EXTRACTING BETTER PERFORMANCE FROM THE PARALLELISM OFFERED BY SSDS

A Dissertation in Computer Science and Engineering by Nima Elyasi

© 2019 Nima Elyasi

Submitted in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy

May 2019
The dissertation of Nima Elyasi was reviewed and approved* by the following:

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

Chita R. Das  
Professor of Computer Science and Engineering  
Head of the Department of Computer Science and Engineering

Mahmut T. Kandemir  
Professor of Computer Science and Engineering

Qian Wang  
Professor of Mechanical Engineering

Changho Choi  
Principal Engineer, Samsung Semiconductor Inc.  
Special Member

*Signatures are on file in the Graduate School.
Abstract

The majority of growth in the industry is driven by massive data processing which in turn is driving a tremendous need for high performance storage. To satisfy the lower latency demands, large-scale computing platforms have been making heavy use of flash-based Solid State Drives (SSDs), which provide substantially lower latencies compared to conventional hard disk drives. To satisfy the capacity and bandwidth demands, SSDs continue to scale by (i) adopting new flash technologies such as V-NAND, and (ii) embodying more flash chips which leads to higher levels of internal parallelism. With the tremendous growth in SSDs capacity and the continuing rise in their internal hardware parallelism, load imbalance and resource contention remain serious impediments towards boosting their performance. Employing and exploiting higher levels of internal parallelism in SSDs can even accentuate the load imbalance as variable-sized requests span more number of flash chips and impose more complexities to the request schedulers when coordinating the individual queues. On the other hand, the widely differential latency of the basic flash operations: read, write, and erase, exacerbates the load imbalance since not all chips are necessarily doing the same operation at the same time. As a consequence of such unbalanced system, SSD requests experience considerable inefficiencies in terms of non-uniformity and non-determinism in their service, which can in turn impair the profitability of client-facing applications. In this dissertation, remedies to alleviate these challenges are proposed, developed and evaluated. The proposed performance-enhancement mechanisms can be incorporated in the device firmware to provide faster and more consistent service.

In this dissertation, we address the load imbalance problem by (i) exploiting the variation in the queue lengths to better take advantage of offered hardware parallelism while serving SSD requests, and (ii) balancing the load across different flash chips by opportunistically re-directing the load to less busy flash chips. First, we propose and develop a scheduling mechanism which orchestrates SSD requests at the (i) arrival time: when inserting requests in their respective queues, and (ii) service time: when issuing requests on flash chips. The former estimates
and leverages the skews in completion of sub-requests in order to allow the new arrivals to jump ahead of the existing ones without affecting their response times. The latter, however, aims at achieving time sharing of available resources by coordinately scheduling sub-requests of each request at the service time. Such scheduling mechanisms are targeted at reducing the response time of SSD requests by coordinating SSD requests in their respective queues. Apart from such optimizations—which are restricted in terms of flash chips servicing a request, one can attempt to re-direct requests to other flash chips which are opportunistically free. Providing this re-direction opportunity is nontrivial for read requests. In the second part of this dissertation, we propose novel approaches to re-direct the load to less busy flash chips. With our proposed techniques, re-direction of read requests is achieved by (i) selective replication, wherein the value popularity is leveraged to replicate the popular data on multiple flash chips and provide more opportunities for read re-direction; and (ii) leveraging existing RAID groups in which flash chips are organized, and reconstructing the read data from the remaining chips of the group, rather than waiting for a long latency operation to complete. While both read re-direction approaches aim at exploiting the existing redundancy to improve response times of SSD requests, the former technique is designed for a content-addressable SSD, while the latter is applicable to normal SSDs with RAID-like capabilities which are more common in high-end SSDs employed in large-scale production environments.
# Table of Contents

List of Figures viii

List of Tables xii

Acknowledgments xiii

Chapter 1

Introduction 1

1.1 Motivation and Objectives 1

1.1.1 Uniqueness of Scheduling and Load Balancing for SSDs 3

1.1.2 Dissertation Outline 5

1.2 Contributions 6

Chapter 2

Background 9

2.1 SSD Architecture 9

2.2 Flow of a Request within the SSD 9

2.3 Garbage Collection 11

Chapter 3

Exploiting Non-uniformity Within Requests’ Service Time 12

3.1 Non-uniformity in Requests’ Service 13

3.1.1 What Is Intra-Request Slack? 13

3.1.2 Why Does Slack Arise? 13

3.1.3 How to Get the Most out of this Slack? 15

3.2 Proposed Scheduling Mechanism – Slacker 17

3.2.1 Reordering 17

3.2.1.1 Reordering Algorithm 18

3.2.1.2 Examples of Rbyp and Wbyp 20

3.2.2 Slack-aware Write Pausing 20
<table>
<thead>
<tr>
<th>Section</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.2.3</td>
<td>Estimating Slack</td>
</tr>
<tr>
<td>3.2.4</td>
<td>Accuracy of Estimation</td>
</tr>
<tr>
<td>3.2.5</td>
<td>Hardware Support and Latency Overhead</td>
</tr>
<tr>
<td>3.3</td>
<td>Co-scheduling of SSD Requests</td>
</tr>
<tr>
<td>3.3.1</td>
<td>Slacker++ Mechanisms</td>
</tr>
<tr>
<td>3.4</td>
<td>Experimental Setting</td>
</tr>
<tr>
<td>3.5</td>
<td>Experimental Evaluation</td>
</tr>
<tr>
<td>3.5.1</td>
<td>Results for Write-Intensive Traces</td>
</tr>
<tr>
<td>3.5.2</td>
<td>Results for Read-Write Balanced Traces</td>
</tr>
<tr>
<td>3.5.3</td>
<td>Results for Read-Intensive Traces</td>
</tr>
<tr>
<td>3.5.4</td>
<td>Sensitivity Analysis</td>
</tr>
</tbody>
</table>

Chapter 4

**Leveraging Content Popularity for Read Re-direction in SSDs**

<table>
<thead>
<tr>
<th>Section</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Motivation and Objectives</td>
</tr>
<tr>
<td>4.1.1</td>
<td>All Chip Replication</td>
</tr>
<tr>
<td>4.2</td>
<td>Selective Replication</td>
</tr>
<tr>
<td>4.2.1</td>
<td>What to Replicate?</td>
</tr>
<tr>
<td>4.2.2</td>
<td>How Much Space for Replication?</td>
</tr>
<tr>
<td>4.2.3</td>
<td>When to Replicate?</td>
</tr>
<tr>
<td>4.2.4</td>
<td>How Many and Where to Replicate?</td>
</tr>
<tr>
<td>4.3</td>
<td>Implementation</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Content-Addressable SSD: The Baseline</td>
</tr>
<tr>
<td>4.3.2</td>
<td>SSD Enhancements for RR-SSD</td>
</tr>
<tr>
<td>4.4</td>
<td>Experimental Setup</td>
</tr>
<tr>
<td>4.4.1</td>
<td>Methodology</td>
</tr>
<tr>
<td>4.5</td>
<td>Experimental Results</td>
</tr>
<tr>
<td>4.5.1</td>
<td>ISO-Capacity Performance Analysis</td>
</tr>
<tr>
<td>4.5.2</td>
<td>Impact of Replication Space Overhead on Read Performance</td>
</tr>
<tr>
<td>4.5.3</td>
<td>Efficiency of Mapping Strategy in RR-SSD</td>
</tr>
<tr>
<td>4.5.4</td>
<td>On-line Replication</td>
</tr>
<tr>
<td>4.5.5</td>
<td>Tail Latency Analysis</td>
</tr>
<tr>
<td>4.5.6</td>
<td>Overhead Analysis</td>
</tr>
</tbody>
</table>

Chapter 5

**Trimming Read Tail Latency in SSDs**

<table>
<thead>
<tr>
<th>Section</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Characterizing Read Tail Latency</td>
</tr>
</tbody>
</table>
List of Figures

1.1 CDF of chip-level parallelism of requests for 4 representative workloads. 2

2.1 A generic architecture of modern SSDs. 10

3.1 Example of (a) a slack-unaware and (b) a slack-aware scheduler (B completes earlier without impacting A). 14

3.2 CDF of slack time for four representative workloads. 14

3.3 The maximum, minimum and average request queue sizes over 100sec for prxy0. 15

3.4 2-D bin packing of sub-requests. 17

3.5 Reordering incoming sub-requests at FSU. 17

3.6 Examples for (a) Rbyp (RB and RC finished 1 cycle earlier without impacting RA and WA), (b) Wbyp (WC finished 4 cycles earlier without impacting WA and WB), and (c) WP (RA and RB finished 1 and 3 cycles earlier without impacting WA and WB). RA, RB, and RC are read requests, WA, WB, and WC are write requests. RAi is the i\textsuperscript{th} sub-request in RA. 19

3.7 Estimation error. 26
3.8 Examples for (a) pure co-scheduling without using empty time slots (flash chips remain idle if the next request cannot be completely serviced), (b) co-scheduling with using empty time slots (the baseline system in our studies – RB finished 1 cycle earlier without impacting others), and (c) co-scheduling and backfilling (RB finished 2 cycles earlier without hurting response time of RA). RAi is the i\textsuperscript{th} sub-request in RA. .................................................. 27

3.9 Response time improvement results over the baseline system for (a) write-intensive, (b) read-write balanced, and (c) read-intensive traces. 32

3.10 The percentage of times when a non-critical sub-request is serviced while a critical sub-request is stuck in the same queue, when using Slacker. ............................................................... 33

4.1 Breakdown of read and write service time in a 64-chip SSD. .... 39

4.2 The percentage of read and total service time improvement given by the Ideal-All-Chip Replication scheme. ...................... 41

4.3 VP for read and write requests for the entire duration of three workloads. The points (values) on X-axis are sorted based on their popularity in reads. .................................................. 42

4.4 Temporal VP for reads for the first 7 consecutive days of three workloads. The points on X-axis are sorted based on their read value popularity in the first day. ......................... 44

4.5 The number of distinct values accessed for the first time during each day in each of the three workloads. ........................ 46

4.6 A high-level view of one period of our replication scheme. .... 47

4.7 (a) Components of the proposed RR-SSD. (b) Tables used in the Mapping Unit. .......................... 50
4.8 Read performance improvement brought by RR-SSD and Oracle system compared to CA-SSD. The solid line in the top graph of each workload is the improvement achieved by Oracle. The bottom graph of each workload plots the fraction of read requests that had opportunity to read the replicated data from previous day (Rep) and data that was newly read (New) for the day under study. 57

4.9 Read service time improvement as a function of allowed space overhead for replicating data. Results of the Oracle system is shown for reference. 60

4.10 Read service time improvement with different mapping strategies (normalized to the CA-SSD). 62

4.11 Read/Total service time improvement for end-of-the-day replication and on-line replication. 62

4.12 Reduction in tail latency. 63

5.1 Impact of queue depth on IOPS and tail latency. $D_i$ refers to the queue depth of $i$. Increasing queue depth puts more stress on the device until it saturates IOPS. 66

5.2 Read latency distribution for 100% random read I/O. 67

5.3 Latency distributions of 8 jobs with iodpth of 1 vs. 1 job with iodepth of 8. 68

5.4 Read latency log (in msec) for a very short window of time. 69

5.5 Read latency distributions for different read-write ratios when (a) no GC is invoked, and (b) in the presence of GC. The bottom line is read-only case. 69

5.6 A representative organization for RAID-equipped SSD. 73

5.7 Sequence of events occurring at a group (x). 74

5.8 Extra read overhead analysis. 75
5.9 Dynamic threshold-based scheduling. ....................... 79

5.10 (a) Average and (b) tail latency of ATLAS and Read-only systems, normalized to the latencies of the baseline system. ............... 85

5.11 Average write latency of ATLAS, normalized to the average write latency of the baseline system. ................................. 88

5.12 Performance analysis of our mechanism with different RAID settings. 89
List of Tables

3.1 Main characteristics of simulated SSD. .................................. 29
3.2 Characteristics of the evaluated I/O traces. .......................... 31
4.1 Comparative analysis of daily-replication versus weekly-replication. 46
4.2 Main characteristics of simulated SSD. ............................... 56
4.3 Workload Characteristics. Unique values represent the percentage of read (write) requests which read (write) unique 4KB chunks. Improvement by CA-SSD denotes the percentage of reduction in read (write) response time by [1] compared to the conventional SSDs. 56
4.4 Read and total (read+write) service time improvement of RR-SSD and Oracle system over the baseline CA-SSD for entire duration of each workload. ................................................ 59
4.5 Ratio of Replication Writes to Total Writes ........................... 63
5.1 Characteristics of the simulated SSD. ................................ 83
5.2 Characteristics of the evaluated I/O traces. ........................... 84
Acknowledgments

I am very thankful to each and everyone who have contributed to this dissertation either by their intellectual contribution or motivational support. I would like to express my highest gratitude to my advisor, Professor Anand Sivasubramaniam, for his endless support and inspiring supervision and encouragement throughout my graduate studies. Besides, I would like to thank my dissertation committee members for their constructive feedback and invaluable advice.

Last but not least, I would like to express my deepest gratitude to my family and friends for their warm love, continued patience, and unwavering support. A special thanks to my beloved parents.

In conclusion, I recognize that this research would not have been possible without the financial assistance provided in part by NSF grants 1302557, 1213052, 1439021, 1302225, 1629129, 1526750, 1629915, 1714389, 1763681, 1439057, 1409095, and 1626251, a DARPA/SRC JUMP grant, and a grant from Intel, and I would like to express my gratitude to them.
Dedication

To My Parents
Chapter 1
Introduction

1.1 Motivation and Objectives

The ever-increasing need for data processing is extremely growing with data-driven applications crunching voluminous datasets. Such large-scale customer-facing applications require the computing systems to provide them with swift and seamless responsiveness in a cost-effective manner. This necessitates the adoption of high-performance storage systems to accelerate the data accesses, and hence improve the whole system performance, since longer delays from storage can impact the profitability of such datacenter-scale applications [2–6]. NAND Flash Solid State Drives (SSDs) are widely adopted in large production environments and provide substantially lower latency and higher throughput than conventional disks, with a continually dropping price per gigabyte to make them increasingly attractive. SSDs continue to scale by the introduction of new forms of NAND technologies (such as V-NAND), and embodying more flash chips which leads to higher levels of internal parallelism. With the tremendous growth in SSDs capacity and the continuing rise in their internal hardware parallelism, load imbalance and resource contention remain serious impediments towards boosting their performance. Employing and exploiting higher levels of internal parallelism in SSDs can even accentuate the load imbalance as variable-sized requests span more number of flash chips and impose more complexities to the request schedulers when coordinating the individual queues. An immediate ramification of the unbalanced load across different flash chips is the non-uniformity in the completion times of sub-tasks within each request. Moreover, the variances in the queue lengths along with the differential operation
latencies in NAND flash SSDs, introduce occasional episodes of high latency, which in turn impairs the consistency in servicing SSD requests, leading to violation in the latency SLOs guaranteed by the storage service providers – non-determinism problem. Thus, this dissertation focuses on addressing the non-uniformity and non-determinism problems in the emerging SSD architectures.

**Non-Uniformity in Completion Times:** Today’s SSDs offer multiple layers of hardware parallelism (multiple flash chips each with multiple dies and planes) to achieve high degrees of parallelism within and across read/write requests. In the quest to maximize parallelism and utilization, schedulers split and direct incoming requests to different flash chips. If each I/O request spans enough pages to exactly match the offered hardware parallelism, then the hardware utilization/parallelism is automatically maximized regardless of the order in which the requests are serviced. However, as can be seen in Figure 1.1, a majority of the requests are relatively small, with their pages spanning just a few chips leading to lower chip-level parallelism. When these small requests intersect on some flash chips, some sub-requests of a request have to wait their turn for their respective chips while their sibling sub-requests of the same request are being serviced at other chips. Moreover, not all operations take the same latency, especially when comparing writes versus reads where the former can be 10–40 times the latter, exacerbating the problem. As a consequence, we will show that even modern schedulers [7] can result in a highly unbalanced system with chip-level queues exhibiting very high variance, which in
turn can skew the end times of the sub-requests (within a request) directed to different flash chips. One focus of this dissertation is to leverage the differences in the completion times to improve the response time of requests in SSDs.

**Non-Determinism in Requests’ Service** The variability in the queue lengths of different flash chips is magnified at the request service level. Consider a situation where an SSD employs several flash chips with each of them typically responding very fast but with a relatively high 99\textsuperscript{th}-percentile latency. If requests of this SSD are all handled by one flash chip, one request in 100 will experience very high latency. However, when requests span on different flash chips, their response times can be extremely affected by very modest fraction of latency outliers, since even a tail delay in a fraction of requests/sub-requests can affect the whole requests’ latency. With SSDs scaling up and encompassing tens or even hundreds of flash chips to provide terabytes of capacity, SSD requests can experience frequent high latency episodes (tail of up to the 99.99\textsuperscript{th} latency percentile) during the course of operation, leading to violation in the latency SLOs guaranteed by the storage service providers. The importance of minimizing the tail has drawn a lot of recent attention from overall system capacity viewpoints, and on the SSD latency front as well [3, 8–15] since storage is a dominant component of many client-facing applications. This dissertation focuses on trimming the tail latencies of reads since reads in SSDs are more latency-sensitive, with its service times much more difficult to hide in the critical path compared to writes - the latter has more options such as write buffering, etc. [16, 17].

### 1.1.1 Uniqueness of Scheduling and Load Balancing for SSDs

The concept of parallelism, load imbalance, and scheduling, have been previously investigated in many other domains such as parallel and distributed systems, cloud computing systems, and virtualized environments. However, due to unique characteristics of SSDs, previous scheduling and load balancing mechanisms cannot be readily applied in the SSDs. Next, we discuss the popular scheduling techniques for parallel environments and explain why they are not suited to be employed in the SSDs.

**Evenly Distributing Load:** Examples of such schedulers are: Join Shortest Queue, Join Idle Queue, Round Robin, Randomized Round Robin and etc. Such
scheduling mechanisms primarily require the flexibility in re-directing SSD requests to idle or less loaded flash chips. Due to out-of-place updates in the SSD, writes are well suited to be re-directed to other flash chips. Reads, on the other hand, are restricted in terms of who can service them – flash chips that are holding the corresponding data. Thus, such techniques cannot be incorporated to schedule reads in the SSD.

Work Stealing: With this technique, idle flash chips can steal requests from congested ones to faster service requests and help balance the load. However, similar to the previous case, such technique is hampered by the lack of read re-direction opportunities in the SSDs.

Preemption: Migration-based preemption/cancellation of requests cannot be incorporated for read requests in the SSDs due to restrictions in read re-direction. Preempting an ongoing write/erase operation to service read requests, however, requires hardware enhancements in the SSD which has been already proposed by the prior works [10,18,19]. This technique, even though currently employed by SSD vendors, as a prior study [8] has shown cannot completely eliminate the large read latencies.

Load prediction and rate limiters: Such techniques are not well suited for SSD requests due to the following reasons: (i) background tasks, such as garbage collection, run non-deterministically and last for an arbitrary period. The occurrences and duration of such operations are impacted by various factors which embarrassingly complicates their predictability; (ii) wear-leveling mechanisms manipulate the logical-to-physical address mapping, hence complicating the prediction of load at each flash chip; and (iii) Execution of advanced commands as well as contention of requests on the shared channels, makes the prediction of response times non-trivial.

Request and Job size distribution-based: The disparity in different operations’ latency in SSDs hinders the adaption of such mechanisms. Moreover, latency of write and erase operations can vary over the time, as SSD controllers, depending on wear, may apply different voltages for these operations.

Partitioning and Clustering Based: Clustering and partitioning requests can help reduce resource contention. Prior work [20–22] employ such mechanisms in the SSDs to improve its performance. However, these solutions are typically heavy-weight, incurring high overheads, and/or introduce additional non-determinism in
requests’ service due to maintenance activities.

1.1.2 Dissertation Outline

The first part of this dissertation focuses on design and implementation of a scheduling mechanism that orders SSD requests when (i) they arrive at the SSD, and (ii) issuing them on flash chips, to minimize the response times of each request. The former exploits the skews between sub-request completion times (called slack), wherein new arrivals can jump ahead in their respective queues without affecting the response times of those already present. The latter, however, attempts to co-schedule sub-requests of a request when issuing them on flash chips, thereby achieving the time-sharing of the available resources. Layered under a state-of-the-art SSD request scheduler, the proposed scheduler estimates the slack of each incoming sub-request to a flash chip and allows them to jump ahead of existing sub-requests with sufficient slack. Later at the service time, the scheduler issues requests on the flash chips only if all the flash chips to service a request are available. These enhancements can improve the response time of some requests without affecting the response time of others.

