FlashFQ: A Fair Queueing I/O Scheduler for Flash-Based SSDs

Kai Shen Stan Park
Department of Computer Science, University of Rochester

Abstract

On Flash-based solid-state disks (SSDs), different I/O operations (reads vs. writes, operations of different sizes) incur substantially different resource usage. This presents challenges for fair resource management in multi-programmed computer systems and multi-tenant cloud systems. Existing timeslice-based I/O schedulers achieve fairness at the cost of poor responsiveness, particularly when a large number of tasks compete for I/O simultaneously. At the same time, the diminished benefits of I/O spatial proximity on SSDs motivate fine-grained fair queueing approaches that do not enforce task-specific timeslices. This paper develops a new Flash I/O scheduler called FlashFQ. It enhances the start-time fair queueing schedulers with throttled dispatch to exploit restricted Flash I/O parallelism without losing fairness. It also employs I/O anticipation to minimize fairness violation due to deceptive idleness. We implemented FlashFQ in Linux and compared it with several existing I/O schedulers—Linux CFQ [2], an Argon [19]-inspired quanta scheduler, FIOS timeslice scheduler [17], FIOS with short timeslices, and 4-Tag SFQ(D) [11]. Results on synthetic I/O benchmarks, the Apache web server, and Kyoto Cabinet key-value store demonstrate that only FlashFQ can achieve both fairness and high responsiveness on Flash-based SSDs.

1 Introduction

NAND Flash devices are increasingly used as solid-state disks (SSDs) in computer systems. Compared to traditional secondary storage, Flash-based SSDs deliver much higher I/O performance which can alleviate the I/O bottlenecks in critical data-intensive applications. At the same time, the SSD resource management must recognize unique Flash characteristics and work with increasingly sophisticated firmware management. For instance, while the raw Flash device desires sequential writes, the write-order-based block mapping on modern SSD firmware can translate random write patterns into sequential writes on Flash and thereby relieve this burden for the software I/O scheduler. On the other hand, different I/O operations on Flash-based SSDs may exhibit large resource usage discrepancy. For instance, a write can consume much longer device time than a read due to the erase-before-write limitation on Flash. In addition, a larger I/O operation can take much longer than a small request does (unlike on a mechanical disk when both are dominated by mechanical seek/rotation delays). Without careful regulation, heavy resource-consuming I/O operations can unfairly block light operations.

Fair I/O resource management is desirable in a multi-programmed computer system or a multi-tenant cloud platform. Existing I/O schedulers including Linux CFQ [2], Argon [19], and our own FIOS [17] achieve fairness by assigning timeslices to tasks that simultaneously compete for the I/O resource. One critical drawback for this approach is that the tasks that complete their timeslices early may experience long periods of unresponsiveness before their timeslices are replenished in the next epoch. Such unresponsiveness is particularly severe when one must wait for a large number of co-running tasks in the system to complete their timeslices. Poor responsiveness is harmful but unnecessary on Flash-based SSDs that often complete an I/O operation in a fraction of a millisecond.

High responsiveness is supported by classic fair queueing approaches that originated from network packet switching [6, 8, 9, 16] but were also used in storage systems [3, 11, 18]. They allow fine-grained interleaving of requests from multiple tasks / flows as long as fair resource utilization is maintained through balanced virtual time progression. The lagging virtual time for an inactive task / flow is brought forward to avoid a large burst of requests from one task / flow and prolonged unresponsiveness for others. One drawback for fine-grained fair queueing on mechanical disks is that frequent task switches induce high seek and rotation costs. Fortunately, this is only a minor concern for Flash-based SSDs due to diminished benefits of I/O spatial proximity on modern SSD firmware.

This paper presents a new operating system I/O scheduler (called *FlashFQ*) that achieves fairness and high responsiveness at the same time. FlashFQ enhances the start-time fair queueing scheduler SFQ(D) [11] with two new mechanisms to support I/O on Flash-based SSDs. First, while SFQ(D) allows concurrent dispatch of requests (called *depth*) to exploit I/O parallelism, it violates fairness when parallel I/O operations on a Flash device interfere with each other. We introduce a throttled dispatch technique to exploit restricted Flash I/O parallelism without losing fairness. Second, existing fair queueing schedulers are work-conserving—they never idle the device when there is pending work to do. However, work-conserving I/O schedulers are susceptible to deceptive idleness [10] that causes fairness violation. We propose anticipatory fair queueing to mitigate the effects of deceptive idleness. We have implemented FlashFQ with the throttled dispatch and anticipatory fair queueing mechanisms in Linux.

The rest of this paper is organized as follows. Section 2 discusses related work. Section 3 characterizes key motivations and challenges for our fair queueing scheduler on Flash-based SSDs. Sections 4 and 5 present the design techniques and implementation issues in our FlashFQ scheduler. Section 6 illustrates our experimental evaluation and comparison with several alternative I/O schedulers. Section 7 concludes this paper with a summary of our findings.

2 Related Work

Flash I/O characterization and operating system support have been recognized in research. Agrawal *et al.* [1] discussed the impact of block erasure (before writes) and parallelism on the performance of Flash-based SSDs. Work by Chen *et al.* [4] further examined strided access patterns and identified abnormal performance issues like those caused by storage fragmentation. File system work [5,14,15] attempted to improve the sequential write patterns through the use of log-structured file systems. These efforts are orthogonal to our research on Flash I/O scheduling.

New I/O scheduling heuristics were proposed to improve Flash I/O performance. In particular, write bundling [12], write block preferential [7], and page-aligned request merging/splitting [13] help match I/O requests with the underlying Flash device data layout. The effectiveness of these write alignment techniques, however, is limited on modern SSDs with write-order-based block mapping. Further, these Flash I/O schedulers have paid little attention to the issue of fairness.

Fairness-oriented resource scheduling has been extensively studied. Fairness can be realized through per-task timeslices (as in Linux CFQ [2], Argon [19], and FIOS [17]) and credits (as in the SARC rate controller [20]). The original fair queueing approaches, including Weighted Fair Queueing (WFQ) [6], Packet-by-Packet Generalized Processor Sharing (PGPS) [16], and Start-time Fair Queueing (SFQ) [8, 9], take virtual time-controlled request ordering over several task queues to maintain fairness. While they are designed for network packet scheduling, later fair queueing approaches like Cello's proportionate class-independent scheduler [18], YFQ [3] and SFQ(D) [11] are adapted to support I/O resources. In particular, they allow the flexibility to reorder and parallelize I/O requests for better efficiency. Most of these fair-share schedulers (with the only exception of FIOS) do not address unique characteristics on Flash-based SSDs and many (including FIOS) do not support high responsiveness.

