Received November 19, 2019, accepted December 10, 2019, date of publication December 30, 2019,
date of current version January 8, 2020.

Digital Object Identifier 10.1109/ACCESS.2019.2963081

I/O Schedulers for Proportionality and Stability on Flash-Based SSDs in Multi-Tenant Environments

JAEHO KIM [1], EUNJAE LEE [2], AND SAM H. NOH [2], (Senior Member, IEEE)
[1]Department of Electrical and Computer Engineering, Virginia Tech, Blacksburg, VA 24061, USA
[2]School of Electrical and Computer Engineering, Ulsan National Institute of Science and Technology (UNIST), Ulsan 44919, South Korea

Corresponding author: Sam H. Noh (samhnoh@unist.ac.kr)

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea Government (MSIT) under Grant NRF-2019R1A2C2009476.

ABSTRACT The use of flash based Solid State Drives (SSDs) has expanded rapidly into the cloud computing environment. In cloud computing, ensuring the service level objective (SLO) of each server is the major criterion in designing a system. In particular, eliminating performance interference among virtual machines (VMs) on shared storage is a key challenge. However, studies on SSD performance to guarantee SLO in such environments are limited. In this paper, we present analysis of I/O behavior for a shared SSD as storage in terms of proportionality and stability. We show that performance SLOs of SSD based storage systems being shared by VMs or tasks are not satisfactory. We present and analyze the reasons behind the unexpected behavior through examining the components of SSDs such as channels, DRAM buffer, and Native Command Queuing (NCQ). We introduce two novel SSD-aware host level I/O schedulers on Linux, called A+CFQ and H+BFQ, based on our analysis and findings. Through experiments on Linux, we analyze I/O proportionality and stability in multi-tenant environments. In addition, through experiments using real workloads, we analyze the performance interference between workloads on a shared SSD. We then show that the proposed I/O schedulers almost eliminate the interference effect seen in CFQ and BFQ, while still providing I/O proportionality and stability for various I/O weighted scenarios.

INDEX TERMS Flash memory based SSDs, cloud computing, virtual machines, I/O schedulers, I/O performance.

I. INTRODUCTION

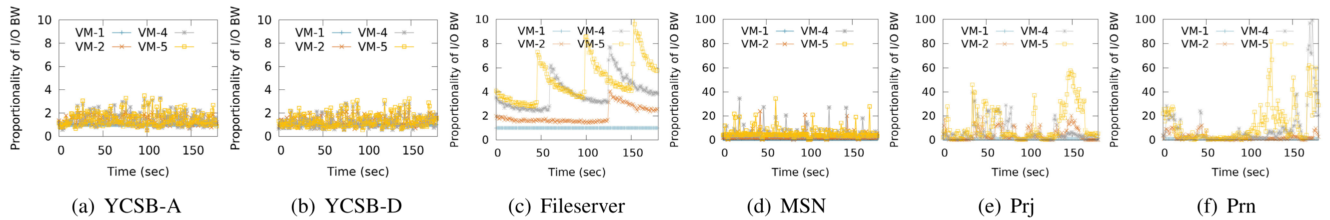
Flash memory semiconductors are increasingly being used as storage devices in smart devices, consumer electronics, and various computing platforms [1]. Recently, the growth of new industries such as Internet of Things (IoT), autonomous vehicles, big-data analysis and cloud computing is expected to further increase the use of flash memory [2]–[7].

Although flash memory based storage devices have various advantages such as high performance and low power consumption, performance variation may occur due to physical characteristics of storage media, sophisticated internal mechanisms and complicated management software [8]–[11]. Therefore, in areas where stable and consistent I/O performance has a critical impact on the system, various analysis and verification of the use of flash memory are needed [8], [12], [13]. Although the use of flash memory based storage is rapidly increasing, studies on practical use cases in this area are still insufficient.

In particular, I/O performance is of critical importance for cloud service providers [14]: "*Amazon found every 100ms of latency cost them 1% in sales. Google found an extra 0.5 seconds in search page generation time dropped traffic by 20%. A broker could lose $4 million in revenues per millisecond if their electronic trading platform is 5 milliseconds behind the competition.*" Virtualization has been a key technology in providing enhanced I/O performance to customers. To further satisfy customers, it is imperative that virtualization platform providers satisfy the service level objectives (SLOs) of each virtual machine (VM). Therefore, we analyze the I/O performance and present solutions in the cloud computing platform, where the use of flash memory

The associate editor coordinating the review of this manuscript and approving it for publication was Xiao-Sheng Si.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see http://creativecommons.org/licenses/by/4.0/



(a) YCSB-A (b) YCSB-D (c) Fileserver (d) MSN (e) Prj (f) Prn

FIGURE 1. Proportionality of I/O throughput (*y*-axis) resulting from using Cgroup for various workloads on a shared SSD. (Notation: VM-*x*, where *x* is the I/O weight used to assign Cgroup proportion with higher *x* meaning more throughput is requested).

based SSDs has greatly increased but where research in the area is still limited.

In this work, we empirically investigate internal components of an SSD among VMs (or tasks) sharing the SSDs as storage in terms of I/O proportionality and stability. Through the analysis, we show the performance characteristics of SSDs in a shared storage environment and figure out the reasons of performance interference among VMs by examining the components of an SSD such as channels, DRAM buffer, and Native Command Queueing (NCQ). From the results, we learn lessons about how these components influence SLO satisfaction of VMs or tasks. In particular, we find that the current Linux default I/O schedulers supporting I/O weights do not satisfy I/O proportionality requirements as requested by users and do not provide I/O stability due to performance interference between VMs or tasks.

I/O proportionality here refers to the relation among the I/O throughput as observed by VMs (or tasks) competing for I/O resources in a shared storage environment. I/O is said to be proportional if each VM observes I/O throughput in proportion to the relative weight of I/O resources that the VMs request [15]. I/O disproportionality and instability are particularly noticeable in fast storage devices such as SSDs.

We find that components of SSDs, the NCQ and the cache in the SSD controller, hurt I/O proportionality and stability. Based on our analysis, we solve the SLO problem with two novel SSD-aware host level I/O schedulers [16], called A+CFQ and H+BFQ, that extend the CFQ (Complete Fairness Queueing) [17] and BFQ (Budget Fair Queueing) [18] schedulers; these are the only two I/O schedulers that provide I/O proportionality on Linux. We show significant improvements in I/O performance proportionality and stability through minimizing performance interference between VMs with the proposed I/O schedulers. Then we experimentally analyze the interference effects between concurrently executing tasks in terms of I/O throughput, latency, and proportionality with the characteristics of the workloads. In existing I/O schedulers, severe performance interference between tasks is observed. We show that the proposed schedulers eliminate performance interference almost completely. The primary contributions of this paper are as follows:

- We empirically examine internal components of a flash based SSD shared by multiple VMs or tasks in terms of I/O performance and proportionality. To the best of our knowledge, our study is the first to analyze the performance impact of the internal components of the flash SSDs as shared storage environment [16].
- We find that the Linux I/O schedulers do not meet the SLO of each user in a shared SSD. Then, we present two I/O schedulers, A+CFQ and H+BFQ, which are designed to satisfy SLO requirements, based on the lessons learned from the analysis.
- We extensively evaluate the existing and proposed I/O schedulers through various workloads in terms of I/O performance, proportionality, and stability on Linux kernel 3.18.x.
- We also evaluate and analyze the performance interference between tasks competing on a shared SSD.

The remainder of this paper is organized as follows. In the next section, we present our motivation and summarize related work. In Section III, we perform experiments and analyze I/O proportionality on a shared storage. We discuss and present the design of the A+CFQ and H+BFQ I/O schedulers in Sections IV and V, respectively. In Section VI, we discuss evaluation results. In addition, we analyze the performance interference between concurrently running workloads on SSDs in Section VII. Finally, in Section VIII, we conclude the paper.

II. MOTIVATION AND RELATED WORK

In this section, we present the motivation of this work by conducting experiments. Then, an overview of flash memory is presented, followed by a discussion on related work.

A. DISPROPORTIONAL I/O PERFORMANCE

Figure 1 depicts the motivation of this work, which shows the disproportionality of I/O throughput of four VMs running the same workload simultaneously for six different workloads. Using Cgroup [19] in Linux, we assign different I/O weights to the VMs denoted as VM-x, where x refers to the I/O weight, and observe the I/O throughput allotted and measured by each VM. Here, higher I/O weight means we expect to be allotted higher bandwidth by Cgroup proportional to the x value. Cgroup is known to match well in proportion to CPU and memory resources as well as HDDs [20]. However, the effectiveness of Cgroup with flash based SSDs has not been explored much. Currently, we are aware of only one study [21] in this regard, which we review in Section II-C.

To conduct experiments, we create VMs, which are kernel-based virtual machines (KVM), and run the same

TABLE 1. KVM environment.

Description	Host	per VM resource
CPU core	8	1
Memory size	32GB	1GB
OS	Ubuntu-14.x with KVM	CentOS 7
Storage	200GB MLC SSD	50GB partition
	1TB HDD	100GB partition

TABLE 2. Characteristics of I/O workloads.

Workload	Request Total	Read Ratio	Average Req. Size
YCSB-A	96GB	0.99	132KB
YCSB-D	95GB	0.99	133KB
Fileserver	34GB	0.5	6.17KB
MSN	63.6GB	0.51	40.4KB
Prj	92.8GB	0.13	30KB
Prn	75.5GB	0.79	17.6KB

workload on each VM. Each VM has the same capacity (50GB partition) of a commodity MLC SSD with a SATA-3 interface. Table 1 shows the summary of the experimental environment. We use three real workload benchmarks (YCSB-A and YCSB-D [22] and the Fileserver emulation in FIO (denoted as Fileserver) [23]) and three I/O trace workloads (MSN [24], Prj [24], and Prn [24]). For the trace workloads, we use an open source tool that generates I/O requests based on the trace workloads to drive the experiments [25]. Table 2 shows the characteristics of the I/O workloads.

