AutoSSD: an Autonomic SSD Architecture

Bryan S. Kim, ***Seoul National University;*** **Hyun Suk Yang,** ***Hongik University;***
Sang Lyul Min, ***Seoul National University***

https://www.usenix.org/conference/atc18/presentation/kim

This paper is included in the Proceedings of the
2018 USENIX Annual Technical Conference (USENIX ATC '18).

July 11–13, 2018 • Boston, MA, USA

ISBN 978-1-939133-02-1

Open access to the Proceedings of the
2018 USENIX Annual Technical Conference
is sponsored by USENIX.

AutoSSD: an Autonomic SSD Architecture

Bryan S.Kim
Seoul National University

Hyun Suk Yang
Hongik University

Sang Lyul Min
Seoul National University

Abstract

From small mobile devices to large-scale storage arrays, flash memory-based storage systems have gained a lot of popularity in recent years. However, the uncoordinated use of resources by competing tasks in the flash translation layer (FTL) makes it difficult to guarantee predictable performance.

In this paper, we present *AutoSSD*, an autonomic SSD architecture that self-manages FTL tasks to maintain a high-level of QoS performance. In AutoSSD, each FTL task is given an illusion of a dedicated flash memory subsystem, allowing tasks to be implemented oblivious to others and making 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 AutoSSD by holistically considering multiple facets of SSD internal management, and by evaluating it across diverse workloads. Compared to state-of-the-art techniques, our design reduces the average response time by up to 18.0%, the 3 nines (99.9%) QoS by up to 67.2%, and the 6 nines (99.9999%) QoS by up to 76.6% for QoS-sensitive small reads.

1 Introduction

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 [5] and 3D stacking [38] 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*.

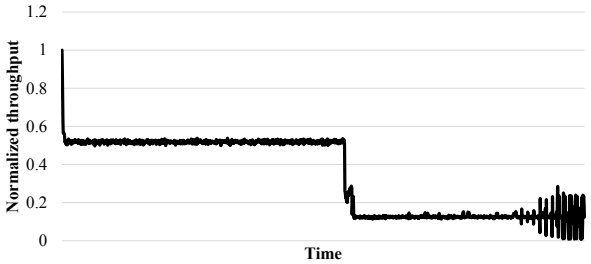


Figure 1: Performance drop and variation under 4KB random writes.

Large-scale deployments and user experiences, however, reveal that despite its low latency and massive parallelism, flash storage exhibits high performance instabilities and variations [9, 17]. Garbage collection (GC) has been pointed out as the main source of the problem [9, 25, 28, 29, 45], 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 [2,23]. 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 the 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 internal management task. Map caching [15] 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 [16] that preventively migrates data before its corruption also creates traffic when blocks are repeatedly read, but failure to perform its duty on time can lead to data loss. As more tasks with unique responsibilities are added to the system, it becomes increasingly difficult to design a system that meets its performance and reliability requirements [13].

In this paper, we present an autonomic SSD architecture called AutoSSD that self-manages its management tasks to maintain a high-level of QoS performance. In our design, each task is given a virtualized view of the flash memory subsystem by hiding the details of flash memory request scheduling. 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. We show that for small read requests, AutoSSD reduces the average response time by up to 18.0%, the 3 nines (99.9%) QoS by up to 67.2%, and the 6 nines (99.9999%) QoS by up to 76.6% compared to state-of-the-art techniques. Our contributions are as follows:

- We present AutoSSD, an autonomic SSD architecture that dynamically manages internal housekeeping tasks to maintain a stable system state. (§ 3)
- We holistically consider multiple facets of SSD internal management, including not only garbage collection and host request handling, but also mapping management and read scrubbing. (§ 4)
- We evaluate our design and compare it to the state-of-the-art techniques across diverse workloads, analyze causes for long tail latencies, and demonstrate the advantages of dynamic management. (§ 5)

The remainder of this paper is organized as follows. § 2 gives a background on understanding why flash storages exhibit performance unpredictability. § 3 presents the overall architecture of AutoSSD and explains our design choices. § 4 describes the evaluation methodology and the SSD model that implements various FTL tasks, and § 5 presents the experimental results under both synthetic and real I/O workloads. § 6 discusses our design in relation to prior work, and finally § 7 concludes.

2 Background

For flash memory to be used as storage, several of its limitations need to be addressed. 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 valid data to another location for reclaiming space. These internal management schemes, collectively known as the flash translation layer (FTL) [11], hide the limitations of flash memory and provide an illusion of a traditional block storage interface.

The role of the FTL has become increasingly important as 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 [12, 41, 44], and the electrons in the floating gate that represent data may gradually leak over time [6, 35, 44]. Sudden power loss can increase bit error rates beyond the error correction capabilities [44, 47], and error rates increase as flash memory blocks wear out [6, 12, 19]. As flash memory becomes less reliable in favor of high-density [13], more sophisticated FTL algorithms are needed to complement existing reliability enhancement techniques.

Even though modern flash storages are equipped with sophisticated FTLs and powerful controllers, meeting performance requirements have three main challenges. First, 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. Second, 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. Lastly, 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.