Despite effectively reducing response times of SSD requests by coordinating them in their respective queues, the proposed scheduling mechanisms are restricted in terms of flash chips that can service a request. Next, we aim at providing redirection opportunity for SSD requests to opportunistically re-directing them to free flash chips rather than waiting for their turn in their respective queues. Due to out-of-place-update property of NAND flash, writes are inherently suited to be redirected to another flash chip which is opportunistically free. This dynamic re-direction is, however, nontrivial for read requests since reads are more restricted in terms of who can service them – flash chips holding the data. While blindly replicating all the data everywhere seems very promising from a read redirection perspective, doing so results in high space overheads, high write/replication overheads and lower endurance.

In the second part of this dissertation, we present novel approaches to achieve redirection for read requests. To this end, we propose a selective replication, wherein the popularity of data is used to figure out the “what”, “how much”, “where” and “when” questions for replication. Leveraging value locality/popularity, that is
often observed in practice [1,23–25], popular data is replicated across multiple chips to provide more opportunities for dynamic read redirection to less loaded flash chips. Such re-direction mechanism mainly relies on the content-addressable design of SSD. Furthermore, to achieve read re-direction in normal SSDs without relying on a content-addressable design, we leverage the existing RAID-like capabilities embodied inside high-end SSDs for performance and fault-tolerance, and while a long latency operation is ongoing, rather than waiting, the read can get its data by reconstructing it from the remaining chips of that group (including parity). However, this introduces additional reads, and we propose an adaptive scheduler that dynamically figures out whether to wait or to reconstruct the data from other chips.

1.2 Contributions

This dissertation specifically makes the following research contributions.

Addressing the Non-uniformity Within Requests (Chapter 3).

- We introduce the notion of slack between sub-requests of macro requests that are directed at different flash chips of an SSD. Across a diverse number of storage workloads, we show that considerable slack exists across the read and write sub-requests that can get as high as several milliseconds.

- The success of a slack-aware scheduler would very much depend on estimating the slack of sub-requests accurately. We present an online methodology for estimating the slack that is based on queue lengths, the type (write/read) of requests, and contention for different hardware resources. We show that our methodology yields fairly accurate slack estimates across these diverse workloads. Even in the very few cases, where the errors are slightly higher, Slacker still provides response time benefits.

- Recognizing the hardness of the problem, we propose a simple heuristic to implement a slack-aware scheduler. Rather than a coordinated global re-arrangement of the distributed queues for each flash chip, which can result in a lot of communication/overheads, our heuristic takes a sub-request at each queue independently and figures out whether to jump ahead of existing ones
based on their slack. The resulting complexity is just linear in the number of queued sub-requests for each chip.

- We further augment our scheduling mechanism with service-time decision making, and present Slacker++, which attempts to co-schedule SSD requests. Such proposed co-scheduling is further combined with a backfilling technique to service requests on unutilized flash chips. With these enhancement, SSD requests can achieve time-sharing of available flash chips, and be more effectively serviced.

- We implement the proposed scheduling mechanisms in SSDSim, together with state-of-the-art scheduler (e.g., O3 scheduler [7]) enhancements that take advantage of SSD advanced commands. We show that request response time is improved by 21%, 16% and 9.65% on the average for write-intensive, read-intensive, and read-write balanced workloads (with improvements up to 27%, 27% and 24%), respectively.

**Content Popularity-based Selective Replication for Read Redirection in SSDs (Chapter 4).**

- We study the resource contention problem in the SSD and show the significance of delays due to queueing for service from individual NAND-flash chips that can take dozens/hundreds of microseconds to perform the read/write operations.

- We investigate the challenges of read redirection in modern SSDs and analyze the overheads associated with simply replicating data on all flash chips. We show that, blindly replicating all the data everywhere despite being very promising from a read redirection perspective, results in high space overheads, high write/replication overheads and lower endurance.

- Leveraging value locality/popularity, that is often observed in practice, popular data is replicated across multiple chips to provide more opportunities for dynamic read redirection to less loaded flash chips. We specifically try to answer the “what”, “how much”, “where” and “when” questions for replication.

- Using extensive workload traces running over weeks from real systems, we show that our Read Redirected SSD (RR-SSD) can provide up to 45%
improvement in read performance, with average improvement of 23.9%, and up to 40% improvement when considering both read and write requests, with 16% improvement on average.

Trimming the Tail for Deterministic Read Performance in SSDs (Chapter 5).

- We systematically investigate the occurrences of long tails on a state-of-the-art high-end SSD, by studying the entire I/O path. We show that contrary to a lot of prior observations, Garbage Collection is not a key contributor, and it is more the variances in queue lengths across the flash chips that is the culprit. Particularly, we show that reads waiting for long latency writes, which has been the target for much of our study, is at the root of this problem.

- We develop a simple yet effective solution that leverages existing RAID groups into which flash chips are organized. While a long latency operation is ongoing, rather than waiting, the read could get its data by reconstructing it from the remaining chips of that group (including parity).

- To overcome the additional reads required for reconstruction, we propose an adaptive scheduler called ATLAS that dynamically figures out whether to wait or to reconstruct the data from other chips.

- The resulting ATLAS optimization cuts the 99.99th percentile read latency by as much as 10X, with a reduction of 4X on the average across a wide spectrum of workloads.
Chapter 2
Background

2.1 SSD Architecture

Modern SSDs [26–28] have several NAND flash chips and control units interconnected by I/O channels (Figure 2.1). Each channel is shared by a set of flash chips and is controlled by an independent controller. Each chip, internally, consists of several dies and a number of planes within each die. Each die is further consisted of a number of blocks, with each block containing multiple SSD pages. A page is the smallest unit for SSD operations (e.g., read and write). The granularity of Read and Program (Write) operations is a page, with the latter typically 10–40 times slower than the former.

2.2 Flow of a Request within the SSD

When the host sends an I/O request, the host interface picks it up and inserts it into a device queue for processing. Since the macro request may span several pages, the host interface parses each request into several page-sized transactions, each with a specific Logical Page Address (LPA). In NAND flash, writes are never done in place. Consequently, mapping tables need to be maintained to keep track of the current location of a page, that is referred to as a Logical Page Number (LPN). LPN is then allocated to flash chips at a specific Physical Page Number (PPN). Translating an LPN to a PPN is accomplished in 2 steps: (i) determining the plane where the block/page resides amongst the numerous choices, and (ii) the eventual physical location of the block/page within that plane. While a single
table to accomplish both steps would allow a complete (and possibly dynamic) write-redirection mechanism with full flexibility for placing any page at any location across the flash chips, this takes additional space. Instead, commercial SSDs use a static mapping scheme to determine the chip, die and plane of each LPN, which can be accomplished by simple modulo calculations (instead of maintaining mapping tables). Once the plane is determined by this mechanism, a Flash Translation Layer (FTL) maps the page to a PPN within that plane using a table (a page-level mapping table here again offers maximum flexibility at the overhead of extra space).

After address translation, the FTL Scheduling Unit (FSU) which is part of the FTL firmware, resolves resource contention and schedules requests for maximizing parallelism across the different hardware entities [7,29]). FSU subsequently uses the well-known First Read-First Come First Served (FR-FCFS) with conditional queue size [30] to order the sub-requests at each chip. FR-FCFS is designed to lower the impact of high write latency on reads. To do so, the scheduler maintains a separate Read Queue (RDQ) and Write Queue (WRQ) for each chip, where WRQ is much larger than RDQ.

Figure 2.1: A generic architecture of modern SSDs.
2.3 Garbage Collection

Writing into a page requires an *erase* operation beforehand. Erase operation takes significantly longer than program operation to complete. The erase-before-write property necessitates out-of-place updates in order to prevent the high latency of erase deteriorating the write performance. Erase is performed at the granularity of a *block*. The out-of-place updates result in invalidation of older versions of pages requiring Garbage Collection (GC) to reclaim certain amount of invalid pages in order to create room for new writes. During GC, the selected block is erased after relocating its valid pages to new pages. GC is invoked periodically to reclaim the invalid pages. Another characteristic of flash memories is limited endurance in terms of number of erases a block can tolerate before it fails. Each block typically has a lifetime of 1K–10K depending on cell density and fabrication technology. Wear-leveling (WL) techniques [31–33] are usually employed to maintain similar lifetime for all of the blocks.
Chapter 3  
Exploiting Non-uniformity Within Requests’ Service Time

In this chapter, we address the non-uniformity in SSD requests service time to improve response time of SSD requests, thereby improving the overall system performance. To this end, we propose and develop a novel scheduling mechanism, called Slacker, which first estimates response time of sub-requests when each request arrives at SSD. It subsequently calculates the slack of each sub-request, and leverages these slacks to significantly reduce response times. Though intuitive, there are a number of practical considerations – estimating the slack, figuring out which requests to bypass, developing an elegant implementation that is not very complex (since the high level problem is NP-hard) and avoiding extensive communication across the hardware components – which need thorough investigation. Besides the arrival-time scheduling decisions, the proposed scheduling mechanism leverages the time-sharing of available resources when issuing requests on flash chips by co-scheduling all sub-requests of the next-in-line request on available flash chips. Although the high-level problem is NP-Hard, we employ a simple heuristic through which, when a number of flash chips are available, the scheduler picks the first request in the queue which can all be serviced by the available resources. Combining the arrival-time scheduling decisions with service-time ones, we develop Slacker++, and show that our enhancements lead to significant reduction in the response time of individual SSD requests.
3.1 Non-uniformity in Requests’ Service

This section introduces the concept of slack, its origin, and how it can be exploited for performance benefits.

3.1.1 What Is Intra-Request Slack?

An I/O request’s size varies from a few bytes to KBs (or MBs). When a request arrives at the SSD, the core breaks it into multiple sub-requests and distributes them over multiple flash chips so that they can get serviced in parallel. Since service of a request is not complete until all its sub-requests are serviced, the sub-request serviced last, called critical sub-request, determines the request’s eventual response time. For each sub-request, we define slack time as the difference between the end time of its service and the end time for the critical sub-request in the same request. Essentially, the slack of a sub-request indicates the latency it could tolerate without increasing the overall latency of the corresponding request.

Figure 3.1.(a) gives an example. The SSD consists of four chips, CHIP1, CHIP2, CHIP3 and CHIP4, where CHIP1 is currently idle while the other three are servicing requests of 7, 4 and 1 sub-requests respectively. Let us assume that servicing a sub-request takes 1 cycle. At time $t_0$, Request A arrives that has sub-requests $A_0$, $A_1$ and $A_2$ each that are in turn mapped to CHIP-1, CHIP-2 and CHIP-3, respectively. With FR-FCFS scheduling, even though $A_0$ can get serviced right away, $A_1$ and $A_2$ have to wait 7 and 4 more cycles, respectively, to get their turns. As a result, $A_0$, $A_1$ and $A_2$ have slacks of 7, 0 and 3 cycles, respectively.

Figure 3.2 plots the cumulative distribution function (CDF) of slack across sub-requests of several workloads on a 4 channel SSD with 16 flash chips. One can see from these results that, over 50% of the sub-requests have tens of milliseconds of slack. While one could try reducing this slack, this work examines the more interesting possibility of leveraging this slack for better scheduling.

3.1.2 Why Does Slack Arise?

There are two main causes of slack. First, as can be seen in Figure 1.1, a majority of the requests are relatively small, with their pages spanning just a few chips leading to lower chip-level parallelism. Large requests, on the other hand, could
span all the chips at the same time, leading to higher (flash) chip-level parallelism. Even though several small requests could be serviced at the same time by the higher parallelism offered by the hardware, it is not necessary that these requests be disjoint in the chips that they exercise. This can lead to some sub-requests of a request having to wait their turn for their respective chips while their sibling sub-requests of the same request are being serviced at other chips. The consequent load imbalance across the chips is evident when we observe the average, maximum, and minimum values for the number of sub-requests in request queues of each chip waiting to be serviced in Figure 3.3. At any instant, the load varies significantly –
sometimes the maximum queue length is 72 times larger than the minimum, which in turn can skew the end times of the sub-requests (within a request) directed to different chips. The second reason for the non-uniform end times of sub-requests is the widely differential latency of the basic flash operations: read, write, and erase. Since not all chips are necessarily doing the same operation at the same time, the sub-requests of a request may have different waiting times even if they are all next in line at their respective queues.

3.1.3 How to Get the Most out of this Slack?

In the presence of slack for sub-requests within a request, the request service time is determined by the completion of the critical sub-request. One can tackle slack in two ways:

- **Exploit Slack:** In the first part of this work, we explore how to leverage any existing slack between sub-requests to improve response time. The basic idea is to identify sub-requests in the request queue with high slack and de-prioritize them to benefit some others. Such re-ordering is done without impacting the waiting times of those being de-prioritized by leveraging knowledge of their slack. For instance, consider Figure 3.1.(a) where the baseline (slack-unaware FR-FCFS) scheduler prioritizes $A_2$, because of its earlier arrival, thereby delaying $B_0$. In contrast, if the scheduler is aware of the sub-request slacks, it would prioritize $B_0$ over $A_2$ as the latter can be delayed without affecting the
response time of $A$. Figure 3.1.(b) shows this. Doing so reduces the response
time of Request $B$ without increasing the response time of Request $A$ (its
critical sub-request, $A_1$, remains unchanged), thereby improving the overall
SSD response time.

- **Reduce slack**: Besides exploiting whatever slack is available, one can try
to mitigate the slack itself by reducing the slack. Reducing slack, in general,
requires finding time slots where all requisite chips for a request are free,
akin to gang scheduling. Waiting for such time slots can lead to significant
under-utilization as has been well studied [34]. SSDs, however, offer some
unique opportunities for reducing slack within write requests owing to their
“no-write-in-place” property, i.e., if the chip is busy, the corresponding write
sub-request could be directed to some other chip that is opportunistically
free. Techniques such as Dynamic Write Mapping [35–37] and Write Order
Base Mapping [38] could be employed to perform such re-direction, which
could lower slack within write requests. A significant drawback with such
write re-direction is that additional mapping tables need to be maintained
to re-direct a subsequent request to the appropriate channel, chip, die and
plane where it has been re-mapped, i.e., a static strategy such as CWDP
which does not require such a table can no longer be employed. Consider
for example a page-level mapping for an SSD with configuration in Table 3.1
used in this work which has 1TB capacity, 16 flash chips, 4 dies per flash chip
and 2 planes per die. Dynamic re-direction requires maintaining 7 bits per
page (4 bits for chip number, 2 bit for die and 1 bit for plane). With 8KB
sized pages, we need $\frac{1TB}{8KB} = 2^{27}$ entries, each of 7 bits, putting the additional
storage requirement at 117MB which is 23% of the 512MB RAM capacity on
state-of-the-art SSDs. Already, the on-board memory storage is very precious,
and needs to be carefully rationed amongst the different needs – caching data
pages, caching FTL translations, temporary storage for garbage collection,
etc. Sacrificing a fourth of this capacity (or adding the required capacity)
can be a significant overhead. Additionally, re-direction is not an option for
reads. Prior work [35,37] has shown that write re-direction can actually hurt
reads which are usually in the critical path. Instead of attempting to reduce
the slack itself – which can lead to under-utilization of available resources,
later in this work, we aim at co-scheduling the sub-requests of a request on
We propose Slacker, that makes slack-aware scheduling decisions at FSU to improve response time. The main idea of Slacker is to identify how much the service for the sub-requests waiting in the queue can be delayed (based on their slack) to accelerate the service of newer sub-requests upon their arrival.

### 3.2.1 Reordering

Slacker works on a generic SSD architecture illustrated in Figure 2.1 that uses FR-FCFS algorithm at FSU. Reordering sub-requests waiting in the queue, using their slack time, is performed by de-prioritizing sub-requests already ordered by FR-FCFS. As FR-FCFS is composed of two prioritization rules, read-first and oldest-first, de-prioritization intuitively has two dimensions: relaxing the read-first order and relaxing the oldest-first order.

**Relaxing read-first order.** If the slack of a read sub-request in RDQ is larger than write latency, this slack can be used by a write sub-request (in WRQ) to be serviced ahead of that read sub-request. However, we do not expect significant improvement in response time of such writes due to two reasons: (1) by analyzing a wide range of workloads, we observed that the average slack seen by a read sub-request is typically much lower than the flash write latency (slack of microseconds compared to write latency of hundreds of microseconds to milliseconds, see Table 5.2 in Section 3.4), and (2) this approach can at best reduce the response time of a write sub-request by a time equal to that of a read latency, which is much smaller.
Hence, this relaxation is not likely to provide meaningful benefits.

Relaxing oldest-first order. Oldest-first policy in FR-FCFS performs FIFO scheduling in RDQ for read sub-requests and in WRQ for write sub-requests. With this relaxation, we propose reordering requests in each of the queues independently. If a read sub-request has slack, we use this slack for allowing other reads in RDQ to bypass this one. We call this scheme, Read Bypassing (Rbyp) which can improve read response time. Similarly, slack of a write sub-request is only used to accelerate service of other write sub-requests in WRQ, and is referred to as Write Bypassing (Wbyp).

3.2.1.1 Reordering Algorithm

At a high level, the scheduling of sub-requests across the RDQs and WRQs of flash chips can be posed as a two-dimensional constrained bin-packing problem – filling in the time slots with sub-requests directed at each flash chip, so that average response times of requests can be minimized, as shown in the matrix in Figure 3.4. While it may be advantageous to co-schedule sub-requests of a macro request in the same time slot (same row in Figure 3.4) to avoid slack and having to wait for the last sub-request to complete (similar to Gang scheduling of parallel processors [39]), such restrictions may fragment rows of this matrix, leading to under-utilized slots. Opportunistically using such slots, without being restricted to co-scheduling, would improve the utilization but can result in the slack problem that has been discussed until now. Regardless, this two-dimensional constrained bin-packing problem is NP-hard [40]. A brute-force evaluation of all permutations at each request arrival in the flash HIL/FSU would be highly inefficient (the matrix contains several dozen rows and columns). Further, since each FSU maintains its own queues, coordinatedly permuting the entries across all these distributed queues can incur tremendous overheads. Instead, Slacker employs a simple heuristic that can be implemented within each FSU to order its request queues independent of the queues of the other FSUs. Also, each FSU takes only $O(N)$ time for inserting a sub-request in its queue (i.e., the column in the matrix of Figure 3.4, where $N$ is the current queue length).

Upon a request arrival, each FSU has little information on the queues of other FSUs to try and co-schedule its assigned sub-request with its sibling sub-requests at other FSUs in the same time slot. Having discussed earlier, queue lengths can vary
Figure 3.6: Examples for (a) Rbyp (RB and RC finished 1 cycle earlier without impacting RA and WA), (b) Wbyp (WC finished 4 cycles earlier without impacting WA and WB), and (c) WP (RA and RB finished 1 and 3 cycles earlier without impacting WA and WB). RA, RB, and RC are read requests, WA, WB, and WC are write requests. RAi is the i\textsuperscript{th} sub-request in RA.

widely across the chips at any instant and hence, simply adding the sub-request at the tail of each FSU queue as in FR-FCFS can lead to wide slacks without availing the flexibility that such slacks allow in scheduling. Instead, in Slacker, each FSU individually examines whether each incoming sub-request can jump ahead of the sub-requests already waiting in its queue, starting from the tail. It can jump ahead as long as the slack of the waiting sub-request is higher than the estimated service time of this new sub-request (i.e., delaying the waiting sub-request by servicing the incoming one will not delay the overall response time of the request it belongs to). As it jumps ahead of each sub-request, their slack is accordingly reduced. The new sub-request is inserted in the queue position, where it cannot jump ahead any more. After the sub-requests of the new macro request are inserted in the respective FSU queues, their slacks can be estimated/computed (slack estimation is explained later). This mechanism is illustrated in Figure 3.5 for an existing queue, with an incoming stream of 3 new sub-requests. Since the incoming sub-request would at best jump over \( N \) waiting sub-requests in the queue, the work at each FSU is \( O(N) \).
3.2.1.2 Examples of Rbyp and Wbyp

Figure 3.6.(a) shows how Rbyp improves performance with an example. It depicts the RDQ in baseline (1) and a system with Rbyp (2). In this example, read request \( RA \) has two sub-requests (\( RA1 \) and \( RA2 \) with slack values of 2 and 0), read request \( RB \) has two sub-requests (\( RB1 \) and \( RB2 \) with slack values of 1 and 0), and read request \( RC \) has two sub-requests (\( RC1 \) and \( RC2 \) with slack values of 1 and 0). At time \( t_3 \), when \( RB \) and \( RC \) arrive, Rbyp pushes \( RB2 \) and \( RC2 \) forward (both had 0-cycle slack before reordering) and pushes \( RA1 \) back (previously had 2-cycle slack). As a result, the response times of requests \( RB \) and \( RC \) improve while the response time of \( RA \) remains unchanged (since the response time of its laggard sub-request did not change). Hence, Rbyp can improve the response time of a read request if there exists enough slack in other requests.

