-place and out-of-place update sequences.

The FTL also hides the latency of block erase and unnecessary read/write operations in copying valid pages that are live in a block [52] [57]. Similarly, to provide data consistency and coherence between the original block (also called the *data block*) and the update block, the FTL internally maintains mapping information and address translations. In this way, by internally managing the flash specific characteristics, the FTL provides compatibility with commodity storage systems. Typically, based on the number of the data block(s) and update block(s) in a logical block, FTLs are classified into three types. Block-mapping FTL manages a logical block by combining one data and one update block (1:1 mapping). Hybrid mapping FTLs manage a logical block by composing n data block(s) and m update block(s) ($n : m$ mapping). Finally, pure-page mapping FTLs leverage only update blocks for serving I/Os, and can allocate them in any physical page location.

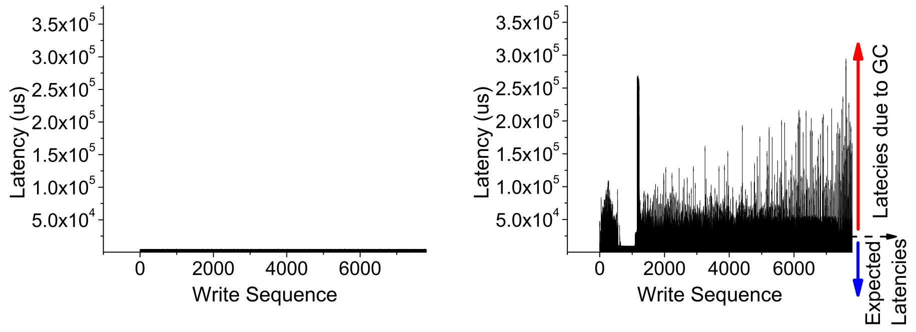
7.2.2 Garbage Collection

If the FTL does not have enough free pages in its update block, it has to perform GC in an attempt to reclaim available blocks to which write request, can be forwarded. This type of GC is referred to as *update block reclaiming GC*. Similarly, in cases where the FTL has insufficient free blocks, it should secure free blocks by evicting some other logical blocks, called *free block reclaiming GC*. These processes require migrating all valid pages from the update and original blocks to a new free block (called *page migration*) and erasing these two blocks. Thus, *the GC latencies are typically much larger than that of normal I/O operations*. In addition, the FTL carries out these GC operations during runtime on a need-basis, meaning that the collections are postponed as long as the SSD can accept new data and are only performed when required. The reason why GCs are executed on demand is that a block erase operation, which is part of the garbage collection activity, can significantly affect the SSD's lifetime and reliability [52]. For example, if a garbage collector heedlessly reclaim blocks, which have the potential to be further utilized or used for new writes, it can introduce unnecessary program/erase (P/E) cycles for relocating valid pages within them. Due to this property, GC latencies typically piggyback on ordinary I/O requests, leading potentially to very high I/O latencies. Several FTL based studies [67] [52] attempted to reduce GC overheads and hide their latencies. Other approaches like the real-time GC [11] and the partial block cleaning [15] aimed to provide stable GC performance by balancing the number pages/blocks between the production and consumption of them using an extra non-volatile buffer.

7.3 Impact of Garbage Collection in Commercial SSDs

To measure the impact of GCs in state-of-the-art SSDs, we evaluated their latencies and bandwidth with/without GCs.

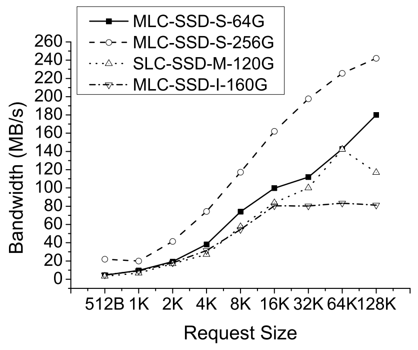
Latency impact: Figures 7.2(a) and 7.2(b) plot normal latencies and extra latencies due to GCs, respectively. In this empirical test, we used a 256GB MLC-based SSD which employs two 128MB internal DRAM buffers and measure latencies of individual I/O operations using ULINK's DriveMaster [99]. The DriverMaster is a commercial tool that captures detailed storage-level latencies and tests SSDs in a physical level. We wrote data with 1MB transfer size into the SSD using a random pattern. While a pristine SSD was used for the normal latency test, we later filled the SSD completely and introduced a one hour period before evaluating the GC latencies. As illustrated in Figure 7.2, GCs introduce numerous blocking I/Os, and once a GC operation begins, the response time for write operations increases substantially. Further, irrespective of the large amount of idleness that we artificially introduced, high latencies of GC are observed from the beginning of the GC latency test.



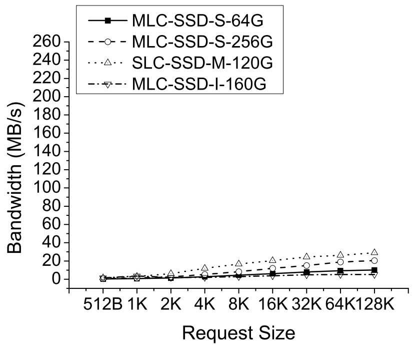
(a) Normal Latencies (b) Latencies with GCs

Fig. 7.2 Latency comparison for a random write access pattern with 1MB request size using a real MLC-based SSD.

Bandwidth impact: From a system designer viewpoint, throughput might be a more important performance metric. In this test, we measured performance with/without GCs of four commercial SSDs (three 64GB, 256GB and 160GB MLC-based SSDs and one 120GB SLC-based SSD) using Intel Iometer [35]. Figures 7.3(a) and 7.3(b) plot bandwidth with the pristine state and bandwidth with fragmented state, respectively. To make an SSD fragmented, we first wrote 4KB data in random order and fully utilized its storage space. Similar to the latency impact test, we artificially introduced a one hour idle time before evaluating this bandwidth impact test. As shown in Figure 7.3, GC overheads significantly reduce available bandwidth in all four commercial SSDs tested, regardless of the idle time introduced.



(a) Normal Bandwidth (Pristine)



(b) Bandwidth with GCs (Fragmented)

Fig. 7.3 System throughput for four state-of-the-art SSDs (different vendors and NAND types). Note that all SSDs tested suffer from significant performance degradation once garbage collections begin.

7.4 High Level View of GC Scheduling

To avoid performance degradation and variations caused by GCs, we propose novel garbage collection strategies. Unlike previous GC strategies that reduce the number of GC invocations or GC overheads at runtime, our proposed GC strategies fully utilize device-level idle times, which are invisible to the user, to perform GC activities. To efficiently exploit such idle times, we classify them into two groups based on their lengths. Using our idle period classification, we then invoke different types of GCs to ensure that the user does not experience long GC-induced latencies. Our approach allows the other components of FTL to work without any modification, making our approach highly portable.

7.4.1 Idle Period Classification

Short idle periods: Several applications exhibit short idle intervals interleaved with parallel I/O requests in a device-level command queue [38], which allows a storage system to determine actual data transfer times and implement out-of-order execution of I/O commands. To enable this, most host interface protocols bring I/O commands, along with preinformation including request type, addresses and request size, to the storage system before the actual data transfer begins.

To measure how many commands with their preinformation are available at a given time and the duration of idle periods, we executed Intel Iometer workloads [35] and employed a 265GB MLC-based SSD that used in Figure 2. The LeCroy protocol analyzer [62] is used for analyzing the SATA protocol at the physical layer. We observed that 3-17 commands are delivered to the device-level command queue in parallel before the actual data communication starts, and the storage-level idle times experienced by the I/O requests vary between 1.8 μs and 15.2 ms, based on the operation type and transfer size.

This storage-level short idle periods that we measured can be detected by looking through the I/O commands with their preinformation. Specifically, one can preview I/O commands in the queue before they get executed, and identify the short idle intervals between successive I/O commands. Even though this interval is short, one benefit gained from utilizing these short idle intervals is that it allows one to investigate a request through preinformation and accurately predict what will happen to the request during the idle time. Each short idle period can be expressed as follows:

$$T_{short-idle} = t_{start_{i+1}} - (t_{start_i} + t_{exe_i} * l_i) \quad : \forall i, 1 < i \leq n, \tag{7.1}$$

where $i+1$ denotes the index of the I/O command following the i^{th} I/O command in the queue, t_{start_i} denotes actual transfer start time, t_{exe_i} is the execution time based on a page, and l_i is the page length of I/O command *i*. Clearly, short idle periods exist only if $T_{short-idle}$ is larger than zero and there are I/O commands sitting in the queue (i.e., at least two commands). Here, *n* depends on the queue size accommodated by the host interface nuance. For example, NCQ [38] provides 32 entries, whereas TCQ [96] typically provides 256 entries. AGC exploits just two entries for previewing the I/O commands.

Long idle periods: We also observed that many applications exhibit relatively long idle periods with no enqueued I/O commands. We classify an idle period as a long idle period if its length is larger than a certain threshold [16] [72] [26]. The fraction of I/O instructions that experience these long idle periods ranges between 38% and 83% under various workloads tested [78] [88] when the threshold is set to 1 sec. Note that, to detect these idle periods, we cannot take advantage of the device-level command queue and preinformation since it is empty most of the time. Consequently, long idle periods should be handled differently.

Depending on whether idle periods are short, long or none, our proposed strategies schedule GC operations and secure free blocks differently.

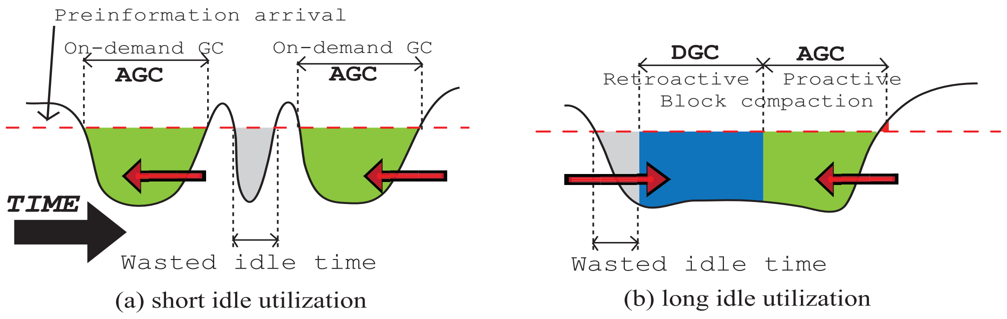


Fig. 7.4 A high-level view of our proposed GC strategy and idle time utilization.