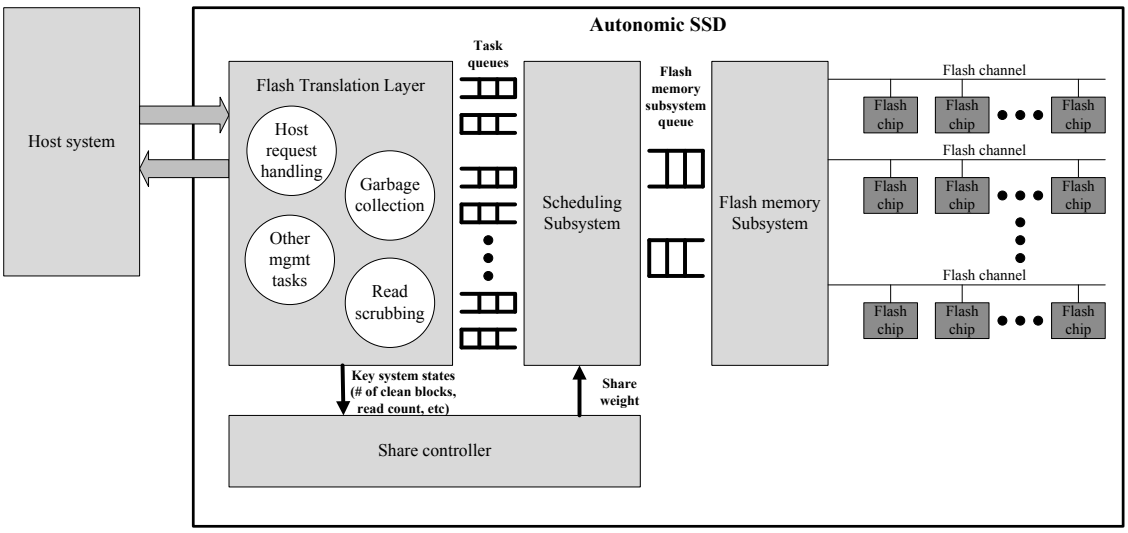


Figure 2: Overall architecture of AutoSSD and its components.

3 AutoSSD Architecture

In this section, we describe the overall architecture and design of the autonomic SSD (AutoSSD) as shown in Figure 3. 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 subsections focus on the overall architecture and design that enable the self-management of the flash storage.

3.1 Virtualization of the Flash Memory Subsystem

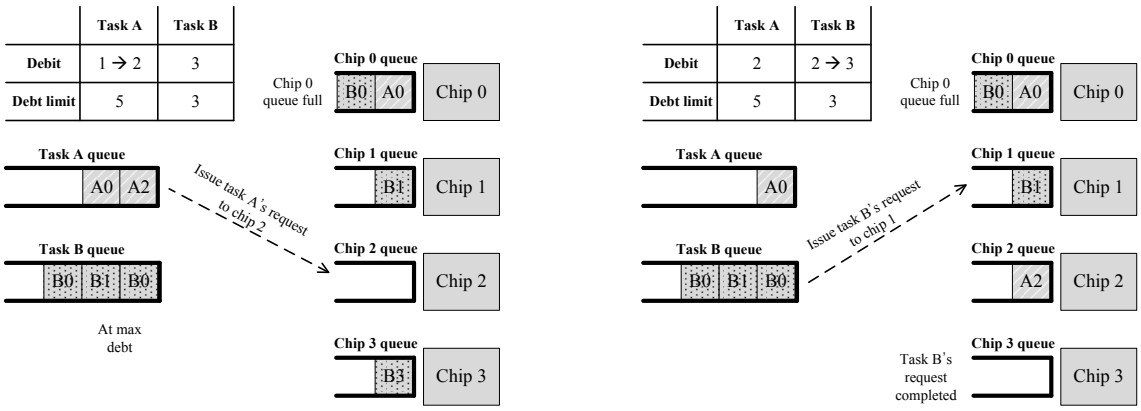
The architecture of AutoSSD allows each task to be independent of others by virtualizing the flash memory subsystem. Each FTL task is given a pair of request and response queues to send and receive flash memory requests and responses, respectively. This interface provides an illusion of a dedicated (yet slower) flash memory subsystem and allows an FTL task to generate flash memory requests oblivious of others (whether idle or active) 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 writes, read scrubbing preventively relocates data to ensure data integrity, and wear leveling swaps contents of data to even out the damage done on flash memory cells. Our design allows seamless integration of new tasks without having to modify existing ones and reoptimize the system.

3.2 Scheduling for Share Enforcement

The scheduling subsystem interfaces with each FTL task and arbitrates the resources of the flash memory subsystem. The scheduler needs to be efficient with low overhead as it manages concurrency (tens and hundreds of flash memory requests) and parallelism (tens and hundreds of flash memory chips) at a small timescale.

In AutoSSD, we consider these unique domain characteristics and arbitrate the flash memory subsystem resource through *debit scheduling*. The debit scheduler tracks and limits the number of outstanding requests per task across all resources, and is based on the request windowing technique [14, 20, 34] from the disk scheduling domain. 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 that 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



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

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

Figure 3: Two debit scheduling examples. In the scenario of Figure 3a, 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 3b, 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.

parallelism, and is set to the total number of requests that can be queued in the flash memory controller.

Figure 3 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 3a, 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 3b, 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 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 [26] and weighted fair queueing (WFQ) [10], 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 [36].

3.3 Feedback Control of Share

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 monitors the system states and adjusts shares to control the rate of progress for individual FTL tasks, so that the system is maintained within stable levels.

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 tasks must not be set too high such that they severely degrade the 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:

$$S_A[t] = P_A \cdot e_A[t] + I_A \cdot S_A[t-1] \quad (1)$$

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 defined as follows:

$$e_{GC}[t] = \max(0, target_{freeblk} - num_{freeblk}[t]) \quad (2)$$

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:

$$e_{RS}[t] = \max(0, \max_{i \in blk}(readcnt_i[t]) - target_{readcnt}) \quad (3)$$

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 our design, 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.

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, non-linear to the share under real-world workloads, and computationally solving for the optimal share involves large overhead, if not impossible. As a result, the two coefficients P and I for FTL tasks are empirically hand-tuned in this work.

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 Evaluation Methodology and Modeling

We model a flash storage system on top of the DiskSim environment [1] by enhancing its SSD extension [3]. In this section, we describe the various components and configuration of the SSD, and the workload and test settings used for the evaluation.

