Split-Level I/O Scheduling

Suli Yang, Tyler Harter, Nishant Agrawal, Salini Selvaraj Kowsalya, Anand Krishnamurthy,
Samer Al-Kiswany, Rini T. Kaushik*, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau

University of Wisconsin-Madison *IBM Research-Almaden**

Abstract

We introduce *split-level I/O scheduling*, a new framework that splits I/O scheduling logic across handlers at three layers of the storage stack: block, system call, and page cache. We demonstrate that traditional block-level I/O schedulers are unable to meet throughput, latency, and isolation goals. By utilizing the split-level framework, we build a variety of novel schedulers to readily achieve these goals: our Actually Fair Queuing scheduler reduces priority-misallocation by 28×; our Split-Deadline scheduler reduces tail latencies by 4×; our Split-Token scheduler reduces sensitivity to interference by 6×. We show that the framework is general and operates correctly with disparate file systems (ext4 and XFS). Finally, we demonstrate that split-level scheduling serves as a useful foundation for databases (SQLite and PostgreSQL), hypervisors (QEMU), and distributed file systems (HDFS), delivering improved isolation and performance in these important application scenarios.

1 Introduction

Deciding which I/O request to schedule, and when, has long been a core aspect of the operating system storage stack [11, 13, 22, 27, 28, 29, 31, 38, 43, 44, 45, 54]. Each of these approaches has improved different aspects of I/O scheduling; for example, research in single-disk schedulers incorporated rotational awareness [28, 29, 44]; other research tackled the problem of scheduling within a multi-disk array [53, 57]; more recent work has targeted flash-based devices [30, 36], tailoring the behavior of the scheduler to this new and important class of storage device. All of these optimizations and techniques are important; in sum total, these systems can improve overall system performance dramatically [22, 44, 57] as well as provide other desirable properties (including fairness across processes [17] and the meeting of deadlines [56]).

Most I/O schedulers (hereafter just “schedulers”) are built at the block level within an operating system, beneath the file system and just above the device itself. Such *block-level* schedulers are given a stream of requests and are thus faced with the question: which requests should be dispatched, and when, in order to achieve the goals of the system?

Unfortunately, making decisions at the block level is problematic, for two reasons. First, and most importantly, the block-level scheduler fundamentally cannot reorder certain write requests; file systems carefully control write ordering to preserve consistency in the event of system crash or power loss [21, 25]. Second, the block-level scheduler cannot perform accurate accounting; the scheduler lacks the requisite information to determine which application was responsible for a particular I/O request. Due to these limitations, block-level schedulers cannot implement a full range of policies.

An alternate approach, which does not possess these same limitations, is to implement scheduling much higher in the stack, namely with system calls [19]. *System-call scheduling* intrinsically has access to necessary contextual information (i.e., which process has issued an I/O). Unfortunately, system-call scheduling is no panacea, as the low-level knowledge required to build effective schedulers is not present. For example, at the time of a read or write, the scheduler cannot predict whether the request will generate I/O or be satisfied by the page cache, information which can be useful in reordering requests [12, 49]. Similarly, the file system will likely transform a single write request into a series of reads and writes, depending on the crash-consistency mechanism employed (e.g., journaling [25] or copy-on-write [42]); scheduling without exact knowledge of how much I/O load will be generated is difficult and error prone.

In this paper, we introduce *split-level I/O scheduling*, a novel scheduling framework in which a scheduler is constructed across several layers. By implementing a judiciously selected set of handlers at key junctures within the storage stack (namely, at the system-call, page-cache, and block layers), a developer can implement a scheduling discipline with full control over behavior and with no loss in high- or low-level information. Split schedulers can determine which processes issued I/O (via graph tags that track causality across levels) and accurately estimate I/O costs. Furthermore, memory notifications make

SOSP'15, October 4–7, 2015, Monterey, CA.

ACM 978-1-4503-3834-9/15/10...$15.00.
http://dx.doi.org/10.1145/2815400.2815421

schedulers aware of write work as soon as possible (not tens of seconds later when writeback occurs). Finally, split schedulers can prevent file systems from imposing orderings that are contrary to scheduling goals.

We demonstrate the generality of split scheduling by implementing three new schedulers: AFQ (Actually-Fair Queuing) provides fairness between processes, Split-Deadline observes latency goals, and Split-Token isolates performance. Compared to similar schedulers in other frameworks, AFQ reduces priority-misallocation errors by 28×, Split-Deadline reduces tail latencies by 4×, and Split-Token improves isolation by 6× for some workloads. Furthermore, the split framework is not specific to a single file system; integration with two file systems (ext4 [34] and XFS [47]) is relatively simple.

Finally, we demonstrate that the split schedulers provide a useful base for more complex storage stacks. Split-Token provides isolation for both virtual machines (QEMU) and data-intensive applications (HDFS), and Split-Deadline provides a solution to the database community's "`fsync` freeze" problem [2, 9, 10]. In summary, we find split scheduling to be simple and elegant, yet compatible with a variety of scheduling goals, file systems, and real applications.

The rest of this paper is organized as follows. We evaluate existing frameworks and describe the challenges they face (§2). We discuss the principles of split scheduling (§3) and our implementation in Linux (§4). We implement three split schedulers as case studies (§5), discuss integration with other file systems (§6), and evaluate our schedulers with real applications (§7). Finally, we discuss related work (§8) and conclude (§9).

2 Motivation

Block-level schedulers are severely limited by their inability to gather information from and exert control over other levels of the storage stack. As an example, we consider the Linux CFQ scheduler, which supports an `ionice` utility that can put a process in idle mode. According to the man page: *"a program running with idle I/O priority will only get disk time when no other program has asked for disk I/O"* [7]. Unfortunately, CFQ has little control over write bursts from idle-priority processes, as writes are buffered above the block level.

We demonstrate this problem by running a normal process `A` alongside an idle-priority process `B`. `A` reads sequentially from a large file. `B` issues random writes over a one-second burst. Figure 1 shows the result: `B` quickly finishes while `A` (whose performance is shown via the CFQ line) takes over five minutes to recover. Block-level schedulers such as CFQ are helpless to prevent processes from polluting write buffers with expensive I/O. As we will see, other file-system features such as journaling and delayed allocation are similarly problematic.

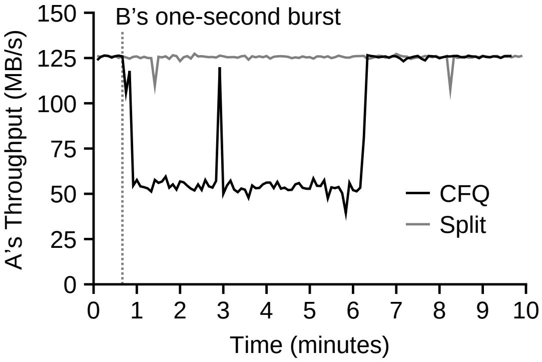

Figure 1: **Write Burst.** *`B`'s one-second random-write burst severely degrades `A`'s performance for over five minutes. Putting `B` in CFQ's idle class provides no help.*

The idle policy is one of many possible scheduling goals, but the difficulties it faces at the block level are not unique. In this section, we consider three different scheduling goals, identifying several shared needs (§2.1). Next, we describe prior scheduling frameworks (§2.2). Finally, we show these frameworks are fundamentally unable to meet scheduler needs when running in conjunction with a modern file system (§2.3).

2.1 Framework Support for Schedulers

We now consider three I/O schedulers: priority, deadline, and resource-limit, identifying what framework support is needed to implement these schedulers correctly.

Priority: These schedulers aim to allocate I/O resources fairly between processes based on their priorities [1]. To do so, a scheduler must be able to track which process is responsible for which I/O requests, estimate how much each request costs, and reorder higher-priority requests before lower-priority requests.

Deadline: These schedulers observe deadlines for I/O operations, offering predictable latency to applications that need it [3]. A deadline scheduler needs to map an application's deadline setting to each request and issue lower-latency requests before other requests.

Resource-Limit: These schedulers cap the resources an application may use, regardless of overall system load. Limits are useful when resources are purchased and the seller does not wish to give away free I/O. Resource-Limit schedulers need to know the cost and causes of I/O operations in order to throttle correctly.

Although these schedulers have distinct goals, they have three common needs. First, schedulers need to be able to *map causes* to identify which process is responsible for an I/O request. Second, schedulers need to *estimate costs* in order to optimize performance and perform accounting properly. Third, schedulers need to be able to *reorder* I/O requests so that the operations most important to achieving scheduling goals are served first. Unfortunately, as we will see, current block-level and system-call schedulers cannot meet all of these requirements.

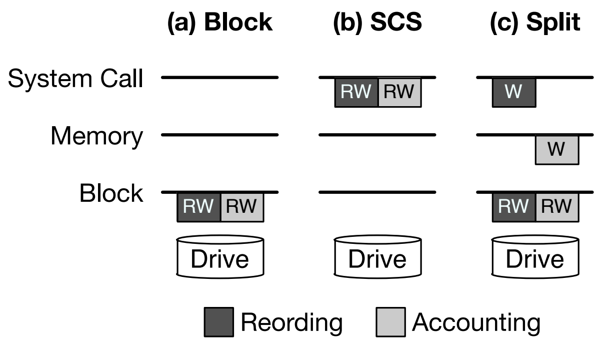

Figure 2: **Scheduling Architectures.** *The boxes show where scheduler hooks exist for reordering I/O requests or doing accounting. Sometimes reads and writes are handled differently at different levels, as indicated by "R" and "W".*

2.2 Framework Architectures

Scheduling frameworks offer hooks to which schedulers can attach. Via these hooks, a framework passes information and exposes control to schedulers. We categorize frameworks by the level at which the hooks are available.

Figure 2(a) illustrates block-level scheduling, the traditional approach implemented in Linux [8], FreeBSD [41], and other systems [6]. Clients initiate I/O requests via system calls, which are translated to block-level requests by the file system. Within the block-scheduling framework, these requests are then passed to the scheduler along with information describing them: their location on storage media, size, and the submitting process. Based on such information, a scheduler may reorder the requests according to some policy. For example, a scheduler may accumulate many requests in its internal queues and later dispatch them in an order that improves sequentiality.