7.4.2 Shifting Garbage Collection Overheads to Idle Periods

We start by observing that scheduling a GC on an arbitrary idle period can introduce extra block reclaimings (P/E cycles) and reduce opportunities for block reuse. This can in turn potentially shorten SSD lifetime and affect its reliability. Therefore, in our proposed schemes, we migrate the GC operations to *carefully-chosen* idle periods without increasing the original number of GC operations. We also minimize the overheads incurred by on-demand GC invocations by securing the available free blocks as much as possible in advance during the idle periods.

We explore two different strategies for shifting GC overheads, depending on the amount of idleness and on-demand GC needs, as shown in Figure 7.4. First, short idle periods are mainly exploited by shifting on-demand GCs that will be invoked during busy periods (Figure 7.4a). In this case, the garbage collector monitors upcoming device-level I/O tasks to determine when a collection will be needed and performs the necessary tasks *proactively*. If there is no on-demand GC need, the garbage collector performs block compactions to reclaim fully-occupied blocks thereby retrieving free blocks in advance (Figure 7.4b). We refer to this strategy as the *Advanced GC (AGC)*. Second, if the amount of short idleness is not sufficient to avoid on-demand GC invocations at a certain point, our proposed GC strategies prevent them from being invoked on the critical path by delaying the GC execution (Figure 7.4b); we refer to this strategy as the *Delayed GC (DGC)*. In other words, AGC shifts GC activities to idle periods in advance, whereas DGC handles the on-demand collections that AGC could not handle, by delaying the GC invocations to future idle periods.

These different GC strategies based on the type of idle periods and GC needs allow our strategies to shift GCs from busy periods to idle periods, as illustrated in Figure 7.1. At the same time, they help us minimize the potential side effects on SSD reliability and eliminate the extra storage space requirement in the SSD for the operation of the proposed schemes.

7.5 Implementation of our GC strategies

Recall that we quantified the impact of garbage collection on commercial SSDs in Section 7.3. To alleviate the overheads caused by garbage collections, we classified the types of idle periods in Section 7.4 and presented a high-level view of our proposed approach. We next describe the technical details of AGC and DGC in Sections 7.5.1 and 7.5.2, respectively. Section 7.5.3 discusses how AGC and DGC works together.

7.5.1 Details of Advanced GC Strategy (AGC)

AGC tries to remove the on-demand GCs from the critical path and secure free blocks in advance so that users do not experience long GC-induced latencies during the I/O congestion periods. Depending on the type of the idle period we are dealing with, one can implement AGC in two different ways. The look-ahead garbage collection (Section 7.5.1.1) is a type of AGC that targets on-demand collections by utilizing short idle periods, whereas the proactive block compaction (Section 7.5.1.2) secures free blocks by utilizing long idle periods.

7.5.1.1 Look-ahead Garbage Collection

To shift GC invocations to earlier idle periods, this scheme exploits the device-level command queue and short idle periods. It starts by calculating the number of GC operations that can be executed in short idle periods. In this step, the look-ahead GC checks the queue entries and extracts I/O request information such as the length of I/O request and the Logical Sector Number (LSN) and associated Logical Block Number (LBN). It then finds the Physical Block Numbers (PBNs) corresponding to the LBN by looking up the mapping table of the underlying FTL. The look-ahead garbage collector then checks whether the available space, especially the number of free pages, is sufficient to service the I/O request of the specified size (length). If not, an update block reclaiming GC is required.

Once the need for GC is identified, our scheme next calculates a GC latency in order to accurately perform on-demand GC in advance. Let κ denote the number of physical blocks per logical block (e.g., in a block mapping scheme, the value of κ is one. On the other hand, if the system employs a 2:8 hybrid mapping scheme, κ can be up to ten). Further, let t_{load}, t_{write}, and t_{erase} denote execution latencies for page load (read), page write and block erase operation, respectively, and let t'_{write} represent the time for writing metadata to confirm the fact that a certain physical block was erased after GC (this helps to ensure mapping consistency in the FTL). Since the look-ahead GC knows PBN(s) for the logical block and has all the relevant mapping information, it can determine the number of valid pages for the PBN(s); say, n_{page}^{valid}. In this way, for each I/O command i that is involved in the GC, its GC latency (T_{gc_i}) can be calculated using the following expression:

$$T_{gc_i} = \underbrace{(t_{load} + t_{write}) * n_{page}^{valid}}_{page\ migration} + \overbrace{(t_{erase} + t'_{write}) * \kappa}^{block\ cleaning}. \tag{7.2}$$

This expression captures the page migration latency for each valid page from the update/data block to a free block, as well as the block cleaning latency for these blocks. Typically, t'_{write}

is approximately the same as the latency it takes to write one page (t_{write}). This is because the metadata is designed to fit in a single page to reduce the overhead of storing the metadata itself. The total amount of time taken by the look-ahead GC to perform collections over n blocks is given by ($\sum_{i=1}^{n} T_{gc_i}$). Using the GC latencies of individual blocks, we determine the number of blocks (n) that can be reclaimed at runtime, under the constraint that the total GC time for the determined number of blocks is less than or equal to the short idle time given by Equation (7.1). Once AGC determines the number of blocks, n, to be claimed, it performs look-ahead GCs for these n blocks in advance.

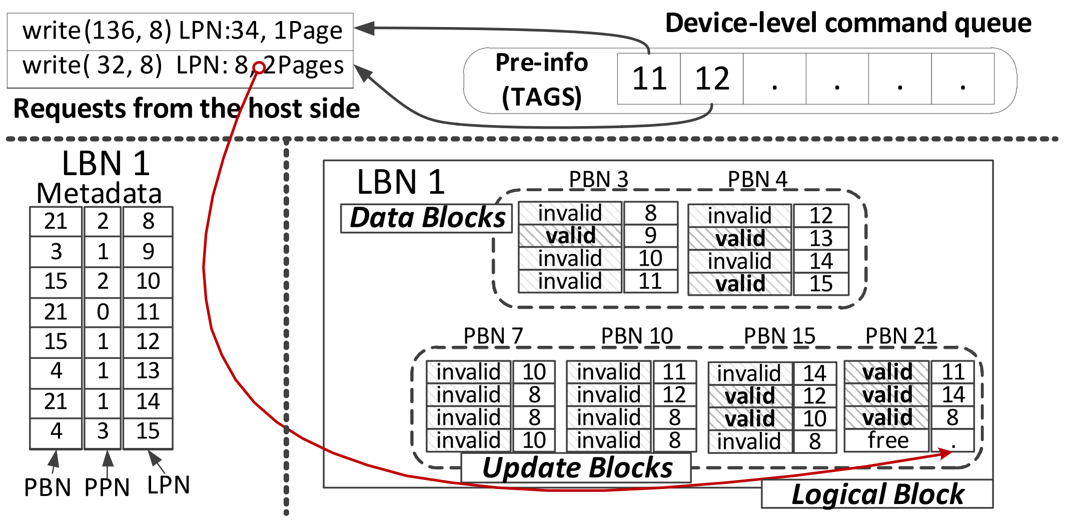


Fig. 7.5 An example of the look-ahead GC with a hybrid mapping scheme. By inquiring the mapping information, our AGC scheme figures out that the GC for LBN 1 will be invoked soon.

For instance, in Figure 7.5, the look-ahead GC identifies the request with an LSN of 32 and I/O size of eight sectors. Since the logical block corresponding to that request has only one free page, our scheme executes the GC operation in the short idle time. In addition to providing stable and better SSD performance, this implementation performs GCs *only when* an on-demand

GC is about to occur and the short idle periods are suitable to perform GC. Therefore, it ensures a similar level of reliability compared to a standard FTL.

7.5.1.2 Proactive Block Compaction

In order to exploit long idle times that could not be exploited by the look-ahead GC, we propose a proactive block compaction mechanism strategy. In this strategy, we detect the blocks (in a logical block) that are fully occupied with valid/invalid pages, and compact them in advance during the long idle periods. Compacting blocks involves enforcing all valid pages from the fully-occupied physical block to a new, clean block, and removing the invalid pages in the former by erasing them. Consider as an example Figure 7.6 where we have two fully-occupied blocks, namely, LBN 5 and LBN 32768. AGC can compact these two blocks in advance during long idle periods. In order to avoid the scanning penalty required to identify the fully-occupied blocks, we add the LBN of the fully-occupied block to the AGC job list, while the FTL is serving the I/O requests so that the blocks can be compacted *proactively* without scanning the entire storage address space.

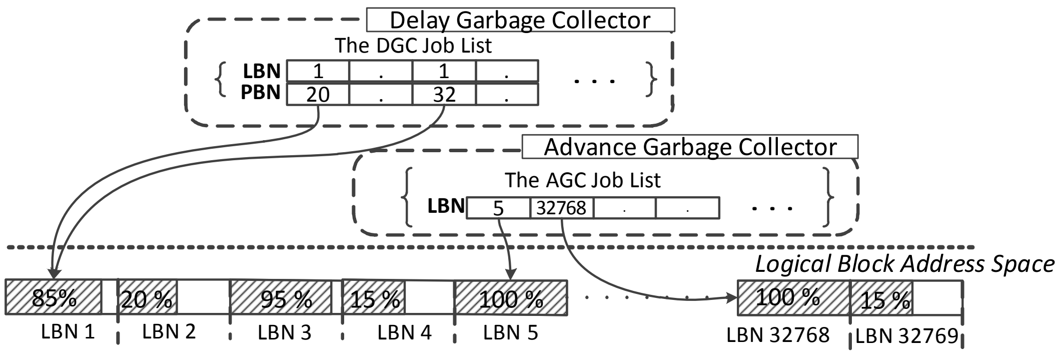


Fig. 7.6 Job lists for AGC and DGC.

Even though proactive block compaction is relatively simple, it can be very effective in practice as far as enhancing idle time utilization and securing free blocks are concerned. It should be noted that the proactive block compaction mechanism is executed *only if* the number of free blocks is less than the free block threshold (i.e., an on-demand GC would be invoked very soon). Therefore, similar to the look-ahead GC mechanism discussed earlier, this proactive block compaction mechanism also minimizes the number of unnecessary erase operations, which in turn helps to improve SSD endurance and reliability.