From the results in Figure 1, we see that the measured I/O throughput does not correspond to the given I/O weights. In particular, in case of YCSB workloads, as shown in Figures 1(a) and (b), VMs with higher I/O weights do not provide higher throughput. Moreover, for the Fileserver results shown in Figure 1(c), VM-5's I/O throughput oscillates periodically and performance of VM-4 also shows large fluctuations. In case of the MSN workload shown in Figure 1(d), we see periodic spikes of performance on VM-2, VM-4, and VM-5. Even though the difference between the weights of VMs is up to 5 times, there is a performance difference from 10 times to 35 times when performance spike occurs. Finally, the workloads of Prj and Prn show very irregular performance fluctuations. We observe that I/O proportionality could deviate as high as around 100 times in excessive cases although the maximum assigned I/O weight is 5.

From the evaluation results, we conclude that I/O proportionality and stability are not being achieved with Cgroup on SSDs as a shared storage. Given that previous reports have shown Cgroup to work fairly well [20], we suspect that causes of I/O disproportionality and instability comes from SSDs. Our own results depicted in Figure 2, which compares the I/O proportionality results when using an HDD (configuration set as in Table 1) and the SSD that averages out the performance over all times in Figure 1, support this conjecture.

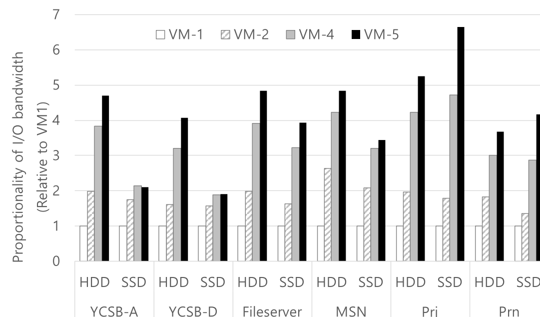


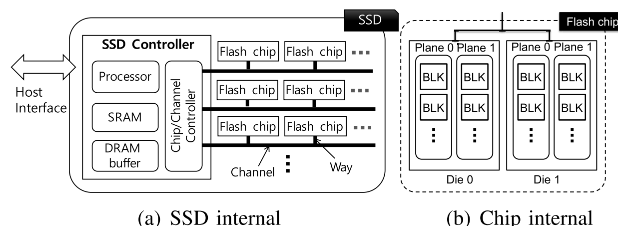
FIGURE 2. Proportionality of I/O performance relative to VM-1 for various workloads.

Also, it is interesting to note that though the MSN, Prj, and Prn workload results in Figure 1 show frequent, high spikes and reversals of proportionality, the averaged out proportionality results in Figure 2 appear not be as disproportionate. This tells us that while the service provider can claim proportionality to some degree, the actual service experienced by customers at particular times of service can turn out to be extremely dissatisfactory.

In the next section, we conduct experiments on a shared SSD to figure out the reasons of the problem. Before so doing, we present an overview of flash memory and related work.

B. FLASH AND SSD BASICS

Typical SSDs employ many NAND flash memory chips connected via channels and ways as depicted in Figure 3(a). NAND flash memory is internally comprised of dies and planes, where planes consist of a fixed number of blocks and multiple pages per block as shown in Figure 3(b). I/O requests from the host can be interleaved across multiple channels, and the requests can be further interleaved on multiple dies and planes. Such parallel access of those resources allow SSDs to achieve high performance. Due to the characteristics of flash memory, cleaning operations (also called garbage collection) that cause performance degradation are required [8], [9], [11], [26].



(a) SSD internal (b) Chip internal

FIGURE 3. SSD architecture.

C. RELATED WORK

Meeting the SLO requirements for VMs in cloud computing is an important criteria, with numerous studies taking varied approaches being conducted [12], [27]–[33]. Isolating the underlying hardware resources among the VMs is the

common way to achieve SLOs. Numerous studies isolating system resources for VMs have been introduced [12], [29], [30], [32]. Commercial products such as VMware ESX server hypervisor provide the ability to isolate CPU and memory among the VMs [34], [35].

Studies to support I/O SLOs among VMs have also been conducted [29], [30], [32], [33]. PARDA [29], mClock [30], and Pisces [32] provide proportional-share fairness among the VMs. IOFlow [33] introduces a new software-defined storage architecture for ensuring QoS in multi-tenant data centers. DeepDive identifies and manages performance interference among VMs sharing hardware resources [31].

Studies on managing an SSD cache shared by VMs have also been introduced [27], [28]. A study shows that controlling the SSD from outside the SSD is difficult as one cannot control the internal workings of GC [12]. In a similar context, the study by Huang et al. also reports that SLOs are difficult to achieve due to shared resources in SSDs [13]. The study proposes a technique to isolate performance between multiple tenants by allowing each tenant to use dedicated channels and dies in SSDs. Recently, a study similar to our study was proposed [21]. This work also provides weighted fair-sharing on a shared flash-based storage and uses a throttling technique to adjust the bandwidth proportionality. The key difference between our study and this work are that this study uses a throttling technique in Linux for I/O proportionality and does not provide insights based on empirical observations of the workings of the SSD.

Since SSDs involve GC, unlike HDDs, it is difficult to ensure stable performance, which is one of the important metrics of SLO [36]. Numerous studies have been conducted to reduce the GC overhead for stable performance of SSDs. Recently, Yan et al. proposed TTFlash to almost eliminate GC overhead by exploiting a combination of current SSD technologies including the powerful flash controller, the Redundant Array of Independent NAND scheme, and the internal large RAM buffer within SSDs [8]. The study, however, is limited in that they make use of the intra-plane copyback operation that may cause reliability problems due to skipping the ECC checks. Unfortunately, this issue is not completely covered, hence requires further study. MittOS proposed an SLO-aware interface that can predict and greatly reduce I/O latency [37]. This study shows the importance of SLOs of storage systems. Multi-stream SSD was also proposed to reduce GC overhead by maintaining multiple streams in an SSD according to the expected lifetime of the data [38]. Approaches to reduce GC by managing flash memory directly from the host have also been proposed. Specifically, LightNVM and Application-managed Flash eliminate GC overhead by letting the host software manage the exposed flash memories [39], [40].

Unlike previous studies, our study empirically analyzes the effects of SSD internal components on performance SLOs. Numerous studies on I/O scheduling have been conducted, but most have not concentrated on proportionality. Most I/O schedulers have done so by managing timeslice [17], [41] and controlling virtual time [42], [43] as the main resource of I/O. The notion of timeslice has also been adopted for I/O schedulers for flash-based SSDs [44], [45]. However, we show in Sections IV that using timeslice as a resource management unit on SSDs is inappropriate.

III. ANALYSIS OF I/O BEHAVIOR ON SSDS

In this section, we conduct a set of experiments to examine how the internal components of SSDs affect the I/O performance of the VMs amongst each other. For the analysis, we use the six workloads shown in Table 2. For the sake of brevity of explanation in the analysis of each component, we exclude similar results. In this work, the target components of SSDs to explore are channels, the write cache, and NCQ.

A. EFFECT OF CHANNELS

Typical commercial SSDs employ multiple channels to connect a controller to flash memories for high performance and large capacity. To examine the effect of the channels on an SSD, we conduct the same set of experiments as in Section II-A, except that we use an SSD that is set to use only a single channel among the numerous channels. This feature is not available to end users, but was done with cooperation from the manufacturer.

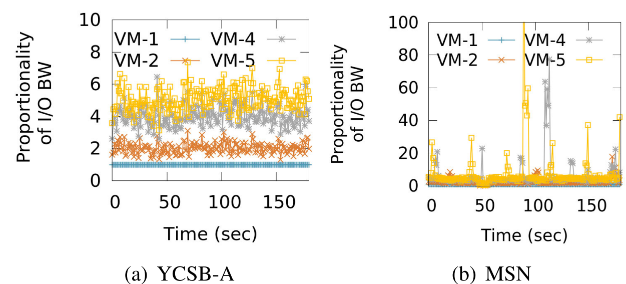
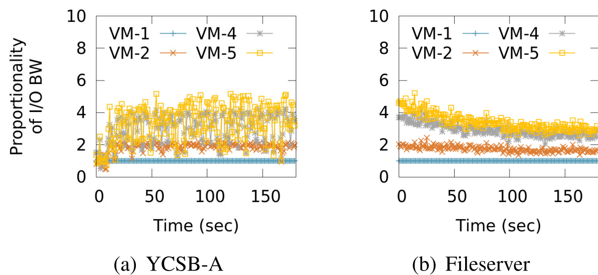


FIGURE 4. Proportionality of I/O bandwidth on single channel SSD.

Figure 4 shows the results for the two workloads analyzed. Compared to Figure 1(a), we observe that the overall trend shown in Figure 4(a) is much more compliant to proportionality according to the requested I/O weight. Of course, the absolute performance drops significantly as only a single channel is being used. (We quantify and discuss the performance degradation later with Figure 8). In case of the MSN workload, as shown in Figures 4(b), the variation of the I/O throughput still occurs but also with longer intervals than those in Figure 1(d).

Overall, we observe that using a single channel rather than using multiple channels helps somewhat with I/O proportionality. We discuss the main reason behind the effect of channels in Section III-D. However, using a single channel does not solve the problem of I/O proportionality completely, and also incurs a significant performance drop.



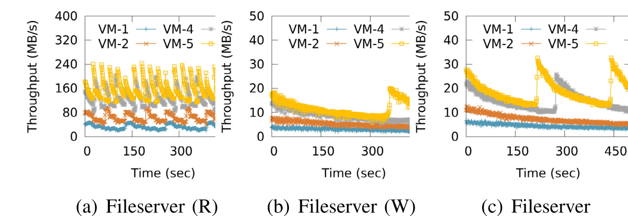
(a) YCSB-A (b) Fileserver

FIGURE 5. Proportionality of I/O bandwidth on cache off SSD.

B. EFFECT OF THE CONTROLLER CACHE

SSDs generally utilize some amount of DRAM as cache for improving performance. To examine the effect of the cache on I/O proportionality, we turn off the cache on the SSD controller. We conduct several experiments under this change. Figure 5 shows the results of the experiments. The proportionality of YCSB-A shown in Figure 5(a) is slightly better than that of Figure 1(a), but performance variations still occur. For the Fileserver workload shown in Figure 5(b), performance fluctuations are mitigated and the period of the fluctuations are longer than those of the typical SSD shown in Figure 1(c).

To get a better understanding of these results and the periodic performance fluctuation observed with the Fileserver workload, we conduct some additional experiments. While the read and write are divided equally for the Fileserver workload for these experiments in Section II-A, in order to clearly identify the effect of caching on reads and writes, the Fileserver workload is adjusted to be read-only and write-only.



