Doctoral Thesis

An Autonomic SSD Architecture

자율 관리 SSD 아키텍쳐

February 2018

Seoul National University
Department of Computer Science and Engineering
Bryan S. Kim

An Autonomic SSD Architecture

자율 관리 SSD 아키텍쳐

지도교수 민 상 렬

이 논문을 공학박사 학위논문으로 제출함

2017년 11월

서울대학교 대학원

컴퓨터 공학부

김 석 준

김석준의 박사학위논문을 인준함

2017년 12월

위 원 장 이 재 진 (인)

부위원장 민 상 렬 (인)

위 원 양 현 석 (인)

위 원 최 종 무 (인)

위 원 엄 현 상 (인)

Abstract

Bryan S. Kim
Department of Computer Science and Engineering
Seoul National University

From small mobile devices to large-scale storage arrays, flash memory-based storage systems have gained a lot of popularity in recent years thanks to flash memory's low latency and collectively massive parallelism. However, despite their apparent advantages, achieving predictable performance for flash storages has been difficult. User experiences and large-scale deployments show that the performance of flash storages not only degrades over time, but also exhibits substantial variations and instabilities. This performance unpredictability is caused by the uncoordinated use of resources by competing tasks in the flash translation layer (FTL)—an abstraction layer that hides the quirks of flash memory. As more FTL tasks are added to address the limitations of flash memory, guaranteeing performance will become increasingly difficult.

In this dissertation, we present an autonomic SSD architecture that self-manages FTL tasks to maintain a high-level of QoS performance. In this architecture, each FTL task is given an illusion of a dedicated flash memory subsystem of its own through virtualization. This resource virtualization allows each FTL task to be implemented oblivious to others and makes it easy

to integrate new tasks to handle future flash memory quirks. Furthermore, each task is allocated a share that represents its relative importance, and its utilization is enforced by a simple and effective scheduling scheme that limits the number of outstanding flash memory requests for each task. The shares are dynamically adjusted through feedback control by monitoring key system states and reacting to their changes to coordinate the progress of FTL tasks.

We demonstrate the effectiveness of the autonomic architecture by implementing a flash storage system with multiple FTL tasks such as garbage collection, mapping management, and read scrubbing. The autonomic SSD provides stable performance across diverse workloads, reducing the average response time by 16.2% and the six nines QoS by 67.8% on average for QoS-sensitive small reads.

Keywords : SSD, QoS, scheduling, control

Student Number : 2014-31116

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Advent of flash memory-based storage systems

Flash memory-based storage systems have become popular across a wide range of applications from mobile systems to enterprise data storages. Flash memory's small size, resistance to shock and vibration, and low power consumption make it the *de facto* storage medium in mobile devices. On the other hand, flash memory's low latency and collectively massive parallelism make flash storage suitable for high-performance storage for mission-critical applications. As multi-level cell technology [3] and 3D stacking [52] continue to lower the cost per GB, flash storage will not only remain competitive in the data storage market, but also will enable the emergence of new applications in this *Age of Big Data*.

However, flash memory has limitations that need to be addressed for it to be used as storage. First, it does not allow in-place updates, mandating a mapping table between the logical and the physical address space. Second, the granularities of the two state modifying operations—program and erase—are different in size, making it necessary to perform garbage collection (GC) that copies out valid data to reclaim space. Furthermore, data stored in memory can be corrupted, requiring reliability enhancement

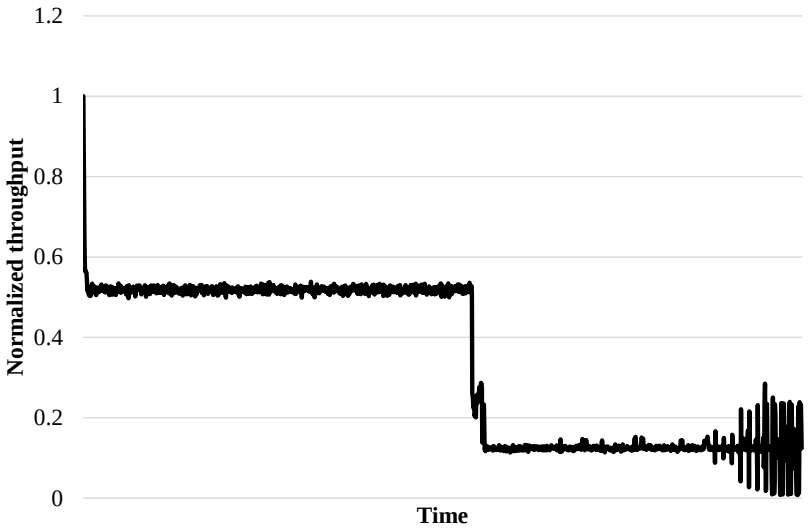


Figure 1: Performance drop and variation under 4KB random writes.

techniques to maintain data integrity. These internal management schemes, collectively known as the flash translation layer (FTL) [15], hide the limitations of flash memory and provide an illusion of a traditional block storage interface.

1.2 Research motivation

Large-scale deployments and user experiences, however, reveal that despite its low latency and massive parallelism, flash storage exhibits high performance instabilities and variations [12, 22]. Garbage collection has been pointed out as the main source of the problem [12, 30, 34, 36, 67], and Figure 1 illustrates this case. It shows the performance degradation of our SSD model under small random writes, and it closely resembles measured results from commercial SSDs [28, 53]. Initially, the SSD's performance is

good because all the resources of the flash memory subsystem can be used to service host requests. But as the flash memory blocks are consumed by host writes, GC needs to reclaim space by compacting data spread across blocks and erasing unused blocks. Consequently, host and GC compete for resources, and host performance inevitably suffers.

However, garbage collection is a necessary evil for the flash storage. Simply putting off space reclamation or treating GC as a low priority task will lead to larger performance degradations, as host writes will eventually block and wait for GC to reclaim space. Instead, garbage collection must be judiciously scheduled with host requests to ensure that there is enough free space for future requests, while meeting the performance demands of current requests. This principle of harmonious coexistence, in fact, extends to every FTL task. Map caching [20] that selectively keeps mapping data in memory generates flash memory traffic on cache misses, but this is a mandatory step for locating host data. Read scrubbing [21] that preventively migrates data before data corruption also creates traffic when blocks are repeatedly read, but failure to perform its duty on time can lead to data loss. As more FTL tasks with unique responsibilities are added to the system, it becomes increasingly difficult to design a system that meets its performance and reliability requirements [18].

1.3 SSD design challenges

Meeting performance requirements in flash storage have three main challenges:

- *Increasing complexity of the FTL.* As new quirks of flash memory are introduced, more FTL tasks are added to hide the limitations, thereby increasing the complexity of the system. Furthermore, existing FTL algorithms need to be fine-tuned for every new generation of flash memory, making it difficult to design a system that universally meets performance requirements.

- *Contention for the flash memory subsystem resources.* Multiple FTL tasks generate sequences of flash memory requests that contend for the resources of the shared flash memory subsystem. This resource contention creates queueing delays that increase response times and causes long-tail latencies.

- *Dynamic change in the importance of FTL tasks.* Depending on the state of the flash storage, the importance of FTL tasks dynamically changes. For example, if the flash storage runs out of free blocks for writing host data, host request handling stalls and waits for garbage collection to reclaim free space. On the other hand, with sufficient free blocks, there is no incentive prioritizing garbage collection over host request handling.

1.4 Dissertation contributions

In this dissertation, we present an autonomic SSD design called *AutoSSD* that self-manages FTL tasks to maintain a high-level of QoS performance. In our design, each FTL task is given a virtualized view of the flash memory subsystem that hides the details of flash memory request scheduling

from the FTL. Each task is allocated a share that represents the amount of progress it can make, and a simple yet effective scheduling scheme enforces resource arbitration according to the allotted shares. The shares are dynamically and automatically adjusted through feedback control by monitoring key system states and reacting to their changes. This achieves predictable performance by maintaining a stable system state. We show that for small read requests, AutoSSD reduces the average response time by 16.2% and the six nines (99.9999%) QoS by 67.8% on average across diverse workloads, with garbage collection, mapping management, and read scrubbing running concurrently. The contributions of this dissertation are as follows:

- *Virtualization of the flash memory subsystem.* Each FTL task is given an illusion of a dedicated flash memory subsystem, thereby not only decoupling algorithm and scheduling, but also making each task oblivious to others. This effectively frees each FTL task from having to worry about resource conflicts with and access patterns of other tasks.

- *Effective scheduling mechanism for share enforcement.* Each task is given a share that represents its relative importance, and the resources of the flash memory subsystem are arbitrated by a simple and effective scheduler that enforces the assigned share. The non-preemptive version of the scheduler enforces the assigned share by limiting the number of outstanding requests for each task; for the preemptive counterpart, the scheduler tracks the number of outstanding requests that preempts other requests, and limits the number of allowed preemptions for each task.

- *Adaptive scheduling policy based on feedback control.* Each task's share is adjusted reactively to the changes in key system states. System states such as the number of free blocks and the maximum read count reflect the stability of the flash storage, and the share controller maintains a stable system state by managing the shares.

1.5 Dissertation layout

The remainder of this dissertation is organized as follows. Chapter 2 gives a background on flash memory and flash translation layer. Chapter 3 presents the overall architecture of AutoSSD and discusses our design choices. Chapter 4 describes the evaluation methodology, and Chapter 5 presents the experimental results. Chapter 6 outlines related work, and Chapter 7 concludes and discusses future work.

Chapter 2

Background

In this chapter, we briefly describe the operation details and reliability trends of flash memory, and outline FTL tasks required to hide its limitations.

2.1 Flash memory

2.1.1 Flash memory organization

Flash memory's density has been increasing exponentially and the organizational complexity has been increasing as well. Figure 2 illustrates the organization of a generic flash memory used in flash storages. Flash storages employ multiple NAND flash *packages*, and each physical package contains multiple *chips* with separate I/O pins that allow them to operate and perform I/O activities independently. Multiple chips may be connected to a shared bus, commonly called a *channel*, and all I/O activities including command, address, and data transfer communicate via the channel. In each chip, there are multiple *dies* that share I/O pins but can perform independent internal operations. Within each die is a set of *blocks*, and a block is the unit for a flash erase operation. Blocks are grouped into *planes*, and blocks from different planes may operate simultaneously under a set of constraints. Within each block is a set of *pages*, and a page is the unit for a flash pro-

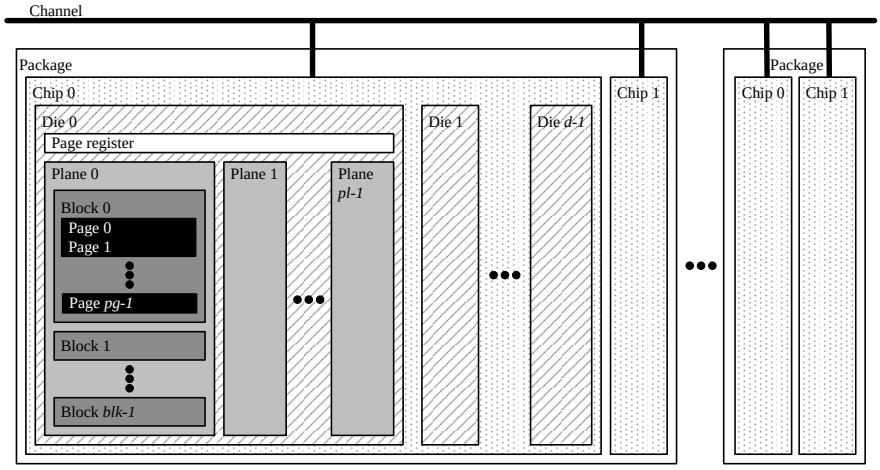
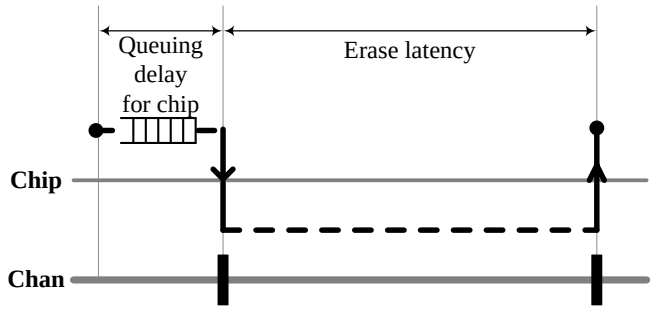


Figure 2: Flash memory organization.

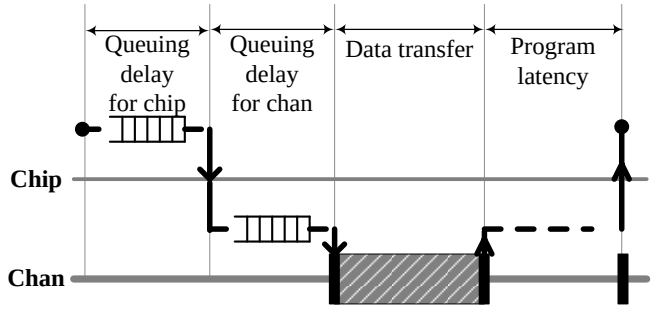
gram and read operations. The non-volatile memory where the data is stored is called the *flash array*, and the I/O buffer that acts as a staging area for data to and from the chip is called the *page register*. For simplicity, we abstract the complexity of the flash memory organization and refer the minimum independent operating unit as a chip.

2.1.2 Flash memory operations

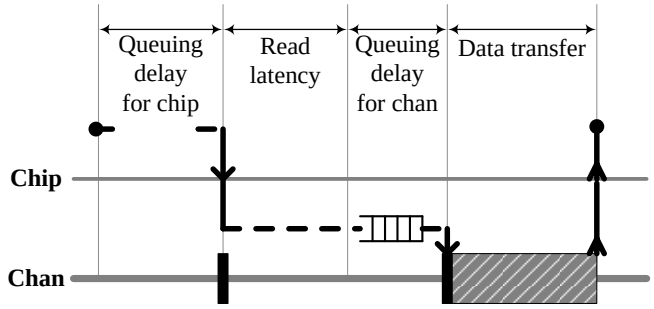
Ozone flash controller [48] abstracts much of the flash memory's organization and operation details, and services operations in an out-of-order fashion for maximum throughput. Figure 3 illustrates the operation diagrams for erase, program, and read under a generalized memory controller. Queueing delay represents the wait time caused by multiple operations sharing a channel or a chip. As shown in Figure 3a, an erase operation starts by sending the command and address to the chip at which point the chip becomes busy. All the bits in the selected block are written to 1 during the



(a) Flash memory erase operation.



(b) Flash memory program operation.



(c) Flash memory read operation.

Figure 3: Flash memory operation diagrams. The dark rectangles on the channel represent command and address cycles; the shaded rectangles, data transfers. Queueing delays represent delays caused by multiple operations sharing flash memory resources.

erase operation, and the operation is complete when the chip becomes ready again. A program operation, as shown in Figure 3b, also starts by sending the command and address for the page to program, and then transfers the data to write. Once all the data have been written to the flash memory's internal page register, another command is sent to write the data from the page register to the flash array, where the data is stored permanently. The chip is busy while the flash array is being written. At the end of the erase or program operation, the status of the chip can be read to confirm the success or failure of the operation. Lastly shown in Figure 3c, a read operation starts by sending the command and address to the chip, and the chip becomes busy while the data in the flash array is read into the page register. Once the chip becomes ready again, the data can be read out from the page register.

2.1.3 Error characteristics of flash memory

While extracting the maximum parallelism in the flash memory subsystem can be delegated to the flash memory controller, hiding the error-prone nature of flash memory can be challenging when relying solely on hardware techniques such as error correction code (ECC) and RAID-like parity schemes. Data stored in the flash array may become corrupt in a wide variety of ways. Bits in a cell may be disturbed when neighboring cells are accessed [17, 56, 63], and the electrons in the floating gate that represent data may gradually leak over time [6, 47, 63]. Sudden power loss can increase bit error rates beyond the error correction capabilities [63, 71], and error rates increase as flash memory blocks wear out [6, 17]. As flash memory becomes less reliable in favor of high-density [18], more sophisticated

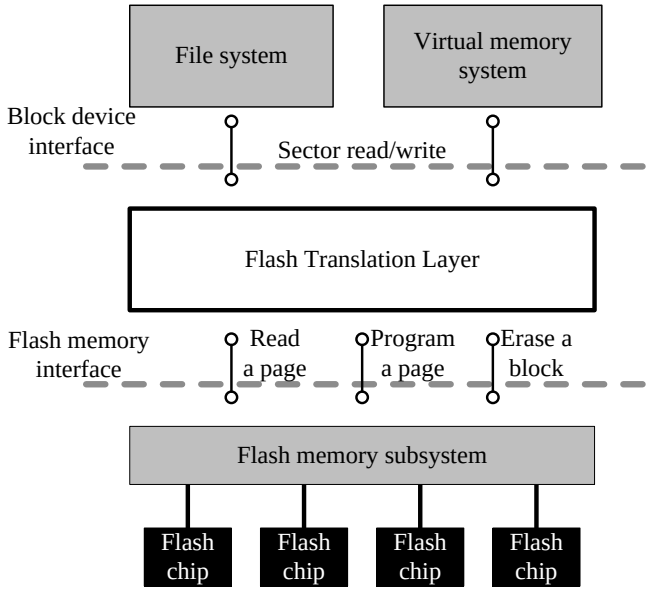


Figure 4: Flash translation layer hides the quirks of flash memory so that it can be used as storage.

FTL algorithms are needed to complement existing reliability enhancement techniques.

2.2 Flash translation layer

The FTL addresses flash memory's limitations so that it can be used as storage. As shown in Figure 4, the FTL hides the flash memory interface of erase, program, and read, and instead exports a traditional block device interface of sector reads and writes. Among the many required FTL tasks, mapping table management and garbage collection (GC) are the two prominent ones. Mapping table schemes are generally categorized by mapping granularities. The three main types of mapping table are page-level map-

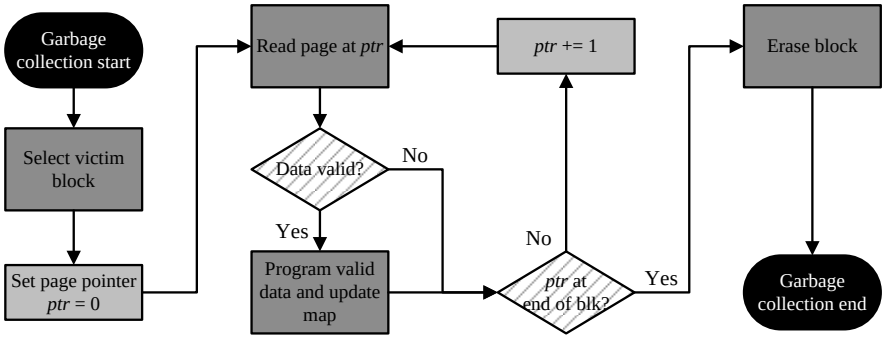


Figure 5: Generalize flowchart for garbage collection. The overall efficiency and parallelization of GC can be affected by the data structures maintained in the algorithm.

ping [20, 45, 66, 70], block-level mapping [11, 15, 57], and hybrid mapping [38, 41, 54, 69]. Page-level mapping translates host's logical address into flash memory's physical address at the page granularity, improving performance and flexibility at the cost of a large mapping table. Block-level mapping, on the other hand, does not require a large memory footprint but suffers from costly block-merge and read-modify-write operations [38]. The hybrid variant combines the benefits of the two by maintaining two types of blocks: data blocks managed by block-level mapping, and log blocks managed by page-level mapping.