7.5.1.3 Incremental Garbage Collection

One concern regarding AGC is that it could lead to undesired performance degradations and prevent the GC latencies from being hidden, if idle periods are too short or do not occur frequently enough. To avoid this, our implementation of AGC splits GC activities into smaller ones delimited by *checkpoints*, and performs the GCs step-by-step based on the checkpoints. As illustrated in Figure 7.7, the checkpoints are inserted at the end of every NAND I/O completion point and constitute the boundaries across the neighboring GC steps. Inspired by the checkpointing strategy described by [39], AGC incrementally performs a given GC operation one step at a time; this is referred to as the *Incremental Garbage Collection* (*Incremental GC*) in the remainder of this Chapte.

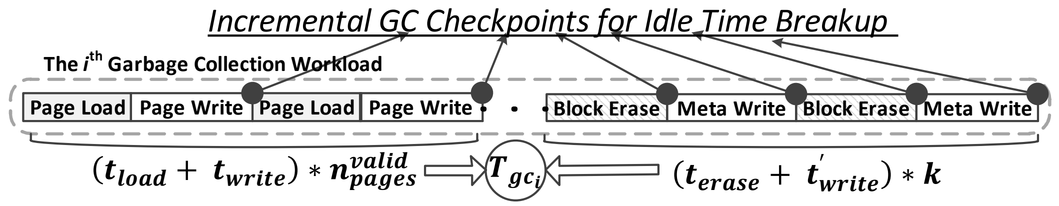

Incremental GC Checkpoints for Idle Time Breakup
The i^{th} Garbage Collection Workload
Page Load
Page Write
Page Load
Page Write
Block Erase
Meta Write
Block Erase
Meta Write
$(t_{load} + t_{write}) * n^{valid}_{pages}$
T_{gc_i}
$(t_{erase} + t'_{write}) * k$

Fig. 7.7 Checkpointing for incremental GC. At each checkpoint, by checking the device-level queue, the garbage collector can decide whether it can perform further collections or not.

Whenever AGC reaches a checkpoint, the incremental GC determines whether further collections can be performed or not by checking the device-level queue. If there are no I/O requests until the next checkpoint, it goes ahead and executes the next step of the GC operation. The same procedure is repeated as long as there are no I/O requests. If on the other hand AGC detects an I/O request at a particular checkpoint, it postpones the remaining GC steps to the next idle period. To do this, it marks the current GC job status and inserts this marked status information into another job list that is managed by DGC (this will be revisited in Section 7.5.2.1). This incremental GC operation allows AGC to avoid the potential drawbacks of very short idle periods, and smoothly pass the control of GC operations to DGC. As a result, the SSD is able to serve the bursty I/O requests that can potentially create very short idle periods.

7.5.2 Details of Delayed GC Strategy (DGC)

Even though idle periods are typically long enough [72] [78] [77] for AGC to prepare available free blocks ahead of time and execute GC in advance, in cases where idleness does not occur frequently, AGC may not be very successful. The main goal behind our *Delayed Garbage Collection Strategy* (DGC) is to address this situation by delaying GC invocations. Its operation can be divided into two steps as explained below.

7.5.2.1 Update Block Replacement

As stated earlier, the main reason why GCs degrade system performance is *page migrations*. To avoid this degradation, DGC defers the page migration activity to *future* idle periods. Whenever an on-demand GC occurs in a busy period, DGC allocates free block(s) as update block(s). Normally, commodity FTLs migrate valid pages from the update and data blocks to an allocated free block. In contrast, DGC skips this process and serves the urgent I/O requests. Rather than migrating pages, DGC adds the LBN and PBN(s) corresponding to the migration into a job list it maintains (called the *DGC job list*). This delayed page migration activity is later resumed in a future idle period by the DGC's retroactive block compaction (see Section 7.5.2.2).

The free block allocation carried out by DGC is similar to what a standard FTL would do during GC. The only difference is that DGC allocates the block as an update block (not a free block). Since the FTL already has an update block (but it is garbage), DGC intercepts the update block information and replaces the PBN of the update block with the allocated free block's PBN in the FTL mapping table. In this way, the FTL treats the allocated free block (called the delay block) as an update block, and is not required to manage the block mapping information. It explicitly manages replacing/updating a block for preserving consistency during information mapping. Further, DGC maintains this information using the DGC job list and hides this information from the FTL until the page migration process completes. In the meantime, if there is an I/O request, the FTL serves that request based on the available mapping information. This replacement and interception procedure is called the *Update Block Replacement*.

The main advantage of the update block replacement is that, as soon as the SSD receives an I/O request, it can serve the request without migrating the valid pages, even when AGC could not handle on-demand GCs in advance. Another benefit is that DGC does not require any additional NV memory space for delaying on-demand GCs, which is essential resources of prior works [15] [51]. This is because it replaces the update block with free blocks that belong to the FTL address space. Note that the mappings employed by the FTL and DGC do *not* interfere with each other, and this allows DGC to work with various other mapping schemes used in current FTLs.

Algorithm 3
IssueCommands(IoRequestPacket irp) of our proposed AGC+DGC algorithm. Note that the SSD just forwards I/O requests to the underlying FTL without performing any GC during the busy periods.

```
if irp.command != empty then
    if ftl.checkOnDemandGc(irp) then
        {delay the on-demand GC}
        UpdateBlockReplacement(irp)
        insertEntry(DgcJobList, irp.getLbn())
    end if
    {call the FTL service}
    ftl.ServeIo(irp.command, irp.lsn, irp.sectors)
else
    targetLbn := getDgcLbn(DgcJobList) {DGC}
    if targetLbn != nullblock then
        consumed = RetroactiveBlockCompaction(targetLbn)
    end if
    {AGC}
    idleType := checkIdleType(CommandQueue, consumed)
    if idleType = short then
        {Calculate GC latency using Equation 1 & 2}
        idletimes := getIdleTime(CommandQueue, consumed)
        requiredTimes := speculateExecutionTime()
        while idletimes $\geq$ requiredTimes do
            LookaheadGc(irp)
        end while
    else if idleType = long then
        ProactiveBlockCompaction(irp)
    end if
end if
```

7.5.2.2 Retroactive Block Compaction

When there is no I/O congestion, DGC performs page migrations and returns the relevant delay block and update/replace block to the free block space. The blocks returned DGC can be recycled as normal free blocks. To return a block, DGC first extracts the LBN and PBN for a replace/update block from the DGC job list. It then queries the PBN for the data and delay blocks by using FTL's block-level mapping table. That is, it looks up the mapping table entry for the LBN extracted from the DGC job list and gets the corresponding PBN from the table.

Once DGC collects all PBN(s) for the blocks related to the delayed logical block, it retroactively compacts the blocks and returns them to the original state (i.e., as free blocks). While compacting, DGC migrates valid pages deferred from all PBNs for each delay, replace/update, and data block. This page migration is simply executed by reading and writing pages in an ascending order. We want to point out that the number of pages requiring migration is less than or equal to the number of pages in a logical block, independent of the number of delay and data blocks involved. Thus, the migration cost of DGC is the same as that of original GC. During busy periods, DGC preferentially reads and writes pages to the delay block rather than the replace/update block to guarantee data consistency. The reason behind this order is that the delay block contains the latest data when compared to the data in the replace/update block(s). This also helps DGC to improve block utilization and reduce the amount of I/O activity while performing the collections since the replace/update block(s) can be erased without any read or write operation in the ideal case.

7.5.3 Putting the Two Schemes Together

When our two schemes, AGC and DGC, are applied together we expect that most GCs are invoked by AGC; DGC will be invoked only if the idleness at hand is insufficient or the number of free blocks secured by AGC is not enough. In fact, we observed during our experiments that the fraction of idle periods DGC handles accounts for at most 20%, and AGC manages the rest. Algorithm 3 describes the steps involved in integrating DGC and AGC (called the *AGC+DGC scheme*). In summary, if an I/O request triggers an on-demand GC, DGC delays page migration to future idle periods using the *update block replacement* mechanism. During idle periods, DGC first performs *retroactive block compaction* **only if** a delayed GC block exists. And, AGC is invoked based on the type of idle period at hand. Specifically, if the idle period is short (just enough to perform the required GC), *look-ahead garbage collection* is invoked. Finally, *proactive block compaction* is invoked when the idle period is long. In each implementation, GC is performed incrementally, as explained in Section 7.5.1.3.

7.6 Experimental Evaluation

To evaluate the effectiveness of our AGC and DGC, we implemented them in a simulator-based platform. The original event-driven simulator [51] was modified to model multiple channels and ways with a bus transaction-level clock accuracy.

SSD configuration. We implemented two different SSD-based disk arrays;

• **6SSDs-RAID**: the first disk array was setup based on the original MSN file server storage configuration [78], which consists of 6 disks (Disk0 $\sim$ Disk5). In this default array, we introduced six of 64GB SSDs and each SSD, which replaces each disk of MSN storage server, has 4 channels and 4 ways architecture. Further, we categorize this SSD array based on each disk of write-intensity.

- *6SSDs-RAID-LO* is the group of SSD0, 1, 2, and 3 with low I/O intensive workloads of which the fraction of write amount is under 20% of total I/Os.
- *6SSDs-RAID-HI* is another SSD group, consisting SSD4 and 5 with high I/O intensive workloads of which the write fraction of total I/Os is 80%.

• **3SSDs-RAID**: Another disk array leverages three SSDs, in which each individual SSD composes of 8 channels and 8 ways (128GB). This disk array was configured to measure performance impacts on a different SSD configuration. In this 3SSDs-RAID, disk0 and disk1 (of the MSN server) are replaced by SSD0, disk2 and disk5 are replaced by SSD1, and disk3 and disk4 are replaced by SSD2.

Both SSD arrays in RAID-0 configuration are viewed by the OS as a single device. Even though we model a Samsung K9KGA0B0M MLC NAND flash[1] [91] in our simulations, our proposed GC strategies can be applied to other NAND flash device models as well.

