공학박사학위논문

OS I/O Path Optimizations for Flash Solid-State Drives

플래시 SSD를 위한 운영체제 I/O경로 최적화

2017년 2월

서울대학교 대학원

전기·컴퓨터 공학부

신 웅

OS I/O Path Optimizations
for Flash Solid-State Drives

플래시 SSD를 위한
운영체제 I/O경로 최적화

지도교수 염 헌 영

이 논문을 공학박사학위논문으로 제출함

2016 년 12 월

서울대학교 대학원

전기·컴퓨터 공학부

신 웅

신웅의 박사학위논문을 인준함

2016 년 12 월

위 원 장	김 지 홍	(인)
부위원장	염 헌 영	(인)
위 원	유 승 주	(인)
위 원	이 재 욱	(인)
위 원	김 영 재	(인)

Abstract

OS I/O Stack Optimizations for Flash Solid-State Drives

Woong Shin
Department of Electrical Engineering and Computer Science
College of Engineering
The Graduate School
Seoul National University

Flash memory technology, in the form of flash solid-state drives (flash SSDs), is steadily replacing prior storage technology based on the value of affordable microsecond level random access memory with high bandwidth. While the cost per bit is now comparable with HDDs, the superior performance of SSDs are now replacing HDDs from our storage systems. However, due to the high latency variability, flash SSDs are yet to be a strong candidate to replace or complement DRAM based in-memory systems which are commonly seen in modern data centers under latency sensitive applications.

Because of this variance, it can be challenging to meet both the IOPS and latency requirements for these latency sensitive applications. While the latency of flash SSDs are small enough to make software overheads of an I/O request not negligible, latency variance increases the overheads by magnifying the impact of context switches which harms both IOPS and latency capability of an I/O path. Also, the latency variance of flash SSDs is exposed in an uncontrolled fashion towards the applications which harm service level throughput of the data center. Such impact of variance has to be tolerated or controlled within the I/O path.

To this end, this dissertation presents a set of host side OS I/O path optimizations which address the impact of latency variance of flash SSDs with the goal of using

flash SSDs under latency sensitive applications. New I/O path designs based on two distinct approaches, which are 1) exploiting additional resource based on parallelism of multiple CPU cores or SSDs, and 2) exploiting the enhanced interactions between the host and the SSD are proposed to cope with the variance. While prior research has limitations on sacrificing one of either IOPS or latency to address variance, our I/O path designs achieve both IOPS and latency by trading additional resources.

To reduce the software overhead caused by the variance, we implemented optimized AHCI based flash SSD device driver which enhances the IOPS capability of the I/O path by reducing the impact of context switches within the I/O completion path. This device driver achieved 100% IOPS enhancement over the original Linux I/O path. Also, an SSD extension was implemented on an SSD prototype platform which can further reduce the latency of individual I/O requests by overlapping scheduling delays with the actual I/O time. Here, the extension was able to introduce average 7 µs of latency reduction per I/O request without diminishing in system parallelism.

To address the leakage of the latency variance, we developed a key-value storage engine as a flash SSD backend for a Memcached. The negative impact of latency spikes caused by write oriented operations was isolated from foreground read operations by exploiting redundant data copies placed on multiple SSDs. While this read-write separation technique provided moderate impact cutting the tail latency of the key value store to millisecond levels, dramatic reduction of was demonstrated by exploiting SSDs with the capability of controlling internal I/O operations such as garbage collection. The extensions achieved latency under 1 ms at the 99.9999th percentiles from the storage engine level.

Keywords: Storage Stack Optimization, Flash SSD, Operating System, Cross-layer Optimization, High Performance Storage Devices, Data Center, Key-value Stores, QoS
Student Number: 2010-23271

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Without a new storage technology emerging, our computer systems have been fixed to an HDD-based architecture for several decades, but the demands of serving data have been increasing over time. Especially, modern data center workloads which are latency sensitive and large in scale has grown almost impossible for our HDD based architectures to support. There have been discussions about the increasing performance gap between the CPU and DRAM which forms a "memory wall." But with the advent of internet scale data centers, "I/O wall" which comes from the performance gap between DRAM and HDD has been overwhelming us.

The scale of such requests from the customers and the stringent requirements of latency to provide better experience demanded our data centers to employ new methods to service data. Here, with the advent of 64 bit multi-core CPUs and the drop of DRAM price per GB made system architects look into in-memory computing systems, which data is primarily serviced from DRAM, eliminating the negative impact of HDDs. This brought a proliferation of in memory systems in the data center, In memory systems such as memcached [3], Redis [4], MongoDB [5] and VoltDB [6] are aggressively used

as main workhorses to support latency sensitive applications. However, DRAM is both space and power hungry, so it can run short of supporting the ever increasing scale of data in our data centers.

1.1.1 Flash SSDs for Latency Sensitive Applications

Fortunately, there has been advances in technology resulting in the emerge of next generation memory technologies such as PCM, STT-MRAM, FeRAM and also NAND Flash. These technologies, each cultivated from the laboratories of both the industry and the academia emerged to be our next storage technology, expected to save us from the shortcomings of our HDD based storage architecture. NAND Flash memory technology, in the form of flash SSDs, was the first breed of such next generation memory to be commercialized, steadily replacing prior technologies based on the value of affordable microsecond level random access memory.

Here, parallel small random reads of flash SSDs are gaining interest in the context of SSD-backed key-value storage which is considered as a good use case of SSDs [7, 8], because of the read-oriented small I/O from such workloads [9]. Even though flash SSDs have higher latencies than DRAM, the IOPS and the latency of small read I/O of flash SSDs are enough to match the service level agreement (SLA) typically found with these systems [10]. This usage of flash SSDs was made possible because of the significant amount of internal parallelism adopted by SSD vendors to keep up with the expectations of such applications. However, flash SSDs are yet to dominate latency sensitive (or critical) applications such as key-value stores because of the latency variability of each access.

1.1.2 The Impact of Flash SSD Latency Variability on the I/O Path

Such latency variability has profound implications on the I/O supporting system software where the variability increases the software overhead which the microsecond latency of flash SSDs already magnified. In the case of flash SSDs, the latency portion of the I/O supporting software is around 10% to 20% depending on the generation

of the underlying flash technology, though it is challenging to achieve both IOPS and latency without sacrificing one another. This dilemma is because of the context switching overheads caused by multiplexing the significant amount of parallelism required for our high die count multi-channel multi-way flash SSDs. If we assume predictable access latency, polling [11] can address such overheads, but flash SSDs are far from predictable. The variable latency of flash SSDs makes it impossible for the method to be effective, which in turn leaving the I/O stack to sacrifice latency to avoid the negative impact in system parallelism.

Resolving such problems would require the system software to be aware of the variability and take measures to mitigate the impact. However, this conflicts with the current trend of research in reducing the amount of system software execution from the I/O path evolving towards direct hardware access. The impact of context switches account at least 7 µs per I/O request and is comparable to the amount of latency reduced by direct hardware access (reduced from 13.5 µs down to a few microseconds).

1.1.3 The Impact of Latency Variability Exposure

With current flash SSDs, the latency variability leaks from the SSD and escalates all the way up to the application which affects end-to-end user experience. With flash SSDs, the user experience of latency sensitive applications are mostly bound by the read request latency, and the millisecond latency of writes and erases are usually hidden under a write buffer at several levels of the I/O path including the flash SSD.

However, the millisecond latencies can impact foreground read latencies in the presence of mixed read-write workloads typically seen in applications because there is a significant difference between read, write and erase latencies. If a pending write or erase request is present on a NAND chip, the following conflicting read request will experience the millisecond delay of the pending requests.

Even though flash SSDs employ large numbers of flash chips to disperse the requests, it is hard to avoid such conflicts under arbitrary read-write requests. Such variance is where flash SSDs fall short even though flash SSDs do have attractive traits

of affordable microsecond scale random access memory. While the multi-GB per second bandwidth of modern flash SSDs are sufficient to provide value over HDDs, the millisecond scaled spikes make it difficult to use flash SSDs in place of DRAM. Such leakage of variance can be challenging to resolve and can negatively impact service level throughput of data centers. Google [12] has presented the negative impact of latency variance experienced from low-level system components.

1.2 Dissertation Goals

In this dissertation, we are motivated to propose new OS I/O path optimizations to enhance the user experience of flash SSDs under latency sensitive applications. The primary goals of this dissertation is as follows:

- **Flash SSDs for Latency Sensitive Applications:** Enable the use of flash SSDs in place of DRAM to serve latency sensitive applications by resolving the impact of the latency variability of flash SSDs.
- **Host Side I/O Path Optimizations to Resolve the Impact of Latency Variance:** Develop OS I/O path optimizations from the host side which resolve (tolerate or eliminate) the impact of the latency variance of flash SSDs without sacrificing IOPS or latency.
- **Overcome the Architectural Challenges which Impacts the I/O Path:** While addressing the impact of latency variance, we aim propose I/O path optimizations which overcomes the architectural challenges of the narrowed gap between flash based storage and post-Moore multi-core CPUs.

1.3 Approach

To address the impact of latency variance of flash SSDs on the I/O path, we have set ourselves to focus on proposing optimizations based on two distinct approaches. The approaches are 1) exploiting additional resource based on parallelism of multiple CPU

cores or SSDs, and 2) exploiting the enhanced interactions between the host and the SSD.

First, the approach of exploiting additional resources in favor of sustaining both IOPS and latency came from the observation where prior research have limitations on sacrificing one of either IOPS or latency when addressing variance. While such tradeoff is not obvious, providing a new tradeoff which does not sacrifice the core values of the system would benefit system designers.

Second, the approach of exploiting enhanced interactions between host and the SSD is based on the observation that the key factor in addressing the impact of variability comes from resolving the oblivious interactions between components within the I/O path. For example, while the key element in addressing the latency spikes is to coordinate each I/O requests to avoid the conflicts, the black-box interface designed around boundaries (i.e., flash SSD interface) prohibits such optimizations. Instead of working around the interface, our approach is to tackle the interface itself directly to support the necessary coordination.

1.4 OS I/O Path Optimizations for Flash SSDs

Based on our goals and approach, this dissertation proposes new OS I/O path optimizations to overcome the challenges of the latency variability of flash SSDs when used under latency-sensitive environments. Here, the proposed OS I/O path optimizations are summarized as follows:

- **Enhancing IOPS by Reducing the Impact of Context Switches:** To enhance the IOPS scalability of the OS I/O path using flash SSDs, we propose a series of optimizations which reduces the impact of context switches. The optimizations simplify the I/O completion path by eliminating bottom half software IRQ based I/O processing contexts or inter-processor based IRQ steering contexts. Negative impacts of removing such contexts are mitigated by optimizations such as lazy I/O processing on the I/O thread contexts or work stealing cooperative I/O pro-

cessing which is based on exploiting the parallelism of multiple CPU cores. The optimizations were made possible by introducing an HAL (Hardware Abstraction Layer) which gives a better abstraction to the upper layers. The HAL exposes the abstraction of the memory mapped queue seen in many HBA (host bus adaptors) which manages multiple in-flight I/O requests and gives fine grained access to the status of each in-flight I/O. The optimized I/O path showed more than 100% IOPS enhancement over the SCSI I/O path while maintaining the same layered architecture.

- **Reducing Latency Using SSD Internal Information:** To further reduce the impact of context switches, we propose an optimization which the host-side system software exploits SSD internal information. Here, the SSD is enhanced to notify the OS device driver to make better decisions on blocking or yielding a CPU upon waiting for an I/O to complete. This enhancement was done by decomposing the behavior of a flash SSD based on I/O contexts and their destined internal H/W components. Decomposition was based on the observation that modeling individual H/W components are far easier than modeling an SSD as a whole. Simple behavioral models fed with accurate parameters explicitly provided by internal monitors or counters within the SSD-enabled low overhead, highly accurate latency prediction. Each behavioral model is activated based on the context of an I/O request and is used to provide future information of the request (i.e., expected latency) to the OS I/O processing software. This exposure of behavioral models inspired aggressive optimizations such as pre-completion and selective polling which lowers the user experienced latency of each I/O requests. The feasibility and impact of our optimizations were demonstrated on an SSD development platform equipped with actual MLC NAND flash chips.

- **Tail Latency Reduction Using Host Side GC Control and Multiple Devices:** User experienced latency variance caused by I/O requests colliding on SSD internal resources are mitigated by exploiting host-side GC control and multiple

SSDs. Here, host-side GC control provides the ability to avoid foreground I/O requests to conflict with background GC operations. The parallelism of multiple SSDs provided means of distributing I/O requests to avoid further conflicts. The benefits of such scheme were demonstrated by implementing a key-value storage engine which employs multiple SSDs with the ability to control the physical destination of each I/O requests and the ability to control SSD internal tasks such as garbage collection. Our storage engine employs additional flash SSDs. However, the engine does not harm the inherent latency of the underlying flash SSDs.

Exposing the abstraction of physical LBA partitions based parallel H/W components turned out to be useful in cutting the long tail of user experienced flash SSD latency.

1.5 Contributions

The OS I/O path optimizations proposed in this dissertation contributes in resolving the impact of the latency variation of flash SSDs under the requirements of latency sensitive applications. The main contributions are summarized as follows:

- **Design and Implementation of an Optimized Device Driver for Flash SSDs** which reduces the variance induced software overheads without compromising IOPS (system parallelism) and latency. This device driver solves the problem of the increasing overhead of context switches which is hard to avoid without sacrificing IOPS or latency because of the highly variant nature of flash SSD latency. This work contributes in proposing a technique which embraces the latency variance of flash SSDs by introducing a practical optimization which tolerates the latency variance.

- **Design and Implementation of a Key-Value Storage Engine for Flash SSDs** which conceals the impact of latency variation of flash SSDs under arbitrary

workloads without sacrificing average latency. While the storage engine trades more NAND chips for stable latency, the engine gives system designers a trade-off to support latency sensitive applications. Also, this host-side storage engine has the capability of masking the latency variance of flash SSDs without requiring a particular type of flash SSD implementation, which is a value appreciated by system architectures designing systems for the data center.

- **SSD Extensions and Interface Designs** which enhances the capability of host side system software to deal with the latency variability of flash SSDs. Both the presented device driver and the key-value storage engine benefits from these extensions and goes and can go an extra mile towards a more efficient cross-layer optimization based I/O path design.

  This work contributes in proposing extensions which the host side system software benefits from information from SSDs. While there are several cross-layer proposals in hinting flash SSDs towards efficient flash SSD behavior, this work focuses and contributes in the opposite direction which was relatively less studied, but now gaining interest within the context of special purpose I/O stack implementations.

1.6 Dissertation Structure

The remainder of this dissertation is organized as follows. In the next Chapter (Chapter 2), the adoption and the impact of modern flash SSDs and the trends which challenge the user experience is discussed. This will be followed by three chapters each describing our engineering efforts on enhancing IOPS (Chapter 3), latency (Chapter 4), and latency variation (Chapter 5). Chapter 6 will conclude this work.

Chapter 2

Background

2.1 Adoption of Flash SSDs and the Impact on Our I/O path

Flash SSDs, because it was first targeted as a drop in replacement, it has been so successful in the market as a disruptive technology. This success brought significant interest from both the industry and academia, and led generations of advancement in flash SSD technology. Even though NAND flash technology had issues with its non-overwritable destructive nature, the success drove the technology to achieve both increase in performance and decrease in price, which made flash SSDs even more viable. Flash SSDs were first intended to be placed in the location of HDDs, but the merits of cheap microsecond access latency to a large density of random access non-volatile memory brought new applications which complements or even replaces the role of DRAM in our systems.

However, the success soon revealed that our I/O path developed around traditional DRAM + HDD based architecture were coming short in supporting flash SSDs. Applying flash SSDs require major changes in our I/O path, or otherwise the potential of flash SSDs would not benefit the user. This is because flash SSD latency scaled in microseconds, even though it was 100 times larger than DRAM technology, was small

enough to magnify the time spent in host portion of the I/O path. This phenomenon has been noticed by the researchers projecting the impact of next generation memory technologies and brought several proposals refactoring the I/O path in order to properly support high performance storage devices such as our flash SSDs. In such studies, the expected latency incurred by the underlying memory technology was in nanoseconds, so it required significant changes in the I/O path minimizing the time spent in the host portion of the I/O path.

2.2 Next Generation Memory Technologies

In the storage perspective, researchers within the context of next generation memory technologies initially noticed this problem. Compared to the latency expected from next-generation technologies such as PCM, RRAM, MRAM and STT-MRAM, the latency induced by prior I/O processing software was unacceptable. The latency of the SCSI I/O path (Linux) for a SATA SSD was reported to be around 13.5 microseconds while the latency of next-generation memory technologies was expected to be at nanosecond levels. This inspired researchers to rethink and refactor our OS I/O path to better support new devices based on next generation memory technologies [13]. Early proposals such as Moneta [14] explored such new designs and was successful to reduce the latency of software overheads down to one or two microseconds, though required holistic reconsideration of the OS I/O path design. Aligned with such holistic efforts, the vastly different character of new technologies also required us to rethink prior HDD based I/O mechanisms and strategies running through the I/O path. Doing so requires an end to end approach which spans from the application all the way down to SSD internals. It was about time to redesign our I/O path end to end.

Nevertheless, such proposals were not enough or not even clear whether they would improve the performance of our flash SSDs. Such research was good enough to project the necessity of I/O path refactoring in the coming future, but failed to be justified in the context of flash SSDs. This is because flash SSDs are different. Compared to

the memory technologies to come which are projected to have nanosecond latencies, flash SSDs have intermittent latencies neither in nanoseconds or milliseconds making it difficult to apply techniques which were applied in the context of next generation memory technologies.

