Bringing Order to Chaos: Barrier-Enabled I/O Stack for Flash Storage

YOUJIP WON and JOONTAEK OH, Hanyang University, Korea
JAEMIN JUNG, Texas A&M University, USA
GYEONGYEOL CHOI and SEONGBAE SON, Hanyang University, Korea
JOOYOUNG HWANG and SANGYEUN CHO, Samsung Electronics, Korea

This work is dedicated to eliminating the overhead required for guaranteeing the *storage order* in the modern IO stack. The existing block device adopts a prohibitively expensive approach in ensuring the storage order among write requests: interleaving the write requests with *Transfer-and-Flush*. For exploiting the cache barrier command for flash storage, we overhaul the IO scheduler, the dispatch module, and the filesystem so that these layers are orchestrated to preserve the ordering condition imposed by the application with which the associated data blocks are made durable. The key ingredients of Barrier-Enabled IO stack are *Epoch-based IO scheduling*, *Order-Preserving Dispatch*, and *Dual-Mode Journaling*. Barrier-enabled IO stack can control the storage order without Transfer-and-Flush overhead. We implement the barrier-enabled IO stack in server as well as in mobile platforms. SQLite performance increases by 270% and 75%, in server and in smartphone, respectively. In a server storage, BarrierFS brings as much as by 43× and by 73× performance gain in MySQL and SQLite, respectively, against EXT4 via relaxing the durability of a transaction.

CCS Concepts: • **Software and its engineering → File systems management**;

Additional Key Words and Phrases: Filesystem, storage, block device, linux

ACM Reference format:
Youjip Won, Joontaek Oh, Jaemin Jung, Gyeongyeol Choi, Seongbae Son, Jooyoung Hwang, and Sangyeun Cho. 2018. Bringing Order to Chaos: Barrier-Enabled I/O Stack for Flash Storage. *ACM Trans. Storage* 14, 3, Article 24 (October 2018), 29 pages.
https://doi.org/10.1145/3242091

1 MOTIVATION

The modern Linux IO stack is a collection of the arbitration layers; the IO scheduler, the command queue manager, and the writeback cache manager shuffle the incoming requests at their own disposal before passing them to the next layers. Despite the compound uncertainties from the multiple

This work was done while Jaemin Jung was a graduate student at Hanyang University.
This work is funded by Basic Research Lab Program (NRF, No. 2017R1A4A1015498), the BK21 plus (NRF), ICT R&D program (IITP, R7117-16-0232) and Future OS project (IITP, No. 2018-0-00549).
Authors' addresses: Y. Won, J. Oh (Corresponding author), G. Choi, and S. Son, Hanyang University, Seoul, Korea; emails: {yjwon, na94jun, chl4651, afireguy}@hanyang.ac.kr; J. Jung, Texas A&M University, College Station, TX, USA; email: jmjung@tamu.edu; J. Hwang and J. S. Cho, Samsung Electronics, Suwon, Korea; emails: {jooyoung.hwang, sangyeun.cho}@samsung.com.

1553-3077/2018/10-ART24 $15.00
https://doi.org/10.1145/3242091

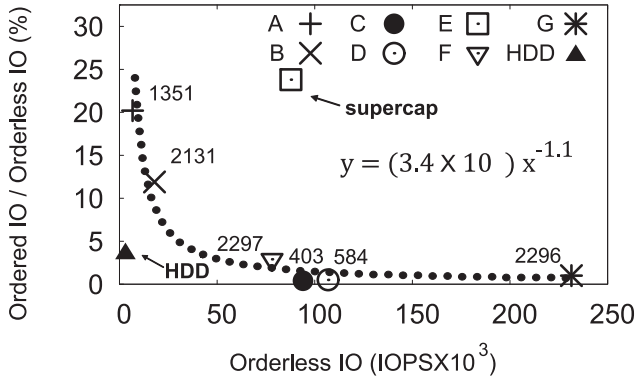


Fig. 1. Ordered write vs. Orderless write, Except "HDD," all are flash storages; A: (1ch)/eMMC5.0, B: (1ch)/UFS2.0, C: (8ch)/SATA3.0, D: (8ch)/NVMe, E: (8ch)/SATA3.0 (supercap), F: (8ch)/PCIe, G: (32ch) Flash array, The number next to each point is the IOPS of `write()` followed by `fdatasync()`.

layers of arbitration, it is essential for the software writers to enforce a certain order in which the data blocks are reflected to the storage surface, *storage order*, in many cases such as in guaranteeing the durability and the atomicity of a database transaction [26, 35, 47], in filesystem journaling [2, 41, 66, 69], in soft-update [42, 63], or in copy-on-write or log-structure filesystems [31, 35, 60, 61]. Enforcing a storage order is being achieved by an extremely expensive approach: Dispatching the following request only after the data block associated with the preceding request is completely transferred to the storage device and is made durable. We call this mechanism a *Transfer-and-Flush*. For decades, interleaving the write requests with a *Transfer-and-Flush* has been the fundamental principle to guarantee the storage order in a set of requests [14, 23].

We observe a phenomenal increase in the performance and the capacity of the flash storage. The performance increase owes much to the concurrency and the parallelism in the Flash storage, e.g., the multi-channel/way controller [4, 75], the large size storage cache [48], and the deep command queue [17, 27, 74]. A state-of-the-art NVMe SSD reportedly exhibits up to 750 KIOPS random read performance [74]. It is nearly 4,000× of a HDD's performance. The capacity increase is due to the adoption of the finer manufacturing process (sub-10 nm) [24, 36], and the multi-bits per cell (MLC, TLC, and QLC) [3, 9]. Despite these splendid advancements, the time to program a Flash cell has barely improved and is even deteriorating in some cases [21]. As the Flash device is manufactured with a finer manufacturing process or as a Flash cell needs to represent the larger number of bits, the Flash controller has to spend more time to detect the minor variance in the voltage level of the cell or to program a Flash cell to a desired voltage level.

The Transfer-and-Flush-based order-preserving mechanism does not align well with the parallelism and the concurrency in the modern flash storage. The Transfer-and-Flush-based order-preserving mechanism neutralizes the parallelism and the concurrency feature of the flash storage. It also exposes the raw Flash cell programming latency to the host. The overhead of the Transfer-and-Flush mechanism will become more significant as the flash storage employs a higher degree of parallelism and the denser Flash device. Figure 1 illustrates an important trend. We measure the sustained throughput of orderless random write (plain buffered write) and the ordered random write in EXT4 filesystem. To enforce the order among the writes, each write request is followed by `fdatasync()` in the ordered random write workload. The X-axis denotes the throughput of orderless write that corresponds to the rate at which the storage device services the write requests at its full throttle. This usually matches the vendor published performance of the storage device.

The number next to each point denotes the sustained throughput of the ordered write. The Y-axis denotes the ratio between the two. In a single-channel mobile storage for smartphone (SSD A), the performance of ordered write is 20% of that of unordered write (1351 IOPS vs. 7000 IOPS). In a 32 channel Flash array (SSD G), this ratio decreases to 1% (2296 IOPS vs. 230K IOPS). In SSD with supercap (SSD E), the ordered write performance is 25% of that of the unordered write. the flash storage uses supercap to hide the flush latency from the host. Even in a flash storage with supercap, the overhead of Transfer-and-Flush is significant.

Many researchers have attempted to address the overhead of storage order guarantee. The techniques deployed in the production platforms include non-volatile writeback cache at the flash storage [22], `no-barrier` mount option at the EXT4 filesystem [13], and transactional checksum [32, 56, 64]. Efforts such as transactional filesystem [16, 35, 50, 54, 70] and transactional block device [30, 43, 52, 72, 76] save the application from the overhead of enforcing the storage order associated with filesystem journaling. A school of works addresses more fundamental aspects in controlling the storage order, such as separating the ordering guarantee from durability guarantee [7], providing a programming model to define the ordering dependency among the set of writes [19], and persisting a data block only when the result needs to be externally visible [49]. Despite their elegance, these works rely on Transfer-and-Flush when it is required to enforce the storage order. OptFS [7] relies on Transfer-and-Flush in enforcing the order between the journal commit and the associated checkpoint. Featherstitch [19] relies on Transfer-and-Flush to implement the ordering dependency between the *patchgroups*.

In this work, we revisit the issue of eliminating the Transfer-and-Flush overhead in the modern IO stack. We develop a *Barrier-Enabled IO stack*, in which the filesystem can issue the following request before the preceding request is serviced, and yet the IO stack can enforce the storage order between them. The barrier-enabled IO stack consists of the cache barrier-aware storage device, the order-preserving block device layer, and the barrier enabled filesystem. For cache barrier-aware storage device, we exploit the "cache barrier" command [28]. The barrier-enabled IO stack is built on the foundation that the host can control a certain partial order in which the cache contents are flushed. The "cache barrier" command precisely serves this purpose. For the order-preserving block device layer, the command dispatch mechanism and the IO scheduler are overhauled so that the block device layer ensures that the IO requests from the filesystem are serviced preserving a certain partial order. For the barrier-enabled filesystem, we define new interfaces, `fbarrier()` and `fdatabarrier()`, to separate the ordering guarantee from the durability guarantee. They are similar to `fsync()` and `fdatasync()`, respectively, except that they return without waiting for the associated blocks to become durable. We modify EXT4 for the order-preserving block device layer. We develop dual-mode journaling for the order-preserving block device. Based on the dual-mode journaling, we newly implement `fbarrier()` and `fdatabarrier()` and rewrite `fsync()`.

Barrier-enabled IO stack removes the flush overhead as well as the transfer overhead in enforcing the storage order. While large body of the works have focused on eliminating the flush overhead, few works have addressed the overhead of *DMA transfer* to enforce the storage order. The benefits of the barrier-enabled IO stack include the following;

- The application can control the storage order virtually without any overheads, including the flush overhead, DMA transfer overhead, and context switch.
- The latency of a journal commit decreases significantly. The journaling module can enforce the storage order between the journal logs and the journal commit block without interleaving them with flush or with DMA transfer.
- Throughput of the filesystem journaling improves significantly. The dual-mode journaling commits the multiple transactions concurrently and yet can guarantee the durability of the individual journal commits.

By eliminating the Transfer-and-Flush overhead, the barrier-enabled IO stack successfully exploits the concurrency and the parallelism in modern flash storage. Relaxing the durability of a transaction, SQLite performance and MySQL performance increase as much as by 73× and by 43×, respectively, in server storage.

The rest of the article is organized as follows. Section 2 explains the background. Section 3 explains the order-preserving block device layer. Section 4 describes the BarrierFS. BarrierFS is the variant of EXT4 that is tailored for order-preserving block device. Section 5 describes the implication of barrier-enabled IO stack over the applications. Section 6 presents the result of the experiment. Section 7 summarizes the preceding works. Section 8 concludes the article.