Figure 3.6.(b) shows the possibility of performance improvements with Wbyp using an example. The status of WRQ in baseline and a system with Wbyp are shown in (3) and (4), respectively. In this scenario, we have three write requests. \( WA \) is a single-page request that has no slack. \( WB \), on the other hand, has three sub-requests: \( WB1 \), \( WB2 \) and \( WB3 \) with response times of 9, 5 and 4 cycles, respectively and slacks of 0, 4 and 5. \( WC \) is also a single-page write request with response time of 8 and slack of 0. \( WC1 \) can be serviced earlier in Wbyp than the baseline by prioritizing it over \( WB2 \), resulting in 4 cycles faster response time for \( WC \). Note that, the response time of request \( WB \) does not get worse as the service of \( WB2 \) is delayed at most by its slack.

3.2.2 Slack-aware Write Pausing

Until now, similar to prior scheduling proposals, our optimization have only looked at requests in the queues without pre-empting sub-requests already being serviced. Going a step further, it is possible that even sub-requests that have started being serviced could have slacks and thus become a candidate for reordering. Utilizing this slack, the controller may decide to cancel service of the currently being processed sub-request to favor an incoming request. However, a simple cancellation midway through service would throw away a lot of the accomplished work. Instead, we look into options for pre-emption of the request being currently serviced in favor of another sub-request in the queue and then restart the canceled sub-request in
later cycles. In effect, we are simply advancing the re-ordering algorithm described earlier by one more step - to include the sub-request currently being serviced.

Pausing a read sub-request, in favor of another read or a write sub-request, is not expected to be beneficial as both read latency and slack of a read sub-request are small. With higher write latency, there may be merit to pre-empt ongoing writes in favor of other reads. Pre-empting a write to service another write is not possible in current hardware\(^1\), and is also not expected to provide significant benefits either since both operations take equally long time. Instead, we only consider pre-empting an ongoing write to service incoming read sub-requests if the former has sufficient slack. We adapt a previously proposed write pausing (WP) technique [18] for our slack-aware reordering. In [18], the authors have shown that the write implementation in modern flash technologies has features that can be leveraged for pausing. The first feature is that read operation does not modify write point [41]. So after reading a page, the write point still refers to the place of the paused write, and we can thus serve the read in the middle and resume the write later. The second feature comes from the programming algorithm, *write-and-verify* technique [42]. As the name suggests, a write-and-verify programming technique consists of applying a programming pulse followed by a read stage which is used to decide whether to stop or to compute the new programming signal to be sent. It is possible to pre-empt an on-going write at the end of each such iteration for servicing a pending read sub-request. Therefore, at each potential pause point, the scheduler checks if there is a pending read request, service it, and resume the write from the point it was left based on the slack of the ongoing write.

Figure 3.6.(c) shows the possibility of improving read response time via write pausing using an example. In the baseline (5), there are two writes: one (WA) with no slack and one (WB) with 4-cycles of slack for two of its sub-requests. With WP (6), the controller cannot pause WA1 for sub-request RA2, but WB3 is paused for two cycles to enable the service of RA1 and RB1, thereby reducing the read latency of RA and RB, without affecting the response time of WB.

To accomplish reordering and pausing, it is of utmost importance to accurately quantify the slack of each sub-request. If the slack is not accurate, the proposed enhancements could delay some sub-requests by more than their actual slack, increasing their response times. Despite knowing the queue positions of all sub-

\(^1\) Writes in flash are sequential, and the write point [41] gets lost.
requests and reasonable estimates of response times of sub-requests before them in an isolated setting, precise estimation is difficult due to non-deterministic and unpredictable behavior of requests contending for shared resources (channels and chips), utilizing advanced commands [43], performing GC, and arrival of future read requests that can get prioritized over writes. To overcome this inaccuracy, we propose a stochastic model working in an online fashion to approximate response time of each sub-request of a request. Having estimated response times of each sub-request of a request, it is straightforward to calculate its slack time by subtracting its response time from the critical sub-request response time. This is calculated as soon as all the sub-requests of an incoming request are inserted into their respective queues availing of information about sub-requests that are ahead in the queues.

3.2.3 Estimating Slack

Calculating response time. The response time of a read/write sub-request, \( T_{RD/WR}^{Response} \), is composed of the wait time seen by the sub-request in a RDQ/WRQ, \( T_{RD/WR}^{Wait} \), and its own service time, \( Latency_{RD/WR} \). Therefore, we have:

\[
T_{RD/WR}^{Response} = T_{RD/WR}^{Wait} + Latency_{RD/WR} \tag{3.1}
\]

Obtaining accurate estimates for \( T_{RD/WR}^{Wait} \) is challenging because (1) FCFS is violated as the new incoming read sub-requests can bypass existing write requests, (2) performing advanced commands affects our estimated time by servicing multiple requests at the same time, (3) performing GC, and (4) there is potential interference between command and data messages sent to different flash chips sharing a channel. The wait time for a sub-request consists of four parts: the first is the delay due to actual queuing latency, \( T_{RD/WR}^{Queue} \); the second is the average stall-time of the chip due to GC, \( T_{RD/WR}^{GC} \); the third is the blocking time over the channel, \( T_{RD/WR}^{Interference} \); and the fourth is the time remaining to complete the ongoing operation on the chip, \( T_{RD/WR}^{Residual} \). Therefore we can write:

\[
T_{RD/WR}^{Wait} = T_{RD/WR}^{Queue} + T_{RD/WR}^{GC} + T_{RD/WR}^{Interference} + T_{RD/WR}^{Residual} \tag{3.2}
\]

Below, we calculate each of these components.

Estimating queuing latency. The scheduler always prioritizes read requests

22
over writes. Then, the queuing latency for a read sub-request, $T_{RD}^{Queue}$, is different from that for a write sub-request, $T_{WR}^{Queue}$. $T_{RD}^{Queue}$ is the sum of the service times of all preceding read entries in RDQ, and $T_{WR}^{Queue}$ is the sum of the service times of all read sub-requests in RDQ and the preceding write sub-requests in WRQ. So:

\[
\begin{align*}
T_{RD}^{Queue} &= Num_{RDQ} \times \text{Latency}_{RD} \\
T_{WR}^{Queue} &= T_{RD}^{Queue} + Num_{WRQ} \times \text{Latency}_{WR}
\end{align*}
\] (3.3)

where $Num_{RDQ}$ and $Num_{WRQ}$ refer to the number of read and write entries in the request queues, respectively. $Num_{RDQ}$ and $Num_{WRQ}$ are obtained after the sub-requests have been inserted in the appropriate queues by the scheduler. On servicing a request, modern controllers check the possibility of employing multi-plane or multi-die operations. This affects the queuing latency of a sub-request by reducing the service times of the requests scheduled earlier. To measure the effect of advanced operations, the controller keeps four counters per-chip: $Cntr_{RD}$, $Cntr_{WR}$, $Cntr_{ARD}$, and $Cntr_{AWR}$. The controller increases $Cntr_{RD}$ and $Cntr_{WR}$ when it services a read sub-request or a write sub-request, respectively, and increments $Cntr_{ARD}$ and $Cntr_{AWR}$ when it commits $N$ read or write requests, respectively, by an advanced command. Assuming for now that the controller knows the values of these counters for each chip, it calculates the probability of executing advanced commands in RDQ or WRQ as $Pr_{Adv}^{RD \ or \ WR} = \frac{Cntr_{ARD \ or \ AWR}}{Cntr_{RD \ or \ WR}}$. So we update Eq. 3.3 as:

\[
\begin{align*}
T_{RD}^{Queue} (New) &= T_{RD}^{Queue} (Old) \times ((1 - Pr_{Adv}^{RD}) + Pr_{Adv}^{RD} / N), \\
T_{WR}^{Queue} (New) &= T_{RD}^{Queue} (Old) + T_{WR}^{Queue} (Old) \times ((1 - Pr_{Adv}^{WR}) + Pr_{Adv}^{WR} / N)
\end{align*}
\] (3.4)

**Estimating chip stall-time due to GC.** When a chip is busy with GC, it cannot service any other request. GC latency has two parts: (1) the latency of moving valid pages, $T^{Move}$; and (2) the erase latency, $Latency_{Erase}$. The former changes over time and the controller keeps two counters per chip, $Cntr_{Move}$ and $Cntr_{ER}$, to compute it as $T^{Move} = (Cntr_{Move}/Cntr_{ER}) \times (Latency_{RD} + Latency_{WR})$. $Cntr_{Move}$ and $Cntr_{ER}$ count the total number of page movements during GC and the number of erases, respectively. Thus, $T^{GC}$ can be estimated as:

\[
T^{GC} = Pr^{GC} \times (T^{Move} + Latency_{Erase})
\] (3.5)
where $P_r^{GC}$ is the probability of executing GC, and is computed as $P_r^{GC} = \frac{Cntr_{RD} + Cntr_{WR}}{Cntr_{RD} + Cntr_{WR}}$. Estimating interference latency. When a read or write sub-request is sent over the channel to a chip, it keeps the channel busy for $T_{Xfer}$ cycles (for a page size of $P$ bytes and a channel width of $W$ bytes, $T_{Xfer} = \frac{P}{W \times 333 \text{MT/s}}$ for ONFi 3.1). During this time, the FSU is not able to schedule a command to any other chips on that channel, even though several commands might be ready to be scheduled. Hence, $T_{\text{Interference}}$ for each sub-request can be estimated as:

$$T_{\text{Interference}} = E[\#\text{ReadyRequests}] \times T_{Xfer}$$

(3.6)

where $E[\#\text{ReadyRequests}]$ is the average number of sub-requests contending for the same shared channel at the same time. The controller maintains two counters per-channel, $Cntr_{Ready}$ and $Cntr_{Total}$, and computes $E[\#\text{ReadyRequests}] = \frac{Cntr_{Ready}}{Cntr_{Total}}$. $Cntr_{Ready}$ is incremented when a command is ready to be issued but stalled due to a busy channel. $Cntr_{Total}$ counts the total number of sub-requests mapped to the flash chips connected to the channel.

Calculating residual time. Upon arrival of a new flash request, the residual time of the current operation to be completed in the target chip is calculated as:

$$T_{\text{Residual}} = \overline{\text{Flag}_{RD/WR}} \times \text{Latency}_{RD} + \text{Flag}_{RD/WR} \times \text{Latency}_{WR} - (T_{\text{Now}} - T_{\text{Start}})$$

(3.7)

$\text{Flag}_{RD/WR}$ and $T_{\text{Start}}$ are attributes of the current operation on the chip: $\text{Flag}_{RD/WR}$ is per-chip flag determining type of the operation ("0": read and "1": write); $T_{\text{Start}}$ is a per-chip register holding the start time of the operation.

Calculating slack time. After estimating the response times of the $N$ sub-requests of a macro request, the slack for the $i^{th}$ sub-request, $T_i^{\text{Stack}}$, can be calculated as:

$$T_i^{\text{Stack}} = \max(T_1^{\text{Response}}, T_2^{\text{Response}}, \ldots, T_N^{\text{Response}}) - T_i^{\text{Response}}$$

(3.8)

where $T_i^{\text{Response}}$ is response time of $i^{th}$ sub-request from Eq. 3.1.
3.2.4 Accuracy of Estimation

Slack estimation primarily depends on $T^{Response}$ of sub-requests. To verify the accuracy of our $T^{Response}$ estimation described above, we compare the estimates with actual response times of requests in a number of workloads. As can be seen in Figure 3.7, our estimates are quite accurate, with errors less than 1% for a number of workloads. Even in the few cases where the errors exceed 5%, we will show that our solution is still able to improve response times\(^2\).

3.2.5 Hardware Support and Latency Overhead

Slack estimation described above, requires additional counters, registers and flags. For each chip, eight up/down counters ($Num_{RDQ}$, $Num_{WRQ}$, $Cnt{rd}$, $Cnt{wr}$, $Cnt{RD}$, $Cnt{WR}$ and $Cnt{RW}$ and $Cnt{RW}$), a flag ($Frq{RD/WR}$), and two registers for $T^{Start}$ and $T^{Now}$ are needed. In addition, for each channel, two counters ($Cnt{Ready}$ and $Cnt{Total}$) are needed. This information should be maintained at the controller (at FSU) and the imposed overhead is relatively small, taking only a few additional bytes to provide the requisite information.

In our Slacker implementation, the latency overheads of slack estimation and re-ordering is around 100 cycles that becomes less than 1us with SSD configuration in the next section; i.e., negligible compared to the read and write latencies.

3.3 Co-scheduling of SSD Requests

Slacker, despite effectively reducing the response times of SSD requests across evaluated workloads (by up to 19% as will be shown in Evaluation Section of this chapter), its benefits are limited by some constraints (listed below), thereby not being able to achieve higher benefits in some workloads:

- **Uncoordinated jumps**: since re-ordering decisions are made separately for each sub-request (to keep the re-ordering algorithm and its implementation

\(^2\)The estimation of wait time in Eq. 3.2 may still have inaccuracies. Using an error term, the wait time of a sub-request can be updated as: $T_{RD/WB}^{Wait}(New) = T_{RD/WB}^{Wait}(Old) + T(Err)$, where $T(Err)$ is average time difference between the actual wait time and the estimated wait time for all the requests serviced in the last second (i.e., a moving average of the estimation error). With this error term, we experienced insignificant reduction in estimation error (less than 1%) compared to Figure 3.7. Thus, we did not consider it in our model.
inexpensive and straightforward), this can lead to a situation where a critical sub-request cannot jump ahead in its respective queue while its sibling sub-requests jump ahead of others. In such cases, even though some sub-requests of a request bypass others in their respective queues, the overall request response time is not affected (reduced) since the critical sub-request cannot jump ahead of others. As a consequence of these unnecessary jumps by non-critical sub-requests, the available slack is unnecessarily used, and not sufficient slack will be left for the subsequent critical sub-requests of incoming requests.

- **Lack of Service Time Scheduling Opportunities**: One limitation of Slacker arising from its arrival-time decision making, is the under-optimal service-time scheduling decisions. Slacker makes its scheduling decisions once inserting requests in the queues and do not touch them, unless some other sub-requests bypass them. This can lead to service of a non-critical sub-request while a critical sub-request is waiting in the same queue (that is not the next in line). If a critical sub-request cannot jump ahead of others (since the available slack is already used by a non-critical jump), the non-critical sub-request will be serviced, which can be detrimental to the success of Slacker.

- **Increasing the Min-Max Completion Times**: Due to the two restrictions associated with Slacker that are discussed above, it is likely for the critical sub-request be stuck in its respective queue, while its non-critical siblings
can jump ahead of others, which can, in turn, even worsen the differences in minimum and maximum completion times. Thus, Slacker may unintentionally exacerbate the skews in the completion times of sub-requests of a request, belittling its overall benefits.

Later in the evaluation section of this chapter, we will examine and analyze the above constraints of Slacker. To enhance our proposed scheduling mechanism, and address its shortcomings, we augment Slacker with service-time scheduling decisions to it, thereby leading to even higher benefits. The proposed mechanism, called Slacker++, attempts to use the available flash chips more effectively when issuing requests, by co-scheduling the sub-requests of a request, if possible. Adding the service-time scheduling dimension to Slacker can alleviate the restrictions of Slacker since it does not confine requests (especially the critical ones) to wait their turns in the queues. Co-scheduling all sub-requests of a request can also lead to less variation in the completion times within a requests, thereby mitigating the slack itself.

---

**Figure 3.8:** Examples for (a) pure co-scheduling without using empty time slots (flash chips remain idle if the next request cannot be completely serviced), (b) co-scheduling with using empty time slots (the baseline system in our studies – RB finished 1 cycle earlier without impacting others), and (c) co-scheduling and backfilling (RB finished 2 cycles earlier without hurting response time of RA). RAi is the i\(^{th}\) sub-request in RA.
3.3.1 Slacker++ Mechanisms

To overcome the challenges associated with Slacker, we next explore different service-time scheduling mechanisms in our design, in addition to our proposed reordering mechanisms:

**Co-scheduling without Using Empty Time Slots**: This is akin to perfect gang scheduling where sub-requests of a request are only scheduled if all of their target flash chips are available. Thus, even if a subset of sub-requests of a request can be serviced by available flash chips, they need to wait, hence, keeping the available flash chips idle (as shown in Figure 3.8.(a)). The consequent under-utilization of available resources leads to very poor performance. Thus, we opt out this scheme and do not consider it in the rest of our studies.

**Co-scheduling + Using Empty Time Slots (CS)**: This scheme is similar to previous one, with only difference that, the scheduler first attempts to co-schedule all sub-requests of a request, if possible. If all sub-requests of the next request in the line cannot be serviced due to chip unavailability, the available flash chips will service the next sub-request in the line (using empty time slots). This leads to better utilization of available resources and hence provides better performance. We give an example in Figure 3.8 part (b). As shown in this example, since all flash chips are not available to service request $RA$, Instead of keeping flash chips 1–3 idle (similar to part (a) of this figure), we use them to partially service $RA$. This avails time slots for servicing $RB$ one cycle earlier, thereby reducing its response time.

**Co-scheduling + Backfilling**: The backfilling technique [44] is used in combination with co-scheduling (CS). Backfilling technique tries to assign the unutilized flash chips to requests that are behind in the requests queue, rather than keeping them idle. We give an example of co-scheduling combined with backfilling (CS + Backfilling) in Figure 3.8.(c). As depicted in this example, combining the CS technique with backfilling leads to better utilization of available resources, thereby improving the overall response times of requests. When all flash chips to service the request $RA$ are not available, instead of partially servicing it – which does not help with its response time, we service the requests which are behind in their respective queues ($RB$ in this example). Thus, $RB$ is serviced 2 and 1 cycles earlier than parts (a) and (b), respectively.
Table 3.1: Main characteristics of simulated SSD.

<table>
<thead>
<tr>
<th>Evaluated SSD Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>4×4 Dimension (4 Channels and 4 Flash Chips per Channel), Channel Width = 8 bits, NAND Interface Speed = 333 MT/s (ONFI 3.1), Page Allocation Scheme = Channel-Way-Die-Plane (CWDP)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>NAND flash (Micron [45])</th>
</tr>
</thead>
<tbody>
<tr>
<td>Page Size=8KB, Metadata Size=448B, Block Size=256 pages, Planes per Die=2, Dies per Chip=4, Flash Chip Capacity=64GB, Read Latency=75μs, Typical Program Latency=1300μs, Worst Program Latency=5000μs, Erase Latency=3.8ms</td>
</tr>
</tbody>
</table>

3.4 Experimental Setting

Evaluation platform. We model a state-of-the-art SSD using SSDSim [35], a discrete-event trace-driven SSD simulator. SSDSim has detailed implementations of page allocation strategies and page mapping algorithms and captures inter-flash and intra-flash parallelism. It also allows studying different SSD configurations with multiple channels, chips, dies, and planes. The accuracy of SSDSim has been validated via hardware prototyping [35].

Configurations studied The baseline configuration consists of four channels, each of which is connected to four NAND flash chips. Each channel works on ONFi 3.1 [43]. Table 3.1 provides specifications of the modeled SSD (which is very similar to [46]) along with parameters of the baseline configuration. The evaluated SSD systems use modern protocols and schedulers at different levels: NVM-e [47] at HIL and an out-of-order scheduler [7] for parallelism-aware scheduling at FSU.

In our experimental analysis, we evaluate eight systems:

- **Baseline** uses FR-FCFS for micro scheduling at FSU.
- **Wbyp** uses our write bypassing scheme on top of FR-FCFS.
- **Rbyp** uses our read bypassing scheme on top of FR-FCFS.
- **WP** is a system with only write pausing on top of FR-FCFS.
- **Rbyp+WP** applies read bypassing and write pausing.
• **Slacker** is a system with both read and write bypassing, as well as write pausing.

• **Co-scheduling (CS) + Using Empty Slots** is a system that employs co-scheduling with using empty slots (this is equivalent to the baseline SSD system, thus we show it as the baseline in our evaluations)

• **CS + Backfilling** is a system that employs co-scheduling and Backfilling.

• **Slacker++** is a system with all enhancements including: read/write bypassing, write pausing, co-scheduling, and backfilling.

We report the amount of reduction in request response time (read and write) as the performance metric. The response time is calculated as the time difference between the arrival of the request at the host interface and the completion of its service.

**I/O workload characteristics:**

We use a diverse set of real disk traces: Online Transaction Processing (OLTP) applications [48] and traces released by Microsoft Research Cambridge [49]. In total, we study 20 disk traces to ensure that we cover a diverse set of workloads. Table 5.2 summarizes characteristics of our disk traces in terms of write-to-read-ratio ($WR-RD$), average/standard deviation of request sizes, and average/standard deviation of slack across the sub-requests within a request for the baseline system.