2.3 The Impact of Modern Flash SSDs

After the success of penetrating into the market, the technology revolving around flash SSDs have been advanced towards to a higher performance storage device with lower price per GB. Two major drivers we observe is the advance in NAND technology and the advance in SSD architectures. NAND technology has been advanced in order to provide much higher densities, and SSD architectures have been advanced to have more parallelism. NAND technology has been advanced in order to provide much higher densities of NAND memory at a lower cost, and SSD architectures have been advanced to have more parallelism to support the higher data demands from the host system. Significant enhancements both in prices and performance have been possible, however at a price which impacts the end to end experience (end user experience) of flash SSDs. Here, we claim that the advances in NAND technology and SSD architectures make it more difficult to preserve end to end experience.

The impact of high density NAND chips has been discussed by Grupp et. al. [15]. While SSD manufacturers strive towards high density for higher bytes per dollars, increasing density using cutting edge flash chips can adversely affect performance, power consumption and lifetime of these SSDs. Especially if we consider performance, higher densities can sacrifice latencies of reads and writes as in Figure **??**. Increasing the density of NAND chips requires both shrinking the feature size and packing more bits (MLC & TLC), though this brings difficulties in both programming and reading NAND cells without errors. This impacts the system with increased magnitude and variation in terms of latencies.

Under the demands of high bandwidth and the under the pressure of increasing

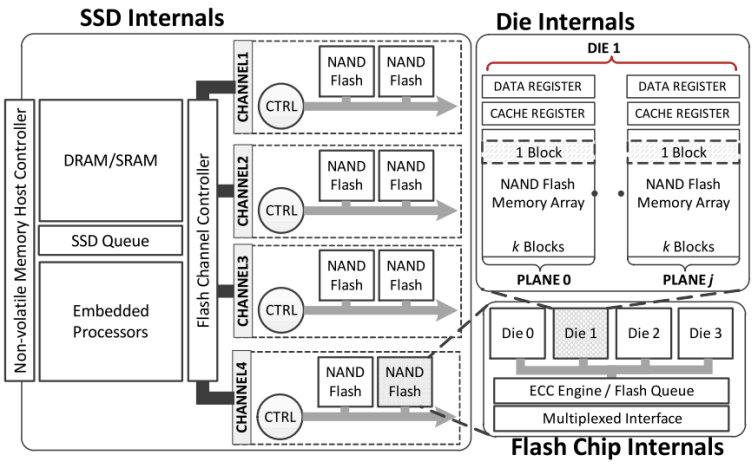


Figure 2.1: Modern SSD architecture (source: Jung et. al. [1])

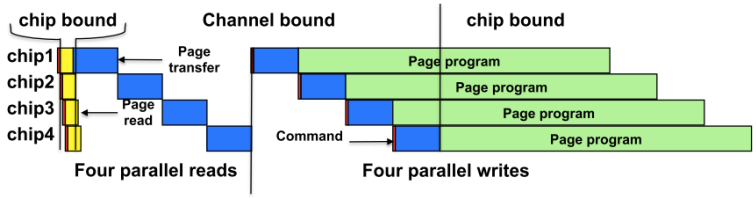
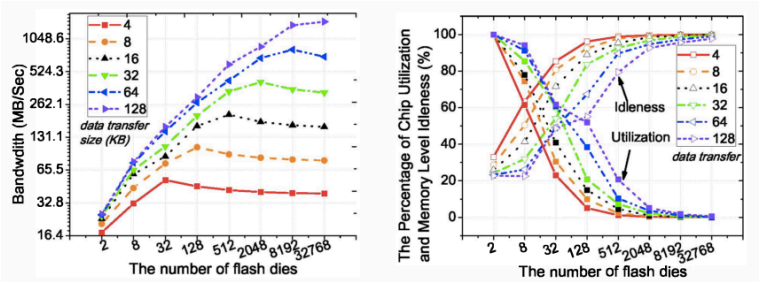


Figure 2.2: SSD internal parallelism (source: Bjørling et. al. [2])

latencies of individual NAND chips, modern SSDs employ significant amount of parallelism to solve the problem [16], by having multiple independent channels, where each channel has multiple NAND flash memory chips (Figure 2.1). Even with the higher latencies of high density NAND chips, SSD architects can exploit I/O parallelism of multiple NAND chips to achieve high performance (Figure 2.2). However, the parallelism does not always lead to higher performance (Figure 2.3). Without careful scheduling of I/O requests on these multiple NAND chips, hotspots as well as idleness can occur as in Figure 2.3) leading to suboptimal performance.

Flash controller technology which supports high density NAND chips and the much sophisticated SSD architectures have been advanced to cope with such changes and the difficulties induced by the advances. However, the everlasting advance in both directions pressures the controller in terms of technological effort, which eventually



(a) Performance Stagnation (b) Utilization and Idleness

Figure 2.3: Performance stagnation with many-chip solid state drives (source: Jung et. al. [1])

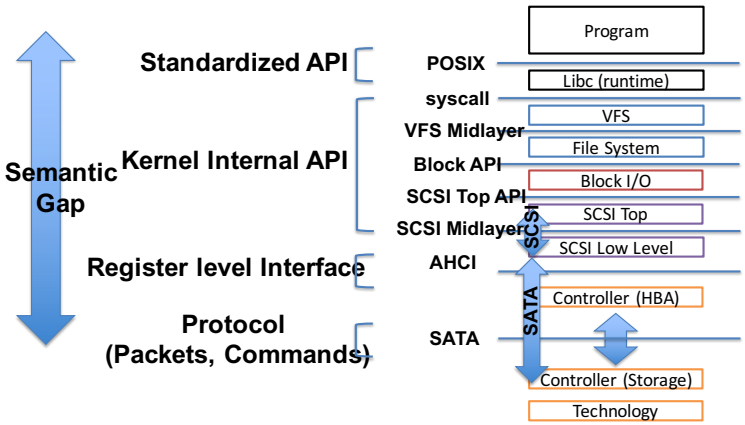


Figure 2.4: Layers and I/O interfaces within the I/O path

increases the price of flash SSDs. Nevertheless, the observation is that the black box based design concept which runs in our I/O path would worsen the situation. Without breakthrough in design concepts, the end to end experience would not reach to an optimal point, which leaves us to devise new approaches in I/O path optimizations.

2.4 I/O Path for Flash SSDs

Figure 2.4 depicts an I/O path we rely on with our third generation SATA3 or SAS SSDs. Starting from the userspace (the application) all the way down to the NAND chips (memory technology), there are several layers in between forming a layered ar-

chitecture. This particular architecture is actually based on SCSI standards, which is designed to accommodate a large ecosystem of various vendors, devices, use cases and applications. Each players in the storage industry, can dive and perform their role within a layer without worrying much about other layers, thanks to the value of transparency and separation of concerns. For example, a storage device vendor can develop a storage device and ship it with a SCSI compliant low level device driver without the burden of developing a file system for it. This feature of the architecture made the I/O path an ecosystem of a the whole storage industry.

However, such layered architecture can introduce difficulties in terms of performance optimizations. While the architecture is good for long term prosper for several players in the industry, end to end performance which requires coordinated optimization of several layers can cause difficulties. Transparency and separation of concerns is beneficial in a way that a layer does not need to care about the details of the other layers, though in other words it means that the layer is on its own. If the layer is not designed to coordinate the effort, due to the semantic gap which can be introduced by the layers can make it difficult to achieve optimal performance.

2.5 Related Work

2.5.1 Reconsidering the I/O path

The necessity of reconsidering the I/O path has been noticed by research related to analyzing the impact of next-generation memory technologies, which inspired several optimizations in order to cope with nanosecond scaled latencies [14, 17, 11, 18, 19]. Compared to the projected nanosecond scaled latencies, the overhead of the OS I/O path was big enough to cause performance problems. These proposals refactored the OS I/O path to a more lightweight and avoided overheads such as interrupt induced context switches using I/O strategies such as polling [11]. Yet, the performance of Flash SSDs were not competitive enough to be a problem, though flash SSD performance increased over the years which also motivated changes in the I/O path. Linux

I/O Path optimizations based on analyzing flash SSD impact on systems were proposed and were explored as well [20, 14, 21, 22, 23, 24].

Though the problem of flash SSDs were not limited to the OS I/O path magnified by the lower latencies. In general, there has been proposals in advocate of refactoring the I/O path [25, 20, 21, 11, 2, 26, 27, 28, 13, 29, 30, 31, 32, 33], by enhancing the interaction between storage devices and the host side I/O path. To this end, optimizations based on new interfaces [34, 35, 36, 37, 38, 39, 28, 40, 41, 42, 43, 19, 44] have been proposed. New interfaces with new commands [40, 34], new data transfer mechanisms [37, 38], efficient host to device interactions [19] and new primitives [35, 28, 39] were proposed to enhance the interactions between storage devices and the host side I/O path.

Cross layer designs can be found in other domain which rely on a strict layered architecture (i.e., OSI 7 layers) to maintain prosper of the industry. In the wireless network community, there has been several proposals which considers breaking the boundaries of the layers in order to optimize performance [45, 46, 47, 48, 49, 50]. One interesting thing is that such performance issues were mostly end to end issues. Despite of concerns such as [47] have been raised, such proposals are constantly being raised by various researchers until now [50].

2.5.2 Black-box Approaches

Optimization based on black box approaches are in fact the ones we already employ in our I/O path [51, 39, 52, 53, 54, 22, 28, 40, 55, 56, 57]. These optimizations are done two ways in which 1) host S/W can be aware of flash SSDs (SSD aware optimizations [54, 52, 40, 22, 57, 53, 55, 51, 56, 39, 28]), or 2) SSD controller S/W can be aware of the workload (workload aware [58, 52, 59, 60, 61, 62, 63, 64]). The benefits of such approaches comes from the value of transparency and separation of concerns. Given the I/O path, evolved into a layered architecture to be an ecosystem, modifications in respect of the optimizations are well isolated from each layers. This has been the default mode of optimizations in various layers in the I/O path.

However as described in the background (Section 2), the pressure from high density unstable NAND chips, the demands of higher performance and the shortage of CPU cycles for an I/O request makes us strive to find a new approach. With the black box approach, the more aggressive optimizations are often discouraged due to the lack of information (awareness) originated from black box interfaces.

2.5.3 Cross-layer Approaches

The difficulties in black box based I/O paths can be alleviated with cross layer optimizations. In such designs, each I/O components (i.e, file system, device driver and the SSD firmware) are aware of each other forming a specialized I/O path which eliminates the inefficiencies from an end to end viewpoint of the I/O path. The awareness cultivates from explicit and also efficient inter layer or communication channels or interfaces exchanging behavioral information to enhance the awareness. With such higher degree of awareness, each I/O components can perform aggressive optimizations without introducing the unnecessary overheads. Because of these benefits, proposals based on cross-layer optimizations are gaining interest [65, 31, 66, 52, 40, 44, 43, 67, 68, 69, 70, 33].

Cross-layer optimizations can also head in two ways in which 1) SSD internal optimization issues are assisted by the host, and 2) host optimization issues are assisted by the SSD. For example, the difficulties of reducing GC overheads can be alleviated by having information about the lifetime of blocks informed by the application [65]. And the latency spikes and performance drops caused by ill scheduled GC operations can be addressed [67, 70, 66] with GC interfaces presented in [66].

Other than this, other forms of cross-layer cooperation can be found in various layers [31, 52, 40, 44, 43, 68, 69, 33]. For example, NVMKV [43] is an FTL aware cross-layer approach involving interactions between host only components, as well as PAQ [52] is an optimization which refactors the I/O path within the SSD exploiting information about the physical address space.

2.5.4 Refactoring the I/O Path

There are I/O path designs which are not directly related cross-layer designs. In this work we identify them as refactoring of an I/O path, but not cross-layer designs. One example is computation offloading [71, 72, 73, 74]. Computation offloading is an I/O path refactor proposal that explores the possibility of relocating computation near to the NAND array, in order to benefit from larger internal bandwidth and the benefits of low power embedded processors. Transactional SSDs providing atomicity to I/O operations can be considered as another example [75, 35, 36, 76, 77, 78, 79]. This type of I/O path refactoring moves the responsibility of providing atomicity to the SSD since the non-overwritable nature of SSDs employing internal mapping tables can be used to support atomicity while removing redundant logging schemes (log on log). And since the role has been moved within the SSD, the SSD now has a higher level interface which expresses the intent of atomicity.

Nevertheless, this does not mean that cross-layer designs are totally unrelated to these I/O path refactoring effort. For example, host side FTL implementations or relocating the role of the FTL to the host is a refactor of the I/O path [68, 43, 44, 32, 33], but it can be a deliberate decision to make cross-layer cooperation more likely to happen (i.e., FTL awareness implemented more easier since the FTL is already in the host).

Chapter 3

IOPS Improvement by Reducing the Impact of Context Switches

3.1 Introduction

In this Chapter, we present OS I/O path optimizations for NAND flash solid-state drives, aimed to minimize scheduling delays caused by additional contexts such as interrupt bottom halves and background queue runs. With our optimizations, these contexts are eliminated and merged into hardware interrupts or I/O participating threads without introducing side effects. This is done by placing I/O operations in hardware interrupts or I/O participating parallel threads (Figure 3.1). Side effects of longer I/O processing delays are addressed by adopting a cooperative I/O processing model. All participating I/O threads actively share the burden of detecting I/O completions, performing I/O post processing and issuing new commands. These I/O operations are exposed at a hardware abstraction layer which provides abstractions such as queues, tags and notifications commonly found in modern host controllers. The interface of the layer allows I/O threads to make synchronous decisions on whether to process pending I/O commands or not.

For evaluation, we implemented an I/O path based on our optimizations for an

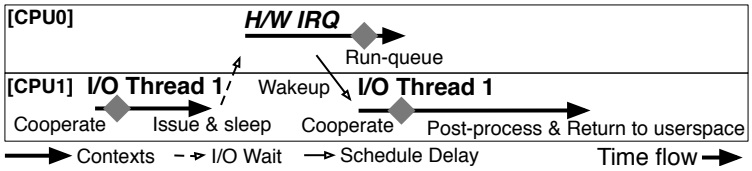


Figure 3.1: Minimizing scheduling delays within the I/O completion path

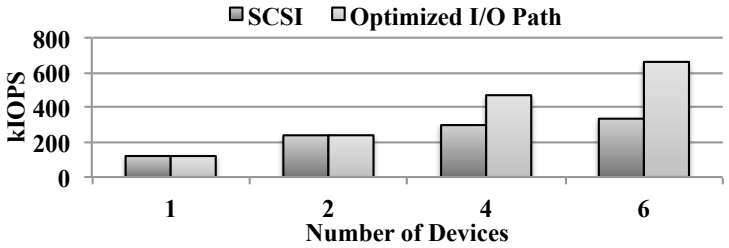


Figure 3.2: I/O performance of parallel 512-byte small random reads (Six SATA 3.0 SSDs attached to a desktop I/O chipset integrated AHCI controller)

AHCI controller which can be considered as a worst case scenario for SSDs. We built a low cost system using six commodity SATA 3.0 SSDs connected to a single AHCI controller. With parallel 512-byte small random reads, our optimized I/O path was capable of accommodating up to five devices at 671k IOPS, while current Linux SCSI based I/O path was limited at 354k IOPS (Figure 3.2). Evaluation on an SSD backed key value system showed IOPS improvement using our I/O optimizations. Performance gain of our I/O path was 7% with the highest throughput (32 clients) and 108% under the highest load (256 clients).

3.2 Motivation

Whenever a new context (interrupt or thread) is introduced in the I/O path, scheduling delays, which can be significant on a busy CPU, are added to the I/O path (Figure 3.3). We were motivated to minimize these scheduling delays, which can be significant for SSDs, within the I/O path (Figure 3.1). However, it is not trivial to remove these contexts since these contexts are employed to maintain system responsiveness and system

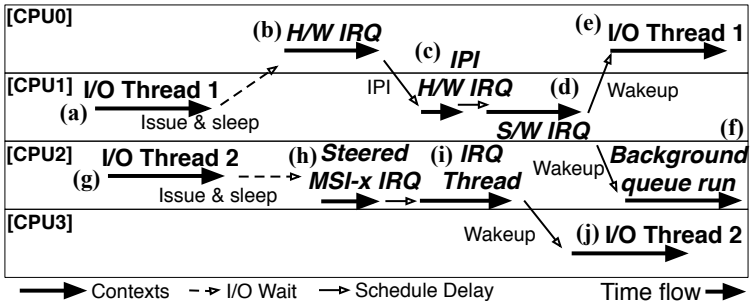


Figure 3.3: Scheduling delays in the I/O completion path

throughput. In the following, we state our motivations to remove these additional contexts and examine how they are employed in I/O paths for SSDs.

High IOPS, Smaller I/O: Software overheads, such as scheduling delays, can be minimized by issuing larger requests. However, bandwidth waste can be significant when the workload is oriented with high rates of small random requests. This motivated our work to remove these contexts. Parallel small random reads are gaining interest in the context of SSD backed key value storage which is considered as a good use case of SSDs [7, 8]. In this type of workload, a 30:1 `GET():SET()` ratio (read:write ratio) is observed, with 90% of values less than 500 bytes [9].

Conventional SCSI I/O Path: In many modern OSes, interrupt service handler routines (ISRs) are split into two parts to minimize system lockdown caused by heavy ISRs. This leads to at least two scheduling points during I/O completions. We show this in Figure 3.3. I/O thread 1 ((a) to (f) in Figure 3.3) shows the I/O completion path of Linux which is the I/O path for current SATA or SAS SSDs attached to AHCI controllers and SCSI based SAS controllers. Software interrupts (d) are scheduled to relieve the main hardware interrupt handler (b) from I/O post processing tasks such as unmapping multiple DMA buffers and de-allocating I/O descriptors. To enhance CPU cache utilization of I/O post processing, software interrupts (d) are *steered* using inter processor interrupts (IPIs) [80] (c). In this case, an IPI to CPU core 1 is made to have I/O thread 1 (a) and the software interrupt (d) run on the same CPU. The background queue run context (f) is used to issue I/O requests which could not be

issued immediately (i.e., a busy device).