FTL implementation. We implemented a log-structured FTL (*L-FTL*) and a 2:8 hybrid mapped FTL (*H-FTL*) on the SSD-based disk array models [57] [52]. We also implemented a partial GC scheme based FTL (*P-FTL*) [15] [51] [11]. After some initial experiments, the percentage of free blocks and GC threshold are set to 3% and 1%, respectively, of the total SSD address space[2]. We also introduced a 14 GB extra space to P-FTL for each SSD in the 6SSDs-RAID and

[1]This has 128 pages per a block. Based on a 4 KB page size, read, write and block erase latencies are 183.2 us, 860.36 us, and 2 ms, respectively.

[2]Some industries employ even higher GC thresholds with more free blocks, which renders SSDs expensive. Since there is a variety of configurations for GC threshold, we choose a lower bound value for our evaluation. We believe that alleviating GC overheads in our configuration (more complex) can be reduced the GC problem in such expensive SSDs configuration.

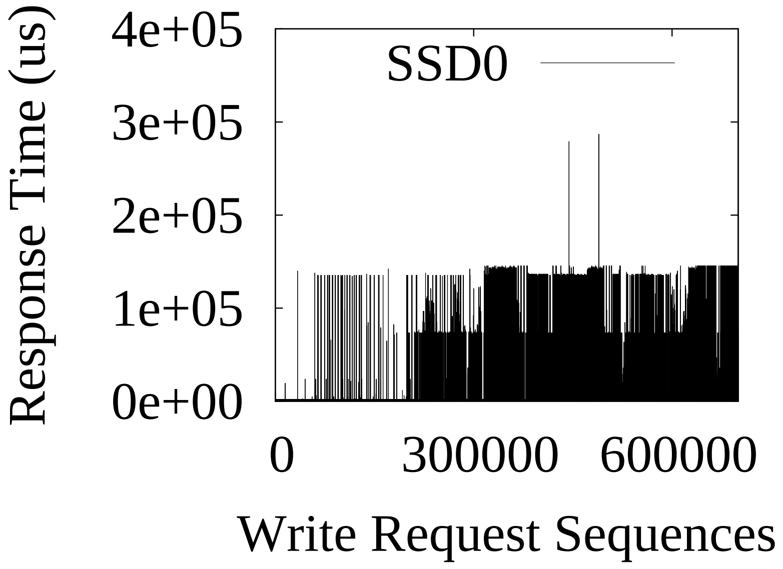
	The number of I/O requests	Total amount of requests (KB)	Total amount of writes (KB)	Idle Periods (%)
Disk0	1,509,397	32,490,240	3,051,918	38.6
Disk1	2,221,728	35,383,340	17,722,159	81.3
Disk2	500	1,958	1,958	56.6
Disk3	4,352	2,392,445	2,387,767	83.0
Disk4	12,627,396	117,607,983	24,835,283	42.4
Disk5	12,981,710	130,033,924	31,777,436	41.3
Total	29,345,083	317,909,889	79,776,520	64.1

Table 7.1 Important characteristics of our traces. The last column gives % of I/O requests containing sufficiently long idle ($>$ 1 sec) periods.

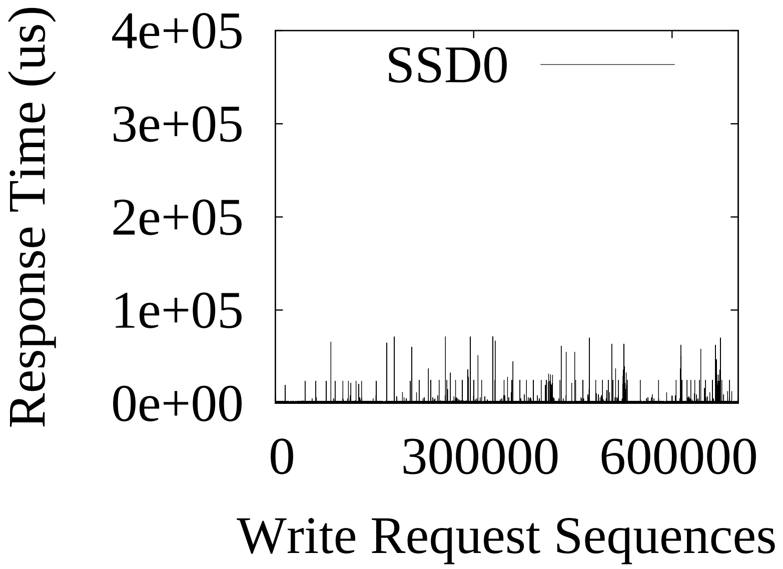
28GB extra spaces to it for each SSD in the 3SSDs-RAID based on the results from the write buffer analysis [15]; these extra spaces are used as the non-volatile write buffer in an attempt to serve urgent I/Os and provide real-time support, and managed through the page-level mapping scheme in P-FTL, instead of employing a block-level mapping scheme.

Workloads. Enterprise traces tested are collected from the MSN file storage server over 5 days [78] [88]. The total I/O traffic studied was up to 1.8TB. Important characteristics of our traces are given in Table 7.1. In the traces used, 34.6% of idle intervals were long (larger than 1 sec) and less than 29.5% were short, and 35.9% of the requests were back-to-back with no idle time in between.

It should be mentioned that each I/O request of any trace we simulate has a time stamp associated with it, and all the different approaches we tested (for reducing GC overheads) take advantage of scheduling the I/O requests based on the corresponding time stamps (using NCQ). Our bus-transaction level simulator extracts access time information from the I/O commands, using which we synchronize the global timer of the simulator and check the I/O latencies at the end of every I/O completion. This enables us to accurately record idle/busy periods on the SSDs.



(a) Baseline GC

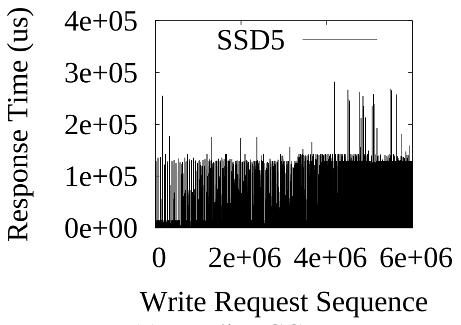


(b) AGC only

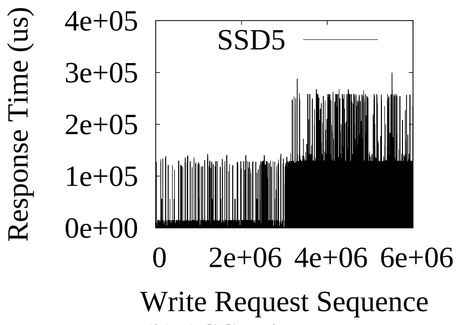
Fig. 7.8 Performance of AGC with relatively low I/O intensive workloads (SSD 0 of 6SSDs-RAID).

7.6.1 Performance Comparison

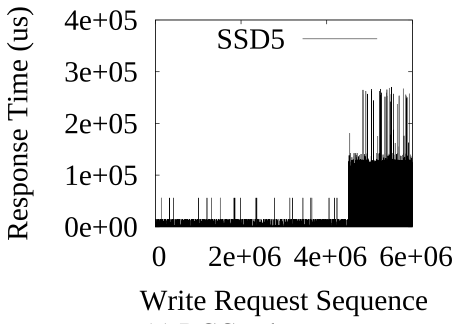
We first evaluate the performance of our two GC strategies (AGC and DGC) using 6SSDs-RAID in isolation. Figures 7.8 and 7.9 plot the response times of SSD0 of 6SSDs-RAID-LO and SSD5 of 6SSDs-RAID-HI, respectively. As illustrated in Figure 7.8, AGC alone successfully hides almost all on-demand GCs in SSD0, leaving nothing for DGC. We see from Figure 7.9(b), however, that AGC alone is not very successful with the high I/O intensive workloads. During the high write-intensive periods, a few on-demand GCs are invoked due to the very small amount of short idle periods in SSD5. Even though the number of these on-demand GC invocations is small, the FTL uses up available free blocks for new requests, which introduces more on-demand GC invocation. In a worst-case scenario, AGC suffers from both increased amount of GC invocations and short idle periods as the execution progresses. This is the reason why AGC requires DGC to handle such on-demand GCs. One can see from Figure 7.9(c) that DGC alone successfully hides the GC latencies until four million write requests are served. However, as soon as the available free blocks run out, DGC starts performing on-demand GCs. One can also see from this result that DGC needs AGC, which supplies free blocks, enabling the former to defer on-demand GCs. Both AGC and DGC, when applied individually, increase the number of GCs compared to the baseline GC, which is used to perform on-demand GC of L-FTL (see Figure 7.9(a)). However, when they are applied together, they successfully hide GC latencies, as illustrated in Figure 7.9(d), and the total number of GCs does *not* exceed the baseline case (Section 7.6.5). In the rest of our experiments, we focus on this integrated AGC+DGC scheme.



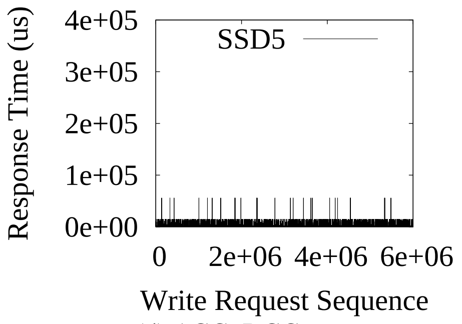
(a) Baseline GC



(b) AGC only



(c) DGC only

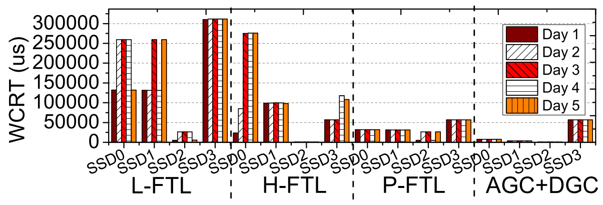


(d) AGC+DGC

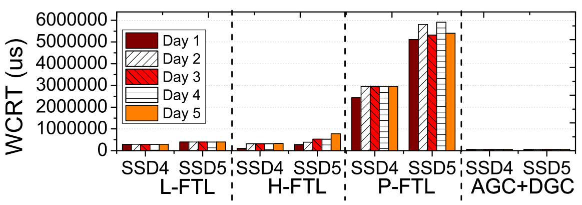
Fig. 7.9 Performance comparison of different garbage collection strategies (SSD5 of 6SSDs-RAID with high I/O intensive workloads).