3 Motivations and Challenges

Timeslice Scheduling vs. Fair Queueing Timeslice-based I/O schedulers such as Linux CFQ, Argon, and FIOS achieve fairness by assigning timeslices to co-running tasks. A task that completes its timeslice early would have to wait for others to finish before its timeslice is replenished in the next epoch, leading to a period of unresponsiveness at the end of each epoch. Figure 1(A) illustrates this effect in timeslice scheduling. While some schedulers allow request interleaving (as shown in Figure 1(B)), the period of unresponsiveness still exists at the end of an epoch. This unresponsiveness is particularly severe in a highly loaded system where one must wait for a large number of co-running tasks to complete their timeslices. One may shorten the per-task timeslices to improve responsiveness. However, outstanding requests at the end of a timeslice may consume resources at the next timeslice that belongs to some other task. Such resource overuse leads to unfairness and this problem is particularly pronounced when each timeslice is short (Figure 1(C)).

In fine-grained fair queueing (as shown in Figure 1(D)), requests from multiple tasks are interleaved in a fine-grained fashion to enable fair progress by all tasks. It achieves fairness and high responsiveness at the same time. Furthermore, since fine-grained fair queueing does not restrict the request-issuing task in each timeslice, it works well with I/O devices possessing internal parallelism (Figure 1(E)).

Finally, due to substantial background maintenance such as Flash garbage collection, Flash-based SSDs provide time-varying capacities (more I/O capacity at one moment and less capacity at a later time). The timeslice scheduling that focuses on the equal allocation of device time may not provide fair shares of time-varying resource capacities to concurrent tasks. In contrast, the fair queueing scheduling targets equal progress of completed work and therefore it can achieve fairness even for resources with time-varying capacities.

Restricted Parallelism Flash-based SSDs have some built-in parallelism through the use of multiple channels. Within each channel, the Flash package may have multiple planes which are also parallel. We run experiments to understand such parallelism. We utilize the following Flash-based storage devices—

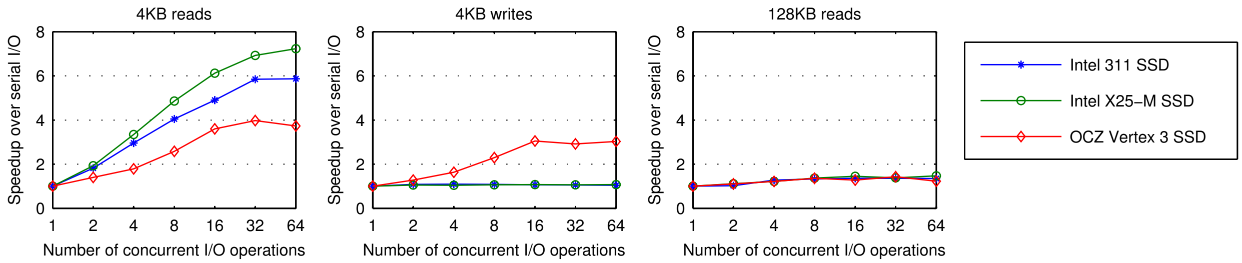

Figure 2: Efficiency of I/O parallelism (throughput speedup over serial I/O) for 4 KB reads, 4 KB writes, and 128 KB reads on three Flash-based SSDs.

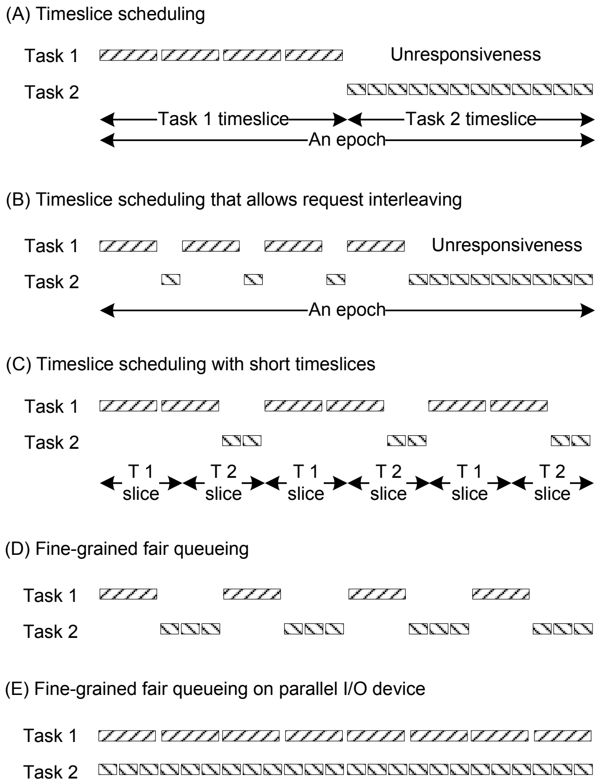

Figure 1: Fairness and responsiveness of timeslice scheduling and fine-grained fair queueing.

- An Intel 311 Flash-based SSD, released in 2011, using single-level cells (SLC) in which a particular cell stores a single bit of information.
- An Intel X25-M Flash-based SSD, released in 2009, using multi-level cells (MLC).
- An OCZ Vertex 3 Flash-based SSD, released in 2011, using MLC.

To acquire the native device properties, we bypass the memory buffer (through direct I/O) and operating system I/O scheduler (configuring Linux noop scheduler) in these measurements. We also disable the device write cache so that all writes reach the durable storage medium.

Figure 2 shows the efficiency of Flash I/O parallelism for 4 KB reads, 4 KB writes, and 128 KB reads on our SSDs. We observe that the parallel dispatch of multiple 4 KB reads to an SSD lead to substantial throughput enhancement (up to 4-fold, 6-fold, and 7-fold for the three SSDs respectively). However, the parallelism-induced speedup is diminished by writes and large reads. We observe significant write parallelism only on the Vertex 3 SSD. Large reads suppress the parallel efficiency because a large read already uses the multiple channels in a Flash device.

Such restricted parallelism leads to new challenges for a fair queueing I/O scheduler. On one hand, the scheduler should allow the simultaneous dispatch of multiple I/O requests to exploit the Flash parallel efficiency when available. On the other hand, it must recognize the unfairness resulting from the interference of concurrently dispatched I/O requests and mitigate it when necessary.

Diminished Benefits of Spatial Proximity One drawback for fine-grained fair queueing on mechanical disks is that frequent task switches lead to poor spatial proximity and consequently high seek and rotation costs. Fortunately, at the absence of such mechanical overhead, Flash I/O performance is not as dependent on the I/O spatial proximity. This is particularly the case for modern SSDs with write-order-based block mapping where random writes become spatially contiguous on Flash due to block remapping.

We run experiments to demonstrate such diminished benefits of spatial proximity. Besides the three Flash-based SSDs, we also include a conventional mechanical disk (a 10 KRPM Fujitsu SCSI drive) for the purpose of comparison. Figure 3 shows the performance discrepancies between random and sequential I/O on the storage devices. The random I/O performance is measured when each I/O operation is applied to a randomized offset address in a 256 MB file.