To better understand the benefits of our slack-aware scheduler, we categorize our disk traces into three groups based on their write intensity as it directly contributes to the efficiency of each proposed scheme. In our categorization, a disk trace is (a) **Write Intensive** if its write-to-read ratio is greater than 0.70; (b) it is **Balanced Read-Write** if its write-to-read ratio is between 0.30–0.70; otherwise, (c) it is **Read Intensive**.

### 3.5 Experimental Evaluation

In this section, we analyze the performance results for the three workload categories separately. For each, we present performance results (Figure 3.9) in terms of the percentage improvement in response times of all requests over the baseline system.
Table 3.2: Characteristics of the evaluated I/O traces.

<table>
<thead>
<tr>
<th>Trace</th>
<th>WR-RD</th>
<th>RD Size</th>
<th>WR Size</th>
<th>RD Slack (mSec)</th>
<th>WR Slack (mSec)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Ratio</td>
<td>KB</td>
<td>KB</td>
<td>Mean SD</td>
<td>Mean SD</td>
</tr>
<tr>
<td>Write Intensive Disk Traces</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>wdev2</td>
<td>0.99</td>
<td>13.4</td>
<td>16.2</td>
<td>0.0 0.0</td>
<td>144.8 205.4</td>
</tr>
<tr>
<td>rsrch1</td>
<td>0.99</td>
<td>25.6</td>
<td>18.7</td>
<td>0.1 0.1</td>
<td>155.9 183.2</td>
</tr>
<tr>
<td>prxy0</td>
<td>0.95</td>
<td>18.3</td>
<td>38.6</td>
<td>1.5 1.7</td>
<td>577.9 725.1</td>
</tr>
<tr>
<td>prn0</td>
<td>0.94</td>
<td>22.3</td>
<td>16.3</td>
<td>2.5 6.7</td>
<td>130.0 164.9</td>
</tr>
<tr>
<td>rsrch0-p</td>
<td>0.9</td>
<td>18.9</td>
<td>16.5</td>
<td>6.8 10.2</td>
<td>43.9 49.5</td>
</tr>
<tr>
<td>fin1</td>
<td>0.77</td>
<td>11.3</td>
<td>12.6</td>
<td>0.9 1.4</td>
<td>41.1 22.3</td>
</tr>
<tr>
<td>msnfs3</td>
<td>0.76</td>
<td>23.6</td>
<td>23.2</td>
<td>2.1 1.7</td>
<td>26.9 65.8</td>
</tr>
<tr>
<td>Balanced Read-Write Disk Traces</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>wdev0</td>
<td>0.7</td>
<td>16.8</td>
<td>17.1</td>
<td>2.6 2.2</td>
<td>63.7 109.8</td>
</tr>
<tr>
<td>src2-0-p</td>
<td>0.64</td>
<td>18.2</td>
<td>22.7</td>
<td>5.1 14.3</td>
<td>193.9 160.7</td>
</tr>
<tr>
<td>ts0-p</td>
<td>0.56</td>
<td>19.6</td>
<td>19.6</td>
<td>1.4 1.5</td>
<td>52.8 61.3</td>
</tr>
<tr>
<td>usr0-p</td>
<td>0.43</td>
<td>66.9</td>
<td>17.7</td>
<td>46.7 48.2</td>
<td>37.1 11.2</td>
</tr>
<tr>
<td>prn1-p</td>
<td>0.42</td>
<td>16.3</td>
<td>17.4</td>
<td>0.5 1.1</td>
<td>72.2 178.6</td>
</tr>
<tr>
<td>Read Intensive Disk Traces</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mds0-p</td>
<td>0.21</td>
<td>43.5</td>
<td>18.4</td>
<td>2.8 5.2</td>
<td>14.1 74.1</td>
</tr>
<tr>
<td>fin2</td>
<td>0.18</td>
<td>10.3</td>
<td>11.0</td>
<td>1.6 1.6</td>
<td>74.4 134.8</td>
</tr>
<tr>
<td>web0-p</td>
<td>0.18</td>
<td>46.9</td>
<td>16.6</td>
<td>1.3 1.7</td>
<td>15.7 43.2</td>
</tr>
<tr>
<td>rsrch2-p</td>
<td>0.08</td>
<td>12.0</td>
<td>12.2</td>
<td>1.3 1.4</td>
<td>67.9 161.2</td>
</tr>
<tr>
<td>usr1-p</td>
<td>0.06</td>
<td>56.3</td>
<td>14.5</td>
<td>30.9 29.4</td>
<td>36.1 93.9</td>
</tr>
<tr>
<td>hm1</td>
<td>0.05</td>
<td>22.9</td>
<td>27.8</td>
<td>2.2 4.9</td>
<td>35.1 17.9</td>
</tr>
<tr>
<td>stg1-p</td>
<td>0.02</td>
<td>68.5</td>
<td>15.7</td>
<td>0.4 0.9</td>
<td>4.5 3.2</td>
</tr>
<tr>
<td>proj4</td>
<td>0.005</td>
<td>32.8</td>
<td>18.4</td>
<td>0.3 0.4</td>
<td>6.4 2.7</td>
</tr>
</tbody>
</table>

3.5.1 Results for Write-Intensive Traces

Results for write-intensive traces is reported in Figure 3.9.(a), which are discussed below:

**Impact of Wbyp.** Wbyp is targeted to improve write request response times. We can see the consequent effect that yields between 3% to 19% improvement across all requests for these write intensive workloads. Workloads such as prxy0, wdev2, msnfs3, and rsrch1 provide high opportunities for leveraging Wbyp, allowing more incoming write sub-requests to jump ahead of ones with higher slack that are already in WRQ. These are also the workloads with higher overall response time improvements. At the same time, it should be noted that simply having higher
opportunities for Wbyp does not automatically translate to better performance. For instance, prn0 and fin1 give only 5% performance improvement, even though 50% and 20% of requests benefit from Wbyp in these respective workloads. Despite opportunities for reordering, the gains for each such reordering is relatively small.

Figure 3.9: Response time improvement results over the baseline system for (a) write-intensive, (b) read-write balanced, and (c) read-intensive traces.
in these workloads.

**Impact of Rbyp, WP and Rbyp+WP.** Both Rbyp and WP enhancements mainly target reads. As a result, in these write-intensive workloads, these enhancements do not provide significant improvements. Of these workloads, msnfs3 has the highest read intensity, and is the only case where any reasonable fraction of reads benefit from these enhancements. This results in a 3%, 6% and 7% (16% collectively compared to 10% with Wbyp) reduction in response times given by Rbyp, WP and Rbyp+WP, respectively, for msnfs3.

**Slacker.** The results for Slacker, which incorporates all 3 enhancements (Wbyp, Rbyp and WP) gives an interesting insight - one enhancement does not counter-act another, thereby all three can be collectively used to reap additive benefits. In the first 5 workloads, read enhancements (Rbyp and WP) are not providing any benefits, and Slacker defaults to Wbyp which provides good improvements. In the last 2 (can be visibly seen for msnfs3), Slacker’s benefits are additive over each of the individual improvements. Overall, Slacker gives between 3% to 19.5% (12% on average) reduction in total request response time compared to the baseline.

**Impact of CS and BF.** When employing service-time scheduling, we can achieve higher response time benefits, as it is evident in Figure 3.9.(a). CS+BF addresses
the restrictions of Slacker (uncoordinated and 1-D scheduling decisions), by scheduling all sub-requests of a request together, if possible, along with incorporating backfilling to service (on unutilized flash chips) the requests that are behind in their respective queues. Thus, sub-requests of a request do not necessarily need to wait for their turn when there is some availability. Using these techniques leads to higher benefit than Slacker, by up to 13% in rsrch1, and 6.15% on average. We further explore the reason behind the higher benefits for these techniques. As already discussed in section 3.3, there are cases where a non-critical sub-request of a request can be serviced while a critical sub-request of another request is waiting for its turn. Figure 3.10 reveals this inefficiency of Slacker by reporting how often a non-critical sub-request is serviced while a critical one is waiting for its turn. As shown in this figure, in a workload such as rsrch1, this percentage is very significant (around 40%), belittling the effectiveness of Slacker. Moreover, Amongst write-intensive workloads, prn0 and rsrch1 achieve higher benefits compared to Slacker. This is mainly due to large amount of slack in these two workloads (Table 5.2).

Slacker++ The results for Slacker++, which incorporates all proposed enhancements (bypassing, write pausing, and co-scheduling and backfilling) gives an interesting insight - one enhancement does not counter-act another, thereby all of them can be collectively used to reap additive benefits. However, as shown in the figure, most benefits of Slacker++ is driven by CS+BF mechanisms as a result of its service-time scheduling decisions which happen later and takes all sub-requests of requests into account – rather than individual scheduling decisions for each sub-request. Overall, Slacker++ yields 21% reduction in the response time of SSD requests on average, and by up to 27.26% in wdev2.

3.5.2 Results for Read-Write Balanced Traces

Results for read-write balanced traces are reported in Figure 3.9.(b).

Impact of Wbyp. With a lower write intensity in these workloads, the improvements with Wbyp are a little smaller than in the previous case. Still, Wbyp is able to reduce response time of all requests by 2% to 11% (6% on the average). Among the workloads, those with medium-sized requests such as wdev0, src2-0, ts0, and prn1, exhibit higher improvements (up to 11%). Note that with larger requests, there is higher slack amongst the sub-requests, that span more chips, to
benefit further from \texttt{Wbyp}. In workloads such as \texttt{web3} and \texttt{usr0}, write slack is much smaller giving less than 5\% improvements with \texttt{Wbyp}.

**Impact of Rbyp, WP and Rbyp+WP.** With a higher fraction of reads, Rbyp and WP provide higher response time improvements in these workloads compared to the earlier set. Amongst these, \texttt{wdev0} and \texttt{ts0} have a larger fraction of requests availing of Rbyp and WP, which translates to a 4\% overall response time reduction in Rbyp+WP system. While increasing request sizes lead to larger number of sub-requests spanning more chips (to create higher slack), there is a point beyond which the slack times can drop. As all requests become large enough to span all the chips, that is already sufficient parallelism and slack-aware reordering is not likely to provide additional benefits. For instance, in \texttt{usr0} (with mean read request size of 66KB), the performance gains with Rbyp and WP are much smaller, with its read request sizes that are significantly higher than the rest (see Table 5.2).

**Slacker.** Wbyp is still giving the highest rewards of the three optimizations in this set of workloads (since write latency are much higher than reads), though the others do contribute to reasonable improvements. Slacker still does enjoy the additive benefits of the three to some extent, with overall response time improvements that range between 2\%–14.5\% (with an average benefit of 8.5\%).

**Impact of CS+BF.** Similar to the write-intensive workloads, employing co-scheduling and backfilling can help reduce requests’ response times beyond and over what Slacker provides. The higher benefits of CS+BF in \texttt{src} is more notable. This is a workload with higher amount of slack across its write requests, and as shown in Figure 3.10, CS+BF can more effectively service requests since it can reach lower number of times where a non-critical sub-request is blocked and cannot be serviced. Overall, \texttt{CS+BF} improves the requests response time by up to 22\% in \texttt{wdev0}, with an average improvement of 12.3\%.

**Slacker++.** Slacker++ incorporates all the enhancements and achieves by up to 27.13\% reduction in response times across read-write balanced workloads. Similar to write-intensive case, the Slacker and co-scheduling and backfilling benefits are additive, as the combination of these leads to higher benefits, by up to 12\% and 5.5\% higher benefits in \texttt{wdev0}, as compared to Slacker and CS+BF, respectively. Overall, Slacker++ improves response times in this set by 16\% on average.
3.5.3 Results for Read-Intensive Traces

Results for read intensive workloads are reported in Figure 3.9.(c).

Impact of Wbyp. As can be expected, with the low fraction of write requests in these workloads, there is little opportunity to benefit from Wbyp.

Impact of Rbyp, WP and Rbyp+WP. The fraction of requests that benefit from these read optimization is higher than those in the previous set. The fraction of requests benefiting from these optimization is over 20% in rsrch2 and hm1. These are also the workloads which reap the highest improvements (of 12% and 13% for Rbyp+WP). Between Rbyp and WP, since the write requests are less prevalent, the latter enhancement does not have much scope in this set of workloads. So the read enhancements are mainly contributed to by Rbyp.

Slacker. Based on the above observations, Slacker’s benefits in this workload set is mainly driven by Rbyp. Overall, response time gains are up to 13% (average of 6.5%). As in the previous two workload sets, very few requests (less than 5%) suffer any performance degradation with Slacker, while 20% of all requests benefit on the average.

Impact of CS+BF. Similar to Slacker benefits, the amount of reduction in response times drops in read-intensive workloads. However, in some cases such as mds0, hm1, and proj4, CS+BF reaps higher benefits than Slacker. Overall, co-scheduling of requests in read-intensive traces, reaps as much as 19.5% improvements in response times, and with 9.65% improvement on average.

Slacker++. Slacker++ combines all the enhancements and provides the largest benefits in this set of workloads. Slacker++ can reduce response times by up to 24.23% (9.65% on average) across read-intensive workloads. In some cases such as hm1, combining co-scheduling with Slacker mechanisms, can notably increase the benefits and achieve 24.23% reduction in read requests’ response time.

3.5.4 Sensitivity Analysis

We have also conducted extensive analysis of the sensitivity of Slacker benefits to different hardware (pages sizes, chips per channel and other parallelism parameters, read and write latency, etc.) and workload (request sizes, inter-request times, read to write ratios, etc.) parameters. We briefly summarize the overall observations from those experiments. Growing chip densities and higher MLC levels can worsen
read/write latency, accentuating the slack. Even if technology improvements drive down latency of these operations, workload intensities would also increase in the future, continuing to stress the importance of slack exploitation techniques. Pages sizes do not have as much impact on slack exploitation for the range of realistic page sizes studied. When the hardware offered parallelism within a channel increases substantially for a given load, the request queue lengths at each chip drops, thus reducing slack. However, as the load also commensurately increases, slack exploitation continues to remain important.
Chapter 4 | Leveraging Content Popularity for Read Re-direction in SSDs

In the previous chapter, we develop a scheduling mechanism to address the non-uniformity problem in the SSDs. Despite exploiting and mitigating the non-uniformity within SSD requests and achieving considerable reduction in response time, our proposed scheduling mechanisms are restricted in terms of the flash chips servicing each SSD request. As explained in the preliminary chapter, due to out-of-place updates in NAND Flash SSDs, writes are inherently suited to be redirected to another flash chip which is opportunistically free. This dynamic re-direction is, however, nontrivial for read requests since reads are more restricted in terms of who can service them. While blindly replicating all the data everywhere seems very promising from a read redirection perspective, doing so results in high space overheads, high write/replication overheads and lower endurance (as each NAND flash cell can tolerate a limited number of program and erase cycles). In the remaining of this dissertation, we propose novel approaches to achieve re-direction for read requests. In this chapter, we present a novel approach to selective replication, wherein the popularity of data is used to figure out the “what”, “how much”, “where” and “when” questions for replication. Leveraging value locality/popularity, that is often observed in practice, popular data is replicated across multiple chips to provide more opportunities for dynamic read redirection to less loaded flash chips.
4.1 Motivation and Objectives

This section studies the resource contention problem in modern SSDs and analyzes the trade-off between replication and read re-direction.

**Resource Contention.** Service time of a request in an SSD consists of three components: (i) the *waiting time at chip-level queue* (a.k.a. *queuing time*), (ii) the *transfer time on shared channels*, and (iii) the *actual latencies of read or write operation*. The first two components are due to conflicts for shared resources and contribute to the latency overhead in the request service time. To quantify this latency overhead, Figure 4.1 presents the breakdown of read and write service time for six workloads. We make two observations from this figure. *First*, contention is large for both read and writes. This overhead is very high for writes (80% to 90%). The reason is that, as writes are typically about 10–40 times slower than reads, the FTL’s schedulers prioritize reads over writes which in turn increases waiting time of writes in the queues [50]. The contention overhead for reads is also significant and exhibits high variability (between 25% and 77% of total read service time). Note that reads are normally more latency sensitive, being on the critical path of the computation. *Second*, the queuing time is much more dominant than the channel transfer time. In fact, with current ONFi signaling used by modern flash memories, transferring one page to a chip takes at most a few microseconds. As
such, the queuing time is the main contributor to the service time, and by reducing or removing it, we can expect significant benefits.

4.1.1 All Chip Replication

The most straightforward way to reduce the queuing time is to map each incoming request to the least-loaded flash chip at that moment, making chip-level traffic balanced. Due to load imbalance across different flash chips (demonstrated in Figure 3.3) not all flash chips are load balanced. As a consequence, a typical request does not necessarily get serviced from the chip with the least load. This motivates the need for a load-aware dynamic mapping/redirection of logical pages to physical pages, so that any incoming request is allocated to the least-loaded chip. As discussed in preliminary sections, Write redirection is easy to implement in SSDs. Prior works [35–37, 51] have explored the potential performance benefits of write redirection and discussed the corresponding overheads.

In contrary to write re-direction, Read redirection is particularly challenging in today’s SSDs since a page is read from the location, where it was written. Thus, at the time of reading a page, it is unlikely that the target page is on the currently least-loaded chip. The simplest approach to enable read redirection is to replicate each page (when it is written) on all chips, i.e., keeping $N$ copies of each logical page in an $N$-chip SSD. To see how much this Ideal-All-Chip Replication scheme can improve performance, we conduct a simple experiment:

With all the data blocks replicated on all the flash chips without worrying about storage space or the replication costs (i.e., no write costs for replication), the FTL chooses the least-loaded chip for serving each read request. Figure 4.2 shows the read and total (reads+writes) service time improvements brought by this scheme over a conventional SSD. We can see that read redirection (enabled by the Ideal-All-Chip Replication) can improve read service time by 17% to 57% (38.37% on average) and total service time by 12% to 48% (25.32% on average).

Despite its performance benefits, this Ideal-All-Chip Replication design is very costly to implement in terms of (write) performance, lifetime and space overheads. It amplifies write traffic by $N$ times (in an SSD with $N$ chips) which can become overwhelming since writes are much slower than reads, and decreases the device lifetime by almost $N$ times. Furthermore, it also increases design complexity in
two ways: (i) it needs $N$-times more capacity, and (ii) FTL has to keep $N$ Physical Page Numbers (PPNs) for each Logical Page Number (LPN) in the mapping table.

Prior work [25, 52] have attempted to enable read re-direction in SSDs by exploiting temporal locality in the accesses. PBLRU [52] proposes a dynamic page replication mechanism to exploit the multi-chip parallelism in SSDs. However, their physical page replication is not able to reap most of the benefits achieved by the Ideal-All-Chip-Replication and its efficacy is very limited by the amount of space reserved for the replication. Later in this chapter, we will quantitatively compare our proposed approach with this work. [25] on the other hand, explores the opportunity of re-directing read requests in an array of SSDs by performing inter-SSD replication taking into account de-duplication of data. However, their proposed approach does not consider the internal architecture and resource contention of SSDs which play a significant role in determining the request’s response time.

To overcome these challenges, while still enabling read redirection, we propose content popularity-based selective replication and introduce a novel SSD design, called Read-Redirected SSD (RR-SSD). The key insight behind RR-SSD is that “value popularity” (or “value locality”), which has often been observed in I/O datasets, can be exploited so that a small portion of all possible values can be selectively replicated so as to enable redirection of a large fraction of read requests.

Figure 4.2: The percentage of read and total service time improvement given by the Ideal-All-Chip Replication scheme.
As such, if the FTL maintains multiple copies of every popular value on multiple chips, at the time of reading a page with the same content, it can redirect the request to any one of the chips containing the target value. This builds upon the use of SSD as a Content-Addressable Storage device\(^1\) as previously proposed [1,24].

An efficient and low overhead implementation of RR-SSD requires us to answer five important questions: (1) which data has to be chosen for replication?, (2) how much space can be tolerated for holding replicated data?, (3) where should each copy of a value be placed?, (4) when should the replication process be invoked?, and (5) what are the required changes on the SSD board and FTL to support replication/redirect? In the following sections of this chapter, we answer these questions separately and describe the design choices and trade-offs that should be taken into account when employing RR-SSD for a storage system.

### 4.2 Selective Replication

This section systematically answers the five questions raised in the introduction for supporting a read redirection mechanism in SSDs.

\(^1\)Content-Addressable SSD (CA-SSD) is a type of de-duplication mechanism, which stores only one copy of each value in flash memories and modifies FTL to relate multiple LPNs with same content to one PPN. There are multiple implementations of CA-SSD available in literature [1,24] which usually employ a cryptographic hash to represent each chunk (usually with size of a page). Similar to these studies, we assume hashed values are collision-resistant (the probability of collision in range of $10^{-9} - 10^{-17}$ [53] for MD5 and SHA-1).
4.2.1 What to Replicate?