Figure 2(b) show the system-call scheduling architecture (SCS) proposed by Craciunas *et al.* [19]. Instead of operating beneath the file system and deciding when block requests are sent to the storage device, a system-call scheduler operates on top of the file system and decides when to issue I/O related system calls (`read`, `write`, etc.). When a process invokes a system call, the scheduler is notified. The scheduler may put the process to sleep for a time before the body of the system call runs. Thus, the scheduler can reorder the calls, controlling when they become active within the file system.

Figure 2(c) shows the hooks of the split framework, which we describe in a later section (§4.2). In addition to introducing novel page-cache hooks, the split framework supports select system-call and block-level hooks.

2.3 File-System Challenges

Schedulers allocate disk I/O to processes, but processes do not typically use hard disks or SSDs directly. Instead, processes request service from a file system, which in turn translates requests to disk I/O. Unfortunately, file systems make it challenging to satisfy the needs of the scheduler. We now examine the implications of writeback, delayed allocation, journaling, and caching for schedulers, showing how these behaviors fundamentally require a restructuring of the I/O scheduling framework.

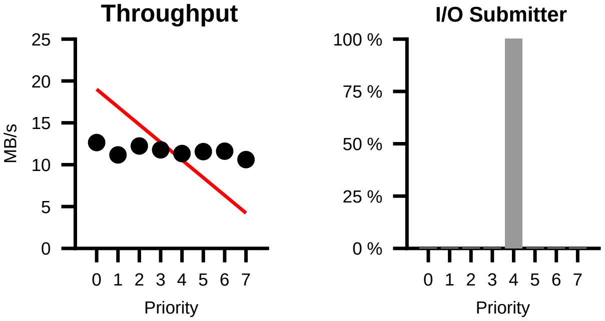

Figure 3: **CFQ Throughput.** *The left plot shows sequential write throughput for different priorities. The right plot the portion of requests for each priority seen by CFQ. Unfortunately, the "Completely Fair Scheduler" is not even slightly fair for sequential writes.*

2.3.1 Delayed Writeback and Allocation

Delayed writeback is a common technique for postponing I/O by buffering dirty data to write at a later time. Procrastination is useful because the work may go away by itself (e.g., the data could be deleted) and, as more work accumulates, more efficiencies can be gained (e.g., sequential write patterns may become realizable).

Some file systems also delay allocation to optimize data layout [34, 47]. When allocating a new block, the file system does not immediately decide its on-disk location; another task will decide later. More information (e.g., file sizes) becomes known over time, so delaying allocation enables more informed decisions.

Both delayed writeback and allocation involve file-system level delegation, with one process doing I/O work on behalf of other processes. A writeback process submits buffers that other processes dirtied and may also dirty metadata structures on behalf of other processes. Such delegation obfuscates the mapping from requests to processes. To block-level schedulers, the writeback task sometimes appears responsible for *all* write traffic.

We evaluate Linux's priority-based block scheduler, CFQ (Completely Fair Queuing) [1], using an asynchronous write workload. CFQ aims to allocate disk time fairly among processes (in proportion to priority). We launch eight threads with different priorities, ranging from 0 (highest) to 7 (lowest): each writes to its own file sequentially. A thread's write throughput should be proportional to its priority, as shown by the expectation line of Figure 3 (left). Unfortunately, CFQ ignores priorities, treating all threads equally. Figure 3 (right) shows why: to CFQ all the requests appear to have a priority of 4, because the writeback thread (a priority-4 process) submits all the writes on behalf of the eight benchmark threads.

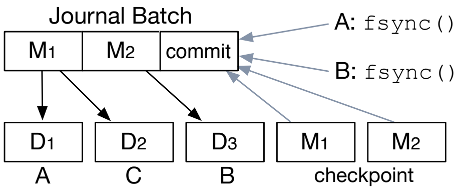

Figure 4: **Journal Batching.** *Arrows point to events that must occur before the event from which they point. The event for the blocks is a disk write. The event for an `fsync` is a return.*

2.3.2 Journaling

Many modern file systems use journals for consistent updates [15, 34, 47]. While details vary across file systems, most follow similar journaling protocols to commit data to disk; here, we discuss ext4's ordered-mode to illustrate how journaling severely complicates scheduling.

When changes are made to a file, ext4 first writes the affected data blocks to disk, then creates a journal transaction which contains all related metadata updates and commits that transaction to disk, as shown in Figure 4. The data blocks (D_1, D_2, D_3) must be written before the journal transaction, as updates become durable as soon as the transaction commits, and ext4 needs to prevent the metadata in the journal from referring to data blocks containing garbage. After metadata is journaled, ext4 eventually checkpoints the metadata in place.

Transaction writing and metadata checkpointing are both performed by kernel processes instead of the processes that initially caused the updates. This form of write delegation also complicates cause mapping.

More importantly, journaling prevents block-level schedulers from reordering. Transaction batching is a well-known performance optimization [25], but block schedulers have no control over which writes are batched, so the journal may batch together writes that are important to scheduling goals with less-important writes. For example, in Figure 4, suppose `A` is higher priority than `B`. `A`'s `fsync` depends on transaction commit, which depends on writing `B`'s data. Priority is thus inverted.

When metadata (e.g., directories or bitmaps) is shared among files, journal batching may be necessary for correctness (not just performance). In Figure 4, the journal could have conceivably batched M_1 and M_2 separately; however, M_1 depends on D_2, data written by a process C to a different file, and thus `A`'s `fsync` depends on the persistence of `C`'s data. Unfortunately (for schedulers), metadata sharing is common in file systems.

The inability to reorder is especially problematic for a deadline scheduler: a block-request deadline completely loses its relevance if one request's completion depends on the completion of unrelated I/Os. To demonstrate, we run two threads `A` (small) and `B` (big) with Linux's Block-Deadline scheduler [3], setting the block-request deadline to 20 ms for each. Thread `A` does 4 KB appends, calling `fsync` after each. Thread `B` does N bytes of random writes (N ranges from 16 KB to 4 MB) followed by an `fsync`. Figure 5 shows that even though `A` only writes one block each time, `A`'s `fsync` latency depends on how much data `B` flushes each time.

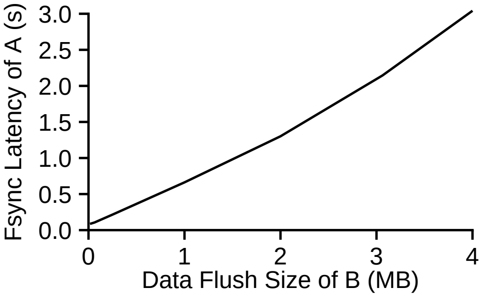

Figure 5: **I/O Latency Dependencies.** *Thread `A` keeps issuing `fsync` to flush one block of data to disk, while thread `B` flushes multiple blocks using `fsync`. This plot shows how `A`'s latency depends on `B`'s I/O size.*

Most file systems enforce ordering for correctness, so these problems occur with other crash-consistency mechanisms as well. For example, in log-structured files systems [42], writes appended earlier are durable earlier.

2.3.3 Caching and Write Amplification

Sequentially reading or writing N bytes from or to a file often does not result in N bytes of sequential disk I/O for several reasons. First, file systems use different disk layouts, and layouts change as file systems age; hence, sequential file-system I/O may become random disk I/O. Second, file-system reads and writes may be absorbed by caches or write buffers without causing I/O. Third, some file-system features amplify I/O. For example, reading a file block may involve additional metadata reads, and writing a file block may involve additional journal writes. These behaviors prevent system-call schedulers from accurately estimating costs.

To show how this inability hurts system-call schedulers, we evaluate SCS-Token [18]. In SCS-Token, a process's resource usage is limited by the number of tokens it possesses. Per-process tokens are generated at a fixed rate, based on user settings. When the process issues a system call, SCS blocks the call until the process has enough tokens to pay for it.

We attempt to isolate a process `A`'s I/O performance from a process `B` by throttling `B`'s resource usage. If SCS-Token works correctly, `A`'s performance will vary little with respect to `B`'s I/O patterns. To test this behavior, we configure `A` to sequentially read from a large file while `B` runs workloads with different I/O patterns. Each of the `B` workloads involve repeatedly accessing R bytes sequentially from a 10 GB file and then randomly seeking to a new offset. We explore 7 values for R (from 4 KB to 16 MB) for both reads and writes (14 workloads total). In each case, `B` is throttled to 10 MB/s.

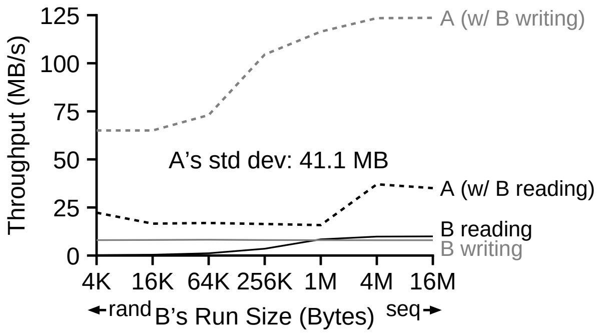

Figure 6: **SCS Token Bucket: Isolation.** *The performance of two processes is shown: a sequential reader, A, and a throttled process, B. B may read (black) or write (gray), and performs runs of different sizes (x-axis).*

Figure 6 shows how A's performance varies with B's I/O patterns. Note the large gap between the performance of A with B reading vs. writing. When B is performing sequential writes, A's throughput is as high as 125 MB/s; when B is performing random reads, A's throughput drops to 25 MB/s in the worst case. Writes appear cheaper than reads because write buffers absorb I/O and make it more sequential. Across the 14 workloads, A's throughput has a standard deviation of 41 MB, indicating A is highly sensitive to B's patterns. SCS-Token fails to isolate A's performance by throttling B, as SCS-Token cannot correctly estimate the cost of B's I/O pattern.