We observe that the sequential I/O for small (4 KB)

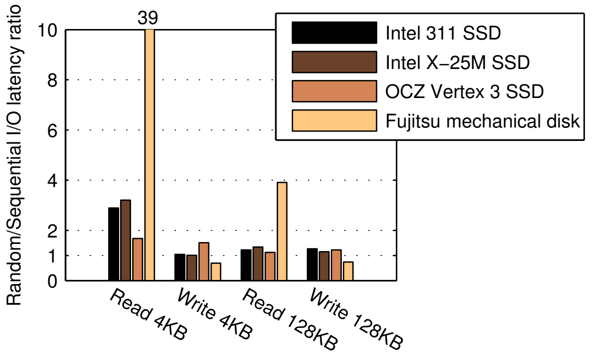

Figure 3: The ratios of random I/O latency over sequential I/O latency for reads / writes at two I/O request granularities (4 KB and 128 KB) on three Flash-based SSDs.

reads are still beneficial for some SSDs (3-fold speedup for the two Intel SSDs). However, such benefits are much diminished compared to the 39-fold sequential read speedup on the mechanical disk. The performance difference between random and sequential I/O is further diminished for writes and large-grained (128 KB) I/O requests on SSDs. These results also match the findings in Chen *et al.*'s 2009 paper [4] for mid- and high-end SSDs at the time. The diminished benefits of I/O spatial proximity mitigates a critical drawback for adopting fine-grained fair queueing on Flash-based SSDs. Particularly in the case of small reads, while the fine-grained fair queueing loses some sequential I/O efficiency, it gains the much larger benefit of I/O parallelism on small reads.

Note that the poor sequential write performance on the rotating mechanical disk in our measurement results is due to the disabling of its write cache, in which case all writes must reach the durable disk. Therefore the disk head has typically rotated beyond the desired location when a new write request arrives after a small delay (software system processing) from the completion of the previous operation. This effect is particularly pronounced in our measurement setup where the disk rotation time dominates the seek time since our random I/O addresses are limited in a small disk area (a 256 MB file).

Deceptive Idleness Fair queueing I/O schedulers like SFQ(D) [11] are work-conserving—they never idle the device when there is pending work to do. However, work-conserving I/O schedulers are susceptible to deceptive idleness [10]—an active task that issues the next request a short time after receiving the result of the previous one may temporarily appear to be idle. For fair queueing schedulers, the deceptive idleness may let an active task to be mistakenly considered as being "inactive". This can result in the premature advance of virtual time for such "inactive tasks" and therefore unfairness.

Consider the simple example of a concurrent run involving two tasks—one continuously issues I/O requests with heavy resource usage (*heavy task*) while the other continuously issues light I/O requests (*light task*). We further assume that the I/O scheduler issues one request to the device at a time (no parallelism). At the moment when a request from the light task completes, the only queued request is from the heavy task and therefore a work-conserving I/O scheduler will dispatch it to the device. This effectively results in one-request-at-a-time alternation between the two tasks and therefore unfairness favoring the heavy task.

4 FlashFQ Design

In a concurrent system, many resource principals simultaneously compete for a shared I/O resource. The scheduler should regulate I/O in such a way that accesses are fair. When the storage device time is the bottleneck resource in the system, fairness is the case that each resource principal acquires an equal amount of device time. At the same time, responsiveness requires that each user does not experience prolonged periods with no response to its I/O requests. We present the design of our FlashFQ I/O scheduler that achieves fairness and high responsiveness for Flash-based SSDs. It enhances the classic fair queueing approach with new techniques to address the problems of restricted parallelism and deceptive idleness described in the last section.

Practical systems may desire fairness and responsiveness for different kinds of resource principals. For example, a general-purpose operating system desires the support of fairness and responsiveness for concurrent applications. A server system wants such support for simultaneously running requests from multiple user classes. A shared hosting platform needs fairness and responsiveness for multiple active cloud services (possibly encapsulated in virtual machines). Our I/O scheduling design and much of our implementation can be generally applied to supporting arbitrary resource principals. When describing the FlashFQ design, we use the term *task* to represent the resource principal that receives the fairness and responsiveness support in a concurrent system.

4.1 Fair Queueing Preliminaries

As described in Section 2, a considerable number of fair queueing schedulers have been proposed in the past. Our FlashFQ design is specifically based on SFQ(D) [11] for two reasons. First, it inherits the advantage of Start-time Fair Queueing (SFQ) [8, 9] that the virtual time can be computed efficiently. Second, it allows the simultaneous dispatch of multiple requests which is necessary for exploiting the internal parallelism on Flash devices.

SFQ(D) maintains a system-wide virtual time $v(t)$. It uses the virtual time to assign a start and finish tag to each arriving request. The start tag is the larger of the current system virtual time (at the request arrival time) and the finish tag of the last request by the same task. The finish tag is the start tag plus the expected resource usage of the request. Request dispatch is ordered by each pending request's start tag. Multiple requests (up to the depth D) can be dispatched to the device at the same time.

A key issue with SFQ(D) is the way the virtual time $v(t)$ is advanced and the related treatment of *lagging tasks*—those that are slower than others in utilizing allotted resources. If $v(t)$ advances too quickly, it could artificially bring forward the request start tags of lagging tasks such that their unused resources are forfeited which leads to unfairness. On the other hand, if the virtual time advances too slowly, it could allow a lagging task to build up its unused resources and utilize them in a sudden burst of request arrivals that cause prolonged unresponsiveness to others. Three versions of the scheduler were proposed [11], with different ways of maintaining the system virtual time—

- **Min-SFQ(D)** assigns the virtual time $v(t)$ to be the minimum start tag of any outstanding request at time t. A request is *outstanding* if it has arrived but not yet completed. A key problem with Min-SFQ(D) is that its virtual time advances too slowly which makes it susceptible to unresponsiveness caused by a sudden burst of requests from a lagging task described above.
- **Max-SFQ(D)**[1] assigns the virtual time $v(t)$ to be the maximum start tag of dispatched requests on or before time t. A drawback with this approach is that its virtual time may advance too quickly and result in unfairness as described above.
- **4-Tag SFQ(D)** attempts to combine the above two approaches to mitigate each's problem. Specifically, it maintains two pairs of start / finish tags for each request according to Min-SFQ(D) and Max-SFQ(D) respectively. The request dispatch ordering is primarily based on the Max-SFQ(D) start tags while ties are broken using Min-SFQ(D) start tags.

4.2 Min-SFQ(D) with Throttled Dispatch

The characterization in Section 3 shows that Flash-based SSDs exhibit restricted parallelism—while parallel executions can sometimes produce higher throughput, simultaneously dispatched requests may also interfere with each other on the Flash device. Utilizing such restricted parallelism may lead to uncontrolled resource usage under any version of the SFQ(D) schedulers described above. Consider two tasks running together in the system and each task issues no more than one request at a time. If the I/O scheduler depth $D \geq 2$, then requests of both tasks will be dispatched to the device without delay at the scheduler. Interference at the Flash device often results in unbalanced resource utilization between the two tasks.