Advanced Block Driver I/O Path: Recent NVM-Express standard [81] can simplify the I/O path with deeper (64k) queues and many (64k) queues. It is possible to eliminate queue runs and IPIs, but multiple scheduling delays within the completion path still exist. In Linux, NVM-Express proposes a device driver [82] which bypasses the block layer (request queue) and the SCSI I/O subsystem. I/O Thread 2 ((g) to (j) in Figure 3.3) shows the I/O completion path of this driver. This driver performs direct issues to a deeper hardware queue, up to 64k in depth, which removes the necessity of the background queue run context (f). Also, it is possible to remove the use of IPIs for IRQ steering by having dedicated queue pairs (issue and completion) and interrupts (MSI/MSI-x) on CPU cores. IRQ handling is natively steered to designated CPU cores. Here, threaded interrupts are used, so software interrupts (d) are removed, but there are still delays of scheduling the IRQ thread (i) and scheduling the completion side of I/O thread 2 (j).

3.3 Design and Implementation

Our work was done to achieve the following goals: 1) Minimize scheduling delay by removing additional contexts, 2) Preserve the semantics of previous optimizations such as H/W IRQ relieving and IRQ steering, and 3) Generalize the optimizations to be applied to various host controllers.

To achieve these goals, we adopted a cooperative I/O processing model based on a set of fine grained operations exposed at a low level hardware abstraction layer (HAL). This HAL was introduced to generalize our optimizations to various host controllers. We first describe the HAL in Section 3.3.1 and discuss the cooperative I/O optimization in Section 3.3.3.

3.3.1 HIOPS Hardware Abstraction Layer

The HAL, HIOPS-HAL (High IOPS - Hardware Abstraction Layer), has a role of exposing access and control of necessary H/W abstractions such as queues, tags and notifications implemented in the underlying H/W interface. These abstractions are commonly found in modern host controllers used for SSDs such as AHCI, NVM-Express, SCSI-Express and various SAS adaptors. Figure 3.4 shows the architectural role of the HAL which provides a generic interface to upper layers. The role is similar to the SCSI middle layer of Linux, though our HAL gives more access and control to upper layers.

Low Level Drivers: Similar to the VFS layer and the SCSI middle layer in Linux, the interface is implemented as a template of standard function pointers. Each entry of the template defines an operation, later invoked by an API call to the HAL. These API calls are implemented in Low Level Drivers (LLDs). Additionally, upper layer specific handlers are registered to LLDs for an upcall, and the upcall is done by LLDs at the point of notification. In this way, LLDs are capable of exposing execution contexts such as interrupts to upper layers. Details of the API calls and the upcalls are described in Section 3.3.2.

Interactions with Upper Layers: Beyond the HAL, an I/O strategy layer is responsible of mediating I/O requests from the upper layers to the HAL. In this work, we implemented an I/O strategy based on a cooperative I/O model. The I/O strategy is essentially a Linux block driver which provides a block interface to the rest of the system. While I/O strategies are not limited to expose block interfaces, the block interface was intended to limit upper layer modifications. Except for a few additional functions exposed for a modified VFS layer (Figure 3.4), all other block interface functions remain the same. At the top layer, ordinary `read()` and `write()` system calls are used to perform I/O, so applications can benefit from the I/O strategy optimizations without any modifications.

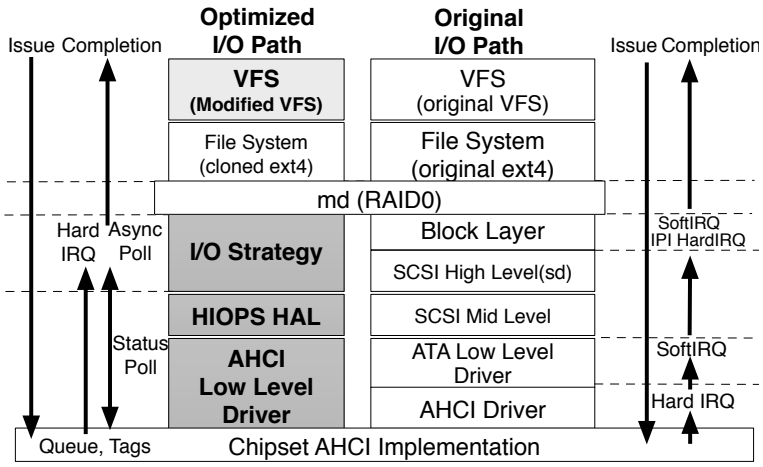


Figure 3.4: Comparison of our I/O path and the original Linux SCSI I/O path

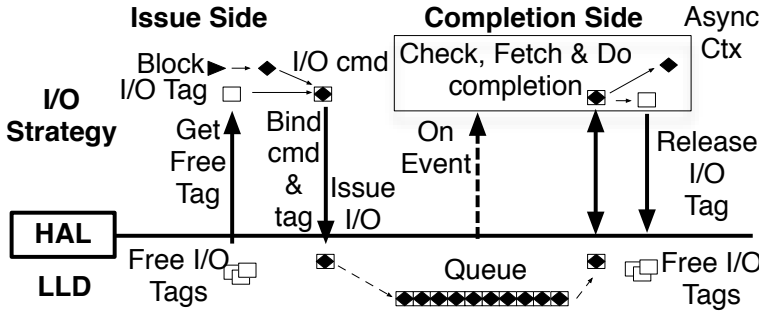


Figure 3.5: Interactions between I/O strategies and low level drivers (LLDs)

3.3.2 HAL API Operations

Figure 3.5 describes interactions between the upper layers and low level drivers (LLDs). The interactions consist of both issue side and completion side operations.

Issue Side Operations: In an I/O strategy, upper layer I/O requests are first converted to a low level command. Then a free tag (`get_free_tags`) is requested to be bound to the command (`bind_tag_cmd`). An implicit `begin_cmd` is called to timestamp the command (i.e., tracking I/O timeouts). Tags bound with commands are issued to the device by `issue_tags` calls. Note that tag related operations are named with plurals because they can be batched.

This interface gives flexibility to the I/O strategy so that it can synchronously determine whether the device is able to issue more I/O or not. If a `get_free_tags` call fails, then the device is busy.

Completion Side Operations: I/O strategies can decide whether to rely on interrupts. For interrupts, I/O strategies register a function pointer to gain synchronous access to the notification context. There, I/O strategies can check the I/O event status with `check_event` calls. Whenever there is an event, `fetch_event` is used to retrieve and process the command. If the I/O strategy does not rely on interrupts, it can synchronously check for events with the same process described above. In this case, the status check context is provided by the I/O strategy itself.

Completions are processed beginning with a `detach_tag_cmd` call to detach commands and tags. Detached tags are released to the controller with `release_tags` and the I/O strategy post processing contexts are initiated by `end_cmd` calls.

3.3.3 OS I/O Path Optimizations

In this Section, we describe OS I/O path optimizations based on a cooperative I/O model. These optimizations are implemented as a HIOPS-HAL I/O strategy which is mainly implemented as a Linux block driver.

Non-blocking I/O: No I/O contexts are blocked to acquire resources such as I/O tags without introducing additional background queue runs. In the issue path, all I/O commands are first enqueued into a simple software FIFO queue. If tags are available, a command is dequeued to be issued. Otherwise, the actual issue is deferred to other parallel issue paths or asynchronous contexts such as interrupts (Figure 3.6-(b)).

Here, the hardware interrupt context is used to issue remaining commands in the software FIFO queue. Since free I/O tags are *generated* in the hardware interrupt context, new I/O commands can be issued immediately using these free I/O tags without being blocked. Impact on the hardware interrupt was not excessive because 1) multiple CPU cores were used, and 2) modern controller features such as DMA64 relieved I/O post processing.

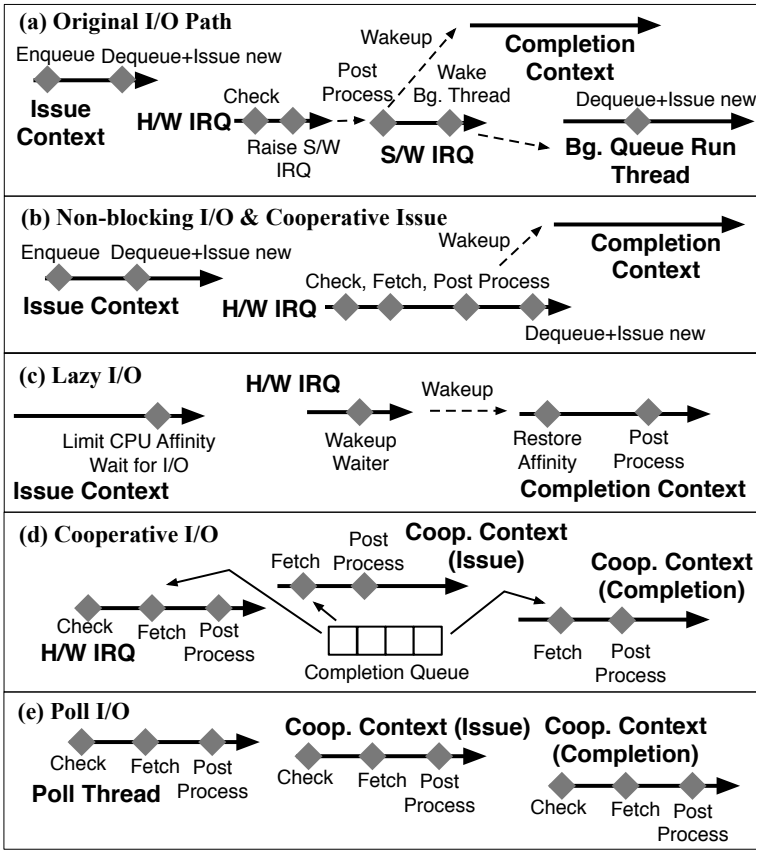


Figure 3.6: OS I/O path optimizations with HIOPS-HAL

Lazy I/O Processing: Lazy I/O processing offloads I/O processing to the actual threads waiting for I/O completions (Figure 3.6-(c)). This eliminates additional context switches introduced by deferred I/O processing schemes such as bottom halves and threaded interrupts. This was done by exposing an alternative I/O-wait function from the I/O strategy to be called instead of the original `io_schedule()`. Here, a modified VFS layer calls this I/O-wait function to provide the contexts of I/O threads calling `read()` and `write()` system calls waiting for an I/O to complete. These I/O threads are blocked inside the provided I/O-wait function. Upon completion, these I/O threads are used for I/O post-processing instead of introducing additional contexts such as bottom halves and threaded contexts. I/O post processing is done by having HIOPS-HAL API calls after the I/O thread wakeup and before the I/O-wait function exits.

After the I/O post processing, I/O threads return from the I/O-wait function and go back to the VFS layer and the userspace without any scheduling delays.

To enhance CPU cache hits during I/O post processing, waiters are awaken on CPUs where they issued the I/O and went to sleep. This is achieved by temporarily limiting the CPU affinity mask of a waiter thread to the current CPU before going to sleep. After the thread wakes up, the CPU affinity mask is restored.

Cooperative I/O Processing: Cooperative contexts are introduced by having HAL API calls from both the I/O issue path and the completion path (Figure 3.6-(d)) inside the I/O strategy. These are helper contexts which perform I/O tasks of other threads. All I/O threads voluntarily enter this cooperative context for every I/O request being frequently scheduled on the CPU. Here, I/O tasks can be carried out in a timely manner, even if the I/O owner thread is not being scheduled on the CPU. In a multi-core machine, the parallelism of I/O threads entering cooperative contexts increases overall I/O processing throughput of the system.

For cooperation, completion contexts make `fetch_event` calls to *steal* I/O processing work from post processing handlers. Issue threads performing non-blocking I/O issues for other I/O threads play another form of cooperation (Figure 3.6-(b)).

Poll Based I/O: Under higher loads, interrupts can be disabled. With interrupts disabled, a poll thread is introduced to poll for new I/O completions. Additionally, cooperative contexts are set to perform opportunistic poll (Figure 3.6-(e)). Note that polling is for the whole controller, not for individual I/O tags. Under high loads, the processing times of individual I/O commands are unpredictable, but the interval between multiple I/O commands completing in parallel is predictable (0us to 20us). After a single poll cycle, the poll thread releases the CPU and relies on high resolution timers to schedule the next poll. Poll thread introduces the overhead of timer interrupts, but the use of additional cooperative contexts lets us perform a rather coarse poll (16us to 32us).

It is possible to implement a hybrid mechanism to switch between the use of interrupts and poll methods, however the complication of determining mode switch leaves

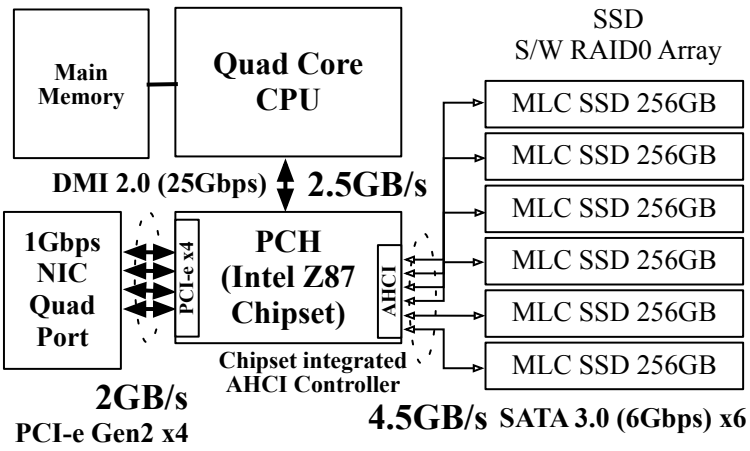


Figure 3.7: Hardware block diagram of our system

this implementation for our future work. Our experiments showed that indicators such as the current level of parallelism (occupied queue depth) combined with the current level IOPS can be a candidate to permit such tasks.

3.4 Evaluation

Figure 3.7 shows the system we built for evaluation. The impact of our optimizations was evaluated using a micro-benchmark application and a key value storage. `fio` 2.1.4 was used for our micro-benchmark evaluations, and Aerospike 2 was used as the key value system [8]. `YCSB` [83] was used to load the key value system.

Implementation: Our optimizations were applied to a Linux 3.2.40 kernel as dynamic loadable modules. These modules include the HAL itself, HAL I/O strategy, modified VFS and our custom AHCI HAL LLD (Figure 3.4). Here, the HAL consists of 6,187 lines of original code and the HAL I/O strategy with 1,766 lines of original code. The AHCI HAL low level driver was based on the AHCI SCSI libata device driver of Linux 3.2.40 but was modified to be a HIOPS-HAL LLD. Total 2,424 lines were original for HIOPS, and 2,632 lines were adopted from Linux 3.2.40. Modifications on the VFS layer was as small as 44 lines. However, the ext4 file system and the

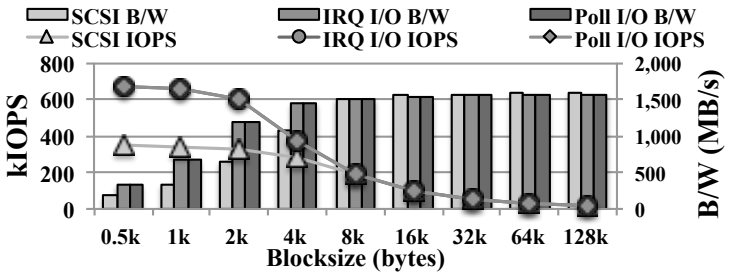


Figure 3.8: IOPS and bandwidth with increasing I/O block size (`fio`, Direct I/O, 128 threads, six SATA 3.0 SSDs, Software RAID0, 128kB stripe, `ext4`, `noop` scheduler(SCSI))

associated VFS layer consisting of the generic page cache and the direct-I/O path were cloned to be a kernel module for ease of experiments.

Experimental Setup: We conducted our evaluations on a PC with an Intel i7-4770 3.40Ghz hyper-threaded quad core CPU and 16GB DRAM. The system was equipped with six Samsung 840 Pro 256GB SATA 3.0 SSDs connected to a single AHCI controller which supports up to six SATA 3.0 ports. The AHCI controller was integrated into an Intel Z87 platform controller hub(PCH) which has a direct media interface uplink capable of 25Gbps.

I/O Throughput: Figure 3.8 shows the performance of our I/O paths with varying I/O blocksize. IRQ I/O was the hardware interrupt based I/O path presented in Section 3.3.3 and Poll I/O was the I/O path with interrupts disabled. With Poll I/O, the poll thread and the cooperative contexts performed poll altogether. All other optimizations were applied in both I/O paths described in Section 3.3.3.

Our I/O paths achieve from 32% (4kB I/O) up to 89% (0.5kB I/O) IOPS gain over the original SCSI I/O path. At the maximum, IRQ I/O achieved 671k IOPS while SCSI was at 354k IOPS. IRQ I/O achieved 671k IOPS at maximum, while SCSI led 354k IOPS. However, there was no significant difference in IOPS between IRQ I/O and Poll I/O.

For requests larger than 8kB, there was no gain since the bandwidth was limited

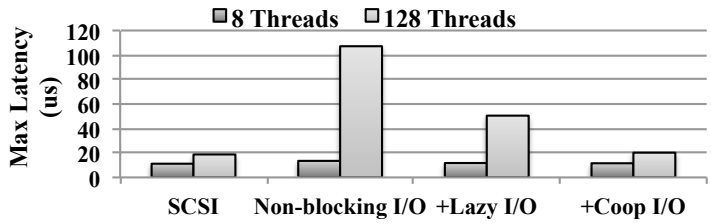


Figure 3.9: Effect of I/O processing optimizations

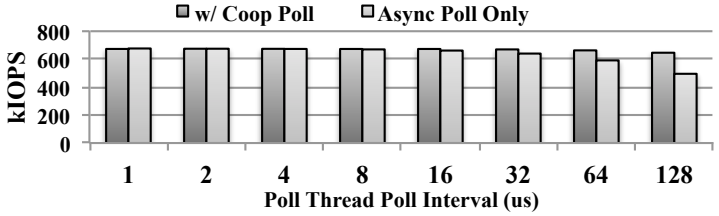


Figure 3.10: Poll interval impact

by the DMI 2.0 uplink bandwidth and the internal interconnects within the PCH. Our I/O paths with 4kB I/O were also limited by the the DMI 2.0 uplink bandwidth. The bandwidth converged to approximately 1.5GB/s which was similar to the bandwidth achievable from x4 PCI-Express 2.0 channels. This bandwidth was smaller than the DMI 2.0 25Gbps (2.5GB/s) uplink to the CPU.