2 BACKGROUND

2.1 Orders in the IO Stack

A write request travels a complicated route until the data blocks reach the storage surface. The filesystem puts the request to the IO scheduler queue. The block device driver removes one or more requests from the queue and constructs a command. It probes the device and dispatches the command if the device is available. The device is available if the command queue is not full. The storage controller inserts the incoming command at the command queue. The storage controller removes the command from the command queue and services it (i.e., transfers the associated data block between the host and the storage). When the transfer finishes, the device signals the host. The contents of the writeback cache are committed to the storage surface either periodically or by an explicit request from the host.

We define four types of orders in the IO stack; *Issue Order*, $\mathcal{I}$; *Dispatch Order*, $\mathcal{D}$; *Transfer Order*, $\mathcal{X}$; and *Persist Order*, $\mathcal{P}$. The issue order $\mathcal{I} = \{i_1, i_2, \ldots, i_n\}$ is a set of write requests issued by the file system. The subscript denotes the order in which the requests enter the IO scheduler. The dispatch order $\mathcal{D} = \{d_1, d_2, \ldots, d_n\}$ denotes a set of the write requests dispatched to the storage device. The subscript denotes the order in which the requests leave the IO scheduler. The transfer order, $\mathcal{X} = \{x_1, x_2, \ldots, x_n\}$, is the set of transfer completions. The persist order, $\mathcal{P} = \{p_1, p_2, \ldots, p_n\}$, is a set of operations that make the data blocks in the writeback cache durable. We say that a partial order is preserved if the relative position of the requests against a designated request, *barrier*, are preserved between two different types of orders. We use the notation "=" to denote that a partial order is preserved. The partial orders between the different types of orders may not coincide due to the following reasons.

- $\mathcal{I} \neq \mathcal{D}$. The order in which a filesystem issues are set of requests is not aligned with the order in which they are dipatched to the storage device. The IO scheduler reorders and coalesces the IO requests subject to the scheduling principle, e.g., CFQ, DEADLINE, and so on. Some scheduler services the incoming requests in FIFO manner, e.g., NO-OP scheduler [1]. Some of the modern IO subsystem such as NVMe [11] interface eliminates the scheduling layer to minimize the unnecessary processing overhead.
- $\mathcal{D} \neq \mathcal{X}$. The order in which the commands are dispatched to the storage device is not aligned with the order in which the storage controller services them. The storage controller removes the command from the head of the command queue and services it. However, the storage controller can freely insert a command anywhere at the command queue subject to the priority of the command. The details of the command insertion algorithm is vendor specific and is unknown to the public. In addition, the commands can be serviced in out-of-order manner due to the errors, the time-outs, and the retry.
- $\mathcal{X} \neq \mathcal{P}$. The order in which the data blocks in the storage's writeback cache reach the storage service is different from the order in which the data blocks are transferred. There are

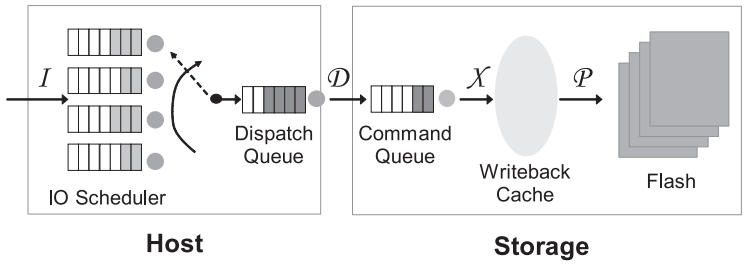


Fig. 2. IO stack organization.

a number of reasons for this discordance. The writeback cache of the storage is not FIFO. The victim selection algorithm of the writeback cache manager controls the persist order. The physical nature of the Flash device adds another dimension of uncertainty in persist order. In flash storage, the order in which the Flash pages are programmed and the order in which the associated mapping table entries are updated may not coincide. A persist order is governed not by the order in which the data blocks are programmed at the destination Flash pages but by the order in which the associated mapping table entries are updated.

Due to all these sources of uncertainties, the modern IO stack is said to be *orderless* [6].

2.2 Transfer-and-Flush

Enforcing a storage order corresponds to preserving a partial order between the order in which the filesystem issues the requests, $\mathcal{I}$, and the order in which the associated data blocks are made durable, $\mathcal{P}$. It is equivalent to collectively enforcing the partial orders between the pair of the orders in the adjacent layers in Figure 2. It can be formally represented as in Equation (1),

$$(\mathcal{I} = \mathcal{P}) \equiv (\mathcal{I} = \mathcal{D}) \wedge (\mathcal{D} = \mathcal{X}) \wedge (\mathcal{X} = \mathcal{P}). \tag{1}$$

The modern IO stack has evolved under the assumption that the host cannot control the persist order, i.e., $\mathcal{X} \neq \mathcal{P}$. The constraint that the host cannot control the persist order is a fundamental limitation in modern IO stack design. Due to the constraint of $\mathcal{X} \neq \mathcal{P}$, Equation (1) is unsatisfiable. This assumption stems from the physical characteristics of the rotating media. For rotating media such as HDDs, a persist order is governed by disk scheduling algorithm. The disk scheduling is entirely left to the storage controller due to its complicated sector geometry that is hidden from outside [20]. When the host blindly enforces a certain persist order in which a set of data blocks in the writeback cache are persisted, the host may experience anomalous delay in IO service.

The block device layer adopts the indirect and the expensive approach to control the storage order in spite of the constraint $\mathcal{X} \neq \mathcal{P}$. First, after dispatching the write command to the storage device, the caller postpones dispatching the following command until the preceding command is serviced, i.e., until the associated DMA transfer completes. We refer to this mechanism as *Wait-on-Transfer*. Wait-on-Transfer mechanism ensures that the commands are serviced in order and to satisfy $\mathcal{D} = \mathcal{X}$. Wait-on-Transfer is expensive; it blocks the caller and interleaves the preceding request and the following request with the latency of DMA transfer. Second, when the caller is notified that the preceding command has been completed, the caller issues the flush command. The caller issues the following command only after the flush command returns. This is to ensure that the associated data blocks are persisted in order. This is to satisfy $\mathcal{X} = \mathcal{P}$. We refer to this mechanism as *Wait-on-Flush*. The modern block device layer uses Wait-on-Transfer and Wait-on-Flush in pair when it needs to enforce the storage order between the write requests. We call this mechanism as *Transfer-and-Flush*.

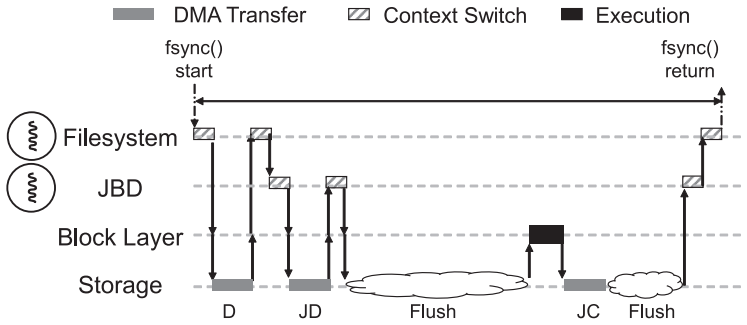


Fig. 3. DMA, flush, and context switches in `fsync()`, "D," "JC," and "JC" denote the DMA transfer time for *D*, *JD* and *JC*, respectively. "Flush" denotes the time to service the flush request.

The cost of Transfer-and-Flush is prohibitive. It disarms the internal parallelism of the flash storage controller, stalls the command queue, and exposes the caller to DMA transfer and raw cell programming delays.

2.3 Analysis: `fsync()` in EXT4

We examine how the EXT4 filesystem controls the storage order in an `fsync()`. In Ordered journaling mode (default), the journaling module guarantees that the data blocks are persisted before the journal transaction does. `fsync()` accounts for dominant fraction of IO's in popular workloads, e.g., OLTP [34], smartphone [26, 65], or mail server [73].

Figure 3 illustrates the behavior of an `fsync()`. The filesystem issues the write requests for a set of dirty pages, *D*. *D* may consist of the data blocks from different files. After issuing the write requests, the application thread blocks waiting for the completion of the associated DMA transfer. When the DMA transfer is completed, the application thread resumes and triggers the JBD thread to commit the journal transaction. After triggering the JBD thread, the application thread sleeps again. When the JBD thread makes journal transaction durable, the `fsync()` returns. It should be emphasized that the application thread triggers the JBD thread only after *D* is transferred. Otherwise, the storage controller may service the write request for *D* and the write requests for journal commit in an out-of-order manner, and the journal transaction may become durable prematurely before *D* is transferred.

A journal transaction is usually committed using two write requests: one for writing the coalesced chunk of the journal descriptor block and the log blocks and the other for writing the commit block. In the rest of the article, we will use $\mathcal{JD}$ and $\mathcal{JC}$ to denote the coalesced chunk of the journal descriptor block and the log blocks, and the commit block, respectively. In committing a journal transaction, JBD needs to enforce the storage orders in two relations: within a transaction and between the transactions. Within a transaction, JBD needs to ensure that *JD* is made durable ahead of *JC*. Between the journal transactions, JBD has to ensure that journal transactions are made durable in order. When either of the two conditions are violated, the file system may recover incorrectly in case of unexpected failure [7, 69]. For the storage order within a transaction, JBD interleaves the write request for *JD* and the write request for *JC* with Transfer-and-Flush. In EXT4, the transactions are guaranteed to be made durable in order, since the filesystem journaling is serial operation; the filesystem commits the following transaction only after the preceding transaction becomes durable. JBD uses Transfer-and-Flush mechanism in enforcing both intra-transaction and inter-transaction storage order.

An `fsync()` can be represented as in Equation (2). D, JD and JC denote the write request for D, JD and JC, respectively. "xfer" and "flush" denote Wait-on-Transfer and Wait-on-Flush, respectively. The host issues a write request for D and waits for DMA transfer completion. Once the DMA completes, it dispatches the write request for JD and waits for DMA transfer completion again. When the DMA completes, it sends the flush command. When the flush command returns, the host issues the write request for JC and waits for DMA completion. When the DMA transfer completes, the host issues a flush command to the storage device. In Equation (2), the first flush is to control the storage order between $\{D, JD\}$ and JC. The second flush is to control the storage order among the transactions,

$$D \rightarrow \text{xfer} \rightarrow JD \rightarrow \text{xfer} \rightarrow \underbrace{\text{flush} \rightarrow JC \rightarrow \text{xfer} \rightarrow \text{flush}}_{\texttt{REQ\_FLUSH|REQ\_FUA}}. \tag{2}$$

In earlier days of Linux, the block device layer explicitly issued a flush command in committing a journal transaction (Figure 3) [13]. The flush command blocks not only the caller but also the other requests in the same dispatch queue, since the block layer has to wait until the flush command returns. Since Linux 2.6.37, the filesystem (JBD) resorts to implicitly issues a flush command [14]. The kernel community defines new flags, `REQ_FUA` and `REQ_FLUSH`. In writing JC, JBD tags the write request for JC with `REQ_FLUSH` and `REQ_FUA` flags. When the write request has the `REQ_FLUSH` flag set, the storage controller flushes the writeback cache before servicing the command. When the write request has `REQ_FUA` flag set, the storage controller persists the associated data block directly to storage surface bypassing the writeback cache. The reasonably capable device supports these flags. The block layer can dispatch the following write command without waiting for the completion of the preceding write command. When the filesystem, i.e., the JBD thread, uses `REQ_FLUSH` and `REQ_FUA` flags in writing JC, only the JBD thread blocks to make JC durable and the other threads that share the dispatch queue can proceed. When the filesystem uses `REQ_FLUSH` and `REQ_FUA` flags in writing JC, the `fsync()` can be represented as in Equation (3). The last four terms in Equation (2) is replaced with a single write command with `REQ_FLUSH|REQ_FUA` flag in Equation (3),

$$D \rightarrow \text{xfer} \rightarrow JD \rightarrow \text{xfer} \rightarrow JC_{\texttt{REQ\_FLUSH|REQ\_FUA}}. \tag{3}$$

The fundamental principle that drives the evolution from Equation (2) to Equation (3) is simple: to make the storage device more capable. In Equation (3), the storage device is made to handle `REQ_FUA` flag and `REQ_FLUSH` flag. Exploiting this feature, the host side IO stack is relieved from the burden of explicitly issuing the flush command. Our effort can be thought as a continuation to this evolutionary path of the modern IO stack. We make the storage device more capable; we make the storage device to support cache barrier command. Then, we offload the responsibility of enforcing the persist order to the storage device. We mitigate the Transfer-and-Flush overhead by redesigning the host side IO stack exploiting the barrier feature at the storage device.

