FIOS: A Fair, Efficient Flash I/O Scheduler*

Stan Park and Kai Shen
Department of Computer Science, University of Rochester
{park, kshen}@cs.rochester.edu

Abstract

Flash-based solid-state drives (SSDs) have the potential to eliminate the I/O bottlenecks in data-intensive applications. However, the large performance discrepancy between Flash reads and writes introduces challenges for fair resource usage. Further, existing fair queueing and quanta-based I/O schedulers poorly manage the I/O anticipation for Flash I/O fairness and efficiency. Some also suppress the I/O parallelism which causes substantial performance degradation on Flash. This paper develops FIOS, a new Flash I/O scheduler that attains fairness and high efficiency at the same time. FIOS employs a fair I/O timeslice management with mechanisms for read preference, parallelism, and fairness-oriented I/O anticipation. Evaluation demonstrates that FIOS achieves substantially better fairness and efficiency compared to the Linux CFQ scheduler, the SFQ(D) fair queueing scheduler, and the Argon quanta-based scheduler on several Flash-based storage devices (including a CompactFlash card in a low-power wimpy node). In particular, FIOS reduces the worst-case slowdown by a factor of 2.3 or more when the read-only SPECweb workload runs together with the write-intensive TPC-C.

1 Introduction

NAND Flash devices [1, 20, 24] are widely used as solid-state storage on conventional machines and low-power wimpy nodes [2, 6]. Compared to mechanical disks, they deliver much higher I/O performance which can alleviate the I/O bottlenecks in critical data-intensive applications. Emerging non-volatile memory (NVRAM) technologies such as phase-change memory [10, 12], memristor, and STT-MRAM promise even better performance. However, these NVMs under today's manufacturing technologies still suffer from low space density (or high $/GB) and stability/durability problems. Until these issues are resolved sometime in the future, NAND Flash devices will likely remain the dominant solid-state storage in computer systems.

*This work was supported in part by the National Science Foundation (NSF) grant CCF-0937571, NSF CAREER Award CCF-0448413, a Google Research Award, and an IBM Faculty Award.

While Flash-based storage devices may offer substantially improved I/O performance over mechanical disks, there are critical limitations with respect to writes. First, Flash suffers from an erase-before-write limitation. That is, in order to overwrite a previously written location, the said location must first be erased before writing the new data. Further aggravating the problem is that the erasure granularity is typically much larger (64–256×) than the basic I/O granularity (2–8 KB). This leads to a large read/write speed discrepancy—Flash reads can be one or two orders of magnitude faster than writes. This is very different from mechanical disks on which read/write performance are both dominated by seek/rotation delays and exhibit similar characteristics.

For a concurrent workload with a mixture of readers and synchronous writers running on Flash, readers may be blocked by writes with substantial slowdown. This means unfair resource utilization between readers and writers. In extreme cases, it may present vulnerability to denial-of-service attacks—a malicious user may invoke a workload with a continuous stream of writes to block readers. At the opposite end, strictly prioritizing reads over writes might lead to unfair (and sometimes extreme) slowdown for applications performing synchronous writes. Synchronous writes are essential for applications that demand high data consistency and durability, including databases, data-intensive network services [28], persistent key-value store [2], and periodic state checkpointing [19].

With important implications on performance and reliability, Flash I/O fairness warrants first-class attention in operating system I/O scheduling. Conventional scheduling methods to achieve fairness (like fair queueing [5, 18] and quanta-based scheduling [3, 36]) fail to recognize unique Flash characteristics like substantial read-blocked-by-write. In addition, I/O anticipation (temporarily idling the device in anticipation of a soon-arriving desirable request) is sometimes necessary to maintain fair resource utilization. While I/O anticipation was proposed as a performance-enhancing seek-reduction technique for mechanical disks [17], its role for maintaining fairness has been largely ignored. Finally, quanta-based scheduling schemes [3, 36] typically suppress the I/O parallelism between concurrent tasks,

which substantially degrades the I/O efficiency on Flash devices with internal parallelism.

This paper presents a new operating system I/O scheduler (called *FIOS*) that achieves fair Flash I/O while attaining high efficiency at the same time. Our scheduler uses timeslice management to achieve fair resource utilization under high I/O load. We employ read preference to minimize read-blocked-by-write in concurrent workloads. We exploit device-level parallelism by issuing multiple I/O requests simultaneously when fairness is not violated. Finally, we manage I/O anticipation judiciously such that we achieve fairness with limited cost of device idling.

We implemented our scheduler in Linux and demonstrated our results on multiple Flash devices including three solid-state disks and a CompactFlash card in a low-power wimpy node. Our evaluation employs several application workloads including the SPECweb workload on an Apache web server, TPC-C workload on a MySQL database, and the FAWN Data Store developed specifically for low-power wimpy nodes [2]. Our empirical work also uncovered a flaw in the current Linux's inconsistent management of synchronous writes across file system and I/O scheduler layers.

The rest of this paper is organized as follows. Section 2 discusses related work. Section 3 characterizes key challenges for supporting Flash I/O fairness and efficiency that motivate our work. Section 4 presents the design of our FIOS scheduler for Flash storage devices. Section 5 describes some implementation notes and Section 6 illustrates our experimental evaluation. Section 7 concludes this paper with a summary of our findings.

2 Related Work

There are significant recent research interests in I/O performance characterization of Flash-based storage devices. Agrawal *et al.* [1] discussed the impact of block erasure (before writes) and parallelism to the performance of Flash-based SSDs. Polte *et al.* [31] found that Flash reads are substantially faster than writes. Past studies identified abnormal performance issues due to read/write interference and storage fragmentation [7], as well as erasure-induced variance of Flash write latency [9]. There is also a recognition on the importance of internal parallelism to the Flash I/O efficiency [8, 30] while our past work identified that the effects of parallelism depend on specific firmware implementations [30]. Previous Flash I/O characterization results provide motivation and foundation for Flash I/O scheduling work in this paper.

Recent research has investigated operating system techniques to manage Flash-based storage. File system work [11, 23, 25] has 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 [21], write block preferential [14], and page-aligned request merging/splitting [22] 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, previous Flash I/O schedulers have paid little attention to the issue of fairness.

Conventional I/O schedulers are largely designed to mitigate the high seek and rotational costs in mechanical disks, through elevator-style I/O request ordering and anticipatory I/O [17]. Quality-of-service objectives (like meeting task deadlines) were also considered in I/O scheduling techniques, including Facade [27], Reddy *et al.* [33], pClock [16], and Fahrrad [32]. Fairness was not a primary concern in these techniques and they cannot address the fairness problems in Flash storage devices.

Fairness-oriented resource scheduling has been extensively studied in the literature. The original fair queueing approaches including Weighted Fair Queueing (WFQ) [13], Packet-by-Packet Generalized Processor Sharing (PGPS) [29], and Start-time Fair Queueing (SFQ) [15] 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 YFQ [5] and SFQ(D) [18] are adapted to support I/O resources. In particular, they allow the flexibility to re-order and parallelize I/O requests for better efficiency. Alternatively, I/O fair queueing can be achieved using dedicated per-task quanta (as in Linux CFQ [3] and Argon [36]) and credits (as in the SARC rate controller [37]). Achieving fairness and efficiency on Flash storage, however, must address unique Flash I/O characteristics like read/write performance asymmetry and internal parallelism. A proper management of I/O anticipation for fairness is also necessary.

3 Challenges and Motivation

We characterize key challenges for supporting Flash I/O fairness and maintaining high efficiency at the same time. They include effects of inherent device characteristics (read/write asymmetry and internal parallelism) as well as behavior of operating system I/O schedulers (role of I/O anticipation). These results and analysis serve as both background and motivation for our new I/O scheduling design.