2.3.4 Discussion

Table 1 summarizes how different needs are met (or not) by each framework. The block-level framework fails to support correct cause mapping (due to write delegation such as journaling and delayed allocation) or control over reordering (due to file-system ordering requirements). The system-call framework solves these two problems, but fails to provide enough information to schedulers for accurate cost estimation because it lacks low-level knowledge. These problems are general to many file systems; even if journals are not used, similar issues arise from the ordering constraints imposed by other mechanisms such as copy-on-write techniques [16] or soft updates [21]. Our split framework meets all the needs in Table 1 by incorporating ideas from the other two frameworks and exposing additional memory-related hooks.

3 Split Framework Design

Existing frameworks offer insufficient reordering control and accounting knowledge. Requests are queued, batched, and processed at many layers of the stack, thus the limitations of single-layer frameworks are unsurprising. We propose a holistic alternative: all important decisions about when to perform I/O work should be exposed as scheduling hooks, regardless of the level at which those decisions are made in the stack. We now discuss how these hooks support correct cause mapping (§3.1), accurate cost estimation (§3.2), and reordering (§3.3).

	Block	Syscall	Split
Cause Mapping	✘	✔	✔
Cost Estimation	✔	✘	✔
Reordering	✘	✔	✔

Table 1: **Framework Properties.** *A ✔ indicates a given scheduling functionality can be supported with the framework, and an ✘ indicates a functionality cannot be supported.*

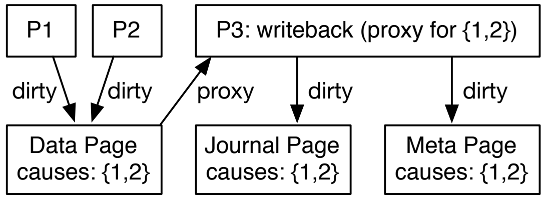

Figure 7: **Set Tags and I/O Proxies.** *Our tags map metadata and journal I/O to the real causes, P1 and P2, not P3.*

3.1 Cause Mapping

A scheduler must be able to map I/O back to the processes that caused it to accurately perform accounting even when some other process is submitting the I/O. Metadata is usually shared, and I/Os are usually batched, so there may be multiple causes for a single dirty page or a single request. Thus, the split framework tags I/O operations with sets of causes, instead of simple scalar tags (e.g., those implemented by Mesnier *et al.* [35]).

Write delegation (§2.3.1) further complicates cause mapping when one process is dirtying data (not just submitting I/O) on behalf of other processes. We call such processes *proxies*; examples include the writeback and journaling tasks. Our framework tags proxy process to identify the set of processes being served instead of the proxy itself. These tags are created when a process starts dirtying data for others and cleared when it is done.

Figure 7 illustrates how our framework tracks multiple causes and proxies. Processes P1 and P2 both dirty the same data page, so the page's tag includes both processes in its set. Later, a writeback process, P3, writes the dirty buffer to disk. In doing so, P3 may need to dirty the journal and metadata, and will be marked as a proxy for {P1, P2} (the tag is inherited from the page it is writing back). Thus, P1 and P2 are considered responsible when P3 dirties other pages, and the tag of these pages will be marked as such. The tag of P3 is cleared when it finishes submitting the data page to the block level.

3.2 Cost Estimation

Many policies require schedulers to know how much I/O costs, in terms of device time or other metrics. An I/O pattern's cost is influenced by file-system features, such as caches and write buffers, and by device properties (e.g., random I/O is cheaper on flash than hard disk).

Costs can be most accurately estimated at the lowest levels of the stack, immediately above hardware (or better still in hardware, if possible). At the block level, re-

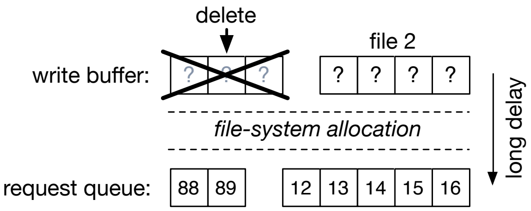

Figure 8: **Accounting: Memory vs. Block Level.** *Disk locations for buffered writes may not be known (indicated by the question marks on the blocks) if allocations are delayed.*

quest locations are known, so sequentiality-based models can estimate costs. Furthermore, this level is below all file-system features, so accounting is less likely to overestimate costs (e.g., by counting cache reads) or underestimate costs (e.g., by missing journal writes).

Unfortunately, writes may be buffered for a long time (e.g., 30 seconds) before being flushed to the block level. Thus, while block-level accounting may accurately estimate the cost of a write, it is not aware of most writes until some time after they enter the system via a `write` system call. Thus, if prompt accounting is more important than accurate accounting (e.g., for interactive systems), accounting should be done at the memory level. Without memory-level information, a scheduler could allow a low-priority process to fill the write buffers with gigabytes of random writes, as we saw earlier (Figure 1).

Figure 8 shows the trade-off between accounting at the memory level (write buffer) and block level (request queue). At the memory level, schedulers do not know whether dirty data will be deleted before a flush, whether other writers will overwrite dirty data, or whether I/O will be sequential or random. A scheduler can guess how sequential buffered writes will be based on file offsets, but delayed allocation prevents certainty about the layout. After a long delay, on-disk locations and other details are known for certain at the block level.

The cost of buffered writes depends on future workload behavior, which is usually unknown. Thus, we believe all scheduling frameworks are fundamentally limited and cannot provide cost estimation that is both prompt and accurate. Our framework exposes hooks at both the memory and block levels, enabling each scheduler to handle the trade-off in the manner most suitable to its goals. Schedulers may even utilize hooks at both levels. For example, Split-Token (§5.3) promptly guesses write costs as soon as buffers are dirtied, but later revises that estimate when more information becomes available (e.g., when the dirty data is flushed to disk).

3.3 Reordering

Most schedulers will want to reorder I/O to achieve good performance as well as to meet more specific goals (e.g., low latency or fairness). Reordering for performance requires knowledge of the device (e.g., whether it is useful to reorder for sequentiality), and is best done at a lower level in the stack. We enable reordering at the block level by exposing hooks for both block reads and writes.

Unfortunately, the ability to reorder writes at the block level is greatly limited by file systems (§2.3.2). Thus, reordering hooks for writes (but not reads, which are not entangled by journals) are also exposed above the file system, at the system-call level. By controlling when write system calls run, a scheduler can control when writes become visible to the file system and prevent ordering requirements that conflict with scheduling goals.

Many storage systems have calls that modify metadata, such as `mkdir` and `creat` in Linux; the split framework also exposes these. This approach presents an advantage over the SCS framework, which cannot correctly schedule these calls. In particular, the cost of a metadata update greatly depends on file-system internals, of which SCS schedulers are unaware. Split schedulers, however, can observe metadata writes at the block level and accordingly charge the responsible applications.

File-system synchronization points (e.g., `fsync` or similar) require all dependent data to be flushed to disk and typically invoke the file system's ordering mechanism. Unfortunately, logically independent operations often must wait for the synchronized updates to complete (§2.3.2), so the ability to schedule `fsync` is essential. Furthermore, writes followed by `fsync` are more costly than writes by themselves, so schedulers should be able to treat the two patterns differently. Thus, the split framework also exposes `fsync` scheduling.

4 Split Scheduling in Linux

Split-style scheduling could be implemented in a variety of storage stacks. In this work, we implement it in Linux, integrating with the ext4 and XFS file systems.

4.1 Cross-Layer Tagging

In Linux, I/O work is described by different function calls and data structures at different layers. For example, a write request may be represented by (a) the arguments to `vfs_write` at the system-call level, (b) a `buffer_head` structure in memory, and (c) a `bio` structure at the block level. Schedulers in our framework see the same requests in different forms, so it is useful to have a uniform way to describe I/O across layers. We add a `causes` tagging structure that follows writes through the stack and identifies the original processes that caused an I/O operation. Split schedulers can thereby correctly map requests back to the application from any layer.

Writeback and journal tasks are marked as I/O proxies, as described earlier (§3.1). In ext4, writeback calls the `ext4_da_writepages` function ("da" stands for "delayed allocation"), which writes back a range of pages of a given file. We modify this function so that as it does allocation for the pages, it sets the writeback thread's proxy

	Split Hook	Origin
System Call	`write-entry(size,…)`	SCS
	`write-return(rv,…)`	SCS
	`fsync-entry(…)`	*new*
	`fsync-return(rv,…)`	*new*
	`creat-entry(…)`	*new*
	`creat-return(rv,…)`	*new*
Mem	`buffer-dirty(oldset,new,…)`	*new*
	`buffer-free(set)`	*new*
Block	`req-add(…)`	block
	`req-complete(…)`	block

Table 2: **Split Hooks.** *The "Origin" column shows which hooks are new and which are borrowed from other frameworks.*

state as appropriate. For the journal proxy, we modify jbd2 (ext4's journal) to keep track of all tasks responsible for adding changes to the current transaction.

4.2 Scheduling Hooks

We now describe the hooks we expose, which are split across the system-call, memory, and block levels. Table 2 lists a representative sample of the split hooks.

System Call: These hooks allow schedulers to intercept entry and return points for various I/O system calls. A scheduler can delay the execution of a system call by simply sleeping in the entry hook. Like SCS, we intercept writes, so schedulers can separate writes before the file system entangles them. Unlike SCS, we do not intercept reads (no file-system mechanism entangles reads, so scheduling reads below the cache is preferable). Two metadata-write calls, `creat` and `mkdir`, and the Linux synchronization call, `fsync`, are also exposed to the scheduler. It would be useful (and straightforward) to support other metadata calls in the future (e.g., `unlink`).

Note that in our implementation, the caller is blocked until the system call is scheduled. Other implementations are possible, such as buffering the system calls and returning immediately, or simply returning EAGAIN to tell the caller to issue the system call later. We choose this particular implementation because of its simplicity and POSIX compliance. Linux itself blocks `writes` (when there are too many dirty pages) and `fsyncs`, and most applications already deal with this behavior using separate threads; what we do is no different.