3 ORDER-PRESERVING BLOCK DEVICE LAYER

3.1 Design

The order-preserving block device layer consists of the newly defined barrier write command, the order-preserving dispatch module, and the Epoch-based IO scheduler. We overhaul the IO scheduler, the dispatch module, and the write command so that they can preserve the partial order between the different types of orders, $\mathcal{I} = \mathcal{D}$, $\mathcal{D} = \mathcal{X}$, and $\mathcal{X} = \mathcal{P}$, respectively. Order-preserving dispatch module eliminates the Wait-on-Transfer overhead and the newly defined barrier write command eliminates the Wait-on-Flush overhead. They collectively together preserve the partial order between the issue order $\mathcal{I}$ and the persist order $\mathcal{P}$ without Transfer-and-Flush.

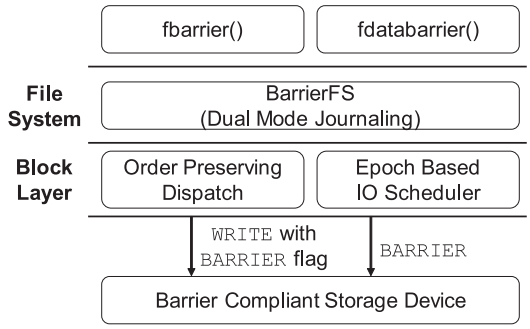


Fig. 4. Organization of the barrier-enabled IO stack.

The order-preserving block device layer categorizes the write requests into two types, *orderless* write and *order-preserving* write. The order-preserving requests are the ones that are subject to the storage ordering constraint. The orderless requests are the ones that are irrelevant to the ordering constraint and that can be scheduled freely. The order preserving block device layer separates the two to avoid imposing unnecessary ordering constraint in scheduling the requests. Let us provide an example. When the JBD thread commits a journal transaction, the block device layer needs to enforce the storage order between the journal logs and the journal commit block. When the `pdflush` flushes the dirty page cache entries, the block layer does not have to enforce the storage order among the write requests triggered by `pdflush`. The write requests for committing a filesystem journal transaction and the write requests triggered by `pdflush` can be interleaved with each other entering the block device layer. The proposed order-preserving block device layer imposes the ordering constraint only to the write requests for committing a journal transaction while leaving the requests originated by `pdflush` freely scheduled.

We refer to a set of the order-preserving requests that can be reordered with each other as an *epoch* [12]. We define a special type of order-preserving write as a *barrier* write. A barrier write is used to delimit an epoch. We introduce two new attributes, `REQ_ORDERED` and `REQ_BARRIER`, for a write request. For an order-preserving write, the associated `bio` object and the associated `request` object have `REQ_ORDERED` attribute. For a barrier write request, the associated `bio` object and the associated `request` object have both `REQ_ORDERED` attribute and `REQ_BARRIER` attribute. The IO scheduler and the dispatch module of the order-preserving block device layer handle the order-preserving request and the orderless request differently. The details are to be explained shortly. Figure 4 illustrates the organization of Barrier-Enabled IO stack.

3.2 `Barrier Write`, the Command

The "cache barrier," or "barrier" for short, command is defined in the standard command set for mobile flash storage [28]. With a barrier command, the host can control the persist order without explicitly invoking the cache flush. When the storage controller receives the barrier command, the controller guarantees that the data blocks transferred before the barrier command becomes durable after the ones that follow the barrier command do. A few eMMC products in the market support cache barrier command [25, 67]. The barrier command is the boon to the modern IO stack. The barrier command satisfies the condition $\mathcal{X} = \mathcal{P}$ in Equation (1), which has been unsatisfiable for the several decades due to the mechanical characteristics of the rotating media. The naive way of using the barrier command is to replace the existing flush operation [68]. This simple replacement saves the caller from the latency of the flush operation in the Transfer-and-Flush overhead. However, the caller is still subject to Wait-on-Transfer overhead to enforce the storage order.

Implementing a barrier as a separate command occupies one entry in the command queue and costs the host the latency of dispatching a command. To avoid this overhead, we define a barrier as a command flag. We designate one unused bit in the SCSI command for a barrier flag. We set the barrier flag of the write command to make itself a barrier write. When the storage controller receives a barrier write command, it services the barrier write command as if the barrier command has arrived immediately following the associated write command. With reasonable complexity, flash storage can be made to support a barrier write command [30, 39, 57]. When flash storage has Power Loss Protection (PLP) feature, e.g., a supercapacitor or non-volatile RAM for writeback cache, the writeback cache contents are guaranteed to be durable. The storage controller can flush the writeback cache fully utilizing its parallelism and yet can guarantee the persist order. In flash storage with PLP, we expect that the performance overhead of the barrier write is insignificant.

For the devices without PLP, the barrier write command can be supported in three ways: in-order writeback, transactional writeback, or in-order recovery. In in-order writeback, the storage controller flushes the data blocks in epoch granularity. The amount of data blocks in an epoch may not be large enough to fully utilize the parallelism of the Flash storage. The in-order writeback style of the barrier write implementation can bring the performance degradation in cache flush. In transactional writeback, the storage controller flushes the writeback cache contents as a single unit [39, 57]. Since all epochs in the writeback cache are flushed together, the persist order imposed by the barrier command is satisfied. The transactional writeback can be implemented without any performance overhead if the controller exploits the spare area of the Flash page to represent a set of pages in a transaction [57]. The in-order recovery method relies on a crash recovery routine to control the persist order. High-performance SSD is loaded with the multiple controller cores. Each of these cores may be assigned an exclusive set of Flash dies to handle and the multiple controller cores can concurrently write the data blocks across the multiple Flash packages. For multi-core multi-channel high performance SSD, one may have to use sophisticated crash recovery protocol such as ARIES [46] to recover the storage to consistent state in case of unexpected system failure. For embedded flash storage such as flash storage for smartphones, the flash storage has only single core with one or two channels. In this storage, the controller can treat the entire flash storage as a single log device. The Flash controller can use simple crash recovery algorithm used in LFS [61]. Since the persist order is enforced when the system recovers from the crash, the storage controller can flush the writeback cache fully utilizing its internal parallelism such as multiple channels and multiple ways as if there is no ordering dependency. The controller is saved from performance penalty at the cost of complexity in the recovery routine.

In this work, we modify the firmware of the UFS storage device to support the barrier write command. We adopt a simple LFS style in-order recovery scheme. The modified firmware is loaded at the commercial UFS product of the Galaxy S6 smartphone.[1] The modified firmware treats the entire storage device as a single log structured device. The FTL of the UFS storage maintains an active segment in memory. FTL appends incoming data blocks to the active segment in the order in which they are transferred. When an active segment becomes full, the controller stripes the active segment across the multiple Flash chips in log-structured manner. Here, the persist order is preserved. The recovery routine works as follows. When the system crashes, the UFS controller locates the beginning of the most recently flushed segment in the recovery phase. It scans the pages in the most recently flushed segment from the beginning until it encounters the page that has not been programmed successfully. The storage controller discards the rest of the pages in the segment including the incomplete one.

[1]Some of the authors are firmware engineers at Samsung Electronics and have an access to the FTL firmware of flash storage products.

Developing a barrier-enabled SSD controller is an engineering exercise. It is governed by a number of design choices and should be addressed in a separate context. In this work, we demonstrate that the performance benefit achieved by the barrier command well deserves its complexity if the host side IO stack can properly exploit it.

3.3 Order-Preserving Dispatch

Order-preserving dispatch is a fundamental innovation in this work. In order-preserving dispatch, the block device layer dispatches the following command immediately after it dispatches the preceding one (Figure 6). The host dispatches the following command without waiting for the completion of the preceding command and yet the order-preserving dispatch mechanism enables the host to ensure that the two commands are serviced in order. We refer to this mechanism as *Wait-on-Dispatch*. The order-preserving dispatch is to satisfy the condition $\mathcal{D} = \mathcal{X}$ in Equation (1) without Wait-on-Transfer overhead.

The dispatch module constructs a command from one or more requests. The dispatch module treats the order-preserving write and the orderless write differently. The order-preserving dispatch module constructs the barrier write command when it encounters the barrier write request, i.e., the write request with barrier bit set. For the other requests, the order-preserving dispatch module constructs the command as the legacy dispatch module used to do.

Despite the profound implication of order-preserving dispatch, implementing an order-preserving dispatch is simple and straightforward: The block device driver sets the priority of a barrier write command as *ordered*. Then, the SCSI compliant storage device services the command satisfying the ordering constraint. Simply by setting the priority of a barrier write command to *ordered*, the dispatch module naturally becomes an order-preserving one. The following is the reason behind the scene. SCSI standard defines three command priority levels: *head of the queue*, *ordered*, and *simple* [59]. For the *head of the queue* priority command, the storage controller places it at the head of the command queue. For the command with *ordered* priority, the storage controller places it at the tail of the command queue. For the command with *simple* priority, the controller determines the position of the command at its disposal. The command with *simple* priority cannot be inserted in front of the existing *ordered* or *head of the queue* command. The default priority is *simple*. Exploiting the command priority of existing SCSI interface, the order-preserving dispatch module ensures that the barrier write is serviced only after the existing requests in the command queue are serviced and before any of the commands that follow the barrier write are serviced.

The storage device can temporarily be unavailable or the caller can be switched out involuntarily after dispatching a write request. Then, the dispatch module needs to retry the command. The order-preserving dispatch module uses the same error handling routine of the existing block device driver; the kernel daemon inherits the task and retries the failed command after a certain time interval, e.g., 3ms for SCSI devices [59]. Figure 5 illustrates the behavior of the order-preserving block device layer when the underlying device is temporarily unavailable.