Experiments in this section and the rest of the paper will utilize the following Flash-based storage devices—

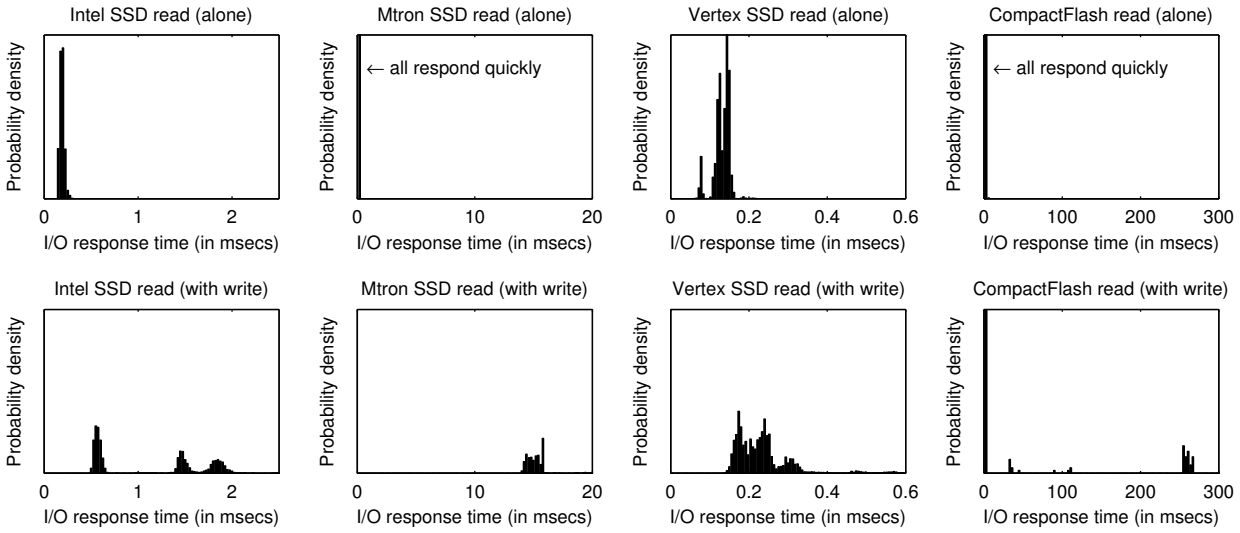

Figure 1: Distribution of 4 KB read response time on four Flash-based storage devices. The first row shows the read response time when a read runs alone. The second row shows the read performance at the presence of a concurrent 4 KB write. The two figures in each column (for one drive) use the same X-Y scale and they can be directly compared. Figures across different columns (for different drives) necessarily use different X-Y scales due to differing drive characteristics. We intentionally do not show the quantitative Y values (probability densities) in the figures because these values have no inherent meaning and they simply depend on the width of each bin in the distribution histogram.

- An Intel X25-M Flash-based SSD released in 2009. This drive uses multi-level cells (MLC) in which a particular cell is capable of storing multiple bits of information.
- An Mtron Pro 7500 Flash-based SSD, released in 2008, using single-level cells (SLC).
- An OCZ Vertex 3 Flash-based SSD, released in 2011, using MLC. This drive employs the SandForce controller which supports new write acceleration techniques such as online compression.
- A SanDisk CompactFlash drive on a 6-Watts "wimpy" node similar to those employed in the FAWN array [2].

Read/Write Fairness Our first challenge to Flash I/O fairness is that Flash writes are often substantially slower than reads and a reader may experience excessive slowdown at the presence of current writes. We try to understand this by measuring the read/write characteristics of the four Flash devices described above. To acquire the native device properties, we bypass the memory buffer, operating system I/O scheduler, and the device write cache in the measurements. We also use incompressible data in the I/O measurement to assess the baseline performance for the Vertex drive (whose SandForce controller performs online compression).

Our measurement employs 4 KB reads or writes to random storage locations. Figure 1 illustrates the read response time distribution in two cases—read alone and read at the presence of a concurrent write. Comparing that with the read-alone performance (first row), we find that a Flash read can experience one or two orders of magnitude slowdown while being blocked by a concurrent write. Further, the Flash read response time becomes much less stable (or more unpredictable) when blocked by a concurrent write. One exception to this finding is the Vertex drive with the SandForce controller. Writes on this drive is only modestly slower than reads and therefore the read-block-by-write effect is much less pronounced on this drive than on others.

We further examine the fairness between two tasks—a *reader* that continuously performs 4 KB reads to random locations (issues another one immediately after the previous one completes) and a *writer* that continuously performs synchronous 4 KB writes to random locations. Figure 2 shows the slowdown ratios for reads and writes during a concurrent execution. Results show that the write slowdown ratios are close to one on all Flash storage devices, indicating that the write performance in the concurrent execution is similar to the write-alone performance. However, reads experience 7×, 157×, 2×, and 42× slowdown on the Intel SSD, Mtron SSD, Vertex SSD, and the low-power CompactFlash respectively.

Existing fairness-oriented I/O schedulers [3, 5, 18, 36, 37] do not recognize the Flash read/write performance asymmetry. Consequently they provided no support to

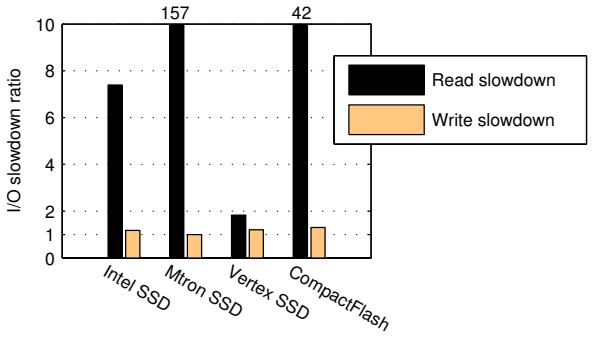

Figure 2: Slowdown of random 4 KB reads and writes in a concurrent execution. The *I/O slowdown ratio* for read (or write) is the I/O latency normalized to that when running alone.

address the problem of excessive read-blocked-by-write on Flash.

Role of I/O Anticipation I/O anticipation (temporarily idling the device in anticipation of a soon-arriving desirable request) was proposed as a performance-enhancing seek-reduction technique for mechanical disks [17]. However, its performance effects on Flash are largely negative because the cost of device idling far outweighs limited benefit of I/O spatial proximity. Due to the lack of performance gain on Flash, the Linux CFQ scheduler disables I/O anticipation for non-rotating storage devices like Flash. Fair queueing approaches like YFQ [5] and SFQ(D) [18] also provide no support for I/O anticipation.

However, I/O anticipation is sometimes necessary to maintain fair resource utilization. Without anticipation, unfairness may arise due to the prematurely switching task queues before the allotted I/O quantum is fully utilized (in quanta-based scheduling) or the premature advance of virtual time for "inactive tasks" (in fair queueing schedulers). Consider the simple example of a concurrent run involving a reader and a writer. After servicing a read, the only queued request at the moment is a write and therefore a work-conserving I/O scheduler will issue it. This breaks up the allotted quantum for the reader. Even if the reader issues another read after a short thinktime, it would be blocked by the outstanding write.

At the opposite end, the quanta-based scheduling in Argon [36] employs aggressive I/O anticipation such that it is willing to wait through a task queue's full quantum even if few requests are issued. Such excessive I/O anticipation can lead to long idle time and drastically reduce performance on Flash storage if useful work could otherwise have been accomplished. Particularly for fast Flash storage, a few milliseconds are often sufficient for completing a significant amount of work.

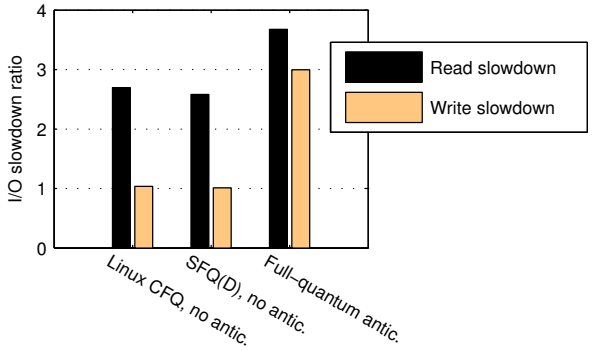

Figure 3: Fairness of different I/O anticipation approaches for concurrent reader/writer on the Intel SSD.

We run a simple experiment to demonstrate the fairness and efficiency effects of improper I/O anticipation on Flash. We run a reader and a writer concurrently on the Intel SSD. Each task induces some thinktime between I/O such that the thinktime time is approximately equal to its I/O device usage time. Figure 3 shows the reader/writer slowdown under three I/O scheduling approaches. Implementation details of the schedulers are provided later in Section 5. The Linux CFQ and SFQ(D) do not support I/O anticipation which leads to poor fairness between the reader and writer. The full-quantum anticipation exhibits better fairness (similar reader/writer slowdown) but this is achieved at excessive slowdown for both reader and writer. Such fairness is not worthwhile.