(a) Fileserver (R) (b) Fileserver (W) (c) Fileserver

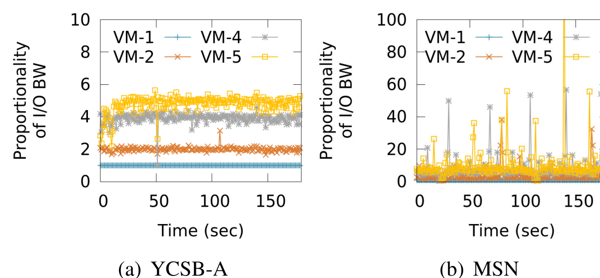
FIGURE 6. I/O bandwidth with cache disabled for Fileserver (a) Read-only, (b) Write-only, (c) Read/Write 50/50 workloads.

Figures 6(a), (b), and (c) are the results with caching turned off for read-only, write-only, and the 50/50 case, respectively. In addition, we conduct experiments with Fileserver workload over a long period of time to observe long-term performance. Figure 6(a) reveals that performance still fluctuates and the period of fluctuation is shorter for read-only workloads, especially for VM-5. However, for the write-only workload, as shown in Figure 6(b), the wild fluctuation disappears though performance itself is dispersed across a wide range and relatively very low due to the cache off. Directly comparing Figure 1(c) and Figure 6(c) that make use of the same workload, we observe that the overall trend of periodic performance fluctuation is noticeably mitigated. However, performance fluctuations still do occur though the fluctuation period is lengthened and the amplitude is reduced due to performance degradation as the cache is turned off.

We again observe a significant performance drop with the cache off. The conclusion from these experiments is that while the cache of the SSD controller is effective at improving performance, it has an unfavorable impact in supporting I/O proportionality of shared SSDs.

C. NATIVE COMMAND QUEUEING

Native Command Queuing (NCQ) is a technique that is used for the SATA interface to optimize performance of disks through reordering read and write commands. As with two previous components, we explore the effect of NCQ on I/O performance proportionality.



(a) YCSB-A (b) MSN

FIGURE 7. Proportionality of I/O bandwidth with NCQ depth = 1.

We conduct the experiments with NCQ disabled, that is, NCQ depth set to 1. We observe that I/O proportionality of YCSB-A shown in Figure 7(a) is distinctly improved compared to those of Figures 1(a). These results are encouraging in terms of I/O proportionality although the I/O performance deteriorates again since queue optimizations are disabled.

For the MSN workload shown in Figure 7(b), we see that it does not do as well as the YCSB-A workload with the bandwidth still fluctuating somewhat. We see, however, that the results slightly improve in terms of proportionality even though bandwidth spikes still remain.

Note that Figure 8 shows the total bandwidth for all VMs on each SSD used in our analysis. The x-axis in the figure denotes the workloads that we experiment with, while the y-axis shows the total bandwidth normalized to the normal SSD denoted as 'SSD'. We see that SSDs denoted 'NCQ off', '1CH' and 'Cache off' representing an SSD with NCQ disabled, single channel SSD, and write cache off, respectively, show considerably reduced performance on most workloads. For YCSB-A and YCSB-D workloads on 'Cache off', there is little performance degradation. Since YCSB-A and YCSB-D are read intensive workloads, they are not affected by turning off the cache in the SSD controller.

From the experiments, we reveal that NCQ has a strong influence on I/O proportionality. This means that exploring NCQ in detail might be the right direction for improving I/O proportionality of a shared SSD.

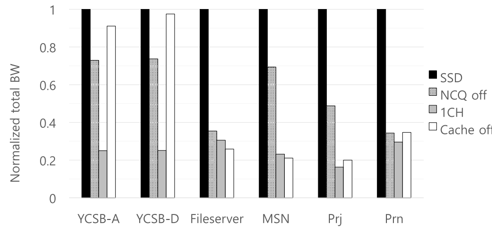


FIGURE 8. Total I/O bandwidth normalized to that of a commercial SSD.

D. EXPANDING ON NCQ

By examining the performance effect of several components of the SSD, we figured out that NCQ has a strong effect on I/O proportionality on a shared SSD. In this section, we take a closer look and figure out the principles that can be exploited for improving I/O proportionality.

To find the reason behind the positive effect of NCQ on I/O proportionality, we analyze and observe the behavior of NCQ and its interaction with other layers. The conclusion of our analysis, as we will show, is that the idle condition is the major reason behind the problem.

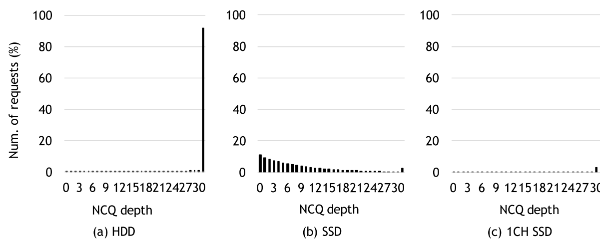


FIGURE 9. NCQ depth observed for four concurrent VMs each executing the YCSB-D workload (*x*-axis: NCQ depth, *y*-axis: percentage in particular NCQ depth).

First, let us briefly explain how NCQ's working is related to I/O proportionality and stability. The CFQ I/O scheduler used in our analysis works in a work conserving manner. In particular, CFQ maintains a request queue per VM, so if the queue meets the idle condition, it immediately switches to another queue. The idle condition for a queue are 1) if there are no pending requests in the queue and 2) if there are no outstanding requests waiting for completion from storage. Since the SSD quickly serves multiple requests in parallel, it greatly reduces the response time and empties the queue quickly. Therefore, on SSDs that support NCQ, the queue frequently reaches the idle condition, causing frequent queue switching. In contrast, single channel SSDs and HDDs are not fast enough to empty the NCQ often. This phenomenon can be seen in Figures 9(a), (b) and (c) where they show the NCQ depth, that is, the number of outstanding commands, for an HDD, SSD, and single channel SSD ('1CH SSD'), respectively, when four VMs each running the YCSB-D workload are executing. We observe that idle conditions rarely occur on the HDD and 1CH SSD, but that they occur often on the SSD. In addition, in case of the HDD and 1CH SSD, the requests fill the maximum NCQ depth in most cases (90% or more), but for the SSD, the NCQ rarely fills up.

Such frequent queue switching due to the idle condition is the main cause of I/O disproportionality and instability on a shared SSD. Recall from YCSB-A in Figure 7(a) that NCQ with depth one showed good I/O proportionality similarly to the HDD. The reason for this is that if the NCQ depth is 1, a pending request will always be waiting to be served in the queue, therefore, the idle condition is rarely met. This also implies why the single channel SSD improves I/O proportionality as shown in Figure 4.

In conclusion, we exposed the fact that I/O proportionality and stability drops significantly due to excessive switching between queues on a shared SSD. Based on this finding, we design I/O scheduling policies to prevent such switching for better I/O proportionality and stability.

IV. I/O SCHEDULERS FOR PROPORTIONALITY

In this section, we introduce a recent I/O scheduler called Budget Fair Queuing (BFQ), which employs I/O anticipation [18], a notion introduced by Iyer and Druschel for HDDs [46]. It turns out that I/O anticipation improves I/O proportionality by preventing frequent queue switching. A fundamental feature of BFQ is that it manages the storage resource via an idea of a budget, which refers to the number of sectors served. BFQ seems suitable as an I/O scheduler for I/O proportionality. However, without careful attention to budget allocation, I/O proportionality and stability may deteriorate on SSDs. Through thorough analysis, we show the reason behind this in Sections IV, VI, and VII.

In the following subsections, we explain the basic workings of the BFQ I/O scheduler. Then, we present our own improvement of CFQ, A+CFQ, which is based on a similar approach taken by BFQ, that is, I/O anticipation and budget (sector) based resource management. We show that these two I/O schedulers (BFQ and A+CFQ) perform similarly and that one can be better off than the other depending on a variety of situations. More importantly, based on these analysis, in Section V, we introduce an extension of BFQ, called H+BFQ, which provides better I/O proportionality and stability without performance sacrifice.

A. BFQ I/O SCHEDULER

The BFQ I/O scheduler employs I/O anticipation and the notion of a budget, which indicates the number of sectors served. Also, BFQ employs the per-process (or per-task) queue approach to support fair allocation of throughput like CFQ. That is, when a queue is chosen among many, the task associated with the queue will have exclusive access to storage. When the task issues a synchronous I/O to storage, BFQ lets the task wait in idle state until the request is completed, which is the core behavior of I/O anticipation. The queue becomes inactive only when one of the following conditions are met: 1) the task has no I/O requests, 2) the budget of the queue is exhausted, or 3) the allotted time of the queue has expired. When the queue becomes inactive, a new budget is

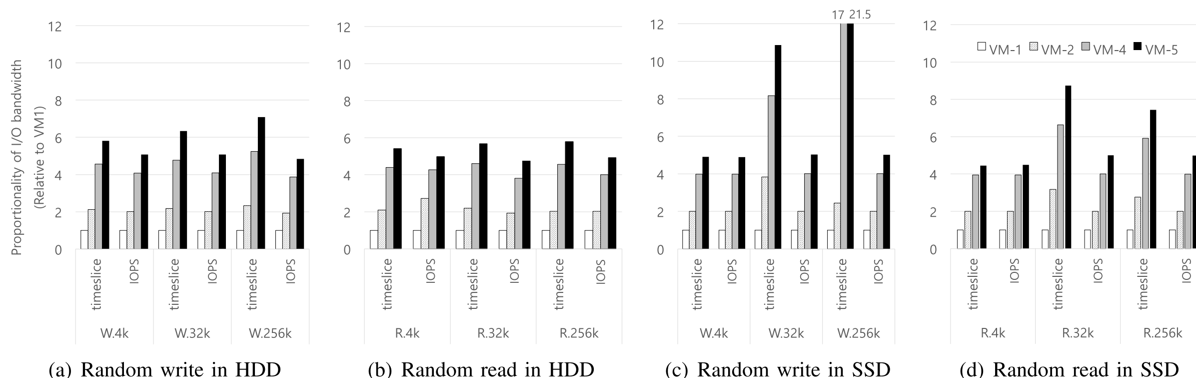


FIGURE 10. **Comparison of I/O performance between timeslice and IOPS mode.**

assigned for the next round. BFQ chooses the next queue by an internal fair queueing scheduler called B-WF^{2}Q+ [18]. B-WF^{2}Q+ maintains a *virtual finish time* for each application as a metric for choosing the next application. The *virtual finish time* is a function of the completion time of the last served request and the actual serviced budget during queue activation with a given weight. BFQ chooses the application with the minimum *virtual finish time* as the next application. In contrast, CFQ maintains either the time consumed (in timeslice mode) or the number of requests served (in IOPS mode) depending on the queue mode [17].