7.6.2 Worst Case Response Time

Figure 7.10 plot the worst-case response times (WCRTs) 6SSDs-RAID. We see from these graphs that, WCRT ranges from 131 *m*s to 311 *m*s in 6SSDs-RAID-LO, under both the L-FTL and H-FTL schemes. However, in both P-FTL and AGC+DGC we observe negligible WCRTs, which results in completely hiding the GC latencies from the I/O operations. We further observe that AGC+DGC reduces the WCRT by 65.2%, 98.6% and 96.4%, on average, over P-FTL, L-FTL and H-FTL, respectively. This is because AGC+DGC performs on-demand GCs only during the idle periods, and consequently, users experience no GC overheads during their I/O services.



(a) 6SSDs-RAID-LO



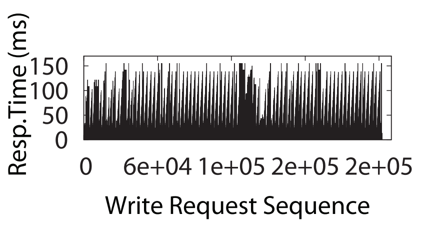
(b) 6SSDs-RAID-HI

Fig. 7.10 Worst-case response time (WCRT) analysis for 6SSDs-RAID. (a) With low I/O intensive workloads, P-FTL and AGC+DGC show deterministic behaviors while the performances of L-FTL and H-FTL fluctuate over time. (b) With high I/O intensive workloads, P-FTL experiences very high WCRT, whereas AGC+DGC continues to provide stable I/O performance.

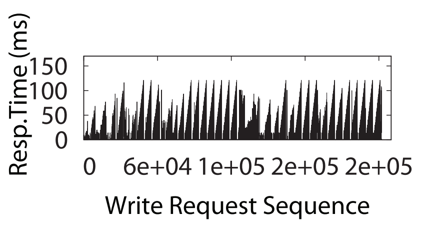
However, in 6SSDs-RAID-HI, P-FTL's WCRT behavior fluctuates due to the *write buffer block thrashing* problem.[3] This causes P-FTL to perform out of order writes for a while and, as a result, WCRTs become ten times worse as compared to the L-FTL case. In contrast, AGC+DGC

[3]This problem arises when the free pages in the write buffer (NV buffer) to which P-FTL writes urgent data are no longer available.

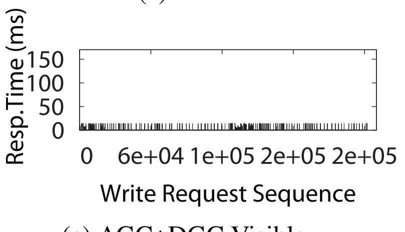
still serves I/O requests within the predefined latencies, and achieves about 53 *m*s latency, including the theoretic minimum for I/O processing, while the other approaches suffer from the performance fluctuations and experience long WCRT under heavy I/O requests. Further, SSDs supported by our AGC+DGC do *not* incur any GC latencies during busy periods, even in execution phases with very low idle times ($\leq$ 10%). This is because AGC eliminates on-demand GCs using idle times, and DGC postpones the GC latencies by shifting them to future idle periods, as plotted in Figure 7.11(c). Figure 7.11 also explains how our proposed GC strategies collectively take GC overheads off the critical path. While L-FTL and H-FTL (see Figures 7.11(a) and 7.11(b)) incur GC latencies during the busy periods, AGC+DGC incurs (see Figure 7.11(d)) GC latencies *only* during the idle periods, which are not perceived by applications. This clearly shows that AGC+DGC provides stable and better SSD performance with no on-demand GCs taking place during the busy periods.



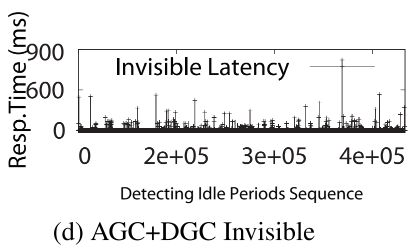
(a) L-FTL



(b) H-FTL



(c) AGC+DGC Visible



(d) AGC+DGC Invisible

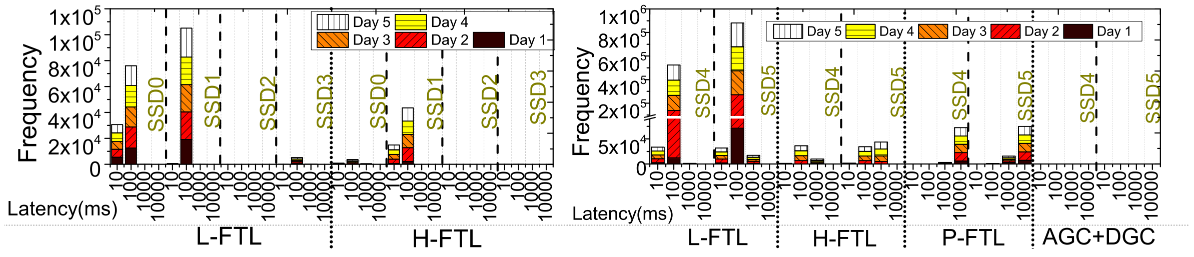
Fig. 7.11 Response times for a write intensive section (where the fraction of I/O executions with no idle time is account for about 90%). While H-FTL removes about 40% of the GC related overheads, AGC+DGC hides all on-demand GC latencies.

7.6.3 Excess Waiting Time

Figure 7.12 plots the amount of excess waiting time (EWT)[4] in 6SSDs-RAID. One can observe from Figure 7.12(a) that H-FTL significantly cuts down the GCs by maximizing the block-level locality. It also dramatically reduces the number of page migrations introduced by GCs. However, it can also be seen that, as the execution progresses (from day 1 to day 5), occurrences of EWTs increase, due to the shortage of available free blocks. To secure free blocks, H-FTL had to merge up to ten blocks into two logical blocks, and merge approximately fifteen thousand times a day, generating significant overheads. In contrast, P-FTL and AGC+DGC successfully hide GC overheads at runtime (and thereby All EWT of them is zero). However, the frequency of EWTs in H-FTL is less than that in P-FTL with high write intensive workloads (see Figure 7.12(b)). In this case, P-FTL could not fully hide GC latencies when the NV buffer was completely used by the large amount of I/O requests. This is because P-FTL incurs much longer latencies than H-FTL, due to the write buffer thrashing problem, which is the same as the one causing high WCRT.

On the other hand, our scheme successfully hides GC latencies because AGC can ahead secure available blocks (delay blocks) to DGC even under high write intensive workloads. Further, because of update block replacement scheme, the delay blocks are the same as the free blocks, thereby not requiring any extra blocks to manage different mapping schemes. Our proposed strategy essentially eliminates on-demand GCs by exploiting different types of idle periods and thus leads to stable GC latencies.

[4]EWT is defined as the difference between the actual wait time and the marginal response time (in this work, it is assumed to be 30 *m*s).



(a) 6SSDs-RAID-LO (b) 6SSDs-RAID-HI

Fig. 7.12 Excess waiting time (EWT). The x-axis represents the upper bound on EWT. (a) L-FTL and H-FTL experience I/O blocking problem stemming from GCs while P-FTL and AGC+DGC have no such problem. (b) With heavy writes, even though P-FTL results in fewer GC invocations, its GC latencies are much longer than others

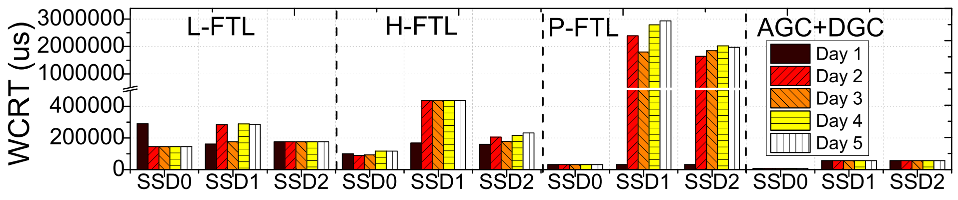


Fig. 7.13 Worst-case response time (WCRT) analysis for the 3SSDs-RAID.

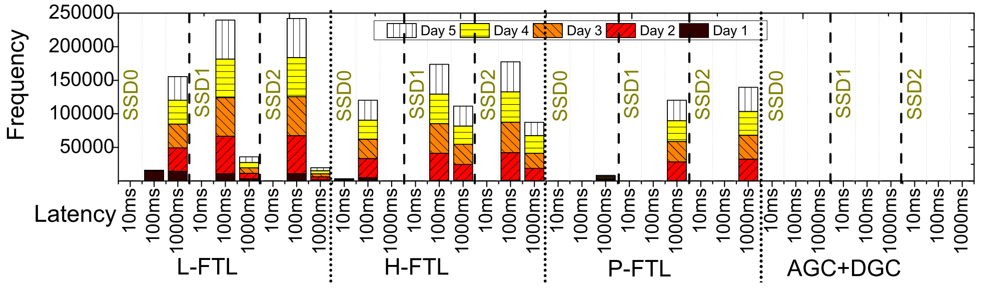


Fig. 7.14 Excess waiting time (EWT) analysis for the 3SSDs-RAID.

7.6.4 Performance Compariosn of 3SSDs-RAID

Figure 7.13 and Figure 7.14 illustrate, respectively, WCRTs and EWTs for 3SSDs-RAID. In both WCRT and EWT analyses, performance of the 3SSDs-RAID is similar to 6SSDs-RAID except for P-FTL. Specifically, in day 1, P-FTL guarantees deterministic performance with the zero EWT value on even high write-intensive workloads (SSD1 and SSD2 of 3SSDs-RAID) because each SSD of 3SSDs-RAID has a larger storage capacity than an SSD in the 6SSDs-RAID configuration. In other words, P-FTL is more tolerant to update block reclaiming GC overheads as its NV buffer has more physical pages. However, as the amount of writes increases, the available physical pages also run out. As a result, P-FTL could not satisfy the deadline requirements again. 3SSDs-RAID with P-FTL has about 50% less impact on the write block thrashing problem compared to 6SSDs-RAID, mainly because, in addition to the larger physical pages on the NV buffer, P-FTL itself can secure abundant free block resource as well, thereby reducing potential GC overheads during free block reclaiming. However, due to reasons similar to the case of 6SSDs-RAID, over the time, P-FTL makes 3SSDs-RAID performance worse than L-FTL and H-FTL. While the performance of P-FTL depends mainly on the size of NV buffer and are not able to essentially take GC overheads off the critical path of SSDs, AGC+DGC satisfies the performance requirements irrespective of different SSD configuration chosen and the I/O traffics tested.