4.1 Flash Memory Subsystem

Flash memory controller is based on Ozone [36] that fully utilizes flash memory subsystem's channel and chip parallelism. There can be at most four requests queued to each chip in the controller. Increasing this 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%.

4.2 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.

Mapping. We implement an FTL with map caching [15] 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

Table 2: Trace workload characteristics.

Workload	Duration (hrs)	Number of I/Os (Millions)		Average request size(KB)		Inter-arrival time (ms)	
		Write	Read	Write	Read	Average	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

data is always kept in memory as it is accessed more frequently and orders of magnitude smaller than the first-level mapping table.

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.

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 [40]. 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 [40]. 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.

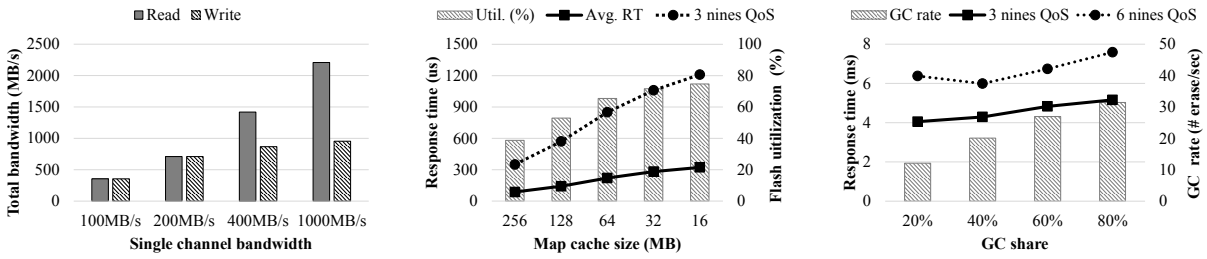
Read scrubbing. The read scrubbing (RS) task also runs as its own stand-alone task. Victims are selected greedily based on the read count of a block: the block with the most number of reads is chosen. Other than that, the process of copying valid data is identical to that of the garbage collection task. 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.

4.3 Workload and Test Settings

We use both synthetic workloads and real-world I/O traces from Microsoft production servers [27] to evaluate the autonomic SSD architecture. Synthetic workloads of 128KB sequential accesses, 4KB random reads, and 4KB random read/writes are used to verify that our model behaves expectedly according to the system parameters.

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 2.

Prior to each experiment, 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 [2]. 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.



(a) 128KB sequential accesses. (b) 4KB random reads. (c) 4KB random reads/writes.

Figure 4: Performance under synthetic workloads. Figure 4a shows the total bandwidth under 128KB sequential accesses with respect to changes in the channel bandwidth. Figure 4b shows the performance (average response time and 3 nines QoS) and the utilization of the flash memory subsystem with respect to changes in the size of the in-memory map cache. Figure 4c shows the performance (3 nines and 6 nines QoS) and the GC progress rate with respect to the GC share.

5 Experiment Results

This section presents experimental results under the configuration and workload settings described in the previous section. The main performance metric we report is the system response time seen at the I/O device driver. We first validate our SSD model using synthetic workloads, and then present experimental results with I/O traces. We replayed the I/O traces with the original request dispatch times, and with the dispatch times scaled down to increase the workload intensity.

5.1 Micro-benchmark Results

Figure 4 illustrates the performance of the autonomic SSD architecture (AutoSSD) with debit scheduling under four micro-benchmarks. Figure 4a plots the total bandwidth under 128KB sequential reads and 128KB sequential writes as we increase the channel bandwidth. As the channel 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 1000MB/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 4b, 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 the 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 probabilistically 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 4c. In this scenario, 4KB random read/write requests are issued at 20K IOPS with a 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 at fixed GC shares from 20% to 80%. As shown by the bar graph, more blocks are erased as the share for GC increases. Furthermore, with more GC share, the overall host performance suffers, as evident by the increase in the 3 nines QoS. Deceptively, however, assigning not enough share to GC will result in worse tail latency as shown by the 6 nines QoS. GC needs to produce sufficient number of free blocks for the host to consume, and failure to do so will cause the host to block.

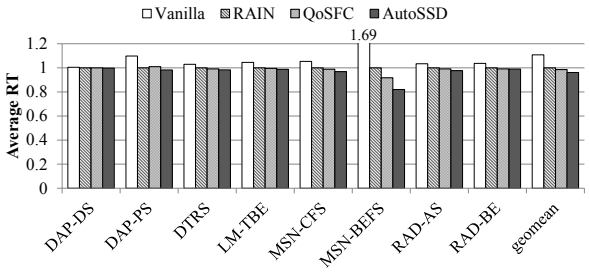
5.2 I/O Trace Results

Using I/O traces, we evaluate the performance of AutoSSD and compare it to following three systems:

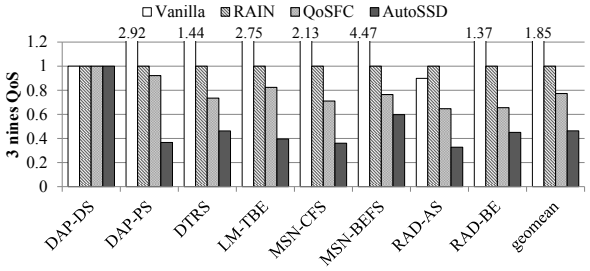
Vanilla represents a design without virtualization and coordination, and all tasks dispatch requests to the controller through a single pair of request/response queue.

RAIN [45] uses parity to reconstruct data when accessing the chip is blocked by background tasks. Resources are arbitrated through fixed priority scheduling, with host requests having the highest priority. This technique requires an additional physical capacity to store parity data.