Garbage collection reclaims space by selecting a victim block, copying out valid data in it, and erasing the block. Figure 5 illustrates a generalized flowchart for performing garbage collection. In addition to the victim block selection, the overall efficiency of GC can be affected by the data structures maintained in the algorithm. If the system does not maintain validity information in memory, then all the data within the block need to be read out

to determine their validities. On the other hand, if the validity bit vector is maintained for each block, only a subset of pages needs to be read out. Also, though not described in the flowchart, the read and program operations for each page can be parallelized and decoupled; multiple page reads and programs can be dispatched together, as long as the dependence between the read and the program for the same data is honored. Similarly, the block should not be erased prior to programming all the valid pages to another location.

The selection of a victim block is not only a matter of performance, but also that of lifetime as flash memory only allows a limited number of erases for blocks. Victim selection algorithms optimized for performance include greedy [9, 14, 23] and cost-benefit [33, 55], and algorithms optimized for lifetime include randomized [42] and data clustering [10].

Other FTL tasks include, but are not limited to wear-leveling [8, 68] that prolongs the overall lifetime of the flash storage, bad block management [35] that handles manufacture-time and run-time bad blocks, and read scrubbing [21] and retention-aware management [7] that proactively migrate data to prevent read errors. All these FTL tasks are needed to ensure the reliability of the storage, thus a careful management of FTL tasks is not only a matter of performance, but also a matter of correctness.

Chapter 3

Architecture of the autonomic SSD

In this chapter, we describe the overall architecture and the design of the autonomic SSD (AutoSSD) as shown in Figure 6. In our model, all FTL tasks run concurrently, with each designed and implemented specifically for its job. Each task independently interfaces the scheduling subsystem, and the scheduler arbitrates the resources in the flash memory subsystem according to the assigned share. The share controller monitors key system states and determines the appropriate share for each FTL task. AutoSSD is agnostic to the specifics of the FTL algorithms (i.e., mapping scheme and GC victim selection), and the following sections focus on the overall architecture and design of AutoSSD that enable the self-management of the flash storage.

3.1 Virtualization of the flash memory subsystem

The architecture of AutoSSD allows each task to be independent of others by virtualizing the flash memory subsystem. Each FTL task is given a pair of request and response queues to send and receive flash memory requests and responses, respectively. This interface provides an illusion of a dedicated (yet slower) flash memory subsystem and allows an FTL task to generate flash memory requests oblivious of others (whether idle or active)

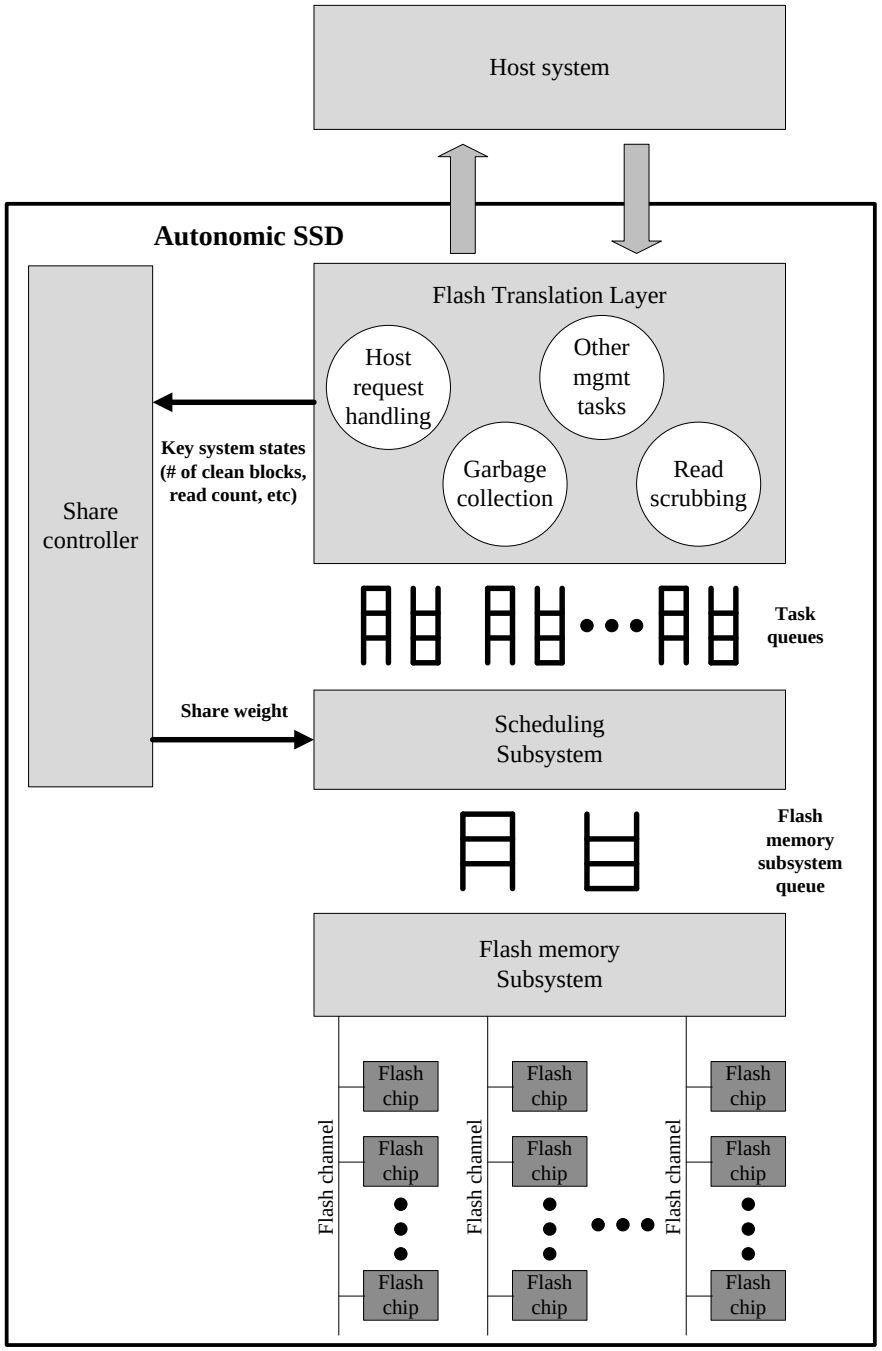


Figure 6: Overall architecture of AutoSSD and its components.

or the requests they generate (intensity or which resources they are using). Details of the flash memory subsystem are completely abstracted by the scheduling subsystem, and only the back-pressure of the queue limits each task from generating more flash memory requests.

This virtualization not only effectively frees each task from having to worry about others, but also makes it easy to add a new FTL task to address any future flash memory quirks. While background operations such as garbage collection, read scrubbing, and wear-leveling have similar flash memory workload patterns (reads and programs, and then erases), the objective of each task is distinctly different. Garbage collection reclaims space for host writes, read scrubbing preventively relocates data to ensure data integrity, and wear-leveling swaps contents of data to even out the wear of blocks. The design of AutoSSD allows seamless integration of new tasks without having to modify existing ones and re-optimize the system.

3.2 Scheduling mechanisms for share enforcement

The scheduling subsystem interfaces with each FTL task and arbitrates the resources of the flash memory subsystem. The scheduler should take advantage of the following characteristics of the flash storage:

- *High-throughput.* The flash memory scheduler manages concurrency (tens and hundreds of flash memory requests) and parallelism (tens and hundreds of flash memory chips) at a small timescale. This requires the scheduler to make efficient decisions on-the-fly with low

overhead.

- *Multi-client multi-server scheduling with constraints.* A flash memory request generated by an FTL task is targeted to a single flash memory chip. Thus, load balancing techniques used in other scheduling domains cannot be applied; instead, queue depth must be monitored and requests should be issued out-of-order to increase throughput and improve responsiveness.

- *Bounded operation time.* The specifications for flash memory operations are known and are bounded to complete within a given time. This allows the scheduler to share resources with bounded deviation from ideal fair sharing.

Considering these unique domain characteristics, we present scheduler designs of two different disciplines: a scheduler based on fair queueing [13, 16] from the network scheduling domain, and a scheduler we call *debit scheduler* based on request windowing technique [19, 25, 46] from the disk scheduling domain.

We then further enhance the schedulers by taking advantage of the operation suspension feature in state-of-the-art flash memory [65]. Operation suspension improves the overall responsiveness by allowing short operations such as reads to preempt ongoing long latency operations such as programs and erases. The change between the non-preemptive and preemptive scheduling is simple and straightforward, and preserves the overall scheduling principles and mechanisms for the fair queueing scheduler and the debit scheduler.

3.2.1 Fair queueing scheduler

The scheduler based on fair queueing (a.k.a. weighted fair queueing, WFQ) maintains a *virtual time* for each task as a measure of progress. Whenever requests from multiple tasks compete for a shared resource, WFQ selects the request whose task has made the least progress—the one with the smallest virtual time. The virtual time is incremented by the work value (normalized latency of operation) divided by the share of the task the selected request belongs. Thus, a task with a higher share will get its virtual time incremented slowly and be able to get its requests serviced more frequently than other tasks.

WFQ scheduling algorithm is formally defined as follows:

$$VST_A^N = \max(AT_A^N, VFT_A^{N-1}) \tag{3.1}$$

$$VFT_A^N = VST_A^N + WV/S_A \tag{3.2}$$

Where VST_A^N is the virtual start time for the N^{th} request belonging to task A, VFT_A^N is the virtual finish time, and AT_A^N is the request arrival time. WV is the work value of the request, and S_A is the share for task A. The work value WV depends on whether the virtual time is maintained globally for the entire flash memory subsystem, or locally for each resource of the flash memory subsystem. Under the global virtual time scheme, the work value WV of a request is normalized by dividing the latency of the operation by the number of resources in the system. This is done because only a single per-task virtual time value is maintained for the entire system, yet the work

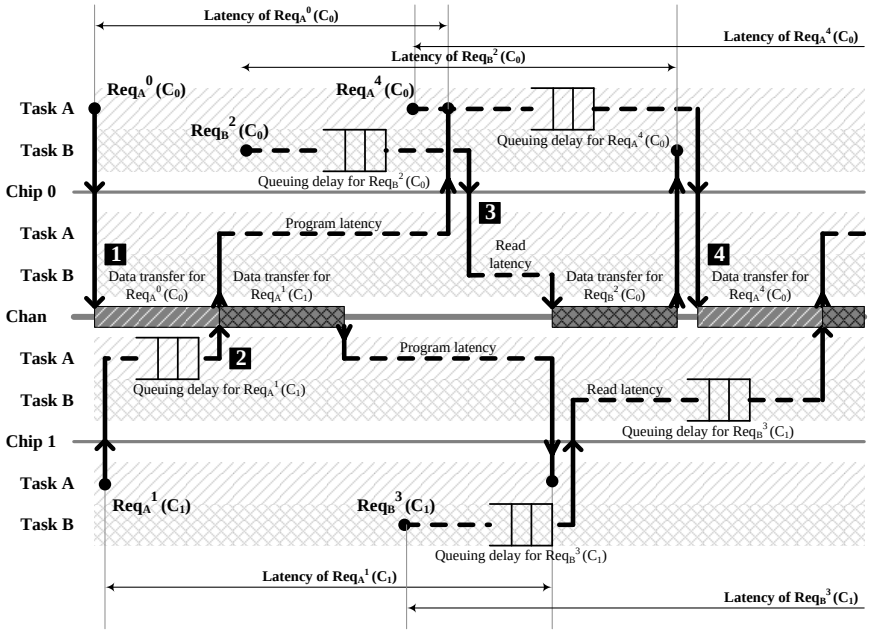


Figure 7: WFQ scheduling example given two tasks, two chips, and a single channel shared by the chips. Inbound arrows represent the acquisition of resources while outbound arrows represent the release. The command and address cycles are omitted as their channel bandwidth utilization is negligible, and they are implemented to interrupt any data transfers. $Req_T^i(C_n)$ denotes i^{th} request from task T for chip n.

for performing the flash memory operation needs to be normalized *as if* the all the resources of the subsystem is utilized to service that operation. In contrast, the work value WV is simply the latency of the operation under the local virtual time scheme. In this dissertation, WFQ maintains the virtual time globally as it achieves a higher degree of perceived fairness, and the local virtual time scheme requires more resources as it maintains per-task virtual time for each flash memory chip.

Figure 7 illustrates an example of the WFQ scheduling given two tasks, two chips, and a single channel shared by the chips. We assume that the two

tasks have the same share for this example. In the figure, $Req_T^i(C_n)$ denotes i^{th} request from task T for chip n, and important events are highlighted and numbered for further explanation.

1. A program request $Req_A^0(C_0)$ from task `A` arrives at chip `0`. As both the target chip and the channel are idle, the program request starts immediately.

2. Shortly after the event 1, $Req_A^1(C_1)$ arrives. It acquires chip `1` as it is idle, but is queued at the channel and waits until $Req_A^0(C_0)$ relinquishes the channel.

3. Both $Req_B^2(C_0)$ and $Req_A^4(C_0)$ have been queued for chip `0`, and when $Req_A^0(C_0)$ is finished, $Req_B^2(C_0)$ is scheduled. This is because task `A` has its virtual time incremented by performing $Req_A^0(C_0)$ and $Req_A^1(C_1)$, while the virtual time for task `B` is set to when $Req_B^2(C_0)$ is queued.

4. When $Req_B^2(C_0)$ is completed, $Req_A^4(C_0)$ is scheduled for the chip and the channel. While task `A` has a higher virtual time than task `B` due to executing more operations, $Req_B^3(C_1)$ has not finished its reading data from the flash array and thus is not yet ready to use the channel.

Note that in the illustration, the command and address cycles are omitted as their channel bandwidth utilization is negligible, and they are implemented to interrupt any data transfers.

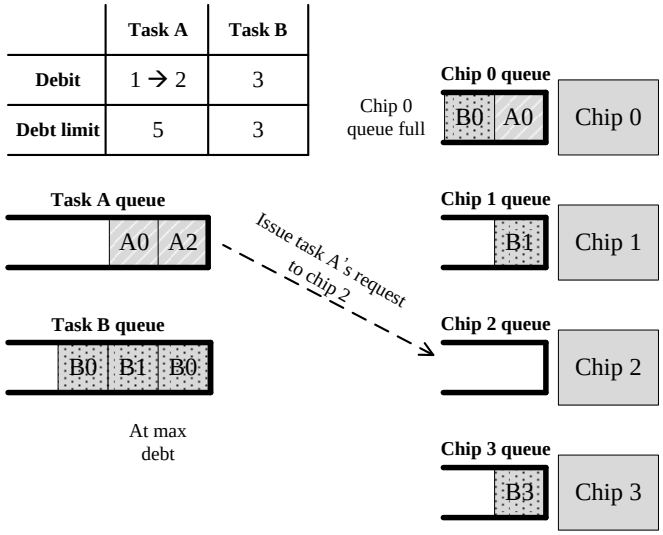
WFQ scheduling is widely used in the network domain and has been applied in the I/O scheduling domain for both disks [25] and SSDs [58]. While WFQ achieves a high-level of fairness and fully utilizes resources

(i.e., work-conserving), it incurs overhead for computing and maintaining the virtual time. In the next subsection, we introduce a simple scheduling scheme that arbitrates resources with less overhead and exhibits different scheduling characteristics.

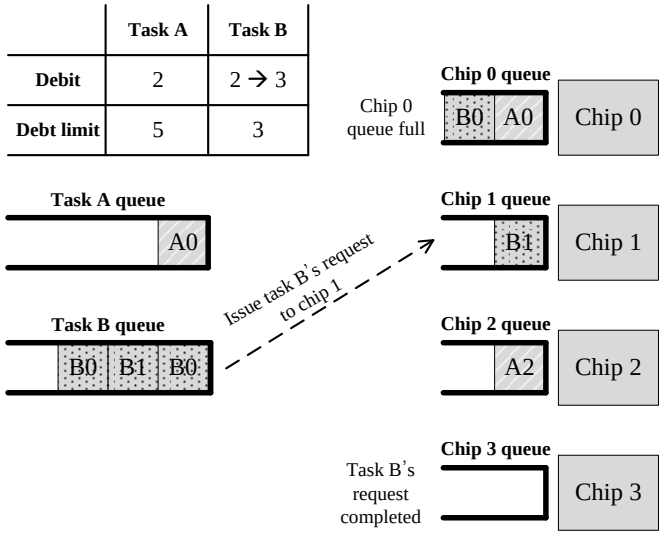
3.2.2 Debit scheduler

The debit scheduler achieves fair sharing by tracking and limiting the number of outstanding requests per task across all resources. If the number of outstanding requests for a task, which we call debit, is under the limit, its requests are eligible to be issued; if it's not, the request cannot be issued until one or more of requests from the task completes. The debt limit is proportional to the share set by the share controller, allowing a task with a higher share to potentially utilize more resources simultaneously. The sum of all tasks' debt limit represents the total amount of parallelism, and is set to the total number of requests that can be queued in the flash memory controller.

Figure 8 illustrates two scenarios of the debit scheduling. In both scenarios, the debt limit is set to 5 requests for task `A`, and 3 for task `B`. In Figure 8a, no more requests can be sent to chip `0` as its queue is full, and task `B`'s requests cannot be scheduled as it is at its debt limit. Under this circumstance, task `A`'s request to chip `2` is scheduled, increasing its debit value from 1 to 2. In Figure 8b, the active operation at chip `3` for task `B` completes, allowing task `B`'s request to be scheduled. Though task `B`'s request to chip `1` is not at the head of the queue, it is scheduled out-of-order as there is no dependence between the requests to chip `0` and chip `1`. Task `A`, although below



(a) Task `A`'s request issued to chip `2`.



(b) Task `B`'s request issued to chip `0`.

Figure 8: Two debit scheduling examples. In the scenario of Figure 8a, no more requests can be sent to chip `0`, and task `B` is at its maximum debit. The only eligible scheduling is issuing task `A`'s request to chip `2`. In the scenario of Figure 8b, while task `A` is still under the debt limit, its request cannot be issued to a chip with a full queue. On the other hand, a request from task `B` can be issued as chip `3`'s operation for task `B` completes.

the debt limit, cannot have its requests issued until chip `0` finishes a queued operation, or until a new request to another chip arrives. Though not illustrated in these scenarios, when multiple tasks under the limit compete for the same resource, one is chosen with skewed randomness favoring a task with a smaller debit to debt limit ratio. Randomness is added to probabilistically avoid starvation.

Debit scheduling only tracks the number of outstanding requests, yet exhibits interesting properties. First, it can make scheduling decisions without complex computations and bookkeeping. This allows the debit scheduler to scale with increasing number of resources. Second, although it does not explicitly track *time*, it implicitly favors short latency operations as they have a faster turn-around-time. In scheduling disciplines such as weighted round robin [31] and weighted fair queueing [13], the latency of operations must be known or estimated to achieve some degree of fairness. Debit scheduling, however, approximates fairness in the time-domain only by tracking the number of outstanding requests. Lastly, the scheduler is in fact not work-conserving. The total debt limit can be scaled up to approximate a work-conserving scheduler, but the share-based resource reservation of the debit scheduler allows high responsiveness, as observed in the resource reservation protocol for Ozone [48].

3.2.3 Preemptive schedulers

State-of-the-art flash memory allows short operations such as reads to preempt long latency operations such as programs and erases [65]. This feature, when used properly, can improve overall responsiveness of the system

as reads experience shorter delays. However, the unconstrained use of preemptions can be detrimental under the following scenarios:

- *Reduced efficiency.* Operation suspension incurs overhead as the internal voltage of flash memory needs to be reset. This overhead, in the order of μs, reduces the overall efficiency.

- *Starvation.* The suspended operation can starve and not make progress if it is repeatedly preempted. A judicious scheduler should prevent indefinite suspension of operations.

- *Dependence violation.* Operations that preempt ongoing operations should not violate data dependence. For example, reads should not suspend a program operation that is writing the data to be read. A simple and conservative solution is to only allow inter-task preemptions, prohibiting its use when ongoing operation belongs to the same task.