While our discussion above uses the example of a reader running concurrently with a writer, the fairness implication of I/O anticipation generally applies to concurrent tasks with requests of differing resource usage. For instance, similar fairness problems with no I/O anticipation or over-aggressive anticipation can arise when a task making 4 KB reads runs concurrently with a task making 128 KB reads.

Parallelism vs. Fairness Flash-based SSDs have some built-in parallelism through the use of multiple channels. Within each channel, each Flash package may have multiple planes which are also parallel. Figure 4 shows the efficiency of Flash I/O parallelism for 4 KB reads and writes on our Intel, Mtron, and Vertex SSDs. We observe that the parallel issuance of multiple reads to an SSD may lead to throughput enhancement. The speedup is modest (about 30%) for the Mtron SLC drive but substantial (up to 7-fold and 4-fold) for the Intel and Vertex MLC drives. On the other hand, writes do not seem to benefit from I/O parallelism on the Intel and Mtron drives while write parallelism on the Vertex drive can have up to 3-fold speedup. We also experimented with parallel I/O at larger (>4 KB) sizes and we found that the speedup of

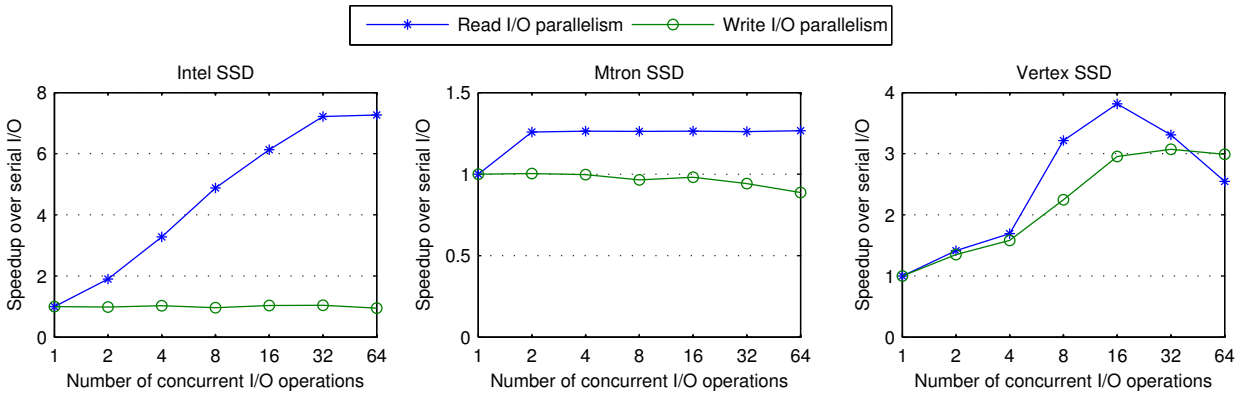

Figure 4: Efficiency of I/O parallelism for 4 KB reads and writes on three Flash-based SSDs.

parallel request issuance is less substantial for large I/O requests. A possible explanation is that a single large I/O request may already benefit from the internal device parallelism and therefore parallel request issuance will see less additional efficiency gain.

The internal parallelism on Flash-based SSDs has significant implication on fairness-oriented I/O scheduling. In particular, the quanta-based schedulers (like Linux CFQ [3] and Argon [36]) only issue I/O requests from one task queue at a time, which limits parallelism. The rationale is probably to ease the accounting and allocation of device time usage for each queue. However, the suppression of I/O parallelism in these schedulers may lead to substantial performance degradation on Flash. A desired Flash I/O scheduler must exploit device-level parallelism by issuing multiple I/O requests simultaneously while ensuring fairness at the same time.

4 FIOS Design

In a multiprocessing system, many resource principals simultaneously compete for the 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. When the storage device is partially loaded, the critical problem is that a read blocked by a write experiences far worse slowdown than a write blocked by a read. Such worst-case slowdown should be minimized.

Practical systems may desire fairness for different kinds of resource principals. For example, a general-purpose operating system may desire fairness support among concurrent processes. A server system may need fairness across simultaneously running requests [4, 34]. A shared hosting platform may want fairness across multiple virtual machines [26]. Our design of fair Flash I/O scheduling and much of our implementation can be generally applied to supporting arbitrary resource principals. When describing the FIOS design, we use the term *task* to represent the resource principal that receives the fairness support in a concurrent execution.

Our I/O scheduler, FIOS, tries to achieve fairness while attaining high efficiency at the same time. Based on our evaluation and analysis in Section 3, our scheduler contains four techniques. We first provide a fair timeslice management that allows timeslice fragmentation and concurrent request issuance (Section 4.1). We then support read preference to minimize the read-blocked-by-write situations (Section 4.2). We further enable concurrent issuance of requests to maximize the efficiency of device-level parallelism (Section 4.3). Finally, we devise limited I/O anticipation to maintain fairness at minimal device idling cost (Section 4.4).

4.1 Fair Timeslice Management

FIOS builds around a fairness mechanism of equal timeslices which govern the amount of time a task has access to the storage device. As each task is given equal time-based access to the storage device, the disparity between read and write access latency of Flash cannot lead to unequal device usage between tasks. In addition, using timeslices provides an upper bound on how long a task may have access to the storage device, ensuring that no task will be starved indefinitely.

Our I/O timeslices are reminiscent of the I/O quanta in quanta-based fairness schedulers like Linux CFQ [3] and Argon [36]. However, the previous quanta-based schedulers suffer two important limitations that make them unsuitable for Flash fairness and efficiency.

- First, their I/O quanta do not allow fragmentation—a task must use its current quantum continuously or it will have to wait for its next quantum in the round-robin order. The rationale (on mechanical disk storage devices) was that long continuous run by a

single task tends to require less disk seek and rotation [36]. But for a task that performs I/O with substantial inter-I/O thinktime, this design leaves two undesirable choices—either its quantum ends prematurely so the remaining allotted resource is forfeited (as in Linux CFQ) or the device idles through a task's full quantum even if few requests are issued (as in Argon).

- Second, the previous quanta-based schedulers only allow I/O requests from one task to be serviced at a time. This was a reasonable design decision for individual mechanical disks that do not possess internal parallelism. It also has the advantage of easy resource accounting for each task. However, this mechanism suppresses Flash I/O parallelism and consequently hurts I/O efficiency.

To address these problems, FIOS allows I/O timeslice fragmentation and concurrent request issuance from multiple tasks. Specifically, we manage timeslices in an epoch-based fashion. An epoch is defined by a collection of equal timeslices, one per task; the I/O scheduler should achieve fairness in each epoch. After an I/O completion, the task's remaining timeslice is decremented by an appropriate I/O cost. The cost is the elapsed time from the I/O issuance to its completion when the storage device is dedicated to this request in this duration. The cost accounting is more complicated in the presence of parallel I/O from multiple tasks, which will be elaborated in Section 4.3. A currently active task does not forfeit its remaining timeslice should another task be selected for service by the scheduler. In other words, the timeslice of a task can be consumed over several non-contiguous periods within an epoch. Once a task has consumed its entire timeslice, it must wait until the next epoch at which point its timeslice is refreshed.

The current epoch ends and a new epoch begins when either 1) there is no task with non-zero remaining timeslice in the current epoch; or 2) all tasks with non-zero remaining timeslices make no I/O request. Fairness must be maintained in the case of deceptive idleness [17]. Specifically, the I/O scheduler may observe a short idle period from a task between two consecutive I/O requests it makes. A fair-timeslice epoch should not end at such deceptive idleness if the task has non-zero remaining timeslice. This is addressed through fairness-oriented I/O anticipation elaborated in Section 4.4.

4.2 Read/Write Interference Management

Our preliminary evaluation in Section 3 shows strong interference between concurrent reads and writes on some of the Flash drives, an effect also observed by others [7]. Considering that reads are faster than writes, reads suffer more dramatically from such interference while the impact on writes appears marginal. A concurrent write not only slows down reads, it also disrupts device-level read parallelism which leads to further efficiency loss. Part of our fairness goal is to minimize the worst-case task slowdown. For such fairness, we adopt a policy of read preference combined with write blocking to reduce the read-interfered-by-write occurrences. Such a policy gives preference to shorter jobs, which tends to produce faster mean response time than a scheduler that is indiscriminate of job service time. This is a side benefit beyond minimizing the worst-case slowdown.