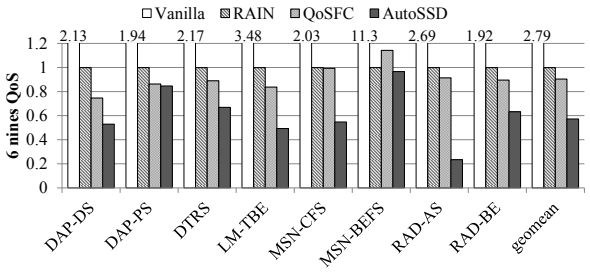
QoSFC [28] schedules using weighted fair queueing (WFQ) and represents a work-conserving system that does not reserve resources. It maintains virtual time as a measure of progress for each FTL task at each flash memory resource.



(a) Average response time.



(b) Three nines QoS.



(c) Six nines QoS.

Figure 5: Comparison of Vanilla, RAIN, QoSFC, and AutoSSD under eight different traces. Results are normalized to the performance of RAIN. AutoSSD reduces the average response time by up to 18.0% under `MSN-BEFS` (by 3.8% on average), the 3 nines QoS by up to 67.2% under `RAD-AS` (by 53.6% on average). and the 6 nines QoS by up to 76.6% under `RAD-AS` (by 42.7% on average).

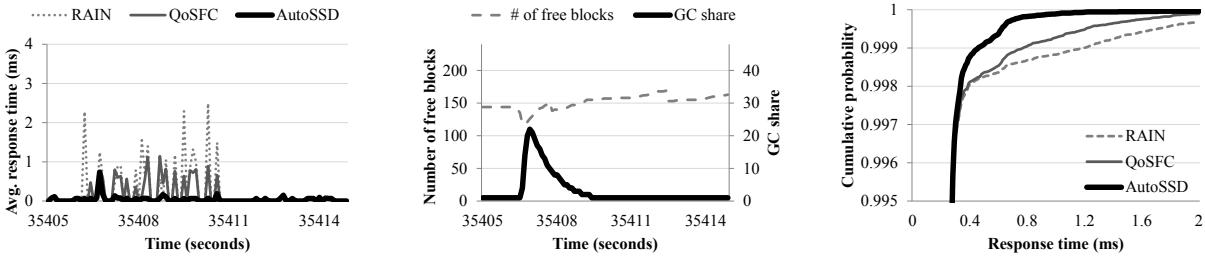
As the focus of this paper is response time characteristics, we only measure the performance of QoS-sensitive small reads (no larger than 64KB) in terms of the average response time, the 3 nines (99.9%) QoS figure, and the 6 nines (99.9999%).

Figure 5 compares the performance of the four systems under eight different traces. Compared to RAIN, AutoSSD reduces the average response time by up to 18.0% under `MSN-BEFS` as shown in Figure 5a. For the 3 nines QoS, AutoSSD shows improvements across most workloads, reducing it by 53.6% on average and as much as by 67.2% under `RAD-AS` (see Figure 5b). For the 6 nines QoS, AutoSSD shows much greater improvements, reducing it as much as by 76.6% under `RAD-AS` (see Figure 5c). Without coordination among FTL tasks, the Vanilla performance suffers, especially for the long tail latencies. In terms of the 6 nines, AutoSSD performs well under bursty workloads such as `RAD-AS` and `LM-TBE` (large difference between average and median inter-arrival times in Table 2). This is because AutoSSD limits the progress of internal FTL tasks depending on the state of the system, making resources available for the host in a non-work-conserving manner. This is in contrast to the scheduling disciplines used by the other systems: Vanilla uses FIFO scheduling; RAIN, priority scheduling; and QoSFC, weighted fair queueing.

To better understand the overall results in Figure 5, we microscopically examine the performance under `RAD-AS` in Figure 6 and `LM-TBE` in Figure 7. Figure 6a shows the average response time of three systems—RAIN, QoSFC, and AutoSSD—during a 10-second window, approximately 10 hours into `RAD-AS`. GC is active during this window for all the three systems, and both RAIN and QoSFC exhibit large spikes in response time. On the other hand, AutoSSD is better able to bound the performance degradation caused by an active garbage collection. Figure 6b shows the number of free blocks and the GC share during that window 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. In effect, AutoSSD improves the overall response time as shown in Figure 6c.

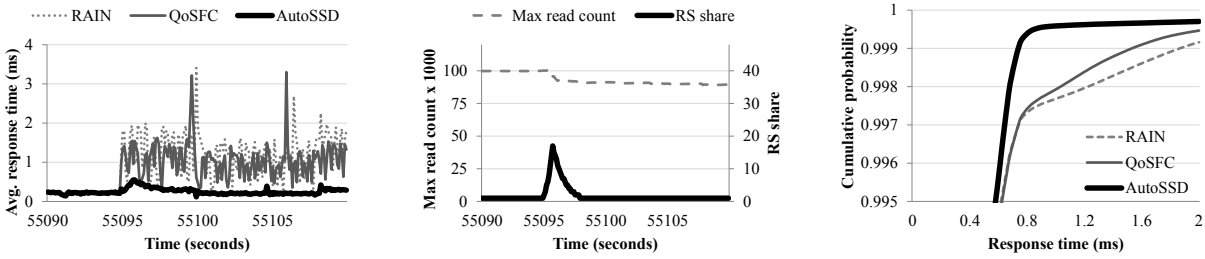
For `LM-TBE`, Figure 7a shows the average response time of the three systems during a 20-second window, approximately 15 hours into the workload. Here we observe read scrubbing (RS) becoming active due to the read-dominant characteristics of `LM-TBE`. We observe that both RAIN and QoSFC show large spikes in response time that lasts longer than the perturbation caused by GC for `RAD-AS` (cf. Figure 6a). While GC is incentivized to select a block with less valid data, RS is likely to pick a block with a lot of valid data that are frequently read but not frequently updated: this causes the performance degradation induced by RS to last longer than that by GC. AutoSSD limits this effect, while still decreasing the maximum read count in the system by dynamically adjusting the share of RS, as shown in Figure 7b. Figure 7c shows the response time CDF of the three systems.