I/O Post-processing Schemes: Figure 3.9 shows the effect of applying I/O post-processing schemes by showing the maximum latency of interrupt handlers. Under high loads (128 threads), basic Non-blocking I/O shows over 100us interrupt handling latency while the original SCSI I/O path shows up to 18us. This is because all I/O processing and next I/O issue had to be done in the hardware interrupt handler. When Lazy I/O is applied (+Lazy I/O), the latency diminishes to 50us. With both Lazy I/O and Coop I/O applied (+Coop I/O), the maximum latency drops to 20us which is similar to the original SCSI I/O path.

Polling: Figure 3.10 shows the impact of the poll interval. The load was 128 threads performing 512-byte direct I/O (`O_DIRECT`) `read()`s. 'Async poll' was with a single poll thread polling, and 'w/ Coop Poll' was with opportunistic completion

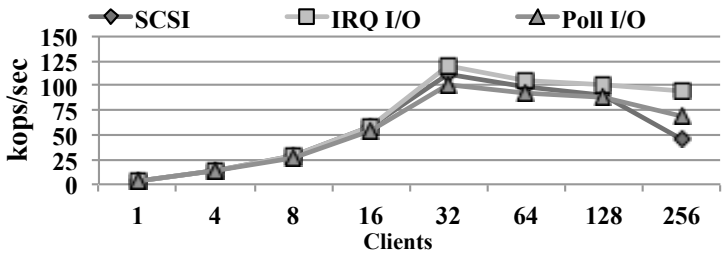


Figure 3.11: Key value storage performance (YCSB 100% `get()` performance)

checks in both issue and completion paths helping the poll thread. The results show that cooperative poll was capable of over 600k IOPS even if the poll interval increased up to 128us.

Key Value Storage: We evaluated our I/O paths under an SSD backed key value storage. For this evaluation, another identical Intel i7 quad core CPU system was linked back to back through a pair of 1Gbps NICs. To minimize the storage latency, this key value storage did not use file systems when it performed storage I/O. Also, all I/O was performed with direct I/O (`O_DIRECT`) to eliminate page cache incurred overheads. In our evaluation, six SSDs were used by the key value storage.

Figure 3.11 shows key value throughput with increasing YCSB client threads. While performance with SCSI I/O degrades beyond 128 clients, our optimizations were able to mitigate the collapse. The highest throughput was 119kops/sec achieved with IRQ I/O while SCSI I/O showed 110kops/sec and Poll I/O showed 101kops/sec. Poll I/O showed lower performance than SCSI I/O, because the storage throughput was not high enough relative to the storage throughput seen in Figure 3.8. This motivates a poll & IRQ hybrid I/O scheme. The key value storage could only load the storage up to 150k IOPS at its peak. Performance gain of IRQ I/O over SCSI was 7% with the highest throughput (32 clients) and 108% under the highest load (256 clients).

3.5 Summary

Previous I/O completion schemes for fast storage are not sufficient to support current flash SSDs. With faster memory technologies, the software delays of multiple context switches can be mitigated with techniques such as polling; however, the noncommittal latencies of flash SSDs, not like DRAM nor like hard disks, require the use of different approaches in addition to such a technique.

In this work, we have presented a low latency I/O completion scheme based on a cooperative I/O processing model. Additional I/O contexts such as bottom halves and background queue runs are eliminated, and their absence is compensated with the opportunistic help of I/O participating threads under parallel high IOPS workloads. I/O workloads with low parallelism can enjoy the lower latency of the our simplified hardware interrupt based I/O post processing. Our evaluation on an SSD backed key value storage suggests that workloads of high I/O parallelism will benefit from our I/O completion scheme. With an AHCI controller, our work presented a read oriented low cost high IOPS configuration.

Chapter 4

Latency Reduction using SSD Internal Information

4.1 Introduction

Flash memory latency, neither in nanoseconds or milliseconds, makes it difficult to decide whether to yield the CPU or not when the CPU awaits I/O completion since the scheduling delay can cause a non-negligible impact on performance. Blocking the CPU (i.e., polling [11]) would avoid such delays but would be at the cost of sacrificing parallelism. Recently, internal parallelism of SSDs (NAND channels, chips, dies, and planes) has been increased to meet the demands on performance, capacity, and costs. To utilize the capability of SSDs fully, the OS should multiplex higher numbers of I/O contexts (i.e., threads or state machines). This, in turn, increases the chance of scheduling. Addressing this issue is not trivial unless flash technology scales its latency down to nanoseconds or we can break current CPU frequency limits in the era of post-Moore's law.

We can minimize these scheduling delays based on accurate estimation of SSD latencies. However, queueing delays caused by high parallelism in SSDs and internal operations, such as garbage collection, make it impossible. To address this issue, we

propose an optimization that enables the OS to make precise decisions on when to yield the CPU or not upon a new I/O request. The optimization eliminates or hides I/O completion scheduling delays while preserving system parallelism. This is done by placing latency predictors supported by an I/O behavior tracker inside the SSD. The tracker gathers information about the whereabouts of each I/O request and the state of each internal resource, such as DRAM buffers and NAND chips.

Here, latency predictors either aid the OS in determining the latency of the next I/O request or interrupt the OS when a pending I/O would finish in the near future. With such information, the OS can make decisions on whether to yield the CPU or not, or it can prepare itself to overlap the I/O time with the expected I/O completion scheduling delays. Such H/W and S/W interactions are done with an extended SSD interface, implemented as an in-band channel that is piggybacked on I/O completion paths.

To evaluate our proposal, we employed a Xilinx Zynq-7000 SoC FPGA-based OpenSSD 2 Cosmos evaluation board [84] accompanied with a flash DIMM module based on MLC technology [85]. Evaluations on a prototype SSD showed that our method was capable of reducing the impact of scheduling delays while having a low impact on system parallelism.

4.2 Motivation

In this work, we aim to optimize user-perceived latency of flash SSDs by minimizing I/O completion scheduling delays without sacrificing system parallelism. Here, we observed that the fact that the OS was blind to the levels of SSD latency that it would experience was the main obstacle of tackling such delays. This motivated us to explore the design space in which the SSD actively inform the OS of its behavior upon performing system optimizations.

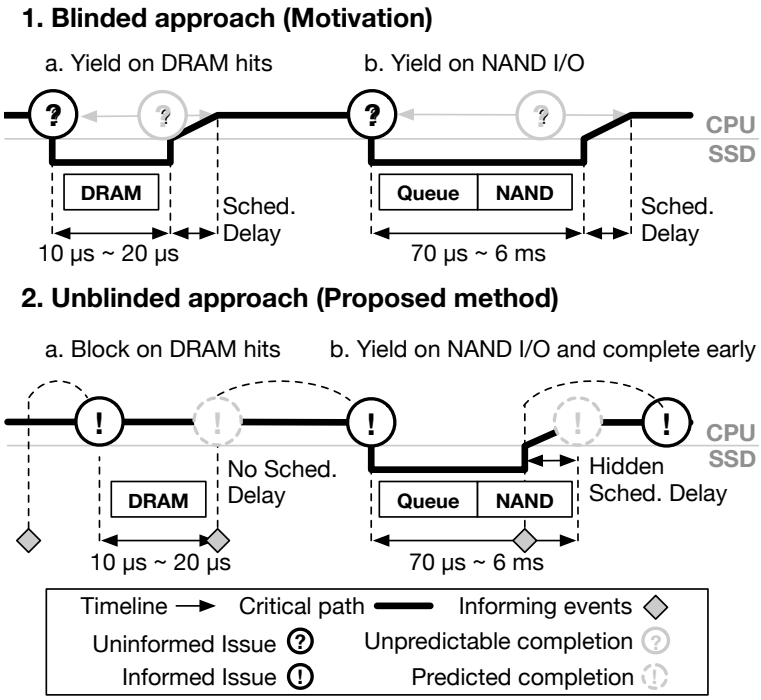


Figure 4.1: Minimizing the impact of I/O completion scheduling delays by having SSDs actively informing the OS

4.2.1 System Impact of Modern SSDs

Modern SSDs employ a significant amount of parallelism in order to keep up with the value of a high-performance random access storage device. Careless I/O control results in low resource utilization with hotspots. SSDs employ various techniques to spread I/O requests to achieve maximum utilization and performance. DRAM buffers, backed with high-capacity capacitors, are employed to serve as staging areas to perform such optimizations. Such performance considerations run deep in SSD design and have an impact on various SSD internal I/O tasks.

The higher degree of parallelism of modern SSDs burdens the host system with context-multiplexing overheads that introduce non-negligible scheduling delays. Multiple I/O contexts competing for CPU cycles vary user-perceived scheduling delays.

SSD internal tasks also cause significant variability in latency observed from the host system. Even with the presence of DRAM hits or NAND read operations, which have (fairly) favorable latencies, the OS has to assume higher levels of latencies and neglect any optimizations based on lower latencies.

4.2.2 SSDs, Unblinding the OS

Blindness of the OS, unaware of SSD latencies, is a root of all evil that leads to sub optimal conservative approaches. Figure 4.1-1) shows the impact of such conservative approach in determining whether to yield or block upon an I/O issue. For example, blocking on a CPU core in the case of DRAM buffer (or cache) hit (Figure 4.1-1 left) would eliminate the scheduling delay, but the OS yields the CPU assuming much higher latencies (Figure 4.1-1 right), and takes the penalties of scheduling delays because it has no information on such hits (at the points of question marks in Figure 4.1-1).

What if we have predictable latency? The negative impacts of scheduling delays can be minimized, since we can make a best decision that benefit the system based on an accurately predicted latency (Figure 4.1-2). To this end, we were motivated achieve predictable latency.

In this work, we positioned ourselves to define such predictability as being able to predict what comes next instead of trying to make SSD latencies adhere a constant latency value. Here, we unblind the OS by informing about SSD internals to enable accurate latency predictions based on such information. To achieve this, our proposal is to reinforce SSDs to inform the OS with appropriate information. The OS is informed to make predictions (at the bold exclamation marks) of the expected completion time of an I/O (the grey exclamation marks), making it possible to eliminate (Figure 4.1-2 left) or mask (Figure 4.1-2 right) the impact of scheduling delays from the critical path.

4.3 Design and Implementation

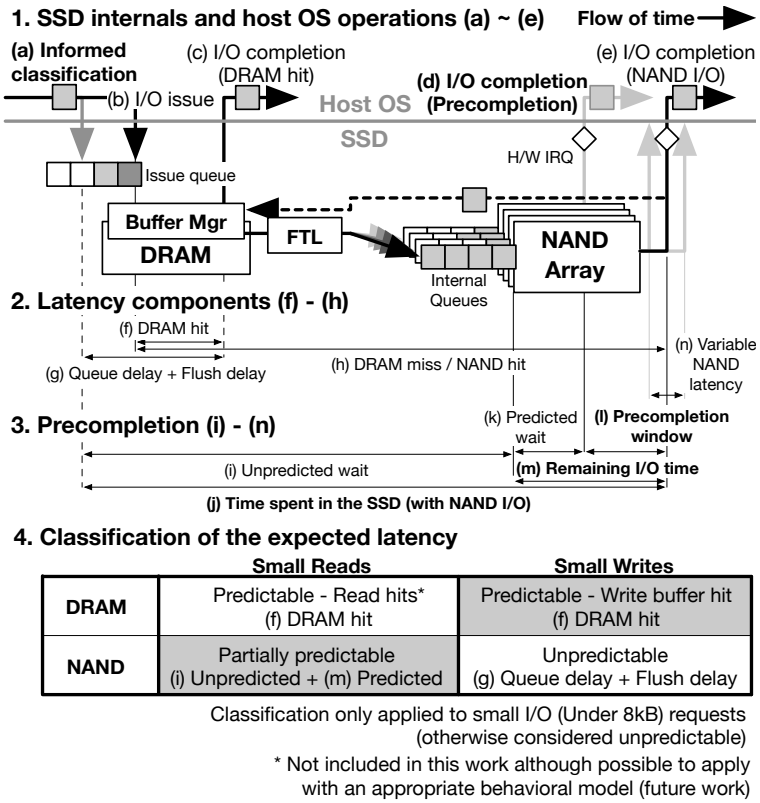
Based on our motivations, we implemented an I/O path in which the SSD actively cooperates with the OS in order to optimize user-perceived performance. The goal of the cooperation is to enable proper decisions, whether yielding or blocking a CPU upon an I/O request would be beneficial. Such cooperation is based on an accurate prediction of SSD latencies, which lies as the main challenge in our work. Our main strategy to the challenge is to predict the latencies within SSDs, not outside SSDs.

To achieve this, the I/O path has an I/O behavior tracker within the SSD controller S/W, which speaks to the predictor in the OS device driver through an extended SSD interface. Our I/O path is based on modest changes only that are limited to S/W components and implemented both in the host OS and the SSD controller.

4.3.1 Predicting the I/O Time of SSDs

The most challenging part of our design is predicting the behavior of SSDs, which is highly variable. Our approach to this problem is to decompose SSD internals into individual components (DRAM buffers and NAND chips), each behaving in a simple way (compared to the whole system), and exploit the simple behavior to ease the prediction. This prediction activity is based on a simple model of SSD internals depicted in Figure 4.2-1 (a), where I/O requests first visit the DRAM buffer for opportunities of caching (reads) or aggregating (writes) and then are issued to the NAND array.

1) Classifying I/O Requests: In the I/O path, each I/O request is classified in terms of its destined components, and the prediction is based on the previous behavior of the individual components. The I/O behavior tracker, implemented within the SSD, tracks these component behaviors, which are translated into parameters of multiple behavioral models, each representing individual components (detailed in Section 4.3.2). Based on the models and the parameters gathered by the tracker, the OS classifies (predicts) the next I/O request at the I/O issue context (Figure 4.2-1 (a)) based on the criteria, as shown in Figure 4.2-4.



4. Classification of the expected latency

	Small Reads	Small Writes
DRAM	Predictable - Read hits* (f) DRAM hit	Predictable - Write buffer hit (f) DRAM hit
NAND	Partially predictable (i) Unpredicted + (m) Predicted	Unpredictable (g) Queue delay + Flush delay

Classification only applied to small I/O (Under 8kB) requests (otherwise considered unpredictable)

* Not included in this work although possible to apply with an appropriate behavioral model (future work)

Figure 4.2: Proposal based on a simplified model of the internals of an SSD having a buffer/cache and an array of NAND chips

2) Remaining I/O Time: Even with the power of accessing internals of SSDs from an SSD controller, predicting the I/O time of an SSD is still challenging. While there are components with predictable latency, such as DRAM buffers, that a simple classification can help (upper row in Figure 4.2-4), predicting latencies of a NAND chip array (lower row in Figure 4.2-4) is difficult with the presence of multiple I/O requests colliding and queued up on resources (Figure 4.2-3 (i)), along with the inherent variability of the chips (Figure 4.2-3 (n)). To overcome the challenge, a predictor within the SSD considers only the remaining part of the I/O time (Figure 4.2-3 (m)) and predicts only for small-read operations, which have low variance in NAND I/O latency (Figure 4.2-2 (n)). Latency prediction begins only after a NAND I/O command (a single page read)

is actually issued to a NAND chip (the beginning of Figure 4.2-3 (m)), effectively eliminating the queuing delays (Figure 4.2-3 (i)) from the prediction landscape. For cases when predictions can be inaccurate (write operations) or have marginal benefits (larger I/O), the predictor simply falls back to not predicting anything.

3) Precompletion: For the remaining I/O time, the SSD side latency predictor takes the moving average of three observed latency values as the prediction and notifies the host OS of a predefined period (precompletion window in Figure 4.2-3 (l)) before (Figure 4.2-1 (d)) the actual completion occurs (Figure 4.2-1 (e)). Optimizations based on this interaction is detailed in Section 4.3.2.

4.3.2 OS I/O Path Optimizations

The simple behavior of each individual component is modeled with coarse-grained implementation neutral behavioral models (Figure 4.3), which serve as an agreement between the latency predictor and the OS, in order to provide accurate predictions as well as protect SSD internals.

1) Applying Behavioral Models: We applied two models (Figure 4.3) to model the behavior of the DRAM buffer (left) and the I/O completion of NAND chips (right). The DRAM buffer model (Figure 4.3-left) is used by the device driver to determine whether a write request would result in a DRAM hit (under buffer full threshold) or a NAND I/O (exceeding buffer full threshold). The I/O completion model (Figure 4.3-right) is used by the latency predictor within the SSD to notify the OS that a predictable I/O operation is underway and that the actual completion would occur within a period called the precompletion window (Figure 4.2-3 (l)). The models are applied to cover the I/O requests classified into the gray areas of Figure 4.2-4, although this can be extended by defining additional behavioral models (i.e., read cache hits, read prefetching hits), which leads to our future work.

2) Eliminating Scheduling Delays: For shorter latencies, experienced when an I/O request hits the DRAM buffer within SSDs (Figure 4.2-2 (f)), the OS device driver knows that this will happen by comparing the remaining space of the buffer and the

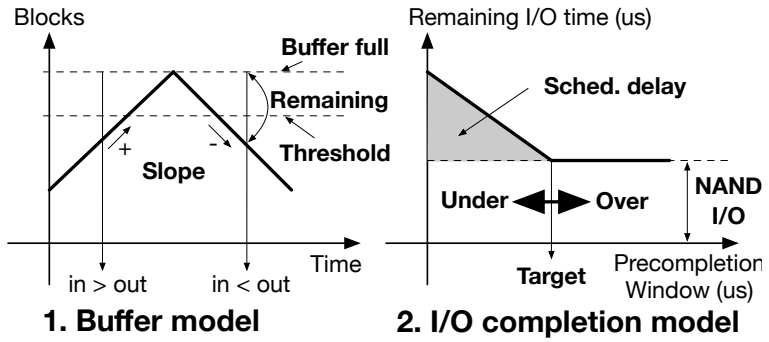


Figure 4.3: Behavioral models of SSD internals exposed to the OS

amount of write I/O it has to issue (Figure 4.3 left). This is possible since the exact amount of free space within the buffer is passed from the SSD through the completion of the previous completion. Upon buffer hits, the OS responds with blocking the CPU core (busy waiting [11]) for the I/O completion in order to eliminate the scheduling delay.