From the non-preemptive versions described in the previous subsections, the fair queueing scheduler and debit scheduler are enhanced in the following manner to take advantage of the operation suspension feature.

The preemptive fair queueing scheduler only allows preemption if the request to be issued belongs to a task with the least virtual time. Otherwise, requests are issued and queued normally, and virtual time management is identical to the non-preemptive counterpart.

For the preemptive debit scheduler, the number of outstanding preempting requests is tracked. In this version, non-preemptive requests do not

count toward the debit mechanism. This makes the debit scheduler fully work-conserving as requests are not throttled based on utilization. Instead, the share for each task reflects the maximum number of allowed preemptions: task with a high share is allowed more preemptions, while that with a low share are allowed less. Similar to the mechanism to resolve race condition in the non-preemptive version, when multiple requests are available for scheduling, the debit to debt ratio with a random skew is used.

3.3 Scheduling policy based on feedback control

The share controller determines the appropriate share for the scheduling subsystem by observing key system states. States such as the number of free blocks and the maximum read count reflect the stability of the flash storage. This is critical for the overall performance and reliability of the system, as failure to keep these states at a stable level can lead to an unbounded increase in response time or even data loss.

For example, if the flash storage runs out of free blocks, not only do host writes block, but also all other tasks that use flash memory programs stall: activities such as making mapping data durable and writing periodic checkpoints also depend on the garbage collection to reclaim free space. Even worse, a poorly constructed FTL may become deadlocked if GC is unable to obtain a free block to write the valid data from its victim. On the other hand, if a read count for a block exceeds its recommended limit, accumulated read disturbances can lead to data loss if the number of errors

is beyond the error correction capabilities. In order to prevent falling into these adverse system conditions, the share controller increases or decreases the progress rates for individual FTL tasks by monitoring key system states and adjusting the tasks' shares. Thus, it is the responsibility of the share controller to monitor these system states and adjust shares to increase or decrease the rate of progress for individual FTL tasks.

In this dissertation, we present two share control functions based on feedback control. First, we describe the proportional control scheme used in QoSFC [34], and then present the proportional-integral control scheme that allows the share of background operations to decay over time in anticipation of host requests.

3.3.1 Proportional control

In QoSFC [34], the share for the garbage collection is inversely proportional to the number of free blocks. In this implementation, the GC share is aggressively increased as the number of free blocks drops low, and it improved QoS across synthetic workloads with regular inter-arrival times. However, under real workloads that exhibit periods of idleness in host requests, this share control strategy unnecessarily penalizes host request handling if garbage collection could opportunistically reclaim space. The following control function describes the behavior of control in QoSFC:

$$S_A[t] = P_A \cdot e_A[t] \tag{3.3}$$

Where $S_A[t]$ is the share for task A at time t, P_A is a non-negative coef-

ficient for task A, and $e_A[t]$ is the error value for task A at time t. The error value function is different for each task, and for the GC task it is defined as follows:

$$e_{GC}[t] = \max(0, target_{freeblk} - num_{freeblk}[t]) \tag{3.4}$$

In this scheme, $target_{freeblk}$ is set to the GC activation threshold, and the share for GC S_{GC} starts out small and increases as the error value increases. However, once the number of free blocks $num_{freeblk}[t]$ exceeds $target_{freeblk}$, the error value is set to zero, effectively deactivating the GC task.

3.3.2 Proportional-integral control

AutoSSD uses feedback to adaptively determine the shares for the internal FTL tasks. While the values of key system states must be maintained at an adequate level, the shares of internal FTL tasks must not be set too high such that they severely degrade host performance. Once a task becomes active, it initially is allocated a small share. If this fails to maintain the current level of the system state, the share is gradually increased to counteract the momentum. The following control function is used to achieve this behavior:

$$S_A[t] = P_A \cdot e_A[t] + I_A \cdot S_A[t-1] \tag{3.5}$$

Where $S_A[t]$ is the share for task A at time t, $S_A[t-1]$ is the previous share for task A, P_A and I_A are two non-negative coefficients for task A, and $e_A[t]$ is the error value for task A at time t. The error value function for GC

is identical to that described for the proportional control:

$$e_{GC}[t] = \max(0, target_{freeblk} - num_{freeblk}[t]) \tag{3.6}$$

With $target_{freeblk}$ set to the GC activation threshold, the share for GC S_{GC} starts out small. If the number of free blocks $num_{freeblk}[t]$ falls far below $target_{freeblk}$, the error function $e_{GC}[t]$ augmented by P_{GC} ramps up the GC share S_{GC}. After the number of free blocks $num_{freeblk}[t]$ exceeds the threshold $target_{freeblk}$, the share S_{GC} slowly decays given $I_{GC} < 1$.

Addition to the GC share control, the error value function for read scrubbing (RS) is defined as follows:

$$e_{RS}[t] = \max(0, \max_{i \in blk}(readcnt_i[t]) - target_{readcnt}) \tag{3.7}$$

Where $\max_{i \in blk}(readcnt_i[t])$ is the maximum read count across all blocks in the system at time t, and $target_{readcnt}$ is the RS activation threshold.

In AutoSSD, the share for internal management schemes starts out small, anticipating host request arrivals and using the minimum amount of resources to perform its task. If the system state does not recover, the error (the difference between the desired and the actual system state values) accumulates, increasing the share over time.

Chapter 4

Evaluation methodology

We model a flash storage system on top of the DiskSim environment [5] by enhancing its SSD extension [1]. In this chapter, we describe the various components and configuration of the SSD, and the workload and test settings used for evaluation.

4.1 Flash memory subsystem

Flash memory controller is based on Ozone [48] that fully utilizes flash memory subsystem's channel and chip parallelism. There can be at most two requests queued for each chip in the controller. Increasing the queue depth does not significantly increase intra-chip parallelism, as cached operations of flash memory have diminishing benefits as the channel bandwidth increases. Instead, a smaller queue depth is chosen to increase the responsiveness of the system.

Table 1 summarizes the default flash storage configuration used in our experiments. Of the 256GB of physical space, 200GB is addressable by the host system, giving an over-provisioning factor of 28%.

Table 1: System configuration.

Parameter	Value	Parameter	Value
# of channels	4	Read latency	50μs
# of chips/channel	4	Program latency	500μs
# of planes/chip	2	Erase latency	5ms
# of blocks/plane	1024	Data transfer rate	400MB/s
# of pages/block	512	Physical capacity	256GB
Page size	16KB	Logical capacity	200GB

4.2 Scheduling subsystem

The scheduling subsystem arbitrates the resources of the flash memory subsystem according to the allotted shares. Each task can dispatch up to 64 requests, and requests for each flash memory chip can be issued out-of-order by the scheduler to improve overall resource utilization. Scheduling decisions are made upon an arrival of a request to an idle chip, or upon a not-full condition of a queue for a resource. Whenever there are multiple available resources, the one with the least number of requests queued is considered prior to considering those with more requests queued. This is so that the scheduler issues requests that are likely to execute earlier than others.

In addition to the weighted fair queueing schedulers and debit schedulers described in the previous chapter, a FIFO scheduler that issues requests for each flash memory chip in the order of arrival is also implemented as a baseline scheduler. For both the non-preemptive and preemptive WFQ schedulers, the arrival time is tagged for each request when queued, and a virtual time is maintained per task globally. For the non-preemptive debit

Table 2: Share controller coefficients.

Task	P coeff	I coeff
GC	0.01	0.99
RS	0.0001	0.99

scheduler, the number of outstanding requests per task is tracked, incremented upon issuing a request to, and decrement upon receiving a response from the flash memory subsystem. For the preemptive debit scheduler, the number of outstanding preemptive requests per task is tracked, incremented upon issuing a preemptive request, and decrementing upon receiving a response to a preemptive request. The maximum number of allowed outstanding request per task (debt limit) is proportional to the share, and the sum of all the debt limit is set to the total number of requests that can be queued at the flash memory controller.

4.3 Share controller

The share controller determines the appropriate share for each task depending on the state of the system. We implement the proportional-integral control using two states: number of free blocks and maximum read count. These two states are used to determine the shares for GC and RS tasks, respectively, and the rest of the share is allocated for the host handler task. The default coefficient values for the control functions are shown in Table 2. These coefficients are not theoretically optimized, but we found empirically that the coefficient values allocate enough shares for the two FTL tasks, and the shares decay fast enough to allow high responsiveness for host requests.

The intuition for setting the `P` coefficient is as follows. For every 100 free blocks under the desired number of free blocks, the GC share is incremented by 1 under the settings in Table 2. With shares computed every 10ms, this means that GC share becomes fully saturated if free blocks do not recover within 1 second. Similar intuition is used for the `P` coefficient for the RS task. On the other hand, the intuition for the `I` coefficient is that, as shares are computed every 10ms, the integral coefficient of 0.99 means that the share decays to 36% every 1 second. As a starting point, we consider this to be an appropriate decay rate.

To prevent divide-by-zero error for the WFQ scheduler (as it divides the work value by the share set by the controller), the minimum value for a share is set to 1%.

4.4 Flash translation layer

We implement core FTL tasks and features that are essential for storage functions, yet cause performance variations. Garbage collection reclaims space, but it degrades the performance of the system under host random writes. Read scrubbing that preventively relocates data creates background traffic on read-dominant workloads. Mapping table lookup is necessary to locate host data, but it increases response time on map cache misses.

4.4.1 Mapping

We implement an FTL with map caching [20] and a mapping granularity of 4KB. The entire mapping table is backed in flash, and mapping data,

also maintained at the 4KB granularity, is selectively read into memory and written out to flash during runtime. The LRU policy is used to evict mapping data, and if the victim contains any dirty mapping entries, the 4KB mapping data is written to flash. By default, we use 128MB of memory to cache the mapping table. The second-level mapping that tracks the locations of the 4KB mapping data is always kept in memory as it is accessed more frequently and orders of magnitude smaller than the first-level mapping table.

4.4.2 Host request handling

Upon receiving a request, the host request handler looks up the second-level mapping to locate the mapping data that translates the host logical address to the flash memory physical address. If the mapping data is present in memory (hit), the host request handler references the mapping data and generates flash memory requests to service the host request. If it is a miss, a flash memory read request to fetch the mapping data is generated, and the host request waits until the mapping data is fetched. Host requests are processed in a non-blocking manner; if a request is waiting for the mapping data, other requests may be serviced out-of-order. In our model, if the host write request is smaller than the physical flash memory page size, multiple host writes are aggregated to fill the page to improve storage space utilization. We also take into consideration of the mapping table access overhead and processing delays. Mapping table lookup delay is set to be uniformly distributed between 0.5μs and 1μs, and the flash memory request generation delay for the host task is between 1μs and 2μs.

4.4.3 Garbage collection

The garbage collection (GC) task runs concurrently and independently from the host request handler and generates its own flash memory requests. Victim blocks are selected based on cost-benefit [55]. Once a victim block is selected, valid pages are read and programmed to a new location. Mapping data is updated as valid data is copied, and this process may generate additional requests (both reads and programs) for mapping management. Once all the valid pages have been successfully copied, the old block is erased and marked free. GC becomes active when the number of free blocks drops below a threshold, and stops once the number of free blocks exceeds another threshold, similar to the segment cleaning policy used for the log-structured file system [55]. In our experiments, the two threshold values for GC activation and deactivation are set to 128 and 256 free blocks, respectively. The garbage collection task also has a request generation delay, set to be uniformly distributed between 1μs and 3μs.

4.4.4 Read scrubbing

The read scrubbing (RS) task also runs as its own stand-alone task. Victims are selected greedily based on the read count of a block: the block with the most number of reads is chosen. Other than that, the process of copying valid data is identical to that of the garbage collection task. RS becomes active when the maximum read count of the system goes beyond a threshold, and stops once it falls below another threshold. The default threshold values for the activation and deactivation are set to 100,000 and 80,000 reads,

respectively. Like the garbage collection task, the request generation delay (modeling the processing overhead of read scrubbing) is uniformly distributed between 1μs and 3μs.

4.5 Workload and test settings

We use both synthetic workloads and real-world I/O traces from Microsoft production servers [32] to evaluate the autonomic SSD architecture. Synthetic workloads are used to verify that our model behaves expectedly according to the system parameters. Performance drop under 4KB random writes for a baseline system that schedules in FIFO order was shown in Figure 1, and we additionally show performance under 128KB sequential accesses, 4KB random reads, and 4KB random reads/writes.

From the original traces, the logical address of each host request is modified to fit into the 200GB range, and all the accesses are aligned to 4KB boundaries. All the traces are run for their full duration, with some lasting up to 24 hours and replaying up to 44 million I/Os. The trace workload characteristics are summarized in Table 3.

Prior to replaying the trace, the entire physical space is randomly written to emulate a pre-conditioned state so that the storage would fall under the steady state performance described in SNIA's SSS-PTS [53]. Furthermore, each block's read count is initialized with a non-negative random value less than the read scrubbing threshold to emulate past read activities.

Table 3: Trace workload characteristics.

Workload	Duration (hrs)	Number of I/Os (M)		Avg. request size (KB)		Inter-arrival time (ms)	
		Write	Read	Write	Read	Avg.	Median
DAP-DS	23.5	0.1	1.3	7.2	31.5	56.9	31.6
DAP-PS	23.5	0.5	0.6	96.7	62.1	79.9	1.7
DTRS	23.0	5.8	12.0	31.9	21.8	4.6	1.5
LM-TBE	23.0	9.2	34.7	61.9	53.2	1.9	0.8
MSN-CFS	5.9	1.1	3.2	12.9	8.9	4.9	2.0
MSN-BEFS	5.9	9.2	18.9	11.6	10.7	0.8	0.3
RAD-AS	15.3	2.0	0.2	9.9	11.0	24.9	0.8
RAD-BE	17.0	4.3	1.0	13.0	106.2	11.7	2.6
WBS	23.7	5.9	6.0	96.7	26.3	7.2	0.7

Chapter 5

Experiment results

This chapter presents experimental results under the configuration and workload settings described in the previous chapter. The main performance metric we report is the system response time seen at the I/O device driver. As the focus of this dissertation is response time characteristics, we only measure the performance of QoS-sensitive small reads that are no larger than 64KB.

We first validate our SSD model using synthetic workloads, and then present experimental results with I/O traces. We replayed the I/O traces in three different experiments: (1) with the original request dispatch times, (2) with the original dispatch times scaled up and down to change the workload intensity, and (3) with collocation that emulates two VMs sharing the same flash storage. Finally, we present sensitivity analysis results for scheduler and controller parameters, as well as the read scrubbing thresholds.

5.1 Micro-benchmark results

Figure 9 and Figure 10 illustrate the performance of the autonomic SSD architecture (AutoSSD) with debit scheduling under four micro-benchmarks. Figure 9 plots the total bandwidth under 128KB sequential reads and 128KB sequential writes as we increase the channel bandwidth. As the channel

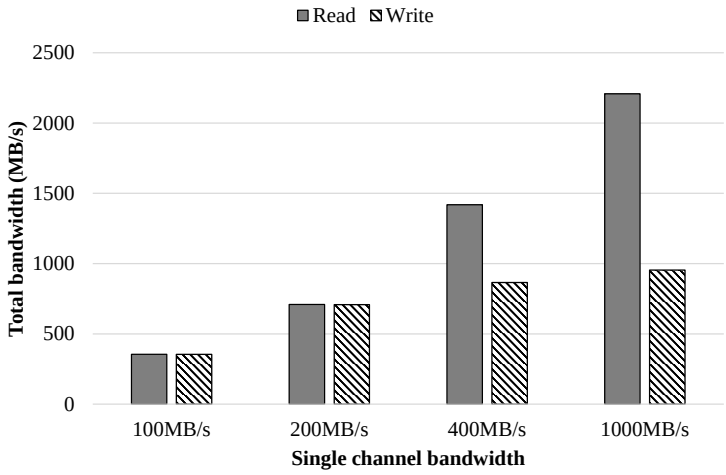
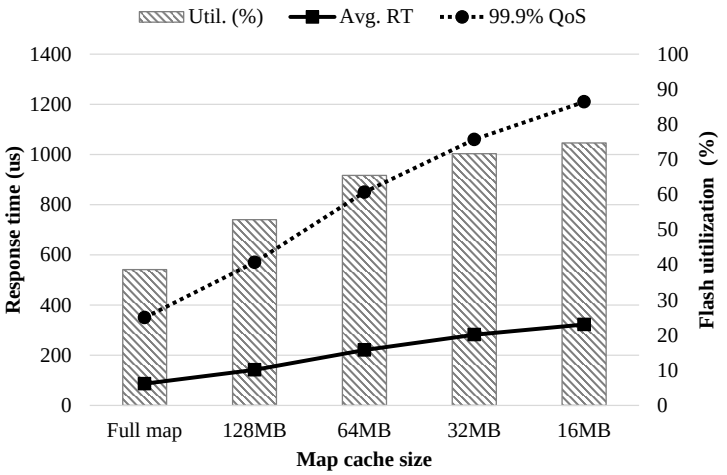


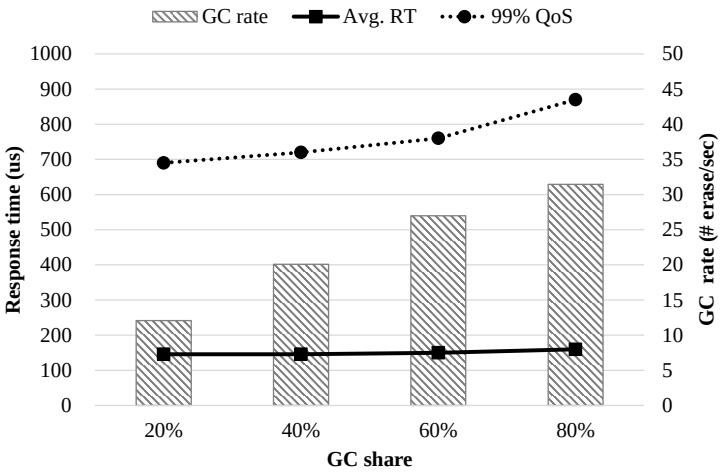
Figure 9: The total bandwidth under 128KB sequential accesses with respect to changes in the channel bandwidth.

bandwidth increases, the flash memory operation latency becomes the performance bottleneck. Write performance saturates early as the program latency cannot be hidden with data transfers. At 1GB/s channel bandwidth, the read operation latency also becomes the bottleneck, unable to further extract bandwidth from the configured four channels. Traffic from GC and mapping management has a negligible effect for large sequential accesses, and RS task was disabled for this run to measure maximum raw bandwidth.

In Figure 10a, we vary the in-memory map cache size and measure the response times of 4KB random read requests when issued at 100K I/Os per second (IOPS). As expected, the response time is smallest when the entire map is in memory, as it does not generate map read requests once the cache is filled after cold-start misses. However, as the map cache becomes smaller, the response time for host reads increases not only because it prob-



(a) 4KB random reads.



(b) 4KB random reads/writes.

Figure 10: Performance under synthetic workloads. Figure 10a shows the performance (average response time and 99.9% QoS) and the utilization of the flash memory subsystem with respect to changes in the size of the in-memory map cache. Figure 10b shows the performance (average response time and 99% QoS) and the GC progress rate with respect to the GC share.

abilistically stalls waiting for map reads from flash memory, but also due to increased flash memory traffic, which causes larger queueing delays.

Lastly, we demonstrate that the debit scheduling mechanism exerts control over FTL tasks in Figure 10b. In this scenario, 4KB random read/write requests are issued at 10K IOPS with 1-to-9 read/write ratio. Both the response time of host read requests and GC task's progress (in terms of the number of erases per second while active) are measured as we increase the share of the GC task. When the share is small, the GC task's progress is limited. As the share for GC increases, GC progress rate increases, but this negatively affects host performance, as evident by the increase in 99% QoS. The average response time is marginally affected due to the relatively low IOPS of this experiment, but the increase in 99% QoS exemplifies the severity of a long-tail problem in SSDs.