Figure 8 illustrates the delay causes for the flash memory requests generated by the host request handling task under `MSN-BEFS`. Note that this is different from the response time of host requests: this shows the average wait time that a flash memory request (for servicing the host) experiences, broken down by different causes. Category



(a) Average response time under RAD-AS. (b) Change in GC share under RAD-AS. (c) Response time CDF under RAD-AS.

Figure 6: Comparison of RAIN, QoSFC, and AutoSSD under RAD-AS. Figure 6a shows the average response time sampled at 100ms in the selected 10-second window. Figure 6b shows the number of free blocks and the GC share of AutoSSD for the same 10-second window. Figure 6c plots the response time CDF for the entire duration.



(a) Average response time under LM-TBE. (b) Change in RS share under LM-TBE. (c) Response time CDF under LM-TBE.

Figure 7: Comparison of RAIN, QoSFC, and AutoSSD under LM-TBE. Figure 7a shows the average response time sampled at 100ms in the selected 20-second window. Figure 7b shows the maximum read count and the RS share of AutoSSD for the same 20-second window. Figure 7c plots the response time CDF for the entire duration.

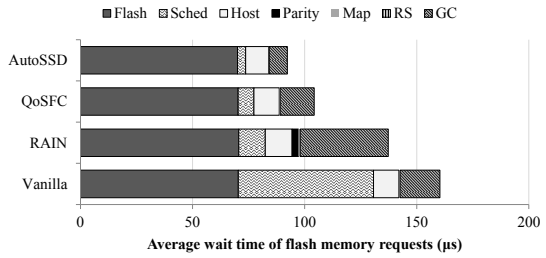


Figure 8: Breakdown of wait time experienced by flash memory requests under MSN-BEFS.

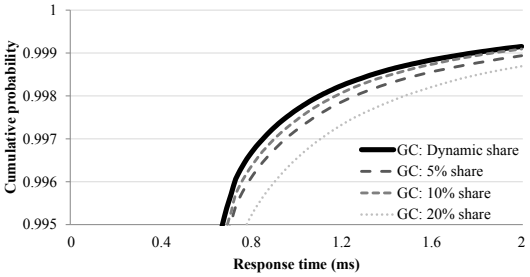


Figure 9: Comparison of AutoSSD with static shares and dynamic share under MSN-BEFS.

Flash represents flash memory latency, combining both flash array access latency and data transfers. Sched is the time spent waiting to be scheduled, either waiting in the queue because the target queue is full, or waiting because the scheduler limits the progress in a non-work-conserving manner (the case for AutoSSD). The large Sched wait time for Vanilla is caused by uncoordinated sharing of resources, while that for AutoSSD is small as the scheduler reserves resources for host requests. The remaining five categories are delays experienced due to resource blocking. Most noticeably, the wait time caused by GC in RAIN is higher than the other systems. When RAIN generates alternate flash memory requests to reconstruct data through parity, these additional requests can, in turn, be blocked again at another resource. In ttFlash [45], this problem is overcome by statically limiting the number of active GC to one per parity group. This technique is not used in our evaluation as a fixed cap on the number of allowed GC can quickly deplete free blocks, especially for high-intensity small random write workloads.

Next, we examine the effectiveness of the dynamic share assignment over the static ones. Figure 9 shows the response time CDF of AutoSSD under MSN-BEFS 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 observe that the curve for 5% dwindles as it approaches higher QoS and

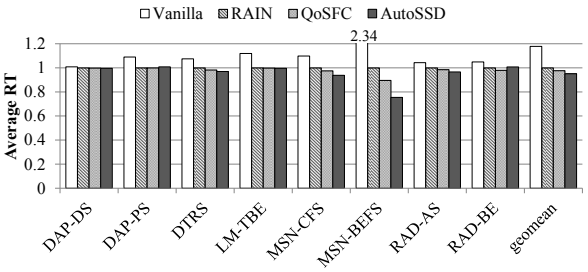
performs worse than the 10% curve. This indicates that while a lower GC share achieves better performance at lower QoS levels, a higher GC share is desirable to reduce long-tail latencies as it generates free blocks at a higher rate, preventing the number of free blocks from becoming critically low. This observation is in accordance with the performance under synthetic workload in Figure 4c. 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.

5.3 I/O Trace Results at Higher Intensity

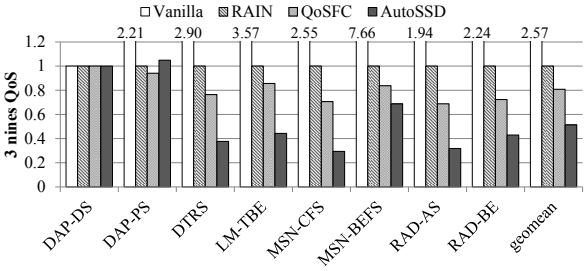
In this subsection, we present experimental results with higher request intensities. Here, the request dispatch times are reduced in half, but other parameters such as the access type and the target address remain unchanged. This experiment is intended to examine the performance of the four systems—Vanilla, RAIN, QoSFC, and AutoSSD—under a more stressful scenario.