To avoid the high cost of Ideal-All-Chip Replication while providing read redirection for a great amount of read traffic, we leverage the phenomenon of Value Popularity (VP) (number of occurrences) of each unique value for reads/writes. In Figure 4.3, we present VP (as CDFs) for reads and writes for three of our workloads (Section 4.4 gives detailed characterization of the studied workloads). In each sub-figure, the points on X-axis are sorted based on their popularity in reads. The following insights emerge from analyzing the results in this figure.

- We observe high Read Value Popularity (RVP) for each workload. For instance, the fraction of total unique values that accounts for 80% of overall reads are 14%, 7%, and 30% for homes, web, and mail, respectively (shown by dotted lines). Therefore, in each workload, only a small set of values are heavily popular, and by replicating only these values, we can reduce the high costs of replication while providing performance benefits of redirecting a majority of read requests.

- We find that these workloads exhibit different behaviors in the popularity of a specific value for reads versus writes. For example, in web, 7% most popular read values (which are 80% of overall read values) contribute to less than 8% of total written content. The case for mail, however, is different: most popular values in reads are still the most popular values in writes. Since we target redirection for read requests, we need a mechanism that can selectively capture the popular values in reads, rather than focusing on universal popularity across both reads and writes, as in the prior works.

The presence of high RVP in our workloads has an important implication on our SSD design. By capturing heavily popular values in reads and replicating them over multiple chips we increase the chances for redirecting a majority of read requests to a chip with low load, while we can limit the replication costs. However, the performance improvement and replication costs are related to three other design factors, discussed in the next three subsections.
Figure 4.4: Temporal VP for reads for the first 7 consecutive days of three workloads. The points on X-axis are sorted based on their read value popularity in the first day.

4.2.2 How Much Space for Replication?

The space provided for replicated data determines both the performance achieved by read redirection and the replication costs (performance and lifetime). Specifically, by allocating a larger space to the replicated data, the write traffic for replication exacerbates (which also increases the corresponding performance and lifetime costs), while increasing the chances for read redirection (probably reducing the average read access latency). Suppose that the maximum allocated space for replication is $S\%$ of the workload’s size (i.e., the total number of unique LPNs accessed in the trace over its entire duration), which has to be determined by the system administrator or the user at the system set-up time. Note that this space overhead ($S\%$) can be zero in our design, while a portion of popular values can be still replicated (with respect to a non-deduplicated store). The reason is that storing dataset in a CA-SSD (which is our baseline system) usually needs less space compared to a traditional SSD which does not remove replicas. Indeed, as prior works [1,24,53] show, most real-world workloads have the same content being written into the same or different addresses which are de-duplicated in a CA-SSD type of architecture. Based on our workload characterization presented in Section 4.4, the space required to store our workload set in a CA-SSD varies between 15% and 70% of the original size, providing a considerable room for replication.
4.2.3 When to Replicate?

The replication process is expensive as it incurs high write traffic in SSD. Since write operations are very slow in flash memories, this is a serious concern. As a result, the replication process should be *infrequent*. To this end, we leverage another characteristic of value popularity, called *Temporal Value Popularity* (TVP). The presence of TVP in a workload implies that, if a certain value is a popular one now, it is likely to remain popular in the near future, even if the LPNs are different. In this work, we specifically employ TVP for reads and discuss its implications on our design. To give a better insight of TVP, Figure 4.4 presents the value popularity of reads (as CDFs) for the first 7 consecutive days of three workloads. The points on X-axis are sorted based on their popularity in the first day of execution. We observe high TVP for reads in a workload such as *webvm* over a long time\(^2\), implying the set of most read popular values does not change in great extent from day to day\(^3\). This suggests a simple mechanism that captures the RVPs during one day and replicating them at the end of day, in order to use them for read redirection in the next day.

One may consider read TVP for periods larger than one day (e.g., one week), in order to further reduce the replication costs. However, if the replication is performed very infrequently, the potential gains of read redirection might be affected, as we may fail to capture part of dynamism in VPs of a workload, which could be captured had we used a shorter period. Figure 4.5 shows the number of distinct values accessed for the first time during each day of the entire execution of three of our workloads. As can be seen, the number of new values accessed each day is variable in a workload, and it is quite considerable on some days (around \(10^6\)). Thus, if we perform replication very infrequently (weekly as an example), we may lose a great amount of benefits gained by the daily read redirection. Table 4.1 reports the replication cost (in terms of the number of page copies as well as the time duration) for the three workloads, when the replication is performed daily or weekly. The results are reported for a typical day and week. In this experiment, we set the space overhead of replicated data to 0% (with respect to the size of the non

\(^2\)This pattern does not seem dominant for homes workload. However, as we show later in the results section, other parameters affect the performance benefits and even for this workload, read service time can be improved by employing our approach.

\(^3\)Similar observation was made in prior works developing a content-based cache for I/O performance improvement in content addressable storage based schemes [53].
Figure 4.5: The number of distinct values accessed for the first time during each day in each of the three workloads.

Table 4.1: Comparative analysis of daily-replication versus weekly-replication.

<table>
<thead>
<tr>
<th>Freq.</th>
<th>mail</th>
<th>homes</th>
<th>webvm</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Copies (Kilo)</td>
<td>Time (Min)</td>
<td>Copies (Kilo)</td>
</tr>
<tr>
<td>Daily</td>
<td>160.9</td>
<td>7.96</td>
<td>5.6</td>
</tr>
<tr>
<td>Weekly</td>
<td>1253.2</td>
<td>62.01</td>
<td>39.4</td>
</tr>
</tbody>
</table>

de-duplicated dataset), the read/write latencies to 75us/400us, and the channel transfer time of one (4KB) page to 10us. Two important insights emerge from the results in this table. First, even if we perform replication daily, its overhead is not significant in real workloads – it only takes a few minutes (a maximum of 8 minutes in mail). Second, the replication latency increases almost linearly as its frequency decreases. In other words, weekly-replication generally takes (almost) seven times longer time than daily-replication (a maximum of 62 minutes per week compared to a maximum of 8 minutes per day). Thus, by performing replication daily, we are able to capture dynamism of VP with almost proportionally-same cost of the weekly-replication.

Figure 4.6 depicts a high-level view of the proposed replication process. The
Figure 4.6: A high-level view of one period of our replication scheme.

daily operation has (i) the normal operation phase which includes read redirection and collecting information for the replication process at the end of the day; and (ii) the Replication phase at the end of the day (for 8 minutes) when replicas are created based on the VP of that day.

**Profiling for VP** – Each day, we first need to collect information that helps determine the value popularity of the dataset. A typical way of profiling requires counters for every hash value and incrementing the counter for the corresponding data on a read. The counters can be implemented in hardware or software. To capture all read activities at high resolution, we generally need large counters which are costly (especially hardware counters). Instead we employ $N$ bit saturating counters –with $N$ referring to the $\log(\text{Number of flashchips})$ – since the most popular data will be, at most, replicated on all flash chips, with each flash chip only storing one copy of that data. Thus, these counters suffice to distinguish very popular values from the rest. As we will show, this counter can be maintained in the flash controller or in the FTL itself which is anyway looked upon each access to this value for a normal operation.

**Replication** – At the end of each day, the profiled data is used to perform replication for the next day. This can be expensive since it involves counter sorting, mapping each replicated data, and updating the mapping tables at the FTL in addition to actual data replication (i.e., reading and writing each data to be copied). Section 4.3 describes how the execution phase can be realized through a series of administrative actions, which have no impact on the system availability and minimal impact on the system performance, even though it takes at most 8 minutes for every 24 hours. Although this replication interval is very short, there might be opportunities of spreading the work during the day and performing the replication in an on-line fashion. As mentioned above, due to high cost of replication, frequently replicating data can impose many temporary episodes of high latency, especially
for reads, thereby diminishing the effectiveness of our mechanism. However, later in this chapter, we will study the on-line replication mechanism and compare it with the end-of-the-day replication.

### 4.2.4 How Many and Where to Replicate?

Due to limited space for replicated data, the read performance efficiency of the proposed mechanism heavily depends on *(i) the number of copies kept for each data*, and *(ii) the place (or chip) that each copy is mapped to*. We describe the impact of each of these factors, separately.

- **Number of copies:** Assuming a fixed space for replication, one can take three actions: *(i) keeping lots of copies of a few very popular data*, *(ii) keeping a few copies of a lot of data*, and *(iii) following VP proportionally (i.e., the more popular, the more copies)*. The two former decisions are static and may not be effective since different workloads and different phases of one workload exhibit different behaviors. The third decision, on the other hand, is efficient to capture a workload’s dynamism and use the collected information to determine the replication degree of the each value.

- **Placement of copies:** The mapping decisions made for data copies at the end of each day determines the chip-level loads in the following day. One may use static or random mapping strategies, but as our experimental results also confirm (Section 4.4), such strategies usually are not able to capture most of the performance benefits resulting from Ideal-All-Chip Replication (Figure 4.2) because they are generally load-unaware. On the other hand, load-aware mapping is challenging since the VP information is the only available information we have at the end of a day. Thus, we need richer profiling information (such as popularity counts and the amount of available space on each chip) to use for mapping.

To address the above issues, we introduce a novel and low-overhead mechanism which collects extra information during the normal execution to help find the number of copies and their placement. When replicating each data on single/multiple chip(s), we attempt to evenly distribute the replicas/load on different flash chips in order not to overwhelm some flash chips while some others experience much lower loads.
To this end, we develop our replica placement algorithm considering the following parameters for each flash chip:

**Available Space ($S$):** One of the main parameters based on which the destination chip for a replica is determined is the available space on that chip. By keeping track of the number of occupied pages on each flash chip, we can select the one with the least number of occupied pages to place a replica.

**Erase Counts ($E$):** Updating a page in NAND flash SSD requires an *erase* operation beforehand which takes much longer time than read and write operations. Thus, SSDs perform out-of-place-update to overcome this issue. This property necessitates FTL to periodically run Garbage Collection (GC), in order to free up space by reading out the valid data, and a long latency erase operation. Apart from performance issues of GC, each NAND flash cell can bear only a limited number of erases before it wears out. We take the number of erases occurred in each flash chip into account when making replica placement decisions, thereby attempting to evenly distribute the wear on different flash chips.

**Popularity Count ($Pop$):** As already discussed, the existing value popularity results in more accesses to popular values while non-popular values are not accessed very frequently. If we blindly select the target chip for replication, without considering popularity degrees, there can be situation where a chip stores a larger number of popular values and gets more congested as a consequence. Thus, we need to take the popularity degree of values already stored on a flash chip into account when determining the target flash chip(s) for replication. By storing the popularity information of different flash chips, we attempt to evenly distribute the popular values on different flash chips in order not to swamp a flash chip with majority of popular values being stored on it.

We take these three parameters into account when making data placement decisions for selective replication: Available Space, $S$, Mean Erase Count, $E$, and Mean Popularity Count, $Pop$. We select a chip to place the replica for which the $\frac{S}{E \times Pop}$ is higher. To determine the values of $S$, $E$, and $Pop$, we maintain several counters in the device controller, having the FTL provide such information. Note than, when studying on-line replication later in this chapter, we will add another parameter which takes the instant chip-level loads when placing replicas into account.

Moreover, we maintain information about popularity of each unique value and
which flash chips it has already been stored on. The details of our implementation is described in the next section. With this information, at the end of the day, one can figure out where the popular values have to be placed based on their read requests for those data values during the day. Such statistics collection will take significantly lower times than the latencies of actual read/write operations to the flash chips. Note the replication can be spread out during the day and execute more frequently on shorter intervals (e.g., on hourly basis). However, as explained before, doing replication more frequently does not reduce relative replication costs (due to linear behaviour of replication costs), neither helps with the performance improvements as our results show, rather results in more frequent SSD slowdown due to replication.

4.3 Implementation

This section describes how a flash-based Content-Addressable SSD works and then provides the details of the modifications to implement our Read-Redirected SSD (RR-SSD) along with details of the changes required in the FTL.

4.3.1 Content-Addressable SSD: The Baseline

A Content-Addressable SSD (CA-SSD) typically requires additional components (compared to a traditional SSD) for implementation [1,24]. We use a recent CA-SSD design proposed by Chen el al. [1] as the baseline in this work. They refer to FTL
in CA-SSD as CA-FTL which needs three key enhancements to a traditional SSD to achieve the CA-SSD functionality. First, CA-FTL employs a dedicated on-board processor (called Hashing Unit) to compute/compare hashes very fast, which is proposed to use the cryptographic processor found in modern SSD devices (e.g., SSD products by Intel [54] and Samsung [55] implement real-time AES encryption in hardware). Second, the Mapping Unit should maintain additional data structures for CA-SSD (called CA-FTL’s meta-data) stored in the on-SSD RAM. The GC also needs to be changed, since compared to conventional SSDs (where each update results in page invalidation requiring an eventual erase operation), CA-FTL only needs to invalidate a page when no LPN points to a value in that page. Third, since the per-page meta-data information in this scheme is large, CA-FTL employs a Battery-Backed RAM (BB-RAM) [24,56] to guarantee loss-less write-back of the CA-FTL’s meta-data in flash chips on power failure. Next, we describe the required data structures in FTL of the proposed Read-Redirected SSD (RR-SSD) for supporting read redirection.

4.3.2 SSD Enhancements for RR-SSD

In Figure 4.7.a, we present a high-level view of the the proposed RR-SSD which requires two key enhancements to the baseline CA-SSD. These components are shown in gray in this figure and described below.

4.3.2.1 Mapping Unit for RR-SSD’s FTL

The one-to-one mapping (from LPN to PPN) in conventional SSD is not applicable to our design as in our design, a PPN can be pointed to by many other LPNs thus, we require a many-to-one mapping structure to begin with – else the relocation and updates due to GC mandate many updates to the Mapping Unit as each page movement will need updates to all LPNs. Mapping Unit must have additional data structures for RR-SSD to maintain the relationship among hashes, LPNs and PPNs. Here, we assume a page-level address mapping. Figure 4.7.b shows the data structures employed by the Mapping Unit for RR-SSD that is composed of three tables. Despite traditional FTL which maintains a mapping table for mapping each LPN to the corresponding PPN, similar to [1], we employ two tables, a primary mapping table and a secondary mapping table, along with an auxiliary
table, called *fingerprint store*, to maintain the hash of the contents (data chunks) and the mapping information required. Below we scrutinize each table. Circled numbers in the text below refer to table numbers in Figure 4.7.b.

1. To avoid storing too many redundant mapping information and also minimize the searches in the Mapping Unit, the primary mapping table (1) maps each LPN to a Virtual Page Number (VPN). A VPN is a virtual address assigned to a set of LPNs mapped to the same PPN. This table is also used in CA-SSD, hence no modification is required to employ it in our design.

2. In CA-SSD, the secondary mapping table (2) maps a VPN to a PPN, with each entry of this table being indexed by the VPN. Thus, a mapping from each LPN to its respective PPN is done in an indirect fashion: first, from LPN to VPN, then from VPN to PPN. In RR-SSD, however, a data can be replicated and reside in multiple physical pages, rather than only one page, thereby, requiring to maintain a list of PPNs for each entry of this table. Lastly, like CA-SSD, we store the reference count for each VPN entry, denoting the number of LPNs mapped to a VPN. This reference count is used as a means to identify the VPNs which are not valid any more, hence, their associated PPNs can be reclaimed by GC.

3. In order to find the information required for de-duplication and replication, we maintain a fingerprint store (3) which maps the hashed value to a VPN. This table has also been used in CA-SSD. However, to find replicas of a hashed value and also collect profiling data, we add the following items to each entry of the fingerprint store: (i) an $N$-bit vector (called *Mapping Vector (MV)*), with $N$ referring to the number of flash chips, which keeps track of the list of flash chips that have same content corresponding to each unique hash of value (i.e., a replica), (ii) and a 6-bit saturating counter (VPC). The last attribute is used for the profiling logic which is later discussed in this section. Entries are inserted in the fingerprint store upon writing a new data to SSD, which is mapped to one of the flash chips and the corresponding bit in MV vector is set to ‘1’. The MV attribute gets updated when data is either replicated or invalidated (setting it to ‘1’ or ‘0’).

**Storing Mapping Unit in RR-SSD:** The primary and secondary mapping tables are maintained in the BB-RAM so that on a power failure, the critical mapping
information does not get lost. Moreover, the fingerprint store is maintained on the RAM space in the SSD and in case of power failure, a capacitor (such as SuperCap [57]) will provide enough current to flush the mapping information to the persistent storage. The space required to store the aforementioned tables exceeds the available RAM/BB-RAM space in SSDs. However, observing significant temporal locality and temporal value locality, similar to [24] and [58], we employ a simple LRU caching mechanism to selectively store a subset of these tables with the most frequently accessed entries. As discussed by Gupta et al. [24], the miss rate of this cache is around 5% and does not impact the performance benefits. We set the size of this cache (after an extensive sensitivity analysis) to accommodate this Mapping Unit as 256MB which is simply affordable in today’s SSDs with GBs of RAM space. To provide the consistency of tables and not miss any profiling information, we propose to periodically checkpoint the Mapping Unit tables in flash chips. Thus, the in-memory tables are periodically synced into the flash.

**Handling Write Requests:** On receiving a write request, the hash of the value for each LPN comprising the request is calculated and the fingerprint store (3) is then looked up with this hash. On a miss in this table, a new page is allocated in one of the chips to store the new data and a write operation is issued. Meantime, a new entry is allocated in each of the mapping tables – it stores (LPN, VPN) in primary table (1), (VPN, PPN) in secondary table (2), and (hash, MV vector) in the fingerprint store (3). On a hit, on the other hand, after updating MV in fingerprint store (3) and (LPN, VPN) in the primary mapping table (1), SSD returns a write request without requiring flash chip writes, indicating that its content already exists in SSD.

**Handling Read Requests:** On a read, the primary mapping table (1) is first looked up to get the VPN of data that is going to be read. Then this VPN is used to determine the PPNs storing the replicas for this VPN. The FTL chooses the chip with the lowest load at that time for servicing the read request and the read operation is then issued.

Finally, note that we do not modify GC or WL policies in this work and assume that RR-SSD continues to employ the GC and WL policies used in the CA-SSD with the only difference that, once a VPN is invalidated, all the respective PPNs

---

4 Only first few bits of each PPN can be examined to determine the flash chip that PPN belongs to.
are marked invalid, waiting to be reclaimed by GC.

4.3.2.2 Profiling Logic

The profiling logic (for what to replicate and where to replicate) has two parts (Figure 4.7). First, in order to reduce the space needed for profiling tables, we use the fingerprint store in Mapping Unit to keep the profiling information. Each entry of this table stores VPC and MV, as already explained. Second, we collect profiling information for each chip as discussed in the previous section. For each chip we maintain the information about the available space, erase counts and popularity counts. To this end, we employ several counters and increase or decrease them with each read, write, and erase event.

Normal Execution and Profiling: At the beginning of profiling, all VPC counters are zero. On reading a value, after finding its hash entry in the fingerprint store, its VPC counter increments (and saturated at 63). On inserting a new entry in the fingerprint store by a write request, the VPC counters have initial values of zero. Also, these counters are not subsequently touched by any writes.

Replication: Data centers usually employ redundancy techniques such as mirroring to achieve high levels of reliability [59, 60]. Some prior work [9, 20] have also burrowed this assumption to accomplish their maintenance time in the SSD. Thus, we assume having a Mirror SSD (probably cheaper and with lower performance) which is a conventional SSD that works in parallel with RR-SSD, receiving the same read/write requests (maintaining the same content). While RR-SSD is busy at the end of the day (for maximum of 8 minutes) performing the data replication, the mirror SSD keeps servicing the incoming requests before the RR-SSD comes back online.

The replication process involves counter sorting, actual data copying, and updating tables in the Mapping Unit. Since counter sorting is costly, we traverse the fingerprint table several times, and during each pass, we decide whether or not to replicate each data. More precisely, in the first pass, we select the hash values with VPC of 63, if there is any, and replicate them in the chip(s) suggested by our data placement metric. This requires (1) reading the actual data from one of the previously-copied places, and (2) updating the MV in the fingerprint store table, to represent where data should be copied in order to be used in the next day for read redirection. In the next passes, we repeat the same process, this time for hashes
with lower VPCs. The replication process terminates when we reach the space limit for replication. Note that during replication, we may need to invalidate one (or more) copies of a data, if they are no longer popular for the next day. To reflect this, FTL has to update all the tables in Mapping Unit. However, this process not need being done every day, and as prior work [61] suggests, this adjustment process can be done with less frequency than the replication. For instance, we can do it on a weekly basis to reduce the overheads associated with it.

4.4 Experimental Setup

This section describes our evaluation methodology and the experimental results using a diverse set of six workloads.