When both read and write requests are queued in the I/O scheduler, our policy of read preference will allow read requests to be issued first. To further avoid interference from later-issued writes, we block all write requests until outstanding reads are completed. Under this approach, a read is only blocked by a write when the read arrives at the I/O scheduler after the write has already been issued. This is due to the non-preemptibility of I/O. Both read preference and write blocking lead to additional queuing time for writes. Fortunately, because reads are serviced quickly, the additional queueing time the write request experiences is typically small compared to the write service time. Note that the read preference mechanism is still governed by the epoch-based timeslice enforcement, which serves as an ultimate preventer of write starvation.

Our preliminary evaluation in Section 3 also shows that while the read/write interference is very strong on some drives, it is quite modest on the Vertex SSD. On such a drive, the benefit of read preference and write blocking is modest and it may be outweighed by its drawbacks of possible write starvation and suppressing the mixed read/write parallelism. Therefore the read/write interference management is an optional feature in FIOS that can be disabled for drives that do not exhibit strong read/write interference.

4.3 I/O Parallelism

Many Flash-based solid-state drives contain internal parallelism that allows multiple I/O requests to be serviced at the same time. To achieve high efficiency and exploit the parallel architecture in Flash, multiple I/O requests should be issued to the Flash device in parallel when fairness is not violated. After issuing an I/O request to the storage device, FIOS searches for additional requests which may be queued, possibly from other tasks. Any I/O requests that are found are issued as long as the owner tasks have enough remaining timeslices and the read/write interference management (if enabled) is observed.

I/O parallelism allows multiple tasks to access the storage device concurrently, which complicates the account-

ing of I/O cost. In particular, a task should not be billed by the full elapsed time from its request issuance to completion if requests from other tasks are simultaneously outstanding on the storage device. The ideal cost accounting for an I/O request should exclude the request queueing time at the device during which it waits for other requests and it does not consume the bottleneck resource. A precise accounting, however, is difficult without the device-level knowledge of resource sharing between multiple outstanding requests.

We support two approaches for I/O cost accounting under parallelism. In the first approach, we calibrate the elapsed time of standalone read/write requests at different data sizes and use the calibration results to assign the cost of an I/O request online depending on its type (read or write) and size. Our implementation further assumes a linear model (typically with a substantial nonzero offset) between the cost and data size of an I/O request. Therefore we only need to calibrate four cases (read 4 KB, read 128 KB, write 4 KB, and write 128 KB) and use the linear model to estimate read/write costs at other data sizes. In practice, such calibration is performed once for each device, possibly at the device installation time. Note that the need of request cost estimation is not unique to our scheduler. Start-time Fair Queueing schedulers [15, 18] also require a cost estimation for each request when it just arrives (for setting its start and finish tags).

When the calibrated I/O costs are not available, our scheduler employs a backup approach for I/O cost accounting. Here we make the following assumption about the sharing of cost for parallel I/O. During a time period when the set of outstanding I/O requests on the storage device remains unchanged (no issuance of a new request or completion of an outstanding request), all outstanding I/O requests equally share the device usage cost in this time period. This is probabilistically true when the internal device scheduling and operation is independent of the task owning the request. Such an assumption allows us to account for the cost of parallel I/O with only information available to the operating system. Since the device parallelism may change during a request's execution, an accurate accounting of a request's execution parallelism would require carefully tracking the device parallelism throughout its execution duration. For simplicity, we use the device parallelism at the time of request issuance to represent the request execution parallelism. Specifically, the I/O cost is calculated as

$$\text{Cost} = \frac{T_{\text{elapsed}}}{P_{\text{issuance}}} \quad (1)$$

where T_{elapsed} is the request's elapsed time from its issuance to its completion, and P_{issuance} is the number of outstanding requests (including the new request) at the issuance time.

4.4 I/O Anticipation for Fairness

Between two consecutive I/O requests made by a task, the I/O scheduler may observe a short idle period. This idle period is unavoidable because it takes non-zero time for the task to wake up and issue another request. Such an idleness is deceptive for tasks that continuously make synchronous I/O requests. The deceptive idleness can be addressed by I/O anticipation [17], which idles the storage device in anticipation of a soon-arriving new I/O request. On mechanical disks, I/O anticipation can substantially improve the I/O efficiency by reducing the seek and rotation overhead. In contrast, I/O spatial proximity has much less benefit for Flash storage. Therefore I/O anticipation has a negative performance effect and it must be used judiciously for the purpose of maintaining fairness. Below we describe two important decisions about fairness-oriented I/O anticipation on Flash—When to anticipate? How long to anticipate?

When to anticipate? Anticipation is always considered when a request is just completed. We call the task that owns the just completed request the *anticipating task*.

Deceptive idleness may break fair timeslice management when it prematurely triggers an epoch switch while the anticipating task will quickly process the just completed I/O request and issue another one soon. I/O anticipation should be utilized to remedy such a fairness violation. Specifically, while an epoch would normally end if there is no outstanding I/O request from a task with non-zero remaining timeslice, we initiate an anticipation before the epoch switch if the anticipating task has non-zero remaining timeslice. In this case the anticipation target can be either a read or write, though it is more commonly write since writers are more likely delayed to the end of an epoch under read preference.

Deceptive idleness may also break read preference. When there are few tasks issuing reads, there may be instances when no read request is queued. In order to facilitate read preference, I/O anticipation is necessary after completing a read request. If a read request has just been completed, we anticipate for another read request to arrive shortly. We do so rather than immediately issuing a write to the device that may block later reads.

How long to anticipate? I/O anticipation duration must be bounded in case the anticipated I/O request never arrives. For maximum applicability and robustness, the system should not assume any application hints or predictor of the application inter-I/O thinktime. For seek-reduction on mechanical disks, the I/O anticipation bound is set to roughly the time of a disk I/O operation which leads to competitive performance compared to the

optimal offline I/O anticipation. In practice, this is often set to 6 or 8 milliseconds. Our I/O anticipation bound must be different for two reasons. First, the original anticipation bound addresses the device idling's tradeoff with performance gain of seek reduction. Anticipation has a negative performance effect on Flash and we instead target the different tradeoff with maintaining fairness. Second, the Flash I/O service time is much smaller than that of a disk I/O operation. This exacerbates the cost of anticipation-induced device idling on Flash.

FIOS sets the I/O anticipation bound according to a configurable threshold of tolerable performance loss for maintaining fairness. This threshold, α, indicates the maximum proportion of time FIOS idles the device (while there is pending work) to anticipate for fairness. Specifically, when the deceptive idleness is about to break fairness, we anticipate for an idling time bound of $T_{\text{service}} \cdot \frac{\alpha}{1-\alpha}$, where T_{service} is the average service time of an I/O request for the anticipating task. This ensures that the maximum device idle time is no more than α proportion of the total device time in a sequence of

$$\text{I/O} \rightarrow \text{anticipation} \rightarrow \text{I/O} \rightarrow \text{anticipation} \rightarrow \cdots$$

In our implementation, FIOS maintains the per-request I/O service time T_{service} for each task using an exponentially-weighted moving average of past request statistics. FIOS sets $\alpha = 0.5$ by default.

Anticipation-induced device idling consumes device time and its cost must be properly accounted and attributed. We charge the anticipation cost to the timeslice of the anticipating task.

5 Implementation Notes

We implemented our FIOS scheduler with the techniques of fair timeslice management, read preference, I/O parallelism, and I/O anticipation for fairness on Linux 2.6.33.4. As part of a general-purpose operating system, our prototype provides fairness to concurrent processes. This implementation can be easily extended to support request-level fairness in a server system [4,34] or virtual machine fairness in a shared hosting platform [26].

Our I/O anticipation may sometimes desire a very short timer (a few hundred microseconds). The default Linux I/O schedulers use the kernel tick-based timer. Specifically with 1000 Hz kernel ticks, the minimum timer is 1 millisecond. Further, because the kernel ticks are not synchronized with the timer setup, the next tick may occur right after the timer is set. This means that setting the timer to fire at the next tick may sometimes lead to almost no anticipation. Our recent research [35] showed that this already happened to some production versions of Linux with coarse-grained tick timers. Our FIOS implementation instead uses the Linux high-resolution timer that can be supported by the processor hardware counter overflow interrupts. This allows us to set precise, fine-grained anticipation timers.