7.6.5 Side-Effects of AGC and DGC

Figure 7.15(a) plots the breakdown of GCs across different collection schemes. Since AGC is responsible for preparing the free blocks, it is desired that the contribution of the AGC be larger than that of the DGC. We see that, as expected, AGC executes for at least 80% of the

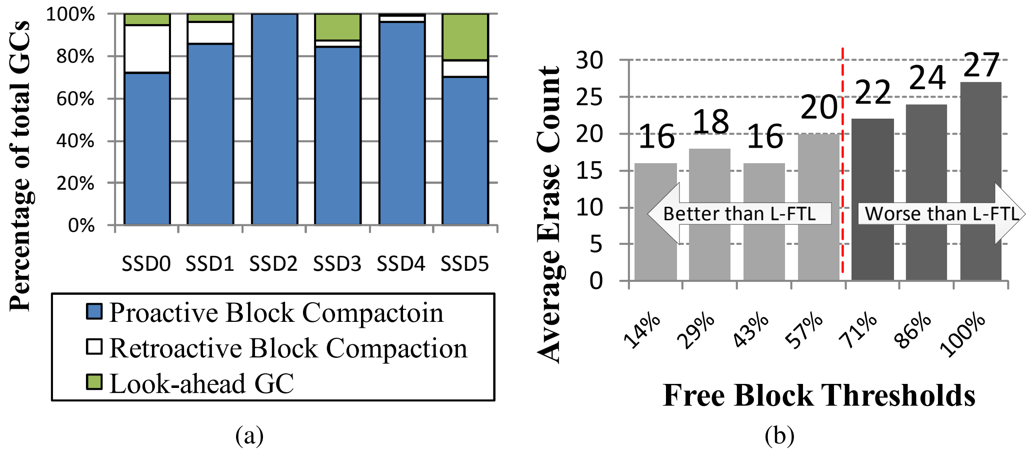


Fig. 7.15 (a) Garbage collection type breakdown of total collection. (b) Block erase impact by free block threshold.

total number of GCs. As a result, DGC is able to secure enough free blocks when it performs update block replacement to delay GCs. We want to point out that the proactive block compaction is applied in a majority of the AGC operations. The proactive block compaction does not execute until the number of free blocks is less than the free block threshold (even though it is under the underlying FTL's GC threshold (3%)). Therefore, our scheme does not introduce any unnecessary erases, and thus reduces the potential side-effects of GC.

Figure 7.15(b) presents the average block erase counts under different free block thresholds when executing AGC. In this figure, the dotted vertical line indicates L-FTL's average erase count per block, which is twenty one. Since AGC is performed only if the target GC block is fully occupied or if an on-demand GC is to be invoked very soon, it only migrates necessary GC activities from busy period, thereby minimizing side effect in terms of SSD reliability.

We observed that the free block threshold should be less than 71% for the average erase count of the proactive block compaction in AGC to be comparable to L-FTL. If the proactive block compaction shifts on-demand GCs beyond this threshold, it makes wear-leveling characteristics worse than L-FTL. Interestingly, the erase counts with low free block thresholds are better than L-FTL. This is because preparing free blocks using fully-occupied blocks in advance helps to prevent the log block thrashing problem (in L-FTL), which can introduce improper erase operations. In our experiments, the best free block threshold for satisfying the wear-leveling requirement was found to be less than 43% of the original GC threshold.

7.7 Conclusions

We proposed novel a garbage collection strategy consisting of two main components, called *Advanced Garbage Collection* (AGC) and *Delayed Garbage Collection* (DGC), that cooperate in hiding GC overheads in SSDs. AGC tries to secure free blocks in advance and remove on-demand GCs from the critical path so that users do *not* experience GC latencies during I/O congestion. In comparison, DGC handles GC invocations that could not be handled by AGC by differing them to future idle periods. Our experimental analysis using both enterprise workloads and high performance I/O workloads indicate that the proposed strategies (AGC and DGC) provide stable I/O performance. Compared to three state-of-the-art GC strategies, P-FTL, L-FTL and H-FTL, our integrated scheme (AGC+DGC) reduces GC overheads dramatically.

Chapter Acknowledgements Chapter 7, in part, is a reprint of the material as it appears in ”Taking Garbage Collection Overheads off the Critical Path in SSDs,” Myoungsoo Jung, Ramya Prabhakar, Mahmut Kandemir, in Proceedings of the 13th ACM/IFIP/USENIX 13th International Conference on Middleware (Middleware), 2012. The dissertation proposal author was the primary investigator and the first author of this paper.

Chapter 8

Future Work

In this chapter, we discuss future work in two categories: near term and long term.

8.1 Near Term Future Research Directions

8.1.1 QoS-aware and GC-aware Host Interface Scheduler

Most existing I/O scheduling algorithms are optimized to reduce the impact of random accesses and are ignorant of the internal details of an SSD architecture. As a result, SSDs may violate quality of service (QoS) requirements by not being able to meet the deadlines of I/O requests. As one of our on-going projects, we are developing a novel host interface I/O scheduler that is both garbage collection aware and QoS aware. More specifically, this QoS-aware and GC-aware host interface scheduler can guarantee for underlying SSDs to satisfy all the deadlines of I/O requests by redistributing the garbage collection overheads across non-critical I/O requests and reducing channel resource contention even under the situation that there exists no idle period.

8.1.2 Out-of-Order Non-Volatile Memory Execution

As stated earlier in this thesis, SSDs are undergoing dramatic technological and architectural changes by employing hundreds of NAND flash chips, multiple I/O channels, multiple cores, and high speed interfaces such as PCI Express. These many-chip SSD architectures enjoy significant performance improvements by parallelizing data accesses across their internal resources. However, we believe that the performance of many-chip SSDs will not be much improved as the amount of internal resources increases (i.e., when more NAND flash chips are added). Main challenges in this context include high device-level idleness and poor resource utilization, caused by parallelism dependency and low flash-level transactional-locality. As one of our on-going projects, we are working on a novel device-level SSD controller, which targets maximizing resource utilization and achieving high performance without additional NAND flash chips. Specifically, this controller can relax parallelism dependency by scheduling I/O requests based on internal resource layout rather than the order imposed by the device-level queue. In addition, this novel controller is expected to improve flash-level parallelism and reduce the number of transactions (i.e., improves transactional-locality) by over-committing flash memory requests to specific resources.

8.1.3 NVM Power Modeling

While multi-channel SSD architecture and many-chip SSDs can offer better performance by taking advantage of internal parallelism, the multiple internal resources such as NVM arrays, cores, busses and controllers can require more power to operate and exhibit unpredictable energy performance behavior. To characterize the energy requirements and understand power dynamics of modern SSDs, our current plan includes extending our cycle-accurate NAND flash simulation model with a high-fidelity power model. This extended simulation framework will be able to capture detailed information related to various types of memory interfaces, data movements, memory island accesses on emerging NVM technologies by modeling power on memory cores, I/O peripherals, drivers, and diverse internal registers (composed by multiple latch circuits).

8.1.4 High-speed Non-Volatile Memory Interface

As NVM performance begins to exceed the maximum bandwidth conventional storage interfaces, we believe that designing a high-speed interface is one of core research topics to expose true performance of underlying NVM systems to user applications or computing cores. Our on-going research includes incorporating a high-speed NVM interface (e.g., SDR 400MHz $\sim$ DDR 800MHz) and designing a specific NVM protocol for SSD to ride the high-speed interface. In addition, we will also study various queuing methodologies and buffer managements, which are mainly optimized for the new NVM protocol that we will demonstrate.

8.2 Long Term Future Research Directions

8.2.1 SSD Redesign

Redesigning SSDs from scratch is aimed at enhancing the NAND flash memory performance by optimizing the storage software stack irrespective of the hardware interface between computing cores and storage. This SSD redesign work will directly tackle performance bottleneck caused by the hardware interface and physical separation between computing cores and storage in order to let SSD and NVM systems realize the full potential of peripheral interface line speeds. Our plans in the context of this redesign work include alleviating such bottleneck with better cooperation between flash firmware (stack) and host stack by modifying interfaces to pass hints/status information between different layers in these software stacks and provisioning compute engines within flash drives, where such additional intelligence and processing bandwidth can considerably boost performance.

8.2.2 Exposing NVM to Computational Resources

One of our long term future plans is elevating NAND flash memory systems to directly connect to the host processor through a dedicated interface, similar to main memory DIMMs interfacing directly to the on-chip cores. This NVM migration (from SSD to computational resources) also transforms recent active SSDs (i.e., SSDs with on-board processor cores) into passive devices similar to main memory. We expect that, in addition to performance improvement, this passive SSD approach will be able to make SSDs “more maintainable” with reasonable economic costs by eliminating costly SSD firmware and extra hardware components from them.

8.2.3 On-Chip NVM Systems

We will also develop an aggressive revolutionary approach to bring NAND-flash on-chip, and explore different placement options for tighter physical integration with the processing cores. The main goal behind this work is removing all the overheads stemming from circuit-level data movement and system-level interface, so that computing cores can fully enjoy the true performance of emerging NVM technologies with much higher memory capacities, compared to existing SRAM/DRAM technologies. We also expect that these on-chip NVM systems will be able to eliminate the needs for refreshing memory cells in an existing DRAM array and reduce the frequency of off-chip memory accesses, which can in turn significantly reduce power consumption in diverse computing domains.

References

[1] AGRAWAL, N., PRABHAKARAN, V., WOBBER, T., DAVIS, J. D., MANASSE, M. S., AND PANIGRAHY, R. Design tradeoffs for SSD performance. In *Proceedings of USENIX ATC* (2008).

[2] ARPACI-DUSSEAU, A. C., ARPACI-DUSSEAU, R. H., AND PRABHAKARAN, V. Removing the costs of indirection in flash-based ssds with namelesswrites. In *Proceedings of HotStorage* (2010).

[3] BATES, K., AND MCNUTT, B. http://traces.cs.umass.edu/index.php/main/traces. In *UMASS Trace Repository*.