Memory: These hooks expose page-cache internals to schedulers. In Linux, a writeback thread (`pdflush`) decides when to pass I/O to the block-level scheduler, which then decides when to pass that I/O to disk. Both components are performing scheduling tasks, and separating them is inefficient (e.g., writeback could flush more aggressively if it knew when the disk was idle). We add two hooks to inform the scheduler when buffers are dirtied or deleted. The `buffer-dirty` hook notifies the scheduler when a process dirties a buffer or when a dirty buffer is modified. In the latter case, the framework tells the scheduler which processes previously dirtied the buffer; depending on policy, the scheduler could revise accounting statistics, shifting some (or all) of the responsibility for the I/O to the last writer. The `buffer-free` hooks tell the scheduler if a buffer is deleted before writeback. Schedulers can either rely on Linux to perform writeback and throttle `write` system calls to control how much dirty data accumulates before writeback, or they can take complete control of the writeback. We evaluate the trade-off of these two approaches later (§7.1.2).

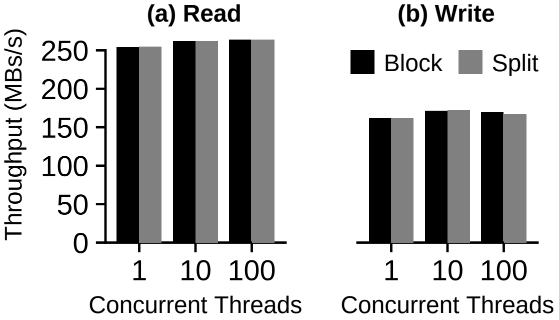

Figure 9: **Time Overhead.** *The split framework scales well with the number of concurrent threads doing I/O to an SSD.*

Block: These hooks are identical to those in Linux's original scheduling framework; schedulers are notified when requests are added to the block level or completed by the disk. Although we did not modify the function interfaces at this level, schedulers implementing these hooks in our framework are more informed, given tags within the request structures that identify the responsible processes. The Linux scheduling framework has over a dozen other block-level hooks for initialization, request merging, and convenience. We support all these as well for compatibility, but do not discuss them here.

Implementing the split-level framework in Linux involves ∼300 lines of code, plus some file-system integration effort, which we discuss later (§6). While representing a small change in the Linux code base, it enables powerful scheduling capabilities, as we will show.

4.3 Overhead

In this section, we evaluate the time and space overhead of the split framework. In order to isolate framework overhead from individual scheduler overhead, we compare no-op schedulers implemented in both our framework and the block framework (a no-op scheduler issues all I/O immediately, without any reordering). Figure 9 shows our framework imposes no noticeable time overhead, even with 100 threads.

The split framework introduces some memory overhead for tagging writes with `causes` structures (§4.1). Memory overheads roughly correspond to the number of dirty write buffers. To measure this overhead, we instru-

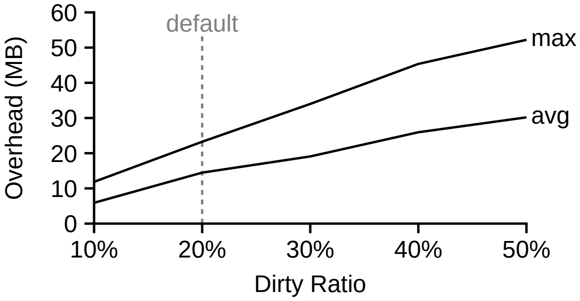

Figure 10: **Space Overhead.** *Memory overhead is shown for an HDFS worker with 8 GB of RAM under a write-heavy workload. Maximum and average overhead is measured as a function of the Linux `dirty_ratio` setting. `dirty_background_ratio` is set to half of `dirty_ratio`.*

ment `kmalloc` and `kfree` to track the number of bytes allocated for tags over time. For our evaluation, we run HDFS with a write-heavy workload, measuring allocations on a single worker machine. Figure 10 shows the results: with the default Linux settings, average overhead is 14.5 MB (0.2% of total RAM); the maximum is 23.3 MB. Most tagging is on the write buffers; thus, a system tuned for more buffering should have higher tagging overheads. With a 50% `dirty_ratio` [5], maximum usage is still only 52.2 MB (0.6% of total RAM).

5 Scheduler Case Studies

In this section, we evaluate the split framework's ability to support a variety of scheduling goals. We implement AFQ (§5.1), Split-Deadline (§5.2), and Split-Token (§5.3), and compare these schedulers to similar schedulers in other frameworks. Unless otherwise noted, all experiments run on top of ext4 with the Linux 3.2.51 kernel (most XFS results are similar but usually not shown). Our test machine has an eight-core, 1.4 GHz CPU and 16 GB of RAM. We use 500 GB Western Digital hard drives (AAKX) and an 80 GB Intel SSD (X25-M).

5.1 AFQ: Actually Fair Queuing

As shown earlier (§2.1), CFQ's inability to correctly map requests to processes causes unfairness, due to the lack of information Linux's elevator framework provides. Moreover, file-system ordering requirements limit CFQ's reordering options, causing priority inversions. In order to overcome these two drawbacks, we introduce AFQ (Actually Fair Queuing scheduler) to allocate I/O fairly among processes according to their priorities.

Design: AFQ employs a two-level scheduling strategy. Reads are handled at the block level and writes (and calls that cause writes, such as `fsync`) are handled at the system-call level. This design allows reads to hit the cache while protecting writes from journal entanglement. Beneath the journal, low-priority blocks may be prerequisites for high-priority `fsync` calls, so writes at the block level are dispatched immediately.

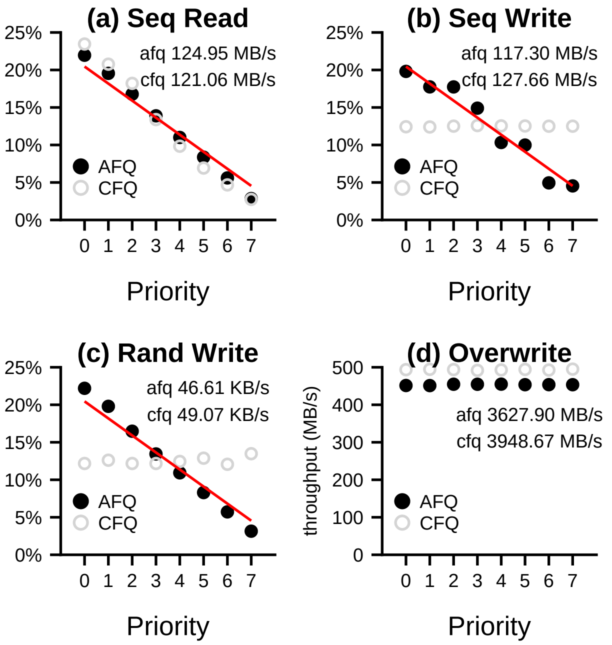

Figure 11: **AFQ Priority.** *The plots show the percentage of throughput that threads of each priority receive. The lines show the goal distributions; the labels indicate total throughput.*

AFQ chooses I/O requests to dequeue at the block and system-call levels using the stride algorithm [51]. Whenever a block request is dispatched to disk, AFQ charges the responsible processes for the disk I/O. The I/O cost is based on a simple seek model.

Evaluation: We compare AFQ to CFQ with four workloads, shown in Figure 11. Figure 11(a) shows read performance on AFQ and CFQ for eight threads, with priorities ranging from 0 (high) to 7 (low), each reading from its own file sequentially. We see that AFQ's performance is similar to CFQ, and both respect priorities.

Figure 11(b) shows asynchronous sequential-write performance, again with eight threads. This time, CFQ fails to respect priorities because of write delegation, but AFQ correctly maps I/O requests via split tags, and thus respects priorities. On average, CFQ deviates from the ideal by 82%, AFQ only by 16% (a 5× improvement).

Figure 11(c) shows synchronous random-write performance: we set up 5 threads per priority level, and each keeps randomly writing and flushing (with `fsync`) 4 KB blocks. The average throughput of threads at each priority level is shown. CFQ again fails to respect priority; using `fsync` to force data to disk invokes ext4's journaling mechanism and keeps CFQ from reordering to favor high-priority I/O. AFQ, however, blocks low-priority `fsyncs` when needed, improving throughput for high-priority threads. As shown, AFQ is able to respect priority, deviating from the ideal value only by 3% on average while CFQ deviates by 86% (a 28× improvement).

Finally, Figure 11(d) shows throughput for a memory-intense workload that just overwrites dirty blocks in the

	A		B	
	Block Write	Fsync	Block Write	Fsync
HDD	10 ms	100 ms	100 ms	6000 ms
SSD	1 ms	3 ms	10 ms	100 ms

Table 3: **Deadline Settings.** *For Block-Deadline, we set deadlines for block-level writes; for Split-Deadline, we set deadlines for `fsync`s.*

write buffer. One thread at each priority level keeps overwriting 4 MB of data in its own file. Both CFQ and AFQ get very high performance as expected, though AFQ is slightly slower (AFQ needs to do significant bookkeeping for each write system call). The plot has no fairness goal line as there is no contention for disk resources.

In general, AFQ and CFQ have similar performance; however, AFQ always respects priorities, while CFQ only respects priorities for the read workloads.

5.2 Deadline

As shown earlier (Figure 5 in §2.3.2), Block-Deadline does poorly when trying to limit tail latencies, due to its inability to reorder block I/Os in the presence of file-system ordering requirements. Split-level scheduling, with system-call scheduling capabilities and memory-state knowledge, is better suited to this task.

Design: We implement the Split-Deadline scheduler by modifying the Linux deadline scheduler (Block-Deadline). Block-Deadline maintains two deadline queues and two location queues (for both read and write requests) [3]. In Split-Deadline, an `fsync`-deadline queue is used instead of a block-write deadline queue. During operation, if no read request or `fsync` is going to expire, block-level read and write requests are issued from the location queues to maximize performance. If some read requests or `fsync` calls are expiring, they are issued before their deadlines.