For comparison purposes, we implemented two alternative fairness-oriented I/O schedulers in our experimental platform. The first alternative is SFQ(D) [18], which is based on the Start-time Fair Queueing approach [15] but also allows concurrent request issuance for I/O efficiency. The concurrency is controlled by a depth parameter *D*. We set the depth to 32 which allows sufficient I/O parallelism in all our experiments. The SFQ(D) scheduler requires a cost estimation for each request when it just arrives (for setting its start and finish tags in SFQ(D)). In our implementation, we estimate a read's cost as the average read service time on the device; similarly, we estimate the cost of a write as the average write service time on the device.

The second alternative is a quanta-based I/O scheduler like the one employed in Argon [36]. This approach 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 Linux CFQ, our FIOS scheduler, and the quanta scheduler all use the concept of per-task timeslice or quantum. In the Linux CFQ, the default timeslice is 100 milliseconds, with minor adjustment according to task priorities. Our FIOS and quanta scheduler implementations follow the same setting of per-task timeslice/quantum.

During our empirical work, we discovered a flaw in Linux that it inconsistently manages synchronous writes across the file system and I/O scheduler layers. Specifically, a synchronous operation at the file system level (such as a write on an `O_SYNC`-opened file and I/O as part of a `fsync()` call) is not necessarily considered to be synchronous at the I/O scheduler. Note that this inconsistency does not lead to wrong synchronous I/O semantics to the application since the file system will force a wait on the I/O completion before returning to the application. However, being treated as asynchronous I/O at the I/O scheduler means that they are scheduled with lowest priority, leading to excessive delay by the applications who perform synchronous I/O. We fixed this problem by patching `mpage_writepage()` functions in the Linux kernel so that file system-level synchronous operations are properly considered synchronous I/O at the scheduler.

We perform experiments on the ext4 file system. The 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.

We also found that the file system journaling writes made the evaluation results less stable and harder to interpret. Therefore we disabled the journaling in our experiments. We do not believe this setup choice affects the fundamental results of our evaluation.

6 Experimental Evaluation

We compare FIOS's fairness and efficiency against three alternative fairness-oriented I/O schedulers—Linux CFQ scheduler [3], SFQ(D) start-time fair queueing with a concurrency depth [18], and a quanta-based I/O scheduler similar to the one employed in Argon [36]. Implementation details for some of these schedulers were provided in the previous section. We also compare against the raw device I/O in which requests are issued to the storage devices as soon as they are passed from the file system.

We explain our fairness and efficiency metrics in evaluation. Fairness is defined as the case that each task gains equal access to resources. In a concurrent execution with n tasks, this can be observed if each task experiences a factor of n slowdown compared to running-alone. We call this *proportional slowdown*. Note that better performance may be achieved when some tasks only contain partial I/O load (*i.e.*, they do not make I/O requests for significant parts of their execution). Some tasks may also gain better performance if they are able to utilize the allotted resources more efficiently (*e.g.*, through exploiting device internal parallelism). However, fairness dictates that none should exhibit substantially worse performance than the proportional slowdown.

We also devise a metric to represent the overall system efficiency of a concurrent execution. This metric, we call *concurrent efficiency*, measures the relative throughput of the concurrent execution to the running-alone throughput of individual tasks. Intuitively, it assigns a base efficiency of 1.0 to each task's running-alone performance (at the absence of resource competition and interference) and then weighs the throughput of a concurrent execution against the base efficiency. Consider n concurrent tasks t_1, t_2, $\cdots$, t_n. Let t_i's running-alone throughput be $\text{Thrput}_i^{\text{alone}}$. Let t_i's throughput in the concurrent execution be $\text{Thrput}_i^{\text{conc}}$. Then formally for the concurrent execution:

$$\text{Concurrent efficiency} = \sum_{i=1}^{n} \frac{\text{Thrput}_i^{\text{conc}}}{\text{Thrput}_i^{\text{alone}}}. \qquad (2)$$

An efficiency of less than 1.0 indicates the overhead of concurrent execution or the lack of full utilization of resources. An efficiency of greater than 1.0 indicates the additional benefit of concurrent execution, *e.g.*, due to exploiting the parallelism in the storage device.

Our experiments utilize the Flash-based storage devices described in the beginning of Section 3. They include three (Intel/Mtron/Vertex) Flash-based SSDs as well as a low-power SanDisk CompactFlash drive.

Section 6.1 will first evaluate the fairness and efficiency using a set of synthetic benchmarks with varying I/O concurrency. Section 6.2 then provides evaluation with realistic applications of the SPECweb workload on an Apache web server and the TPC-C workload on a MySQL database. Finally, Section 6.3 performs evaluation on a CompactFlash drive in a low-power wimpy node using the FAWN Data Store workload [2].

6.1 Evaluation with Synthetic I/O Benchmarks

Synthetic I/O benchmarks allow us to flexibly vary parameters in the resource competition. Each synthetic benchmark contains a number of tasks issuing I/O requests of different types and sizes. Evaluation here considers four benchmark cases:

- *1-reader 1-writer* that concurrently runs a reader continuously issuing 4 KB reads and a writer continuously issuing 4 KB writes;
- *4-reader 4-writer* that concurrently runs four 4 KB readers and four 4 KB writers;
- *4-reader 4-writer (with thinktime)* that is like the above case but each task also induces some exponentially distributed thinktime between I/O such that the total thinktime time is approximately equal to its I/O device usage time;
- *4 KB-reader and 128 KB-reader* that concurrently runs a reader continuously issuing 4 KB reads and another reader continuously issuing 128 KB reads.

The last case helps evaluate the value of FIOS for read-only workloads or workloads in which writes are asynchronous and delayed to the background.

Fairness Figure 5 illustrates the fairness and performance of the three read/write benchmark cases under different I/O schedulers. On the two drives (Intel/Mtron SSDs) with strong read/write interference, the raw device I/O, Linux CFQ, and SFQ(D) fail to achieve fairness.