3) Hiding Scheduling Delays: For I/O requests headed for the NAND chips (Figure 4.2-2 (h)), the I/O path yields the CPU in order to preserve system parallelism at the cost of scheduling delays. To deal with this delay, the I/O path overlaps scheduling delays to hide the impact from the critical path. This is achieved by aligning the size of the precompletion window with the size of scheduling delays (target window size in Figure 4.3-right). If the precompletion window undershoots the target, scheduling delays are exposed depending on how much the window undershot (under: gray area in Figure 4.3-right). However, the window cannot overshoot since parallelism can be harmed due to the penalty of busy waits (over: right side of the target window in Figure 4.3-right). In addition, these scheduling delays can vary depending on system load, so the window should consider system load as well. Currently, the precompletion window is a fixed value given a priori that is planned to be reinforced with a dynamic feedback mechanism based on runtime measurements.

4.4 Evaluation

4.4.1 Experimental Setup

1) Implementation: We implemented a prototype SSD on top of the *"Greedy FTL firmware"*, which was included in the commercial distribution of the OpenSSD2 Cosmos evaluation board [84]. The implementation of our SSD was not a full-blown SSD, although it implements key features described in Section 4.3. One key limitation was that only a single I/O context (I/O depth 1) could be handled at a time while state-of-the-art SSDs are capable of handling 32 I/O requests (i.e., SATA 3.0) or more (i.e., NVM-Express) simultaneously.

2) Methodology: The OpenSSD2 evaluation board was connected to the host as an end point with a PCI-Express Gen1 x4 connection. The host system was equipped with an Intel i7-4770 3.30 Ghz Quad-core CPU (hyper-thread enabled), loaded with a custom block device driver that we developed on Linux 3.5.0. The I/O depth limitation limited the evaluation scenarios to a single thread competing with other parallel contexts, such as I/O threads or CPU threads. In the scenarios, we used Fio 2.1.3 for I/O threads (including the precompletion-based I/O thread) and a custom-built program that burns CPU cycles. To see the benefits of precompletions, we show the average latency without NAND latency and the throughput of the background task (CPU or I/O oriented) compared to when it was executed alone.

3) Higher IOPS with Multidepth I/O: Since the SSD prototype was limited with an I/O depth of 1, we further investigated the impact of the precompletion by implementing a DRAM backed SSD emulator on a Xilinx VC709 evaulation board. Here, multidepth I/O and a SSD latency emulator was implemented on the Virtex-7 FPGA chip which communicated with the host system via a PCI-e Gen3 x4 link. On the host system, we ported the device driver to talk with the SSD emulator which has an interface similar to NVM-e. Because this board was not equipped with NAND flash modules, data was stored in one of the 4GB on-board DRAM memory SODIMM module.

Table 4.1: Accuracy of Latency Predictions (three-value moving average)

H/W	Measured	Std. dev	Predicted	Error
Flash	352 µs	0.66 µs	352 µs	0.94 µs
DMA	9 µs	0.26 µs	9 µs	0.56 µs

4.4.2 Results

In this study, we report 1) the effect of predicting DRAM buffer hits through classification, 2) the accuracy of device-side remaining time predictions, and 3) & 4) the impact of precompletions to project the impact on full-featured SSDs.

1) Classifying I/O Requests: The impact of I/O classification was verified by measuring the latency of a single I/O thread performing small random write operations. To limit the latency impact of DRAM buffer flushes and garbage collection overwhelming the average latency, we limit the frequency of buffer full situations and separately report buffer hit latencies. Our I/O path was able to reduce average latency up to 5.8 µs from the baseline. While the baseline experiences scheduling delays even when I/O requests hit the write buffer, I/O classification allows write buffer hits to be identified and minimizes the scheduling delays by blocking the CPU.

2) Predicting Flash Latency: To verify the I/O latency predictability of flash memory when in independent devices, we measured the latency of flash commands on an LP-DDR flash DIMM equipped with four MLC 25 nm 16 GB flash chips [85] provided by the OpenSSD2 project. This measurement was done inside the SSD on top of the flash controller logic, which includes ECC correction [1]. Flash latencies for read operations had very little variance (Table 4.1) compared to other operations (i.e., DMA engine). The error, which is a root of the sum of squared differences, was less than 1 µs.

3) Precompletion I/O vs I/O Threads: Light gray bars and lines in Figure 4.4 show the system impact of the completion schemes interacting with I/O threads. Polling

[1]The high latency was due to the unoptimized implementation of the stock NAND controller of the OpenSSD2 project. Scheduling delays in this work, in terms of latency, are small with respect to the high latency of the evaluation board, although modern SSDs have lower latencies.

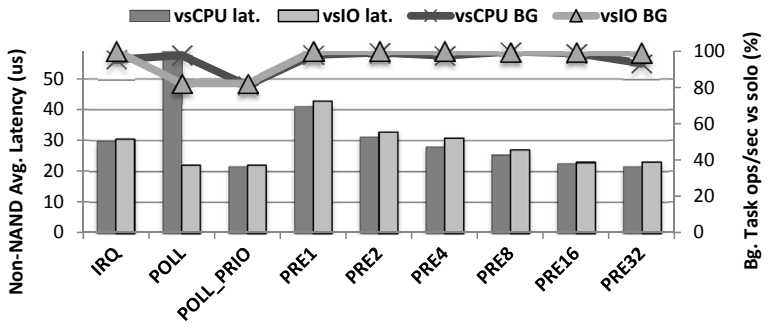


Figure 4.4: Precompletion I/O threads vs CPU threads

with (`POLL_PRIO`) or without (`POLL`) task priority (niceness) showed the best latency, while interrupts (`IRQ`) showed scheduling delays up to 11 µs. The cost of reducing this amount of scheduling delays was the excessive CPU cycles causing 17.75% throughput degradation of the background I/O threads.

Precompletions solve this dilemma between latency and parallelism by masking the scheduling delays (11 µs) underneath the SSD I/O time while doing no harm to background threads. However, a precise precompletion window should be given based on observations on scheduling delays. In Figure 4.4, the best latency was achieved with a 16 µs precompletion window (`PRE16`). Having a window larger than 16 µs burns more CPU cycles, although this was marginal compared to poll-based methods.

4) Precompletion I/O vs CPU Threads: Dark gray bars and lines in Figure 4.4 show the interactions with CPU threads. With these interactions, we had to increase the priority of the polling thread (`POLL_PRIO`) since poll turns an I/O task into a CPU thread. The CPU scheduler, Linux CFS in this case, gave the poll thread an equal share of the CPU, so the poll thread (`POLL`) had difficulties in acquiring the CPU on time. The measured latency of (`POLL`) was significantly greater (1,488 µs), while there was no significant drop in background throughput. In contrast, interrupt-based I/O threads, which are threads other than `POLL` and `POLL_PRIO`, did not experience this problem.

Here, the cost of a shortened latency of `POLL_PRIO` was a significant drop in background CPU task performance, which recorded only 80% throughput (ops/sec) compared to the solo-run scenario. Yet, precompletion effectively reduced this latency

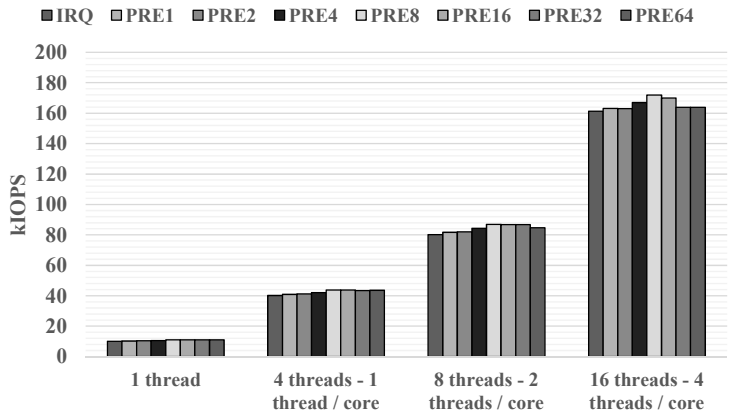


Figure 4.5: Impact of precompletion on IOPS with multidepth I/O (80 µs device latency)

without severely degrading CPU tasks. In addition, the right size of the precompletion window had to be used (`PRE8`) in this case.

5) Precompletion I/O with Higher IOPS and Multidepth I/O:

Figure 4.5 and Figure 4.6 shows the impact of precompletion on our SSD emulated environment. This evaluation was focused on confirming the impact of precompletion with multiple I/O depths and lower latency which we could not do with the OpenSSD platform. This environment was configured to emulate a 8 channel 4 way flash array which has 80 µs latency, which was similar to a SATA 6.0Gb/s SSD seen in the market. We measured the impact of precompletion under varied the number of threads performing a parallel 4kB random read I/O. Here, the results show that precompletion is able to achieve both IOPS and latency of a parallel random I/O from multiple threads. While the precompletion window has to be configured where it strikes a sweep spot (`PRE8`), the gain of both IOPS and latency was approximately 8% compared to the prior IRQ only scheme.

4.5 Summary

In this work, we presented a flash SSD latency optimization technique and reviewed the preliminary results toward minimizing the impact of scheduling delays. Our opti-

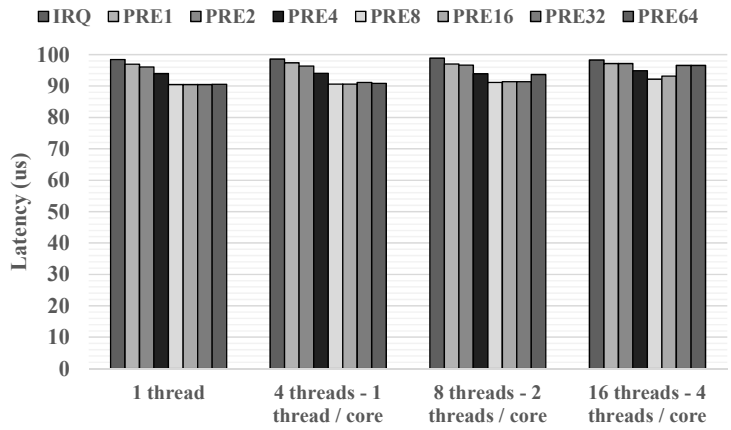


Figure 4.6: Impact of precompletion on latency with multidepth I/O (80 µs device latency)

mization exploits accurate latency predictions that were enabled by tracking behavioral parameters within the SSD. These predictions are based on simplified SSD behavioral models, which are agreed between the OS and the SSD a priori. The accuracy of such predictions is backed with the help of SSDs actively filling in the crucial parameters required for the models.

This was based on the insight that it is far easier to predict the behavior of individual components within an SSD than to predict the behavior of multiple components (the SSD as a whole) as a system. Based on this preliminary work, in future work we plan to evaluate the impact of our optimization on a full-featured SSD under high parallel workloads.

Chapter 5

Tail Latency Reduction using Host Side GC Control and Multiple Devices

5.1 Introduction

In order to support large scale high performance data intensive applications, key value stores have been gaining great interest from both industry and academia. To support these workloads, key value stores rely on in-memory architectures which has been proliferated due to the dropping price per performance of DRAM. However, the increasing scale of these applications pushes in-memory architectures to its physical limits. The limits, in which DRAM is both space and power hungry, can limit the scalability of these architectures in large deployments.

To solve the problem, flash based solid-state drives (SSDs) have become more viable as an alternative. With access latencies far lower than HDDs and being more accessible, flash SSDs are now being employed into various system architectures to fill the gap between DRAM and HDD. Several research explores the use of flash SSDs [7, 86, 87, 88, 89, 90, 91] as the main means of data storage for key value stores.

However, flash based architectures are yet to dominate applications such as key value stores. This is because of the high latency variablility of SSDs caused by internal

resource conflicts on flash channels, chips, dies and planes. Even though flash SSDs are known for their competitive read latencies (50us to 150us), conflicting write operations and erase operations can cause high spikes of uncontrollable latencies. Both triggered by either foreground I/O or background SSD internal operations, these spikes are in orders of milliseconds which can be unacceptable.

To deal with variance in latency, we present a host and SSD cooperative approach, developing a host side storage engine capable of cooperating with SSDs. The storage engine exploits the authority and capability of scheduling regular I/O and SSD internal operations to avoid resource conflicts causing unpredictable latency spikes. The capability of SSD internal control was provided by the SSDs enhanced with a proprietary SSD API [92]. While the SSD H/W architecture remains the same with off-the-shelf SATA 6.0 Gb/s MLC SSDs, the SSD API comes with modest extensions only in the firmware. This SSD API provides the means of triggering GC operations or suppressing all internal activities (i.e., static wear leveling, write flushes) without excessively exposing proprietary details of the SSD itself.

Yet, even with the total control of SSD internal operations, it is very difficult to find application idle periods to schedule such operations when we consider classes of applications serving external requests. Here, we adopted the scheme presented in [55] which exploits redundancy provided by replicated data placed on multiple instances of SSDs. This scheme places potentially conflicting requests (i.e., read and writes) on distinct hardware resources (SSDs) to resolve resource conflicts at the cost of additional flash chips. The storage engine issues I/O operations to replicas of data blocks spanned on distinct physical SSDs, having latency sensitive operations, such as reads, issued on one replica while other latency heavy background operations, such as GC and flushes, triggered on the other. When latency sensitive operations are served, all internal SSD operations are suppressed with the help of the SSD API. While separating reads and writes [55] significantly reduces the tail latency already, our storage engine goes further eliminating the effects of SSD internal operations.

Four SATA 6.0 Gb/s SSDs enhanced with the SSD API [92] were used in our

evaluations. With micro-benchmarks, results show that our storage engine is capable of cutting the 99.9999th percentile latency of flash SSDs, from 19ms down to 520us (38 times). Also, having our storage engine integrated into memcached as a multi-SSD backend replacing the in-memory hash table, 4.5 times reduction of latency spikes at the 99th percentile was achieved.

5.2 Motivation

5.2.1 Large Scale Key Value Stores and Flash SSDs

Since flash technology have read latency smaller than write latency in an order of magnitude, the read intensive workloads of key value stores fit particularly well with flash technology. According to a recent study by Facebook [9], key value workloads are read heavy and dominated by small sized key value pairs. Read write (`get():set()`) ratio is generally 30:1 with most keys less than 32 bytes and most values less than 500 bytes. With the feature of low power and high density, this read optimized characteristic is also another reason why flash technology is considered as an alternative. Yet, there are concerns because flash latency is an order of magnitude larger than DRAM even for reads.

Fortunately, flash technology in forms of SSDs have high structural parallelism. Flash latency, which is inferior than DRAM, can be compensated by the value of having multiple operations in-flight. Data from I/O requests are interleaved or spread on multiple flash chips inside SSDs. Multiple flash chips can serve several small I/O requests in parallel. Because of this, modern SSD I/O interfaces, such as SATA 3.0 and NVM-Express, introduce queues to serve multiple in-flight I/O requests for maximum performance. The value of overall throughput due to the higher parallelism gives system architects an option of trading latency with higher throughput and capacity (compared to DRAM).

5.2.2 Latency Spikes of Flash SSDs

However, the main hurdle of using flash SSDs is the uncontrollable latency spikes caused by SSD internal resource conflicts. Since flash SSDs have limited knowledge and control on applications, ill data placements can occur where hot spots on SSD internal resources (i.e., channels) introduce significant resource conflicts. Here, read, write and block erase operations each having different levels of latencies combined with arbritrary input of mixed reads and writes can cause unpredictable latency spikes when conflicts occur. Randomly (from the perspective of applications) scheduled SSD internal operations such as garbage collection worsen the situation. Front end application performance are affected by these latency spikes, where these latencies can be up to a few milliseconds.

The negative impact of latency variance experienced from low level system components has been presented by Google [12]. Long tail latency distributions can introduce huge performance penalties even if less than 1% of requests are outliers. Highly parallel, high fanout design of modern data center applications are heavily affected by these outliers at the service level [12]. Here, flash SSDs are considered one of the sources causing latency variance. While techniques of long tail immune message delivery and processing were mostly discussed by the authors, cutting the long tail of individual systems is also shown to provide significant performance benefits to the service. This partially indicates why DRAM based in-memory systems are widely adopted to serve latency sensitive data center applications. Flash SSDs, if used naively, are not even close to have less than 1% of latency outliers (shown in Section 5.4).