The *ordered* priority command has rarely been used in the existing block device implementations. When the host cannot control the persist order, *ordered* priority barely carries any meaning. This is because the partial order in which the commands are serviced is not aligned with the partial order in which the writeback cache contents are flushed. In the emergence of the barrier write, the ordered priority plays a vital role in making the entire IO stack an order-preserving one.

The importance of order-preserving dispatch cannot be emphasized further. With order-preserving dispatch, the host can control the transfer order without releasing the CPU and without stalling the command queue. IO latency can become more predictable, since there exists less chance that the CPU scheduler interferes with the caller's execution. Δ_{WoT} and Δ_{WoD} in Figure 6 illustrate the delays between the consecutive requests in Wait-on-Transfer and Wait-on-Dispatch,

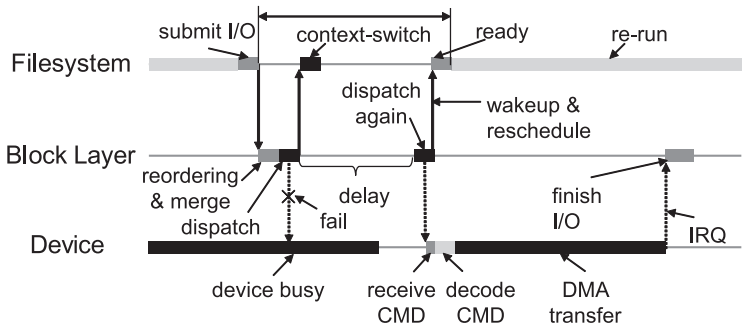


Fig. 5. Wait-On-Dispatch when the device is Busy.

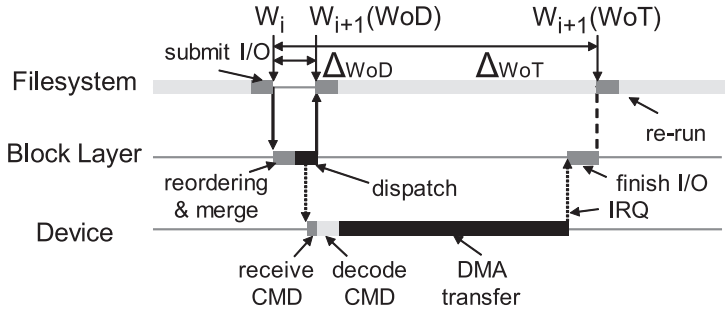


Fig. 6. Wait-on-Dispatch vs. Wait-on-Transfer, W_i: ith write, W_{i+1}(WoD): $(i+1)$th write under Wait-on-Dispatch, W_{i+1}(WoT): $(i+1)$th write under Wait-on-Transfer.

respectively. In Wait-on-Dispatch, the host issues the next request $W_{i+1}(WoD)$ immediately after it issues W_i. In Wait-on-Transfer, the host issues the next request $W_{i+1}(WoT)$ only after W_i is serviced. Δ_{WoD} is an order of magnitude smaller than Δ_{WoT}.

3.4 Epoch-Based IO Scheduling

Epoch-based IO scheduling is designed to preserve the partial order between the issue order and the dispatch order. It satisfies the condition $\mathcal{I} = \mathcal{D}$. It is designed with three principles: (i) It preserves the partial order between the epochs, (ii) the requests within an epoch can be freely scheduled with each other, and (iii) an orderless request can be scheduled across the epochs.

When an IO request enters the scheduler queue, the IO scheduler determines if it is a barrier write. If the request is a barrier write, then the IO scheduler removes the barrier flag from the request and inserts it into the queue. Otherwise, the scheduler inserts it to the queue as is. When the scheduler inserts a barrier write to the queue, it stops accepting more requests. Since the scheduler blocks the queue after it inserts the barrier write, the requests in the queue are either orderless requests or the order-preserving requests belonging to the same epoch. The requests in the queue can be freely re-ordered and merged with each other without compromising the ordering constraint. The order-preserving IO scheduler is built on top of the existing scheduling discipline, e.g., CFQ. The merged request will be order-preserving if one of the components is an order-preserving request. The IO scheduler designates the last order-preserving request that leaves the queue as a new barrier write. This mechanism is called *Epoch-Based Barrier Reassignment*. When there are no order-preserving requests in the queue, the IO scheduler starts accepting the IO requests again.

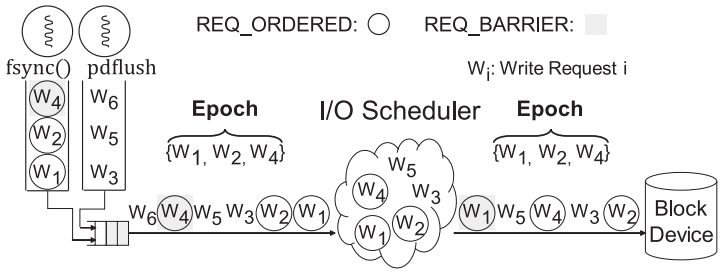


Fig. 7. Epoch-based barrier reassignment.

When the IO scheduler unblocks the queue, there can be one or more orderless requests in the queue. These orderless requests are scheduled with the requests in the following epoch. Differentiating orderless requests from the order-preserving ones, we avoid imposing unnecessary ordering constraint on the irrelevant requests.

Figure 7 illustrates an example. The circle and the rectangle that enclose the write request denote the order-preserving flag and barrier flag, respectively. An `fdatasync()` creates three write requests: w_1, w_2, and w_4. The barrier-enabled filesystem, which will be detailed shortly, marks the write requests as ordering preserving ones. The last request, w_4, is designated as a barrier write and an epoch, $\{w_1, w_2, w_4\}$, is established. A `pdflush` creates three write requests, w_3, w_5, and w_6. They are all orderless. The requests from the two threads are fed to the IO scheduler as $w_1, w_2, w_3, w_5, w_4^{barrier}, w_6$. When the barrier write, w_4, enters the queue, the scheduler blocks the queue. Thus, w_6 cannot enter the queue. The IO scheduler reorders the requests in the queue and dispatches them as $w_2, w_3, w_4, w_5, w_1^{barrier}$ order. The IO scheduler relocates the barrier flag from w_4 to w_1. The epoch is preserved after IO scheduling.

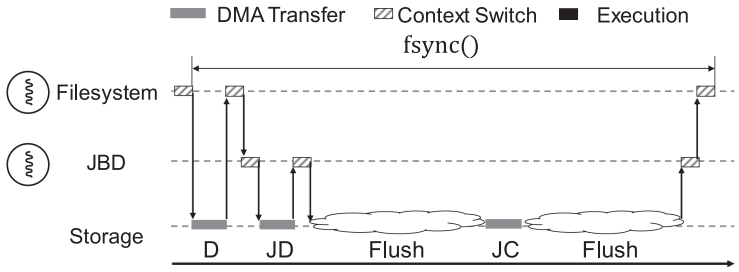
The order-preserving block device layer now satisfies all three conditions, $\mathcal{I} = \mathcal{D}$, $\mathcal{D} = \mathcal{X}$, and $\mathcal{X} = \mathcal{P}$ in Equation (1) with an Epoch-based IO scheduling, an order-preserving dispatch, and a barrier write, respectively. The order-preserving block device layer successfully eliminates the Transfer-and-Flush overhead in controlling the storage order and can control the storage order with only Wait-on-Dispatch overhead.

4 BARRIER-ENABLED FILESYSTEM

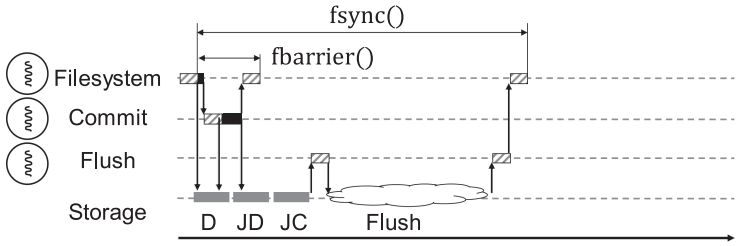
4.1 Programming Model

The barrier-enabled IO stack offers four synchronization primitives: `fsync()`, `fdatasync()`, `fbarrier()`, and `fdatabarrier()`. We propose two new filesystem interfaces, `fbarrier()` and `fdatabarrier()`, to separately support ordering guarantee. `fbarrier()` and `fdatabarrier()` synchronize the same set of blocks with `fsync()` and `fdatasync()`, respectively, but they return without ensuring that the associated blocks become durable. `fbarrier()` bears the same semantics as `osync()` in OptFS [7] in that it writes the data blocks and the journal transactions in order but returns without ensuring that they become durable.

`fdatabarrier()` synchronizes the same set of blocks as `fdatasync()`. Unlike `fdatasync()`, `fdatabarrier()` returns without persisting the associated blocks. `fdatabarrier()` is a generic storage barrier. By interleaving the `write()` calls with `fdatabarrier()`, the application ensures that the data blocks associated with the write requests that precede `fdatabarrier()` are made durable ahead of the data blocks associated with the write requests that follow `fdatabarrier()`. It plays the same role as `mfence` for memory barrier [53]. Refer to the following codelet. Using `fdatabarrier()`, the application ensures that the "world" is made durable only after "Hello" does,



(a) `fsync()` in EXT4; JBD writes JC with `FLUSH/FUA`. The latter of the two 'Flush's persists 'JC' directly to the storage surface.



(b) `fsync()` and `fbarrier()` in BarrierFS

Fig. 8. Details of `fsync()` and `fbarrier()`.

```
write(fileA, "Hello") ;
fdatabarrier(fileA) ;
write(fileA, "World") ;
```

The order-preserving block device layer is filesystem agnostic. In our work, we modify EXT4 for barrier enabled IO stack and call it as BarrierFS.

4.2 Dual Mode Journaling

Committing a journal transaction essentially consists of two separate tasks: (i) dispatching the write commands for *JD* and *JC* and (ii) making *JD* and *JC* durable. Exploiting the order-preserving nature of the underlying block device, we physically separate the control plane activity (dispatching the write commands) and the data plane activity (persisting the associated data blocks and the journal transaction) of a journal commit operation.

We allocate the separate threads for control plane and for data plane so that the two activities can proceed in parallel with minimum dependency. The two threads are called the *commit* thread and *flush* thread, respectively. We refer to this mechanism as *Dual Mode Journaling*. The Dual Mode Journaling mechanism can support two journaling modes, the durability guarantee mode and ordering guarantee mode, in a versatile manner.

The commit thread is responsible for dispatching the write requests for *JD* and *JC*. The commit thread writes each of *JD* and *JC* with a barrier write so that *JD* and *JC* are persisted in order. The commit thread dispatches the write requests without any delay in between (Figure 8(b)). After dispatching the write request for *JC*, the commit thread inserts the journal transaction to the committing transaction list and hands over the control to the flush thread.