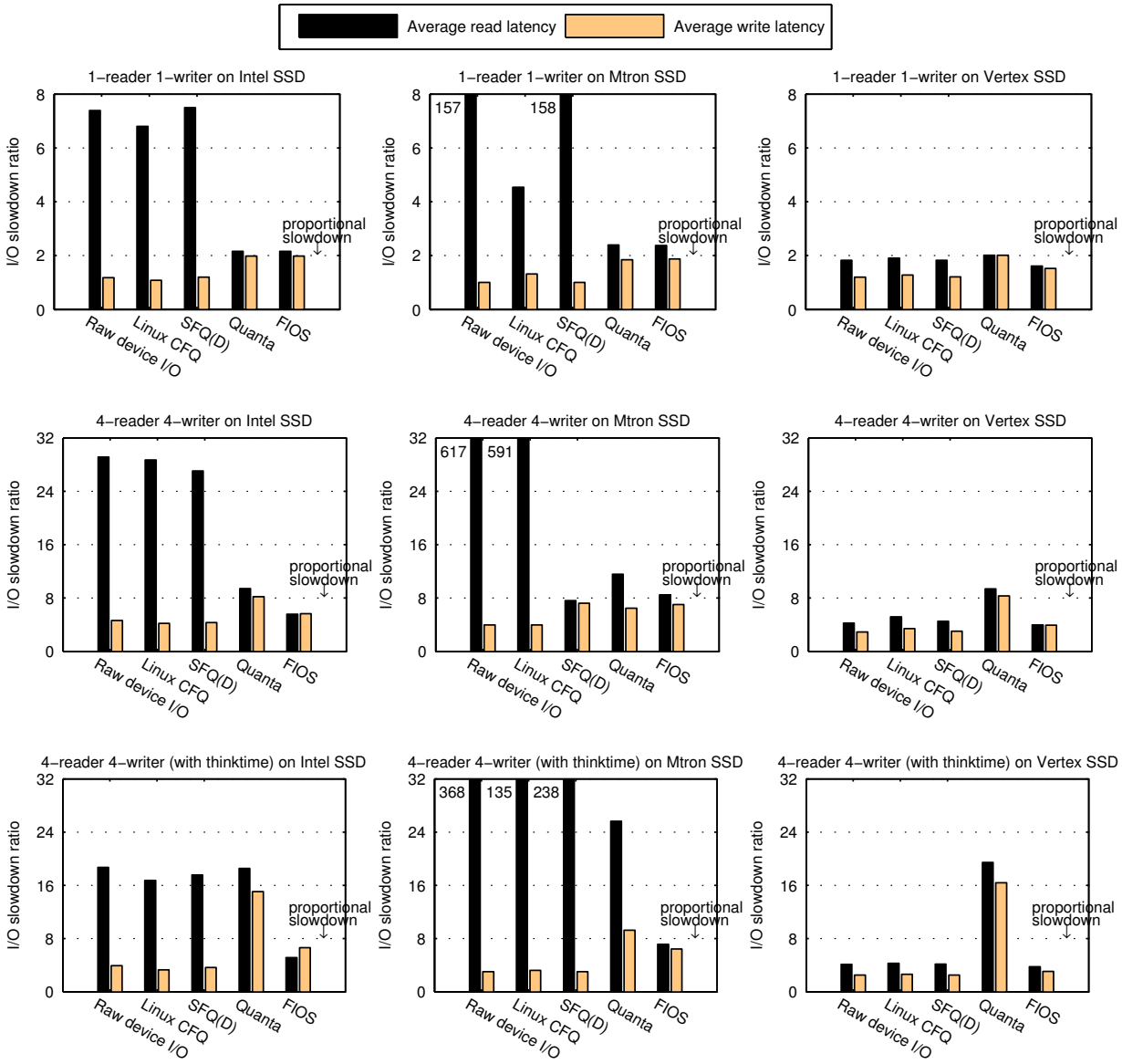

Figure 5: Fairness and performance of synthetic read/write benchmarks under different I/O schedulers. The *I/O slowdown ratio* for read (or write) is the I/O latency normalized to that when running alone. Results cover three Flash-based SSDs (corresponding to the three columns) and three workload scenarios with varying reader/writer concurrency (corresponding to the three rows). For each case, we mark the slowdown ratio that is proportional to the total number of tasks in the system, which is a measure of fairness.

Specifically, readers experience many times the proportional slowdown while writers are virtually unaffected. Because raw device I/O makes no attempt to schedule I/O, reads and writes are interleaved as they are issued by applications, severely affecting the response of read requests. The Linux CFQ does not perform much better because it disables I/O anticipation for non-rotating storage devices like Flash and it suppresses I/O parallelism between concurrent tasks. SFQ(D) also suffers from poor fairness due to its lack of I/O anticipation. For instance, without anticipation, two-task executions degenerate to one-read/one-write interleaved I/O issuance and poor fairness. The quanta scheduler achieves better fairness than other alternatives due to its aggressive maintenance of per-task quantum. However, it suffers from the cost of excessive I/O anticipation and suppression of I/O parallelism. In contrast, FIOS maintains fairness (approximately at or below proportional slowdown) in all the evaluation cases due to our proposed techniques.

On the Vertex SSD, most schedulers achieve good fair-

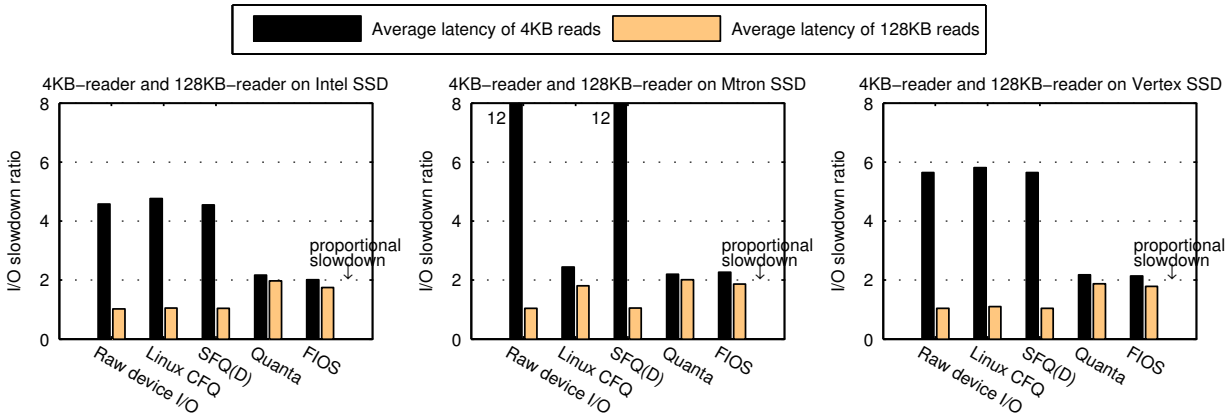

Figure 6: Fairness and performance of two-reader (at different read sizes) benchmark under different I/O schedulers.

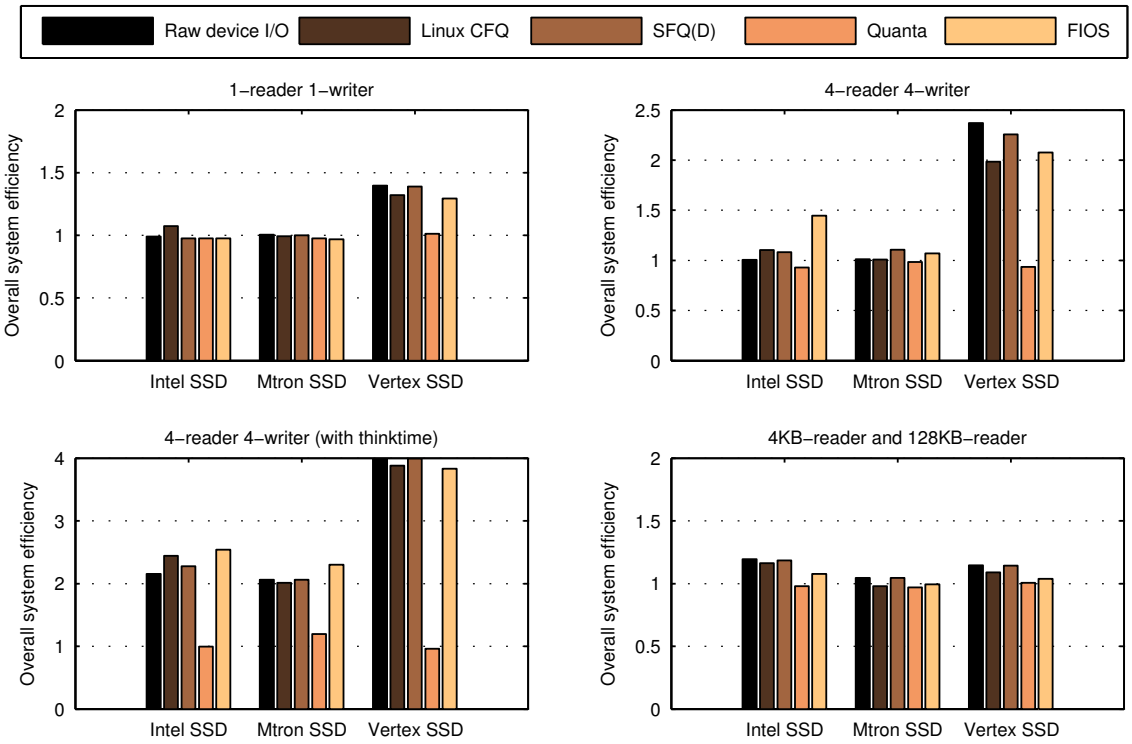

Figure 7: Overall system efficiency of synthetic I/O benchmarks under different I/O schedulers. We use the metric of concurrent efficiency defined in Equation 2. Results cover four benchmark cases and three SSDs.

ness for the read/write benchmark cases due to its modest read/write interference. However, the quanta scheduler still exhibits high cost of excessive I/O anticipation.

Figure 6 shows the fairness and performance of the 4 KB-reader and 128 KB-reader benchmark under different I/O schedulers. Results show that only FIOS and quanta schedulers can maintain fairness in this case. The benefit manifests on all three drives including the Vertex SSD.

Efficiency We next evaluate the overall system efficiency. Figure 7 illustrates the concurrent efficiency (defined in Equation 2) under different I/O schedulers. Results show FIOS achieves higher efficiency when devices allow substantial internal parallelism. These particularly include the two cases with four readers on the Intel and Vertex SSDs. The quanta scheduler exhibits the worst efficiency. This is because its aggressive fairness measures lead to substantial efficiency loss.