While such unbalanced resource utilization affects all three SFQ(D) versions, it is particularly problematic for Max-SFQ(D) and 4-Tag SFQ(D) who advance the system virtual time too quickly—any request dispatch from an aggressive task leads to an advance of the system virtual time, and consequently the forfeiture of unused resources by the lagging tasks. In comparison, Min-SFQ(D) properly accounts for the unbalanced resource utilization for all active tasks. Therefore we employ Min-SFQ(D) as the foundation of our scheduler.

Proper resource accounting alone is insufficient for fairness, we need an additional control to mitigate the imbalance of resource utilization between concurrent tasks. Our solution is a new *throttled dispatch* mechanism. Specifically, we monitor the relative progresses of concurrently active tasks and block a request dispatch if the progress of its issuing task is excessively ahead of the most lagging task in the system (i.e, the difference between those tasks' progress exceeds a threshold). Under SFQ(D) schedulers, the progress of a task is represented by its last dispatched start tag—the start tag of its most recently dispatched request. When requests from aggressive tasks (using more resources relative to their shares) are blocked, lagging tasks can catch up with less interference at the device. The blocking is relieved as soon as the imbalance of resource utilization falls below the triggering threshold.

4.3 Anticipation for Fairness

A basic principle of fair queueing scheduling is that when a task becomes inactive (it has no I/O requests to issue), its resource share is not allowed to accumulate. The rationale is simple—one has no claim to resources when it has no intention of using them. Even Min-SFQ(D)—which, among the three SFQ(D) versions, advances the system virtual time most conservatively—ignores tasks that do not have any outstanding I/O requests. As explained in Section 3, this approach may mistakenly consider an active task to be "inactive" due to deceptive idleness in I/O—an active task that issues the next request a short time after receiving the result of the previous one may temporarily appear to be idle to the I/O scheduler. Even during a very short period of deceptive idleness, the system virtual time may advance with no regard to the deceptively "inactive" task, leading to the forfeiture of its unused resources.

[1] This version is called SFQ(D) in the original paper [11]. We use a different name to avoid the confusion with the general reference to all three SFQ(D) versions.

The deceptive idleness was first recognized to cause undesirable task switches on mechanical disks that result in high seek and rotation delays. It was addressed by anticipatory I/O [10] which temporarily idles the disk (despite the existence of pending requests) to hope for a soon-arriving new request (typically issued by the process that is receiving the result of the just completed request) with better locality. We adopt anticipatory I/O for a different purpose—ensuring the continuity of a task's "active" status when deceptive idleness appears between its two consecutive requests. Specifically, when a synchronous I/O request completes, the task that will be receiving the result of the just completed request is allowed to stay "active" for a certain period of time. During this period, we adjust Min-SFQ(D) to consider the anticipated next request from the task as a hypothetical outstanding request in its virtual time maintenance. The start tag for the anticipated request, if arriving before the anticipation expires, should be the finish tag of the last request by the task.

The "active" status anticipation ensures that an active task's unused resources are not forfeited during deceptive idleness. It also enables the dispatch-blocking of excessively aggressive tasks (described in Section 4.2) when the lagging task is deceptively idle for a short amount of time. While both goals are important, anticipation for these two cases have different implications. Specifically, the anticipation that blocks the request dispatch from aggressive tasks is not work-conserving—it may leave the device idle while there is pending work—and therefore may waste resources. We distinguish these two anticipation purposes and allow a shorter timeout for the non-work-conserving anticipation that blocks aggressive tasks.

4.4 Knowledge of Request Cost

Recall that the request finish tag assignment requires knowledge of the resource usage of a request (or its cost). The determination of a request's cost is an important problem for realizing our fair queueing scheduler in practice and it deserves a careful discussion.

A basic question on this problem is by what time a request's cost must be known. This question is relevant because it may be easier to estimate a request's cost after its completion. According to our design, this is when the request's finish tag is assigned. In theory, for fair queueing schedulers that schedule requests based on their start tag ordering [8, 9, 11] (including ours), only the start tag assignments are directly needed for scheduling. A request's finish tag assignment can be delayed to when it is needed to compute some other request's start tag. In particular, one request (r_1)'s finish tag is needed to compute the start tag of the next arriving request (r_2) by the same task. Since the two requests may be dispatched in parallel, r_2's start tag (and consequently r_1's finish tag) might be needed before r_1's completion.

Given the potential need of knowing request costs early, our system estimates a request's cost at the time of its arrival. Specifically, we model the cost of a Flash I/O request based on its access type (read / write) and its data size. For reads and writes respectively, we assume a linear model (typically with a substantial nonzero offset) between the cost and data size of an I/O request. Our estimation model requires the offline calibration of the Flash I/O time for only four data access cases—read 4 KB, read 128 KB, write 4 KB, and write 128 KB. In general, such calibration is performed once for each device. Additional (but infrequent) calibrations can be performed for devices whose gradual wearout affects their I/O performance characteristics.

5 Implementation Issues

FlashFQ can be implemented in an operating system to regulate I/O resource usage by concurrent applications. It can also be implemented in a virtual machine monitor to allocate I/O resources among active virtual machines. As a prototype, we have implemented our FlashFQ scheduler in Linux 2.6.33.4. Below we describe several notable implementation issues.

Implementation in Linux An important feature of Linux I/O schedulers is the support of plugging and request merging—request queue is plugged (blocking request dispatches) temporarily to allow physically contiguous requests to merge into a larger request before dispatch. This is beneficial since serving a single large request is much more efficient than serving multiple small requests. Request merging, however, is challenging for our FlashFQ scheduler due to the need of re-computing request tags and task virtual time when two requests merge. For simplicity, we only implemented the most common case of request back-merging—merging a new arriving request (r_2) to an existing queued request (r_1) if r_2 contiguously follows (on the back of) r_1.

While the original anticipatory I/O [10] requires a single timer, our anticipation support may require multiple outstanding timers due to the nature of parallelism in our scheduler. Specifically, we may need to track deceptive idleness of multiple parallel tasks. To minimize the cost of parallel timer management, our implementation maintains a list of pending timers ranked by their fire time (we call them *logical timers*). Only the first logical timer (with the soonest fire time) is supported by a physical system timer. Most logical timer manipulations (add / delete timers) do not involve the physical system timer unless the first logical timer is changed.

Our prototype implementation runs on the ext4 file system. We mount the file system with the `noatime` option to avoid metadata updates on file reads. Note that the metadata updates (on modification timestamps) are still necessary for file writes. The original ext4 file system uses very fine-grained file timestamps (in nanoseconds) so that each file write always leads to a new modification time and thus triggers an additional metadata write. This is unnecessarily burdensome to many write-intensive applications. We revert back to file timestamps in the granularity of seconds (which is the default in Linux file systems that do not make customized settings). In this case, at most one timestamp metadata write per second is needed regardless how often the file is modified.

Parameter Settings We describe important parameter settings and their tuning guidelines in FlashFQ. The depth D in SFQ(D) represents the maximum device dispatch parallelism. A higher depth allows the exploitation of more parallel efficiency (if supported on the device) while large parallel dispatches weaken the scheduler's ability to regulate I/O resources in a fine-grained fashion. Our basic principle is to set a minimum depth that can exploit most of the device-level I/O parallelism. According to the parallel efficiency of the three SSDs in Figure 2, we set the depth D to 16 for all three SSDs.