B. A+CFQ I/O SCHEDULER

In this section, we present the A+CFQ (Anticipation added CFQ) I/O scheduler that we design based on lessons learned from the analysis of Section III. The main features of A+CFQ are basically the similar as BFQ in that it employs heuristic I/O anticipation for fair share of I/O resources and that it adopts sector-based management for fine-grained resource allocation. We now discuss the design details.

1) SECTOR BASED SCHEDULING

Generally, I/O schedulers manage storage resources through time (timeslice) or number of I/O requests (IOPS). Traditionally, most I/O schedulers focusing on performance and fairness adopt timeslices as the mean of managing resources [41], [42]. However, the timeslices approach has limitations with I/O proportionality and stability when we use fast storage such as SSDs. As discussed in Section III, an SSD with NCQ, the I/O scheduler issues as many requests as possible to storage concurrently. In situations where many requests are handled in parallel, the I/O scheduler is not able to accurately measure I/O usage with the timeslices. Figure 10 presents how much I/O proportionality is affected when timeslice and IOPS are adopted to measure I/O usage at the I/O scheduler for an HDD and an SSD. We conduct a simple set of experiments with the same environment as in Section II-A for workloads that issues random 4KB, 32KB, and 256KB sized read and write requests. We observe in Figures 10(a) and (b), in case of the HDD, I/O proportionality matches well for both the timeslice and IOPS modes. In contrast, in case of the SSD, we see that with timeslice mode, I/O proportionality for large request sizes (32KB and 256KB) is distorted as presented in Figures 10(c) and (d). For IOPS mode with SSD, I/O proportionality seems to perform fairly well compared to that of timeslice mode as presented in Figures 10(c) and (d). However, as shown in Figure 1, it does not work well for all workloads. Note that all the experiments in Section II-A were conducted with IOPS mode.

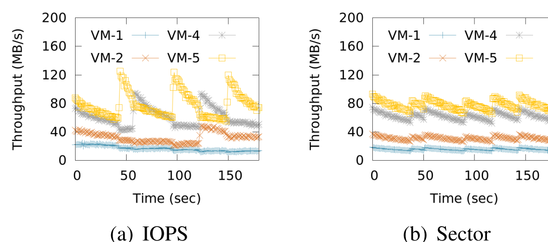


FIGURE 11. **Comparison of I/O performance when unit of I/O usage measurement is IOPS and sectors for Fileserver workload.**

In contrast, Figure 11 shows the difference in performance proportionality between when IOPS and sectors based resource management is employed for the Fileserver workload. From the figure, we observe that sector based measurement shown in Figure 11(b) improves performance proportionality and stability as well. We conclude that the unit of I/O resource usage measurement for resource management is a cause of disproportionality and fluctuation on a faster storage. Since SSDs adopt a parallel architecture and relatively large capacity cache within the controller resulting in an order of magnitude faster performance than disks, both IOPS and timeslice become a too coarse-grained unit to accurately measure the usage of resources.

It turns out the sector unit is a finer and more precise unit to measure resource usage. This is one of the reasons why we choose to take the sector unit measurement similar to BFQ. Even though A+CFQ is similar to BFQ, the details in the budget management is different. Specifically, in A+CFQ, the history of resource usage, indicated as the number of

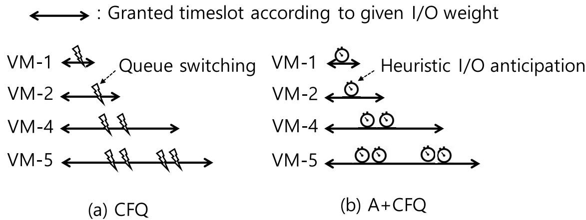


FIGURE 12. **Comparison of timeslot usage for CFQ and A+CFQ I/O scheduler.**

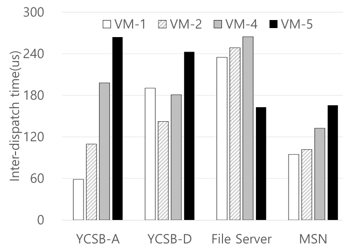


FIGURE 13. **Average inter-dispatch time per VM.**

sectors served, is maintained for each queue to provide fairness among processes. The resource usage is accumulated during queue activation. When the queue is deactivated, the accumulated usage is transformed into a virtual time by dividing the accumulated usage by the given I/O weight of the queue. The queue with minimum virtual time is chosen as the next queue when serving requests among processes.

2) HEURISTIC I/O ANTICIPATION

In a study many years ago, Iyer and Druschel introduced Anticipatory scheduling of I/O to reduce excessive seek overhead of disks (HDDs) for synchronous I/O requests [46]. The study reveals an inefficiency of handling synchronous I/O on disks and then propose a way to overcome the problem. The inefficiency in handling synchronous I/O requests is also related to idleness. When a task issues a new request once completing the previous I/O request, there is a short idle period between when a previous request completes and issuing a new request. If a new request is issued by another job during this idle period, the movement of the disk head can cause excessive seek overhead. This situation was referred to as deceptive idleness. Instead, if the task takes a short period delay to anticipate a new request, then issues the request, disk seek penalty is largely reduced due to spatial locality of the task. I/O anticipation has also been adopted in non-mechanical disks such as SSDs to ensure the continuity of a task's execution and to avoid violations of fairness [44], [45].

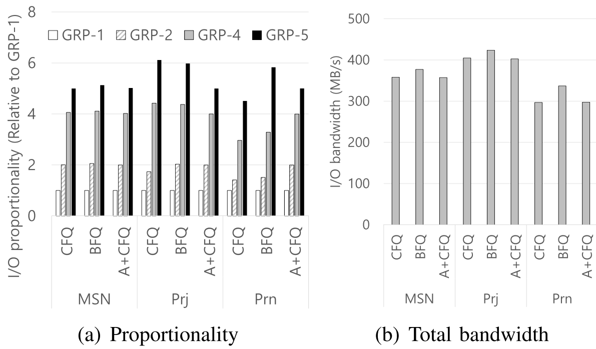
As discussed, BFQ also employs I/O anticipation by stalling for the synchronous I/O. A+CFQ also adopts an anticipatory policy, but again, differing in detail. Figure 12(a), depicts a situation where queue switching occurs before the granted timeslot ends when different I/O weighted VMs are running concurrently for CFQ. Each VM is given a timeslot proportional to the given I/O weight. During handling I/O requests, once the condition of the queue switching discussed in Section III-D is met, queue switching occurs regardless the allotted timeslot and continuity of task execution is broken. The queue switching causes fluctuation of the I/O throughput resulting in I/O disproportionality and instability. Naturally, VMs with higher I/O weights tend to have more queue switching since they are allotted longer timeslots. Eventually, they experience worse I/O proportionality and stability.

Figure 12(b) depicts how we adopt I/O anticipation. Where queue switching occurs, we employ heuristic I/O anticipation. (This is where the A, for anticipation, of our name A+CFQ, comes from). The key approach of heuristic I/O anticipation is to maintain the average inter dispatch time for each VM and if the waiting time after dispatching is less than the average dispatch time, we continue to wait in anticipation of the next request to arrive. Average inter-dispatch time is continually kept while each VM is running. We observed the average inter-dispatch time of the VMs for a set of our workloads. Figure 13 presents the observed average time. The x-axis is the workloads and the VMs, while the y-axis shows inter-dispatch time in microseconds. We observe here that the inter-dispatch time of each VM varies quite widely. Hence, it is important that each VM be able to maintain it own inter-dispatch time value so as to avoid deceptive idleness.

Algorithm 1 *Heuristic I/O Anticipation*

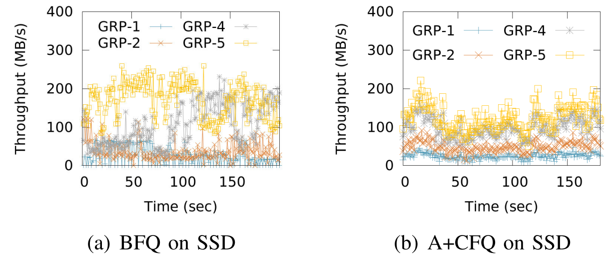
1: *Avg_Disp(VM$_i$): Avg. inter-(request) dispatch time of VM$_i$*
2: *Wait_Time(VM$_i$): Time between previous I/O completion time and the next request dispatch*
3: *Idle_condition(VM$_i$): Is VM$_i$ in idle condition?*
4:
5: Dispatch request(VM$_i$)
6: **if** Idle_condition(VM$_i$) AND (Wait_Time(VM$_i$) $\geq$ Avg_Disp(VM$_i$)) **then**
7: switch to VM$_j$ queue /* *VM$_j$ queue is selected as next queue* */
8: **else**
9: keep VM$_i$ queue activated
10: /* *Either not Idle or keep waiting due to Heuristic I/O Anticipation* */
11: **end if**

The details of the heuristic I/O anticipation scheme is presented in Algorithm 1. Lines 1 and 2 represent the data structures that are maintained. We keep the average inter-dispatch time for each VM$_i$ in Avg_Disp(VM$_i$) and the interval between the previous I/O completion time and the most recent request dispatch time of VM$_i$ in Wait_Time(VM$_i$). Idle_condition(VM$_i$), in line 3, holds true when VM$_i$ satisfy the idle condition. As was discussed in Section III-D, Idle_condition() becomes true when there is no request to dispatch and there are no outstanding requests waiting for a response. After every synchronous I/O request is dispatched to an SSD (lines 5), the I/O scheduler decides whether to



(a) Proportionality (b) Total bandwidth

FIGURE 14. Comparison of CFQ, BFQ, and A+CFQ (a) proportionality and (b) I/O bandwidth.