4.4.1 Methodology

Evaluation Platform: Similar to previous chapter, We use SSDSim for simulating both a CA-SSD [1] and the proposed RR-SSD. We have modified SSDSim to model the CA-SSD (which is used as “baseline” of our design) and augmented it with additional components and functionalities required by our RR-SSD. We assume that the overhead of hash calculation is 12us [62] and modeled its impact on the queuing latency of the incoming read and write requests.

Studied Configurations: The baseline SSD consists of eight channels, each of which is connected to eight NAND flash chips. Each channel works on ONFi 4.0 [63]. Table 5.1 provides specifications of the modeled SSD (which is very similar to [64]) along with parameters of the baseline configuration. Also we equip our model with NVMe [21] standard interface at HIL. We compare the performances of three systems:

1. **RR-SSD** employs the proposed data replication and read redirection schemes. RR-SSD reserves a space (overhead) of $\alpha\%$ of the workload’s size (original non-deduplicated dataset) for replicated data. The $\alpha$ parameter is specified by the user or system administrator).

2. **CA-SSD** is a content addressable SSD based on [1].
Table 4.2: Main characteristics of simulated SSD.

<table>
<thead>
<tr>
<th>Evaluated SSD Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 Channels and 8 Flash Chips per Channel, Channel Width = 8 bits, Chip Interface Speed = 533 MT/s (ONFI 4.0), Hashing Latency = 12us [62]</td>
</tr>
</tbody>
</table>

NAND flash

| Page Size = 4KB, Metadata Size = 448B, Block Size = 256 pages, Planes per Die = 2, Dies per Chip = 4, Flash Chip Capacity = 16GB, Read Latency = 75µs, Typical Program Latency = 400µs, Erase Latency = 3.8ms |

Table 4.3: Workload Characteristics. Unique values represent the percentage of read (write) requests which read (write) unique 4KB chunks. Improvement by CA-SSD denotes the percentage of reduction in read (write) response time by [1] compared to the conventional SSDs.

<table>
<thead>
<tr>
<th>Trace Name</th>
<th>WR %</th>
<th>Unique Value</th>
<th>Imp. by CA-SSD</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>WR</td>
<td>RD</td>
<td>WR</td>
</tr>
<tr>
<td>webvm</td>
<td>77</td>
<td>42</td>
<td>32</td>
</tr>
<tr>
<td>homes</td>
<td>96</td>
<td>66</td>
<td>80</td>
</tr>
<tr>
<td>mail</td>
<td>77</td>
<td>8</td>
<td>80</td>
</tr>
<tr>
<td>hadoop</td>
<td>30</td>
<td>63.9</td>
<td>17.5</td>
</tr>
<tr>
<td>trans</td>
<td>55</td>
<td>77.4</td>
<td>13.8</td>
</tr>
<tr>
<td>desktop</td>
<td>42</td>
<td>74.7</td>
<td>49.7</td>
</tr>
</tbody>
</table>

3. Oracle uses Ideal-All-Chip Replication and redirects all reads to the least-loaded chip at any moment. This system in a sense represents the maximum achievable performance gain by read redirection without considering associated replication costs.

For each of the evaluated systems, we report the amount of reduction in read response time as well as total (read+write) response time as our performance metrics.

Workloads: We use a set of six disk traces [1,53] which contain the values read or written for each request. These traces have been extensively used by previous related studies [1,24,53]. They are collected from different kinds of big data applications and servers (including mail, web, database servers) and systems (an experimental system and an office system) at FIU and OSU universities. Individual requests in these workloads are of size 4KB, along with hash (SHA-1 or MD5) of the contents. Table 4.3 summarizes the main characteristics of the disk traces. This table also reports the read and write performance improvements achieved by
CA-SSD compared to conventional SSD (with same internal organization). We see that although the primary goal of CA-SSD is to improve write performance, it also results in some read response time improvement, which is mainly due to huge reduction in write traffic (queue lengths for reads also goes down). We use CA-SSD as our baseline in this work and normalize the results of both RR-SSD and the Oracle system to that of CA-SSD.

4.5 Experimental Results

In the next three subsections, we analyze the performance of RR-SSD. We first present the results for a system with no space overhead for replication, i.e., ISO-capacity analysis compared to original non-deduplicated dataset. We then analyze the performance and overhead trade-off as we increase the replication space overhead. Then, we study the need for a sophisticated way of figuring out where to create
the replicas compared to more naive strategies (such as random and/or static approaches). Lastly, we discuss the on-line replication and compare our approach with prior work.

4.5.0.1 ISO-Capacity Performance Analysis

RR-SSD is targeted to improve the read request service times with low overhead in terms of performance and storage. Figure 4.8 compares the read service time of different systems in an ISO-capacity design (space overhead is 0% of the non-deduplicated dataset). The results are given for 6 consecutive days of each workload starting from the second day\(^5\). The upper chart for each workload shows the read service time reduction for RR-SSD and Oracle system (shown by a solid line), compared to the CA-SSD.

We observe different behaviors for different workloads/days. For three workloads (mail, homes and webvm) which are less read-intensive, RR-SSD’s read improvement is very close to that of the Oracle system for most of the time – it is around 90% of the maximum improvement for all days of these workloads, except for one day of homes and mail. RR-SSD is able to improve read response time by 24%–65% for mail, 5%–80% for homes, and 17%–32% for webvm. For the other workloads (hadoop, trans and desktop) which are either highly read-intensive or have balanced number of reads and writes, RR-SSD is able to improve the read service time by up to 25% for hadoop, 35% for trans, and 14% for desktop. However, the gap between RR-SSD and the Oracle system is variable and significant in some days for these three.

To better explain the performance gap between RR-SSD and the Oracle system, Figure 4.8 (lower chart of each workload) also shows the workload’s dynamism during each day by reporting (i) the percentage of values read for the first time (beginning from the first day) – called New, and (ii) the percentage of read values in each day that have been replicated from the prior day – called Rep. As expected, if we see high percentage of Replicated values and low percentage of New values during a day, this usually (except for homes) corresponds to a lower gap between RR-SSD and the Oracle system performance. In fact, we observe three behaviors. First, in case of webvm with very high contribution of Replicated values, FTL has a

---

\(^5\)As replication starts at the end of the first day, the performance results for RR-SSD are the same as CA-SSD in the first day.
Table 4.4: Read and total (read+write) service time improvement of RR-SSD and Oracle system over the baseline CA-SSD for entire duration of each workload.

<table>
<thead>
<tr>
<th>Trace Name</th>
<th>RR-SSD's Imp. [%]</th>
<th>Oracle's Imp. [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>RD</td>
<td>Total</td>
</tr>
<tr>
<td>webvm</td>
<td>23.43</td>
<td>15.71</td>
</tr>
<tr>
<td>Homes</td>
<td>35.11</td>
<td>16.89</td>
</tr>
<tr>
<td>Mail</td>
<td>45.37</td>
<td>40.26</td>
</tr>
<tr>
<td>Hadoop</td>
<td>11.70</td>
<td>9.55</td>
</tr>
<tr>
<td>Trans</td>
<td>19.97</td>
<td>11.01</td>
</tr>
<tr>
<td>Desktop</td>
<td>7.83</td>
<td>5.11</td>
</tr>
</tbody>
</table>

reasonable chance to redirect reads to a lower-loaded chip, which results in reducing response time of many read requests. Similarly, the huge gap between RR-SSD and Oracle in desktop is because the fraction of New values introduced every day is very high. Second, in case of hadoop and trans, although the percentage of New values is small, there is still a considerable gap between the two systems. The reason is that, as Table 4.3 indicates, the amount of space we save in these two workloads and use for replication (at the end of day) and redirection (in the next day) is small (i.e., 17.5% for hadoop and 14% for trans), resulting in low contribution of Replicated values in the next day. Third, in case of the other three workloads (mail, homes, desktop), the small performance gap between RR-SSD and Oracle system is due to both huge space saved by de-duplication and low fraction of New values (except for the first few days of homes execution).

Finally, Table 4.4 reports read and total (read+write) performance improvement given by RR-SSD and the Oracle compared to the CA-SSD.

4.5.0.2 Impact of Replication Space Overhead on Read Performance

The RR-SSD’s performance efficiency largely depends on the amount of space used for replicated data. This is specially important for workloads like hadoop and trans where we do not save significant space with de-duplication. Figure 4.9 shows the sensitivity of RR-SSD’s read performance improvement to the replication space overhead. We only show the results for hadoop and trans, since the other workloads generally save large space with de-duplication and already achieve near-ideal (Oracle) read improvement. We can see that, as a larger replication space is used, a higher read performance improvement is achieved (around 25% and 33%
Figure 4.9: Read service time improvement as a function of allowed space overhead for replicating data. Results of the Oracle system is shown for reference.

read improvement for hadoop and trans, respectively), but it saturates after a point (i.e., around 20%). Consequently, by employing RR-SSD with almost 20% space overhead for replication, one can achieve a performance improvement which is quite close to that of Oracle (last column for each application in Figure 4.9).

4.5.0.3 Efficiency of Mapping Strategy in RR-SSD

Instead of our proposed replica placement mechanism in RR-SSD, one may employ simpler strategies for mapping the replicated data. Figure 4.10 compares our RR-SSD mechanisms with two other naive strategies: Random and Static. In the Random scheme, data copies are randomly allocated over all flash chips. In the Static scheme, on the other hand, data copies are located on flash chips in a round robin fashion, starting from the chip that the original data was mapped to, i.e., if it was originally in chip $i$, and $j$ copies have to be created, replicas are made at chips $(i + 1)\% N, \ldots, (i + j)\% N$, with $N$ representing the number of chips. Note that the number of copies for each data is equal in all these three schemes, and is determined by the same mechanism that we used in our approach (only chip-level mappings are different). The results plotted in Figure 4.10 are averages over all read requests for the entire workload’s run. We can see that our history-based mapping (shown as RR-SSD in the figure) significantly increases the chance of
finding a low-loaded chip when servicing each read request. More accurately, the read improvement achieved by employing RR-SSD mapping is generally twice that for the random and static mappings.

We also show the improvements gained by prior work [52] (PBLRU in this Figure 4.10) for comparison. Note that, prior work does not consider the content-addressable design and cannot exploit the content popularity to selective replication. They however, consider only the temporal locality among the physical pages and employ a priority-based least recently used (PBLRU) mechanism to dynamically replicate the frequently accessed physical pages. To conduct a fair comparison, we implemented this approach in CA-SSD – else the significant write reduction due to deduplication would blur the benefits of PBLRU. Also, like Random and Static, we fix the space available for replication to be same in all experiments. As shown in the figure, although PBLRU outperforms Random and Static replica placement and yields superior benefits in terms of read service time improvement, it falls behind RR-SSD as it is not able to capture the content popularity of data and use it towards a more efficient replication. More specifically, two main reasons for our approach to outperform PBLRU are: (i) replicating a set of very popular values leads to more efficient use of the allocated space for replication, rather than only considering the temporal locality in the accesses to each physical page, and (ii) replicating values with higher popularity on more number of flash chips can help provide re-direction opportunity for larger number of read requests as the popular values account for a larger number of accesses.

4.5.0.4 On-line Replication

As mentioned in section 4.2.3, one may attempt to spread the replication during the day. We conduct an experiment in which replication is done on-line instead of doing it at the end of the day. Note that in order to determine which flash chip to place the replica, we slightly need to modify our replica selection algorithm to take the instant load of each flash chip into account. We define the parameter $QL$ to represent the chip-level load and select the flash chip for which $S_{E \times Pop \times QL}$ is higher. Also, to minimize the performance degradation due to replication, we can exploit the idle time of flash chips as discussed in the prior work [52]. We report the read and total service time improvements in Figure 4.11. As illustrated in this figure, on-line replication also improves both read and total
service time with average of 20.4% and 8.9% improvements, respectively. However, due to high cost of replication, spreading the work during the day may impose some temporary episodes of high latency. This is evident in the total service time improvements, which is much less than read service time improvement in this case. This is mainly due to slowing down writes to perform data copies, thus diminishing
the effectiveness of read re-direction.

4.5.0.5 Tail Latency Analysis

The importance of minimizing the tail has drawn a lot of recent attention from an SSD latency viewpoint [3, 8, 9, 11] since storage is a dominant component of many client facing applications. RR-SSD optimizations, in addition to improving the mean latency, result in a reduction in the tail latency. We report the reduction in tail latency (99\textsuperscript{th} percentile) for all requests, in Figure 4.12. Our proposed mechanism, results in around 2X improvement in the tail latency of the studied workloads, where the majority of benefits is achieved by enabling read-redirection to provide flexibility for a read request to circumvent a highly loaded flash chip.

4.5.0.6 Overhead Analysis

Although our proposed RR-SSD design aims at reducing the overheads associated with the Ideal-All-Chip Replication, it still incurs some overheads in terms of performance, lifetime and on-SSD RAM space. However, these overheads are not significant, considering current SSD architectures. More specifically, the overheads
of our RR-SSD design can be classified into three categories:

- **Memory Overheads.** The memory space required to store the mapping data structures, as discussed in Section 4.3, is similar to that in CA-SSD. The only additional information are those required for the profiling logic, which add only several bits (with regards to the number of chips) to each entry already storing a 20B hashed value. On the other hand, with increasing the number of flash chips, the amount of RAM space on SSD will also scale, providing sufficient space to accommodate the mapping tables of RR-SSD.

- **Replication Overheads.** Replicating a set of very popular values helps reducing the overheads associated with replication while providing high performance benefits. We report the ratio of replication writes to all writes in table 4.5. As reported in this table, our proposed mechanism incurs around 10% more writes compared to CA-SSD, on average. However, compared to a conventional SSD which does not employ content deduplication, we do not incur extra writes as we use up the write traffic saved by deduplication.

- **Table Look-up and Updates.** Modern SSDs have multiple cores and accommodate complicated scheduling and mapping mechanisms. Equipping SSDs with such strong computation power (e.g., with clock speeds over 2GHz), will limit the table look-ups in our design to few micro seconds, which is negligible compared to the actual read and program latencies.
Chapter 5  |  Trimming Read Tail Latency in SSDs

The selective replication and read re-direction technique presented in the preliminary chapter, relies on a content-addressable design in the SSD. In this chapter, we aim at proposing a read re-direction technique in the normal SSDs. To this end, we attempt to leverage the existing RAID capabilities embodied inside the high-end SSDs for performance and fault-tolerance. Apart from delivering low average response times, in such high-end SSDs which are more common in the large-scale production environments, minimizing read tail latencies (tail of up to 99th-percentile) are of very profound importance. Thus, in this chapter, we aim at developing a read re-direction mechanism in normal high-end SSDs with an emphasis on trimming read tail latencies. To this end, we first conduct a systematic analysis, removing one bottleneck after another, to study the root causes behind long tail latencies on a state-of-the-art high-end SSD. The focus of this chapter is mainly on developing architecture solutions to trim the long read tail latencies. To this end, we explore a more simple solution that leverages existing RAID groups into which flash chips are organized. Specifically, an adaptive scheduler, called ATLAS, is proposed, developed and evaluated, that dynamically figures out whether to wait or to reconstruct the data from other chips.

5.1 Characterizing Read Tail Latency

In this section we describe the experimental setup used to investigate tail latencies, and the resulting findings that help answer the "where", "why" and "how" questions
Figure 5.1: Impact of queue depth on IOPS and tail latency. $D_i$ refers to the queue depth of $i$. Increasing queue depth puts more stress on the device until it saturates IOPS.

when addressing Read tail latency in SSDs.

### 5.1.1 Experimental Setup

In order to study the latency of Reads and identify the sources of unpredictable and large latency, we conduct an extensive set of experiments on a state-of-the-art Samsung datacenter NVMe SSD. This SSD has 1TB of capacity with an NVMe interface, overcoming the SATA interface bottlenecks [65]. We study the latency of the entire I/O path (including kernel, driver, and device) and report both average and tail latencies for Reads. As the latency metric, we report mean, median, and 95th – 99.99th latency percentiles.

We have used *FIO* [66] to generate synthetic IO traffic with different patterns. In FIO, we have enabled direct option to bypass the page cache and measure raw performance of the device. To narrow down our region of study and isolate the impact of some FIO parameters, we set blocksize to 16K (device page size), and numjobs to 4. To determine queue depth parameter, we run an experiment (shown in Figure 5.1) through which we increase level of I/O concurrency by increasing the queue depth (iodepth in FIO), which leads to larger IOPS. However, beyond some
point (iodepth of 32), increasing queue depth only overwhelms the drive, resulting in long queueing delays in requests’ service. Thus, throughout our experiments we confine iodepth to 8∼32 to have a more meaningful latency measurement. Among other FIO options, we have used (i) libaio asynchronous I/O engine, (ii) 5-10 minutes for ramp_time and (iii) 10-20 minutes as runtime. Eventually, we use clat\(^1\) (completion latency) reported by FIO for our analysis.

Next, we analyze the latency of I/O requests by examining different I/O patterns using FIO.

### 5.1.2 Read Only Mode

First, our goal is to study the interaction of read requests with each other without any interference from Write and GC. To this end, we isolate the impact of Writes and GC on Read latency by running a 100% random read workload and reporting the latency numbers in Figure 5.2. This figure shows the latency distribution of read requests in milliseconds. There are two main observations from this figure: (i) in the average case, a read request is delayed in service by nearly 10X over the NAND flash page read latency, due to other reads; (ii) 99.99\(^{th}\)-percentile latency is around 4X the average latency. This indicates the presence of a long tail even when there are no Writes or GC.

**Where:** In order to figure out the bottleneck component in the I/O path, we

---

\(^1\)Time taken from submission of an I/O request to its completion.
conduct two experiments through which we put the same stress on the device. In the first, we capture only the device latency by setting the number of on-the-fly I/O units to 1, with 8 jobs. In other words, the requests submitted to the SSD will not be queued until they arrive at the device. Thus, the latency reported in this case is mainly caused by delays inside the device and does not relate to other components in the I/O path. The second case, on the other hand, has only 1 job but 8 on-the-fly I/O units for that job (queue depth of 8). In this case, the requests could be queued/delayed prior to reaching to the device. We draw the latency distributions of the two experiments in Figure 5.3 which shows nearly identical distributions for the two cases. This leads us to conclude that the device latency dominates the total latency of read requests\(^2\). Note that latency numbers in these experiments are different from that of Figure 5.2, since we have varied number of jobs and iodepth in these experiments in order to identify the bottleneck component.

**Why and How:** Figure 5.4 shows the time varying read latencies for the experiment reported in Figure 5.2. We zoom into a very short window of time (few milliseconds) for illustration. As outlined with a rectangular solid line in this figure, at any instant (i) multiple requests are completing with different latencies implying they are getting serviced from different underlying physical resources (e.g.,

\(^2\)We have also examined the same scenario for mixed read-write workloads and other number of jobs and iodepth, and observed the same trend in the latency distributions. Also we have used Blktrace [67] and by comparing the D2C latency with FIO output, we have reached the same conclusion.
different flash chips), and (ii) while most requests are getting serviced close to the average latency, there are few requests (lying in the $99.99^{th}$-percentile latency) which experience very high delays since they are getting serviced from a very congested flash chip, thus, \textit{Resource contention and load imbalance are an important source of large Read tail latency}. \footnote{Prior work has also reported such load imbalance \cite{68,69}.}

We next present the impact of Writes and GC in the following subsections.

### 5.1.3 Impact of Write

To observe the impact of write requests on the latency of Reads, we experiment with mixed read-write workloads with $20\% \sim 100\%$ read-write ratios (with uniformly random access pattern) and draw the read latency distributions in Figure 5.5.(a).

Note that in these experiments, by determining the \texttt{size} option in FIO, we do not...
let the device to enter GC phase. So no GC is invoked in these set of experiments.\textsuperscript{4}

The bottom line in this figure shows the read-only mode latency (as depicted in Figure 5.2). From this figure one can see that Read tail latency worsening is dramatically amplified in the presence of Writes (around 6X as compared to read-only mode). We conclude \textit{read-write interference is also a major contributor to the read tail latency}.

### 5.1.4 Impact of GC

Next, we attempt to measure the impact of GC on read tail latency. To this end, we run the same workloads (from Section 5.1.3) as preliminary experiment, and let the SSD proceed to GC phase. This is done by pre-conditioning the SSD before running the experiment (by sequentially and randomly writing on the entire device). Subsequently, we collect the results and present them in Figure 5.5.(b). It can be deduced from this figure that in the presence of GC the device is noticeably slower as well (there is 2.5ms increase which is almost in par with erase latency). More importantly, where comparing the 99.99\textsuperscript{th}-percentile latency (which is important in storage and database systems [3, 4, 70, 71]) in the topmost line (20% Read) of Figures 5.5.(a) and 5.5.(b), presence of GC does not impact the read tail latency since the two cases experience almost similar read tail latencies despite having very different average latencies. This observation is quite contrary to many prior observations [3, 8–11, 20, 70, 72], which have pointed to GC as being a primary contributor to a large tail.