Split-Deadline monitors how much data is dirtied for one file using the `buffer-dirty` hook and thereby estimates the cost of an `fsync`. If there is an `fsync` pending that may affect other processes by causing too much I/O, it will not be issued directly. Instead, the scheduler asks the kernel to launch asynchronous writeback of the file's dirty data and waits until the amount of dirty data drops to a point such that other deadlines would not be affected by issuing the `fsync`. Asynchronous writeback does not generate a file-system synchronization point and has no deadline, so other operations are not forced to wait.

Evaluation: We compare Split-Deadline to Block-Deadline for a database-like workload on both hard disk drive (HDD) and solid state drive (SSD). We set up two threads `A` (small) and `B` (big); thread `A` appends to a small file one block (4 KB) at a time and calls `fsync` (this mimics database log appends) while thread `B` writes 1024 blocks randomly to a large file and then calls `fsync` (this mimics database checkpointing).

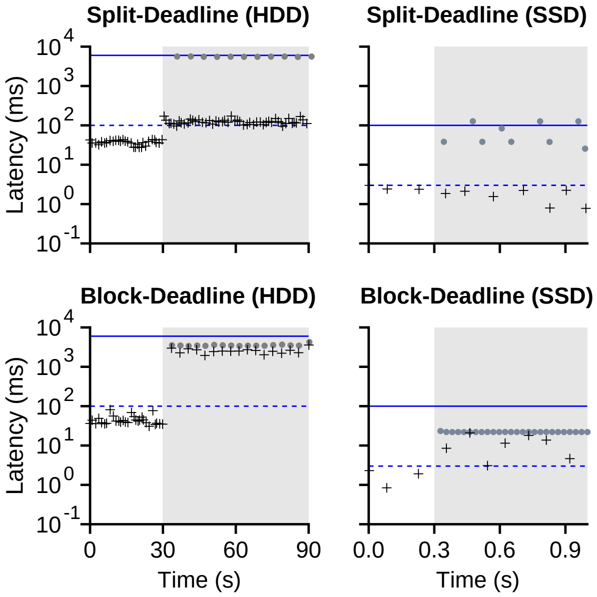

Figure 12: **Fsync Latency Isolation.** *Dashed and solid lines present the goal latencies of `A` and `B` respectively. Dots represent the actual latency of `B`'s calls, and pluses represent the actual latency of `A`'s calls. The shaded area represents the time when `B`'s `fsync`s are being issued.*

The deadline settings are shown in Table 3. We choose shorter block-write deadlines than `fsync` deadlines because each `fsync` causes multiple block writes; however, our results do not appear sensitive to the exact values chosen. Linux's Block-Deadline scheduler does not support setting different deadlines for different processes, so we add this feature to enable a fair comparison.

Figure 12 shows the experiment results on both HDD and SSD. We can see that when no I/O from `B` is interfering, both schedulers give `A` low-latency `fsync`s. After `B` starts issuing big `fsync`s, however, Block-Deadline starts to fail: `A`'s `fsync` latencies increase by an order of magnitude; this happens because `B` generates too much bursty I/O when calling `fsync`, and the scheduler has no knowledge of or control over when they are coming. Worse, `A`'s operations become dependent on these I/Os.

With Split-Deadline, however, `A`'s `fsync` latencies mostly fluctuate around the deadline, even when `B` is calling `fsync` after large writes. Sometimes `A` exceeds its goal slightly because our estimate of the `fsync` cost is not perfect, but latencies are always relatively near the target. Such performance isolation is possible because Split-Deadline can reorder to spread the cost of bursty I/Os caused by `fsync` without forcing others to wait.

5.3 Token Bucket

Earlier, we saw that SCS-Token [18] fails to isolate performance (Figure 6 in §2.3.3). In particular, the throughput of a process `A` was sensitive to the activities of another process `B`. SCS underestimates the I/O cost of some `B` workloads, and thus does not sufficiently throttle `B`. In

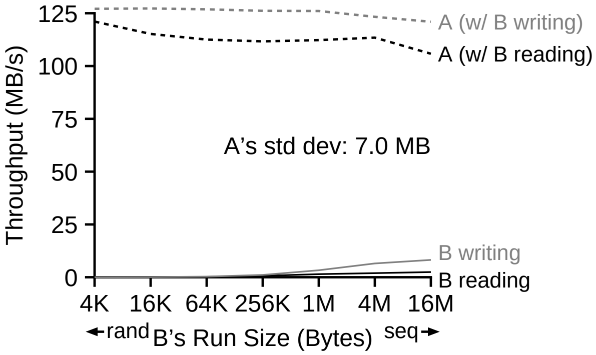

Figure 13: **Isolation: Split-Token with ext4.** *The same as Figure 6, but for our Split-Token implementation. A is the unthrottled sequential reader, and B is the throttled process performing I/O of different run sizes.*

this section, we evaluate Split-Token, a reimplementation of token bucket in our framework.

Design: As with SCS-Token, throttled processes are given tokens at a set rate. I/O costs tokens, I/O is blocked if there are no tokens, and the number of tokens that may be held is capped. Split-Token throttles a process's system-call writes and block-level reads if and only if the number of tokens is negative. System-call reads are never throttled (to utilize the cache). Block writes are never throttled (to avoid entanglement).

Our implementation uses memory-level and block-level hooks for accounting. The scheduler promptly charges tokens as soon as buffers are dirtied, and then revises when the writes are later flushed to the block level (§3.2), charging more tokens (or refunding them) based on amplification and sequentiality. Tokens represent bytes, so accounting normalizes the cost of an I/O pattern to the equivalent amount of sequential I/O (e.g., 1 MB of random I/O may be counted as 10 MB).

Split-Token estimates I/O cost based on two models, both of which assume an underlying hard disk (simpler models could be used on SSD). When buffers are first dirtied at the memory level, a preliminary model estimates cost based on the randomness of request offsets within the file. Later, when the file system allocates space on disk for the requests and flushes them to the block level, a disk model revises the cost estimate. The second model is more accurate because it can consider more factors than the first model, such as whether the file system introduced any fragmentation, and whether the file is located near other files being written.

Evaluation: We repeat our earlier SCS experiments (Figure 6) with Split-Token, as shown in Figure 13. We observe that whether B does reads or writes has little effect on A (the A lines are near each other). Whether B's pattern is sequential or random also has little impact (the lines are flat). Across all workloads, the standard deviation of A's performance is 7 MB, about a 6× improvement over SCS (SCS-Token's deviation was 41 MB).

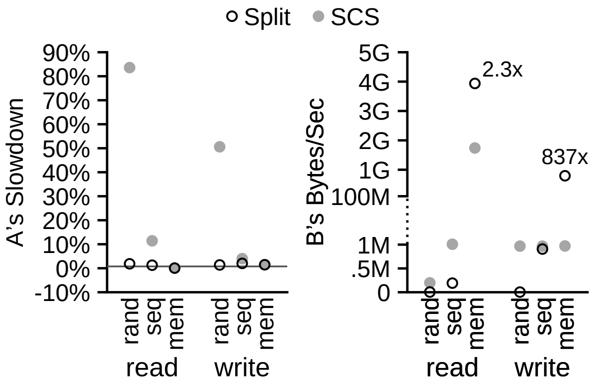

Figure 14: **Split-Token vs. SCS-Token.** *Left: A's throughput slowdown is shown. Right: B's performance is shown. Process A achieves about 138 MB/s when running alone, and B is throttled to 1 MB/s of normalized I/O, so there should be a 0.7% slowdown for A (shown by a target line). The x-axis indicates B's workload; A always reads sequentially.*

We now directly compare SCS-Token with Split-Token using a broader range of read and write workloads for process B. I/O can be random (expensive), sequential, or served from memory (cheap). As before, A is an unthrottled reader, and B is throttled to 1 MB/s of normalized I/O. Figure 14 (left) shows that Split-Token is near the isolation target all six times, whereas SCS-Token significantly deviates three times (twice by more than 50%), again showing Split-Token provides better isolation.

After isolation, a secondary goal is the best performance for throttled processes, which we measure in Figure 14 (right). Sometimes B is faster with SCS-Token, but only because SCS-Token is incorrectly sacrificing isolation for A (e.g., B does faster random reads with SCS-Token, but A's performance drops over 80%). We consider the cases where SCS-Token did provide isolation. First, Split-Token is 2.3× faster for "read-mem". SCS-Token logic must run on every read system call, whereas Split-Token does not. SCS-Token still achieves nearly 2 GB/s, though, indicating cache hits are not throttled. Although the goal of SCS-Token was to do system-call scheduling, Craciunas *et al.* needed to modify the file system to tell which reads are cache hits [19]. Second, Split-Token is 837× faster for "write-mem". SCS-Token does write accounting at the system-call level, so it does not differentiate buffer overwrites from new writes. Thus, SCS-Token unnecessarily throttles B. With Split-Token, B's throughput does not reach 1 MB/s for "read-seq" because the intermingled I/Os from A and B are no longer sequential; we charge it to both A and B.

We finally evaluate Split-Token for a large number of threads; we repeat the six workloads of Figure 14, this time varying the number of B threads performing the I/O task (all threads of B share the same I/O limit). Figure 15 shows the results. For sequential read, the number of B threads has no impact on A's performance, as desired.

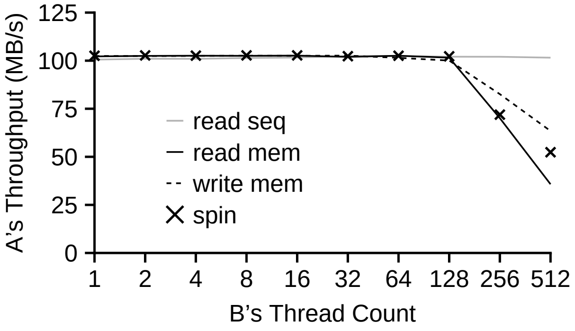

Figure 15: **Split-Token Scalability.** *A's throughput is shown as a function of the number of B threads performing a given activity. Goal performance is 101.7 MB (these numbers were taken on a 32-core CloudLab node with a 1 TB drive).*

We do not show random read, sequential write, or random write, as these lines would appear the same as the read-sequential line (varying at most by 1.7%). However, when B is reading or writing to memory, A's performance is only steady if B has 128 threads or less. Since the B threads do not incur any disk I/O, our I/O scheduler does not throttle them, leaving the B threads free to dominate the CPU, indirectly slowing A. To confirm this, we do an experiment (also shown in Figure 15) where B threads simply execute a spin loop, issuing no I/O; A's performance still suffers in this case. This reminds us of the usefulness of CPU schedulers in addition to I/O schedulers: if a process does not receive enough CPU time, it may not be able to issue requests fast enough to fully utilize the storage system.