(a) BFQ on SSD (b) A+CFQ on SSD

FIGURE 15. Comparison of I/O bandwidth with respect to time for BFQ and A+CFQ for the Prn workload.

maintain the VM_i queue (lines 8 and 9) or to switch to a different queue (lines 6 and 7). The current queue is switched to that of a different VM if the current VM_i is idle and the wait is longer than the average dispatch time. Otherwise, that is, if either VM_i is not idle or the wait time was shorter than the average dispatch time, the current queue is maintained.

C. A+CFQ PERFORMANCE EVALUATION

In this section, we evaluate I/O performance, proportionality, and stability of A+CFQ by comparing with CFQ and BFQ on Linux kernel 3.18.x. Since BFQ does not support an interface to the virtualization system as shown in Table 1, we conduct experiments by concurrently executing four tasks (or groups) on a host (denoted GRP-x, where x is given I/O weight). From here on, we only present results for three workloads, namely, MSN, Prj, and Prn. This choice is made to present a consistent set of results despite the limitations on the experimental environment due to the fact that the VM interface is not supported for BFQ. We note that the results of our proposed schedulers reported here are consistent for the other workloads in various experimental settings that we performed them in.

Figures 14(a) and (b) present the results for I/O proportionality and performance in total bandwidth, respectively. We observe that A+CFQ guarantees I/O proportionality corresponding to the I/O weights while BFQ does not guarantee it for Prj and Prn workloads. However, BFQ performs slightly better than A+CFQ in terms of throughput as shown in Figure 14(b).

Let us now take a closer look by observing the I/O throughput of each group over time. Figure 15 shows the distribution of the I/O throughput of BFQ and A+CFQ for the Prn workload over time. We observe in Figure 15(a), in case of BFQ, that large performance oscillations and even performance reversals occur. In contrast, in case of A+CFQ shown in Figure 15(b), I/O proportionality matches well according to the I/O weight throughout its execution and overall performance stability is also significantly better than BFQ.

Let us figure out where the I/O disproportionality and instability of BFQ are coming from. The main cause of this phenomenon is that I/O throughput on a shared SSD tends to be significantly affected by the size of the request. This is exemplified by Figure 16(a), which shows the relation between I/O throughput and average request size of each group for the Prn workload throughout its execution with BFQ. We observe that the I/O throughput of all groups to be dependent on the size of requests. On the other hand, H+BFQ that we introduce in the next section performs fairly well on I/O proportionality and stability regardless of request size.

V. H+BFQ: HISTORY ADDED BFQ

In the previous section, we showed that I/O performance of BFQ is excessively sensitive to the size of requests. In this section, we introduce the H+BFQ I/O scheduler we designed. We show that I/O proportionality and stability of H+BFQ outperforms BFQ.

Figure 17 compares the budget usage distribution of each task for BFQ and H+BFQ. The arrow lines for each group denoted represent the allotted time (or rounds) to dispatch requests to storage. Naturally, the higher the I/O weight x, the more time slots are allocated. The bar denotes the number of sectors served and the bar height represents the relative number in comparison with other groups.

We observe in Figure 17(a), in case of BFQ, that the first round budget (or number of sectors) of GRP-4 and GRP-5 are allocated and serviced much more than GRP-1 and GRP-2. Notice that the budget of GRP-4 and GRP-5 are replenished to its full regardless of usage of the previous rounds. This causes excessive allocation of budget, eventually resulting in I/O disproportionality and, more seriously, instability among the tasks sharing the SSD. The I/O instability is exacerbated with fast storage devices such as SSDs due to frequent budget allocations.

The solution to this problem is to use the budget usage information of previous rounds in allocating budget for the next round. We refer to the policy that we design incorporating this solution, H+BFQ (History added BFQ) as it makes use of budget usage history. The policy is implemented by checking the reason behind the deactivation of a task. Recall that there can be three reasons for queue deactivation: 1) the allocated time slot has expired, 2) there are no more requests (idle), or 3) the task has run out of budget. H+BFQ behaves the same as BFQ if the reason of task deactivation is either 1) or 2), that is, the budget is simply the remaining budget. However, if the reason for deactivation is 3), H+BFQ regards the task to have been too aggressive. As a result, a value lower

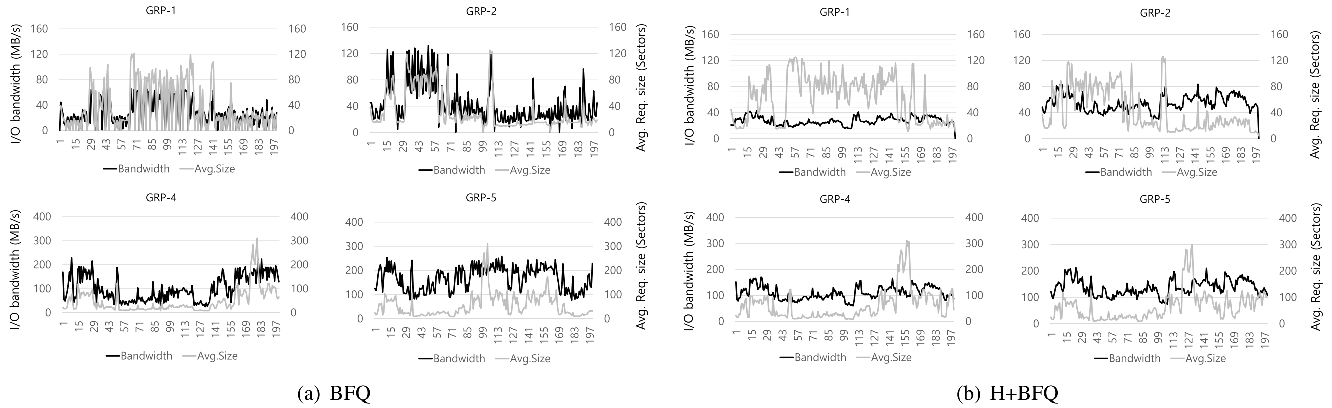


FIGURE 16. Relation between I/O throughput (black line) and average request size (gray line) of Prn workload.

Algorithm 2 *H+BFQ Budget (Re)Allocation as Task T_i is Deactivated*

1: *One deactivation reason is set to TRUE: EXPIRE, IDLE, EXHAUST*
2: *EXPIRED: Queue timer has expired*
3: *IDLE: Task has no requests*
4: *EXHAUSTED: Queue budget has run out*
5:
6: *BUDGET(T_i): Current budget of task T_i*
7: *Default_budget: Default budget size*
8: *Min_budget: Minimum budget size set to Default_budget divided by 32*
9: *Serviced_budget: Budget consumed during current task activation period*
10:
11: *N: Number of tasks running concurrently*
12:
13: BUDGET(T_i) ← BUDGET(T_i) - Serviced_budget
14: **if** (EXPIRED OR IDLE) AND (BUDGET(T_i) ≤ Min_budget) **then**
15: BUDGET(T_i) ← Default_budget *//Reset to Default budget*
16: **else if** EXHAUSTED **then**
17: BUDGET(T_i) ← MAX(Default_budget/N, Min_budget)
18: **end if**

than the default budget is allocated as the new budget. For our experiments, we assign this minimized budget to the greater of the default budget divided by the number of tasks N, or the Min_budget, which is the minimum budget allowed in BFQ. Recall that Figure 16(b) presents the relation between the I/O bandwidth and the request size dispatched to storage on H+BFQ. From the results, we see that the I/O bandwidth remains fairly constant regardless of the size of the request. The reason for this is that H+BFQ prevents processes from using excessive budgets through the management of the newly allocated budget size.

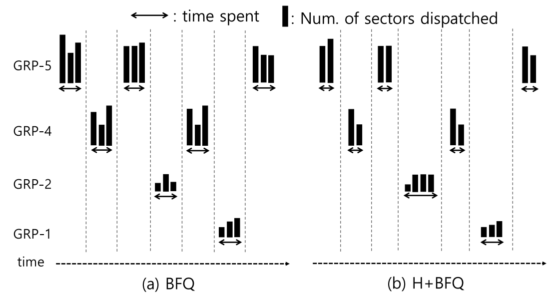
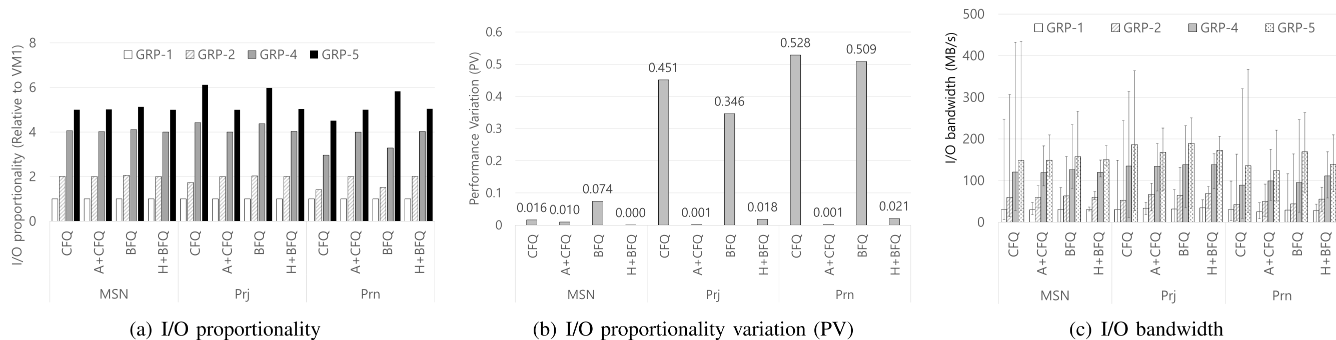


FIGURE 17. Comparison of budget usage distribution between BFQ and H+BFQ.

Algorithm 2 shows the procedure used to allocate a new budget for the next round of a task when the current group is deactivated. As explained, there are 3 conditions for deactivating a queue: EXPIRE, IDLE, and EXHAUST of which one will be set to true (lines 1 through 4). BUDGET(T_i), line 6, denotes the current budget of task T_i. All tasks, as they are initiated, are allocated the same default budget denoted Default_budget (line 7). As shown in line 13, once the task is deactivated, BUDGET(T_i) is updated by subtracting the serviced budget, Serviced_budget (defined in line 9), from the current budget, BUDGET(T_i). If the reason for deactivation is either EXPIRE or IDLE and the remaining budget, BUDGET(T_i), is less than or equal to a threshold budget, Min_budget (defined in line 8), the current budget is renewed to the Default_budget (lines 14 and 15). Otherwise if the reason for deactivation is EXHAUSTED, the budget is set to a much smaller one by assigning it the Default_budget divided by *N*, the number of tasks, or Min_budget, whichever is larger (lines 16 and 17).