For throttled dispatch, we set the task progress difference threshold that triggers the dispatch-blocking to be 100 millisecs. This threshold represents a tradeoff between fairness and efficiency—how much temporary resource utilization imbalance is tolerated to utilize restricted device parallelism?

The "active" status anticipation timeout is set to 20 millisecs—a task is considered to be continuously active as long as its inter-request thinktime does not exceed 20 millisecs. We set a shorter timeout (2 millisecs) for the anticipation that blocks aggressive tasks while leaving the device idle. The latter anticipation timeout is shorter because it may waste resources (as explained in Section 4.3).

I/O Context Our FlashFQ design in Section 4 uses a *task* to represent a resource principal that receives fairness support. In Linux I/O schedulers, each resource principal is represented by an *I/O context*. By default, a unique I/O context is created for each process or thread. However, it is sometimes more desirable to group a number of related processes as a single resource principal—for instance, all `httpd` processes in an Apache web server. In Linux, such grouping is accomplished for a set of processes created by the `fork()`/`clone()` system call with the `CLONE_IO` flag. We added the `CLONE_IO` flag to relevant `fork()` system calls in the Apache web server so that all `httpd` processes in a web server share a unified I/O context. We also fixed a problem in the original Linux that fails to unify the I/O context if `fork(CLONE_IO)` is called when the parent process has not yet initialized its I/O context.

One problem we observed in our Linux/ext4-based prototyping and experimentation is that the journaling-related I/O requests are issued from the I/O context of the JBD2 journaling daemon and they compete for I/O resources as if they represent a separate resource principal. However, since journaling I/O are by-products of higher-layer I/O requests originated from applications, ideally they should be accounted in the I/O contexts of respective original applications. We have not yet implemented this accounting in our current prototype. To avoid resource mis-management due to the JBD2 I/O context in Linux, we disabled ext4 journaling in our experimental evaluation.

6 Experimental Evaluation

We compare FlashFQ against alternative fairness-oriented I/O schedulers. One alternative is Linux CFQ. The second alternative (*Quanta*) is our implementation of a quanta-based I/O scheduler that follows the basic principles in Argon [19]. Quanta puts a high priority on achieving fair resource use (even if some tasks only have partial I/O load). All tasks take round robin turns of I/O quanta. Each task has exclusive access to the storage device within its quantum. Once an I/O quantum begins, it will last to its end, regardless of how few requests are issued by the corresponding task. However, a quantum will not begin, if no request from the corresponding task is pending. The third alternative is the FIOS I/O scheduler developed in our earlier work [17]. FIOS allows simultaneous request dispatches from multiple tasks to exploit Flash I/O parallelism, as long as the per-task timeslice constraint is maintained. FIOS also prioritizes reads over writes and it reclaims unused resources by inactive tasks. The fourth alternative is 4-Tag SFQ(D) [11]. Finally, we compare against the raw device I/O in which requests are always dispatched immediately (without delay) to the storage device.

Three of the alternative schedulers (Linux CFQ, Quanta, and FIOS) are timeslice-based. Timeslice parameters for these schedulers follow the default settings for synchronous I/O operations in Linux. Specifically, Linux tries to limit the epoch size and the maximum unresponsiveness at 300 millisecs. Therefore when multiple (n) tasks compete for I/O simultaneously, the per-task timeslice is set at $\frac{300}{n}$ millisecs. This setting is subject to the lower bound of 16 millisecs and the upper bound of 100 millisecs in Linux. To assess the effect of timeslice scheduling with short timeslices, we include a new setting that configures the per-task timeslice at $\frac{60}{n}$ millisecs when n tasks compete for I/O simultaneously (with the goal of limiting the maximum unresponsiveness at 60 millisecs). We also shorten the timeslice lower bound to 1 millisec. We include FIOS with such short timeslice setting in our evaluation and we call it *FIOS-ShortTS*.

Our experiments utilize the three Flash-based storage devices (Intel 311, Intel X25-M, and OCZ Vertex 3 SSDs) that were described earlier in Section 3. On both Intel SSDs, writes are substantially slower than reads (by about 4-fold and 6-fold on Intel 311 and Intel X25-M respectively). The Vertex drive employs a SandForce controller which supports new write acceleration techniques such as online compression. The Vertex write performance only moderately lags behind the read performance. For instance, a 4 KB read and a 4 KB write take 0.18 and 0.22 millisec respectively on the drive.

6.1 Evaluation on Task Fairness

Fairness is defined as the case that each task gains its share of resources in concurrent execution. When n tasks compete for I/O simultaneously, equal resource sharing suggests that each task should experience a factor of n slowdown compared to running-alone, or *proportional slowdown*. This is our first fairness measure. We further note that better performance for some tasks may be achieved when others do not utilize all of their allotted resource shares. Some tasks may also gain better I/O efficiency during concurrent runs by exploiting the device-level I/O parallelism. When all tasks experience better performance than the proportional slowdown, we further measure fairness according to the slowdown of the slowest task. Specifically, scheduler $\mathcal{S}_1$ achieves better fairness than scheduler $\mathcal{S}_2$ if the slowest task under $\mathcal{S}_1$ makes more progress than the slowest task does under $\mathcal{S}_2$.

We use a variety of synthetic I/O benchmarks to evaluate the scheduling fairness in different resource competition scenarios. Each benchmark contains a number of tasks issuing I/O requests of different types and sizes—

- a concurrent run with a reader continuously issuing 4 KB reads and a writer continuously issuing 4 KB writes;
- a concurrent run with sixteen 4 KB readers and sixteen 4 KB writers;
- a concurrent run with sixteen 4 KB readers and sixteen 128 KB readers;
- a concurrent run with sixteen 4 KB writers and sixteen 128 KB writers.

In order for these I/O patterns to reach the I/O scheduler at the block device layer, we perform direct I/O to bypass the memory buffer in these tests.

Figure 4 shows the fairness and performance under different schedulers. The raw device I/O, Linux CFQ, and 4-Tag SFQ(D) fail to achieve fairness by substantially missing the proportional slowdown in many cases. Specifically, lighter tasks (issuing reads instead of writes, issuing smaller I/O operations instead of larger ones) experience many times the proportional slowdown while heavy tasks experience much less slowdown in concurrent runs. Because raw device I/O makes no scheduling attempt, I/O operations are interleaved as they are issued by applications, severely affecting the response of light requests. The Linux CFQ does not perform much better because it disables I/O anticipation for non-rotating storage devices like Flash. For instance, without anticipation, two-task executions degenerate to one-request-at-a-time alternation between the two tasks and therefore poor fairness. 4-Tag SFQ(D) also suffers from poor fairness since its unthrottled parallel dispatches make it behave like the raw device I/O in many cases.