Figure 10 compares the performance in the new setting. AutoSSD reduces the average response time by up to 24.6% under MSN-BEFS (see Figure 10a), the 3 nines QoS figure by 48.6% on average and as much as 70.6% under MSN-CFS (see Figure 10b), and the 6 nines QoS figure by as much as 55.3% under MSN-CFS (see Figure 10c). With workload intensity increased, the overall improvement in long tail latency decreases due to a smaller wiggle room for AutoSSD to manage FTL tasks. This is especially true for high-intensity workloads such as MSN-BEFS: with host requests arriving back-to-back (cf. halve the inter-arrival time in Table 2), debit scheduling has little advantage over other scheduling schemes. However, AutoSSD nevertheless outperforms prior techniques across the diverse set of workloads. Workloads such as RAD-AS and LM-TBE that showed the most reduction in long tail latency under the original intensity (cf. Figure 5c) still exhibit performance improvements with AutoSSD in the 6 nines, even with increased workload intensity.

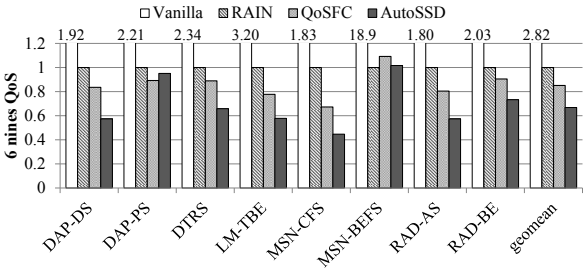
We examine DTRS more closely in Figure 11. Figure 11a shows the average response time of the three systems—RAIN, QoSFC, and AutoSSD—during a 20-second window, approximately 2 hours into the workload. GC is active during this window for all the three systems, and AutoSSD is better able to bound the performance degradation caused by an active garbage collection, while both RAIN and QoSFC exhibit large spikes in response time. Figure 11b shows the number of free blocks and the GC share during that window for AutoSSD. Similar to the results in the previous section, the share for GC reactively increases at a lower number of blocks, and decays once the number of free blocks



(a) Average response time at 2x intensity.



(b) Three nines QoS at 2x intensity.



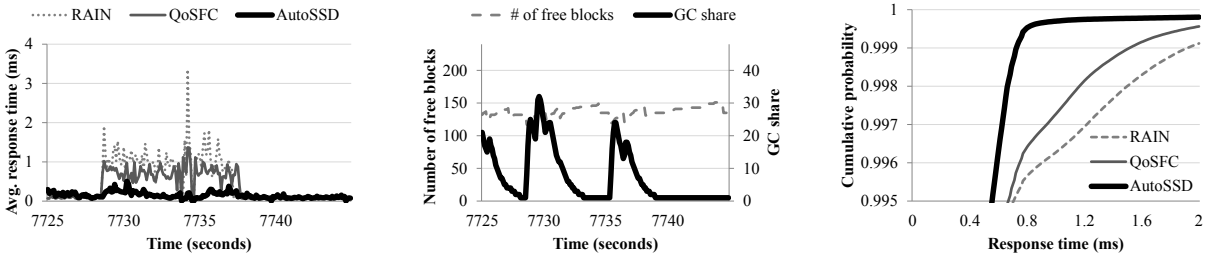
(c) Six nines QoS at 2x intensity.

Figure 10: Comparison of Vanilla, RAIN, QoSFC, and AutoSSD under eight different traces at 2x intensity. Results are normalized to the performance of RAIN at 2x intensity. AutoSSD reduces the average response time by up to 24.6% under MSN-BEFS (by 4.9% on average), the 3 nines QoS by up to 70.6% under MSN-CFS (by 48.6% on average). and the 6 nines QoS by up to 55.3% under MSN-CFS (by 33.2% on average).

reaches a stable region. Again, the number of free blocks shows a sawtooth behavior, and the ridges of GC share curve matches the valleys of the free block curve. Figure 11c plots the response time CDF of the three systems, demonstrating the effectiveness of our dynamic management.

6 Discussion and Related Work

There are several studies on real-time performance guarantees of flash storage, but they depend on RTOS support [7], specific mapping schemes [8, 39, 46], a number of reserve blocks [8, 39], and flash operation latencies [46]. These tight couplings make it difficult to extend performance guarantees when system requirements



(a) Average response time under 2xDTRS. (b) Change in GC share under 2xDTRS. (c) Response time CDF under 2xDTRS.

Figure 11: Comparison of the RAIN, QoSFC, and AutoSSD under DTRS running at 2x intensity. Figure 11a shows the average response time sampled at 100ms in the selected 20-second window. Figure 11b shows the number of free blocks and the GC share of AutoSSD for the same 20-second window. Figure 11c plots the response time CDF for the entire duration.

and flash memory technology change. On the other hand, our architecture is FTL implementation-agnostic, allowing it to be used across a wide range of flash devices and applications.

Some techniques focus on when to perform GC (based on threshold [31], slack [22], or host idleness [25, 37]). These approaches complement our design that focuses on the fine-grained scheduling and dynamic management of multiple FTL tasks running concurrently. By incorporating workload prediction techniques to our design, we can extend AutoSSD to increase the share on background tasks when host idleness is expected, and decrease it when host requests are anticipated.

Exploiting redundancy to reduce performance variation has been studied in a number of prior art. Harmonia [30] and Storage engine [42] duplicate data across multiples SSDs, placing one in read mode and the other in write mode to eliminate GC's impact on read performance. ttFlash [45] uses multiple flash memory chips to reconstruct data through a RAID-like parity scheme. Relying on redundancy effectively reduces the storage utilization, but otherwise complements our design of dynamic management of various FTL tasks.

Performance isolation aims to reduce performance variation caused by multiple hosts through partitioning resources (vFlash [43], FlashBlox [18]), improving GC efficiency by grouping data from the same source (Multi-streamed [24], OPS isolation [29]), and penalizing noisy neighbors (WA-BC [21]). These performance isolation techniques are complementary to our approach of fine-grained scheduling and dynamic management of concurrent FTL tasks.