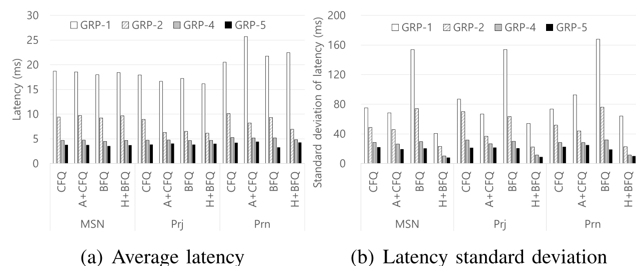
VI. EVALUATION OF H+BFQ

In this section, we conduct evaluations of H+BFQ by comparing with CFQ, BFQ, and A+CFQ in terms of I/O proportionality, stability and performance. For the evaluations, we implement H+BFQ in the same experimental setting as before. We present results of the three workloads, MSN, Prj,



(a) I/O proportionality (b) I/O proportionality variation (PV) (c) I/O bandwidth

FIGURE 18. Comparison of (a) I/O proportionality, (b) I/O proportionality variation (PV), and (c) average I/O bandwidth with the minimum and maximum denoted by the thin lines.



(a) Average latency (b) Latency standard deviation

FIGURE 19. Comparison of latency.

and Prn, as previously mentioned, concentrating on H+BFQ and BFQ.

A. I/O PROPORTIONALITY

Figure 18(a) presents the results of I/O proportionality for MSN, Prj, and Prn workloads. As shown in the figure, I/O proportionality of H+BFQ corresponds to the I/O weight for all workloads. A+CFQ also performs fairly well while CFQ and BFQ suffer disproportionality on Prj and Prn workloads.

To quantify how well I/O proportionality matches with the given I/O weights, we introduce a simple metric that we call Proportionality Variation (PV) represented by the following equation.

$$PV = \frac{1}{N} \cdot \sum_{for\ all\ i} \left| prp_{ideal}\left(GRP_i\right) - prp_{result}\left(GRP_i\right) \right|$$

where N is the number of task groups considered, $prp_{ideal}(GRP_i)$ and $prp_{result}(GRP_i)$ are the ideal proportionality and the obtained proportionality, respectively, of GRP_i. Essentially, PV presents how much the measured I/O proportionality deviates from the ideal proportionality. Therefore, lower PV means that better I/O proportionality is achieved. Figure 18(b) presents the PVs for each of the I/O schedulers, where PVs of A+CFQ and H+BFQ are significantly lower than CFQ and BFQ.

B. I/O THROUGHPUT AND STABILITY

In this section, we discuss I/O throughput and stability with experimental results. Figure 18(c) presents the I/O throughput with minimum and maximum denoted by the thin line on each bar of each task for the three workloads. The overall I/O throughput of H+BFQ is similar to BFQ for all three workloads in the result. GRP-5 performance of H+BFQ is slightly lower than that of BFQ, but the total performance is almost the same. More importantly, we observe that H+BFQ achieves the shortest range of minimum and maximum performance. This means that H+BFQ provides fairly stable throughput without any performance penalty.

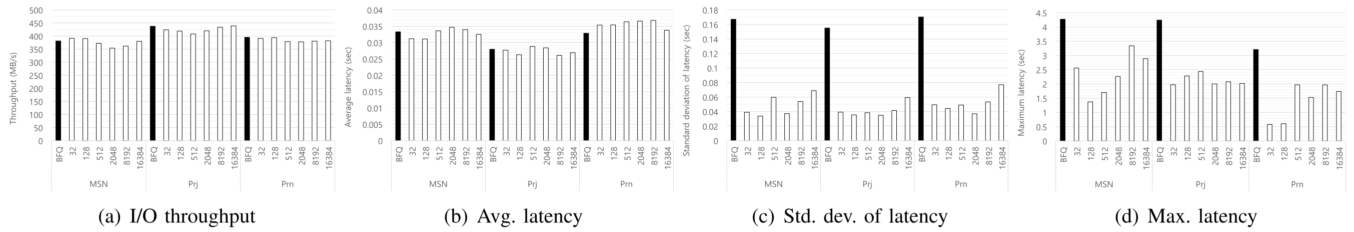
C. LATENCY AND STABILITY

Figure 19(a) shows the average latency of I/O requests for the three workloads. We observe from these results that the latency of H+BFQ is comparable with BFQ and that there is no sacrifice in performance despite the I/O proportionality improvement. Furthermore, as shown in Figure 19(b), H+BFQ significantly reduces the standard deviation of the I/O latency compared to that of other I/O schedulers for all three workloads. This again confirms that H+BFQ improves I/O stability as well.

D. IMPACT OF BUDGET SIZE OF H+BFQ

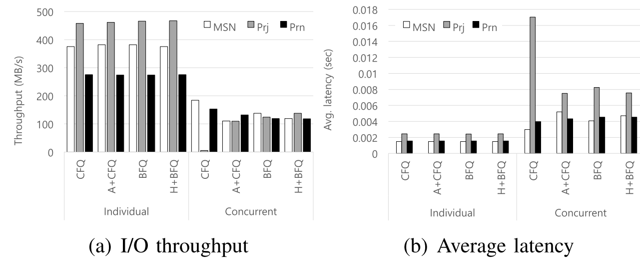
An important feature of the H+BFQ scheduler is to provide stable performance by preventing excessive budget usage of a particular group. Recall that when service is deactivated in case the budget is exhausted, a new budget is assigned with the larger of the default budget divided by the number of tasks N, or the minimum budget used in BFQ. In this section, we discuss how the size of new budget affects I/O performance in terms of throughput and latency. To do this, we observe performance while the budget size is changed with constant values when the budget is exhausted as in lines 16 and 17 in Algorithm 2. The experimental environment and workloads are all the same as those of experiments of H+BFQ in Sections VI-A through VI-C.

Figures 20(a), (b), (c), and (d) show the average throughput, average latency, latency standard deviation, and maximum latency performance results, respectively. The x-axis denotes the BFQ case and the constants, which are the new budget size of H+BFQ, that are assigned to BUDGET(T_i) in line 17 of Algorithm 2 used in these experiments. The y-axis



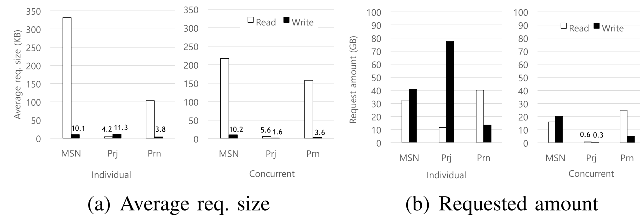
(a) I/O throughput (b) Avg. latency (c) Std. dev. of latency (d) Max. latency

FIGURE 20. Impact of budget size of H+BFQ for various workloads.



(a) I/O throughput (b) Average latency

FIGURE 21. Comparison of individual and concurrent execution of workloads on CFQ, A+CFQ, BFQ, and H+BFQ.



(a) Average req. size (b) Requested amount

FIGURE 22. I/O request summary for concurrent execution of MSN, Prj, and Prn workloads.

represents the measured performance, which is the sum of the performance of the four groups running simultaneously.

Figure 20(a) shows that there are performance differences on all workloads depending on the size of the budget. The throughput of H+BFQ according to budget sizes are similar to or differ slightly from BFQ. The budget sizes with the largest performance drop are 2048, 512, and 2048 on the MSN, Prj, and Prn workloads, respectively. Compared to BFQ, the maximum performance degradation are 7, 7, and 4% on MSN, Prj, and Prn workloads, respectively. On the other hand, the performance improvements of H+BFQ are observed only for the MSN workload.

In the case of average latency as shown in Figure 20(b), the latencies also vary with different budget sizes, but performance gaps are not significant. For the MSN and Prj workloads, the latencies are the shortest when the assigned budget size is 128. In case of the Prn workload, average latency increases by 3 to 7% compared to BFQ depending on the assigned budget size.

One of the distinct benefit of H+BFQ is that the standard deviations of the latencies are reduced as shown in Figure 20(c). We observe that the standard deviations are considerably reduced, especially when budget size is small.

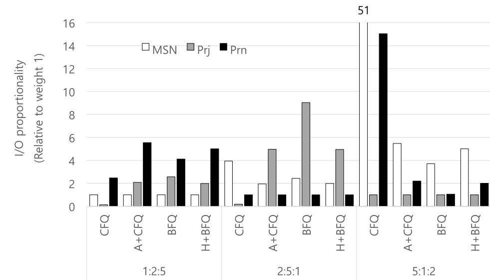


FIGURE 23. Comparison of I/O proportionality for concurrent execution of MSN, Prj, and Prn workloads on CFQ, A+CFQ, BFQ, and H+BFQ. The numbers below the x-axis indicate the I/O weights. x:y:z represents the weights of the MSN, Prj, and Prn workloads, respectively.

This is because the budget size limits the use of the SSD, thus preventing the overuse by a particular group.

Finally, Figure 20(d) shows the measured maximum latencies. We see, as with the standard deviation results, that maximum latency of H+BFQ decreases significantly, in particular, by up to 3.1, 2.1, and 5.5 times for MSN, Prj, and Prn workloads, respectively, compared to BFQ.

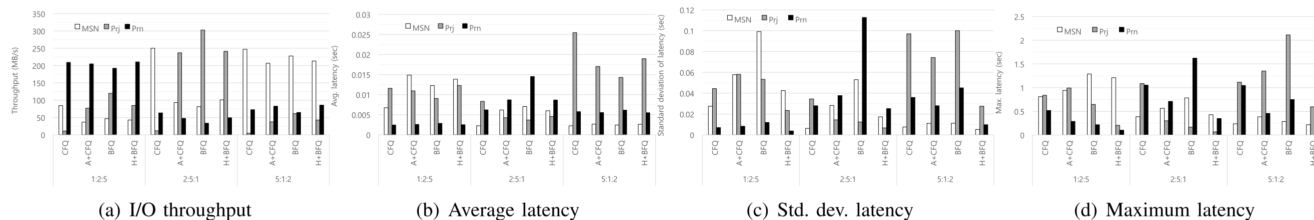
Overall, we see that performance generally improves with smaller budget values, with 128 showing the best for most cases. This is because reducing the size of the budget prevents one group from monopolizing the SSDs for a long time.