The flush thread is responsible for (i) issuing the flush command, (ii) handling error and retry, and (iii) removing the transaction from the committing transaction list. The behavior of the flush

thread varies subject to the durability requirement of the journal commit. If the journal commit is triggered by `fbarrier()`, then the flush thread does not have much things to do. It omits issuing the flush command and returns after removing the transaction from the committing transaction list. If the journal commit is triggered by `fsync()`, then the flush thread involves more steps. It issues a flush command and waiting for the completion. When the flush completes, it removes the the associated transaction from the committing transaction list and returns. BarrierFS supports all journal modes in EXT4: WRITEBACK, ORDERED, and DATA.

The dual thread organization of BarrierFS journaling bears profound implications in filesystem design. First, the separate support for the ordering guarantee and the durability guarantee naturally becomes an integral part of the filesystem. Ordering guarantee involves only the control plane activity. Durability guarantee requires the control plane activity as well as data plane activity. BarrierFS partitions the journal commit activity into two independent components, control plane and data plane, and dedicates separate threads to each. This modular design enables the filesystem primitives to adaptively adjust the activity of the data plane thread with respect to the durability requirement of the journal commit operation; `fsync()` vs. `fbarrier()`. Second, the filesystem journaling becomes concurrent activity. Thanks to the dual thread design, there can be multiple committing transactions in flight. In most journaling filesystems that we are aware of, the filesystem journaling is a serial activity: The journaling thread commits the following transaction only after the preceding transaction becomes durable. In dual thread design, the commit thread can commit a new journal transaction without waiting for the preceding committing transaction to become durable. The flush thread asynchronously notifies the application thread about the completion of the journal commit.

4.3 Synchronization Primitives

In `fbarrier()` and `fsync()`, BarrierFS writes D, JD, and JC in a piplelined manner without any delays in between (Figure 8(b)). BarrierFS writes D with one or more order-preserving writes, whereas it writes JD and JC with the barrier writes. In this manner, BarrierFS creates two epochs $\{D, JD\}$ and $\{JC\}$ in an `fsync()` or in an `fbarrier()` and ensures the storage order between these two epochs. `fbarrier()` returns when the filesystem dispatches the write request for JC. `fsync()` returns after it ensures that JC is made durable. The order-preserving block device satisfies the prefix constraint [71]. When JC becomes durable, the order-preserving block device guarantees that all blocks associated with preceding epochs have been made durable. An application may repeatedly call `fbarrier()` committing multiple transactions simultaneously. By writing JC with a barrier write, BarrierFS ensures that these committing transactions become durable in order. The latency of an `fsync()` reduces significantly in BarrierFS. It reduces the number of flush operations from two in EXT4 to one and eliminates the Wait-on-Transfer overheads (Figure 8). An `fsync()` and an `fbarrier()` in BarrierFS can be represented as in Equation (4). D_{OP} denotes the order-preserving write for D. JC_{BAR} and JC_{BAR} denote the barrier-writes for JD and JC, respectively. `farrier()` returns after the filesystem dispatches a write command for JC. A write command for JC is a barrier write. `fsync()` returns when the flush command finishes,

$$\overbrace{\underbrace{D_{\mathsf{OP}} \rightarrow JD_{\mathsf{BAR}} \rightarrow JC_{\mathsf{BAR}}}_{\texttt{fbarrier()}} \rightarrow \text{ xfer } \rightarrow \text{ flush}}^{\texttt{fsync()}}. \tag{4}$$

In `fdatabarrier()` and `fdatasync()`, BarrierFS writes D with a barrier write. If there are more than one write requests in writing D, then only the last one is set as a barrier write and the others are set as the order-preserving writes. An `fdatasync()` returns after the data blocks, D,

become durable. An `fdatabarrier()` returns immediately after dispatching the write requests for *D*. `fdatabarrier()` is the crux of the barrier-enabled IO stack. With `fdatabarrier()`, the application can control the storage order virtually without any overheads: without waiting for the flush, without waiting for DMA completion, and even without the context switch. `fdatabarrier()` is a very lightweight storage barrier.

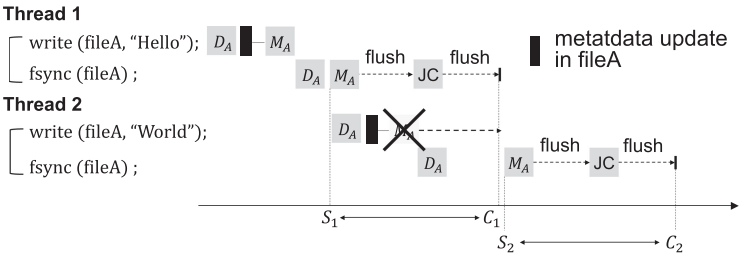
An `fdatabarrier()` (or `fdatasync()`) may not find any dirty pages to synchronize on its execution. In this case, BarrierFS explicitly triggers the journal commit. It forces BarrierFS to issue the barrier writes for *JD* and *JC*. Through this mechanism, `fdatabarrier()` or `fdatasync()` can delimit an epoch as desired by the application even in the absence of any dirty pages.

4.4 Handling the Page Conflicts

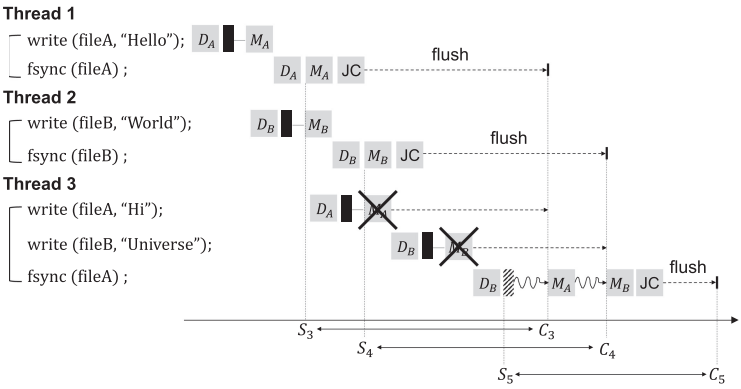
A buffer page may have been held by the committing transaction when an application tries to insert it to the running transaction. We refer to this situation as *page conflict*. Blindly inserting a conflict page into the running transaction yields its removal from the committing transaction before it becomes durable. The EXT4 filesystem checks for the page conflict when it inserts a buffer page to the running transaction [69]. If the filesystem finds a conflict, then the thread stores a copy of the metadata page being held by the committing transaction to a temporary location(`b_frozen_data`) and overwrites the existing metadata block with a new content. This mechanism is similar to *undo logging* [18]. When the committing transaction becomes durable, the JBD thread identifies the conflict pages in the committed transaction and inserts them to the running transaction. In EXT4, there can be at most one committing transaction. The running transaction is guaranteed to be free from page conflict when the JBD thread has made it durable and finishes inserting the conflict pages to the running transaction. Figure 9(a) illustrates an example. Thread 1 and thread 2 write "Hello" and "World" to fileA, respectively. Both of the threads synchronize the result of their updates with `fsync()`. Assume that thread 2 starts after thread 1 does. They update the same file. Thread 2 updates the data block. When thread 2 attempts to update the metadata block for file A, the metadata block already has been held by the committing transaction. When the JBD finishes committing the current transaction at C_1, it inserts the updated metadata block that has been subject to the page conflict to the running transaction. The JBD thread changes the state of the associated transaction from running to committing and commits the transaction. The JBD commits the updated metadata by Thread 2 at time S_2.

In BarrierFS, there can be more than one committing transactions. A running transaction can have more than one conflict page. Each of the conflict pages may be associated with the different committing transaction. We refer to this situation as *multi-transaction page conflict*. *Multi-transaction page conflict* calls for a different mechanism for resolving the page conflict. As in EXT4, BarrierFS inserts the conflict pages to the running transaction when it makes a committing transaction durable. However, BarrierFS cannot commit a running transaction even though one of the committing transaction has become durable and has resolved the conflicts associated with it. Each time when a committing transaction becomes durable, the flush thread notifies the commit thread about the completion of a journal commit. To commit a running transaction, BarrierFS needs to ensure that the running transaction is free from any page conflicts. The commit thread has to scan all buffer pages of the existing committing transactions for page conflicts. When there exists a large number of committing transactions, the scanning overhead to check for the page conflict can be prohibitive in BarrierFS.

To reduce the overhead of handling multi-transaction page conflict, we propose the *conflict-page list* for a running transaction. The conflict-page list represents the set of conflict pages associated with a running transaction. The filesystem inserts the buffer page to the conflict-page list when it finds that the buffer page that it needs to insert to the running transaction is subject to the page



(a) EXT4: Journal commit triggered by Thread2 can start only after the preceding journal commit finishes.



(b) BarrierFS: Journal commit triggered by Thread 3 can start only after the journal commits from Thread 1 and from Thread 2 finishes.

Fig. 9. Handling Page Conflict in EXT4 and in BarrierFS. M_A, metadata block for file A; D_A, Data block for file A; JC, Journal commit block.

conflict. When the flush thread has made a committing transaction durable, it removes the conflict pages from the conflict-page list in addition to inserting them to the running transaction. A running transaction can only be committed when the conflict-page list is empty. An order-preserving IO stack is carefully designed not to make any substantial changes in the existing filesystem and the block device layer. Conflict-page list is the only data structure newly introduced by BarrierFS.

Figure 9(b) illustrates a multi-page conflict in BarrierFS. Thread 1 writes "Hello" to fileA and calls `fsync()`. Thread 2 writes "World" to fileB and calls `fsync()`. These two threads work with different files. The commit thread of BarrierFS commits the two journal transactions in concurrent manner, since they are independent. Thread 3 writes "Hi" to fileA and and "University" to fileB. Then, Thread 3 calls `fsync()`to make the result of the updates durable. There are two committing transactions when the Thread 3 calls `fsync()`. The metadata for fileA and the metadata for fileB are still being committed when Thread 3 calls `fsync()`. The commit transaction that has been triggered by Thread 1 finishes at C_3. The commit transaction that has been triggered by Thread 2 finishes at C_4. The metadata for fileA that has been updated by thread 3 can be inserted to the running transaction at C_3. The metadata for fileB that has been updated by thread 3 can be inserted to the running transaction at C_4. At C_4, the running transaction is free from the page conflicts. The commit thread of BarrierFS commits the running transaction created by Thread 3 after both committing transactions become durable (at C_4).

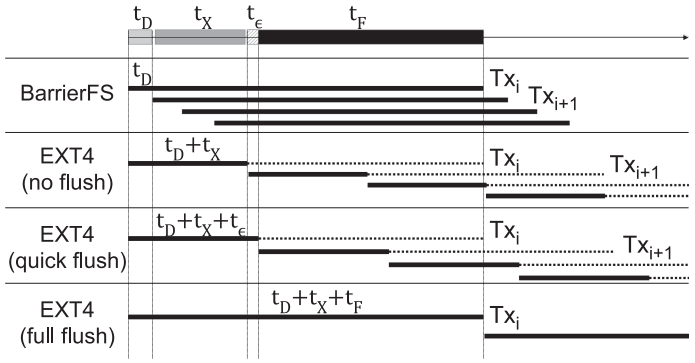


Fig. 10. Concurrency in filesystem journaling under varying storage order guarantee mechanisms; t_D, dispatch latency; t_X, transfer latency; t_ϵ, flush latency in supercap SSD; t_F, flush latency.