The design of the autonomic SSD architecture borrows ideas from prior work on shared disk-based storage systems such as Façade [33], PARDA [14], and Maestro [34]. These systems aim to meet performance requirements of multiple clients by throttling request rates 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 the foreground (host I/O) and the background work (garbage collection and other management schemes).

Aqueduct [32] and Duet [4] 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 depends on the background task, necessitating a different approach.

7 Conclusion

In this paper, 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, AutoSSD reduces the average response time by up to 18.0%, the 3 nines (99.9%) QoS by up to 67.2%, and the 6 nines (99.9999%) QoS by up to 76.6% for QoS-sensitive small reads.

Acknowledgment

We thank the anonymous reviewers for their constructive and insightful comments, and also thank Jaejin Lee, Jongmoo Choi, Hyeonsang Eom, and Eunji Lee for reviewing the early stages of this work. This work was supported in part by SK Hynix and the National Research Foundation of Korea under the PF Class Heterogeneous High Performance Computer Development (NRF-2016M3C4A7952587). Institute of Computer Technology at Seoul National University provided the research facilities for this study.

References

[1] The disksim simulation environment version 4.0 reference manual (cmu-pdl-08-101). `http://www.pdl.cmu.edu/PDL-FTP/DriveChar/CMU-PDL-08-101.pdf`, 2008. Parallel Data Laboratory.

[2] Solid state storage (SSS) performance test specification (PTS) enterprise. `http://snia.org/sites/default/files/SSS_PTS_Enterprise_v1.1.pdf`, 2013. Storage Networking Industry Association.

[3] AGRAWAL, N., PRABHAKARAN, V., WOBBER, T., DAVIS, J. D., MANASSE, M. S., AND PANIGRAHY, R. Design tradeoffs for SSD performance. In *USENIX Annual Technical Conference* (2008), pp. 57–70.

[4] AMVROSIADIS, G., BROWN, A. D., AND GOEL, A. Opportunistic storage maintenance. In *ACM Symposium on Operating Systems Principles* (2015), pp. 457–473.

[5] ARITOME, S. *NAND flash memory technologies*. John Wiley & Sons, 2015.

[6] CAI, Y., HARATSCH, E. F., MUTLU, O., AND MAI, K. Error patterns in MLC NAND flash memory: Measurement, characterization, and analysis. In *Design, Automation and Test in Europe* (2012), pp. 521–526.

[7] CHANG, L.-P., KUO, T.-W., AND LO, S.-W. Real-time garbage collection for flash-memory storage systems of real-time embedded systems. *ACM Transactions on Embedded Computing Systems 3*, 4 (2004), 837–863.

[8] CHOUDHURI, S., AND GIVARGIS, T. Deterministic service guarantees for NAND flash using partial block cleaning. In *IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis* (2008), pp. 19–24.

[9] DEAN, J., AND BARROSO, L. A. The tail at scale. *Communications of the ACM 56*, 2 (2013), 74–80.

[10] DEMERS, A., KESHAV, S., AND SHENKER, S. Analysis and simulation of a fair queueing algorithm. In *ACM SIGCOMM* (1989), pp. 1–12.

[11] GAL, E., AND TOLEDO, S. Algorithms and data structures for flash memories. *ACM Computing Surveys 37*, 2 (2005), 138–163.

[12] GRUPP, L. M., CAULFIELD, A. M., COBURN, J., SWANSON, S., YAAKOBI, E., SIEGEL, P. H., AND WOLF, J. K. Characterizing flash memory: anomalies, observations, and applications. In *IEEE/ACM International Symposium on Microarchitecture* (2009), pp. 24–33.

[13] GRUPP, L. M., DAVIS, J. D., AND SWANSON, S. The bleak future of NAND flash memory. In *USENIX Conference on File and Storage Technologies* (2012), pp. 17–24.

[14] GULATI, A., AHMAD, I., WALDSPURGER, C. A., ET AL. PARDA: Proportional allocation of resources for distributed storage access. In *USENIX Conference on File and Storage Technologies* (2009), pp. 85–98.

[15] GUPTA, A., KIM, Y., AND URGAONKAR, B. DFTL: a flash translation layer employing demand-based selective caching of page-level address mappings. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (2009), pp. 229–240.

[16] HA, K., JEONG, J., AND KIM, J. 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 (2016), 1079–1091.

[17] HAO, M., SOUNDARARAJAN, G., KENCHAMMANA-HOSEKOTE, D. R., CHIEN, A. A., AND GUNAWI, H. S. The tail at store: A revelation from millions of hours of disk and SSD deployments. In *USENIX Conference on File and Storage Technologies* (2016), pp. 263–276.

[18] HUANG, J., BADAM, A., CAULFIELD, L., NATH, S., SENGUPTA, S., SHARMA, B., AND QURESHI, M. K. FlashBlox: Achieving both performance isolation and uniform lifetime for virtualized SSDs. In *USENIX Conference on File and Storage Technologies* (2017), pp. 375–390.

[19] JIMENEZ, X., NOVO, D., AND IENNE, P. Wear unleveling: improving NAND flash lifetime by balancing page endurance. In *USENIX Conference on File and Storage Technologies* (2014), pp. 47–59.

[20] JIN, W., CHASE, J. S., AND KAUR, J. Interposed proportional sharing for a storage service utility. In *ACM SIGMETRICS* (2004), pp. 37–48.

[21] JUN, B., AND SHIN, D. Workload-aware budget compensation scheduling for NVMe solid state drives. In *IEEE Non-Volatile Memory System and Applications Symposium* (2015), pp. 1–6.

[22] JUNG, M., CHOI, W., SRIKANTAIAH, S., YOO, J., AND KANDEMIR, M. T. HIOS: A host interface I/O scheduler for solid state disks. In *ACM International Symposium on Computer Architecture* (2014), pp. 289–300.