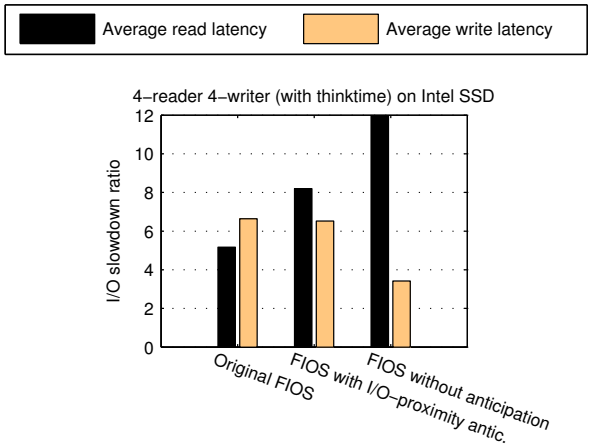

Figure 8: Evaluation on the effect of fairness-oriented I/O anticipation in FIOS on the Intel SSD.

I/O Anticipation for Fairness Figure 8 individually evaluates the effect of fairness-oriented I/O anticipation in FIOS. We compare with two alternatives—no anticipation and anticipation for I/O proximity (as designed in [17] and implemented in Linux). We use the 4-reader 4-writer with thinktime to demonstrate the effect of I/O anticipation. When there is no anticipation, reads suffer substantial additional latency because the deceptive idleness sometimes breaks read preference. While some degree of I/O anticipation is necessary, the conventional I/O anticipation for I/O proximity leads to high performance cost due to excessive idling. The I/O anticipation in FIOS achieves fairness at modest performance cost.

Summary of Results FIOS exhibits better fairness than all alternative schedulers. In terms of efficiency, it is competitive with the best of alternative schedulers in all cases. It is particularly efficient on the Intel SSD because it can exploit its parallelism while managing the read-blocked-by-write problem at the same time.

Among the alternative schedulers, the quanta scheduler is most fair but very inefficient in many cases due to the high cost of its aggressive I/O anticipation. The raw device I/O is most efficient but it is unfair in many situations, particularly in penalizing the reads.

FIOS is not only effective for maintaining fairness between reads and synchronous writes, it is also beneficial for regulating read tasks with different I/O costs. This demonstrates the value of FIOS to support read-only workloads and workloads in which writes are asynchronous and delayed to the background. Further, this makes FIOS valuable for the Vertex drive even though its read/write performance discrepancy is small.

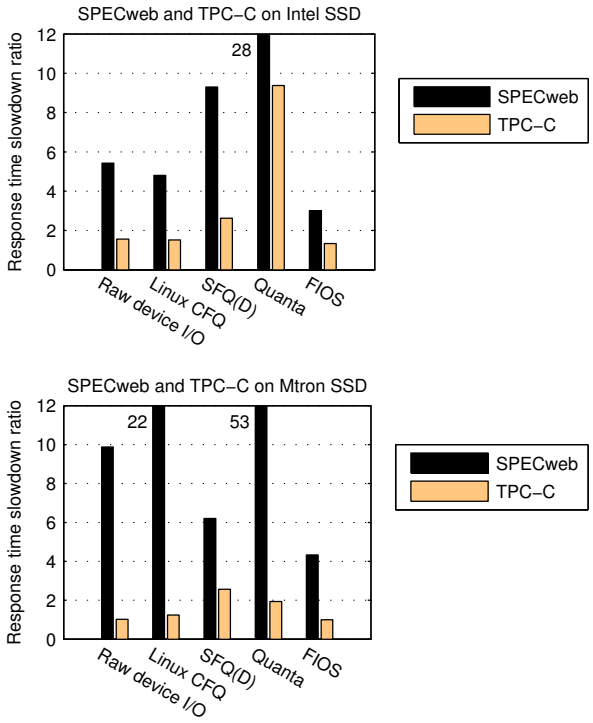

Figure 9: Fairness and performance of SPECweb running with TPC-C under different I/O schedulers. The slowdown ratio for an application is the average request response time normalized to that when the application runs alone. Results cover two Flash-based SSDs.

6.2 Evaluation with SPECweb and TPC-C

Beyond the synthetic benchmarks, we also perform evaluation with realistic workloads. We run the read-only SPECweb99 workload (running on an Apache 2.2.3 web server) along with the write-intensive TPC-C (running on a MySQL 5.5.13 database). Each application is driven by a closed-loop load generator that contains four concurrent clients, each of which issues requests continuously (issuing a new request right after the outstanding one receives a response). The load generators run on a different machine and send requests through the network. This evaluation employs the two drives (Intel/Mtron SSDs) that exhibit large read/write interference effects.

Figure 9 illustrates the fairness and performance results under different I/O schedulers. Unsurprisingly, the read-only SPECweb tends to experience more slowdown than the write-intensive TPC-C does on Flash storage. Among all scheduling approaches, the quanta scheduler exhibits the worst performance and fairness. This is due to its excessive I/O anticipation. Realistic application workloads (like SPECweb and TPC-C) perform significant computation and networking between storage I/O

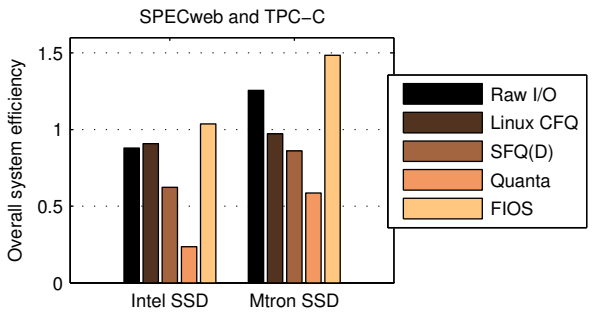

Figure 10: Overall system efficiency of SPECweb running with TPC-C under different I/O schedulers. We use the metric of concurrent efficiency defined in Equation 2. Results cover two Flash-based SSDs.

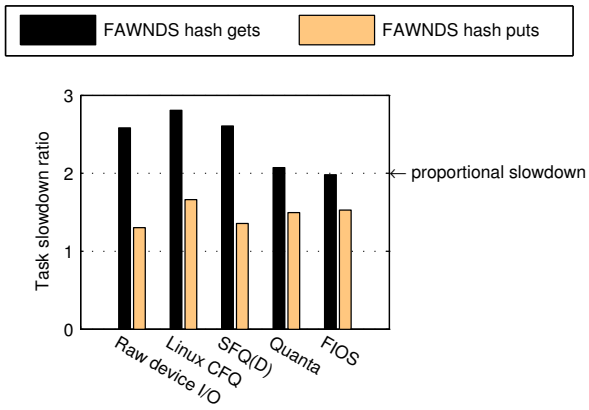

Figure 11: Performance of concurrent FAWN Data Store hash gets (data reads) and hash puts (data writes) on a low-power CompactFlash. The slowdown ratio for a task is defined as its running-alone throughput divided by its throughput at the concurrent run. Higher slowdown ratio means worst performance.

that appears as inter-I/O thinktime. Idling the storage device through such thinktime (as in the quanta scheduler) leads to excessive waste. On the other hand, the poor fairness of the raw device I/O, Linux CFQ, and SFQ(D) is due to a lack of I/O anticipation and poor management of read/write interference on Flash.

FIOS exhibits better fairness and performance than all the alternative approaches, and its performance is more stable across the two SSDs. We measure the fairness as the worst-case application slowdown in a concurrent execution (SPECweb slowdown in all cases). Compared to the quanta scheduler, FIOS reduces the worst-case slowdown by a factor of nine or more on both SSDs. Compared to the raw device I/O, FIOS reduces the worst-case slowdown by a factor of 2.3× on the Mtron SSD. Compared to the Linux CFQ, FIOS reduces the worst-case slowdown by a factor of five on the Mtron SSD. Compared to SFQ(D), FIOS reduces the worst-case slowdown by about 3.1× on the Intel SSD.

Figure 10 shows the overall system efficiency of SPECweb running with TPC-C under different I/O schedulers. Results show that FIOS improves the efficiency above the best alternative scheduler by 14% and 18% on the Intel and Mtron SSDs respectively. FIOS achieves high efficiency due to its proper management of read/write interference, I/O parallelism, and controlled I/O anticipation.

6.3 Evaluation on Low-Power CompactFlash

We also test FIOS on a low-power wimpy node like the ones used in the FAWN work [2]. Specifically, the node contains an Alix board with a single-core 500 MHz AMD Geode CPU, 256 MB SDRAM memory, and a 16 GB SanDisk CompactFlash drive. The full node consumes about 5.9 Watts of power at peak load. The CompactFlash, while also NAND Flash-based, is significantly less sophisticated than solid state drives. CompactFlash cards lack the sophisticated firmware and degree of parallelism available in solid-state drives. Despite these differences, CompactFlash still exhibits some of the intrinsic Flash characteristics that FIOS is designed to consider and exploit.