5.2.3 Predictable latency, High IOPS but Higher Price

In this work, we were motivated to explore the use of multiple low cost storage components to build a competitive storage system. To make flash technology more attractive, eliminating or at least mitigating the unpredictable latency spikes of flash SSDs have been a high priority agenda for both the academia and the industry. Significant amount of effort have been put into development of SSD controllers to control the latency

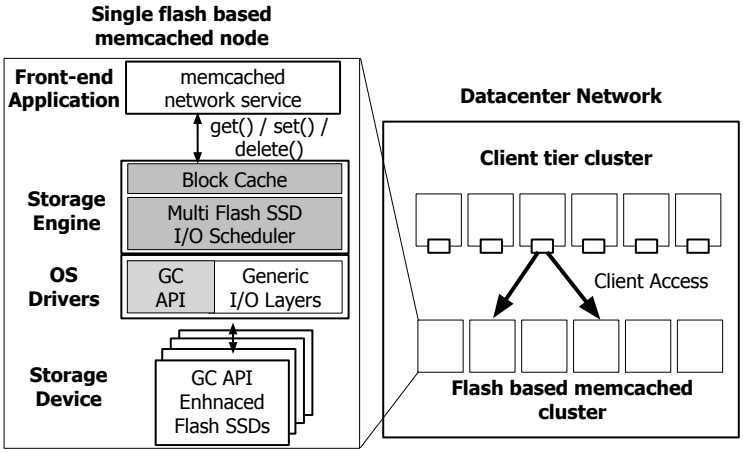


Figure 5.1: Overview of our multi-SSD storage engine integrated as a flash SSD back-end for memcached

spikes, resulting in a more sophisticated design with higher price tags. As a result, to have more storage IOPS (I/O Per Second) at predictable levels of latency with SSDs, one should spend up to multi-thousand dollars to purchase even a single device (i.e., PCI-e based high-end SSDs). This motivates us to look for alternatives, which can provide us reasonably predictable latency and higher IOPS at lower costs. Here, we intend to fill this gap by proposing software methods which exploit multiple low cost SSDs to control the latency predictability of SSDs.

5.3 Design

5.3.1 Overview

At the core of our work, we present a host side storage engine which is developed as a shared library object which applications can integrate the engine's functionality via simple API calls. The purpose of this storage engine is to conceal the complexities of I/O scheduling and SSD internal control into a single software component. Figure 5.1 shows an example of our storage engine integrated to memcached as a flash SSD back-end, which is also used in our evaluations in Section 5.4.

Table 5.1: SSD Control API

Type	Function	Description
Basic	`get_user/device_capacity(device)`	Queries the capacity of the device and the user visible capacity (result of over-provisioning)
	`get_eb_size(device)`	Queries the erase block size of the SSD
	`get_free_block_count(device)`	Queries the amount of erase blocks left in the SSD
Garbage Collection	`get/set_gc_threshold(device, threshold)`	Set / get GC thresholds which controls the behavior of SSD initiated GC
	`is_gc_running(device)`	Determines whether a GC activity is currently underway
	`run_gc(device, time)`	An asynchronous function call which triggers the garbage collection activity to run for a fixed amount of time
Optimized Read	`is_ready_for_read(device)`	Determines whether the SSD is in a read-only ready state

5.3.2 SSD Control API

While normal I/O requests are issued along a regular OS I/O path, SSD control related operations are issued through a proprietary library, provided as a shared library object (Figure 5.1). With the API, host software can trigger GC operations or suppress all internal operations for a certain period. The API is provided by the SSD firmware which was enhanced to accept commands from the library [92]. This library exploits vendor specific extensions defined in the ATA command specification, to communicate with the SSDs. All the other SSD H/W resources remain the same. Table 5.1 outlines the API provided by the this library.

Threshold Based GC

While this API aims to provide control to the host, there are certain situations where the SSD should trigger mandatory internal operations. For GC, two thresholds, *low block watermark* and *high block watermark*, are set and used to control the behavior. When the level of free blocks goes below the *low block watermark* threshold, then GC automatically starts until it restores the level of *low block watermark* again. The *high block watermark* threshold limits the amount of free blocks being generated by host initiated GC operations. These thresholds can be adjusted by the application using the `set_gc_threshold()` API call.

Host Initiated GC

When the host initiates GC activity, it is triggered with a `run_gc()` call with a time period provided. The device executes GC activities until the time expires, or until enough (*high block watermark*) free blocks are produced. Any `run_gc()` call is ignored by the device, when the level of free blocks exceeds the *high block watermark* threshold. This `run_gc()` call is an asynchronous operation which returns to the user right after it triggers a GC activity. For this reason, the activity of GC should be monitored by API calls such as `is_gc_running()` and `get_free_block_count()`.

Optimized Read

As a byproduct of the host controlled GC scheduling scheme, the device provides a period where no SSD internal operations are being performed. In this period, read operations are guaranteed to have no interference with SSD internal operations. All internal operations are suppressed not to have any interference with the foreground operations. This period can be triggered and acknowledged by the user program with the `is_ready_for_read()` call. Upon the first function call, the SSD preempts pending GC or writes and returns 0 if the preemption is not yet finished. Subsequent calls with a non-zero value indicates that there will be no pending GC or write which can interfere with the subsequent read operations. This period lasts until the SSD receives a new write operation or a GC triggering event occurs (both upon a `run_gc()` call or crossing the *low block watermark* threshold).

5.3.3 Multi-SSD Storage Engine

The storage engine itself is divided into two parts: the block cache and the I/O scheduler. This is equivalent to the OS I/O path with the buffer cache and the block I/O scheduler below without file systems and RAID components.

Block Cache

The block cache is the equivalent of the OS buffer cache, but is not associated with a file system. The block cache faces the application with an API in front (i.e., `get()/set()`, and it caches or mediates I/O requests to the I/O scheduler. A unified key value address space is provided to the application, backed with multiple SSDs and a relatively small DRAM cache in front. This cache implements a delayed write semantic, in which the write storage I/O latency is decoupled from the foreground update operation. While flash SSD read I/O has direct impact on foreground latency upon cache misses, flash SSD write I/O does not have direct impact on foreground update latency, unless the cache is full with dirty blocks. Here, we focus to minimize the impact of GC operations interfering especially with the foreground operations.

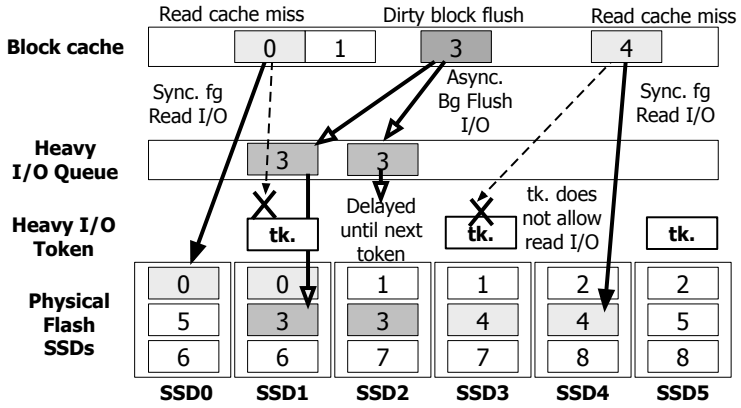


Figure 5.2: Latency control scheme which employs a heavy I/O token being passed around multiple SSDs

I/O Scheduler

The I/O scheduler is the equivalent of the block I/O scheduler, but the scheduler has central control of scheduling I/O among multiple SSDs. The scheduler can coordinate I/O requests taking multiple SSDs into account. Here, the scheduler maintains a striped replica data layout similar to RAID1+0 and exploits the layout to avoid collisions between latency heavy background I/O operations and latency sensitive foreground I/O operations (Figure 5.2). As seen in Figure 5.2, data blocks are replicated among multiple SSDs having each replica located on different physical SSDs. Here, the I/O scheduler schedules latency heavy I/O operations on one replica, and latency sensitive I/O operations on the other replica; assuring these I/O operations do not interfere each other.

5.3.4 Latency Control

Latency Control via Multi-SSDs

The I/O scheduler controls SSD I/O latencies by introducing heavy I/O tokens and epochs assigned to a subgroup of SSDs in the array (Figure 5.2). Heavy I/O tokens represent the opportunity for an SSD to perform latency heavy I/O operations, whereas epochs represent the period of time where the SSD receives a token. When an SSD

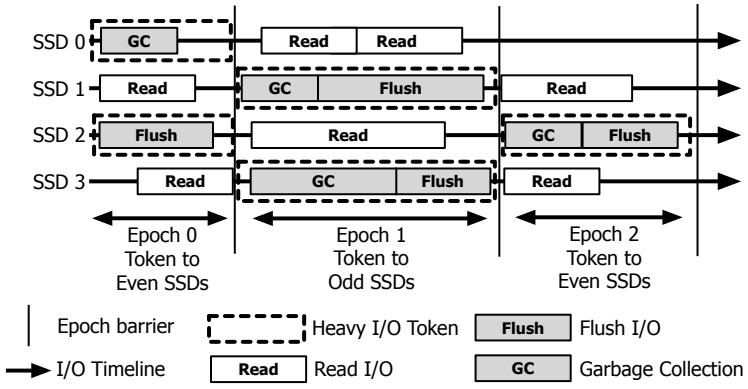


Figure 5.3: Example timeline I/O being scheduled on multiple SSDs

receives a heavy I/O token, latency heavy I/O operations, such as flushes (block cache) or GC operations, are allowed to be performed on the SSD. For example, the flush operation of block 3 of SSD1 is allowed to be performed since SSD1 has a token (tk.). The flush operation of block 3 (since blocks have replicas) on SSD3 is postponed until SSD3 receives a token. Read I/O, which are issued from read cache misses, are diverted to the replicas placed on SSDs which are free from heavy I/O tokens (block 0 and block 4 in Figure 5.2).

Token Assignment

To make sure at least one data block replica is available to serve latency sensitive operations, token assignment is aligned to epoch barriers (Figure 5.3). Also, considering the data layout depicted in Figure 5.2, our scheduler repeatedly reassigns tokens to odd index SSDs or to even index SSDs on each epoch barrier in turns (Figure 5.3). The length of each epoch is determined by the amount of work to be done with the token.

5.3.5 Scheduling I/O Operations

Scheduling GC and Writes

Heavy I/O operations, GC and writes in this work, are scheduled on a token assigned SSD until the epoch ends (token epoch). As seen in Figure 5.3, GC related opera-

tions are issued before the write operations. The scheduler tries to make sure there are enough free blocks to accept the amount of write operations during the epoch. This is done by adjusting the *max free blocks* threshold to be just enough to accept the amount of blocks which could be written during a write epoch. When the level of free blocks is lower than the threshold, GC activity is triggered via the `run_gc()` API call before writes.

Optimized Reads

After all write operations are performed within the token epoch, the scheduler waits for an `is_ready_for_read()` indication and finalizes the epoch to make sure the device is ready for optimized reads. This is to make sure the read operations performed on SSDs are not interfered with write operations or GC operations.

Managing Replicated Blocks

Since the storage engine manages replicated blocks, there is a subtle issue of preserving data consistency under data updates. Here, we guarantee all updates are immediately visible to other queries. This affects how the I/O scheduler manages I/O. In our implementation, the update is first updated and served from the block cache as an authoritative copy until all replicated blocks are updated on the SSDs. The cached copy lasts until all replicated blocks are guaranteed to serve the same version. The replica updates are queued in per-device queues and are served when the device receives a heavy I/O token.

However, when another update comes to the block cache, the new copy is selected as an authoritative copy, and the corresponding replica updates are queued on corresponding devices. Here, the old version is invalidated and the associated writes in the per-device queue are cancelled, unless they are not pending. All replica updates are queued in a FIFO order, so it is guaranteed that the latest updates are seen. Yet, at events of process failures or power failures, on-storage data can be corrupted since the in-flight replica updates can be lost. This is clearly a limitation of our implementation

which will be discussed in Section 5.5.3.

5.4 Evaluation

5.4.1 Implementation and Environment

Our storage engine was implemented in C language with total 8,421 lines of original code. We conducted our evaluations on a PC with an Intel i7-4770 3.40Ghz hyper-threaded quad core CPU and 16GB DRAM. The system was equipped with four GC API enhanced SATA 6.0 Gb/s SSDs [92] connected to a single AHCI controller. Upon all experiments, all SSDs were preconditioned with a secure erase followed with a pass of sequential writes of random data.

For comparison, we set an array of four SSDs in RAID0 as a baseline, while setting the four SSDs as RAID1+0 for our storage engine. Even though the baseline has more logical space (i.e., RAID0 vs RAID1+0), we used the same number of SSDs for the baseline to provide the same level of storage IOPS and bandwidth support from the array. The RAID style block layouts were both implemented and managed by our storage engine.

5.4.2 Micro Evaluation

To accurately measure the latency of SSDs, we built a small I/O kernel which issues I/O patterns to our multi-SSD I/O storage engine. We used the following configurations for our evaluations: `EPCH` and `+GCCTRL`, compared to the baseline `MIX`. `EPCH` reflects our storage engine performing reads and writes isolated based on epochs and replicas, which was reproducing the I/O scheme presented in [55]. `+GCCTRL` adds GC control via the GC API [92] having `EPCH` implied. Both `EPCH` and `+GCCTRL` performs I/O on four SSDs showing a similar timeline shown in Figure 5.3. For comparison, `MIX` performs reads and writes mixed on the baseline array of four SSDs in RAID0.

All I/O operations were performed having eight 4kB block random reader threads and four 1MB block random writer threads. Lower level parallelism was intended

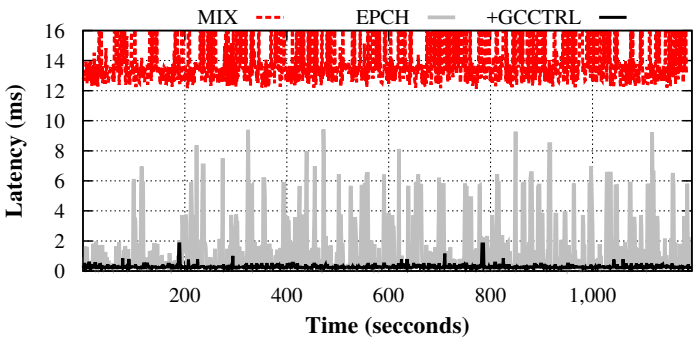


Figure 5.4: Maximum read latency of multiple SSDs, under control of our storage engine, in the presence of write activity (aggregate of four SATA 6.0 Gb/s SSDs, 8 random 4kB readers and 4 random 1MB writers)

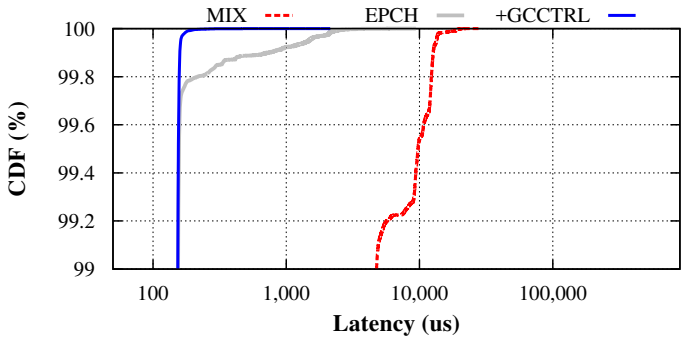


Figure 5.5: Cumulative distribution of the foreground read latencies

to avoid queueing effects on resources other than SSDs (i.e., CPU cores). Here, we mainly show the latency of read operations to reflect the implementation of the block cache which operates similarly to the OS buffer cache. Read operations are sensitive to I/O latency, synchronously performed upon block cache misses, and write operations are sensitive to flush throughput, asynchronously performed by block cache flush workers.

Stable Latency

Figure 5.4 and Figure 5.5 shows the impact of our optimizations on read latency, experienced in front of the SSD array. `MIX` showed significant fluctuations and a long tail due to the mixed reads and writes and the impact of uncontrollable background GC

operations. EPCH added more stability by separating reads and writes, which confirms the effectiveness of the I/O scheme presented in [55] even with different implementations, however it was not enough to cut the latency tail above the 99.7th percentiles. The uncontrollable GC periods leaked out of heavy I/O token epochs and caused latency spikes as high as 8ms–9ms. +GCCTRL was added to limit the GC operations to be performed only in the *background* device, only when desired. +GCCTRL successfully suppresses the latency spikes, achieving sub-milli-second latency and significant decrease of the latency tail above the 99.7th percentile zone. With +GCCTRL, we could see stable latencies at much higher percentiles even as high as the 99.9999th. While the 99.9999th percentile latency of the baseline (MIX) was over 19ms, EPCH was measured at 2,204us. Our work (+GCCTRL) recorded 520us (38 times vs. MIX and 4.2 times vs. EPCH).

Application Latency and Throughput

Figure 5.6 shows the 99.9999th percentile latency and the throughput behavior under increasing application load (reader threads). At the 99.9999th percentile latency, there were significant gaps between each configuration. At 16 threads, the gaps between MIX, EPCH and +GCCTRL were at their largest, both MIX and EPCH ranging in tens of milli-seconds while +GCCTRL remained under a milli-second. From 32 threads, we could see the increase in +GCCTRL latency due to the queuing effect on other shared resources such as CPU cores (i.e., 64 threads on a 8 core CPU machine). The aggregated bandwidth of EPCH and +GCCTRL was far larger than MIX which suggests the advantage of SSDs dedicated to either reads and writes free from conflicts, though there were no significant bandwidth difference between EPCH and +GCCTRL. Considering the DMI 2.0 25Gbps uplink from the PCH to the CPU, measured to be 1.5GB/s, +GCCTRL was able to utilize approx. over two thirds (1.148GB/s, including the write bandwidth) of the H/W bandwidth while maintaining the 99.9999th percentile latency under a milli-second.

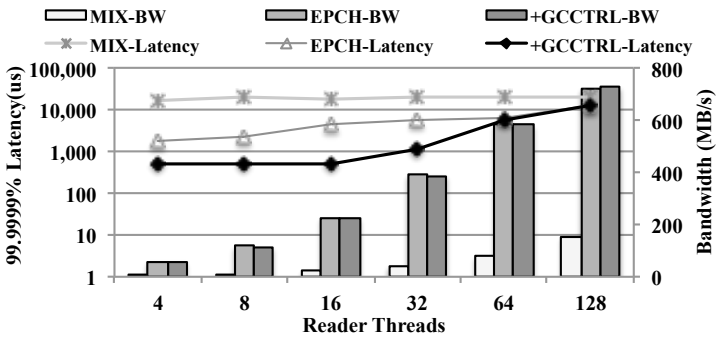


Figure 5.6: Foreground 99.9999th percentile read latency of multiple SSDs and their impact on read bandwidth under varied number of reader threads (aggregate of four SATA 6.0 Gb/s SSDs, 4 random 1MB writers)

5.4.3 Full System Performance

To demonstrate the impact of our GC scheduling scheme on a full system, the storage engine was integrated into memcached 1.4.17 [3] as an item storage backend. YCSB [83] on another identical system, linked back to back through a Quad 1Gbps NIC channel bonded, was used to load our system. The evaluation was focused on comparing the optimizations (`EPCH` and `+GCCTRL`) to `MIX`. Table 5.2 shows the results.