Under the Quanta scheduler, tasks generally experience similar slowdown in most cases. But such "fairness" is attained at substantial degradation of I/O efficiency due to its aggressive maintenance of per-task quantum. Specifically, its strict quanta enforcement throws away unused resources by some tasks. It also fails to exploit device I/O parallelism, as demonstrated by its poor performance in cases with large numbers of concurrent tasks.

Both FIOS and FlashFQ maintain fairness (approximately at or below proportional slowdown) in all the evaluation cases. Furthermore, both FIOS and FlashFQ can exploit the device I/O parallelism when available and achieve the best performance in all evaluation cases.

FIOS-ShortTS achieves good fairness for the single reader, single writer case (first row in Figure 4). However, it exhibits degraded fairness (compared to the original FIOS and FlashFQ) in cases with large numbers of concurrent tasks due to very short timeslices. In particular, it fails to maintain proportional slowdown for 16 4KB-writers, 16 128KB-writers on the two Intel SSDs (substantially so on Intel X25-M). It also produces relatively poor worst-task-slowdown compared to the original FIOS and FlashFQ in some other cases (particularly tests with 16 4KB-readers, 16 128KB-readers).

6.2 Evaluation on Responsiveness

The fairness evaluation shows that only FIOS and FlashFQ consistently achieve fairness for a variety of workload scenarios on the three SSDs. As a timeslice scheduler, however, FIOS achieves fairness at the cost of poor responsiveness. Even though FIOS allows simultaneous request dispatches from multiple tasks, the timeslice constraint at the end of each epoch still leads to long unresponsiveness for light tasks who complete their timeslices early.

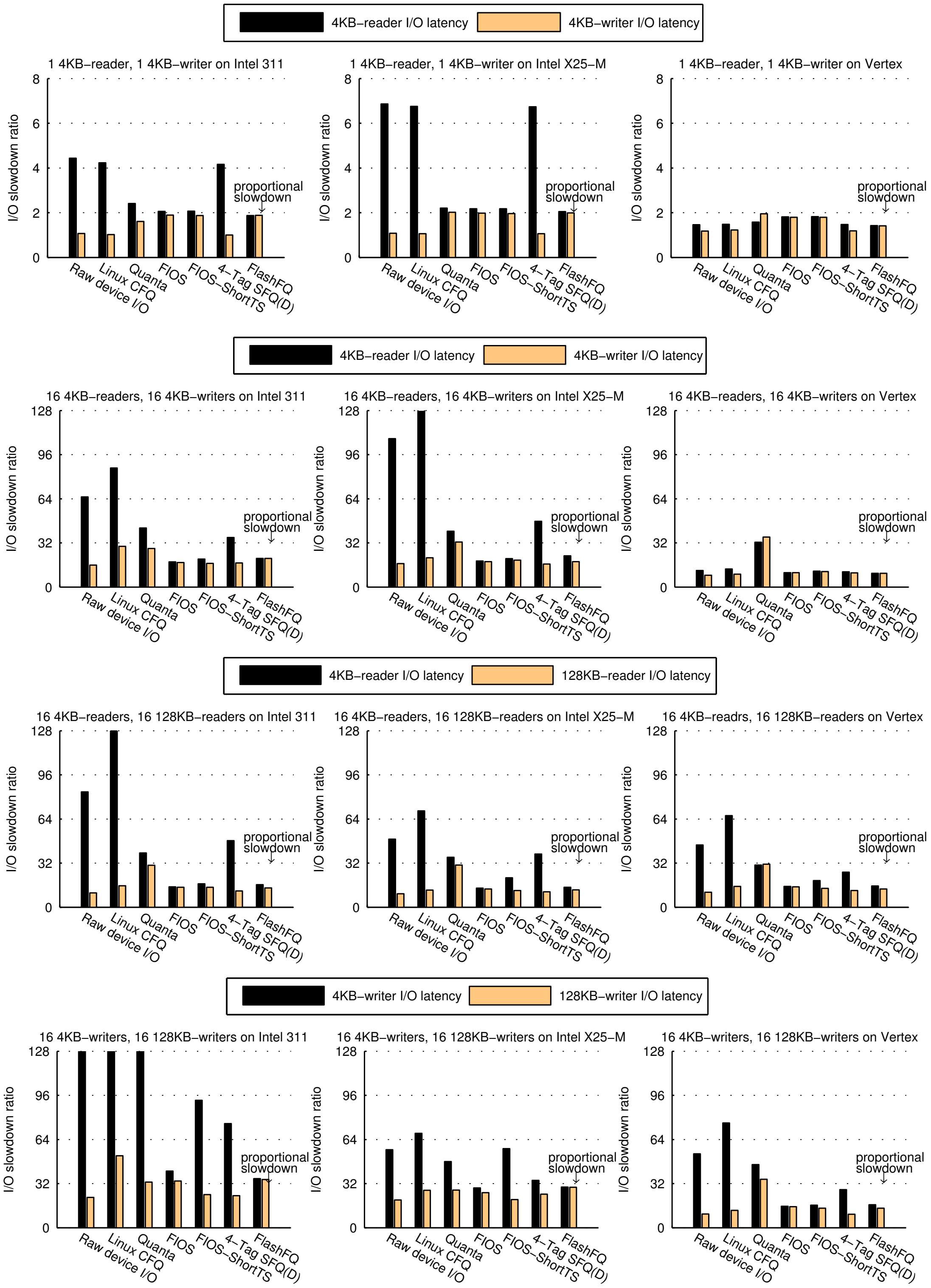

Figure 4: Fairness and performance of synthetic benchmarks under different I/O schedulers. The *I/O slowdown ratio* for a task is its average I/O latency normalized to that when running alone. For a run with multiple tasks per class (e.g., 16 readers and 16 writers), we only show the performance of the slowest task per class (e.g., the slowest reader and slowest writer). Results cover four workload scenarios (corresponding to the four rows) and three Flash-based SSDs (corresponding to the three columns). For each case, we mark the slowdown ratio that is proportional to the total number of tasks in the system.