We requested and acquired the FAWN Data Store application from the authors [2]. In our experiments, we concurrently run two FAWN Data Store tasks, one performing hash gets (data reads) and the other performing hash puts (data writes). We run hash puts synchronously to ensure that the data is made persistent before its result is externalized to client. These tasks run against data stores of 1 million records.

Figure 11 presents the resulting get/put slowdown ratios under different I/O schedulers. Only FIOS keeps both hash gets and puts below the proportional slowdown. The quanta scheduler also exhibits good fairness because its suppression of parallelism has no harmful effect on the CompactFlash which does not allow any I/O parallelism. Further, the quanta scheduler's excessive I/O anticipation causes little efficiency loss for FAWN Data Store that performs batched I/O with almost no inter-I/O thinktime. Under all other approaches (raw device I/O, Linux CFQ, and SFQ(D)), hash gets experience worse performance degradation than the proportional slowdown, which indicates poor fairness.

7 Conclusion

Flash-based storage devices are capable of alleviating I/O bottlenecks in data-intensive applications. However, the unique performance characteristics of Flash storage

must be taken into account in order to fully exploit their superior I/O capabilities while offering fair access to applications. In this paper, we have characterized the performance of several Flash-based storage devices. We observed that during concurrent access, writes can dramatically affect the response time of read requests. We also observed that Flash-based storage exhibits support for some degree of parallel I/O, though the benefit of parallel I/O varies across devices. Further, the lack of seek/rotation overhead eliminates the performance benefit of anticipatory I/O, but proper I/O anticipation is still needed for the purpose of fairness.

Based on these motivations, we designed a new Flash I/O scheduling approach that contains four essential techniques to ensure fairness with high efficiency—fair timeslice management that allows timeslice fragmentation and concurrent request issuance, read/write interference management, I/O parallelism, and I/O anticipation for fairness. We implemented these design principles in a new I/O scheduler for Linux.

We evaluated our I/O scheduler alongside three alternative fairness-oriented I/O schedulers (Linux CFQ [3], SFQ(D) [18], and a quanta-based I/O scheduler similar to that in Argon [36]). Our evaluation uses a variety of synthetic benchmarks and realistic application workloads on several Flash-based storage devices (including a CompactFlash card in a low-power wimpy node). The results expose the shortcomings of existing I/O schedulers while validating our design principles for Flash resource management. In conclusion, this paper makes the case that fairness warrants the first-class concern in Flash I/O scheduling and it is possible to achieve fairness while attaining high efficiency.

While FIOS is primarily motivated by the Flash read/write interference, we also demonstrate that FIOS is beneficial for regulating the resource usage fairness between read tasks with different I/O costs (a task performing small reads runs concurrently with a task performing large reads). This illustrates the value of FIOS to support read-only workloads and workloads in which writes are asynchronous and delayed to the background. Further, FIOS is also valuable for Flash drives that have modest read/write performance discrepancy.

Acknowledgments We thank Amal Fahad (University of Rochester) for setting up the low-power wimpy node for our evaluation. We also thank Zhuan Chen (University of Rochester) for device installation and machine maintenance when both authors were away from the school. Finally, we thank the anonymous FAST reviewers and our shepherd Randal Burns 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] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. FAWN: A fast array of wimpy nodes. In *SOSP'09: 22th ACM Symp. on Operating Systems Principles*, pages 1–14, Big Sky, MT, Oct. 2009.

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

[4] G. Banga, P. Druschel, and J. Mogul. Resource containers: A new facility for resource management in server systems. In *OSDI'99: Third USENIX Symp. on Operating Systems Design and Implementation*, pages 45–58, New Orleans, LA, Feb. 1999.

[5] 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.

[6] A. M. Caulfield, L. M. Grupp, and S. Swanson. Gordon: Using flash memory to build fast, power-efficient clusters for data-intensive applications. In *ASPLOS'09: 14th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 217–228, Washington, DC, Mar. 2009.

[7] 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.

[8] F. Chen, R. Lee, and X. Zhang. Essential roles of exploiting internal parallelism of Flash memory based solid state drives in high-speed data processing. In *HPCA'11: 17th IEEE Symp. on High Performance Computer Architecture*, pages 266–277, San Antonio, TX, Feb. 2011.

[9] S. Chen. FlashLogging: Exploiting Flash devices for synchronous logging performance. In *SIGMOD'09: 35th Int'l Conf. on Management of Data*, pages 73–86, Providence, RI, June 2009.

[10] Choi et al. A 20nm 1.8V 8Gb PRAM with 40MB/s program bandwidth. In *ISSCC'12: Int'l Solid-State Circuits Conf.*, San Francisco, CA, Feb. 2012.

[11] 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.

[12] De Sandre et al. A 4 Mb LV MOS-selected embedded phase change memory in 90 nm standard CMOS technology. *IEEE Journal of Solid-State Circuits*, 46(1):52–63, Jan. 2011.

[13] 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.

[14] 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.

[15] 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.

[16] A. Gulati, A. Merchant, and P. J. Varman. pClock: An arrival curve based approach for QoS guarantees in shared storage systems. In *ACM SIGMETRICS*, pages 13–24, San Diego, CA, June 2007.

[17] 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.

[18] 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.

[19] T. Kelly, A. H. Karp, M. Stiegler, T. Close, and H. K. Cho. Output-valid rollback-recovery. Technical Report HPL-2010-155, HP Laboratories, Oct. 2010.

[20] J. Kim, J. M. Kim, S. Noh, S. L. Min, and Y. Cho. A space-efficient Flash translation layer for CompactFlash systems. *IEEE Trans. on Consumer Electronics*, 48(2):366–375, May 2002.

[21] 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.

[22] 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.

[23] 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.

[24] J. Lee, S. Kim, H. Kwon, C. Hyun, S. Ahn, J. Choi, D. Lee, and S. H. Noh. Block recycling schemes and their cost-based optimization in NAND Flash memory based storage system. In *EMSOFT'07: 7th ACM Conf. on Embedded Software*, pages 174–182, Salzburg, Austria, Oct. 2007.

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

[26] P. Lu and K. Shen. Virtual machine memory access tracing with hypervisor exclusive cache. In *USENIX Annual Technical Conf.*, pages 29–43, Santa Clara, CA, June 2007.

[27] C. R. Lumb, A. Merchant, and G. A. Alvarez. Facade: Virtual storage devices with performance guarantees. In *FAST'03: Second USENIX Conf. on File and Storage Technologies*, pages 131–144, San Francisco, CA, Apr. 2003.

[28] E. B. Nightingale, K. Veeraraghavan, P. M. Chen, and J. Flinn. Rethink the sync. In *OSDI'06: 7th USENIX Symp. on Operating Systems Design and Implementation*, Seattle, WA, Nov. 2006.

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

[30] S. Park and K. Shen. A performance evaluation of scientific I/O workloads on flash-based SSDs. In *IASDS'09: Workshop on Interfaces and Architectures for Scientific Data Storage*, New Orleans, LA, Sept. 2009.

[31] M. Polte, J. Simsa, and G. Gibson. Comparing performance of solid state devices and mechanical disks. In *3rd Petascale Data Storage Workshop*, Austin, TX, Nov. 2008.

[32] A. Povzner, T. Kaldewey, S. Brandt, R. Golding, T. M. Wong, and C. Maltzahn. Efficient guaranteed disk request scheduling with Fahrrad. In *EuroSys'08: Third ACM European Conf. on Computer Systems*, pages 13–25, Glasgow, Scotland, Apr. 2008.

[33] A. L. N. Reddy, J. Wyllie, and K. B. R. Wijayaratne. Disk scheduling in a multimedia I/O system. *ACM Trans. on Multimedia Computing, Communications, and Applications*, 1(1):37–59, Feb. 2005.

[34] K. Shen. Request behavior variations. In *ASPLOS'10: 15th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 103–116, Pittsburg, PA, Mar. 2010.

[35] K. Shen, C. Stewart, C. Li, and X. Li. Reference-driven performance anomaly identification. In *ACM SIGMETRICS*, pages 85–96, Seattle, WA, June 2009.

[36] 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.

[37] 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.