Similar to the trends seen in Figure 5.6, flash SSD latency irregularities can heavily impact on in-memory systems such as memcached. While the baseline flash array `MIX` experiences super milli-seconds of latency spikes above the 90th percentiles, our storage engine (both with `EPCH` and `+GCCTRL`) was able to limit the latency spikes to a manageable level. However, the much higher levels of latencies measured in all levels of percentiles, suggests further optimizations, dealing with software issues such as concurrency, limited CPU parallelism, network I/O bottleneck and shared interrupts. We leave these issues for our future work to improve the performance of flash SSD based in-memory systems.

Table 5.2: Full System Performance (Flash Memcached)

Type	ops/sec	50%	90%	95%	99%
MIX	51,362	696us	3,578us	4,420us	30,090us
EPCH	66,661	451us	791us	1,059us	9,514us
+GCCTRL	76,917	437us	779us	1,099us	6,548us

5.5 Limitations

5.5.1 Additional Flash Chips

The main concern of adopting our storage engine would be trading excessive SSD capacity to enhance the latency stability of flash SSD storage systems. However, our approach is targeted for in-memory systems, where DRAM is dominating. Even with the double price of flash chips (assuming 2 copies), it is cheaper than DRAM in terms of capacity. Also, the value of stable access latency, which enables the possibility of low power / high density in-memory system with flash supplements, exceeds the cost of sacrificing additional flash chips. Similar approaches based on replicas placed on redundant H/W resources can be found in prior work such as [55, 59].

5.5.2 Non Standard API

Non-standard APIs can lead to problems such as vendor lock-ins, non-portable solutions and the cost of modifying previous applications. The API we used is designed to only expose abstractions of GC activity which provides enough control to the host program while hiding SSD proprietary details. Applications can safely use the API even if the actual SSD implementation would differ from vendor to vendor. This design provides a foundation of API standards similar to the TRIM command. Additionally, there are several proposals on new interfaces which would enhance overall efficiency of S/W & H/W interactions, expecting the efficiency to outweigh the cost of extensions [93, 94, 95, 35].

5.5.3 Data Inconsistency upon Power Failures

With our storage engine, data corruption can occur upon power failures, since the engine does not have a recovery mechanism. Upon failures, there is no way to guarantee that the current on-storage data is consistent. Currently, our storage engine relies on a stable shutdown of the storage engine, which waits for all replica updates to be applied to the SSDs, reaching a consistent state. We aim to address this issue in our future research.

5.5.4 Unbounded Write Latency

While the storage engine guarantees stable read latency, our implementation is not capable of bounding or stabilizing write latency. When the block cache has enough space left, this would not be a problem, since the block cache would serve DRAM latency. But if the block cache is full under a heavy write load, then the latency is directly affected by the speed of free blocks being generated from invalidating previous blocks. Since the speed of these invalidations are, at the core, bounded by the speed of flushing the blocks (replica updates), foreground write operations at the block cache are directly affected by the latency of flushing a block, when the flush speed cannot catch up with the incoming updates. There should be either a way to limit the incoming writes (write throttling) or a way to speed up the flush speed. We leave this issue to be addressed in our future research.

5.6 Summary

While there are GC optimization approaches in various levels of the I/O path, from the application and down to the SSD controller, the difficulty of predicting workload idle periods limits these approaches from fundamentally eliminating foreground latency spikes.

In this work, we have implemented a storage engine which exploits explicit GC control provided by SSDs and the redundancy of data block replicas spanned on multiple SSDs. Our storage engine can reserve device-wise idle periods for GC, while

guaranteeing the availability of conflict free I/O paths for foreground I/O operations at all times. We were able to build a low latency storage system with predictable latencies, enabling flash technology to be used in latency sensitive environments such as in-memory systems. We demonstrated this by integrating our storage engine into memcached, a popular in-memory system used in data centers, showing that flash technology can take place of DRAM providing the benefits of low power and high density with stable latency.

There are future research which can be drawn from our work. In terms of QoS, it would be interesting to integrate our GC scheduling scheme with queue based I/O schedulers, such as [51], since our work can enhance the accuracy of such methods by providing a more predictable I/O model. Also, it would be worth exploring additional tradeoffs, in order to reduce the space overhead of replicas. Additionally, implementing our GC scheduling scheme in a single SSD by exploiting SSD internal parallelism or redundancy of H/W resources would be interesting.

Chapter 6

Conclusion

6.1 Summary and Conclusions

Under the increasing demands of high rates of low latency access towards at scale, our data centers aggressively use DRAM based in-memory systems, especially under latency sensitive requirements. However, DRAM is both space and power hungry so it can run short in supporting the ever increasing scale of data. Here, flash SSDs have been gaining interest due to the affordable microsecond latency based on the character of low power, high density of NAND flash technology with the expectation of replacing the role of DRAM.

However, flash SSDs are yet favorable for such latency-sensitive environments due to the high latency variance of flash SSDs. While the latency of flash SSDs are small enough to make software overheads of an I/O request non-negligible, latency variance increases the overheads by increasing the context multiplexing cost. Also, the latency variance exposed all the way up to the applications to have an adverse impact on overall throughput of data center applications. Such variance has to be tolerated or blocked within the I/O path when using flash SSDs in place of DRAM under latency-sensitive environments. It is challenging to do so with the limited amount of cycles per

CPU cores in the era of post-Moore. New I/O path designs are required to overcome such challenges.

In this dissertation, we have proposed a series of OS I/O path optimizations which address the impact of latency variance of flash SSDs with the aim of using flash SSDs as DRAM alternatives in our latency-sensitive data centers. Here, we set our goals to 1) tolerate the impact of the variance from the I/O path, and 2) to conceal the impact of variance at the I/O path. Our goals were achieved by presenting new I/O path designs which seek to exploit the parallel hardware resources such as multiple CPU cores, multiple SSDs or NAND chips, and also access to abstracted resources and information inside SSDs. Such exploits are coordinated to cope with the latency variance of flash SSDs. To make the case of our new I/O path designs and evaluate the impact of our optimizations, we built two separate I/O path supporting software which addresses each goal.

First, we optimized the magnified impact of context switches by introducing an optimized device driver for flash SSDs. The device driver exploited the parallelism of multiple cores to accelerate I/O completion processing and enhanced IOPS scalability of the Linux I/O path for flash SSDs. Further extensions in hiding the impact of context switches from foreground applications even under low loads were made by exploiting completion events monitored inside the SSD and exposed to the device driver. This enhancement masked the impact of such context switches from the foreground application without affecting both system parallelism and the latency itself.

Second, we optimized the foreground read I/O latency of flash SSDs by building a key-value storage engine backend. The storage engine aimed to conceal the latency spikes by explicitly controlling each I/O requests to meet the latency requirements without sacrificing average latency. On each parallel hardware (SSD), low latency read requests and high latency write oriented requests were temporally separated by exploiting redundant copies of data placed on multiple SSDs. Further optimizations by exploiting a host-controlled garbage collection SSD extension enabled the storage engine to achieve significant latency stability (under 500 µs) even in higher percentiles

(i.e., 99.9999th percentiles).

6.2 Future Work

6.2.1 Extending the Scope of I/O Path Optimizations

The I/O path optimization techniques presented were limited to I/O path related to flash SSDs. However, there are other portions of the I/O path which involves end-to-end performance which is not flash SSD specific. The system software for flash SSD specific I/O paths was heavily optimized, though the overheads in the other software paths were extensive. These overheads have been noticed during the implementation the key-value storage engine we proposed. While the primary purpose of the key-value storage engine was to prohibit the latency spikes from leaking towards the application, the average latency has increased merely due to the increase of CPU cycles used in the I/O path without any contribution from sophisticated latency controls such as read prioritization and preemptive background operations. The results were rather surprising since the role of the storage engine was simple I/O routing and switching tasks without relying on long loops or algorithms.

Such result opens up a future research towards performing a holistic end-to-end I/O path optimization towards reducing the software overheads which spans through 1) the network I/O code, 2) the application code, and 3) flash SSD I/O code. Providing such optimizations have several technical and architectural challenges since the optimizations are not simply a matter of combining previous effort of three independent component level optimizations. For example, having two I/O code from the network and the flash SSD would have redundant I/O handling code involving DMA mapping, interrupt handling and device control which requires special interest. The impact of such redundant operations can be significant considering the microsecond scaled latencies of flash SSDs. Also, the semantics of the data service routines from applications should play in harmony with such optimizations.

Here, an interesting research would be figuring out how to present a new I/O stack to the users, which is capable of performing optimizations across boundaries. While

the lack of CPU cycles per core in our post-Moore's CPUs demands such optimizations, such cross-boundary optimizations need a way to be rationalized under the criticism of being ad-hoc, unconventional, or unorthodox.

6.2.2 Extended Use Cases of Host Assisting SSD Extensions

The SSD extensions presented in this dissertation focus on enhancing the behavior of host side system software. Such focus contrasts from many of the prior cross-layer SSD optimizations which improve the behavior of the SSDs under challenging workloads. It is necessary to optimize the behavior of host side system software due to the limits of cycles per CPU core with our post-Moore's CPUs. Since the advance in speed has been stagnated at the CPU, the advance in memory or storage technologies is forming technical challenges towards the host. The relative scarcity of cycles per CPU core leaves the host side software to behave well about the usage of CPU resources.

With modern CPUs, the character of underlying CPU resources such as cores, caches, I/O ports, memory distance (i.e., NUMA), and power consumption are no longer transparent to the host side software and has to be managed efficiently. While such efficient usage of CPU resources are well studied in the context of the standalone CPUs, the narrowed gap between CPUs and storage devices opens a new dimension where we should ask how the CPUs and storage devices should be coordinated. As demonstrated in this dissertation, a cooperative coordination of CPU and storage resource usage can be more efficient.

This coordination aligns with the movement of end-to-end vertical optimizations done in our data centers. In these environments, each components within the critical service path should cooperate well since the best behavior of all individual components does not necessarily translate into the most efficient overall system. While having an optimized software makes a huge difference in many aspects of the data center, engineers are seeking tools and techniques to gain maximum efficiency out of the available resources. Our approach can be used to achieve such efficiency.

6.2.3 Applications for Different Technologies

Another direction of future research is towards addressing the impact of latency variance in the context of other upcoming storage technologies. With the success of multiple generations of NAND flash storage devices, the industry is now moving towards perfecting the ecosystem around NAND flash technology. Also, the industry is advancing towards the next generation memory technologies which are expected to give us better characteristics than NAND flash.

For NAND flash, 3D NAND technology is giving us better characteristics regarding performance, density, and reliability. With the 3rd generation of PCI-e being mainstream, the standard interface towards flash SSDs has been advanced to a more NVM specific standard (i.e., NVM-e). And, new form factors such as NVMDIMM are gaining momentum based on the idea of using NAND flash as memory alternatives. For the next memory technology, in the midst of several candidates (i.e., PCM, RRAM, STT-MRAM), vendors such as Intel and Micron recently announced the imminent arrival of 3D cross-point memory which is believed to be a form of PCM, meaning that such technologies are no longer a distant future.

Here, the variance tolerating optimization techniques proposed in this dissertation will gain more relevance with the advances in technologies stated above. Such relevance is because the same trend of the latency variance and its impact can be found in these technologies. The actual occurrence of the impact of such trend will differ from flash SSDs from now, but the trends remain the same. Software overheads will increase with better memory technologies, and the impact of latency variance will still be there because of the resistive non-volatile characteristics of the candidate memory technologies.

For example, PCM promises nanosecond read latencies but has a much higher write latency which can introduce latency spikes. Such latency spikes can leak beyond the boundaries of micro-architecture components and can impact service level user experience under latency sensitive applications. While the latency variance of flash

SSDs had an impact on increasing the context multiplexing costs of the CPU, different costs such as cache utilization, power consumption, or remote memory access. Future research on understanding and optimizing the impact of such latency variations regarding the effectiveness of CPU resource usage by the system software will be valuable.

Bibliography

[1] M. Jung and M. Kandemir, "Revisiting Widely Held SSD Expectations and Rethinking System-Level Implications," in *Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2013.

[2] M. Bjørling, P. Bonnet, L. Bouganim, and N. Dayan, "The Necessary Death of the Block Device Interface," in *Proceedings of the 6th Biennial Conference on Innovative Data Systems Research (CIDR)*, 2013.

[3] "Memcached. a distributed memory object caching system." `http://www.memcached.org`, 2011.

[4] "Redis." `http://www.memcached.org`.

[5] "Mongodb." `http://mongodb.com`.

[6] "Voltdb. voltdb, the newsql database for high velocity applications.." `http://voltdb.com`.

[7] B. Debnath, S. Sengupta, and J. Li, "FlashStore: High Throughput Persistent Key-Value Store," *Proceedings of the VLDB Endowment*, vol. 3, pp. 1414–1425, Sep 2010.

[8] "Aerospike. aerospike2 free community version.." `http://www.aerospike.com`.

[9] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload Analysis of a Large-Scale Key-Value Store," in *Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2012.

[10] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani, "Scaling Memcache at Facebook," in *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2013.

[11] J. Yang, D. B. Minturn, and F. Hady, "When Poll is Better than Interrupt," in *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST)*, 2012.

[12] J. Dean and L. A. Barroso, "The Tail at Scale," *Communications of the ACM*, vol. 56, p. 74, Feb 2013.

[13] A. M. Caulfield and S. Swanson, "Refactor, Reduce, Recycle: Restructuring the I/O Stack for the Future of Storage," *Computer*, vol. 46, pp. 52–59, Aug 2013.

[14] A. M. Caulfield, A. De, J. Coburn, T. I. Mollow, R. K. Gupta, and S. Swanson, "Moneta: A High-Performance Storage Array Architecture for Next-Generation, Non-volatile Memories," in *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2010.

[15] L. M. Grupp, J. D. Davis, and S. Swanson, "The Bleak Future of NAND Flash Memory," in *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST)*, 2012.

[16] J.-U. Kang, J.-S. Kim, C. Park, H. Park, and J. Lee, "A Multi-Channel Architecture for High-Performance NAND Flash-Based Storage System," *Journal of Systems Architecture*, vol. 53, pp. 644–658, Sep 2007.

[17] A. Akel, A. M. Caulfield, T. I. Mollov, R. K. Gupta, and S. Swanson, "Onyx: a Protoype Phase Change Memory Storage Array," in *Proceedings of the 3rd USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, 2011.

[18] A. M. Caulfield, T. I. Mollov, L. A. Eisner, A. De, J. Coburn, and S. Swanson, "Providing Safe, User Space Access to Fast, Solid State Disks," in *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012.

[19] D. Vučinić, Q. Wang, C. Guyot, R. Mateescu, F. Blagojević, L. Franca-neto, D. L. Moal, H. San, T. Bunker, J. Xu, S. Swanson, and Z. Bandić, "DC Express : Shortest Latency Protocol for Reading Phase Change Memory over PCI Express," in *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST)*, 2014.

[20] E. Seppanen, M. T. O'Keefe, and D. J. Lilja, "High Performance Solid State Storage Under Linux," in *Proceedings of the 26th IEEE Symposium on Mass Storage Systems and Technologies (MSST)*, 2010.

[21] A. Foong, B. Veal, and F. Hady, "Towards SSD-Ready Enterprise Platforms," in *Proceedings of the 1st International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures (ADMS)*, 2010.

[22] M. Bjørling, J. Axboe, D. Nellans, and P. Bonnet, "Linux Block IO: Introducing Multi-Queue SSD Access on Multi-Core Systems," in *Proceedings of the 6th International Systems and Storage Conference on (SYSTOR)*, 2013.

[23] W. Shin, Q. Chen, M. Oh, H. Eom, and H. Y. Yeom, "OS I/O Path Optimizations for Flash Solid-state Drives," in *Proceedings of the 2014 USENIX Annual Technical Conference (ATC)*, 2014.

[24] M. Wei, M. Bjørling, P. Bonnet, and S. Swanson, "I/O Speculation for the Microsecond Era," in *Proceedings of the 2014 USENIX Annual Technical Conference (ATC)*, 2014.

[25] G. R. Ganger, "Blurring the Line Between OSes and Storage Devices," tech. rep., Carnegie Mellon University, Pittsburgh, PA, USA, 2001.

[26] A. Trivedi, P. Stuedi, B. Metzler, R. Pletka, B. G. Fitch, and T. R. Gross, "Unified High-performance I/O: One Stack to Rule Them All," in *Proceedings of the 14th USENIX Conference on Hot Topics in Operating Systems (HotOS)*, 2013.

[27] M. Jung, E. H. Wilson, W. Choi, J. Shalf, H. M. Aktulga, C. Yang, E. Saule, U. V. Catalyurek, and M. Kandemir, "Exploring the Future of Out-of-Core Computing with Compute-Local Non-Volatile Memory," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2013.

[28] M. Wei, J. D. Davis, T. Wobber, M. Balakrishnan, and D. Malkhi, "Beyond block I/O: Implementing a Distributed Shared Log in Hardware," in *Proceedings of the 6th International Systems and Storage Conference (SYSTOR)*, 2013.

[29] M. Jung, "Exploring Design Challenges in Getting Solid State Drives Closer to CPU," *IEEE Transactions on Computers*, vol. 65, pp. 1103–1115, Apr 2014.

[30] J. Yang, N. Plasson, G. Gillis, N. Talagala, and S. Sundararaman, "Don't Stack Your Log on My Log," in *Proceedings of 2nd Workshop on Interactions of NVM/Flash with Operating Systems and Workloads (INFLOW)*, 2014.