[4] BOBOILA, S., AND DESNOYERS, P. Write endurance in flash drives: Measurements and analysis. In *Proceedings of FAST* (2010).

[5] BUCY, J. S., SCHINDLER, J., SCHLOSSER, S. W., AND GANGER, G. R. The disksim simulation environment version 4.0 reference manual.

[6] CAI, Y., YALCIN, G., MUTLU, O., HARATSCH, E. F., CRISTAL, A., UNSAL, O. S., AND MAI, K. Flash correct-and-refresh: Retention-aware error management. In *Proceedings of ICCD* (2012).

[7] CANIM, M., MIHAILA, G. A., BHATTACHARJEE, B., ROSS, K. A., AND LANG, C. A. SSD bufferpool extensions for database systems. *Proceedings of VLDB* (2010).

[8] CAULFIELD, A. M., COBURN, J., MOLLOV, T., DE, A., AKEL, A., HE, J., JAGATHEESAN, A., GUPTA, R. K., SNAVELY, A., AND SWANSON, S. Understanding the impact of emerging non-volatile memories on high-performance, IO-intensive computing. In *Proceedings of SC* (2010).

[9] CAULFIELD, A. M., DE, A., COBURN, J., MOLLOW, T. I., GUPTA, R. K., AND SWANSON, S. Moneta: A high-performance storage array architecture for next-generation, non-volatile memories. In *Proceedings of MICRO* (2010).

[10] CAULFIELD, A. M., GRUPP, L. M., AND SWANSON, S. Gordon: Using flash memory to build fast, power-efficient clusters for data-intensive applications. In *Proceedings of ASPLOS* (2009).

[11] CHANG, L.-P., AND KUO, T.-W. Real-time garbage collection for flash-memory storage systems of real-time embedded systems. *ACM Transactions on Embedded Computing Systems 3*, 4 (November 2004).

[12] CHEN, F., KOUFATY, D. A., AND ZHANG, X. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In *Proceedings of SIGMETRICS* (2009).

[13] CHEN, F., LEE, R., AND ZHANG, X. Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing. In *Proceedings of HPCA* (2011).

[14] CHOI, H., LIU, W., AND SUNG, W. VLSI implementation of BCH error correction for multilevel cell nand flash memory. In *Proceedings of VLSI* (2010).

[15] CHOUDHURI, S., AND GIVARGIS, T. Deterministic service guarantees for NAND flash using partial block cleaning. In *Proceedings of CODES+ISSS* (2008).

[16] COLARELLI, D., AND GRUNWALD, D. Massive arrays of idle disks for storage archives. In *Proceedings of the ACM/IEEE Conference on Supercomputing* (2002).

[17] COOKE, J. How ClearNAND flash simplifies and enhances system designs. In *Micron White Paper* (2011).

[18] CYPRESS. *CY14B256LA nvSRAM*. 2012.

[19] DAVIDSON, J. W., AND JINTURKAR, S. Memory access coalescing: a technique for eliminating redundant memory accesses. *Proceedings of PLDI* (1994).

[20] DIRIK, C., AND JACOB, B. The performance of PC solid-state disks (SSDs) as a function of bandwidth, concurrency, device architecture, and system organization. In *Proceedings of ISCA* (2009).

[21] EMC. Raw drive capacity cost trends http://wikibon.org/w/images/a/a4/emcrawdrivecapacitycosttrends.jpg.

[22] FISHER, R. Optimizing nand flash performance. In *Proceedings of FlashMemory Summit* (August 2008).

[23] FUSION-IO. ioCache. In *datasheet* (2012).

[24] FUSION-IO. ioMemory. In *datasheet* (2012).

[25] FUSION-IO. ioTurbine. In *datasheet* (2012).

[26] GOLDING, R., BOSCH, P., STAELIN, C., SULLIVAN, T., AND WILKES, J. Idleness is not sloth. In *Proceedings of the USENIX Annual Technical Conference* (1995), pp. 201–212.

[27] GRUPP, L. M., CAULFIELD, A. M., COBURN, J., SWANSON, S., YAAKOBI, E., SIEGEL, P. H., AND WOLF, J. K. Characterizing flash memory: Anomalies, observations,and applications. In *Proceedings of SC* (2009).

[28] GUPTA, A., KIM, Y., AND URGAONKAR, B. DFTL: A flash translation layer employing demand-based selective caching of page-level address mappings. In *Proceedings of ASPLOS* (2009).

[29] HONG, S. I., MCKEE, S. A., SALINAS, M. H., KLENKE, R. H., AYLOR, J. H., AND WULF, W. A. Access order and effective bandwidth for streams on a direct rambus memory. In *Proceedings of HPCA* (1999).

[30] HU, X.-Y., ELEFTHERIOU, E., HAAS, R., ILIADIS, I., AND PLETKA, R. Write amplification analysis in flash-based solid state drives. In *Proceedings of SYSTOR* (2009).

[31] HU, Y., JIANG, H., FENG, D., TIAN, L., LUO, H., AND ZHANG, S. Performance impact and interplay of ssd parallelism through advanced commands, allocation strategy and data granularity. In *Proceedings of ISC* (2011).

[32] HU, Y., JIANG, H., FENG, D., TIAN, L., LUO, H., AND ZHANG, S. Performance impact and interplay of SSD parallelism through advanced commands,allocation strategy and data granularity. In *Proceedings of ICS* (2011).

[33] HUR, I., AND LIN, C. Adaptive history-based memory schedulers for modern processors.

[34] HYNIX, INC. NAND flash memory MLC datasheet, H27UBG8T2A. In *http://www.hynix.com/* (2009).

[35] INTEL. http://www.iometer.org/. In *Iometer User's Guide* (2003), Intel.

[36] INTEL. *NVM Express Revision 1.0*. Intel, March, 2011.

[37] INTEL, AND SEAGATE. *Serial ATA Native Command Queuing: An Exciting New Performance Feature for Serial ATA*. Intel and Seagate, July, 2003.

[38] INTEL, AND SEAGATE. *Serial ATA Native Command Queuing: An Exciting New Performance Feature for Serial ATA*. Intel and Seagate, July, 2003.

[39] J. H. KIM ET AL. *Incremental Merge Methods and Memory Systems Using the Same*. U.S. Patent #2006004971A1, Jan. 5, 2006.

[40] JO, H., KANG, J.-U., PARK, S.-Y., KIM, J.-S., AND LEE, J. FAB: flash-aware buffer management policy for portable media players. In *Proceedings of Consumer Electronics, IEEE Transactions on* (May 2006).

[41] JOSEPHSON, W. K., ET AL. Dfs: A file system for virtualized flash storage. In *Proceedings of FAST* (2010).

[42] JUNG, J.-Y. S., ET AL. FTL design exploration in reconfigurable high-performance SSD for server applications. In *Proceedings of ICS* (2009).

[43] JUNG, M., ET AL. Cooperative memory management.

[44] JUNG, M., ET AL. Memory system and data storing method thereof. *U.S. Patent 20090248987* (2009).

[45] JUNG, M., AND KANDEMIR, M. An evaluation of different page allocation strategies on high-speed SSDs. In *Proceedings of HotStorage* (2012).

[46] JUNG, M., AND KANDEMIR, M. Middleware - firmware cooperation for high-speed solid state drives. In *Proceedings of Middleware D&P* (2012).

[47] JUNG, M., AND KANDEMIR, M. Revisiting widely held ssd expectations and rethinking system-level implications. In *Proceedings of SIGMETRICS* (2013).

[48] JUNG, M., PRABHAKAR, R., AND KANDEMIR, M. T. Taking garbage collection overheads off the critical path in ssds. In *Proceedings of Middleware* (2012).

[49] JUNG, M., WILSON, E. H., DONOFRIO, D., SHALF, J., AND KANDEMIR, M. NAND-FlashSim: Intrinsic latency variation aware NAND flash memory system modeling and simulation at microarchitecture level. In *Proceedings of MSST* (2012).

[50] JUNG, M., WILSON III, E. H., AND KANDEMIR, M. Physically addressed queueing (PAQ): Improving parallelism in solid state disks. In *Proceedings of ISCA* (2012).

[51] JUNG, M., AND YOO, J. Scheduling garbage collection opportunistically to reduce worst-case I/O performance in solid state disks. In *Proceedings of IWSSPS* (2009).

[52] KANG, J.-U., JO, H., KIM, J.-S., AND LEE, J. A superblock-based flash translation layer for NAND flash memory. In *Proceedings of the ACM International Conference on Embedded Software* (October 2006).

[53] KANG, J.-U., KIM, J.-S., PARK, C., PARK, H., AND LEE, J. A multi-channel architecture for high performance NAND flash-based storage system. *Journal of Systems Architecture* (2007).

[54] KGIL, T., ROBERTS, D., AND MUDGE, T. Improving NAND flash based disk caches. In *Proceedings of ISCA* (2008).

[55] KGIL, T., ROBERTS, D., AND MUDGE, T. Improving NAND flash based disk caches. In *Proceedings of the 35th Annual International Symposium on Computer Architecture* (2008).

[56] KIM, H., AND AHN, S. BPLRU: A buffer management scheme for improving random writes in flash storage. In *Proceedings of USENIX Conference on File and Storage Technologies* (2008).

[57] KIM, J., KIM, J. M., NOH, S. H., MIN, S. L., AND CHO, Y. A space-efficient flash translation layer for Compact Flash systems. In *Proceedings of IEEE Transactions on Consumer Electronics* (2002).

[58] KIM, Y., HAN, D., MUTLU, O., AND HARCHOL-BALTER, M. Atlas: A scalable and high-performance scheduling algorithm for multiple memory controllers. In *Proceedings of HPCA* (2010).

[59] KIM, Y., PAPAMICHAEL, M., MUTLU, O., AND HARCHOL-BALTER, M. Thread cluster memory scheduling: Exploiting differences in memory access behavior. In *Proceedings of MICRO* (2010).

[60] KIM, Y., TAURAS, B., GUPTA, A., AND URGAONKAR, B. Flashsim: A simulator for NAND flash-based solid-state drives. In *Proceedings of SIMUL* (2009).

[61] KOLTSIDAS, I., AND VIGLAS, S. The case for flash-aware multi level caching.

[62] LECROY. http://www.lecroy.com/.