[23] JUNG, M., AND KANDEMIR, M. Revisiting widely held SSD expectations and rethinking system-level implications. In *ACM International Conference on Measurement and Modeling of Computer Systems* (2013), pp. 203–216.

[24] KANG, J.-U., HYUN, J., MAENG, H., AND CHO, S. The multi-streamed solid-state drive. In *USENIX Workshop on Hot Topics in Storage and File Systems* (2014).

[25] KANG, W., SHIN, D., AND YOO, S. Reinforcement learning-assisted garbage collection to mitigate long-tail latency in SSD. *ACM Transactions on Embedded Computing Systems 16*, 5s (2017), 134.

[26] KATEVENIS, M., SIDIROPOULOS, S., AND COURCOUBETIS, C. Weighted round-robin cell multiplexing in a general-purpose ATM switch chip. *IEEE Journal on selected Areas in Communications 9*, 8 (1991), 1265–1279.

[27] KAVALANEKAR, S., WORTHINGTON, B., ZHANG, Q., AND SHARDA, V. Characterization of storage workload traces from production Windows servers. In *IEEE International Symposium on Workload Characterization* (2008), pp. 119–128.

[28] KIM, B. S., AND MIN, S. L. QoS-aware flash memory controller. In *IEEE Real-Time and Embedded Technology and Applications Symposium* (2017), pp. 51–62.

[29] KIM, J., LEE, D., AND NOH, S. H. Towards SLO complying SSDs through OPS isolation. In *USENIX Conference on File and Storage Technologies* (2015), pp. 183–189.

[30] KIM, Y., ORAL, S., SHIPMAN, G. M., LEE, J., DILLOW, D. A., AND WANG, F. Harmonia: A globally coordinated garbage collector for arrays of solid-state drives. In *IEEE Symposium on Mass Storage Systems and Technologies* (2011), pp. 1–12.

[31] LEE, J., KIM, Y., SHIPMAN, G. M., ORAL, S., WANG, F., AND KIM, J. A semi-preemptive garbage collector for solid state drives. In *IEEE International Symposium on Performance Analysis of Systems and Software* (2011), pp. 12–21.

[32] LU, C., ALVAREZ, G. A., AND WILKES, J. Aqueduct: Online data migration with performance guarantees. In *USENIX Conference on File and Storage Technologies* (2002), pp. 219–230.

[33] LUMB, C. R., MERCHANT, A., AND ALVAREZ, G. A. Façade: Virtual storage devices with performance guarantees. In *USENIX Conference on File and Storage Technologies* (2003), pp. 131–144.

[34] MERCHANT, A., UYSAL, M., PADALA, P., ZHU, X., SINGHAL, S., AND SHIN, K. Maestro: quality-of-service in large disk arrays. In *ACM International Conference on Autonomic Computing* (2011), pp. 245–254.

[35] MEZA, J., WU, Q., KUMAR, S., AND MUTLU, O. A large-scale study of flash memory failures in the field. In *ACM SIGMETRICS* (2015), pp. 177–190.

[36] NAM, E. H., KIM, B. S. J., EOM, H., AND MIN, S. L. Ozone (O3): An out-of-order flash memory controller architecture. *IEEE Transactions on Computers 60*, 5 (2011), 653–666.

[37] PARK, S.-H., KIM, D.-G., BANG, K., LEE, H.-J., YOO, S., AND CHUNG, E.-Y. An adaptive idle-time exploiting method for low latency NAND flash-based storage devices. *IEEE Transactions on Computers 63*, 5 (2014), 1085–1096.

[38] PRINCE, B. *Vertical 3D memory technologies*. John Wiley & Sons, 2014.

[39] QIN, Z., WANG, Y., LIU, D., AND SHAO, Z. Real-time flash translation layer for NAND flash memory storage systems. In *IEEE Real-Time and Embedded Technology and Applications Symposium* (2012), pp. 35–44.

[40] ROSENBLUM, M., AND OUSTERHOUT, J. K. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems 10*, 1 (1992), 26–52.

[41] SCHROEDER, B., LAGISETTY, R., AND MERCHANT, A. Flash reliability in production: The expected and the unexpected. In *USENIX Conference on File and Storage Technologies* (2016), pp. 67–80.

[42] SHIN, W., KIM, M., KIM, K., AND YEOM, H. Y. Providing QoS through host controlled flash SSD garbage collection and multiple SSDs. In *International Conference on Big Data and Smart Computing* (2015), pp. 111–117.

[43] SONG, X., YANG, J., AND CHEN, H. Architecting flash-based solid-state drive for high-performance I/O virtualization. *IEEE Computer Architecture Letters 13*, 2 (2014), 61–64.

[44] TSENG, H.-W., GRUPP, L., AND SWANSON, S. Understanding the impact of power loss on flash memory. In *Design Automation Conference* (2011), pp. 35–40.

[45] YAN, S., LI, H., HAO, M., TONG, M. H., SUNDARARAMAN, S., CHIEN, A. A., AND GUNAWI, H. S. Tiny-tail flash: Near-perfect elimination of garbage collection tail latencies in NAND SSDs. In *USENIX Conference on File and Storage Technologies* (2017), pp. 15–28.

[46] ZHANG, Q., LI, X., WANG, L., ZHANG, T., WANG, Y., AND SHAO, Z. Optimizing deterministic garbage collection in NAND flash storage systems. In *IEEE Real-Time and Embedded Technology and Applications Symposium* (2015), pp. 14–23.

[47] ZHENG, M., TUCEK, J., QIN, F., AND LILLIBRIDGE, M. Understanding the robustness of SSDs under power fault. In *USENIX Conference on File and Storage Technologies* (2013), pp. 271–284.