[31] S. Yang, T. Harter, N. Agrawal, S. S. Kowsalya, A. Krishnamurthy, S. Al-Kiswany, R. T. Kaushik, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Split-Level I/O Scheduling," in *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, 2015.

[32] S. Lee, J. Kim, and A. Mithal, "Refactored Design of I/O Architecture for Flash Storage," *IEEE Computer Architecture Letters*, vol. 14, pp. 70–74, Jan 2015.

[33] S. Lee, M. Liu, S. Jun, , S. Xu, J. Kim, and Arvind, "Application-Managed Flash," in *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST)*, 2016.

[34] J. Kim, H. Kim, L. Seongjin, and Y. Won, "FTL Design for TRIM Command," in *Proceedings of the 5th International Workshop on Software Support for Portable Storage (IWSSPS)*, 2010.

[35] X. Ouyang, D. Nellans, R. Wipfel, D. Flynn, and D. K. Panda, "Beyond block I/O: Rethinking Traditional Storage Primitives," in *Proceedings of the IEEE 17th International Symposium on High Performance Computer Architecture (HPCA)*, 2011.

[36] J. Coburn, T. Bunker, R. K. Gupta, and S. Swanson, "From ARIES to MARS: Reengineering Transaction Management for Next-Generation, Solid-State Drives," tech. rep., UCSD Computer Science and Engineering, San Diego, CA, USA, 2012.

[37] Y. Yu, D. Shin, W. Shin, and N. Song, "Exploiting peak device throughput from random access workload," in *Proceedings of the 4th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, 2012.

[38] V. Vasudevan, M. Kaminsky, and D. G. Andersen, "Using Vector Interface to Deliver Millions of IOPS from a Networked Key-Value Storage Server," in *Proceedings of the 3rd ACM Symposium on Cloud Computing (SoCC)*, 2012.

[39] M. Balakrishnan, D. Malkhi, V. Prabhakaran, T. Wobber, M. Wei, and J. D. Davis, "CORFU: a Shared Log design for Flash Clusters," in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2012.

[40] V. Chidambaram, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Optimistic Crash Consistency," in *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

[41] M. Saxena, Y. Zhang, M. M. Swift, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Getting Real: Lessons in Transitioning Research Simulations into Hardware Systems," in *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST)*, 2013.

[42] J. Ouyang, S. Lin, S. Jiang, Z. Hou, Y. Wang, and Y. Wang, "SDF: Software-Defined Flash for Web-Scale Internet Storage Systems," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[43] L. Marmol, "NVMKV : A Scalable and Lightweight Flash Aware Key-Value Store," in *Proceedings of the 6th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, 2014.

[44] L. Marmol, S. Sundararaman, N. Talagala, and R. Rangaswami, "NVMKV: A Scalable , Lightweight , FTL-Aware Key-Value Store," in *Proceedings of the 2015 USENIX Annual Technical Conference (ATC)*, 2015.

[45] S. Shakkottai, T. S. Rappaport, and P. Karlsson, "Cross-layer Design for Wireless Networks," *IEEE Communications Magazine*, vol. 41, no. 10, pp. 74–80, 2003.

[46] R. Braden, T. Faber, and M. Handley, "From Protocol Stack to Protocol Heap: Role-Based Architecture," *ACM SIGCOMM Computer Communication Review*, vol. 33, pp. 17–22, Jan 2003.

[47] V. Kawadia and P. Kumar, "A Cautionary Perspective on Cross-Layer Design," *IEEE Wireless Communications*, vol. 12, pp. 3–11, Feb 2005.

[48] M. Van Der Schaar and S. Shankar N, "Cross-Layer Wireless Multimedia Transmission: Challenges, Principles, and New Paradigms," *IEEE Wireless Communications*, vol. 12, pp. 50–58, Aug 2005.

[49] V. Srivastava and M. Motani, "Cross-Layer Design: a Survey and the Road Ahead," *IEEE Communications Magazine*, vol. 43, pp. 112–119, Dec 2005.

[50] G. Vivekananda and P. Reddy, "Critical Analysis of Cross-Layer Approach," in *Proceedings of the 2015 International Conference on Green Computing and Internet of Things (ICGCIoT)*, 2015.

[51] S. Park and K. Shen, "FIOS: A Fair, Efficient Flash I/O Scheduler," in *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST)*, 2012.

[52] M. Jung, E. H. I. Wilson, and M. Kandemir, "Physically Addressed Queueing (PAQ): Improving Parallelism in Solid State Disks," in *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA)*, 2012.

[53] K. Shen and S. Park, "FlashFQ: A Fair Queueing I/O Scheduler for Flash-Based SSDs," in *Proceedings of the 2013 USENIX Annual Technical Conference (ATC)*, 2013.

[54] M. Balakrishnan, A. Zuck, D. Malkhi, T. Wobber, M. Wu, V. Prabhakaran, M. Wei, J. D. Davis, S. Rao, and T. Zou, "Tango: Distributed Data Structures over a Shared Log," in *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

[55] D. Skourtis, D. Achlioptas, N. Watkins, C. Maltzahn, and S. Brandt, "Flash on Rails : Consistent Flash Performance through Redundancy," in *Proceedings of the 2014 USENIX Annual Technical Conference (ATC)*, 2014.

[56] C. Gao, L. Shi, M. Zhao, C. J. Xue, K. Wu, and E. H.-M. Sha, "Exploiting Parallelism in I/O Scheduling for Access Conflict Minimization in Flash-based Solid

State Drives," in *Proceedings of the 30th Symposium on Mass Storage Systems and Technologies (MSST)*, 2014.

[57] W. Shin, M. Kim, J. Choi, H. Eom, and H. Y. Yeom, "HIOPS-KV : Exploiting Multiple Flash Solid-State Drives for Key Value Stores," in *Proceedings of the 20th IEEE International Conference on Parallel and Distributed Systems (ICPADS)*, 2014.

[58] Q. Wei, B. Gong, S. Pathak, B. Veeravalli, L. Zeng, and K. Okada, "WAFTL: A Workload Adaptive Flash Translation Layer with Data Partition," in *Proceedings of the 27th Symposium on Mass Storage Systems and Technologies (MSST)*, 2011.

[59] M. Jung and M. T. Kandemir, "Sprinkler: Maximizing Resource Utilization in Many-Chip Solid State Disks," in *Proceedings of the 20th IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2014.

[60] S.-W. Lee, B. Moon, C. Park, J.-M. Kim, and S.-W. Kim, "A Case for Flash Memory SSD in Enterprise Database Applications," in *Proceedings of the 2008 ACM International Conference on Management of Data (SIGMOD)*, 2008.

[61] M. Jung, W. Choi, and S. Srikantaiah, "HIOS: A Host Interface I/O Scheduler for Solid State Disks," in *Proceedings of the 41st International Symposium on Computer Architecture (ISCA)*, 2014.

[62] W.-H. Lin and L.-P. Chang, "Dual Greedy: Adaptive Garbage Collection for Page-Mapping Solid-State Disks," in *Proceedings of the 2012 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2012.

[63] E. Lee, H. Bahn, and S. H. Noh, "Unioning of the Buffer Cache and Journaling Layers with Non-volatile Memory," in *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST)*, 2013.

[64] D. Park, B. Debnath, and D. H. Du, "A Workload-Aware Adaptive Hybrid Flash Translation Layer with an Efficient Caching Strategy," in *Proceedings of the*

19th Annual International Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS), 2011.

[65] J.-u. Kang, J. Hyun, H. Maeng, and S. Cho, "The Multi-streamed Solid-State Drive," in *Proceedings of the 6th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, 2014.

[66] W. Shin, M. Kim, K. Kim, and H. Y. Yeom, "Providing QoS through Host Controlled Flash SSD Garbage Collection and Multiple SSDs," in *Proceedings of the 2nd IEEE International Conference on Big Data and Smart Computing (BigComp)*, 2015.

[67] Y. Kim, S. Oral, G. M. Shipman, J. Lee, D. A. Dillow, and F. Wang, "Harmonia: A Globally Coordinated Garbage Collector for Arrays of Solid-State Drives," in *Proceedings of the 27th IEEE Symposium on Massive Storage Systems and Technologies (MSST)*, 2011.

[68] W. K. Josephson, L. A. Bongo, K. Li, and D. Flynn, "DFS: A File System for Virtualized Flash Storage," *ACM Transactions on Storage*, vol. 6, Sep 2010.

[69] Y. Zhang, L. P. Arulraj, A. C. Arpaci-dusseau, and R. H. Arpaci-dusseau, "De-Indirection for Flash-Based SSDs with Nameless Writes," in *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST)*, 2012.

[70] Y. Kim, J. Lee, S. Oral, D. A. Dillow, F. Wang, and G. M. Shipman, "Coordinating Garbage Collection for Arrays of Solid-State Drives," *IEEE Transactions on Computers*, vol. 63, pp. 888–901, Apr 2014.

[71] S. Boboila, Y. Kim, S. S. Vazhkudai, P. Desnoyers, and G. M. Shipman, "Active Flash: Out-of-core Data Analytics on Flash Storage," in *Proceedings of the 28th IEEE Symposium on Mass Storage Systems and Technologies (MSST)*, 2012.

[72] J. Ouyang, S. Lin, Z. Hou, P. Wang, Y. Wang, and G. Sun, "Active SSD Design for Energy-Efficiency Improvement of Web-Scale Data Analysis," in *Proceedings of*

the International Symposium on Low Power Electronics and Design (ISLPED), 2013.

[73] S. Cho, C. Park, H. Oh, S. Kim, Y. Yi, and G. R. Ganger, "Active Disk Meets Flash," in *Proceedings of the 27th ACM International Conference on Supercomputing (ICS)*, 2013.

[74] D. Tiwari, S. Boboila, and S. S. Vazhkudai, "Active Flash: Towards Energy-Efficient, In-Situ Data Analytics on Extreme-Scale Machines," in *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST)*, 2013.

[75] V. Prabhakaran, T. L. Rodeheffer, and L. Zhou, "Transactional Flash," in *Proceedings of the 8th USENIX conference on Operating Systems Design and Implementation (OSDI)*, 2008.

[76] W.-h. Kang, S.-w. Lee, B. Moon, G.-H. Oh, and C. Min, "X-FTL: Transactional FTL for SQLite Databases," in *Proceedings of the 2013 ACM International Conference on Management of Data (SIGMOD)*, 2013.

[77] J. Coburn, T. Bunker, M. Schwarz, R. Gupta, and S. Swanson, "From ARIES to MARS," in *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

[78] Y. Lu, J. Shu, J. Guo, S. Li, and O. Mutlu, "LightTx: A Lightweight Transactional Design in Flash-based SSDs to Support Flexible Transactions," in *Proceedings of the 31st IEEE International Conference on Computer Design (ICCD)*, 2013.

[79] W. Shi, D. Wang, Z. Wang, and D. Ju, "Mobius : A High Performance Transactional SSD with Rich Primitives," in *Proceedings of the 30th IEEE International Conference on Massive Storage Systems and Technology (MSST)*, 2014.

[80] J. Axboe, "Io queuing and complete affinity." `http://lwn.net/Articles/268713/`, Feb 2008.

[81] A. Huffman, "NVM Express specification 1.1a." http://www.nvmexpress.org/specifications/, September 2013.

[82] NVM-Express. http://www.nvmexpress.org/resources/linux-driver-information/.

[83] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking Cloud Serving Systems with YCSB," in *Proceedings of the 1st ACM Symposium on Cloud computing (SoCC)*, 2010.

[84] "The OpenSSD Project. Cosmos OpenSSD Platform." http://www.openssd-project.org/wiki/The_OpenSSD_Project.

[85] "SK Hynix / H27QDG8VEBIR-BCB 16GB 2bit MLC chip."

[86] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky, "SILT: A Memory-Efficient, High-Performance Key-Value Store," in *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[87] B. Debnath, S. Sengupta, and J. Li, "SkimpyStash: RAM Space Skimpy Key-Value Store on Flash-based Storage," in *Proceedings of the 2011 ACM International Conference on Management of Data (SIGMOD)*, (New York, New York, USA), 2011.

[88] X. Ouyang, N. S. Islam, R. Rajachandrasekar, J. Jose, M. Luo, H. Wang, and D. K. Panda, "SSD-Assisted Hybrid Memory to Accelerate Memcached over High Performance Networks," in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation (NSDI)*, 2012.

[89] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan, "FAWN: a Fast Array of Wimpy Nodes," in *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, 2009.

[90] T. Kissinger, B. Schlegel, M. Boehm, D. Habich, and W. Lehner, "A High-Throughput In-Memory Index, Durable on Flash-based SSD: Insights Into the Winning Solution of the SIGMOD Programming Contest 2011," *ACM SIGMOD Record*, vol. 41, p. 44, Oct 2012.

[91] A. Anand, C. Muthukrishnan, S. Kappes, A. Akella, and S. Nath, "Cheap and Large CAMs for High Performance Data-Intensive Networked Systems," in *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2010.

[92] "Samsung, GC control API enhanced SSD." MZ7WD960HCGP-000PU 960GB MLC SSD.

[93] J.-U. Kang, J. Hyun, H. Maeng, and S. Cho, "The Multi-Streamed Solid-State Drive," in *Proceedings of the 6th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, 2014.

[94] A. Soga, C. Sun, and K. Takeuchi, "NAND Flash Aware Data Management System for High-Speed SSDs by Garbage Collection Overhead Suppression," in *Proceedings of the 6th IEEE International Memory Workshop (IMW)*, 2014.

[95] T. Frankie, G. Hughes, and K. Kreutz-Delgado, "A Mathematical Model of the Trim Command in NAND-flash SSDs," in *Proceedings of the 50th ACM Annual Southeast Regional Conference (SE)*, 2012.

초 록

플래시 SSD (플래시 솔리드 스테이트 드라이브) 형태의 플래시 메모리 기술은 고대역폭의 저렴한 마이크로 초 수준의 랜덤 액세스 메모리의 가치를 기반으로 이전의 스토리지 기술을 꾸준히 대체하고 있다. 비트 당 비용이 현재 HDD (하드디스크 드라이브)와 근사하나, 보다 우수한 현재의 SSD의 성능은 현재의 스토리지 시스템에서 HDD를 대체하고 있다. 그러나 지연시간이 매우 비균일하기 때문에 플래시 SSD는 접근지연시간에 민감한 응용 프로그램에서 일반적으로 사용되는 DRAM 기반 메모리 내 시스템을 대체하거나 보완 할 우선적인 후보가 되지 못한다.

이러한 비균일성은 접근지연시간에 민감한 응용프로그램의 IOPS와 지연시간 요구사항 모두를 충족시키는 것을 어렵게 한다. 플래시 SSD의 접근지연시간은 I/O 요청의 소프트웨어 비용을 무시못할 수준으로 만들만큼 작지만, 접근지연시간의 비균일성은 문맥전환의 영향을 확대시키며 이에따라 소프트웨어 비용이 증가되어 I/O 경로의 IOPS 및 접근지연시간 능력은 저해될 수 있다. 또한 이러한 플래시 SSD의 접근지연시간 비균일성은 통제되지 않은 채 데이터 센터 수준에까지 노출되어 어플리케이션 서비스의 처리량에 영향을 미친다. 이러한 비균일성의 영향은 I/O 경로 내에서 극복되거나 통제되어야한다.

이러한 기조로 본 논문은 접근지연시간에 민감한 애플리케이션에서 플래시 SSD를 사용한다는 목표로 플래시 SSD의 접근지연시간 비균일성의 영향을 극복하기 위한 호스트 측 운영체제 I/O경로 최적화들을 제시한다. 본 논문에서는 1) 다중 CPU 코어 또는 SSD 내부의 병렬성 기반한 추가 자원 활용 및 2) 호스트와 SSD 간의 향상된 상호 작용 사용을 통해 비균일성의 영향을 극복하는 새로운 I/O 경로 설계가 제안되었다. 이전 연구에서는 IOPS 나 지연 시간 중 하나가 희생되는 한계가 있었지만 본 논문의 I/O 경로 설계는 추가 리소스를 사용하여 IOPS와 대기 시간을 모두 달성했다.

본 논문에서는 비균일성으로 인한 소프트웨어 비용을 줄이기 위해 I/O 완료 경로 내에서 문맥전환의 영향을 줄임으로써 I/O 경로의 IOPS 기능을 향상시킨 최적화

된 AHCI 기반 플래시 SSD 장치 드라이버를 구현했으며, 기존 Linux I/O경로보다 100%의 IOPS향상을 달성하였다. 더불어, 실제 SSD프로토타입 플랫폼에서 SSD기능의 확장을 통해 I/O로 인한 대기시간과 문맥전환의 스케쥴링 지연시간이 동시에 겹쳐질 수 있도록 하였으며, 이를 통해 호스트 시스템의 병렬성을 해치지 않으면서도 I/O요청당 평균 대기시간을 7 마이크로초 감소시킬 수 있었다.

또한, 비균일성의 영향이 응용으로 노출되는 문제를 해결하기 위해 Memcached의 플래시 SSD 백엔드로 사용하는 키밸류 스토리지 엔진이 개발되었으며, 여러 SSD에 중복 데이터의 복사본을 둔 배치를 활용한 전경 읽기 작업과 배경 쓰기 작업의 분리 기법을 통해 쓰기 위주 연산시 발생하는 접근지연시간 스파이크의 부정적인 영향을 격리시켜 키 밸류 저장 응용 수준에서의 롱테일 지연시간을 밀리 초 수준으로 줄일 수 있었다. 더 나아가, 가비지 수집과 같은 SSD내부의 I/O작업을 제어할 수 있는 SSD 확장기능을 활용하여 이 밀리 초 수준의 롱 테일 접근지연시간을 99.9999th 퍼센타일에서 1밀리 초 미만으로 낮출 수 있었다.

주요어: 스토리지 스택 최적화, 플래시SSD, 운영체제, 교차계층 최적화, 고성능 저장장치, 데이터센터, 키밸류 스토리지, 서비스품질
학번: 2010-23271