4.5 Concurrency in Filesystem Journaling

In most journaling filesystems, e.g., EXT4 and XFS, the filesystem journaling is a serial activity; the filesystem commits the following journal transaction only after the preceding journal transaction becomes durable. A number of techniques are being used to introduce some degree of concurrency into the filesystem journaling. The objective of these techniques is to enable the host to commit the following transaction before the preceding transaction reaches the disk surface. One of the popular approaches is to use super-capacitor-based SSD. In supercap SSD, the storage controller returns the flush command without actually flushing the writeback cache. With supercap SSD, the filesysem commits the following journal transaction before the preceding journal transaction reaches the disk surface. Another widely used approach is using a `no-barrier` mount option in the EXT4 filesystem. When the filesystem is mounted with the `no-barrier` option, the filesystem omits issuing the flush command in committing a journal transaction. With this option, the EXT4 guarantees neither durability nor ordering in the journal commit operation, since the storage controller may make the data blocks durable out-of-order. When a server is equipped with uninterruptible power supply (UPS) or maintains multiple replicas of the data, one can choose to use the `no-barrier` mount option for the filesystem without the risk of data loss.

We examine the details of filesystem journaling behavior under four different ways to committing a journal transaction: BarrierFS, EXT4 with `no-barrier` option (EXT4 (no flush)), EXT4 with supercap SSD (EXT4 (quick flush)), and plain EXT4 (EXT4 (full flush)). We examine the EXT4 (no flush) to illustrate the filesystem journaling behavior when the flush command is removed in the journal commit operation.

Figure 10 schematically illustrates how the filesystem commits the journal transactions in four different ways of filesystem journal commit. In Figure 10, each horizontal line segment represents a journal commit activity. A horizontal line segment consists of the solid line segment and the dotted line segment. The end of the horizontal line segment denotes the time when the transaction reaches the disk surface. The end of the solid line segment represents the time when the journal commit returns. If they do not coincide, then it means that the journal commit finishes before the transaction actually reaches the disk surface. EXT4 (full flush), EXT4 (quick flush), and EXT4 (no flush) share the same journaling mechanism; the filesystem commits the new transaction only after the preceding journal commit finishes. They differ in the time when the journal commit finishes. In EXT4 (full flush), the journal commit finishes only after all associated blocks reaches the disk surface. The consecutive journal commit operations are interleaved with dispatch latency (t_D), transfer latency (t_X), and flush latency (t_F). In EXT4 (quick flush), the journal commit finishes

before the journal transaction reaches the disk surface. The durability of a journal commit is still guaranteed, since the contents in the writeback cache of the storage device are guaranteed to be durable thanks to supercap. The journal commit operation finishes more quickly than in EXT4 (full flush). The consecutive journal commit operations are interleaved with dispatch latency (t_D), transfer latency (t_X), and very short flush latency (t_ϵ). In EXT4 (no flush), the journal commit finishes when the associated filesystem blocks reach the writeback cache of the storage device. In EXT4 (no flush), the journal commit finishes more quickly than EXT (quick flush). The consecutive journal commit operations are interleaved with dispatch latency (t_D) and transfer latency (t_X).

In BarrierFS, the filesystem journaling is grounded on the fundamentally different operating principle from the above-mentioned three schemes, EXT4 (full flush), EXT4 (quick flush), and EXT4 (no flush). In BarrierFS, the filesystem commits the following journal transaction without waiting for the preceding journal commit to finish. BarrierFS commits the following journal transaction immediately after it dispatches the write requests for the preceding journal commit. Dedicating the separate threads for control plane activity and for data plane activity, respectively, the commit thread that is in charge of control plane activity can keep committing the transactions without waiting for the preceding journal transaction to finish. Dual thread design of the BarrierFS journal module changes the operating principle of the modern filesystem journaling. In BarrierFS, the consecutive journal commit operations are interleaved by dispatch latency (t_D). From the aspect of the throughput of the filesystem journaling, BarrierFS prevails the remainders by far, since the interval between the consecutive journal commits is as small as the dispatch latency in BarrierFS.

The performance improvements in EXT4 (no flush) and in EXT4 (quick flush) have their price. EXT4 (quick flush) requires the additional hardware component, the supercap, in the SSD. EXT4 (quick flush) guarantees neither durability or ordering in the journal commit. BarrierFS commits multiple transactions concurrently and yet can guarantee the durability of the individual journal commit without the assistance of the additional hardware.

The barrier enabled IO stack does not require any major changes in the existing in-memory or on-disk structure of the IO stack. The only new data structure we introduce is the "conflict-page-list" for a running transaction. Barrier enabled IO stack consists of approximately 3K LOC changes in the existing block device layer and EXT4 code base of the Linux kernel.

4.6 Comparison with OptFS

As the closest approach of our sort, OptFS deserves an elaboration. OptFS and barrier-enabled IO stack differ mainly in three aspects; the target storage media, the technology domain, and the programming model. First, OptFS is not designed for flash storage but the barrier-enabled IO stack is. OptFS is designed to reduce the disk seek overhead in a filesystem journaling. Via committing multiple transactions together (delayed commit), OptFS gives the storage controller more of an opportunity to reduce the seek time associated with persisting a set of blocks. Via logging some of the updated data blocks to the journal transaction instead of updating them in place, OptFS transforms the disk access for journal commit operation into a sequential one (selective data mode journaling). Second, OptFS is the filesystem technique while the barrier-enabled IO stack deals with the entire IO stack; the storage device, the block device layer, and the filesystem. OptFS is built on the legacy block device layer. It suffers from the same overhead as the existing filesystems do. OptFS uses Wait-on-Transfer to control the transfer order between *D* and *JD*. OptFS relies on Transfer-and-Flush to control the storage order between the journal commit and the associated checkpoint in `osync()`. Barrier-enabled IO stack eliminates the overhead of Wait-on-Transfer and Transfer-and-Flush in controlling the storage order. Third, OptFS focuses on revising the filesystem journaling model. BarrierFS is not limited to revising the filesystem journaling model but also exports generic storage barrier with which the application can group a set of writes into an epoch.

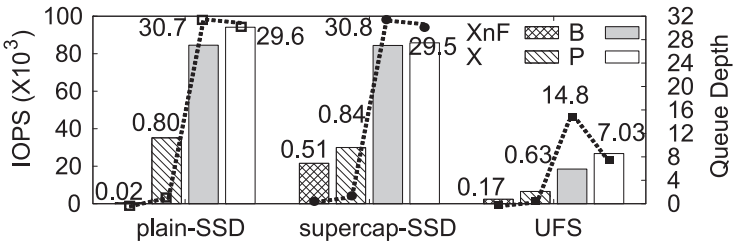


Fig. 11. 4KB Randwom Write; XnF, `write()` followed by `fdatasync()`; X, `write()` followed by `fdatasync()`(`no-barrier` option); B, `write()` followed by `fdatabarrier()`; P, orderless `write()`.

5 APPLICATIONS

To date, `fdatasync()` has been the sole resort to enforce the storage order between the write requests. The virtual disk managers for VM disk image, e.g., *qcow2*, use `fdatasync()` to enforce the storage order among the writes to the VM disk image [5]. SQLite uses `fdatasync()` to control the storage order between the undo-log and the journal header and between the updated database node and the commit block [37]. In a single insert transaction, SQLite calls `fdatasync()` four times, three of which are to control the storage order. In these cases, `fdatabarrier()` can be used in place of `fdatasync()`. In some modern applications, e.g., mail server [62] or OLTP, `fsync()` accounts for the dominant fraction of IO. In TPC-C workload, 90% of IOs are created by `fsync()` [51]. With improved `fsync()` of BarrierFS, the performance of the application can increase significantly. Some applications prefer to trade the durability and the freshness of the result for the performance and scalability of the operation [10, 15]. One can replace all `fsync()` and `fdatasync()` with ordering guarantee counterparts, `fbarrier()` and `fdatabarrier()`, respectively, in these applications.

6 EXPERIMENT

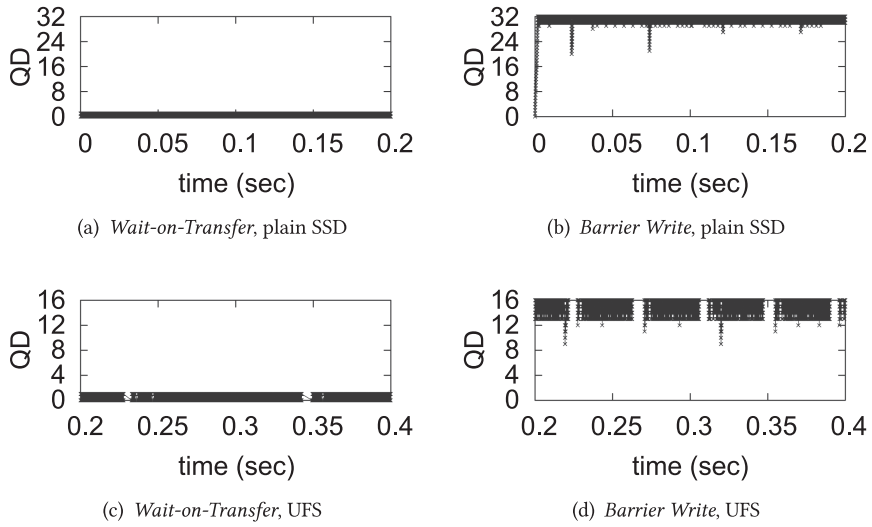
We implement a barrier-enabled IO stack on three different platforms: enterprise server (12 cores, Linux 3.10.61), a PC server (4 cores, Linux 3.10.61), and a smartphone (Galaxy S6, Android 5.0.2, Linux 3.10). We test three storage devices: 843TN (SATA 3.0, QD[2] = 32, 8 channels, supercap), 850PRO (SATA 3.0, QD = 32, 8 channels), and mobile storage (UFS 2.0, QD = 16, single channel). We refer to each of these as supercap-SSD, plain-SSD, and UFS, respectively. We compare the BarrierFS against EXT4 and OptFS [7]. We implement the barrier write command in the UFS device. In plain-SSD and supercap SSD, we assume that the performance overhead of barrier write is 5% and none, respectively.

6.1 Order-Preserving Block Layer

We examine the performance of 4KB random write with different ways of enforcing the storage order: P (orderless write [i.e., plain buffered write]), B (barrier write), X (Wait-on-Transfer), and XnF (Transfer-and-Flush). Figure 11 illustrates the result.

The overhead of Transfer-and-Flush is severe. With Transfer-and-Flush, the IO performances of the ordered write are 0.5% and 10% of orderless write in plain-SSD and UFS, respectively. In supercap SSD, the performance overhead is less significant but is still considerable; the performance of the ordered write is 35% of the orderless write. In supercap SSD, the flush command returns instantly and therefore the overhead of flush command is not significant. However, the write requests are still interleaved with Wait-on-Transfer overhead, which leaves the performance of

[2]QD: queue depth.