5.4 Implementation Effort

Implementing different schedulers within the split framework is not only possible, but relatively easy: Split-AFQ takes ∼950 lines of code to implement, Split-Deadline takes ∼750 lines of code, and Split-Token takes ∼950 lines of code. As a comparison, Block-CFQ takes more than 4000 lines of code, Block-Deadline takes ∼500 lines of code, and SCS-Token takes ∼2000 lines of code (SCS-Token is large because there is not a clean separation between the scheduler and framework).

6 File System Integration

Thus far we have presented results with ext4; now, we consider the effort necessary to integrate ext4 and other file systems, in particular XFS, into the split framework.

Integrating a file system involves (a) tagging relevant data structures the file system uses to represent I/O in memory and (b) identifying the proxy mechanisms in the file system and properly tagging the proxies.

In Linux, part (a) is mostly file-system independent as many file systems use generic page buffer data structures to represent I/O. Both ext4 and XFS rely heavily on the `buffer_head` structure, which we already tag properly. Thus we are able to integrate XFS buffers with split tags by adding just two lines of code, and ext4 with less than 10 lines. In contrast, btrfs [33] uses its own buffer structures, so integration would require more effort.

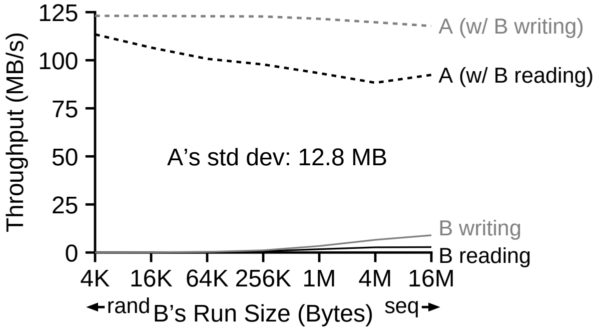

Figure 16: **Isolation: Split-Token with XFS.** *This is the same as Figure 6 and Figure 14, but for XFS running with our Split implementation of token bucket.*

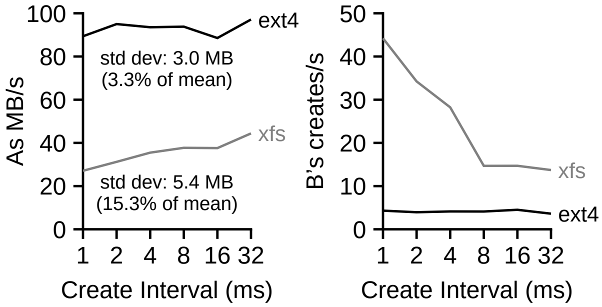

Figure 17: **Metadata: Split-Token with XFS and ext4.** *Process A sequentially reads while B creates and flushes new, empty files. A's throughput is shown as function of how long B sleeps between operations (left). B's create frequency is also shown for the same experiments (right).*

Part (b), on the other hand, is highly file-system specific, as different file systems use different proxy mechanisms. For ext4, the journal task acts as a proxy when writing the physical journal, and the writeback task acts as a proxy when doing delayed allocation. XFS uses logical journaling, and has its own journal implementation. For a copy-on-write file system, garbage collection would be another important proxy mechanism. Properly tagging these proxies is a bit more involved. In ext4, it takes 80 lines of code across 5 different files. Fortunately, such proxy mechanisms typically only involve metadata, so for data-dominated workloads, partial integration with only (a) should work relatively well.

In order to verify the above hypotheses, we have fully integrated ext4 with the split framework, and only partially integrated XFS with part (a). We evaluate the effectiveness of our partial XFS integration on both data-intensive and metadata-intensive workloads.

Figure 16 repeats our earlier isolation experiment (Figure 13), but with XFS; these experiments are data-intensive. Split-Token again provides significant isolation, with A only having a deviation of 12.8 MB. In fact, all the experiments we show earlier are data intensive, and XFS has similar results (not shown) as ext4.

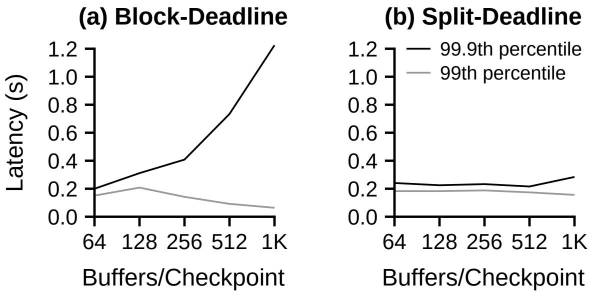

Figure 18: **SQLite Transaction Latencies.** *99th and 99.9th percentiles of the transaction latencies are shown. The x-axis is the number of dirty buffers we allow before checkpoint.*

Figure 17 shows the performance of a metadata-intense workload for both XFS and ext4. In this experiment, A reads sequentially while B repeatedly creates empty files and flushes them to disk with fsync. B is throttled, A is not. B sleeps between each create for a time varied on the x-axis. As shown in the left plot, B's sleep time influences A's performance significantly with XFS, but with ext4 A is isolated. The right plot explains why: with ext4, B's creates are correctly throttled, regardless of how long B sleeps. With XFS, however, B is unthrottled because XFS does not give the scheduler enough information to map the metadata writes (which are performed by journal tasks) back to B.

We conclude that some file systems can be partially integrated with minimal effort, and data-intense workloads will be well supported. Support for metadata workloads, however, requires more effort.

7 Real Applications

In this section, we explore whether the split framework is a useful foundation for databases (§7.1), virtual machines (§7.2), and distributed file systems (§7.3).

7.1 Databases

To show how real databases could benefit from Split-Deadline's low-latency fsyncs, we measure transaction-response time for SQLite3 [26] and PostgreSQL [10] running with both Split-Deadline and Block-Deadline.

7.1.1 SQLite3

We run SQLite3 on a hard disk drive. For Split-Deadline, we set short deadlines (100 ms) for fsyncs on the write-ahead log file and reads from the database file and set long deadlines (10 seconds) for fsyncs on the database file. For Block-Deadline, the default settings (50 ms for block reads and 500 ms for block writes) are used. We make minor changes to SQLite to allow concurrent log appends and checkpointing and to set appropriate deadlines. For our benchmark, we randomly update rows in a large table, measure transaction latencies, and run checkpointing in a separate thread whenever the number of dirty buffers reaches a threshold.

Figure 18(a) shows the transaction tail latencies (99th and 99.9th percentiles) when we change the checkpointing threshold. When checkpoint thresholds are larger, checkpointing is less frequent, fewer transactions are affected, and thus the 99th line falls. Unfortunately, this approach does not eliminate tail latencies; instead, it concentrates the cost on fewer transactions, so the 99.9th line continues to rise. In contrast, Figure 18(b) shows that Split-Deadline provides much smaller tail latencies (a 4× improvement for 1K buffers).

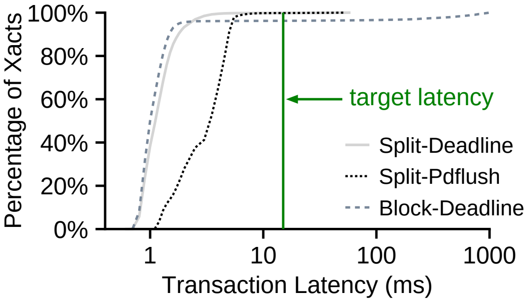

Figure 19: **PostgreSQL Transaction Latencies.** *A CDF of transaction latencies is shown for three systems. Split-Pdflush is Split-Deadline, but with pdflush controlling write-back separately.*

7.1.2 PostgreSQL

We run PostgreSQL [10] on top of an SSD and benchmark it using pgbench [4], a TPC-B like workload. We change PostgreSQL to set I/O deadlines for each worker thread. We want consistently low transaction latencies (within 15 ms), so we set the foreground fsync deadline to 5 ms, and the background checkpointing fsync deadline to 200 ms for Split-Deadline. For Block-Deadline, we set the block write deadline to 5 ms. For block reads, a deadline of 5 ms is used for both Split-Deadline and Block-Deadline. Checkpoints occur every 30 seconds.

Figure 19 shows the cumulative distribution of the transaction latencies. We can see that when running on top of Block-Deadline, 4% of transactions fail to meet their latency target, and over 1% take longer than 500 ms. After further inspection, we found that the latency spikes happen at the end of each checkpoint period, when the system begins to flush a large amount of dirty data to disk using fsync. Such data flushing interferes with foreground I/Os, causes long transaction latency and low system throughput. The database community has long experienced this "fsync freeze" problem, and has no great solution for it [2, 9, 10]. We show next that Split-Deadline provides a simple solution to this problem.

When running Split-Deadline, we have the ability to schedule fsyncs and minimize their performance impact to foreground transactions. However, pdflush (Linux's writeback task) may still submit many writeback I/Os without scheduler involvement and interfere with foreground I/Os. Split-Deadline maintains deadlines in this case by limiting the amount of data pdflush may flush at any given time by throttling write system calls. In Fig-

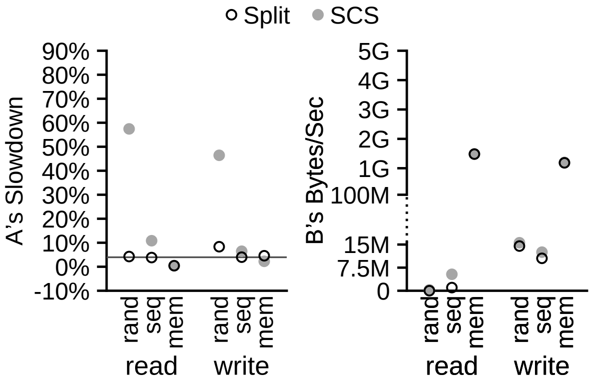

Figure 20: **QEMU Isolation.** *This is the same as Figure 14, but processes* `A` *and* `B` *run in different QEMU virtual machines ext4 on the host.* `B` *is throttled to 5 MB/s. Reported throughput is for the processes at the guest system-call level.*

ure 19 we can see that this approach effectively eliminates tail latency: 99.99% of the transactions are completed within 15 ms. Unfortunately, the median transaction latency is much higher because write buffers are not fully utilized.