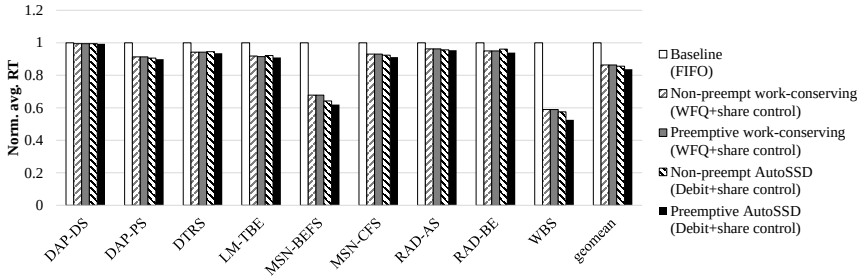
It is important to note that the progress rate for a task depends not only on the share, but also on the workload, algorithm, and system state. For example, the number of valid data in the victim block, the location of the mapping data associated with the valid data, and the access patterns at the flash memory subsystem all affect the rate of progress for GC. A task's progress rate is, in fact, nonlinear to the share under real-world workloads, and computationally solving for the optimal share involves large overhead, if not impossible.

5.2 I/O trace results

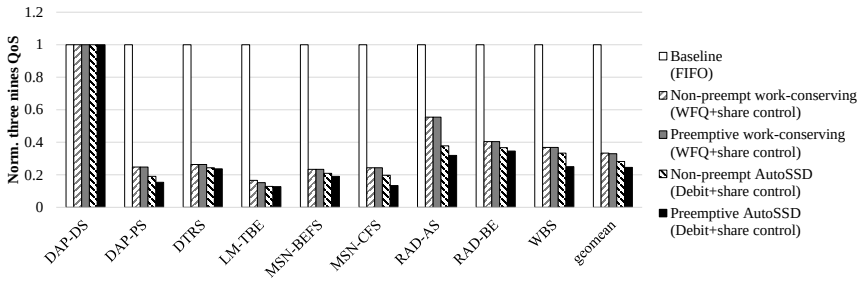
We compare the performance of AutoSSD against that of a baseline system that schedules flash memory requests in the FIFO order, and that of an idealized work-conserving system (WC) that schedules using weighted fair queueing (WFQ). The baseline system represents a design without virtualization and coordination, and all tasks dispatch requests to the controller through a single pair of request/response queues. The WC is used to compare the effectiveness of debit scheduling that controls the number of outstanding requests per-task. For both WC and AutoSSD, the shares dynamically change over time according to the feedback control function defined in Chapter 3.3. Furthermore, both non-preemptive and preemptive versions of scheduling are used for both WC and AutoSSD.

Figure 11 compares the performance of the five systems for nine different traces. As shown in Figure 11a, preemptive AutoSSD reduces average response time by 16.2% on average (at most 47.4% under `WBS`). For the six nines (99.9999%) QoS shown in Figure 11c, preemptive AutoSSD shows a much greater improvement, reducing it by 67.8% on average (at most 82.9% under `MSN-BEFS`). For the non-preemptive AutoSSD, the average response time is reduced by 14.4% on average (at most 42.5% under `WBS`), and the six nines (99.9999%) QoS is reduced by 67.3% on average (at most 84.7% under `MSN-BEFS`).

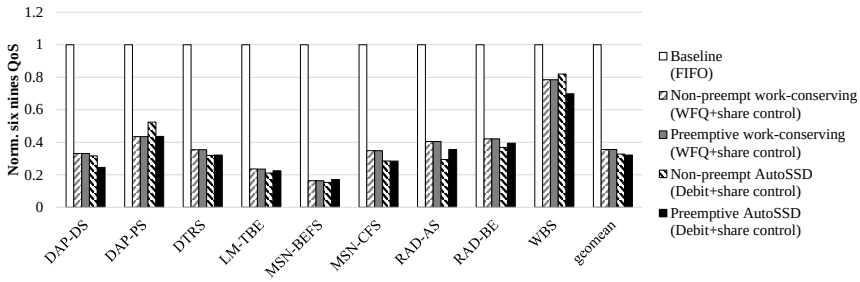
The performance difference between the preemptive and non-preemptive AutoSSD is at its greatest for `WBS`, and Figure 12 illustrates this. It shows the maximum response time in a 100ms window for the two systems about



(a) Average response time.



(b) Three nines QoS.



(c) Six nines QoS.

Figure 11: Comparison of the baseline, work-conserving, and AutoSSD for nine different traces. Results are normalized to the performance of the baseline. Preemptive AutoSSD reduces the average response time by 16.2% on average (at most 47.4% under `WBS`), reduces the three nines QoS by 75.4% on average (at most 87.3% under `LM-TBE`), and reduces the six nines QoS by 67.8% on average (at most 82.9% under `MSN-BEFS`).

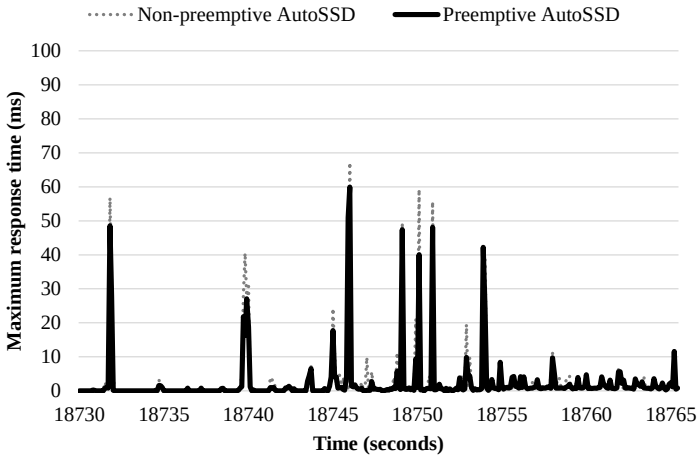
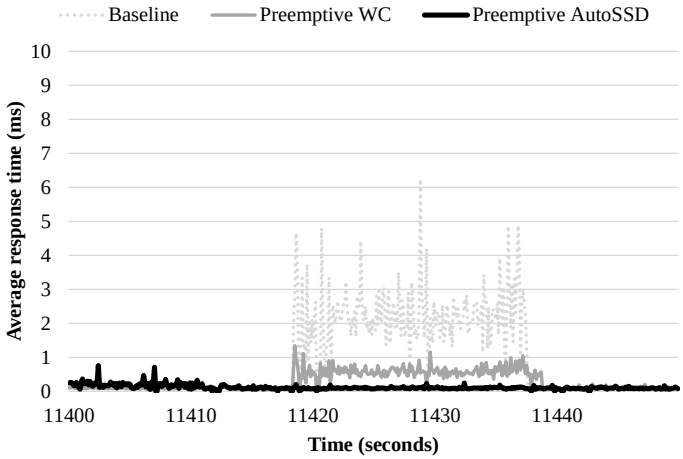


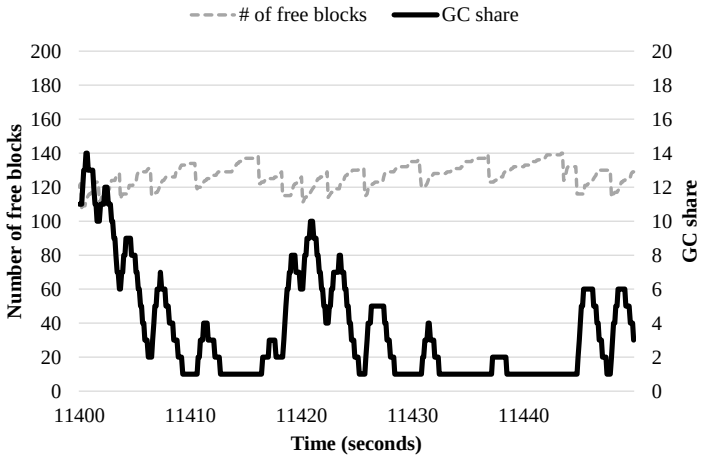
Figure 12: Preemptive vs. non-preemptive AutoSSD under WBS.

5 hours into WBS. In the figure, the preemptive version shows smaller maximum response times than its non-preemptive counterpart during the observed period. Preemptive scheduling allows the system to process read operations faster than the non-preemptive version by suspending ongoing long latency operations and servicing reads. This scheme is particularly useful when garbage collection is fully utilizing resources when host requests arrive. As the overall performance benefits of scheduling preemptively are greater than those of the non-preemptive version, further experiment results presented are based on the preemptive scheduling.

To understand the performance improvements over the baseline, we examine WBS microscopically in Figure 13 and LM-TBE in Figure 14. Figure 13a shows the average response time of the three systems, approximately 3 hours into WBS. GC is active during this observed period for all the three systems, and the baseline system with FIFO scheduling exhibits large spikes

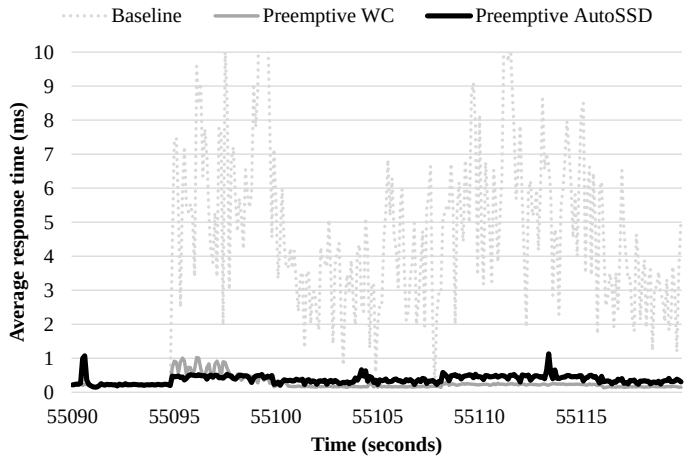


(a) Average response time under `WBS`.

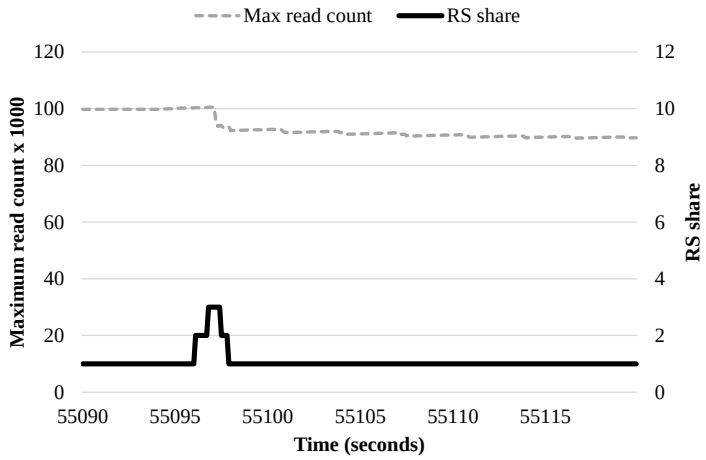


(b) Free blocks and GC share under `WBS`.

Figure 13: Comparison of the baseline, work-conserving, and AutoSSD under `WBS`. Figure 13a shows the average response time sampled at 100ms. Figure 13b shows the number of free blocks and the GC share of AutoSSD.



(a) Average response time under LM−TBE.



(b) Read count and RS share under LM−TBE.

Figure 14: Comparison of the baseline, work-conserving, and AutoSSD under LM−TBE. Figure 14a shows the average response time sampled at 100ms. Figure 14b shows the maximum read count and the RS share of AutoSSD.

in response time that reaches up to 6ms. WC and AutoSSD are better able to bound the performance degradation caused by an active garbage collection. Figure 13b shows the number of free blocks and the GC share for AutoSSD. The sawtooth behavior for the number of free blocks is due to host requests consuming blocks, and GC gradually reclaiming space. GC share is reactively increased when the number of free blocks becomes low, thereby increasing the rate at which GC produces free blocks. If the number of free blocks exceeds the GC activation threshold, the share decays gradually to allow other tasks to use more resources.

For `LM-TBE`, Figure 14a shows the average response time of the three systems, approximately 15 hours into the workload. Here we observe read scrubbing (RS) becoming active due to the read-dominant characteristics of `LM-TBE`. The baseline system with FIFO scheduling shows a large spike in response time that reaches up to 12ms. While both GC and RS generate similar flash memory request patterns, GC is incentivized to select a block with less valid data, while RS is likely to pick a block with a lot of valid data that are frequently read but rarely updated. WC and AutoSSD limit the performance degradation induced by active read scrubbing by distributing the work over a longer period, showing the benefits of the proposed feedback control. Figure 14b shows the maximum read count in the system and the share of RS for AutoSSD. The RS share is increased initially to reduce the maximum read count, and once it reaches a stable region, the RS share is maintained at the lowest value.

Next, we examine the effectiveness of the dynamic share assignment over the static ones. Figure 15 shows the response time CDF of AutoSSD

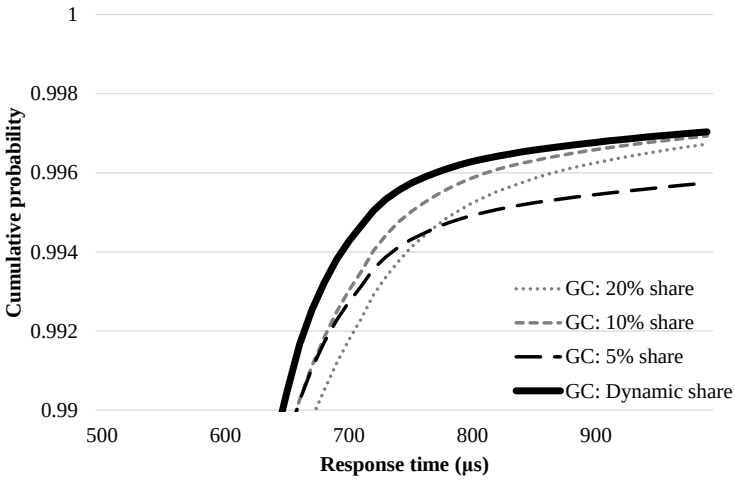


Figure 15: Comparison of AutoSSD with static shares and dynamic share under `WBS`.

under `WBS` with static shares of 5%, 10%, and 20% for GC, along with the share controlled dynamically. As illustrated by the gray lines, decreasing the GC share from 20% to 10% improves the overall performance. However, when further reducing the GC share to 5%, we can observe a cross-over between the curves for the 5% and the 20%. This indicates that while a lower GC share achieves better performance at lower QoS levels, a higher GC share is desirable to reduce long-tail latencies as it generates free blocks at a higher rate, preventing the number of free blocks from becoming critically low. Using feedback control to adjust the GC share dynamically shows better performance over all the static values, as it can adapt to an appropriate share value by reacting to the changes in the system state.

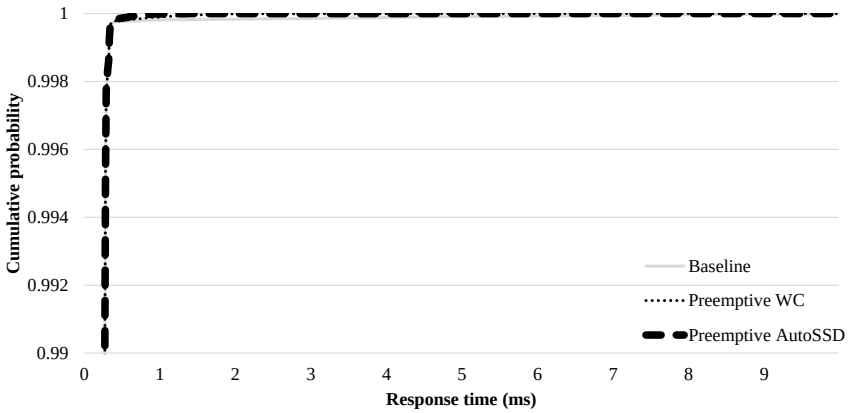
Figure 16, Figure 17 and Figure 18 shows the response time CDF for the nine I/O trace workloads for the baseline, preemptive WC, and preemp-

tive AutoSSD system. In general, workloads with high intensity such as MSN-BEFS and WBS show a greater gap between the curves for the baseline and AutoSSD, while workloads with lower intensity (DAP-DS as an extreme example) show a smaller gap. Because of this, the relative benefit of AutoSSD over WC seems smaller in Figure 11 for high-intensity workloads because the baseline system performs much worse compared to its performance in low-intensity workloads. However, CDF graphs such as Figure 17a for LM-TBE, Figure 17b for MSN-BEFS, and Figure 18c for WBS reveal the performance benefit of AutoSSD over the work-conserving system.

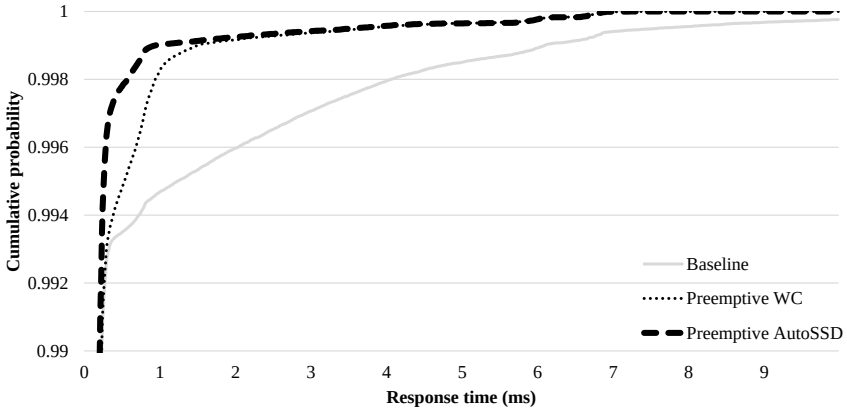
5.3 I/O trace results with scaled intensity

In this section, we present experimental results with higher and lower workload intensities. For higher workload intensity, the request dispatch times are reduced in half from the original, but otherwise the access types and the target addresses remain unchanged. Similarly, the workload intensity is lowered by doubling the original request dispatch times. This experiment is intended to examine the performance of the three systems—baseline, work-conserving (WC), and AutoSSD—under more stressful and relaxed scenarios, respectively. For the WC and AutoSSD system, the schedulers are preemptive, utilizing the operation suspension feature of the flash memory.

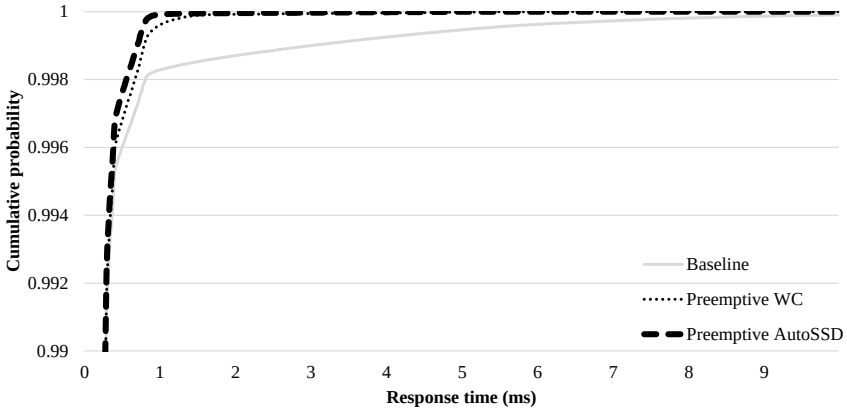
Figure 19 compares the performance under higher workload intensity. On average, AutoSSD reduces the average response time by 40.8% as shown in Figure 19a. This is more than double the improvement over the results



(a) Response time CDF under DAP−DS.