(a) *Wait-on-Transfer*, plain SSD

(b) *Barrier Write*, plain SSD

(c) *Wait-on-Transfer*, UFS

(d) *Barrier Write*, UFS

Fig. 12. Queue Depth, 4KB Random Write.

ordered write in supercap SSD is 35% of the orderless write. Supercap SSD successfully mitigates the overhead of flush command and improves the performance of ordered write significantly.

In this experiment, we find that the overhead of Wait-on-Transfer is significant in modern flash storage. When we interleave the write requests with DMA transfer, the IO performance is less than 40% of the orderless write in all three storage devices. We carefully argue that based on this observation, it is vital to eliminate the Wait-on-Transfer overhead in storage order guarantee to exploit the hardware potential of the modern high-performance storage device.

Order-preserving block device layer brings substantial performance gain. In all storage devices, via using barrier write instead of Wait-on-Transfer, the performance of the ordered write increases to 2× and becomes nearly as good as the performance of the orderless write. The barrier write is successful enforcing the storage order while fully exploiting the performance of the underlying SSD. With a barrier write, the ordered write exhibits 90% performance of the orderless write in plain-SSD and super-cap SSD. For UFS, it exhibits 80% performance of the orderless write.

Figure 11 also illustrates the average depth of the command queue in each storage device. We observe that the average queue depth is lower in buffered IO than in barrier write. This is because in barrier write, each ordered write request is dispatched as a single command, whereas in buffered IO, a number of write requests can be merged together to form a single IO command.

The barrier write drives the queue to its maximum in all three flash storages. The storage performance is closely related to the command queue utilization [33]. In Wait-on-Transfer, the queue depth never goes beyond one (Figure 12(a) and Figure 12(c)). In barrier write, the queue depth grows near to its maximum in all storage devices (Figure 12(b) and Figure 12(d)). Note that the command queue depths of plain SSD and UFS storage are 32 and 16, respectively.

6.2 Filesystem Journaling

We examine the latency, the number of context switches and the queue depth in filesystem journaling in EXT4 and BarrierFS. We use Mobibench [26]. For latency, we perform 4KB allocating `write()` followed by `fsync()`. With this, an `fsync()` always finds the updated metadata to journal and the `fsync()` latency properly represents the time to commit a journal transaction. For

Table 1. fsync() Latency Statistics (ms)

(%)	UFS		plain-SSD		supercap-SSD	
	EXT4	BFS	EXT4	BFS	EXT4	BFS
μ	1.29	0.51	5.95	3.52	0.15	0.09
Median	1.20	0.44	5.43	3.01	0.15	0.09
99th	4.15	3.51	11.41	8.96	0.16	0.10
99.9th	22.83	9.02	16.09	9.30	0.28	0.24
99.99th	33.10	17.60	17.26	14.19	4.14	1.35

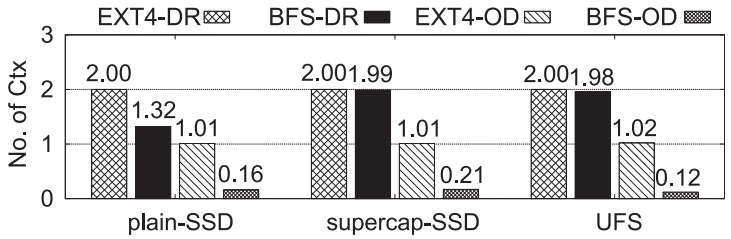


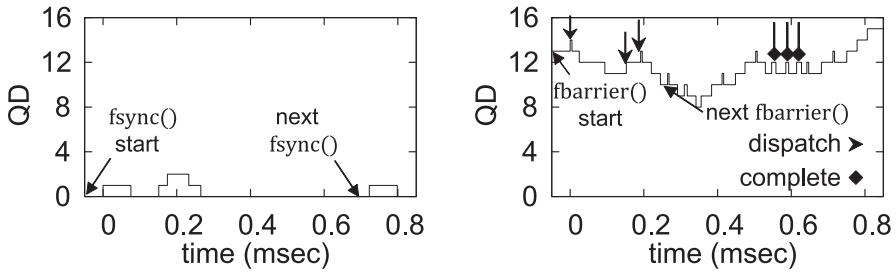
Fig. 13. Average Number of Context Switches, EXT4-DR: fsync(), BFS-DR: fsync(), EXT-OD: fsync() with no-barrier, BFS-OD: fbarrier(), "DR" = durability guarantee, "OD" = ordering guarantee, "EXT4-OD" guarantees only the transfer order but not storage order.

context switch and queue depth, we use 4 KB non-allocating random write followed by different synchronization primitives.

Latency: In plain-SSD and supercap-SSD, the average fsync() latency decreases by 40% when we use BarrierFS against when we use EXT4 (Table 1). In UFS, the fsync() latency decreases by 60% in BarrierFS compared against EXT4. UFS experiences more significant reduction in fsync() latency than the other SSDs do.

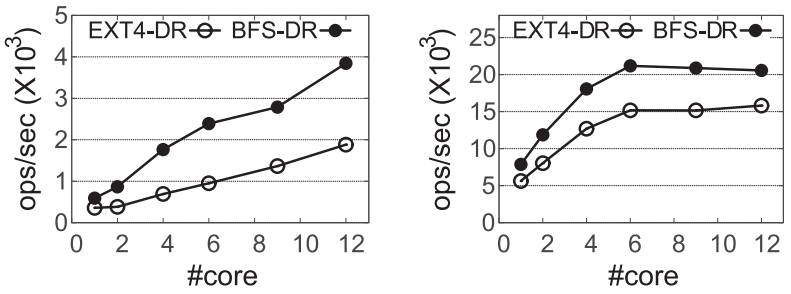
BarrierFS makes the fsync() latency less variable. In supercap-SSD and UFS, the fsync() latencies at the 99.99th percentile are 30× of the average fsync() latency (Table 1). In BarrierFS, the tail latencies at 99.99th percentile decrease by 50%, 20%, and 70% in UFS, plain-SSD, and supercap-SSD, respectively, against EXT4.

Context Switches: We examine the number of application-level context switches in different journaling modes (Figure 13). In EXT4, fsync() wakes up the caller twice: after D is transferred and after the journal transaction is made durable(EXT4-DR). This applies to all three storages. In BarrierFS, the number of context switches in an fsync() varies subject to the storage device. In UFS and supercap SSD, fsync() of BarrierFS wakes up the caller twice, as in the case of fsync() of EXT4. However, the reasons are entirely different. In UFS and supercap-SSD, the intervals between the consecutive write requests are much smaller than the timer interrupt interval due to small flush latency. UFS uses MLC Flash. Supercap SSD returns the flush command instantly. A write() request rarely finds the updated metadata and an fsync() often resorts to an fdatasync(). fdatasync() wakes up the caller (the application thread) twice in BarrierFS: after transferring D and after flush completes. In plain SSD, fsync() of BarrierFS wakes up the caller once: after the transaction is made durable. The plain-SSD uses TLC Flash. The interval between the successive write()s is longer than the timer interrupt interval. The application thread blocks after triggering the journal commit and and wakes up after the journal commit operation completes.



(a) Durability Guarantee (b) Ordering Guarantee

Fig. 14. Queue Depth in BarrierFS: `fsync()` and `fbarrier()`.



(a) plain-SSD (b) supercap-SSD

Fig. 15. fxmark: scalability of filesystem journaling.

BFS-OD manifests the benefits of BarrierFS. The `fbarrier()` rarely finds the updated metadata, since it returns quickly, and, as a result, most `fbarrier()` calls are serviced as `fdatabarrier()`. `fdatabarrier()` does not block the caller and therefore does not accompany any involuntary context switch.

Command Queue Depth: In BarrierFS, the host dispatches the write requests for *D*, *JD*, and *JC* in tandem. Ideally, there can be as many as three commands in the queue. We observe only up to two commands in the queue in servicing an `fsync()` (Figure 14(a)). This is due to the context switch between the application thread and the commit thread. Writing *D* and writing *JD* are 160μs apart, but it takes 70μs to service the write request for *D*. In `fbarrier()`, BarrierFS successfully drives the command queue to its full capacity (Figure 14(b)).

Throughput and Scalability: The filesystem journaling is a main obstacle against building a manycore scalable system [44]. We examine the throughput of filesystem journaling in EXT4 and BarrierFS with a varying number of CPU cores in a 12-core machine. We use modified DWSL workload in `fxmark` [45]; each thread performs a 4KB allocating write followed by `fsync()`. Each thread operates on its own file. BarrierFS exhibits much more scalable behavior than EXT4 (Figure 15). In plain-SSD, BarrierFS exhibits 2× performance against EXT4 in all numbers of cores (Figure 15(a)). In supercap-SSD, the performance saturates with six cores in both EXT4 and BarrierFS. BarrierFS exhibits 1.3× journaling throughput against EXT4 (Figure 15(b)).

6.3 Server Workload

We run two workloads: `varmail` [73] and OLTP-insert [34]. OLTP-insert workload uses MySQL DBMS [47]. `varmail` is a metadata-intensive workload. It is known for the heavy `fsync()` traffic.

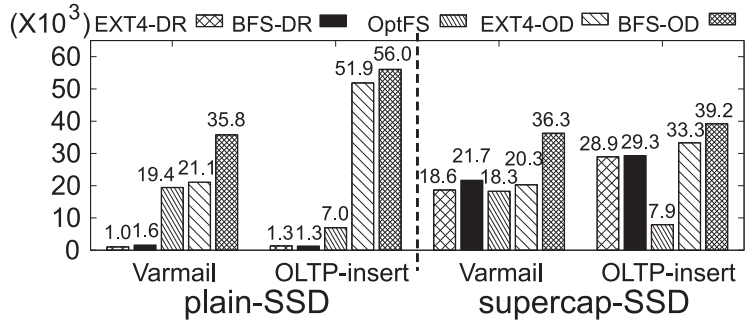


Fig. 16. `varmail` (ops/s) and OLTP-insert (Tx/s).

There are total four combinations of the workload and the SSD (plain-SSD and supercap-SSD) pair. For each combination, we examine the benchmark performances for durability guarantee and ordering guarantee, respectively. For durability guarantee, we leave the application intact and use two filesystems, the EXT4 and the BarrierFS (EXT4-DR and BFS-DR). The objective of this experiment is to examine the efficiency of `fsync()` implementations in EXT4 and BarrierFS, respectively. For ordering guarantee, we test three filesystems, OptFS, EXT4, and BarrierFS. In OptFS and BarrierFS, we use `osync()` and `fdatabarrier()` in place of `fsync()`, respectively. In EXT4, we use `nobarrier` mount option. This experiment examines the benefit of Wait-on-Dispatch. Figure 16 illustrates the result.