When `pdflush` is disabled, though, Split-Deadline has complete control of writeback, and can allow more dirty data in the system without worrying about untimely writeback I/Os. It then initiates writeback in a way that both observes deadlines and optimizes performance, thus eliminating tail latencies while maintaining low median latencies, as shown in Figure 19.

7.2 Virtual Machines (QEMU)

Isolation is especially important in cloud environments, where customers expect to be isolated from other (potentially malicious) customers. To evaluate our framework's usefulness in this environment, we repeat our token-bucket experiment in Figure 14, this time running the unthrottled process `A` and throttled process `B` in separate QEMU instances. The guests run a vanilla kernel; the host runs our modified kernel. Thus, throttling is on the whole VM, not just the benchmark we run inside. We use an 8 GB machine with a four-core 2.5 GHz CPU.

Figure 20 shows the results for QEMU running over both SCS and Split-Token on the host. The isolation results for `A` (left) are similar to the results when we ran `A` and `B` directly on the host (Figure 14): with Split-Token, `A` is always well isolated, but with SCS, `A` experiences major slowdowns when `B` does random I/O.

The throughput results for `B` (right) are more interesting: whereas before SCS greatly slowed memory-bound workloads, now SCS and Split-Token provide equal performance for these workloads. This is because when a throttled process is memory bound, it is crucial for performance that a caching/buffering layer exist above the scheduling layer. The split and QEMU-over-SCS stacks have this property (and memory workloads are fast), but the raw-SCS stack does not.

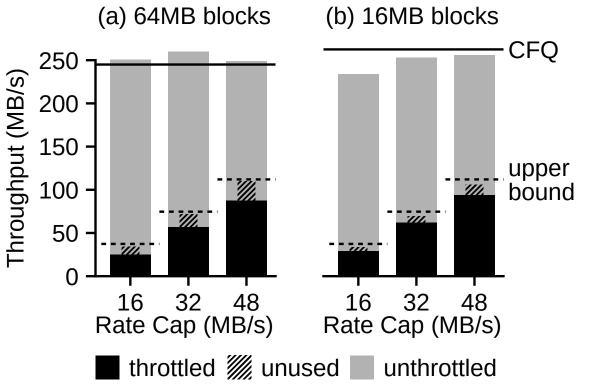

Figure 21: **HDFS Isolation.** *Solid-black and gray bars represent the total throughput of throttled and unthrottled HDFS writers, respectively. Dashed lines represent an upper bound on throughput; solid lines represent Block-CFQ throughput.*

7.3 Distributed File Systems (HDFS)

To show that local split scheduling is a useful foundation to provide isolation in a distributed environment, we integrate HDFS with Split-Token to provide isolation to HDFS clients. We modify the client-to-worker protocol so workers know which account should be billed for disk I/O generated by the handling of a particular RPC call. Account information is propagated down to Split-Token and across to other workers (for pipelined writes).

We evaluate our modified HDFS on a 256-core CloudLab cluster (one NameNode and seven workers, each with 32 cores). Each worker has 8 GB of RAM and a 1 TB disk. We run an unthrottled group of four threads and a throttled group of four threads. Each thread sequentially writes to its own HDFS file.

Figure 21(a) shows the result for varying rate limits on the x-axis. The summed throughput (i.e., that of both throttled and unthrottled writers) is similar to throughput when HDFS runs over CFQ without any priorities set. With Split-Token, though, smaller rate caps on the throttled threads provide the unthrottled threads with better performance (e.g., the gray bars get more throughput when the black bars are locally throttled to 16 MB/s).

Given there are seven datanodes, and data must be triply written for replication, the expected upper bound on total I/O is $(ratecap/3) * 7$. The dashed lines show these upper bounds in Figure 21(a); the black bars fall short. We found that many tokens go unused on some workers due to load imbalance. The hashed black bars represent the potential HDFS write I/O that was thus lost.

In Figure 21(b), we try to improve load balance by decreasing the HDFS block size from 64 MB (the default) to 16 MB. With smaller blocks, fewer tokens go unused, and the throttled writers achieve I/O rates nearer the upper bound. We conclude that local scheduling can be used to meet distributed isolation goals; however, throttled applications may get worse-than-expected performance if the system is not well balanced.

8 Related Work

Multi-Layer Scheduling: A number of works argue that efficient I/O scheduling requires coordination at multiple layers in the storage stack [45, 50, 52, 56]. Riska *et al.* [40] evaluated the effectiveness of optimizations at various layers of the I/O path, and found that superior performance is yielded by combining optimizations at various layers. Redline [56] tries to avoid system unresponsiveness during `fsync` by scheduling at both the buffer cache level and the block level. Argon [50] combines mechanisms at different layers to achieve performance insulation. However, compared to these ad-hoc approaches, our framework provides a systematic way for schedulers to plug in logic at different layers of the storage stack while still maintaining modularity.

Cause Mapping and Tagging: The need for correctly accounting resource consumption to the responsible entities arises in different contexts. Banga *et al.* [14] found that kernel consumes resources on behalf of applications, causing difficulty in scheduling. The hypervisor may also do work on behalf of a virtual machine, making it difficult to isolate performance [24]. We identify the same problem in I/O scheduling, and propose tagging as a general solution. Both Differentiated Storage Services (DSS) [35] and IOFlow [48] also tag data across layers. DSS tags the type of data, IOFlow tags the type and cause, and split scheduling tags with a set of causes.

Software-Defined Storage Stack: In the spirit of moving toward a more software-defined storage (SDS) stack, the split-level framework exposes knowledge and control at different layers to a centralized entity, the scheduler. The IOFlow [48] stack is similar to split scheduling in this regard; both tag I/O across layers and have a central controller.

IOFlow, however, operates at the distributed level; the lowest IOFlow level is an SMB server that resides above a local file system. IOFlow does not address the core file-system issues, such as write delegation or ordering requirements, and thus likely has the same disadvantages as system-call scheduling. We believe that the problems introduced by the local file systems, which we identify and solve in this paper, are inherent to any storage stack. We argue any complete SDS solutions would need to solve them and thus our approach is complementary. Combining IOFlow with split scheduling, for example, could be very useful: flows could be tracked through hypervisor, network, and local-storage layers.

Shue *et al.* [46] provision I/O resources in a key-value store (Libra) by co-designing the application and I/O scheduler; however, they noted that “OS-level effects due to filesystem operations [. . .]are beyond Libra's reach”; building such applications with the split framework should provide more control.

Exposing File-System Mechanisms: Split-level scheduling requires file systems to expose certain mechanisms (journaling, delayed allocation, etc.) to the framework by properly tagging them as proxies. Others have also found that exposing file-system information is helpful [20, 37, 55]. For example, in Featherstitch [20], file-system ordering requirements are exposed to the outside as dependency rules so that the kernel can make informed decisions about writeback.

Other I/O Scheduling Techniques: Different approaches have been proposed to improve different aspects of I/O scheduling: to better incorporate rotational-awareness [28, 29, 44], to better support different storage devices [30, 36], or to provide better QoS guarantees [23, 32, 39]. All these techniques are complementary to our work and can be incorporated into our framework as new schedulers.

9 Conclusion

In this work, we have shown that single-layer schedulers operating at either the block level or system-call level fail to support common goals due to a lack of coordination with other layers. While our experiments indicate that simple layering must be abandoned, we need not sacrifice modularity. In our split framework, the scheduler operates across all layers, but is still abstracted behind a collection of handlers. This approach is relatively clean, and enables pluggable scheduling. Supporting a new scheduling goal simply involves writing a new scheduler plug-in, not re-engineering the entire storage system. Our hope is that split-level scheduling will inspire future vertical integration in storage stacks. Our source code is available at http://research.cs.wisc.edu/adsl/Software/split.

Acknowledgement

We thank the anonymous reviewers and Angela Demke Brown (our shepherd) for their insightful feedback. We thank the members of the ADSL research group for their suggestions and comments on this work at various stages.

This material was supported by funding from NSF grants CNS-1421033, CNS-1319405, and CNS-1218405 as well as generous donations from Cisco, EMC, Facebook, Google, Huawei, IBM, Los Alamos National Laboratory, Microsoft, NetApp, Samsung, Symantec, Seagate, and VMware as part of the WISDOM research institute sponsorship. Tyler Harter is supported by the NSF Fellowship. Samer Al-Kiswany is supported by the NSERC Postdoctoral Fellowship. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and may not reflect the views of NSF or other institutions.

References

[1] CFQ (Complete Fairness Queueing). https://www.kernel.org/doc/Documentation/block/cfq-iosched.txt.

[2] Database/kernel community topic at collaboration summit 2014. http://www.postgresql.org/message-id/20140310101537.GC10663@suse.de.

[3] Deadline IO scheduler tunables. https://www.kernel.org/doc/Documentation/block/deadline-iosched.txt.

[4] Documentation for pgbench. http://http://www.postgresql.org/docs/9.4/static/pgbench.html.

[5] Documentation for /proc/sys/vm/*. https://www.kernel.org/doc/Documentation/sysctl/vm.txt.

[6] Inside the Windows Vista Kernel: Part 1. http://technet.microsoft.com/en-us/magazine/2007.02.vistakernel.aspx.

[7] ionice(1) - Linux man page. http://linux.die.net/man/1/ionice.