[63] LEE, J., ET AL. Memory system and method of accessing a semiconductor memory device. In *US2009/0310408A1* (December 2009).

[64] LEE, S., HA, K., ZHANG, K., KIM, J., AND KIM, J. Flexfs: A flexible flash file system for MLC NAND flash memory. In *Proceedings of ATC* (2009).

[65] LEE, S., LEE, Y.-T., HAN, W.-K., KIM, D.-H., KIM, M.-S., MOON, S.-H., CHO, H. C., LEE, J.-W., BYEON, D.-S., LIM, Y.-H., ET AL. A 3.3v 4gb four-level NAND flash memory with 90nm cmos technology. In *Proceedings of IEEE International Solid-State Circuits Conference* (2004).

[66] LEE, S.-W., MOON, B., PARK, C., KIM, J.-M., AND KIM, S.-W. A case for flash memory SSD in enterprise database applications. In *Proceedings of SIGMOD* (June 2008), pp. 9–12.

[67] LEE, S.-W., PARK, D.-J., CHUNG, T.-S., LEE, D.-H., PARK, S., AND SONG, H.-J. A log buffer-based flash translation layer using fully-associative sector translation. *ACM Transactions on Embedded Computing Systems 6*, 3 (2007).

[68] LIU, N., COPE, J., CARNS, P., CAROTHERS, C., ROSS, R., GRIDER, G., CRUME, A., AND MALTZAHN, C. On the role of burst buffers in leadership-class storage systems. In *Proceedings of MSST* (2012).

[69] LIU, Y., HUANG, J., XIE, C., AND CAO, Q. Raf: A random access first cache management to improve SSD-based disk cache. *NAS* (2010).

[70] MAGHRAOUI, K. E., ET AL. Modeling and simulating flash based solid-state disks for operating systems. In *Proceedings of WOSP/SIPEW* (2010).

[71] MCKEE, S., AND WULF, W. Access ordering and memory-conscious cache utilization. In *Proceedings of HPCA* (1995).

[72] MI, N., RISKA, A., ZHANG, Q., SMIRNI, E., AND RIEDEL, E. Efficient management of idleness in storage systems. In *Proceedings of the ACM Transactions on Storage Journal* (June 2009).

[73] MICHAEL F. WEHNER AND OTHERS. Hardware/software co-design of global cloud system resolving models. *JAMES* (2011).

[74] MICHELONI, RINO ET AL. *Inside NAND Flash Memories*. Springer, 2010.

[75] MICRON TECHNOLOGY, INC. NAND flash memory MLC datasheet, MT29F8G08MAAWC, MT29F16G08QASWC. In *http://www.micron.com/* (2004).

[76] MUTLU, O., AND MOSCIBRODA, T. Stall-time fair memory access scheduling for chip multiprocessors. In *Proceedings of MICRO* (2007).

[77] NARAYANAN, D., DONNELLY, A., THERESKA, E., AND ELNIKETY, S. Everest: Scaling down peak loads through I/O off-loading. In *Proceedings of EuroSys Conference* (2008).

[78] NARAYANAN, D., THERESKA, E., DONNELLY, A., ELNIKETY, S., AND ROWSTRON, A. Migrating server storage to ssds: analysis of tradeoffs. In *Proceedings of EuroSys* (2009).

[79] NESBIT, K. J., AGGARWAL, N., LAUDON, J., AND SMITH, J. E. Fair queuing memory systems. In *Proceedings of MICRO* (2006).

[80] ONFI WORKING GROUP. Open nand flash interface. In *http://onfi.org/*.

[81] ONFI WORKING GROUP. Open nand flash interface 3.0. In *http://onfi.org/* (2012).

[82] OU, Y., HÄRDER, T., AND JIN, P. Cfdc: a flash-aware replacement policy for database buffer management. In *Proceedings of the Fifth International Workshop on Data Management on New Hardware* (2009).

[83] OUYANG, X., MARCARELLI, S., AND PANDA, D. K. Enhancing checkpoint performance with staging I/O and SSD. In *Proceedings of International Workshop on Storage Network Architecture and Parallel I/O* (2010).

[84] PARK, S.-H., HA, S.-H., BANG, K., AND CHUNG, E.-Y. Design and analysis of flash translation layers for multi-channel NAND flash-based storage devices. In *Proceedings of TCE* (2009).

[85] PARK, S.-Y., ET AL. Exploiting internal parallelism of flash-based ssds. In *Proceedings of Computer Architecture Letters* (January 2010), p. 9.

[86] PATTERSON, D. A. Latency lags bandwidth. In *Proceedings of Communication of The ACM* (October 2004), p. 71.

[87] RAFIQUE, N., LIM, W.-T., AND THOTTETHODI, M. Effective management of DRAM bandwidth in multicore processors. In *Proceedings of PACT* (2007).

[88] REPOSITORY, S. http://iotta.snia.org/.

[89] RIXNER, S., DALLY, W. J., KAPASI, U. J., MATTSON, P., AND OWENS, J. D. Memory access scheduling. In *Proceedings of ISCA* (2000).

[90] ROOHPARVAR, F. F. Single level cell programming in a multiple level cell non-volatile memory device. In *U.S. Patent 7529129* (2007).

[91] SAMSUNG. K9GAG0B0M http://www.samsung.com/global/business/semiconductor/. In *Data Sheet* (March 2008).

[92] SAXENA, M., AND SWIFT, M. M. FlashVM: Virtual memory management on flash. In *Proceedings of USENIX ATC* (2010).

[93] SHALF, J., ET AL. http://www.lbl.gov/cs/html/greenflash.html. In *A New Breed of Supercomputers for Improving Global Climate Predictions*.

[94] SHIBATA, N., KANDA, K., HISADA, T., ISOBE, K., SATO, M., SHIMIZU, Y., SHIMIZU, T., SUGIMOTO, T., KOBAYASHI, T., INUZUKA, K., ET AL. A 19nm 112.8 mm¡ sup¿ 2¡/sup¿ 64gb multi-level flash memory with 400mb/s/pin 1.8 v toggle mode interface. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2012 IEEE International* (2012).

[95] SRINIVASAN, M., AND CALLAGHAN, M. Flashcache at facebook. In *Facebook White Paper* (2010).

[96] T10. http://www.t10.org/. In *SCSI Storage Interfaces* (September 2009), Technical Committee T10.

[97] T13. *Serial ATA Specification 3.1*. 2012.

[98] TENSILICA. http://www.tensilica.com/products/hw-sw-dev-tools/. In *Hardware and Software Development Tools*.

[99] ULINK TECHNOLOGY. http://www.ulinktech.com/.

[100] WEI, M. Y. C., GRUPP, L. M., SPADA, F. E., AND SWANSON, S. Reliably erasing data from flash-based solid state drives. In *Proceedings of FAST* (2011).

[101] WON, B. Y. Y., KANG, S. C. S., CHOI, J., AND YOON, S. SSD characterization: From energy consumption's perspective. In *Proceedings of HotStorage* (2011).

[102] YEONG PARK, S., SEO, E., SHIN, J.-Y., MAENG, S., AND LEE, J. Exploiting internal parallelism of flash-based ssds. *IEEE CAL.* (2010).

[103] ZHANG, Y., ARULRAJ, L. P., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. De-indirection for flash-based ssds with namelesswrites. In *Proceedings of FAST* (2012).

[104] ZURAVLEFF, W. K., AND ROBINSON, T. Controller for a synchronous DRAM that maximizes throughput by allowing memory requests and commands to be issued out of order. *U.S. Patent No: 5,630,096* (1997).

Curriculum Vitae

Brief Biography.

I earned Ph.D. in computer science at Pennsylvania State University (Advisor: *Mahmut Kandemir*) and master of science in computer science from Georgia Institute of Technology (Mentor: *Sung Kyu Lim* and *Hsien-Hsin S. Lee*). As a guest research scientist, I am also co-working with Lawrence Berkeley National Laboratory in modeling and simulating diverse memory technologies on scientific applications (Mentor: *John Shalf*). In addition to these academic activities, I have 6+ years industry experience, several U.S. patents related to multi-channel SSDs, and approximately thirty technical papers regarding flash firmware and kernel-level file systems.

Selected Publications.

• **Myoungsoo Jung**, Mahmut Kandemir, "Challenges in Getting Flash Drives Closer to CPU," *Proceedings of the USENIX Workshop on Hot Topics in Storage and File Systems*, 2013.
• **Myoungsoo Jung**, Mahmut Kandemir, "Design of a Large-Scale Storage-Class RRAM System," *Proceedings of the International Conference on Supercomputing*, 2013.
• **Myoungsoo Jung**, Mahmut Kandemir, "Revisiting Widely-held Expectations of SSD and Rethinking Implications for Systems," *Proceedings of the ACM SIGMETRICS*, 2013.
• **Myoungsoo Jung**, Mahmut Kandemir, "Middleware - Firmware Cooperation for High-Speed Solid State Drives," *Proceedings of the ACM/IFIP/USENIX International Conference on Middleware (D&P)*, 2012.
• **Myoungsoo Jung**, Ramya Prabhakar, Mahmut Kandemir, "Taking Garbage Collection Overheads off the Critical Path in SSDs," *Proceedings of the ACM/IFIP/USENIX International Conference on Middleware*, 2012.
• **Myoungsoo Jung**, Mahmut Kandemir, "An Evaluation of Different Page Allocation Strategies on High-Speed SSDs," *Proceedings of the USENIX Workshop on Hot Topics in Storage and File Systems*, 2012.
• **Myoungsoo Jung**, Ellis Herbert Wilson III, Mahmut Kandemir, "Physically Addressed Queueing (PAQ): Improving Parallelism in Solid State Disks," *Proceedings of the International Symposium on Computer Architecture*, 2012.
• **Myoungsoo Jung**, Ellis Herbert Wilson III, David Donofrio, John Shalf, Mahmut Kandemir, "NANDFlashSim: Intrinsic Latency Variation Aware NAND Flash Memory System Modeling and Simulation at Microarchitecture level," *Proceedings of the IEEE Conference on Massive Data Storage*, 2012.
• **Myoungsoo Jung**, Joonhyuk Yoo, "A Re-configurable Flash Translation Layer Architecture for NAND Flash based Applications,", *Proceedings of the International Workshop on Software Support for Portable Storage*, 2009.