VII. INTERFERENCE ANALYSIS ON REAL TRACE WORKLOADS

In this section, we first discuss experimental results of running different workloads concurrently, which would be a more realistic scenario. Notice that all evaluations prior to this section are scenarios in which the same workloads are running simultaneously. In particular, we focus on the interference among tasks that are simultaneously executing.

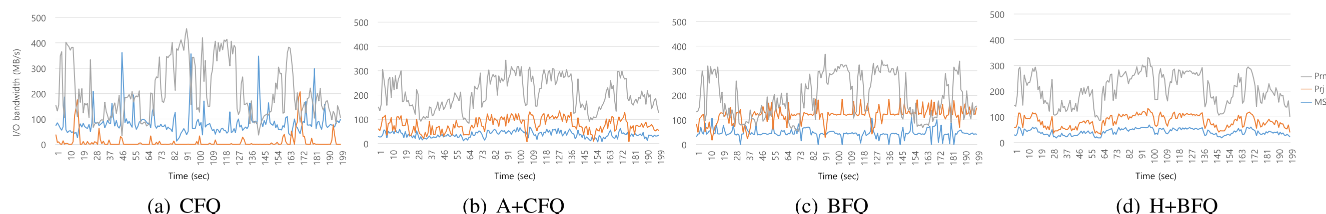
A. CONCURRENT EXECUTION WITH THE SAME I/O WEIGHT

To analyze the performance impact of concurrently executing different workloads on a shared SSD, we conduct a pair of experiments. First, we run a task to drive MSN, Prj, and Prn workloads independently, one by one and measure their performance. Then, we run three tasks to drive these three workloads concurrently and observe their performance. These two experiments will, hereafter, be called Individual and Concurrent, respectively. We conduct these experiments



(a) I/O throughput (b) Average latency (c) Std. dev. latency (d) Maximum latency

FIGURE 24. Comparison of I/O performance for concurrent execution of MSN, Prj, and Prn workloads on CFQ, A+CFQ, BFQ, and H+BFQ.



(a) CFQ (b) A+CFQ (c) BFQ (d) H+BFQ

FIGURE 25. Comparison of I/O bandwidth of running the MSN, Prj, Prn workloads concurrently with I/O weights set to 1, 2, and 5, respectively, for CFQ, A+CFQ, BFQ and H+BFQ.

on the host system described in Table 1 and are performed with the CFQ, A+CFQ, BFQ, and H+BFQ I/O schedulers and all workloads have the same I/O weight.

Figure 21 shows the measured performance of the two experiment sets, where the x-axis lists the I/O schedulers with the left side being the results for the Individual case and the right side is for the Concurrent case. In Figure 21(a), where the y-axis denotes throughput, we see that for the Individual case, I/O throughput of MSN, Prj, and Prn is roughly 370, 460, and 270 MB/s, respectively, irrespective of the I/O scheduler. However, for the Concurrent, we observe that performance of each task is different depending on the I/O scheduler used. Most notable here is that the throughput of Prj is drastically decreased with CFQ in the Concurrent experiment. In case of the other I/O schedulers, we see that the results are quite similar although each scheduler show somewhat different results for each of the workloads. We see similar results for latency as shown in Figure 21(b) with Prj performing considerably worse with the CFQ scheduler in the Concurrent case.

Let us now see why Prj does so badly with CFQ in the Concurrent case. To explain this, let us start with a couple of characteristics of the I/O requests. Figure 22 shows the average request size and the total I/O requested by the workloads for the Individual and Concurrent experiments. In Figure 22(a), we notice that the average read request size of Prj is significantly smaller than the other workloads for both the Individual and Concurrent cases, while we see no significant difference in case of the write request size. In contrast, Figure 22(b) shows the request amount of I/O. Here, in case of the Individual, Prj workload has the smallest amount of read requests and the largest amount of write requests compared to other workloads. In the Concurrent case, however, the requested amount for Prj is dramatically decreased, especially for write requests, while for other workloads it is only decreased by roughly half. Now recall that, as was described in Section IV-B.1, CFQ makes use of IOPS and the timeslice as a measuring unit for I/O usage. This results in disk usage being over-charged for small size requests as is the case for Prj. Hence, we see the drastic performance drop for Prj when running concurrently with other workloads. We note that this problem is explained in Section IV-B.1 in detail and also shown through experiments.

B. CONCURRENT EXECUTION WITH DIFFERENT I/O WEIGHTS

In this section, we discuss experimental results of Concurrent experiment with various I/O weight sets. Figure 23 shows the proportionality of the measured I/O throughput for various I/O weighted scenarios. From the figure, we see that, for all I/O weighted sets, CFQ fails to process I/O requests according to the given I/O weight, while the results for BFQ are partly correct. However, the A+CFQ and H+BFQ schedulers show I/O proportionality that closely matches the given I/O weights.

Figure 24 shows the actual observed I/O performance for each I/O scheduler as the I/O weights are varied. Figure 24(a) shows the throughput, while the average latency results are shown in Figure 24(b). The standard deviation and maximum latency results shown in Figure 24(c) and (d), show that H+BFQ retains considerably more stable performance than the other I/O schedulers.

Finally, Figure 25 shows I/O throughput throughout execution time when concurrently executing the MSN, Prj, and Prn workloads with I/O weight set to 1, 2, and 5, respectively. Hence, the x-axis is time in seconds, while the y-axis is the I/O bandwidth. In the case of CFQ, as shown in Figure 25(a), the measured bandwidth is not at all proportional to the given I/O weight. We observe that the performance of Prj is less

than that of MSN, even though Prj has twice the I/O weight than that of MSN. Furthermore, for most of the execution Prn, which has the highest weight, shows the lowest performance. On the other hand, in case of A+CFQ, the performance of all workloads is always proportional to the given I/O weight during execution time as shown in Figure 25(b). For BFQ, the performance of all workloads are generally proportional to the I/O weight, but we do observe occasional inversions in performance as shown in Figure 25(c). Finally, for H+BFQ, whose results are shown in Figure 25(d), we see that the measured performance always coincides with the I/O weight during execution time.

VIII. CONCLUSION

I/O performance has a critical impact on user experience in cloud computing platforms. Recently, the use of flash-based SSDs in cloud servers has been rapidly increasing. However, there are only limited studies on the effect of using SSDs on cloud platforms in terms of SLOs.

In this study, we revealed that I/O proportionality of VMs or task groups using HDD based approaches was not satisfactory on a shared SSD. We analyzed and found the reason behind this by examining the components of SSDs that affect performance such as channels, DRAM buffer, and NCQ. We presented two new SSD-aware host level I/O schedulers called A+CFQ and H+BFQ, which are improvements on state-of-the-art I/O schedulers CFQ and BFQ, respectively. We showed that the schedulers significantly reduce performance fluctuations and provide stable performance compared to the Linux I/O schedulers without sacrifice of performance. Furthermore, we analyzed the interference phenomenon of I/O performance when multiple tasks simultaneously generate workloads having different characteristics. We found that A+CFQ and H+BFQ eliminate the performance interference seen in the CFQ and BFQ schedulers and greatly reduce performance anomalies observed with shared SSD storages.

REFERENCES

[1] Statista. (2017). *Flash Memory Market Revenues Worldwide From 2013 to 2021*. [Online]. Available: https://www.statista.com/statistics/553556/worldwide-flash-memory-market-size/

[2] Sandisk. (2018). *iNAND Automotive Embedded Flash Drives*. [Online]. Available: https://www.sandisk.com/oem-design/automotive/inand

[3] Micron. (2017). *Micron Reveals Critical Technologies for Autonomous Vehicles*. [Online]. Available: https://www.sandisk.com/oem-design/automotive/inand

[4] PwC. (2015). *The Internet of Things: The Next Growth Engine for the Semiconductor Industry*. [Online]. Available: https://www.pwcaccelerator.com/pwcsaccelerator/docs/pwcs-accelerator-the-internet-of-things.pdf

[5] NetworkComputing. (2016). *The Future of Data Storage: Flash and Hybrid Cloud*. [Online]. Available: https://www.networkcomputing.com/storage/future-data-storage-flash-and-hybrid-cloud/396697859

[6] Forbes. (2016). *Flash Memory for Data Centers and Clouds*. [Online]. Available: https://www.forbes.com/sites/tomcoughlin/2016/04/27/flash-memory-for-data-centers-and-clouds/#23f9953a489b

[7] Micron. (2018). *When Does Cloud Computing Need Flash?* [Online]. Available: https://www.micron.com/about/blogs/2018/july/when-does-cloud-computing-need-flash

[8] S. Yan, H. Li, M. Hao, M. H. Tong, S. Sundararaman, A. A. Chien, and S. H. Gunawi, "Tiny-tail flash: Near-perfect elimination of garbage collection tail latencies in NAND SSDs," in *Proc. USENIX Conf. File Storage technol. (FAST)*, 2017, pp. 15–28.

[9] N. Agrawal, V. Prabhakaran, T. Wobber, D. John Davis, M. Manasse, and R. Panigrahy, "Design tradeoffs for SSD performance," in *Proc. USENIX Annu. Tech. Conf. (ATC)*, 2008, pp. 57–70.

[10] Y. Hu, H. Jiang, D. Feng, L. Tian, H. Luo, and S. Zhang, "Performance impact and interplay of SSD parallelism through advanced commands, allocation strategy and data granularity," in *Proc. Int. Conf. Supercomput. (ICS)*, 2011, pp. 96–107.

[11] F. Chen, D. A. Koufaty, and X. Zhang, "Understanding intrinsic characteristics and system implications of flash memory based solid state drives," in *Proc. Int. Joint Conf. Meas. Modeling Comput. Syst. (SIGMETRICS)*, 2009, pp. 181–192.

[12] J. Kim, D. Lee, and H. Sam Noh, "Towards SLO complying SSDs through OPS isolation," in *Proc. USENIX Conf. File Storage Technol. (FAST)*, 2015, pp. 183–189.