### 5.1.5 Findings

Although SSDs are notably fast in general, they experience frequent high-latency episodes during the course of operation. This slowdown despite not being very important in low- and moderate-end SSDs, tends to be noticed in high-end SSDs which are used in the warehouse scale. Furthermore, by SSDs offering high parallelism and striping requests on multiple flash chips within the drive, even a tail

\textsuperscript{4}To ensure that no GC occurs in this experiment, we use the “secure erase” command to erase the SSD before running the experiment. Also, we measure the Write Amplification Factor (WAF) using the “smartlog” feature of NVMe. Measuring WAF of 1 is an indication of no GC occurrence.
delay in a fraction of requests/sub-requests\textsuperscript{5} may affect SLAs. Higher levels of parallelism can actually result in longer latency tail since a long delay seen from a flash chip can be magnified at the request’s service level\textsuperscript{6}. Based on a thorough analysis we conclude that, the source of large read tail latency is threefold: (i) read-read interference, (ii) read-write interference, and (iii) read-erase interference. However, we showed that read-write interference is the main source of read tail latency due to relatively larger latency of write/program operation in comparison with read operation, and high frequency of write/program operation as compared to erase operation.

Note that as discussed in [70], the tail at datacenter scale can be a result of many other factors such as thermal throttling, power optimization, and network layer traffic. The tail latency of SSDs can also be affected by some of these means such as thermal issues and power optimizations. However, in this work, we focus on the resource contention and GC costs (as contributors to read tail latency) in SSDs. More specifically, we propose to reduce the read-write interference in a RAID-enabled SSD through an adaptive scheduling mechanism. Lastly, we leave the read-read interference to the future work.

\textbf{5.2 Reducing Read-Write Interference}

To combat the read-write interference explained in the previous section, we propose to take advantage of existing capabilities embodied inside the SSD for fault-tolerance, along with an adaptive scheduling mechanism, to get close to the read-only latency for read requests. The fine-grained parity-based redundancy technique inside the SSD (aka RAID or RAIN in the context of NAND flash SSDs) is widely used by SSD manufacturers to provide higher reliability [65, 73–79]. Exploiting redundancy to gain superior performance in SSD has already been investigated by prior work [8, 14, 20, 80–84]. These optimizations are either (i) targeted to surpass average read latency—which does not necessarily trim the tail, (ii) incur high buffering and maintenance costs, or (iii) partially reduce the tail by alleviating GC costs. Flash on Rail [20] isolates the read requests, in an array of SSDs, by

\textsuperscript{5}A large request spans several pages, is divided into several page-sized sub-requests, each of which is directed at a flash chip

\textsuperscript{6}We can use the same reasoning as [70] wherein increasing number of servers results in higher variation in system-level service time.
dedicating a number of SSDs only to read requests. With SSDs continuing to scale and providing several terabytes of capacity, their approach cannot be readily employed as it is oblivious to the SSD’s internal architecture. Purity [84] takes the available redundancy into account when scheduling requests in All-Flash-Arrays. Their optimization is aiming at improving average read latency and is not capable of adapting itself to the variations in the workload and device behaviour. Tiny tail flash [8] and a proposal from Violin Memory [80], on the other hand, takes the internal architecture of SSD into account, and devises different mechanisms to improve read performance. The former only addresses GC costs when optimizing read tail latency. The latter, however, tries to reduce average read latency by tuning the write traffic –they only consider the queue length of writes when making scheduling decisions. We quantitatively compare our proposal with these two prior works in the evaluation section.

In this section, we are aiming at devising a mechanism to service the outlier read request blocked by an on-going write or GC, by exploiting this available redundancy. To this end, given an SSD with multiple channels and multiple flash chips per channel, we cluster flash chips into several independent groups. In each group, writes are confined to only one flash chip at any time, and rotating over other flash chips in the group in a Round-robin fashion. Reads targeting the program-busy flash chip, will be reconstructed having the parity blocks associated with each group stored in a parity-holding flash chip in that group. Note that we assume an LBA-based RAID implementation such as [73, 74] in the rest of this chapter and base our design on this assumption.

To better illustrate our proposal, we give an example in Figure 5.6. This figure illustrates a representative organization for a RAID-equipped SSD. Here, the SSD has 4 channels, each containing 4 flash chips. As shown in this figure, 4 flash chips are involved in a group (forming a 3+1 parity group) with one flash chip storing the parity blocks. Note that parity blocks can be allocated in a rotating manner similar to RAID-5. In this figure, however, we assume parity blocks all stored on one channel, for the simplicity of illustration. Next, we describe the details and constraints, with regards to this example, for different operations in our design.

**Write.** There is at most one active write flash chip in each group (i.e., one

---

7SSDs offer higher levels of parallelism by embodying multiple planes in each flash package, however, here we assume only one plane per flash chip. Our proposed mechanism can be easily adapted for higher number of planes per each flash chip.
program operation occurs in each group at any time), to enable reconstructing a read request targeting at a program-busy flash chip, by involving other flash chips in that group. Once the program operation completes, the next write to the same group can begin. Note that, while there is only one write point over any flash chip group (e.g., flash chip 2 in group 1), other groups can service write requests with the restriction of one write at any time in each group.

**Read.** Considering the example given in Figure 5.6, if a read request targets any flash chip in group 1 other than flash chip 2, it can be serviced through normal means. On the contrary, if a read request is targeting flash chip 2 which

Figure 5.6: A representative organization for RAID-equipped SSD.
is busy servicing a write request, the requested data from this flash chip can be reconstructed by (i) reading respective pages from flash chips 1, 3, and 4, and (ii), XORing the data read from these flash chips. This example also highlights the intuition behind the “one write at any time in each group” constraint, as reconstruction of a read blocked by an on-going write necessitates engagement of all other flash chips in the group. This would not be feasible, if any other flash chip in the group is busy with servicing a write.

**Parity update.** Considering group 1 in Figure 5.6, once writes complete on flash chips 1–3 of this group, the parity-holding chip (i.e., flash chip 4) is updated, if required. Then, this group begins servicing incoming writes. Once flash chip 4 is busy updating the required parity blocks, any read to flash chips 1-3 can be serviced without encountering any on-going program operation in these flash chips. It is of utmost importance to note that, while in each group we decouple the flash chip writes and parity updates, and delay the parity updates, this constraint need not be held globally with regards to all flash chips in all groups. In other words, each group operates independently and services requests oblivious to the other groups.

To better demonstrate the sequence of events occurring in each flash group, we provide Figure 5.7. As shown in this figure, in group x (in this example), first, writes are happening in the flash chips of this group. Once writes issued consecutively to the flash chips in this group are done, the parity-holding flash chip in this group, subsequently, updates the required parity pages depending on the RAID implementation. When parity updates are done, flash chips of this group can continue servicing writes.
5.2.1 Challenges

Even though this scheduling can reduce read-write interference considerably on flash chips, it has two main consequences: (i) confining writes to only a subset of flash chips impedes utilizing the high levels of parallelism provided in SSDs, thereby degrading write performance; and (ii) additional read operations required for reconstructing each read request that is blocked by an on-going write. We address these issues by first quantifying the impact of increased number of reads on tail latency, and then proposing an adaptive scheduling which significantly alleviates these overheads.

5.2.1.1 Extra Read Overhead

We conduct an experiment (with the setup explained in Section 5.1.1) in which the number of reads is increased by incrementing the number of read jobs in FIO. The latency distributions are plotted in Figure 5.8. In this figure, the bottom and topmost lines are 100% and 20% read (identical to Figure 5.5), respectively, each with 4 parallel I/O jobs. We repeat the 100% read experiment with increasing number of FIO jobs to 8 (doubling number of jobs) and plotting the latency results in the same figure (Figure 5.8). By incrementing the number of jobs in the read-only case, the tail increases linearly/sub-linearly, i.e., doubling the number of reads to the drive, will at most double the tail which is still far below the tail in the mixed
read-write case (topmost line in this figure)—the difference noted by a dashed arrow. Thus, even if our proposed mechanism does not perfectly capture the read-only case latency, it could still greatly mitigate read-write interference and reduce the tail as compared to the mixed read-write case. In fact, the dashed arrow on this figure shows the room for improvement for tail latency (99.99th-percentile). Hence, increasing the number of reads with our proposal, appears to be an acceptable trade-off towards substantially reducing the tail latency.

5.2.1.2 Write Performance Overhead

The proposed mechanism, despite promising substantial reduction in read tail latency, does not take the workload dynamism into consideration. For instance, given a write-only workload, all the underlying physical resources could be exploited to service the incoming write traffic. However, the proposed mechanism confines writes to only a subset of flash chips, leading to a potential drop in write performance. To prevent this situation, we augment our proposal by making the scheduling and read reconstruction decisions dynamic. This adaptive scheduling mechanism is described in the following subsection.

5.2.2 Adaptive Scheduling—ATLAS

There are two determining decisions to be made: (i) Write Scheduling, and (ii) Read Reconstruction. To cope with the aforementioned overheads, we make these decisions in a dynamic fashion, wherein the scheduling process adapts itself with the workload dynamism and device intrinsic. It works based on the following two intuitions:

- **To Read Reconstruct or Not**: If a read request is directed at a flash chip which is currently servicing a write request, but that request is close to finishing, the read request could wait until the on-going write completes, instead of being redirected. Thus, the read request can be serviced thereafter, without the need for any extra reads. However, if the wait time for this read (i.e., the remaining time of the current write) is beyond a specific threshold, the controller redirects the read to the other flash chips in the corresponding group, in order to reconstruct the demanded data.
• **To Issue the Write or Delay it:** Similarly, if there is no on-going write in a group or the on-going write is close to finishing, the next write to this group can be issued. Otherwise, the next write request has to wait until the remaining time of the on-going write in that group drops below a specific threshold, i.e., it gets close to finishing. This will lead to a situation where multiple writes can be serviced in a group, simultaneously—one write just started while others are close to completion. As previously described, our goal is to get close to the read-only mode. Thus, if an on-going write is close to finishing (its remaining time is less than latency of a few reads) the situation is akin to the read-only case where a read can be blocked by other reads.

Thus, three parameters are taken into account for the scheduling decisions in our proposal: the first is the time remaining to complete the current program operation in the target flash chip (e.g., flash chip $j$, in group $i$), $T_{\text{Residual}}^{i,j}$; the second is maximum time remaining to complete the current program operation in the group except the target flash chip, $T_{\text{MaxResidual},i}^{i\setminus\{j\}}$; and the third is the time taken to service a page-sized read request, $T_{\text{Read}}$. We incorporate these parameters in our proposed model and define the following inequalities:

$$T_{\text{Residual}}^{i,j} > \alpha \times T_{\text{Read}}, \quad (0 < \alpha), \quad (5.1)$$

$$T_{\text{MaxResidual},i}^{i\setminus\{j\}} > \alpha \times T_{\text{Read}}, \quad (5.2)$$

where, $\alpha$ is a positive constant (obtaining $\alpha$ is discussed later in this section). If either of these inequalities hold, implying that the remaining time of an on-going write in the group is larger than a specific threshold ($\alpha$ times the service time of a page-sized read request), the next write cannot be issued and has to wait until the remaining time goes below that threshold. For reads, however, if the remaining time of the current program operation on the target flash chip is smaller than the threshold (i.e., In.5.1 does not hold), no reconstruction is required. On the other hand, when Inequality 5.1 holds, the read request will be reconstructed only if the maximum remaining time of current operation on the rest of flash chips of group is less than that threshold. In other words, to make reconstruction decision, Inequality 5.1 should hold while (5.2) should not. The intuition behind this decision is to ensure that, other flash chips in the target group are available to participate in the read reconstruction.
Obtaining residual time. The residual time of the current operation to be completed in the target flash chip (e.g., flash chip \( j \) in group \( i \)) is calculated as:

\[
T_{i,j}^{\text{Residual}} = \text{Flag}_{\text{Write}} \times (\text{Latency}_{\text{Write}} - (T_{\text{Now}} - T_{\text{Start}}))
\] (5.3)

\( \text{Flag}_{\text{Write}} \) and \( T_{\text{Start}} \) are attributes of the current operation on the flash chip: \( \text{Flag}_{\text{Write}} \) is per-chip flag determining type of the operation ("0": read and "1": write); \( T_{\text{Start}} \) is a per-chip register holding the start time of the operation; and \( T_{\text{Now}} \) is the current clock time.

After calculating the residual time for \( N \) flash chips in group \( i \), the maximum residual time of the current operation on flash chips of this group (except \( j \)th flash chip), \( T_{i,\{j\}}^{\text{MaxResidual}} \), is calculated as below:

\[
T_{i,\{j\}}^{\text{MaxResidual}} = \max(T_{i,1}^{\text{Residual}}, \ldots, T_{i,j-1}^{\text{Residual}}, T_{i,j+1}^{\text{Residual}}, \ldots, T_{i,N}^{\text{Residual}})
\] (5.4)

where, \( T_{i,j}^{\text{Residual}} \) is residual time of the current program operation of \( j \)th flash chip in \( i \)th group from Eq. 5.3.

Calculating service time of a read. \( T_{\text{Read}} \) consists of two parts: (i) channel transfer time of a page, \( T_{\text{ChannelXfer}} \), and (ii) and the latency of a read operation, \( \text{Latency}_{\text{Read}} \), which is a constant. Therefore, we have:

\[
T_{\text{Read}} = T_{\text{ChannelXfer}} + \text{Latency}_{\text{Read}}
\] (5.5)

Obtaining \( \alpha \). In this threshold-based scheduling, writes can still suffer from the performance drop in a write-only (or periodically write-only) workload, since it prevents writes from utilizing the entire underlying physical resources even if they are idle. Therefore, we dynamically tune the threshold by adjusting \( \alpha \) (that relates to the remaining time of an on-going write request in each group) based on the workload behaviour. In other words, this threshold is gradually increased if no read is submitted to the drive for a long time interval, hence availing the underlying physical resources to service the incoming write traffic, and decreased when the read:write ratio increases.

We reveal our proposed Adaptive Tail Latency-tolerant Scheduling mechanism, called \( \text{ATLAS} \), in the diagram depicted in Figure 5.9. This diagram shows how a
read or write request is serviced. In each iteration, the scheduler picks a request and upon its type and the state of flash chips in the target group, decides whether or not to issue it (for Write), or reconstruct the demanded data (for Read), by checking inequalities 5.2 and 5.1. Moreover, the parameter $\alpha$ is updated in each iteration by either resetting it to the average read queue length\textsuperscript{8} (upon arrival of a read) or increasing it. To enable tuning $\alpha$ with respect to the workload variation, we employ another parameter, $\beta$, and increase $\alpha$ by multiplying it with $\beta$, where $\beta > 1$ (the perfect value of $\beta$ that works to be described in the experimental setup section).

\textsuperscript{8}Since our aim is to achieve the performance of read-only case for read requests, we set the minimum value of $\alpha$ be the average read queue length in previous time intervals. The average read queue length can also be extracted from benchmarking the read-only execution.
5.2.3 Handling Write Flush

Our scheduling mechanism is designed conservatively in order to minimize read-write interference. The devised technique is based on the assumption that SSD encompasses enough buffering space to temporarily absorb the incoming write traffic and gradually issue them on the flash chips. In practice, however, write traffic may occasionally fill up the available buffer space and trigger an enforced flush. This write flush may perturb our optimizations and cause temporary spikes in read latency. To avoid/minimize the occurrence of write flush, we aim at distributing such spikes in read latency, over time, taking the write traffic intensity into account. As an example, assume that we strive to reduce the $99^{th}$-percentile read latency. Therefore, if we let one read request in every 100 request be blocked by an on-going write, we can still provide the low tail of up to $99^{th}$-percentile. This little flexibility can be used to slightly increase the number of simultaneous writes allowed in each group, in order to improve write throughput as well as reducing the likelihood of enforced flush.

Inspired by the previous work [85] that attempts to spread the latency SLO violations over time for consolidated VM storage, we monitor the latency SLO violations and adjust the constraint on the number of write points in each flash chip group, denoted as $N_{WP}$, accordingly. Our goal here is to increase the number of write points in each group to improve write throughput while complying with $X^{th}$-percentile read latency objectives. After finding the the right value for $N_{WP}$, we can modify the decision regarding issuing the next write request on a flash chip group as follows:

$$\sum_{i=1}^{N} \mathcal{I}(T_{i}^{Residual} > \alpha \times T_{Read}) > N_{WP}, \quad (0 < \alpha).$$

(5.6)

In this inequality, $N$ refers to the number of flash chips in a group (excluding the parity flash chip), and $\mathcal{I}$ is the indicator function which returns 1 if its input condition holds. This inequality loosens the “one-write-point in each group” constraint by granting more flash chips to write requests, depending on write traffic and read latency SLOs, in the hope of eliminating/reducing the enforced flushes. Based on this inequality, the next write request will be issued on a flash group chip when there are at most $N_{WP}$ on-going writes (with remaining time of greater than $\alpha \times T_{Read}$) in that group. Thus, determining the appropriate value for $N_{WP}$ is of
utmost importance in order to achieve high concurrency for servicing writes. To dynamically adjust $N_{WP}$, we consider two criteria:

**$X^{th}$-percentile Read Latency:** Similar to [85], we assume that the number of SLO violations at each time is proportional to the number of requests completed until that time. For $X^{th}$-percentile, we define the upper bound of accumulated SLO violations allowed until time $t$, denoted as $U(t)$, as $NUM_{REQ}(t) \times (1 - X)$, where $NUM_{REQ}(t)$ refers to the number of requests completed until time $t$. Thus, the key is to monitor the actual SLO violations in the previous time intervals (until time $t$), denoted as $M(t)$, and compare it to $U(t)$. We intend to minimize the difference between these two parameters and increase the write pressure, subject to the condition $M(t) \leq U(t)$. Note that, as discussed in preliminary sections, our goal is to lower the read tail latency and get close to that of read-only case. Thus, the latency SLO for read requests can be either extracted from read-only execution, or specified by the user.

**Write Load:** The other parameter that is used to adjust $N_{WP}$ is the write load. An indication of write load is the write queue length. In a typical SSD, if write queue length gets larger than a pre-defined threshold (e.g., 80% of max), the SSD will flush the writes. In order to avoid reaching to that threshold, we set a maximum for the write queue length, denoted as $Q_{MAX}$, which is strictly lower than the SSD threshold. Thus, we take the difference between the average write queue length over time ($Q_{WR}$) and $Q_{MAX}$ into account to figure out the proper number of write points in each group for the next time interval. A larger difference in these two variables is an indication of lower write load, which in turn implies that smaller number of concurrent write points in each group can suffice to service writes. A smaller deviation of queue length from $Q_{MAX}$, on the other hand, can be used as a hint to increase the number of write points.

Based on these criteria, we define Equation 5.7 for extracting the right value of $N_{WP}$ for the next time interval:

$$N_{WP}(t + 1) = \begin{cases} 
1 & \text{if } M(t) > U(t), \\
\max(N_{WP}(t) - 1, 1) & \text{elseif } Q_{WR} < \sigma * Q_{MAX}, \\
N & \text{elseif } Q_{WR} > Q_{MAX}, \\
N_{WP}(t) + 1 & \text{elseif } 1 - M(t)/U(t) > \gamma, \\
N_{WP} & \text{otherwise.}
\end{cases}$$ (5.7)
We tune the value of $N_{WP}$ for the next time interval, by initiating it to 1 in the beginning. If condition $M(t) > U(t)$ holds, then the number of write points is set to its lowest value (i.e., 1), in order to provide lower tail latency for reads. The variable $\sigma$ (set to 0.2) is used to determine if the number of write points is higher than necessary. If so, $N_{WP}$ will be decremented (given that its minimum value is 1). On the other hand, if the queue length is larger than the maximum value allowed, $N_{WP}$ is set to its largest value (N) in order to exploit all available parallelism to service writes to prevent a later unexpected flush. Lastly, if $M(t)$ is larger than $U(t)$ by up to $\gamma$, then $N_{WP}$ is increased by 1. This is specifically done in our attempt to coordinate the concurrency of write requests provided that more number of long read latencies can be tolerated. We explain the experimental environment and parameters in the next section.

5.2.4 Handling Garbage Collection

To handle GC in ATLAS, we perform GC within each flash chip as opposed to performing GC across all flash chips - else it complicates the movement/placement of parity data. Integrating the intra-flash chip GC into our proposal is straightforward. To this end, inequalities 5.1, 5.2, and 5.6 are accordingly modified to bring GC into the picture. More specifically, when computing the term $T^{Residual}$ in Eq. 5.3, the remaining time of on-going GC, if any, is also considered. Thus, our proposed model takes GC into account when making any scheduling or read reconstruction decisions. Also, it updates the decision-making constraint to address minimizing read and GC interference with the restriction of one write or background activity at any time in each group. In other words, when a group is involved in any background operation, no writes shall be issued on any flash chips on that group. However, it is worth mentioning that, similar to the previous description for ATLAS, while a group is busy with a write or background operation, other flash chips in other groups could service writes or GC operations which are waiting to get serviced on flash chips associated with those groups.
Table 5.1: Characteristics of the simulated SSD.