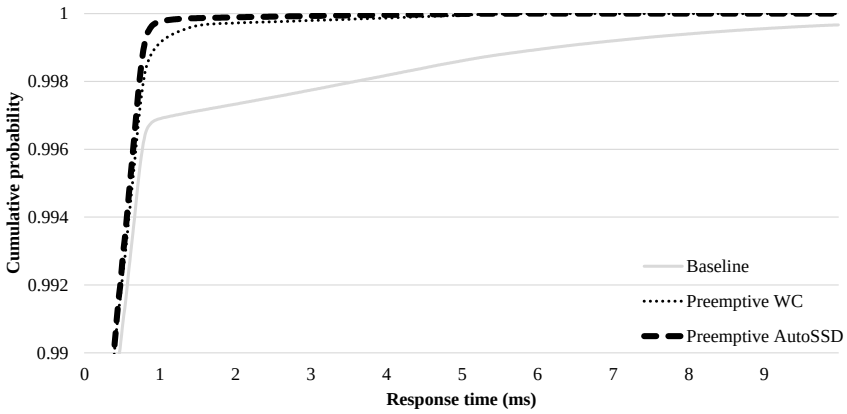


(b) Response time CDF under DAP−PS.

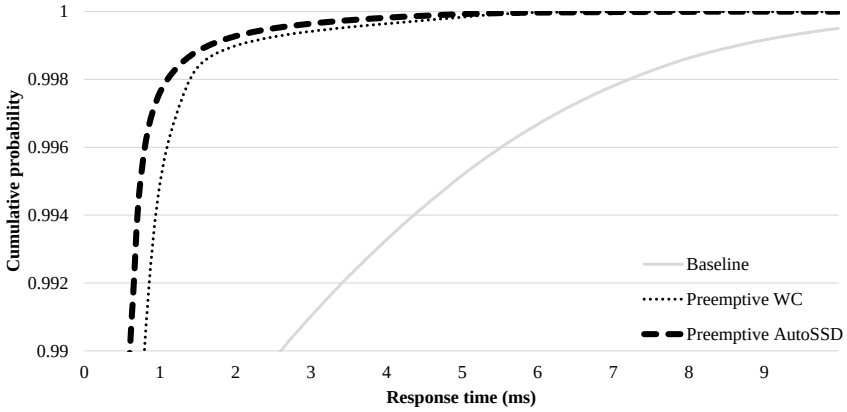


(c) Response time CDF under DTRS.

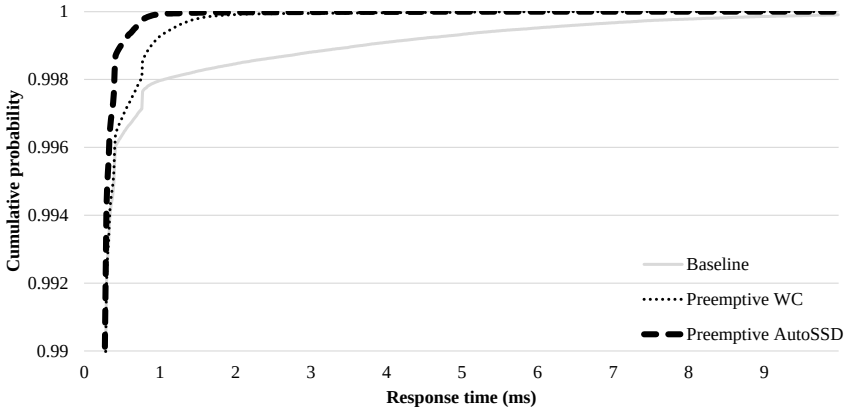
Figure 16: Response time CDF under DAP−DS, DAP−PS, and DTRS.



(a) Response time CDF under LM−TBE.

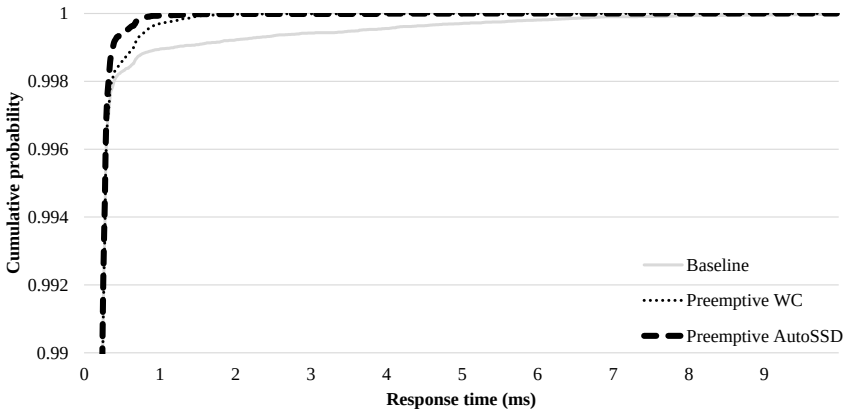


(b) Response time CDF under MSN−BEFS.

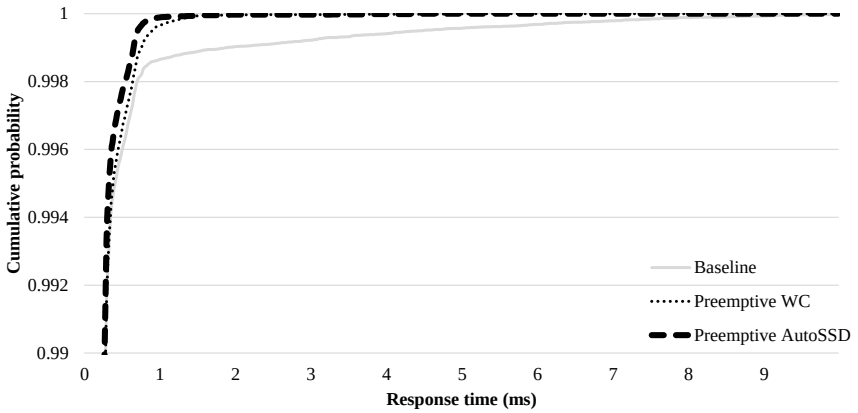


(c) Response time CDF under MSN−CFS.

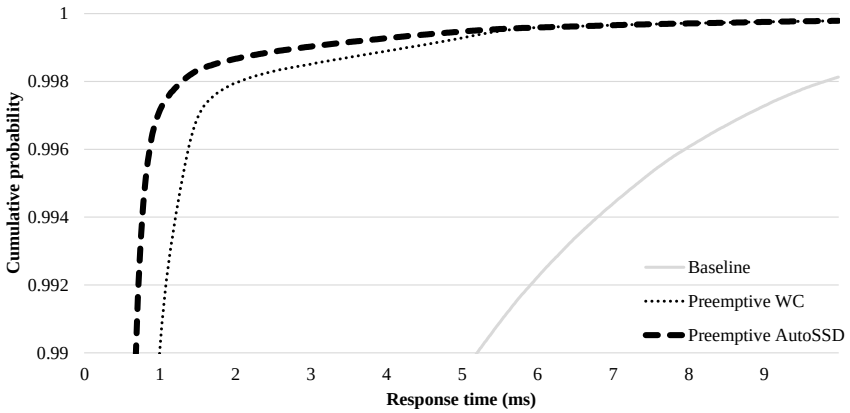
Figure 17: Response time CDF under LM−TBE, MSN−BEFS, and MSN−CFS.



(a) Response time CDF under RAD−AS.

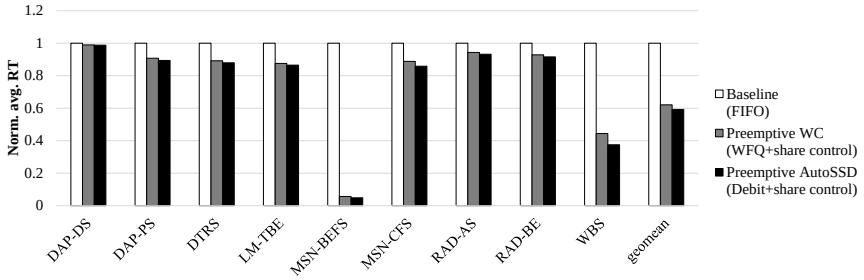


(b) Response time CDF under RAD−BE.

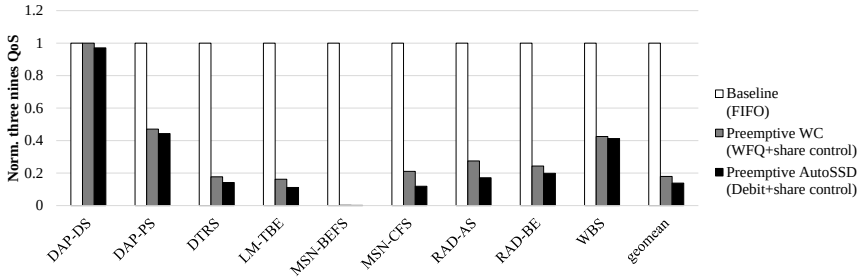


(c) Response time CDF under WBS.

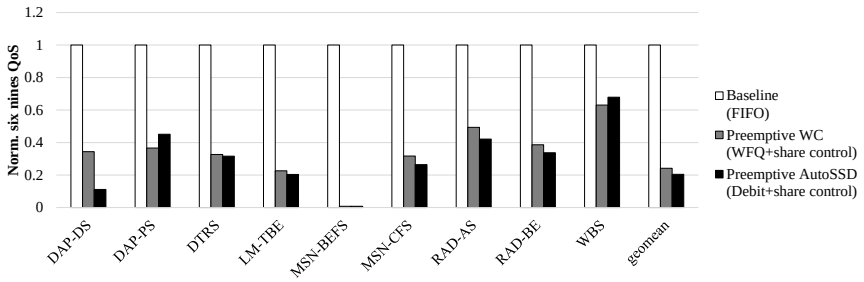
Figure 18: Response time CDF under RAD−AS, RAD−BE, and WBS.



(a) Average response time at 2x intensity.



(b) Three nines QoS at 2x intensity.

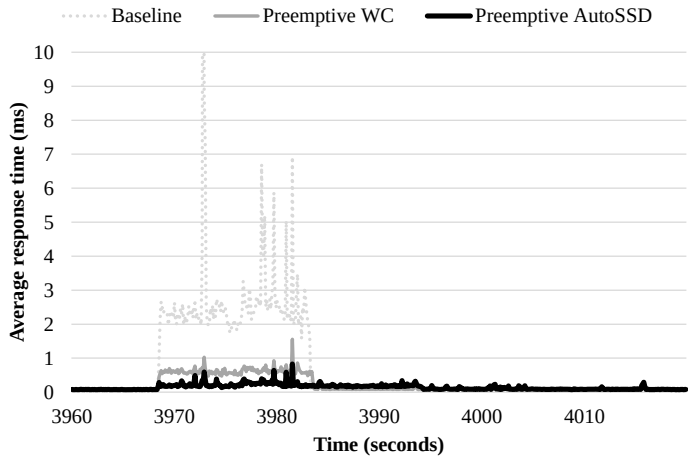


(c) Six nines QoS at 2x intensity.

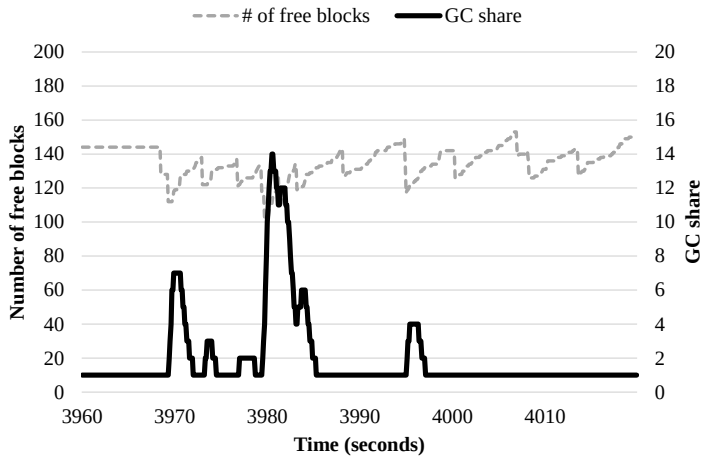
Figure 19: Comparison of the baseline, work-conserving, and AutoSSD for the nine traces at 2x intensity. Results are normalized to the performance of the baseline at 2x intensity. Under a more stressful condition, AutoSSD reduces the average response time by 40.8% on average (at most 95.2% under `MSN-BEFS`), reduces the three nines QoS by 86.1% on average (at most 99.8% under `MSN-BEFS`), and reduces the six nines QoS by 79.5% on average (at most 99.2% under `MSN-BEFS`).

from the normal intensity (cf. Figure 11a). In addition, AutoSSD reduces the six nines QoS by 79.5% as shown in Figure 19c. Particularly, MSN-BEFS improves the most across the two performance metrics. This is because of the already high request intensity of MSN-BEFS, severely degrading the performance of the baseline system. On average, AutoSSD's improvement in tail latency reduction is greater than that of WC. However, we notice that under workloads DAP-PS and WBS, WC outperforms AutoSSD. Both these workloads are characterized by high burst accesses, as evident by the large difference between the average and the median inter-arrival times in Table 3. For them, fully utilizing resources for block reclamation with WFQ allows WC to start with more free blocks when burst writes arrive and results in improved long-tail latency. On the other hand, for other workloads with more sustained characteristics such as MSN-CFS and RAD-AS, AutoSSD performs better in long-tail latency reduction than WC.

We examine MSN-BEFS more closely in Figure 20. Figure 20a shows the average response time of the three systems, approximately 1 hour into MSN-BEFS run at a higher intensity. GC is active during this time for all the three systems, and the baseline system with FIFO scheduling exhibits a large response time spike that reaches up to 22ms (not within the range of the y-axis). The high workload intensity makes it difficult for the baseline system to bound the response time; on the other hand, the work-conserving system and AutoSSD can reduce the performance degradation caused by GC. Figure 20b shows the number of free blocks and the GC share for AutoSSD. Similar to the results in the previous chapter, the share for GC reactively increases when the number of free blocks is low, and it decays once the



(a) Average response time under `MSN-BEFS`.



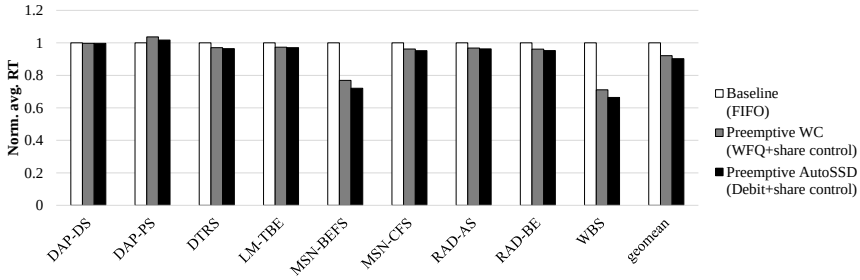
(b) Free blocks and GC share under `MSN-BEFS`.

Figure 20: Comparison of the baseline, work-conserving, and AutoSSD under `MSN-BEFS` running at 2x intensity. Figure 20a shows the average response time sampled at 100ms. Figure 20b shows the number of free blocks and the GC share of AutoSSD.

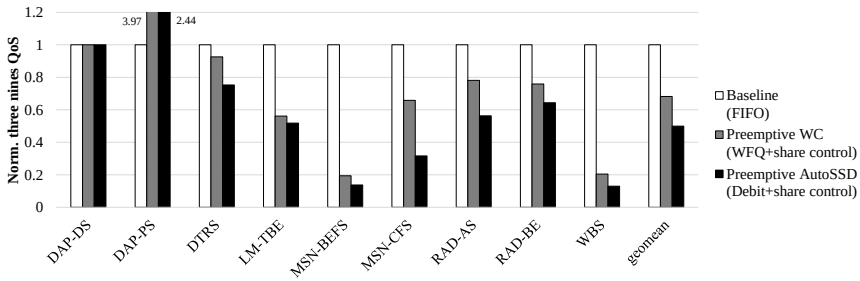
number of free blocks reaches a stable region.

Figure 21 compares the performance under lower workload intensity. On average, AutoSSD reduces the average response time by 9.8% as shown in Figure 21a. The performance improvement of AutoSSD is reduced compared to that of the normal intensity (cf. Figure 11a). However, this is not because AutoSSD performs worse under reduced intensity, but because the baseline system performs much better instead. AutoSSD improves the most under workloads with a relatively high intensity such as `WBS` and `MSN-BEFS`, showing average response time reduction of 33.6% and 27.9% respectively. As for the long tail latency, AutoSSD reduces the six nines QoS by 67.7% on average as shown in Figure 21c. Similar to the observations from the results with the higher intensity, WC outperforms AutoSSD in terms of six nines QoS for workloads with high burst accesses such as `DAP-PS` and `WBS`. Overall, however, AutoSSD exhibits better tail latency characteristics over WC, and improves six nines QoS the most for `DAP-DS`, reducing it by 87.5%.

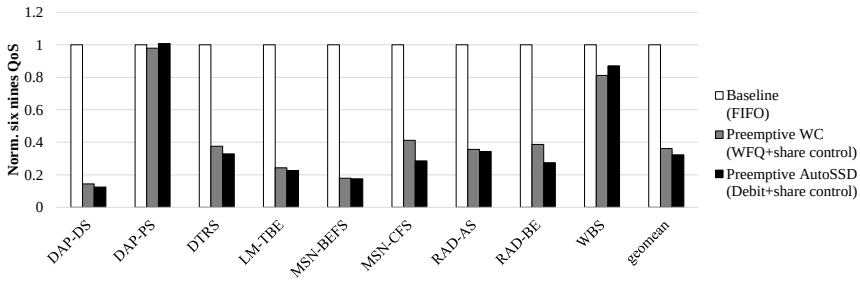
We examine `DAP-DS` more closely in Figure 22. It shows the maximum response time of the three systems, at the start of `DAP-DS` run at a lower intensity. GC is active during this time for all the three systems, and the baseline system with FIFO scheduling exhibits a large response time spike that reaches up to 22ms (not within the range of the y-axis). WC and AutoSSD, on the other hand, are better able to bound the performance degradation caused by an active garbage collection.



(a) Average response time at 0.5x intensity.



(b) Three nines QoS at 0.5x intensity.



(c) Six nines QoS at 0.5x intensity.

Figure 21: Comparison of the baseline, work-conserving, and AutoSSD for the nine traces at 0.5x intensity. Results are normalized to the performance of the baseline at 0.5x intensity. AutoSSD reduces the average response time by 9.8% on average (at most 33.6% under `WBS`), reduces the three nines QoS by 50.0% on average (at most 87.0% under `WBS`), and reduces the six nines QoS by 67.7% on average (at most 87.5% under `DAP-DS`).

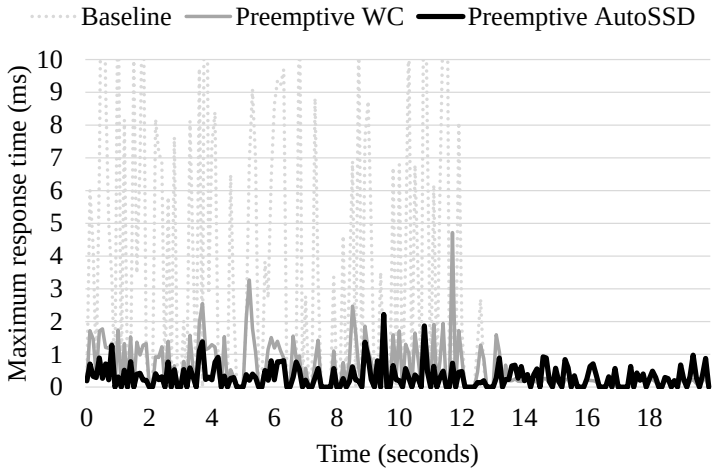


Figure 22: Maximum response time of the baseline, work-conserving, and AutoSSD under DAP-DS running at 0.5x intensity.