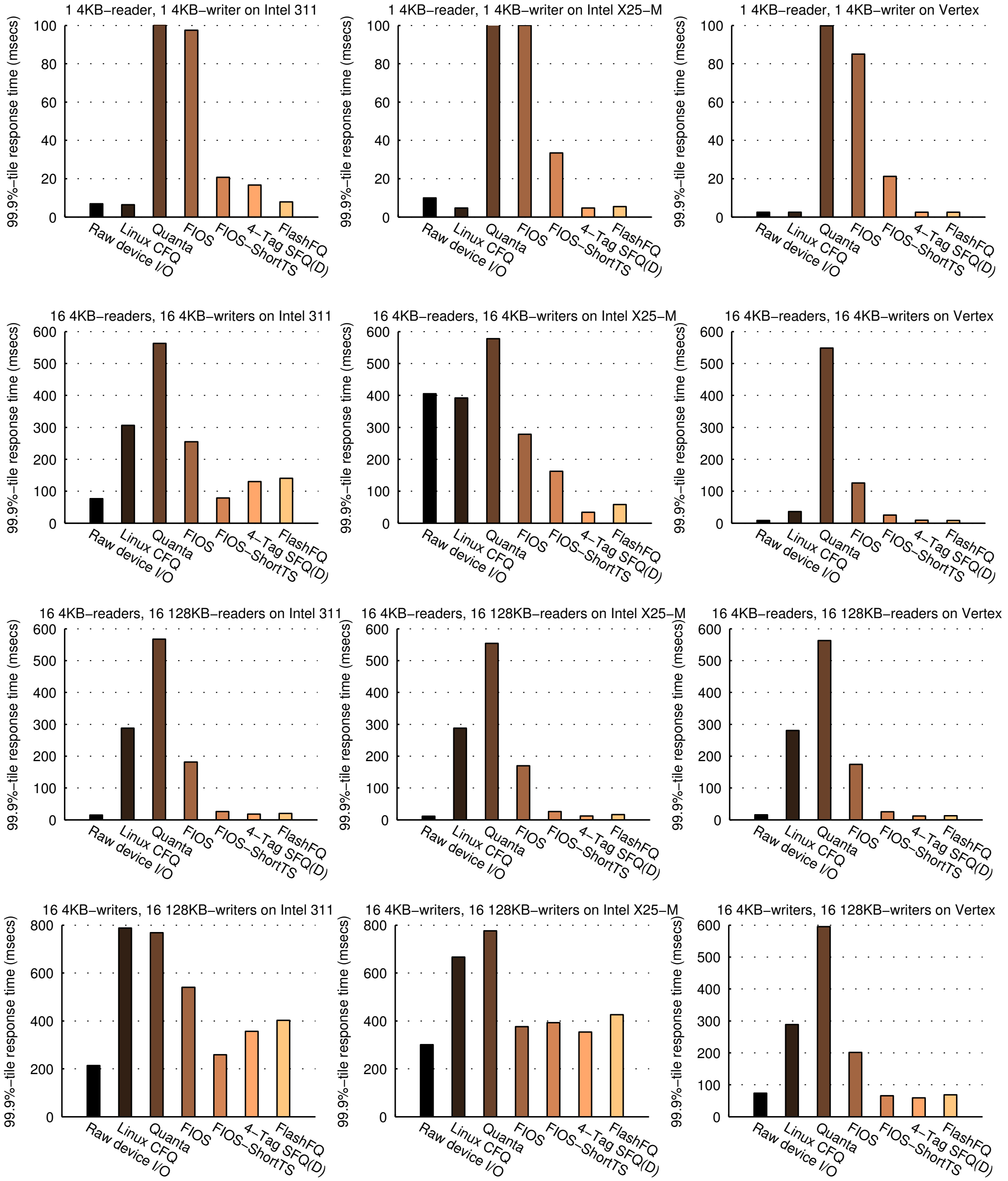

Figure 5: Worst-case (99.9-percentile) response time for four workload scenarios (rows) on three SSDs (columns) under different I/O schedulers.

In a system with high responsiveness, no task should experience prolonged periods of no response to its outstanding requests. We use the worst-case (99.9-percentile) I/O request response time during the execution as a measure of the system responsiveness. Figure 5 shows the responsiveness for our four workload scenarios on the three SSDs. Results clearly show poor responsiveness for the three timeslice schedulers (Linux CFQ, Quanta, and FIOS) in many of the test scenarios. In particular, they exhibit worst-case response time at half a second or more in some highly concurrent executions.

In comparison, FlashFQ shows much better responsiveness than these approaches (reaching an order of magnitude response time reduction in many cases). At the same time, we observe that FlashFQ's worst-case response time is quite long for the case of 16 4KB-writers and 16 128KB-writers on the two Intel drives (left two plots in the bottom row). This is due to the long write

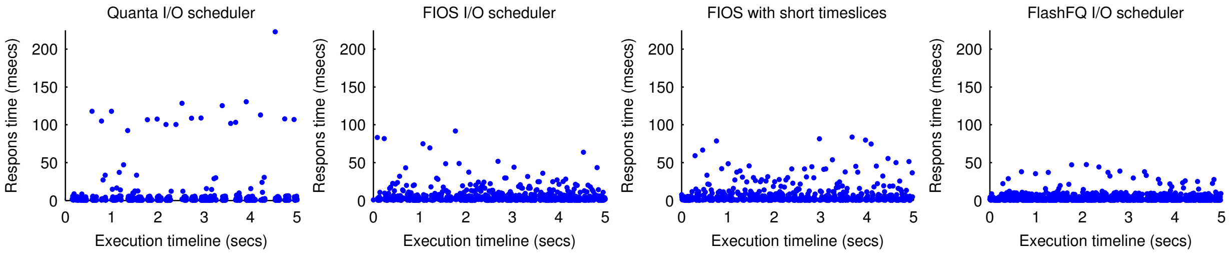

Figure 7: Time of Apache request responses under Quanta, FIOS, FIOS-ShortTS, and FlashFQ I/O schedulers. Each dot represents a request, whose X-coordinate indicates its timestamp in the execution while its Y-coordinate indicates its response time.

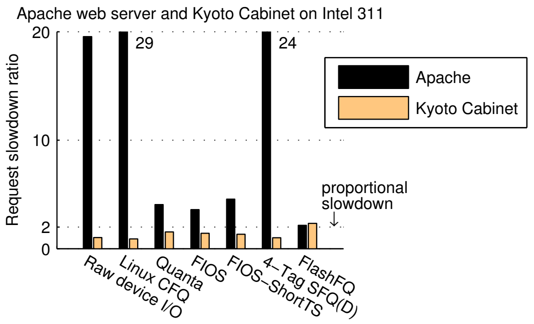

Figure 6: Fairness and performance of the read-only Apache web server workload running with a write-mostly Kyoto Cabinet key-value workload. The slowdown ratio for an application is its average request response time normalized to that when running alone.

time on these drives and the sheer amount of time to simply iterate through all 32 tasks while processing at least one request from each. This is evidenced by the long response time even under raw device I/O.

FIOS-ShortTS indeed exhibits much better responsiveness than the original FIOS. But this comes at the cost of degraded fairness (as shown in Section 6.1). Furthermore, FlashFQ still achieves better responsiveness than FIOS-ShortTS as any timeslice maintenance (even for very short timeslices) adds some scheduling constraint that impedes the system responsiveness.

6.3 Evaluation with the Apache Web Server and Kyoto Cabinet

Beyond the synthetic benchmarks, we evaluate the effect of I/O schedulers using realistic data-intensive applications. We run the Apache 2.2.3 web server over a set of HTTP objects according to the size distribution in the SPECweb99 specification. The total data size is 15 GB and the workload is I/O-intensive on a machine with 2 GB memory. As explained in Section 5, we attached the `CLONE_IO` flag to relevant `fork()` system calls in the Apache web server so that all `httpd` processes in the web server share a unified I/O context. Our web server is driven by a client that issues requests back-to-back (i.e., issuing a new request as soon as the previous one returns). The client runs on a different machine in a low-latency local area network.