<table>
<thead>
<tr>
<th>Evaluated SSD Configuration</th>
<th>NAND flash [87]</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 Channels and 8 Flash Chips per Channel, Channel Width = 8 bits, Chip Interface Speed = 533 MT/s (ONFI 4.0)</td>
<td>Page Size = 4KB, Metadata Size = 448B, Block Size = 256 pages, Planes per Die = 2, Dies per Chip = 4, Flash Chip Capacity = 16GB, Read Latency = 75µs, Typical Program Latency = 500µs, Erase Latency = 2.5ms</td>
</tr>
</tbody>
</table>

5.3 Experimental Setting

Evaluation Platform Similar to preliminary chapter, we use SSDSim to model a state-of-the-art SSD. We have modified SSDSim to model a RAID-5 implementation inside the SSD where flash chips are treated as independent storage units and multiple flash chips from different channels (as shown in Figure 5.6) are chosen to form a RAID group. More specifically, we employ 3+1 parity policy, performance of which, as reported by Micron [79], is comparable to that of a no-RAID SSD. To mitigate wear-levelling issues emerging from uneven distribution of wear across parity pages, we can adapt a migration approach similar to [8], where hot parity pages are periodically migrated within the parity group or to the other groups. Subsequently, we have augmented the RAID-equipped SSD with additional components and functionalities required for our proposed ATLAS scheduling mechanism. In addition, we employ a 256 MB battery-backed RAM [86] to temporarily buffer the write traffic.

Configurations Studied The baseline SSD consists of eight channels, each connected to eight NAND flash chips. Table 5.1 provides specifications of the modeled SSD (which is very similar to [64]) along with parameters of the baseline configuration.

We compare the performances of five systems:

1. **Baseline** uses default scheduling at the controller\(^9\) without supporting any RAID or tail-tolerance implementation.

2. **ATLAS** applies our proposed scheduling mechanism with \(\beta = 1.001\)\(^10\). The

\(^9\)First Read-First Come First Served (FR-FCFS) [30] at flash scheduling unit.

\(^10\)This value is set to a constant via extensive experiments, for simplicity. It can also be dynamically altered with minimal changes in the algorithm.
Table 5.2: Characteristics of the evaluated I/O traces.

<table>
<thead>
<tr>
<th>Trace</th>
<th>RD-WR (%)</th>
<th>Trace</th>
<th>RD-WR (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MySQL</td>
<td>18.16</td>
<td>MSNFS</td>
<td>67.20</td>
</tr>
<tr>
<td>Cassandra</td>
<td>35.95</td>
<td>Radius</td>
<td>15.85</td>
</tr>
<tr>
<td>1Cass1MySQL</td>
<td>8.96</td>
<td>Prxy</td>
<td>65.48</td>
</tr>
<tr>
<td>2Cass2MySQL</td>
<td>45.69</td>
<td>USR</td>
<td>82.24</td>
</tr>
<tr>
<td>RocksDB</td>
<td>41.03</td>
<td>RSRCH1</td>
<td>3.05</td>
</tr>
<tr>
<td>SRC</td>
<td>30.33</td>
<td>RSRCH2</td>
<td>65.69</td>
</tr>
</tbody>
</table>

time intervals for tuning different parameters is set to 100 ms. Moreover, we set the value of $\gamma$ to 0.5 based on our sampling period and the bandwidth of device.

3. **Read-only** is a system in which read and write latencies are assumed to be identical. For comparison, writes incur same latency as reads (by manipulating the configurations of SSDSim) to provide an ideal read-only case where no read is blocked by a write.

4. **Violin Memory**, prior work [80] that reduces read-write interference by reconstructing blocked reads, taking write traffic into account for their scheduling decisions.

5. **TinyTail Flash**, prior work [8] that reduces read tail latency by circumventing the read requests which are blocked by GC.

For each of the evaluated systems, we report the amount of reduction in read latency (average and tail) as our performance metric.

**I/O Workload Characteristics** To evaluate our proposed approach we use a diverse set of real-world block traces from database applications such as MySQL, Cassandra [88], and RocksDB [89]. Each trace is collected by running the corresponding application for several minutes to few hours, in a system equipped with Samsung datacenter SSD. In addition, to analyze a wide spectrum read-world usages, we run some instances of these applications simultaneously (e.g., 1 instance of MySQL and 1 instance of Cassandra). Moreover, we use seven block traces collected from Microsoft production and enterprise servers [49,90]. Table 5.2 summarizes the studied workloads and their read to write ratio (noted as RD-RW (%)).
Figure 5.10: (a) Average and (b) tail latency of ATLAS and Read-only systems, normalized to the latencies of the baseline system.

5.4 Evaluation Results

In the next three subsections we analyze the performance results for the 12 workloads studied. For each, we present the latency results normalized to the baseline system (Figures 5.10 and 5.11) in terms of (i) average read latency, (ii) 99.99th percentile read latency, and (iii) average write latency. Note that in these figures, x-axis is sorted based on read-to-write ratio, from lowest to highest.
5.4.1 Average Read Latency Improvement

Figure 5.10.(a) reports the average read latency for our proposal (ATLAS) along with the Read-only case (ReadOnly), for comparison. We also report the latency improvement achieved by the prior works, Violin Memory (Violin) and TinyTail Flash (TT-Flash), for comparison. These results are normalized to the average read latency in the baseline system. Although our proposal is targeted to improve read tail latency, it also leads to reduction in mean read latency by attempting to reduce the read-write interference, thereby preventing reads getting blocked by an on-going write request. In particular, reducing the latency of the laggard read request/sub-request (when requests span multiple flash chips), improves the average read latency since a larger request is considered as complete only when all of its sub-requests are serviced.

On average, ATLAS reduces the mean read latency by around 47%, with RSRCH1 benefiting the most from it (87% reduction in mean read latency). This workload has a relatively low number of reads, whose acceleration results in substantial improvements, while not incurring high overheads due to read reconstruction. Overall, ATLAS provides larger mean read latency improvements for write-intensive workloads, due to higher probability of a read being blocked by an on-going write request in these workloads. This is evident in Figure 5.10.(a), where latency improvements are roughly decreasing when moving to the right. Among these workloads, USR gains less improvement (21%) since it is a read-intensive workload. However, it is worth noting that in this workload, ATLAS performs very close to the read-only pattern, due to less interference from writes in this figure which leads to less consequent read reconstruction overhead. Furthermore, as shown in this figure, TT-Flash and Violin also reduce the mean read latency. However, TT-Flash improvements are still well below what ATLAS provides, since TT-Flash only considers circumventing the read requests which are blocked by GC, and does not effectively reduce read-write interference. Violin, on the other hand, reduces the read-write interference, leading to reduction in mean read latency. It gives better mean read latencies in comparison with TT-Flash, and in some instances it performs slightly better than ATLAS (such as in USR and RSRCH2). However, as we show next, Violin’s optimizations are not enough for lowering very high percentiles of read latency and its improvements are well beyond that of ATLAS.
5.4.2 Read Tail Latency Improvement

Figure 5.10.(b) reports the 99.99\textsuperscript{th} percentile read latency normalized to that of the baseline system. As demonstrated in this figure, ATLAS provides significant improvements in read tail latency – around 4X on average, and by up to 10X in workloads such as RSRCH\textsubscript{1} and 1Cass-1MySQL. Comparing Figures 5.10.(a) and 5.10.(b), ATLAS provides much larger improvements in read tail latency, than average latency. This is mainly due to optimizing the latency outlier read request. Also, compared to the average read latency results, ATLAS captures most of the achievable benefits as it reduces read tail latency nearly as much as read-only case. This holds in most cases such as Cassandra, Prxy, and RSRCH\textsubscript{2}, falling short by less than 5\% difference of the read-only case. However, in cases such as Radius, SRC and 2Cass-2MySQL, the benefits of read-only outweighs those of ATLAS. The reason is (i) the presence of very large requests in these workloads, and (ii) not devoting the entire set of physical resources to the read traffic (some reserved for bookkeeping to hold parity information). Note that in the read-only case, in fact, no RAID functionality is implemented, thus no parity information is stored and all the resources are available to read requests. We believe that a more efficient implementation of RAID will significantly blur this gap, however, we leave evaluating ATLAS with other RAID implementations to future work.

Similar to mean read latency results, TT-Flash is not able to achieve the tail latency improvements that ATLAS provides. The main reason is that, as we discussed in the characterization section, GC is not the main contributor the read tail latency, thereby eliminating GC is not expected to completely mitigate long read tail latencies. Violin, on the other hand, also improves read tail latency compared to the baseline. But as discussed in preliminary sections, its reduction in read tail latency is relatively lower than benefits it provides for mean read latency, as it only considers write traffic when making scheduling decisions, which is yet not sufficient to trim high percentiles (e.g., 99.99\textsuperscript{th}) of read latencies.

5.4.3 Write Latency Degradation

As described in Section 5.2.1.2, confining write requests to only a subset of flash chips has performance consequences. Figure 5.11 reports the degradation in average write latency by plotting the average write latency of ATLAS for each workload,
Figure 5.11: Average write latency of ATLAS, normalized to the average write latency of the baseline system.

normalized to the average write latency in the baseline system. Our proposal, despite marginally impacting average write latency in many cases (e.g., RSRCH1, MySQL, and Prxy), increases average write latency by around 35.6%, on average. It almost doubles in a few workloads such as Cassandra and RocksDB. In these two workloads, ATLAS is not able to capture all the dynamism in the workload. Using a larger non-volatile buffer such as a battery-backed RAM, which is increasingly used as a non-volatile store [86], can effectively hide the write slowdown.

5.4.4 Sensitivity to Different RAID Settings

To lower the capacity requirements associated with RAID, SSD vendors may trade-off capacity overhead for less reliability, by offering other configurations for SSD’s internal RAID. To this end, they increase the size of parity group and offer 7+1p and 15+1p RAID groups, through which for every 7 and 15 SSD pages, respectively, only one parity page is maintained. When the RAID setting is changed and more number of pages are grouped to form a parity group, our ATLAS enhancements can be affected, since the rise in read reconstruction costs can shrink ATLAS benefits. To investigate the impact of different RAID settings on the efficacy of our proposal, and study the scalability of our proposed approach, we reveal the amount
of reduction in read tail latency (99.99th-percentile) for different RAID settings, across evaluated workloads, in Figure 5.12. As presented in this figure, increasing the size of parity group can lower the benefits of ATLAS, however, our proposal can still effectively trim the read tail latency even for 15+1p setting (by around 3X on average). As already discussed in Figure 5.8, with increasing number of reads, the tail increases linearly/sub-linearly, which is still far below the tail latency induced by writes and GC. Thus, our enhancements appear to be an acceptable trade-off towards substantially reducing the tail latency.
Chapter 6  
Related Work

**Scheduling Efforts:** New schedulers order incoming requests to take advantage of high storage parallelism and at the same time resolve the resource contention problem. A number of these schedulers are implemented at the host side (such as [91–93]) that send the requests to the SSD. Others [7, 12, 29, 30, 68, 69, 94–96] operate inside the SSD and distribute requests across various internal resources (channels, chips, etc). Jung et al [12] proposed a QoS-aware and GC-aware scheduler, considering the low-level contention in SSD, which strives to reduce the resource contention by redistributing the overheads of GC across non-critical I/O requests. In comparison, to eliminate the contention problem for read requests, [97, 98] proposed SSD caches and pre-fetching mechanisms.

**Replication and Read Re-direction:** Prior work have addressed replication either as a means to improve reliability [61] or to boost performance [25, 52]. The former discusses object request popularity in distributed storage systems equipped with HDDs, and attempts to customize replication degree to enhance the system availability. These optimizations are not directly applicable to a system with SSDs. PBLRU [52] as discussed in preliminary section, does not consider the value popularity when making replication decisions, resulting in lower improvements compared to our proposed RR-SSD mechanisms. Moreover, [25] operates on an array of SSDs and proposes inter-SSD replication. This study does not take the internal architecture of SSD and chip-level parallelism into account. Also their replication decision is not based on value popularity. [99] on the other hand, explores the opportunity of re-directing read requests by exploiting the existing redundancy in the SSD (provided for the reliability purposes). This work is completely orthogonal to our proposed approach and can be combined with our proposal to provide higher
improvements.

**Optimizing Write and GC:** Prior studies concentrate mostly on reducing the write traffic to enhance lifetime and performance. Apart from studies on spatial- and temporal-locality [100–103], some previous papers [1, 24] provide several methodologies to help reduce write traffic to the SSD by exploiting *value locality* and *data de-duplication* [53, 104–109]. In the context of SSDs, existing works propose several enhancements, primarily to the FTL, to avoid redundant writes on the SSD. Gupta et al [24], study several workload characteristics from the value popularity point of view. They attempt to remove the redundant writes by augmenting the FTL with required mapping tables. CAFTL [1], on the other hand, focuses on employing different sampling and mapping techniques to reduce overheads associated with mapping tables and the hashing mechanism. We want to emphasize that, all these techniques, have focused on optimization for writes, while reads have not received such attention. Thus, we fill this void by proposing a read-redirection mechanism through which reads can enjoy flexibility of selecting the target flash chip to read from.

**Reducing Read Tail Latency:** The importance of minimizing the tail has drawn a lot of recent attention from overall system capacity viewpoints, and on the SSD latency front as well [3, 8–15, 110, 111] since storage is a dominant component of many client-facing applications. Prior work on SSD tail latencies have primarily focused on (i) Garbage Collection [3, 8–12] and have tried to mitigate the GC-induced tail; and (ii) read-write interference and has studied different solutions [20–22, 112–116]. At the device level, prior efforts [10,18,19] have proposed pre-empting writes/erases in favor of servicing reads. This technique, even though currently employed by SSD vendors, as a prior study has shown [8] cannot completely eliminate the large read tail latency.
Chapter 7  |  Concluding Remarks

7.1 Summary of Contributions

In this dissertation, we propose design, implementation, and evaluation of different scheduling and replication mechanisms to address the performance issues in emerging SSD architectures. Our proposed enhancements can be incorporated in device firmware to effectively reduce response times from SSDs.

Addressing Non-uniformity Within SSD Requests (Chapter 3)

With SSDs offering high levels of internal hardware parallelism, SSD requests span multiple flash chips to get faster service. Applications are, on the other hand, sending variable-sized requests to the SSD, which does not necessarily match with the offered hardware parallelism, leading to a considerable load imbalance, which in turn skews the completion times of sub-requests of each request. To address this non-uniformity within request’s service, we presented Slacker [69], a mechanism for estimating and exploiting the slack present in any sub-request while it is waiting in the queue of a flash chip. We have shown that Slacker provides fairly accurate slack time estimates with low error percentages. Slacker incorporates a simple heuristic that avoids coordinated shuffling of multiple queues, allowing incoming sub-requests at each queue to independently move ahead of existing ones with sufficient slack. This can benefit incoming requests without impacting the completion times of existing ones. We next augmented Slacker with service-time scheduling decisions, and presented Slacker++, to achieve time-sharing of available flash chips when issuing different requests. The proposed enhancements leads to significant reduction
in SSD requests’ response time.

Leveraging Content Popularity for Read Re-direction in SSDs (Chapter 4)
A straightforward way to address resource contention problem in modern SSDs is to evenly distribute the incoming traffic across all chips, making them load-balanced. This needs to redirect both reads and writes. While implementing write redirection is easy (due to no-write-in-place property of flash memories), read redirection is not feasible in conventional SSDs. First, we showed that, although Ideal-All-Chip Replication (i.e., replicating each data everywhere) gives an impressive performance improvement, it is very costly in terms of performance, lifetime and storage space. Alternatively, we devised a “content popularity-based selective replication” mechanism, to reduce the costs associated with replication, while still providing read redirection for a majority of read requests. The proposed design is called Read-Replicated SSD (RR-SSD) [117], which tries to answer five complementary questions: (1) which data to replicate?, (2) how much space for replication?, (3) where to replicate?, (4) when to replicate?, (5) how to replicate?

Trimming Read Tail Latency in SSDs (Chapter 5)
We addressed the problem of read re-direction as well as trimming the read tail latency in high-end SSDs. Minimizing read tail latency and delivering deterministic I/O in high-end SSDs has recently become a very important problem. Towards achieving our re-direction capabilities for read requests while effectively trimming read tail latencies, we conducted a careful characterization on a state-of-the-art high-end SSD to figure out that (i) device latency dominates the total latency of read requests, and (ii) resource contention – read-write interference in particular – is the major contributor to read tail latency. Then, we took advantage of existing RAID capabilities to reconstruct the read data, rather than waiting, while a long latency operation is ongoing. To overcome the reconstruction overheads of our proposed approach, we proposed a complementary scheduling mechanism (called ATLAS) through which read tail latency is considerably reduced. Our proposed techniques achieved high reduction (by up to 4X) in 99.99th-percentile read latency.
7.2 Future Work

The work that has been accomplished in this dissertation has made notable steps towards scheduling SSD requests and improving SSD performance. Nonetheless, there are several interesting directions to enhance and advance our current work.

7.2.1 SSD Scheduling Enhancements

Due to its unique characteristics, scheduling SSD requests has been very challenging over the years. The scheduling and data replication mechanisms developed in this dissertation can, to a great extent, alleviate the load imbalance and resource contention caveats. However, with SSDs continuing to scale and embodying higher degrees of parallelism, the load imbalance problem continues to hinder their performance efficiency. In the future, the scheduling of SSD requests can be improved by: (i) a better prediction of device non-determinism by employing a more complicated online statistical model to predict the device behaviour under a given load, leveraging more processing power on the SSD, and (ii) workload prediction by employing a statistical and learning-based models. A more accurate stochastic modeling of device and workload, taking the underlying hardware idiosyncrasies into account, can help extract better performance, and also help reduce the overheads associated with replication of the data as storage devices are highly sensitive in terms of their to cost per gigabyte.

7.2.2 Distributed Storage

With SSDs employed in large-scale and scale-out systems, their non-determinism and performance issues can be magnified at the system level, and hence introduce new challenges to the system. Moreover, SSDs can be also used as a caching layer in such systems where they are exposed to different workload patterns with different bandwidth and latency demands. The proposed mechanisms in this dissertation can be directly/indirectly leveraged towards addressing performance issues of SSDs in scale-out systems by: (i) devising high-level scheduling mechanisms, taking into account the fine-grained scheduling techniques developed in this work, including slack-aware and tail latency-tolerant scheduling, (ii) trading off inter- and intra-SSD parallelism in accordance to the application demands to boost performance, and
(iii) trading off inter- and intra-SSD replication.

### 7.2.3 Emerging Non-volatile Memory Technologies

Emerging non-volatile memories are receiving more acceptance and are projected to be more widely employed as the memory, caching layer for storage, and even storage medium. Such technologies have different characteristics as compared to NAND Flash. Thus, a new set of optimizations should be devised in order to overcome the challenges in such technologies and accelerate their adoption. In emerging non-volatile technologies, load imbalance and resource contention is expected to still remain a major performance culprit. Inspired by the enhancements developed in this dissertation to address load imbalance and resource contention, a better scheduling and data replication can be devised for such technologies to expedite data accesses in a cost-effective manner. Moreover, such non-volatile memories can be used in front of SSD as a caching layer. Thus, trading off the capacity and cost per gigabyte for emerging NVM technologies and NAND Flash SSD is of utmost importance, which is a future direction to be explored.
Bibliography


URL http://doi.acm.org/10.1145/1837915.1837921


URL http://doi.acm.org/10.1145/3079856.3080236


URL http://doi.acm.org/10.1145/176979.176981


105


Vita

Nima Elyasi

Nima was born in Gachsaran, a small town in southwest of Iran, in 1991. He completed his high school in mathematics where he found himself highly interested in math and computer programs. This led him to study computer engineering at Sharif University of Technology in Iran, where he received his BS. degree in 2013. Nima arrived at United States in August of 2013 to embark on his graduate studies in computer science and engineering, at the Pennsylvania State University. During his graduate he studies, he worked as a research assistant where he explored various projects on emerging storage devices and data-intensive applications. He has contributed to the computer science and engineering society by multiple publications in the area of his studies. Nima has finished his graduate studies at Pennsylvania State University in March 2019, and will join Samsung Semiconductor Inc., as a senior system architecture engineer, where he will make endeavours in architecting and devising performance-enhancement mechanisms for large-scale data-driven applications.