5.4 I/O trace results with collocated workloads

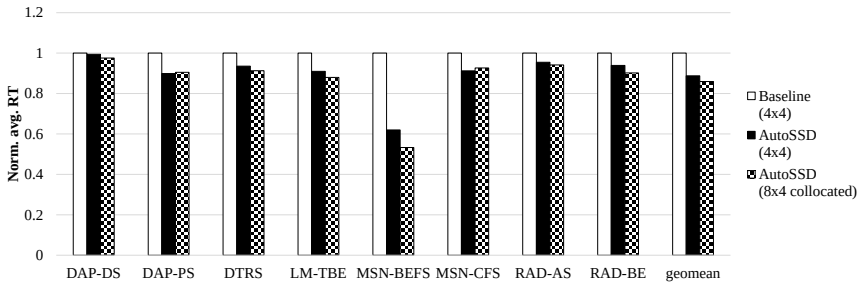
In this section, we examine the performance of AutoSSD when workloads are collocated (i.e., workloads from different VMs share the same storage). The trace workloads are paired in the following manner: First, workloads with similar durations are paired, combining MSN-CFS with MSN-BEFS, and RAD-AS with RAD-BE. Then, the workload with the smallest average inter-arrival time is paired with the one with the largest, combining LM-TBE with DAP-PS, and DTRS with DAP-DS. For this experiment, we offset the logical address so that the address ranges from each paired workload do not overlap, but the host request dispatch times are intermingled. This combined workload is run on a larger system with double the capacity, and we measure the response time for requests from each VM.

We compare this performance against the workload running alone with the original configuration.

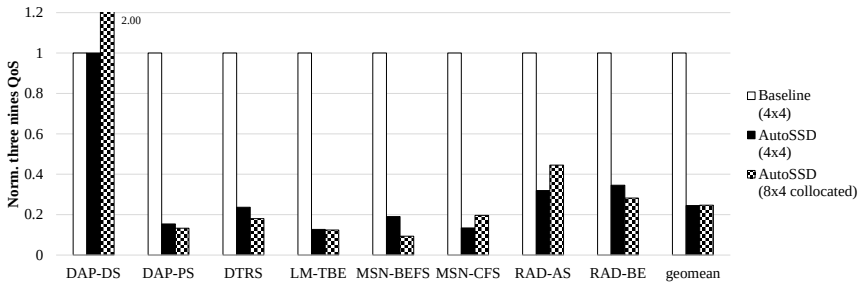
Figure 23 shows the results for collocated workloads. It compares the average response time of the combined VMs sharing a larger AutoSSD system against the baseline and AutoSSD running alone. On average, AutoSSD with collocated workloads reduces average response time by 14.1% (see Figure 23a), and the six nines QoS by 74.9% (see Figure 23c) against the baseline system. Although the improvement is marginal compared to the AutoSSD running alone, it shows that workloads with a high request intensity such as `MSN-BEFS` benefit from the increased parallelism, while performance degradation from workload interferences are negligible for those with lower intensity. This result shows that the architecture of AutoSSD is desirable over a simple resource partitioning scheme, as it can achieve high performance even under mixed workloads.

5.5 Sensitivity analysis with I/O trace workloads

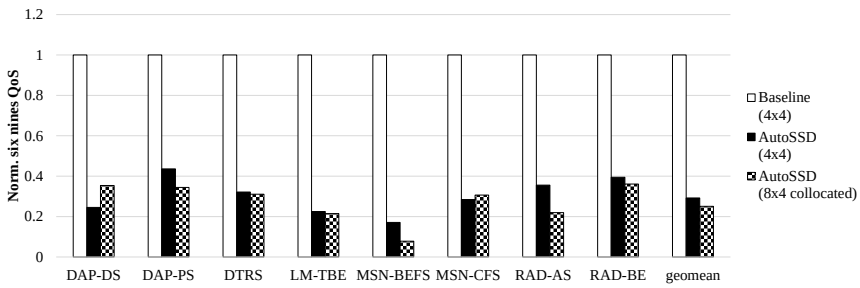
Lastly, we perform sensitivity analysis on key AutoSSD parameters and flash memory reliability characteristics. First, we vary the parameter value for the scheduler called the *concurrency level* that scales the debt limit (maximum allowed outstanding requests) per task, and examine its effect on performance. Then, we investigate the effects of the proportional and integral coefficients for the share controller by varying them independently and measuring the performance. Lastly, we reduce the activation and deactivation thresholds for the read scrubbing task to observe the impact of



(a) Average response time with workload collocation.



(b) Three nines QoS with workload collocation.



(c) Six nines QoS with workload collocation.

Figure 23: Comparison of the baseline running alone, AutoSSD running alone, and AutoSSD with workload collocation. Results are normalized to the performance of the baseline running alone. For collocation, the workload pairs are `MSN-CFS` and `MSN-BEFS`, `RAD-AS` and `RAD-BE`, `LM-TBE` and `DAP-PS`, and `DTRS` and `DAP-DS`. AutoSSD with two workloads collocated in an 8x4 system reduces the average response time by 14.1% on average, reduces the three nines QoS by 75.3% on average, and reduces the six nines QoS by 74.9% on average.

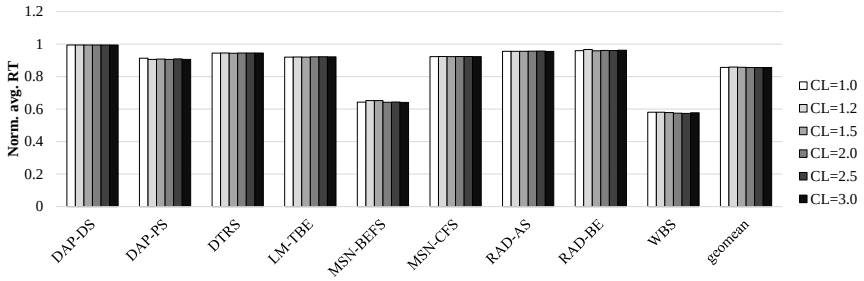
worsening flash memory reliability traits on the overall performance.

5.5.1 Debit scheduler parameters

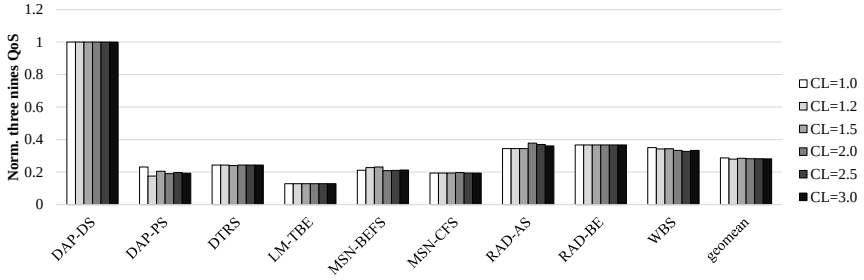
The debit scheduler enforces resource arbitration by limiting the number of outstanding requests per task. The maximum allowed outstanding requests is called the debt limit for each task, and this value is proportional to the assigned share. The parameter concurrency level (CL) of the scheduler scales the debt limit. Low CL means that there are less outstanding requests in the flash memory subsystem: this can cause under-utilization of resources, but may also improve system responsiveness. High CL, on the other hand, means that there are more outstanding requests queued in the subsystem: this improves the utilization of resources, but may cause longer queueing delays.

For a flash memory subsystem with n resources, CL value of 1 means that there can be at most $1 \times n$ outstanding requests across all tasks. CL value of 2 is used in previous experiment results (up to two requests can be queued for each resource), and in this subsection, we present results by varying CL from 1 to 3. We examine the impact of CL on both non-preemptive and preemptive debit scheduling of AutoSSD.

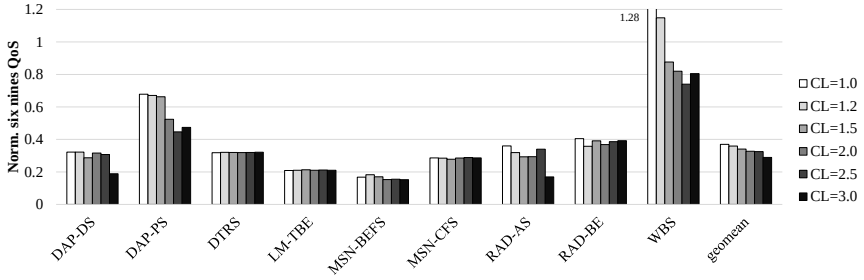
Figure 24 shows the performance of non-preemptive AutoSSD with various CL values across the I/O trace workloads. As shown in Figure 24a, the CL value does not have much impact on the average response time. However, changes in CL significantly affects the long tail latency as shown in Figure 24c. This is most noticeable for `WBS` that shows the downside of setting CL too low or too high. A CL value too small degrades performance



(a) Sensitivity to the concurrency level (CL) parameter of the non-preemptive debit scheduler on the average response time.



(b) Sensitivity to the concurrency level (CL) parameter of the non-preemptive debit scheduler on the three nines QoS.



(c) Sensitivity to the concurrency level (CL) parameter of the non-preemptive debit scheduler on the six nines QoS.

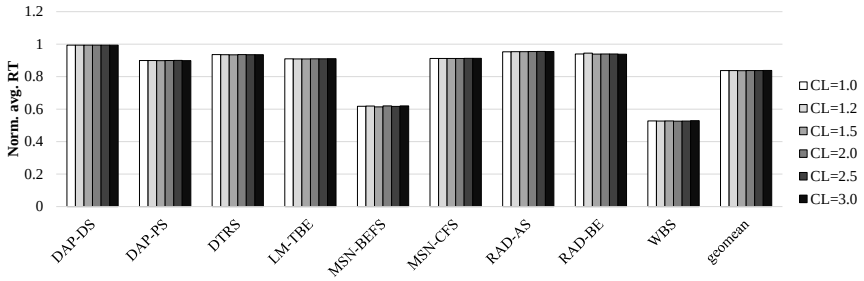
Figure 24: Sensitivity to the concurrency level (CL) parameter of the non-preemptive debit scheduler. While its impact on average response time is negligible, a significant change in performance is observed for the six nines QoS figure.

by under-utilizing resources, and a CL value too large causes longer queueing delays.

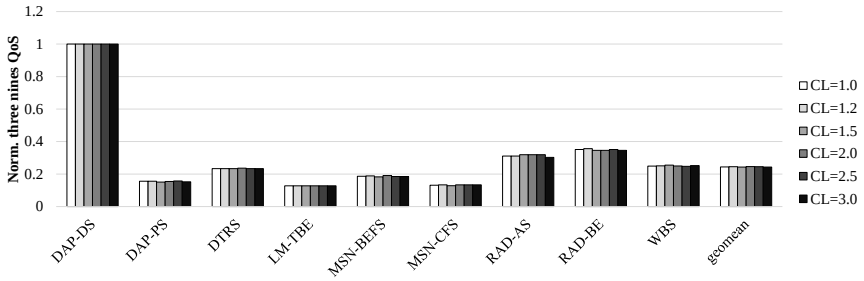
Figure 25 shows the performance of preemptive AutoSSD. Similar to the results of the non-preemptive scheduler analysis (cf. Figure 24a), the CL value does not have much impact on the average response time as shown in Figure 25a. However, unlike the results of the non-preemptive counterpart (cf. Figure 24c), it is difficult to discern the impact on long tail latency for the preemptive scheduler as shown in Figure 25c. Although there certainly are changes in the six nines QoS figure, a significant performance trend is not observed. This is because of the nature of preemptive scheduling: while CL in the non-preemptive scheduler effectively controls the degree of work-conserving-ness, the preemptive scheduler is fully work-conserving and simply controls which task can preempt others more.

5.5.2 Share controller parameters

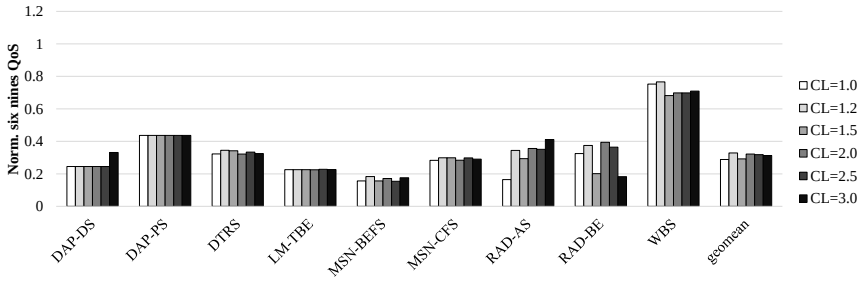
The share controller determines the appropriate share for the FTL management tasks based on the feedback of key system states: number of free blocks for garbage collection task and maximum read count for read scrubbing task. The share is computed using two coefficients: proportional coefficient (`P`) that scales the error value, and integral coefficient (`I`) that accumulates error values by propagating some portion of the past error. Deriving a theoretically optimal coefficient values requires a model for the system; this involves detailed modeling of not only the scheduler and flash memory subsystem, but also all the FTL tasks and host workloads. Instead, in this subsection, we explore the change in performance when varying the two



(a) Sensitivity to the concurrency level (CL) parameter of the preemptive debit scheduler on the average response time.



(b) Sensitivity to the concurrency level (CL) parameter of the preemptive debit scheduler on the three nines QoS.



(c) Sensitivity to the concurrency level (CL) parameter of the preemptive debit scheduler on the six nines QoS.

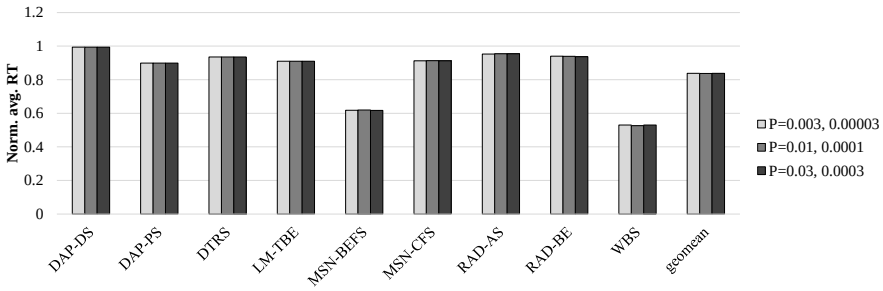
Figure 25: Sensitivity to the concurrency level (CL) parameter of the preemptive debit scheduler. Its impact on average response time is negligible, and the preemptive scheduling makes it difficult to discern its impact on long tail latency.

coefficients for the share controller.

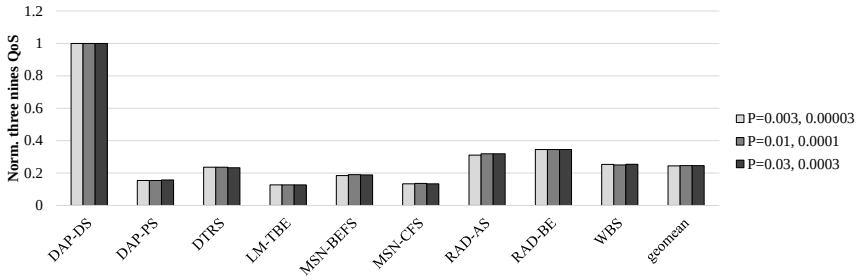
Intuitively, a large P value allows the system to react rapidly to the change in key system states. This is beneficial for workloads with high burstiness or intensity as FTL tasks need to quickly accommodate and adapt to the workload. However, a P value too large may adversely affect performance because a high share of the background task limits the progress for the foreground task. For the integral coefficient, a large I value allows the share to decay gradually and to maintain some level even after key system state becomes stable. A small I, on the other hand, makes the system forget the past activities and makes it highly reliant on current error levels. We examine the effect of P and I by varying them independently and observing the change in performance.

Figure 26 shows the performance of AutoSSD with changes to the P value of the share controller. As shown in Figure 26a, the P value does not have much impact on average response time. However, the change in the proportional value has a noticeable effect on the long tail latency as shown in Figure 26c. In particular, large P is beneficial for high-intensity workloads such as WBS and MSN-BEFS, while small P is better for low-intensity workloads such as RAD-AS and RAD-BE.

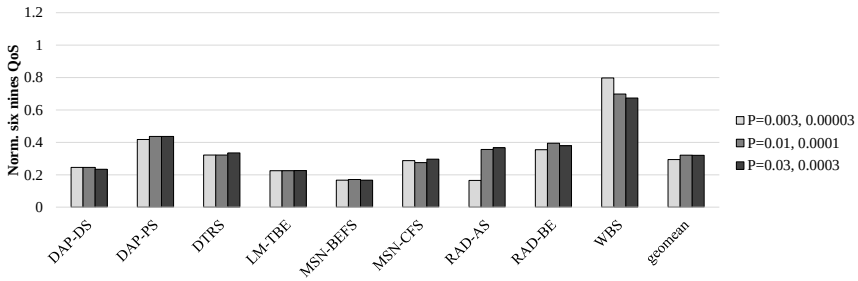
Figure 27 shows the performance of AutoSSD with respect to changes in the I value. Similar to the results of sensitivity testing of the P value (cf. Figure 26a), the integral coefficient does not have much impact on the average response time. However, the long tail latency in Figure 27c shows that large I is beneficial for high-intensity workloads such as WBS, while small I is better for low-intensity workloads such as RAD-AS. These results



(a) Sensitivity to the proportional coefficient of the share controller on the average response time.

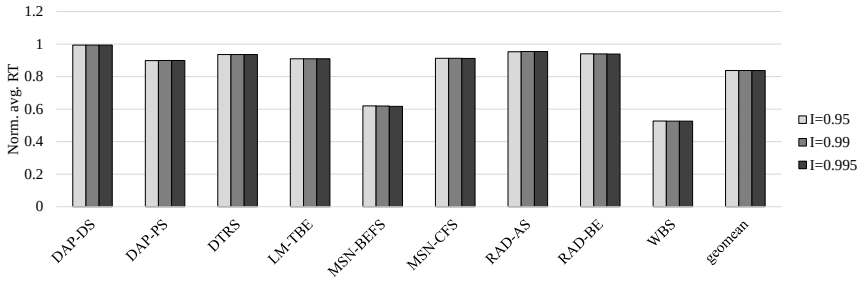


(b) Sensitivity to the proportional coefficient of the share controller on the three nines QoS.

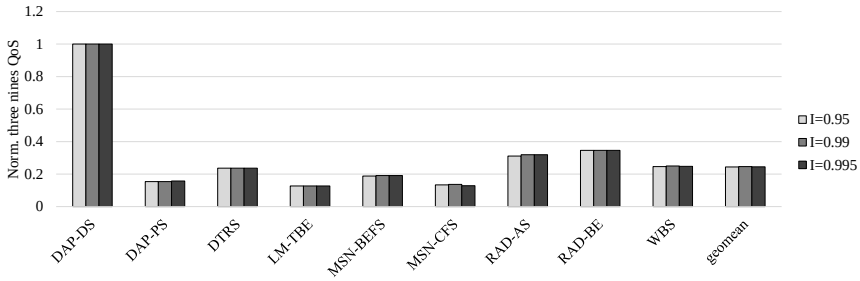


(c) Sensitivity to the proportional coefficient of the share controller on the six nines QoS.

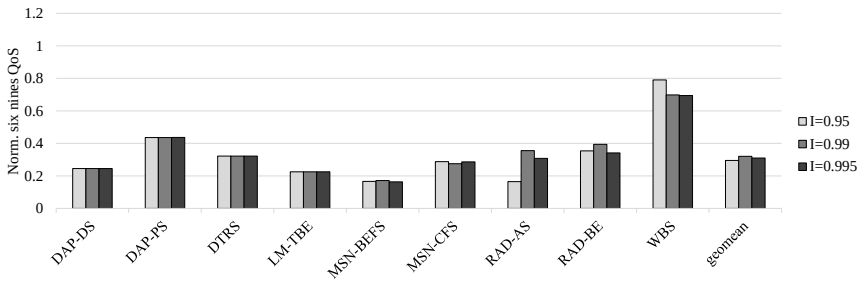
Figure 26: Sensitivity to the proportional coefficient of the share controller. While its impact on average response time is negligible, noticeable change in performance is observed for the six nines QoS figure.