[8] Notes on the Generic Block Layer Rewrite in Linux 2.5. https://www.kernel.org/doc/Documentation/block/biodoc.txt.

[9] pgsql-hackers maillist communication. http://www.postgresql.org/message-id/CA+Tgmobv6gm6SzHx8e2w-0180+jHbCNYbAot9KyzG_3DxRYxaw@mail.gmail.com.

[10] Postgresql 9.2.9 documentation. http://www.postgresql.org/docs/9.2/static/wal-configuration.html.

[11] Anand, A., Sen, S., Krioukov, A., Popovici, F., Akella, A., Arpaci-Dusseau, A. C., Arpaci-Dusseau, R. H., and Banerjee, S. Avoiding File System Micromanagement with Range Writes. In *OSDI '08* (San Diego, CA, December 2008).

[12] Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. Information and Control in Gray-Box Systems. In *ACM SIGOPS Operating Systems Review* (2001), vol. 35, ACM, pp. 43–56.

[13] Arpaci-Dusseau, R. H., and Arpaci-Dusseau, A. C. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 2014.

[14] Banga, G., Druschel, P., and Mogul, J. C. Resource containers: A new facility for resource management in server systems. In *OSDI* (1999), vol. 99, pp. 45–58.

[15] Best, S. JFS Overview. http://jfs.sourceforge.net/project/pub/jfs.pdf, 2000.

[16] Bonwick, J., and Moore, B. ZFS: The Last Word in File Systems. http://opensolaris.org/os/community/zfs/docs/zfs_last.pdf, 2007.

[17] Bosch, P., and Mullender, S. Real-time disk scheduling in a mixed-media file system. In *Real-Time Technology and Applications Symposium, 2000. RTAS 2000. Proceedings. Sixth IEEE* (2000), pp. 23–32.

[18] Craciunas, S. S., Kirsch, C. M., and Röck, H. The TAP Project: Traffic Shaping System Calls. http://tap.cs.uni-salzburg.at/downloads.html.

[19] Craciunas, S. S., Kirsch, C. M., and Röck, H. I/O Resource Management Through System Call Scheduling. *SIGOPS Oper. Syst. Rev. 42*, 5 (July 2008), 44–54.

[20] Frost, C., Mammarella, M., Kohler, E., de los Reyes, A., Hovsepian, S., Matsuoka, A., and Zhang, L. Generalized File System Dependencies. In *SOSP '07* (Stevenson, WA, October 2007), pp. 307–320.

[21] Ganger, G. R., McKusick, M. K., Soules, C. A., and Patt, Y. N. Soft Updates: A Solution to the Metadata Update Problem in File Systems. *ACM Transactions on Computer Systems (TOCS) 18*, 2 (2000), 127–153.

[22] Gu, W., Kalbarczyk, Z., Iyer, R. K., and Yang, Z. Characterization of Linux Kernel Behavior Under Errors. In *DSN '03* (San Francisco, CA, June 2003), pp. 459–468.

[23] Gulati, A., Merchant, A., and Varman, P. J. pClock: An Arrival Curve Based Approach for QoS Guarantees in Shared Storage Systems. In *Proceedings of the 2007 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems* (New York, NY, USA, 2007), SIGMETRICS '07, ACM, pp. 13–24.

[24] Gupta, D., Cherkasova, L., Gardner, R., and Vahdat, A. Enforcing performance isolation across virtual machines in xen. In *Middleware 2006*. Springer, 2006, pp. 342–362.

[25] Hagmann, R. Reimplementing the Cedar File System Using Logging and Group Commit. In *SOSP '87* (Austin, TX, November 1987).

[26] Hipp, D. R., and Kennedy, D. SQLite, 2007.

[27] Hofri, M. Disk scheduling: FCFS vs.SSTF revisited. *Communications of the ACM 23*, 11 (1980), 645–653.

[28] Huang, L., and Chiueh, T. Implementation of a Rotation-Latency-Sensitive Disk Scheduler. Tech. Rep. ECSL-TR81, SUNY, Stony Brook, March 2000.

[29] Jacobson, D. M., and Wilkes, J. Disk Scheduling Algorithms Based on Rotational Position. Tech. Rep. HPL-CSP-91-7, Hewlett Packard Laboratories, 1991.

[30] Kim, J., Oh, Y., Kim, E., Choi, J., Lee, D., and Noh, S. H. Disk Schedulers for Solid State Drivers. In *Proceedings of the Seventh ACM International Conference on Embedded Software* (New York, NY, USA, 2009), EMSOFT '09, ACM, pp. 295–304.

[31] Lumb, C., Schindler, J., Ganger, G., Nagle, D., and Riedel, E. Towards Higher Disk Head Utilization: Extracting "Free" Bandwidth From Busy Disk Drives. In *OSDI '00* (San Diego, CA, October 2000), pp. 87–102.

[32] Lumb, C. R., Merchant, A., and Alvarez, G. A. Facade: Virtual storage devices with performance guarantees. In *FAST '03* (San Francisco, CA, April 2003).

[33] Mason, C. The Btrfs Filesystem. oss.oracle.com/projects/btrfs/dist/documentation/btrfs-ukuug.pdf, September 2007.

[34] Mathur, A., Cao, M., Bhattacharya, S., Dilge, A., Tomas, A., and Vivier, L. The New Ext4 Filesystem: Current Status and Future Plans. In *Ottawa Linux Symposium (OLS '07)* (Ottawa, Canada, July 2007).

[35] Mesnier, M., Chen, F., Luo, T., and Akers, J. B. Differentiated Storage Services. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)* (Cascais, Portugal, October 2011).

[36] Park, S., and Shen, K. FIOS: A Fair, Efficient Flash I/O Scheduler. In *FAST* (2012), p. 13.

[37] Pillai, T. S., Chidambaram, V., Alagappan, R., Al-Kiswany, S., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-Consistent Applications. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)* (Broomfield, CO, October 2014).

[38] Popovici, F. I., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. Robust, Portable I/O Scheduling with the Disk Mimic. In *USENIX Annual Technical Conference, General Track* (2003), pp. 297–310.

[39] Povzner, A., Kaldewey, T., Brandt, S., Golding, R., Wong, T. M., and Maltzahn, C. Efficient Guaranteed Disk Request Scheduling with Fahrrad. In *EuroSys '08* (Glasgow, Scotland UK, March 2008).

[40] RISKA, A., LARKBY-LAHET, J., AND RIEDEL, E. Evaluating Block-level Optimization Through the IO Path. In *USENIX '07* (Santa Clara, CA, June 2007).

[41] RIZZO, L., AND CHECCONI, F. GEOM_SCHED: A Framework for Disk Scheduling within GEOM. http://info.iet.unipi.it/~luigi/papers/20090508-geom_sched-slides.pdf.

[42] ROSENBLUM, M., AND OUSTERHOUT, J. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems 10*, 1 (February 1992), 26–52.

[43] RUEMMLER, C., AND WILKES, J. An Introduction to Disk Drive Modeling. *IEEE Computer 27*, 3 (March 1994), 17–28.

[44] SELTZER, M., CHEN, P., AND OUSTERHOUT, J. Disk Scheduling Revisited. In *USENIX Winter '90* (Washington, DC, January 1990), pp. 313–324.

[45] SHENOY, P., AND VIN, H. Cello: A Disk Scheduling Framework for Next-generation Operating Systems. In *SIGMETRICS '98* (Madison, WI, June 1998), pp. 44–55.

[46] SHUE, D., AND FREEDMAN, M. J. From application requests to Virtual IOPs: Provisioned key-value storage with Libra. In *Proceedings of the Ninth European Conference on Computer Systems* (2014), ACM, p. 17.

[47] SWEENEY, A., DOUCETTE, D., HU, W., ANDERSON, C., NISHIMOTO, M., AND PECK, G. Scalability in the XFS File System. In *USENIX 1996* (San Diego, CA, January 1996).

[48] THERESKA, E., BALLANI, H., O'SHEA, G., KARAGIANNIS, T., ROWSTRON, A., TALPEY, T., BLACK, R., AND ZHU, T. IOFlow: A Software-Defined Storage Architecture. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (2013), ACM, pp. 182–196.

[49] VAN METER, R., AND GAO, M. Latency Management in Storage Systems. In *OSDI '00* (San Diego, CA, October 2000), pp. 103–117.

[50] WACHS, M., ABD-EL-MALEK, M., THERESKA, E., AND GANGER, G. R. Argon: Performance insulation for shared storage servers. In *FAST '07* (San Jose, CA, February 2007).

[51] WALDSPURGER, C. A., AND WEIHL, W. E. *Stride Scheduling: Deterministic Proportional Share Resource Management*. Massachusetts Institute of Technology. Laboratory for Computer Science, 1995.

[52] WANG, H., AND VARMAN, P. J. Balancing fairness and efficiency in tiered storage systems with bottleneck-aware allocation. In *FAST '13* (San Jose, CA, February 2014).

[53] WILKES, J., GOLDING, R., STAELIN, C., AND SULLIVAN, T. The HP AutoRAID Hierarchical Storage System. *ACM Transactions on Computer Systems 14*, 1 (February 1996), 108–136.

[54] WORTHINGTON, B. L., GANGER, G. R., AND PATT, Y. N. Scheduling Algorithms for Modern Disk Drives. In *SIGMETRICS '94* (Nashville, TN, May 1994), pp. 241–251.

[55] YANG, J., SAR, C., AND ENGLER, D. EXPLODE: A Lightweight, General System for Finding Serious Storage System Errors. In *OSDI '06* (Seattle, WA, November 2006).

[56] YANG, T., LIU, T., BERGER, E. D., KAPLAN, S. F., AND MOSS, J. E. B. Redline: First Class Support for Interactivity in Commodity Operating Systems. In *OSDI '08* (San Diego, CA, December 2008).

[57] YU, X., GUM, B., CHEN, Y., WANG, R. Y., LI, K., KRISHNAMURTHY, A., AND ANDERSON, T. E. Trading Capacity for Performance in a Disk Array. In *OSDI '00* (San Diego, CA, October 2000).