Together with the read-only web server, we run a write-intensive workload on the Kyoto Cabinet 1.2.76 key-value store. In our workload, the value field of each key-value record is 128 KB. We pre-populate 1000 records in a database and our test workload issues "replace" requests each of which updates the value of a randomly chosen existing record. Each record replace is performed in a synchronous transaction supported by Kyoto Cabinet. In our workload, eight back-to-back clients operate on eight separate Kyoto Cabinet databases. All databases belong to a single I/O context that competes with the Apache I/O context.

Figure 6 illustrates the fairness under different I/O schedulers on the Intel 311 SSD. Since the Kyoto Cabinet workload consists of large write requests at high concurrency, it tends to be an aggressive I/O resource consumer and the Apache workload is naturally susceptible to more slowdown. Among the seven scheduling approaches, only FlashFQ can approximately meet the fairness goal of proportional slowdown for both applications. Among the alternatives, Quanta, FIOS and FIOS-ShortTS exhibit better fairness than others. Specifically, the Apache slowdown under Quanta, FIOS and FIOS-ShortTS are 4.1×, 3.6×, and 4.6× respectively.

Among the four schedulers with best fairness (Quanta, FIOS, FIOS-ShortTS, and FlashFQ), we illustrate the timeline of Apache request responses in Figure 7. Under Quanta, we observe periodic long responses (up to 200 millisecs) due to its timeslice management. The worst-case responses are around 100 millisecs under FIOS and FIOS-ShortTS. In comparison, FlashFQ achieves the best responsiveness with all requests responded within 50 millisecs.

7 Conclusion

This paper presents FlashFQ—a new Flash I/O scheduler that attains fairness and high responsiveness at the same time. The design of FlashFQ is motivated by unique characteristics on Flash-based SSDs—1) restricted parallelism with interference on SSDs presents a tension between efficiency and fairness, and 2) the diminished benefits of I/O spatial proximity on SSDs allow fine-grained task interleaving without much loss of I/O performance. FlashFQ enhances the start-time fair queueing schedulers with throttled dispatch to exploit restricted Flash I/O parallelism without losing fairness. It also employs I/O anticipation to minimize fairness violation due to deceptive idleness. We evaluated FlashFQ's fairness and responsiveness and compared against several alternative schedulers. Only FIOS [17] achieves fairness as well as FlashFQ does but it exhibits much worse responsiveness. FIOS with short timeslices can improve its responsiveness, but it does so at the cost of degraded fairness.

Acknowledgments This work was supported in part by the National Science Foundation grants CCF-0937571, CNS-1217372, and CNS-1239423. Kai Shen was also supported by a Google Research Award. We thank Jeff Chase for clarifying the design of the SFQ(D) scheduler. We also thank the anonymous USENIX ATC reviewers and our shepherd Prashant Shenoy for comments that helped improve this paper.

References

[1] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy. Design tradeoffs for SSD performance. In *USENIX Annual Technical Conf.*, pages 57–70, Boston, MA, June 2008.

[2] J. Axboe. Linux block IO — present and future. In *Ottawa Linux Symp.*, pages 51–61, Ottawa, Canada, July 2004.

[3] J. Bruno, J. Brustoloni, E. Gabber, B. Ozden, and A. Silberschatz. Disk scheduling with quality of service quarantees. In *IEEE Int'l Conf. on Multimedia Computing and Systems*, pages 400–405, Florence , Italy, June 1999.

[4] F. Chen, D. A. Koufaty, and X. Zhang. Understanding intrinsic characteristics and system implications of Flash memory based solid state drives. In *ACM SIGMETRICS*, pages 181–192, Seattle, WA, June 2009.

[5] H. Dai, M. Neufeld, and R. Han. ELF: An efficient log-structured Flash file system for micro sensor nodes. In *SenSys'04: Second ACM Conf. on Embedded Networked Sensor Systems*, pages 176–187, Baltimore, MD, Nov. 2004.

[6] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. In *ACM SIGCOMM*, pages 1–12, Austin, TX, Sept. 1989.

[7] M. Dunn and A. L. N. Reddy. A new I/O scheduler for solid state devices. Technical Report TAMU-ECE-2009-02, Dept. of Electrical and Computer Engineering, Texas A&M Univ., Apr. 2009.

[8] P. Goyal, H. M. Vin, and H. Cheng. Start-time fair queueing: A scheduling algorithm for integrated services packet switching networks. *IEEE/ACM Trans. on Networking*, 5(5):690–704, Oct. 1997.

[9] A. G. Greenberg and N. Madras. How fair is fair queuing. *Journal of the ACM*, 39(3):568–598, July 1992.

[10] S. Iyer and P. Druschel. Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous I/O. In *SOSP'01: 18th ACM Symp. on Operating Systems Principles*, pages 117–130, Banff, Canada, Oct. 2001.

[11] W. Jin, J. S. Chase, and J. Kaur. Interposed proportional sharing for a storage service utility. In *ACM SIGMETRICS*, pages 37–48, New York, NY, June 2004.

[12] J. Kim, Y. Oh, E. Kim, J. Choi, D. Lee, and S. H. Noh. Disk schedulers for solid state drives. In *EMSOFT'09: 7th ACM Conf. on Embedded Software*, pages 295–304, Grenoble, France, Oct. 2009.

[13] J. Kim, S. Seo, D. Jung, J.-S. Kim, and J. Huh. Parameter-aware I/O management for solid state disks (SSDs). *IEEE Trans. on Computers*, Apr. 2011.

[14] R. Konishi, Y. Amagai, K. Sato, H. Hifumi, S. Kihara, and S. Moriai. The Linux implementation of a log-structured file system. *ACM SIGOPS Operating Systems Review*, 40(3):102–107, July 2006.

[15] A. Leventhal. Flash storage memory. *Communications of the ACM*, 51(7):47–51, July 2008.

[16] A. K. Parekh. *A generalized processor sharing approach to flow control in integrated services networks*. PhD thesis, Dept. Elec. Eng. Comput. Sci., MIT, 1992.

[17] S. Park and K. Shen. FIOS: A fair, efficient Flash I/O scheduler. In *FAST'12: 10th USENIX Conf. on File and Storage Technologies*, San Jose, CA, Feb. 2012.

[18] P. J. Shenoy and H. M. Vin. Cello: A disk scheduling framework for next generation operating systems. In *ACM SIGMETRICS*, pages 44–55, Madison, WI, June 1998.

[19] M. Wachs, M. Abd-El-Malek, E. Thereska, and G. R. Ganger. Argon: Performance insulation for shared storage servers. In *FAST'07: 5th USENIX Conf. on File and Storage Technologies*, pages 61–76, San Jose, CA, Feb. 2007.

[20] J. Zhang, A. Sivasubramaniam, Q. Wang, A. Riska, and E. Riedel. Storage performance virtualization via throughput and latency control. *ACM Trans. on Storage*, 2(3):283–308, Aug. 2006.