(a) Sensitivity to the integral coefficient of the share controller on the average response time.



(b) Sensitivity to the integral coefficient of the share controller on the three nines QoS.



(c) Sensitivity to the integral coefficient of the share controller on the six nines QoS.

Figure 27: Sensitivity to the integral coefficient of the share controller. While its impact on average response time is negligible, noticeable change in performance is observed for the six nines QoS figure.

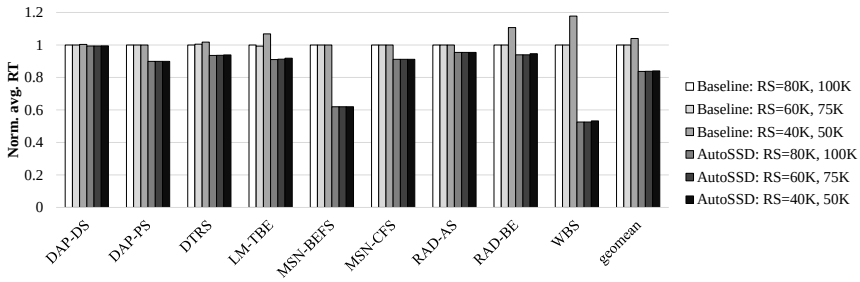
on sensitivity testing on P and I values are consistent with the intuitive explanation that we reasoned earlier in this subsection.

5.5.3 Read scrubbing thresholds

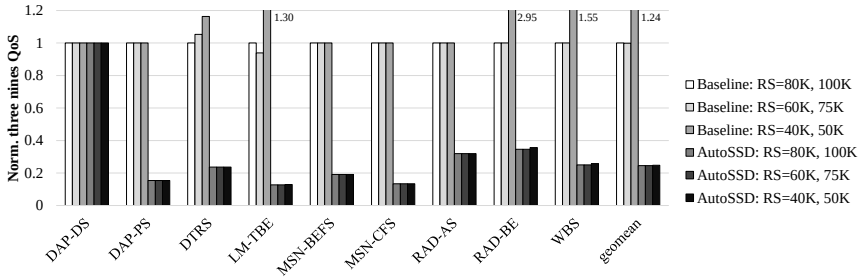
Read disturbance is one of the major concerns among the deteriorating reliability characteristics of flash memory as repeated reads can increase the error rate of neighboring data. Read scrubbing (RS) is an FTL task that preventively migrates data before data loss, and uses the number of reads for each block as a proxy to the error rate of the data within the block. As flash memory's reliability characteristics worsen, the effects of read disturbance will become greater, requiring more frequent read scrubbing to maintain data integrity. The experiment presented in this subsection varies the activation and deactivation threshold for the read scrubbing task and observes its impact on performance.

The default threshold values for RS are set to 100,000 for activation, and 80,000 for deactivation. Here, we further reduce them to 75,000 and 50,000 for activation, and 60,000 and 40,000 for deactivation. We measure the performance of both the baseline system and preemptive AutoSSD.

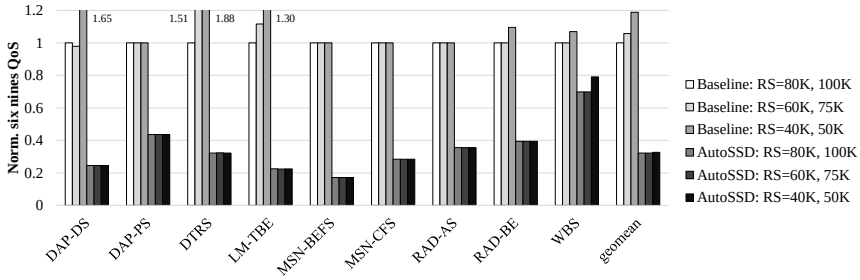
Figure 28 shows the performance impact of lower RS thresholds on both the baseline and AutoSSD. In general, reduced RS thresholds degrade the overall performance of the system by increasing RS traffic. However, while the performance of the baseline system degrades notably, as evident by the 18% increase in average response time of WBS, the performance degradation of AutoSSD is negligible with only a 1.1% increase on average as shown in Figure 28a. The difference between the two systems is



(a) Sensitivity to the read scrubbing thresholds on the average response time.



(b) Sensitivity to the read scrubbing thresholds on the three nines QoS.



(c) Sensitivity to the read scrubbing thresholds on the six nines QoS.

Figure 28: Sensitivity to the read scrubbing thresholds. The performance of the baseline system degrades as flash memory characteristics worsen, while AutoSSD minimizes the performance degradation caused by increased read scrubbing traffic.

especially noticeable in terms of long tail latency as shown in Figure 28c. The performance of the baseline system degrades by 19% on average and as much as 88% for `DTRS`. AutoSSD, on the other hand, shows a negligible impact on the six nines QoS figure from increased read scrubbing traffic with only a 1.4% increase on average.

Chapter 6

Related work

6.1 Real-time FTL

There have been several studies on real-time performance guarantees of flash storage [9, 11, 54, 70]. They focus on bounding the worst-case response time, but depend on RTOS support [9], specific mapping schemes [11, 54, 70], a number of reserve blocks [11, 54], and flash operation latencies [70]. These tight couplings make it difficult to extend performance guarantees when system requirements and flash memory technology change. On the other hand, the proposed architecture is FTL implementation-agnostic, allowing it to be used across a wide range of flash devices and applications.

- RTGC [9] is the first study that addresses the need for a predictable block reclamation scheme to provide real-time performance guarantees in flash storages. To achieve this, it uses a real-time operating system for scheduling and implements garbage collection as a periodic task with real-time constraints. However, by modeling garbage collection as a non-preemptible task that blocks other storage accesses while it is active, the worst-case performance bound of RTGC is very high.

- GFTL [11] introduces the idea of partial garbage collection to re-

duce the worst-case response time. It allows the interleaving of host request handling and garbage collection with the use of centralized write buffer blocks to decouple the two tasks. However, the backing store is organized using a block-level mapping scheme that incurs more valid page copies for random workloads; thus the average-case performance of GFTL is poor.

- RFTL [54] improves both the worst-case and the average-case performance through a distributed partial garbage collection. Similar to GFTL, host and garbage collection requests can be interleaved, but RFTL is organized using a hybrid mapping scheme with a write buffer block for each logical block. While this scheme removes the centralized bottleneck and improves average performance, it comes at a heavy cost of a large overhead in storage capacity, as only 50% of physical blocks can be used to store host data.

- WAO-GC [70] defers garbage collection until the very last moment at which any further delay will violate the worst-case response time. This improves the average-case performance by increasing GC efficiency, and the storage overhead is reduced compared to RFTL as it uses a page-level mapping scheme instead. However, in order to bound the worst-case response time to the flash erase latency, it still requires a large amount of over-provisioning, as high as 40% with the current state-of-the-art flash memory parameters.

6.2 Scheduling techniques inside the SSD

This section covers FTL-agnostic scheduling techniques that are implemented inside the SSD. Prior work such as PGC [40], HIOS [27], ITEX [50], and RL-assisted scheduling [30] are implemented at the host interface side and focus on when to perform GC (based on threshold [40], slack [27], or host idleness [30, 50]). These approaches complement our design that focuses on the fine-grained scheduling of multiple FTL tasks running concurrently. ttFlash [67] and QoSFC [34] are FTL-agnostic and are implemented at the flash memory interface side, allowing host and GC to share the flash memory subsystem at a finer granularity. However, ttFlash relies on a RAID-like parity scheme and flash memory's copy-back operation, and QoSFC has a high resource overhead for implementing a fair queueing scheduler for every flash memory chip.

- In PGC [40], host I/O requests are prioritized over GC operations. While servicing host requests, the progress of GC is halted until host workload is idle, or until the system hits a hard limit (i.e. almost out of free blocks). In this design, the host requests and the FTL internal tasks do not share the resources fairly, and it is exhibits long tail latency once the system is out of free blocks.

- ITEX [50] detects idle periods in the host workload and schedules FTL tasks such as garbage collection during this period. The decision module based on a hidden Markov model determines when to schedule garbage collection through an adaptive time-out method.

The measured inter-arrival time is feedbacked to the decision module to increase the estimation accuracy. Thus, the performance of ITEX heavily depends on the predictability of host requests, and although ITEX is implemented inside the flash storage, it remains oblivious to the details of the flash memory subsystem, limiting its scheduling efficiency.

- HIOS [27] also schedules contending FTL tasks based on host idleness, but differs from ITEX as it requires explicit deadline assignments for host requests. The deadlines are marked using the *PRIO* and *ICC* fields in the SATA protocol, and once enough slack is detected, garbage collection requests are scheduled. In addition, HIOS detects channel conflicts and avoids scheduling to channels with high contention. However, its dependence on deadline assignments and limited awareness to the scheduling in the flash memory subsystem restrict the benefits of this technique.

- RL-assisted scheduling [30] predicts the duration of host idle periods and determines the number of GC operations to be executed during these idle periods. It uses reinforcement-learning to train the scheduling agent by rewarding it with the outcome of the selected policy given the history of host request inter-arrival times. Similar to ITEX, this technique heavily depends on the predictability of the host workload, and does not allow fine-grained scheduling of FTL tasks at the flash memory subsystem.

- ttFlash [67] reduces the tail latency of host requests caused by garbage

collection with the use of a RAID-like parity scheme and the copyback operation of flash memory. The parity scheme allows reads to be serviced by reconstructing data from other flash memory chips. However, ttFlash does not allow interleaving of host and garbage collection request on resources used by garbage collection. Moreover, the reconstruction of data through parity increases the overall traffic to the flash memory subsystem, and the use of copyback operations accumulates error, and is not recommended for use even in state-of-the-art flash memory chips.

- QoSFC [34] improves both the average response time and the 99.9% QoS by arbitrating flash memory requests from host request handler and garbage collection with a fair queueing scheduler [13, 16]. Although QoSFC achieves fine-grained scheduling and a high degree of fairness, its design is limited to arbitrating two FTL tasks, and the resource overhead is high as it implements fair queueing scheduling for every flash memory chip.

6.3 Scheduling for SSD performance on the host system

Optimizing I/O performance on the host side has been an active area of research. Prior works include improving I/O schedulers to take advantage of SSD's performance characteristics (IRBW [37], FIOS [51], FlashFQ [58], and ParDispatcher [64]), coordinating multiple SSDs to hide GC-induced performance degradation (Harmonia [39], Rails [60], and Storage engine

[59]), and directly managing the flash memory hardware (SDF [49], LightNVM [4]).

- IRBW [37] is an I/O scheduler that arranges write requests into larger bundles and schedules reads independently with a higher priority. The bundling of writes mitigates flash storage's vulnerability to small random writes as they cause more garbage collection overhead. However, the bundling of writes reorders requests and may violate the durability ordering requirements for the storage.

- FIOS [51] takes advantage of the faster response time and higher throughput of flash storage by issuing multiple I/O requests from different host threads as long as the fairness is not violated. It tracks fairness using timeslice management, and implements an anticipatory scheduling scheme to prevent host threads from relinquishing the timeslice prematurely. However, FIOS remains oblivious to the internal FTL tasks such as garbage collection that run inside the flash storage, and simply only exploits the high performance of flash storage.

- FlashFQ [58] is an extension of FIOS that applies fair queueing algorithm [13, 16] to manage fairness among host threads (instead of the timeslice management in FIOS). Furthermore, it implements a throttled dispatch mechanism to limit the number of outstanding requests of a single host thread in order to prevent overload. Like IRBW and FIOS, FlashFQ is oblivious to the internal details of the flash storage.

- ParDispatcher [64] attempts to exploit parallelism inside the flash storage while treating it as a black box. It divides the LBA space into sub-regions, each with a sub-queue, and dispatches requests from the sub-queues in a round-robin manner. This approach assumes that the mapping scheme inside the flash storage divides the LBA space equally among multiple flash memory chips, and thus dispatching in this fashion would increase I/O parallelism. However, this assumption does not take into consideration of other FTL tasks, and is an over-simplification of the flash storage and the intricacies of mapping schemes.

- Harmonia [39] coordinates the GC activities of SSDs belonging to the same RAID array group. By suppressing and triggering GC activities through a custom control interface, it creates a window of time when no garbage collection is active for all the SSDs. While this allows time periods free of active running GC, it does not address performance degradation while GC is active. Furthermore, the custom control interface is vendor-specific and non-standard, making it difficult to be universally used.

- The architecture of Rails [60] requires an I/O array made of redundant flash storage devices, and each device is put in either read mode or write mode. The modes periodically switch, and read requests are serviced only from drives under read mode. Write requests are buffered and written in bulk when the device is in write mode. This scheme physically separates reads and writes such that the performance of

read requests are not degraded by write requests and garbage collection caused by writes. However, there is no guarantee that the devices in read mode will not internally perform garbage collection (or any internal FTL task for that matter), and the effective storage capacity is halved by redundancy.

- The storage engine [59] improves upon Rails by suppressing the activation of garbage collection when under read mode, and forcing garbage collection when in write mode through a control API of a Samsung SSD. Similar to the limitations of Harmonia, this interface is not only vendor-specific, but also non-standard, making it difficult to be universally used. Like Rails, the storage capacity reduction still remains an issue in this work.

- SDF [49] exposes individual flash memory channels to the host software that directly manages the raw flash memory hardware. This allows the entire system to be vertically integrated from the flash memory chip to the key-value store used in Baidu's data centers. However, this also exposes the eccentricities of flash memory, placing a burden on the host system.

- LightNVM [4] also exposes flash memory operations of erase, program, and read to the host system. However, some flash memory's quirks such as bad blocks and error-prone nature are hidden from the host system by delegating their handling to the device. While this approach attempts to allow cross-vendor compatibility and abstraction, it still complicates the host system when multiple SSDs of different

flash memory generations are used together as storage.

6.4 Performance isolation of SSDs

Performance isolation of SSDs aims to reduce performance variation caused by multiple hosts through partitioning resources (vFlash [61]), improving GC efficiency by grouping data from the same source (Multi-streamed [29], and OPS isolation [36]), and penalizing noisy neighbors (WA-BC [26]). Partitioning resources for isolation is a double-edged sword as it also slashes the average performance by reducing the parallelism in flash storage; segregating data from different hosts may improve the overall efficiency of garbage collection; penalizing host that induces more garbage collection benefits the performance of those that do not. These performance isolation techniques are complementary to our approach of fine-grained scheduling of concurrent FTL tasks.

- vFlash [61] physically partitions the hardware and dedicates separate channels and chips to each VM. However, this also reduces the parallelism in the flash storage and can cause resource under-utilization if the workloads from the VMs are skewed. Furthermore, it binds performance isolation not only with parallelism, but also with usable capacity, making it inefficient when required performance and capacity of the VMs are asymmetric.

- Multi-streamed SSD [29] tags host requests with similar expected lifetime together, and allocates blocks separately for each tag (called *stream*). In doing so, it improves the efficiency of garbage collection,

thereby also improving the overall performance. However, this technique does not explicitly address garbage collection-aware scheduling or performance isolation among hosts.

- OPS isolation [36] further improves multi-streamed SSD by managing the pool size of reserved blocks for each VM. Adjusting the pool size affects the GC efficiency for each VM, and this is used to control each VM's performance. However, there are other FTL tasks besides garbage collection that affects the performance, and ideal isolation requires direct management of the FTL task's progress.

- WA-BC [26] selects requests from different VMs in a round-robin manner, and compensates or penalizes the scheduling budget based on which VM caused more garbage collection. While WA-BC does not directly deal with FTL tasks scheduling, it indirectly contributes to it by throttling host requests from different VMs.

6.5 Scheduling in shared disk-based storages

The design of the autonomic SSD architecture borrows ideas from prior work on shared disk-based storage systems such as Façade [44], PARDA [19], and Maestro [46]. These systems aim to meet performance requirements of multiple clients by throttling request rates or limiting the number of outstanding I/Os, and dynamically adjusting the bound through a feedback control. However, while these disk-based systems deal with fair sharing of disk resources among multiple hosts, we address the interplay between foreground (host I/O) and background work (garbage collection and

other management schemes). Aqueduct [43] and Duet [2] address the performance impact of background tasks such as backup and data migration in disk-based storage systems. However, background tasks in flash storage are triggered at a much smaller timescale, and SSDs uniquely create scenarios where the foreground task may wait for the background task, necessitating a different approach.

- Façade [44] throttles the number of I/O requests from multiple clients so that the disks do not saturate. It schedules requests from each client based on earliest deadline first order, and adjusts the total number of requests that can be queued for each device through latency feedback. While the core mechanism of feedback control on the number of outstanding requests is similar to that of ours, the relationship between the FTL task's progress and share is more complex than that of performance and share, making it difficult to directly apply control algorithms from disk-based shared storage.

- PARDA [19] uses latency measurement of a request as feedback, and adjusts the number of requests that can be issued to provide fairness to clients. The feedback control used in this technique is similar to that of TCP congestion control [24]; it decreases the number of outstanding requests if the latency feedback exceeds a set threshold, and increases it if the overload subsides.

- Maestro [46] provides performance differentiation among multiple applications in large disk arrays. It achieves this through a feedback controller that adjusts the total number of outstanding requests for

each application, and the number of requests that can be queued for each disk is proportionately divided based on the recently-observed demand from each application. However, in this scheme, the scheduler views each disk as a black box and does not limit the number of requests that can be queued for each disk.

- Aqueduct [43] reduces performance degradation caused by data migration by throttling the rate at which data migration can proceed. In this study, however, there is no dependence between foreground work and background work. In SSDs, on the other hand, deferring garbage collection indefinitely will cause host requests to experience performance degradation.

- Duet [2] prioritizes maintenance (background) tasks that have data currently cached in memory. This technique opportunistically schedules background work so that it not only benefits the background work by increasing its efficiency, but also mitigates its impact on foreground work by reducing cache pollution. Like Aqueduct, however, this does not cover the intricate scenario when the foreground work depends on the background work.

Chapter 7

Conclusion

7.1 Summary

In this dissertation, we presented the design of an autonomic SSD architecture that self-manages concurrent FTL tasks in the flash storage. By judiciously coordinating the use of resources in the flash memory subsystem, the autonomic SSD manages the progress of concurrent FTL tasks and maintains the internal system states of the storage at a stable level. This self-management prevents the SSD from falling into a critical condition that causes long tail latency. In effect, the presented autonomic SSD reduces the average response time by up to 16.2% and the six nines (99.9999%) QoS by 67.8% on average for QoS-sensitive small reads. The three main contributions are (1) the virtualization of the flash memory subsystem, thereby decoupling FTL tasks and their scheduling, (2) a simple yet effective resource arbitration scheme that enforces resource share and (3) the feedback control of the shares of FTL tasks to maintain a stable system state. We evaluated our design and show that it achieves predictable performance across diverse workloads, with multiple FTL tasks such as garbage collection, read scrubbing, and map cache management running concurrently with host request handling.

7.2 Future work and directions

Future work beyond this study include, but are not limited to, the followings:

- *Intra-task scheduling.* The architecture presented in this dissertation focuses on inter-task scheduling through the dynamic control of shares, and treats requests from the same task equally. However, there are needs to explicitly prioritize a subset of requests to manage the outliers in response time. This can be achieved by request prioritization and intra-task scheduling that are complementary to the presented share-based inter-task scheduling.

- *Prototype validation.* Development platforms such as Open-Channel SSD [4] and OpenSSD [62] are possible candidates for implementing the autonomic SSD architecture and validating the performance benefits on a real system.

- *Integration with host workload prediction.* In this dissertation, we maintain the share of an FTL internal task at a minimum level to improve the responsiveness for host requests. However, with host workload prediction, we can design a share control policy that would increase the background task's shares when it is confident that the host would remain idle, and decrease them when it anticipates the arrival of host requests.