[13] J. Huang, A. Badam, L. Caulfield, S. Nath, S. Sengupta, B. Sharma, and K. M. Qureshi, "FlashBlox: Achieving both performance isolation and uniform lifetime for virtualized SSDs," in *Proc. USENIX Conf. File Storage Technol. (FAST)*, 2017, pp. 375–390.

[14] F. Khan. *The Cost of Latency*. Accessed: 2015. [Online]. Available: https://www.digitalrealty.com/blog/the-cost-of-latency/

[15] Y. Wang and A. Merchant, "Proportional-share scheduling for distributed storage systems," in *Proc. USENIX Conf. File Storage Technol. (FAST)*, 2007, pp. 47–60.

[16] J. Kim, E. Lee, and S. H. Noh, "I/O scheduling schemes for better I/O proportionality on flash–based SSDs," in *Proc. IEEE 24th Int. Symp. Modeling, Anal. Simulation Comput. Telecommun. Syst. (MASCOTS)*, Sep. 2016, pp. 221–230.

[17] J. Axboe. (2016). *CFQ IO Scheduler*. [Online]. Available: https://www.kernel.org/doc/Documentation/block/cfq-iosched.txt

[18] P. Valente and F. Checconi, "High throughput disk scheduling with fair bandwidth distribution," *IEEE Trans. Comput.*, vol. 59, no. 9, pp. 1172–1186, Sep. 2010.

[19] Paul Menage. (2007). *CGROUPS*. [Online]. Available: https://www.kernel.org/doc/Documentation/cgroups/cgroups.txt

[20] P. Ondrejka, E. Majorĭnová, M. Prpic, and D. Silas. (2016). *Managing System Resources on Red Hat Enterprise Linux 6*. [Online]. Available: https://https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/6/html-single/Resource_Management_Guide/

[21] S. Ahn, K. La, and J. Kim, "Improving I/O resource sharing of Linux Cgroup for NVMe SSDs on multi-core systems," in *Proc. USENIX Workshop Hot Topics Storage File Syst. (HotStorage)*, 2016.

[22] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proc. ACM Symp. Cloud Comput. (SoCC)*, 2010, pp. 143–154.

[23] J. Axboe. (2006). *FIO*. [Online]. Available: https://github.com/axboe/fio

[24] D. Narayanan, A. Donnelly, and A. Rowstron, "Write Off-loading: Practical power management for enterprise storage," in *Proc. USENIX Conf. File Storage Technol. (FAST)*, 2008, pp. 253–267.

[25] Y. Oh. (2015). *Trace-Replay*. [Online]. Available: https://bitbucket.org/yongseokoh/trace-replay

[26] A. Gupta, Y. Kim, and B. Urgaonkar, "DFTL: A flash translation layer employing demand-based selective caching of page-level address mappings," in *Proc. Int. Conf. Archit. Support Program. Lang. Oper. Syst. (ASPLOS)*, 2009, pp. 229–240.

[27] T. Luo, S. Ma, R. Lee, X. Zhang, D. Liu, and L. Zhou, "S-CAVE: Effective SSD caching to improve virtual machine storage performance," in *Proc. Int. Conf. Parallel Archit. Compilation Techn. (PACT)*, 2013, pp. 103–112.

[28] F. Meng, L. Zhou, X. Ma, S. Uttamchandani, and D. Liu, "vCacheShare: Automated server flash cache space management in a virtualization environment," in *Proc. USENIX Annu. Tech. Conf. (ATC)*, 2014, pp. 133–144.

[29] A. Gulati, I. Ahmad, and A. C. Waldspurger, "PARDA: Proportional allocation of resources for distributed storage access," in *Proc. USENIX Conf. File Storage Technol. (FAST)*, 2009, pp. 85–98.

[30] A. Gulati, A. Merchant, and J. Peter Varman, "mClock: Handling throughput variability for hypervisor IO scheduling," in *Proc. USENIX Conf. Oper. Syst. Design Implement. (OSDI)*, 2010, pp. 437–450.

[31] D. Novaković, N. Vasić, S. Novaković, D. Kostić, and R. Bianchini, "Deep-Dive: Transparently identifying and managing performance interference in virtualized environments," in *Proc. USENIX Annu. Tech. Conf. (ATC)*, 2013, pp. 219–230.

[32] D. Shue, J. Michael Freedman, and A. Shaikh, "Performance isolation and fairness for multi-tenant cloud storage," in *Proc. USENIX Conf. Oper. Syst. Design Implement. (OSDI)*, 2012, pp. 349–362.

[33] E. Thereska, H. Ballani, G. O'Shea, T. Karagiannis, A. Rowstron, T. Talpey, R. Black, and T. Zhu, "IOFlow: A software-defined storage architecture," in *Proc. ACM Symp. Oper. Syst. Principles (SOSP)*, 2013, pp. 182–196.
[34] VMware Inc. (2012). *Distributed Resource Scheduler*. [Online]. Available: http://www.vmware.com/files/pdf/VMware-Distributed-Resource-Scheduler-DRS-DS-EN.pdf
[35] C. A. Waldspurger, "Memory resource management in VMware ESX server," in *Proc. ACM SIGOPS Oper. Syst. Rev.*, vol. 36, 2002, pp. 181–194.
[36] X.-Y. Hu, E. Eleftheriou, R. Haas, I. Iliadis, and R. Pletka, "Write amplification analysis in flash-based solid state drives," in *Proc. ACM Int. Syst. Storage Conf. (SYSTOR)*, 2009, pp. 10-1–10-9.
[37] M. Hao, H. Li, M. H. Tong, C. Pakha, O. R. Suminto, A. C. Stuardo, A. A. Chien, and S. Haryadi Gunawi, "MittOS: Supporting millisecond tail tolerance with fast rejecting SLO-aware OS interface," in *Proc. 26th Symp. Oper. Syst. Principles (SOSP)*, 2017, pp. 168–183.
[38] J.-U. Kang, J. Hyun, H. Maeng, and S. Cho, "The multi-streamed solid-state drive," in *Proc. USENIX Workshop Hot Topics Storage File Syst. (HotStorage)*, 2014.
[39] M. Bjørling, J. Gonzalez, and P. Bonnet, "LightNVM: The Linux open-channel SSD subsystem," in *Proc. USENIX Conf. File Storage Technol. (FAST)*, 2017, pp. 359–374.
[40] S. Lee, M. Liu, S. Jun, S. Xu, and J. Kim, "Application-managed flash," in *Proc. USENIX Conf. File Storage Technol. (FAST)*, 2016, pp. 339–353.
[41] M. Wachs, M. Abd-El-Malek, E. Thereska, and G. R. Ganger, "Argon: Performance insulation for shared storage servers," in *Proc. USENIX Conf. File Storage Technol. (FAST)*, 2007, pp. 61–76.
[42] J. Bruno, J. Brustoloni, E. Gabber, B. Ozden, and A. Silberschatz, "Disk scheduling with quality of service guarantees," in *Proc. IEEE Int. Conf. Multimedia Comput. Syst.*, vol. 2, 1999, pp. 400–405.
[43] W. Jin, S. Jeffrey Chase, and J. Kaur, "Interposed proportional sharing for a storage service utility," in *Proc. ACM Int. Conf. Meas. Modeling Comput. Syst. (SIGMETRICS)*, 2004, pp. 37–48.
[44] S. Park and K. Shen, "FIOS: A fair, efficient flash I/O scheduler," in *Proc. USENIX Conf. File Storage Technol. (FAST)*, 2012, pp. 155–169.
[45] K. Shen and S. Park, "FlashFQ: A fair queueing I/O scheduler for flash-based SSDs," in *Proc. USENIX Annu. Tech. Conf. (ATC)*, 2013, pp. 67–78.
[46] S. Iyer and P. Druschel, "Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous I/O," in *Proc. ACM Symp. Oper. Syst. Principles (SOSP)*, 2001, pp. 117–130.

JAEHO KIM received the B.S. degree in information and communications engineering from Inje University, Gimhae, South Korea, in 2004, and the M.S. and Ph.D. degrees in computer science from the University of Seoul, Seoul, South Korea, in 2009 and 2015, respectively.

He is currently a Postdoctoral Researcher with the Department of Electrical and Computer Engineering, Virginia Tech, Blacksburg, VA, USA.

His research interests include storage systems, operating systems, and concurrency.

EUNJAE LEE received the B.S. and M.S. degrees in computer science from the University of Seoul, Seoul, South Korea, in 2011 and 2015, respectively.

He is currently pursuing the Ph.D. degree with the School of Electrical and Computer Engineering, Ulsan National Institute of Science and Technology (UNIST), Ulsan, South Korea.

His research interests include server storage technologies and graph processing systems.

SAM H. (HYUK) NOH received the B.S. degree in computer engineering from Seoul National University, Seoul, South Korea, in 1986, and the Ph.D. degree from the Department of Computer Science, University of Maryland, College Park, MD, USA, in 1993. He held a visiting faculty position at the George Washington University, Washington, DC, USA, from 1993 to 1994, before joining Hongik University, Seoul, where he was a Professor with the School of Computer and Information Engineering until the spring 2015. In the fall 2015, he joined the Ulsan National Institute of Science and Technology (UNIST), a young science and tech focused national university, where he is currently a Professor (and a former Dean from 2016 to 2018) with the School of Electrical and Computer Engineering. From August 2001 to August 2002, he was a Visiting Associate Professor with the Institute of Advanced Computer Studies (UMIACS), University of Maryland. His current research interests include operating system issues pertaining to embedded/computer systems with a focus on the use of new memory technologies, such as flash memory and persistent memory. He has served as the General Chair, the Program Chair, Steering Committee Member, and Program Committee Member on a number of technical conferences and workshops, including the ACM Eurosys, USENIX FAST (Co-PC Chair), ACM SOSP, ACM EMSOFT, USENIX ATC, IEEE RTAS, ACM ASPLOS, USENIX HotStorage (Co-PC Chair), USENIX OSDI, ACM LCTES (General Chair), IEEE ICPADS and WWW, among others. He has also been serving as the Editor-in-Chief of the *ACM Transactions on Storage* since 2016. He is a Distinguished Member of the ACM and a member of USENIX and KIISE.