- *Scheduling with multiple hosts with asymmetric performance requirements.* The experiment results with collocated workloads showed that

the autonomic SSD architecture achieves high performance while perserving flash parallelism. However, the design itself is unaware of the collocated workloads, and the performance benefits are byproducts of the architecture. We can extend the SSD design to be aware of multiple hosts and their performance requirements by managing the individual progress of multiple hosts.

- *Performance-reliability tradeoff.* Although we implemented the read scrubbing task that preventively migrates data before data loss, the actual error and reliability characteristics of the memory has not been modeled. The management of FTL tasks can be enhanced by taking into consideration of in-device error correction capabilities. Furthermore, advanced techniques such as read retry (reads after adjusting the threshold voltage) and RAID-like parity scheme add complexity to this performance-reliability tradeoff.

- *Optimal scheduling and competitive analysis.* Given oracle knowledge of the host workload, we can determine the optimal scheduling for requests from concurrent FTL tasks offline. In the autonomic SSD architecture, dynamic manipulation of shares can approximate this optimal scheduling, and we can competitively analyze the effectiveness of the online scheduling algorithms to the offline oracle scheduling.

- *Theoretical modeling and optimal control.* The complexity and interaction among multiple FTL tasks and SSD subsystems make it difficult to perfectly model the system theoretically. However, theoretical

modeling not only allows the analysis of upper and lower bounds for performance, but also enables the study of optimal control for the share controller.

Bibliography

[1] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D Davis, Mark S Manasse, and Rina Panigrahy. Design tradeoffs for ssd performance. In *USENIX Annual Technical Conference*, volume 8, pages 57–70, 2008.

[2] George Amvrosiadis, Angela Demke Brown, and Ashvin Goel. Opportunistic storage maintenance. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 457–473. ACM, 2015.

[3] Seiichi Aritome. *NAND flash memory technologies*. John Wiley & Sons, 2015.

[4] Matias Bjørling, Javier González, and Philippe Bonnet. Lightnvm: The linux open-channel ssd subsystem. In *FAST*, pages 359–374, 2017.

[5] John S Bucy, Jiri Schindler, Steven W Schlosser, and Gregory R Ganger. The disksim simulation environment version 4.0 reference manual (cmu-pdl-08-101). *Parallel Data Laboratory*, page 26, 2008.

[6] Yu Cai, Erich F Haratsch, Onur Mutlu, and Ken Mai. Error patterns in mlc nand flash memory: Measurement, characterization, and analysis. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 521–526. EDA Consortium, 2012.

[7] Yu Cai, Gulay Yalcin, Onur Mutlu, Erich F Haratsch, Adrian Cristal, Osman S Unsal, and Ken Mai. Flash correct-and-refresh: Retention-aware error management for increased flash memory lifetime. In *Computer Design (ICCD), 2012 IEEE 30th International Conference on*, pages 94–101. IEEE, 2012.

[8] Li-Pin Chang. On efficient wear leveling for large-scale flash-memory storage systems. In *Proceedings of the 2007 ACM symposium on Applied computing*, pages 1126–1130. ACM, 2007.

[9] Li-Pin Chang, Tei-Wei Kuo, and Shi-Wu Lo. Real-time garbage collection for flash-memory storage systems of real-time embedded systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 3(4):837–863, 2004.

[10] Mei-Ling Chiang, Paul CH Lee, Ruei-Chuan Chang, et al. Using data clustering to improve cleaning performance for flash memory. *Software-Practice & Experience*, 29(3):267–290, 1999.

[11] Siddharth Choudhuri and Tony Givargis. Deterministic service guarantees for nand flash using partial block cleaning. In *Proceedings of the 6th IEEE/ACM/IFIP international conference on Hardware/Software codesign and system synthesis*, pages 19–24. ACM, 2008.

[12] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, 2013.

[13] Alan Demers, Srinivasan Keshav, and Scott Shenker. Analysis and simulation of a fair queueing algorithm. In *ACM SIGCOMM Computer Communication Review*, volume 19, pages 1–12. ACM, 1989.

[14] Peter Desnoyers. Analytic modeling of ssd write performance. In *Proceedings of the 5th Annual International Systems and Storage Conference*, page 12. ACM, 2012.

[15] Eran Gal and Sivan Toledo. Algorithms and data structures for flash memories. *ACM Computing Surveys (CSUR)*, 37(2):138–163, 2005.

[16] Pawan Goyal, Harrick M Vin, and Haichen Chen. Start-time fair queueing: a scheduling algorithm for integrated services packet switching networks. In *ACM SIGCOMM Computer Communication Review*, volume 26, pages 157–168. ACM, 1996.

[17] Laura M Grupp, Adrian M Caulfield, Joel Coburn, Steven Swanson, Eitan Yaakobi, Paul H Siegel, and Jack K Wolf. Characterizing flash

memory: anomalies, observations, and applications. In *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, pages 24–33. IEEE, 2009.

[18] Laura M Grupp, John D Davis, and Steven Swanson. The bleak future of nand flash memory. In *Proceedings of the 10th USENIX conference on File and Storage Technologies*, pages 2–2. USENIX Association, 2012.

[19] Ajay Gulati, Irfan Ahmad, Carl A Waldspurger, et al. Parda: Proportional allocation of resources for distributed storage access. In *FAST*, volume 9, pages 85–98, 2009.

[20] Aayush Gupta, Youngjae Kim, and Bhuvan Urgaonkar. *DFTL: a flash translation layer employing demand-based selective caching of page-level address mappings*, volume 44. ACM, 2009.

[21] Keonsoo Ha, Jaeyong Jeong, and Jihong Kim. An integrated approach for managing read disturbs in high-density nand flash memory. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(7):1079–1091, 2016.

[22] Mingzhe Hao, Gokul Soundararajan, Deepak R Kenchammana-Hosekote, Andrew A Chien, and Haryadi S Gunawi. The tail at store: A revelation from millions of hours of disk and ssd deployments. In *FAST*, pages 263–276, 2016.

[23] Xiao-Yu Hu, Evangelos Eleftheriou, Robert Haas, Ilias Iliadis, and Roman Pletka. Write amplification analysis in flash-based solid state drives. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, page 10. ACM, 2009.

[24] Van Jacobson. Congestion avoidance and control. In *ACM SIGCOMM computer communication review*, volume 18, pages 314–329. ACM, 1988.

[25] Wei Jin, Jeffrey S Chase, and Jasleen Kaur. Interposed proportional sharing for a storage service utility. *ACM SIGMETRICS Performance Evaluation Review*, 32(1):37–48, 2004.

[26] Byunghei Jun and Dongkun Shin. Workload-aware budget compensation scheduling for nvme solid state drives. In *Non-Volatile Memory System and Applications Symposium (NVMSA), 2015 IEEE*, pages 1–6. IEEE, 2015.

[27] Myoungsoo Jung, Wonil Choi, Shekhar Srikantaiah, Joonhyuk Yoo, and Mahmut T Kandemir. Hios: A host interface i/o scheduler for solid state disks. *ACM SIGARCH Computer Architecture News*, 42(3):289–300, 2014.

[28] Myoungsoo Jung and Mahmut Kandemir. Revisiting widely held ssd expectations and rethinking system-level implications. In *ACM SIGMETRICS Performance Evaluation Review*, volume 41, pages 203–216. ACM, 2013.

[29] Jeong-Uk Kang, Jeeseok Hyun, Hyunjoo Maeng, and Sangyeun Cho. The multi-streamed solid-state drive. In *HotStorage*, 2014.

[30] Wonkyung Kang, Dongkun Shin, and Sungjoo Yoo. Reinforcement learning-assisted garbage collection to mitigate long-tail latency in ssd. *ACM Transactions on Embedded Computing Systems (TECS)*, 16(5s):134, 2017.

[31] Manolis Katevenis, Stefanos Sidiropoulos, and Costas Courcoubetis. Weighted round-robin cell multiplexing in a general-purpose atm switch chip. *IEEE Journal on selected Areas in Communications*, 9(8):1265–1279, 1991.

[32] Swaroop Kavalanekar, Bruce Worthington, Qi Zhang, and Vishal Sharda. Characterization of storage workload traces from production windows servers. In *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*, pages 119–128. IEEE, 2008.

[33] Atsuo Kawaguchi, Shingo Nishioka, and Hiroshi Motoda. A flash-memory based file system. In *USENIX*, pages 155–164, 1995.

[34] Bryan S Kim and Sang Lyul Min. Qos-aware flash memory controller. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2017 IEEE*, pages 51–62. IEEE, 2017.

[35] Hong Seok Kim, Eyee Hyun Nam, Ji Hyuck Yun, Sheayun Lee, and Sang Lyul Min. P-bms: A bad block management scheme in parallelized flash memory storage devices. *ACM Transactions on Embedded Computing Systems (TECS)*, 16(5s):140, 2017.

[36] Jaeho Kim, Donghee Lee, and Sam H Noh. Towards slo complying ssds through ops isolation. In *FAST*, pages 183–189, 2015.

[37] Jaeho Kim, Yongseok Oh, Eunsam Kim, Jongmoo Choi, Donghee Lee, and Sam H Noh. Disk schedulers for solid state drivers. In *Proceedings of the seventh ACM international conference on Embedded software*, pages 295–304. ACM, 2009.

[38] Jesung Kim, Jong Min Kim, Sam H Noh, Sang Lyul Min, and Yookun Cho. A space-efficient flash translation layer for compactflash systems. *IEEE Transactions on Consumer Electronics*, 48(2):366–375, 2002.

[39] Youngjae Kim, Sarp Oral, Galen M Shipman, Junghee Lee, David A Dillow, and Feiyi Wang. Harmonia: A globally coordinated garbage collector for arrays of solid-state drives. In *Mass Storage Systems and Technologies (MSST), 2011 IEEE 27th Symposium on*, pages 1–12. IEEE, 2011.

[40] Junghee Lee, Youngjae Kim, Galen M Shipman, Sarp Oral, Feiyi Wang, and Jongman Kim. A semi-preemptive garbage collector for solid state drives. In *Performance Analysis of Systems and Software (ISPASS), 2011 IEEE International Symposium on*, pages 12–21. IEEE, 2011.

[41] Sang-Won Lee, Dong-Joo Park, Tae-Sun Chung, Dong-Ho Lee, Sangwon Park, and Ha-Joo Song. A log buffer-based flash translation layer using fully-associative sector translation. *ACM Transactions on Embedded Computing Systems (TECS)*, 6(3):18, 2007.

[42] Yongkun Li, Patrick PC Lee, and John Lui. Stochastic modeling of large-scale solid-state storage systems: analysis, design tradeoffs and optimization. *ACM SIGMETRICS Performance Evaluation Review*, 41(1):179–190, 2013.

[43] Chenyang Lu, Guillermo A Alvarez, and John Wilkes. Aqueduct: Online data migration with performance guarantees. In *FAST*, volume 2, page 21, 2002.

[44] Christopher R Lumb, Arif Merchant, and Guillermo A Alvarez. Façade: Virtual storage devices with performance guarantees. In *FAST*, volume 3, pages 131–144, 2003.

[45] Dongzhe Ma, Jianhua Feng, and Guoliang Li. Lazyftl: a page-level flash translation layer optimized for nand flash memory. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 1–12. ACM, 2011.

[46] Arif Merchant, Mustafa Uysal, Pradeep Padala, Xiaoyun Zhu, Sharad Singhal, and Kang Shin. Maestro: quality-of-service in large disk arrays. In *Proceedings of the 8th ACM international conference on Autonomic computing*, pages 245–254. ACM, 2011.

[47] Justin Meza, Qiang Wu, Sanjev Kumar, and Onur Mutlu. A large-scale study of flash memory failures in the field. In *ACM SIGMETRICS Performance Evaluation Review*, volume 43, pages 177–190. ACM, 2015.

[48] Eyee Hyun Nam, Bryan Suk Joon Kim, Hyeonsang Eom, and Sang Lyul Min. Ozone (o3): An out-of-order flash memory controller architecture. *IEEE Transactions on Computers*, 60(5):653–666, 2011.

[49] Jian Ouyang, Shiding Lin, Song Jiang, Zhenyu Hou, Yong Wang, and Yuanzheng Wang. Sdf: software-defined flash for web-scale internet storage systems. *ACM SIGPLAN Notices*, 49(4):471–484, 2014.

[50] Sang-Hoon Park, Dong-gun Kim, Kwanhu Bang, Hyuk-Jun Lee, Sungjoo Yoo, and Eui-Young Chung. An adaptive idle-time exploiting method for low latency nand flash-based storage devices. *IEEE Transactions on Computers*, 63(5):1085–1096, 2014.

[51] Stan Park and Kai Shen. Fios: a fair, efficient flash i/o scheduler. In *FAST*, page 13, 2012.

[52] Betty Prince. *Vertical 3D memory technologies*. John Wiley & Sons, 2014.

[53] Performance Test Specification PTS. Solid state storage (sss) performance test specification (pts) enterprise. 2013.

[54] Zhiwei Qin, Yi Wang, Duo Liu, and Zili Shao. Real-time flash translation layer for nand flash memory storage systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2012 IEEE 18th*, pages 35–44. IEEE, 2012.

[55] Mendel Rosenblum and John K Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems (TOCS)*, 10(1):26–52, 1992.

[56] Bianca Schroeder, Raghav Lagisetty, and Arif Merchant. Flash reliability in production: The expected and the unexpected. In *FAST*, pages 67–80, 2016.

[57] Yoon Jae Seong, Eyee Hyun Nam, Jin Hyuk Yoon, Hongseok Kim, Jin-yong Choi, Sookwan Lee, Young Hyun Bae, Jaejin Lee, Yookun Cho, and Sang Lyul Min. Hydra: A block-mapped parallel flash memory solid-state disk architecture. *IEEE Transactions on Computers*, 59(7):905–921, 2010.

[58] Kai Shen and Stan Park. Flashfq: A fair queueing i/o scheduler for flash-based ssds. In *USENIX Annual Technical Conference*, pages 67–78, 2013.

[59] Woong Shin, Myeongcheol Kim, Kyudong Kim, and Heon Y Yeom. Providing qos through host controlled flash ssd garbage collection and multiple ssds. In *Big Data and Smart Computing (BigComp), 2015 International Conference on*, pages 111–117. IEEE, 2015.

[60] Dimitris Skourtis, Dimitris Achlioptas, Noah Watkins, Carlos Maltzahn, and Scott A Brandt. Flash on rails: Consistent flash performance through redundancy. In *USENIX Annual Technical Conference*, pages 463–474, 2014.

[61] Xiang Song, Jian Yang, and Haibo Chen. Architecting flash-based solid-state drive for high-performance i/o virtualization. *IEEE Computer Architecture Letters*, 13(2):61–64, 2014.

[62] Yong Ho Song, Sanghyuk Jung, Sang-Won Lee, and Jin-Soo Kim. Cosmos openssd: A pcie-based open source ssd platform. *Proc. Flash Memory Summit*, 2014.

[63] Hung-Wei Tseng, Laura Grupp, and Steven Swanson. Understanding the impact of power loss on flash memory. In *Proceedings of the 48th Design Automation Conference*, pages 35–40. ACM, 2011.

[64] Hua Wang, Ping Huang, Shuang He, Ke Zhou, Chunhua Li, and Xubin He. A novel i/o scheduler for ssd with improved performance and lifetime. In *Mass Storage Systems and Technologies (MSST), 2013 IEEE 29th Symposium on*, pages 1–5. IEEE, 2013.

[65] Guanying Wu and Xubin He. Reducing ssd read latency via nand flash program and erase suspension. In *FAST*, volume 12, pages 10–10, 2012.

[66] Michael Wu and Willy Zwaenepoel. envy: a non-volatile, main memory storage system. In *ACM SIGOPS Operating Systems Review*, volume 28, pages 86–97. ACM, 1994.

[67] Shiqin Yan, Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Andrew A Chien, and Haryadi S Gunawi. Tinytail flash: Near-perfect elimination of garbage collection tail latencies in nand ssds. In *FAST*, pages 15–28, 2017.

[68] Ming-Chang Yang, Yuan-Hao Chang, Tei-Wei Kuo, and Fu-Hsin Chen. Reducing data migration overheads of flash wear leveling in a progressive way. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 24(5):1808–1820, 2016.

[69] Jin Hyuk Yoon, Eyee Hyun Nam, Yoon Jae Seong, Hongseok Kim, Bryan Kim, Sang Lyul Min, and Yookun Cho. Chameleon: A high performance flash/fram hybrid solid state disk architecture. *IEEE computer architecture letters*, 7(1):17–20, 2008.

[70] Qi Zhang, Xuandong Li, Linzhang Wang, Tian Zhang, Yi Wang, and Zili Shao. Optimizing deterministic garbage collection in nand flash storage systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2015 IEEE*, pages 14–23. IEEE, 2015.

[71] Mai Zheng, Joseph Tucek, Feng Qin, and Mark Lillibridge. Understanding the robustness of ssds under power fault. In *FAST*, pages 271–284, 2013.

초 록

김 석 준
컴퓨터 공학부
서울대학교 대학원

소형 휴대 장치부터 대규모 저장 시스템까지 플래시 메모리 기반 저장 시스템은 빠른 응답시간과 높은 병렬성 덕에 최근 몇 년 동안 많이 인기를 얻었다. 그러나 이러한 장점에도 불구하고 플래시 메모리 저장 장치에서 예측 가능한 성능을 달성하는 것은 어렵다. 사용자와 대규모 시스템에서의 경험은 플래시 저장 장치의 성능이 시간이 지남에 따라 저하 될뿐 아니라 상당한 변동과 불안정성을 나타냄을 보여준다. 이러한 성능 예측 불가능성은 플래시 변환 계층 (FTL)에서 서로 경쟁하는 작업들의 조율되지 않은 자원 사용으로 인해 야기되며, 플래시 메모리의 한계를 숨기기 위해 더 많은 FTL 작업들이 추가됨에 따라 성능을 보장하는 것이 점점 더 어려워 질 것이다.

본 논문에서는 자율적으로 FTL 작업들을 관리하는 SSD 아키텍쳐를 제안하는데, 이는 자원의 가상화를 통해 FTL 작업들의 개발을 용이하게 하고, 자원 중재를 효과적으로 하는 스케쥴링 메커니즘과, 성능 목표를 달성하기 위해 FTL 작업들을 조정하는 유연한 스케쥴링 정책을 포함한다. 이 아키텍처에서 각 FTL 작업들은 가상화를 통해 자체적으로 플래시 메모리 서브시스템이 있는 환상을 얻게 되고, 이 자원 가상화는 각 FTL 작업들이 서로 독립적으로 구현되게 할 뿐 아니라, 향후 새 작업의 통합도

쉽게 할 수 있게 한다. 플래시 메모리 서브 시스템의 자원들은 share에 의해 중재되며, 작업의 share는 각 작업이 수행 할 수 있는 일의 양을 나타낸다. Share 기반의 간단한 스케쥴링 메커니즘은 현재 수행되고 있는 요청의 수를 제한함으로서 효과적으로 스케쥴링 한다. Share는 주요 시스템 상태에 따라 feedback 제어를 통해 동적으로 조정되어 여러 FTL 작업들의 진행을 조율한다.

자율적으로 FTL 작업들을 관리하는 SSD 아키텍처의 효율성은 쓰레기 수집, 사상정보 관리, 읽기 재생등 다수의 FTL 작업들이 구현함으로서 입증하였다. 본 논문에서 제안하는 SSD 아키텍처는 다양한 워크로드에서 평균 응답시간을 16.2% 줄이고 평균 six nines QoS를 67.8% 줄여 안정된 성능을 보인다.

주요어 : SSD, QoS, 스케쥴링, 제어

학번 : 2014-31116