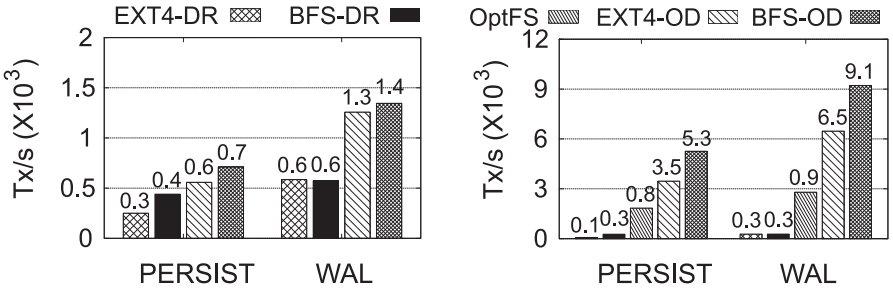
Let us examine the performances of `varmail` workload. In plain-SSD, BFS-DR brings 60% performance gain against EXT4-DR in `varmail` workload. In supercap-SSD, BFS-DR brings 10% performance gain against EXT4-DR. The experimental result of supercap-SSD case clearly shows the importance of eliminating the Wait-on-Transfer overhead in controlling the storage order. The benefit of BarrierFS manifests itself when we relax the durability guarantee. In ordering guarantee, BarrierFS achieves 80% performance gain against EXT4-OD. Compared to the baseline, EXT4-DR, BarrierFS achieves 36× performance (1.0 vs. 35.6 IOPS) when we enforce only ordering guarantee with BarrierFS (BFS-OD) in plain SSD .

In MySQL, BFS-OD prevails EXT4-OD by 12%. Compared to the baseline, EXT4-DR, BarrierFS achieves 43× performance (1.3 vs. 56.0 IOPS) when we enforce only ordering guarantee with BarrierFS (BFS-OD) in plain SSD.

6.4 Mobile Workload: SQLite

We examine the performances of the library-based embedded DBMS, SQLite, under the durability guarantee and the ordering guarantee, respectively. We examine two journal modes, PERSIST and WAL. We use "Full Sync," and the WAL file size is set to 1,000 pages, both of which are default settings [58]. In a single `insert` transaction, SQLite calls `fdatasync()` 4 times. Three of them are to control the storage order, and the last one is for making the result of a transaction durable.

For durability guarantee mode, we replace the first three `fdatasync()`'s with `fdatabarrier()`'s and leave the last `fdatasync()` intact. In mobile storage, BarrierFS achieves 75% performance improvement against EXT4 in default PERSIST journal mode under durability guarantee (Figure 17). In ordering guarantee, we replace all four `fdatasync()`'s with `fdatabarrier()`'s. In UFS, SQLite exhibits 2.8× performance gain in BFS-OD against EXT4-DR. The benefit of eliminating the Transfer-and-Flush becomes more dramatic as the storage controller employs higher degree of parallelism. In plain-SSD, SQLite exhibits 73× performance gain in BFS-OD against EXT4-DR (73 vs. 5,300inserts per second).



(a) UFS (b) plain-SSD

Fig. 17. SQLite Performance: inserts per second, 100K inserts.

Notes on OptFS: OptFS does not perform well in our experiment (Figure 16 and Figure 17), unlike that in Reference [7]. We find two reasons. First, the benefit of delayed checkpoint and selective data mode journaling becomes marginal in flash storage. Second, in flash storage (i.e., the storage with short IO latency) the delayed checkpoint and the selective data mode journaling negatively interact with each other and bring substantial increase in the memory pressure. The increased memory pressure severely hurts the performance of `osync()`. Here is the reason. When an `osync()` is called, it scans all dirty pages to determine if each of them can be checkpointed. This scanning overhead turns out to be non-trivial when OptFS is used for flash storage. In selective data mode journaling, OptFS inserts the updated data blocks to the journal transaction when the update is about modifying the existing block, not about allocating a new one. The selective data mode journaling can quickly increase the number of dirty pages in the system. Delayed checkpoint prohibits the data blocks in the journal transaction from being checkpointed until the associated Asynchronous Durability Notification (ADN) arrives. When combined together, the selective data mode journaling can create a large amount of dirty pages in a journal transaction, whereas the delayed checkpoint bars the filesystem from removing the dirty pages from a journal transaction. As a result, `osync()` checkpoints only a small fraction of dirty pages each time it is called. On the same token, the dirty pages in the journal transactions are scanned multiple times before they are checkpointed. The `osync()` shows particularly poor performance in OLTP workload (Figure 16). This is because in OLTP workload, most writes are for update operations. They overwrite the existing file blocks. Selective data mode journaling leaves most write operations to data mode journaling.

6.5 Crash Consistency

We test if the BarrierFS recovers correctly against the unexpected system failure. We use `CrashMonkey` for the test [40]. We modify CrashMonkey to understand the barrier write so that the CrashMonkey can properly delimit an epoch when it encounters a barrier write. We run two workloads; `rename_root_to_sub` and `create_delete`. Table 2 summarizes the result of the test. For durability guarantee (`fsync()`), BarrierFS passes all 1,000 test cases as EXT4 does in both workloads. For ordering guarantee (`fsync()` in EXT4-OD and `fbarrier()` in BarrierFS), BarrierFS passes all 1,000 test cases, whereas EXT4-OD fails in some cases. This is not surprising, since EXT4 with `nobarrier` option guarantees neither the transfer order nor the persist order in committing the filesystem journal transaction.

7 RELATED WORK

Featherstitch [19] proposes a programming model to specify the set of requests that can be scheduled together, `patchgroup`, and the ordering dependency between them, `pg_depend()`. While

Table 2. Crash Consistency Test of EXT4 and BarrierFS, Scenario A: `rename_root_to_sub`; Scenario B: `create_delete`

Scenario	—	EXT4-DR	BFS-DR	EXT4-OD	BFS-OD
A	clean	1,000	1,000	547	1,000
	fixed	0	0	0	0
	failed	0	0	453	0
B	clean	1,000	1,000	109	1,000
	fixed	0	0	891	0
	failed	0	0	0	0

`xsyncfs` [49] mitigates the overhead of `fsync()`, it needs to maintain complex causal dependencies among buffered updates. `NoFS` (no order file system) [8] introduces "backpointer" to eliminate the Transfer-and-Flush–based ordering in the file system. It does not support transaction.

A few works proposed to use multiple running transactions or multiple committing transactions to circumvent the Transfer-and-Flush overhead in filesystem journaling [29, 38, 55]. IceFS [38] allocates a separate running transaction for each container. SpanFS [29] splits a journal region into multiple partitions and allocates committing transactions for each partition. CCFS [55] allocates separate running transactions for individual threads. In these systems, each journaling session still relies on the Transfer-and-Flush mechanism.

A number of file systems provide a multi-block atomic write feature [16, 35, 54, 70] to relieve applications from the overhead of logging and journaling. These file systems internally use the Transfer-and-Flush mechanism to enforce the storage order in writing the data blocks and the associated metadata blocks. Exploiting the order-preserving block device layer, these filesystems can use the Wait-on-Dispatch mechanism to enforce the storage order between the data blocks and the metadata blocks and can be saved from the Transfer-and-Flush overhead.

8 CONCLUSION

Flash storage provides the cache barrier command to allow the host to control the persist order. Mechanical characteristics of the HDD prohibits the host from controlling the order in which the contents in the writeback cache reach the disk surface. This fundamental nature of HDD prohibits itself from providing the storage function such as cache barrier. It is time for designing the new IO stack for the flash storage that is free from the unnecessary constraint inherited from the old legacy that the host cannot control the persist order. We built a barrier-enabled IO stack based on the foundation that the host can control the persist order. In the barrier-enabled IO stack, the host can dispense with *Transfer-and-Flush* overhead in controlling the storage order and can successfully saturate the underlying flash storage. We like to conclude this work with two key observations. First, the "cache barrier" command is a necessity rather than a luxury. It should be supported in all flash storage products ranging from the mobile storage to the high-performance flash storage with supercap. Second, the block device layer should be designed to eliminate the DMA transfer overhead in controlling the storage order. As the flash storage becomes quicker, the relative cost of tardy "Wait-on-Transfer" will become more substantial. To saturate the flash storage, the host should be able to control the transfer order with neither Flush overhead nor DMA transfer overhead. We hope that this work provides a useful foundation in designing a new IO stack for the flash storage.[3]

[3]The source code for barrier enabled IO stack is available at https://github.com/ESOS-Lab/barrieriostack.

ACKNOWLEDGMENT

We thank Vijay Chidambaram and the anonymous reviewers of USENIX FAST 2018 for their valuable comments. Their comments and guidance have been extremely helpful in strengthening this article. We also like to thank Jayashree Mohan for her help with CrashMonkey.

REFERENCES

[1] Jens Axboe. 2004. Linux block IO present and future. In *Proceedings of the Ottawa Linux Symposium*. Ottawa, Ontario, Canada.
[2] Steve Best. 2000. JFS Overview. Retrieved from http://jfs.sourceforge.net/project/pub/jfs.pdf.
[3] Yu-Ming Chang, Yuan-Hao Chang, Tei-Wei Kuo, Yung-Chun Li, and Hsiang-Pang Li. 2015. Achieving SLC performance with MLC flash memory. In *Proceedings of the Design Automation Conference (DAC'15)*.
[4] F. Chen, R. Lee, and X. Zhang. 2011. Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing. In *Proceedings of the IEEE Symposium on High Performance Computer Architecture (HPCA'11)*.
[5] Qingshu Chen, Liang Liang, Yubin Xia, Haibo Chen, and Hyunsoo Kim. 2016. Mitigating sync amplification for copy-on-write virtual disk. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST'16)*. 241–247.
[6] Vijay Chidambaram. 2015. *Orderless and Eventually Durable File Systems*. Ph.D. Dissertation. University of Wisconsin–Madison.
[7] Vijay Chidambaram, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2013. Optimistic crash consistency. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP'13)*.
[8] Vijay Chidambaram, Tushar Sharma, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2012. Consistency without ordering. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST'12)*.
[9] Yong Sung Cho, Il Han Park, Sang Yong Yoon, Nam Hee Lee, Sang Hyun Joo, Ki-Whan Song, Kihwan Choi, Jin-Man Han, Kye Hyun Kyung, and Young-Hyun Jun. 2013. Adaptive multi-pulse program scheme based on tunneling speed classification for next generation multi-bit/cell NAND flash. *IEEE J. Solid-State Circ.* 48, 4 (2013), 948–959.
[10] James Cipar, Greg Ganger, Kimberly Keeton, Charles B Morrey III, Craig AN Soules, and Alistair Veitch. 2012. LazyBase: Trading freshness for performance in a scalable database. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys'12)*.
[11] Danny Cobb and Amber Huffman. 2012. NVM express and the PCI express SSD revolution. In *Proceedings of the Intel Developer Forum*.
[12] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. 2009. Better I/O through byte-addressable, persistent memory. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP'09)*.
[13] Jonathan Corbet. 2010. Barriers and journaling filesystems. Retrieved from http://lwn.net/Articles/283161/.
[14] Jonathan Corbet. 2010. The end of block barriers. Retrieved from https://lwn.net/Articles/400541/.
[15] Henggang Cui, James Cipar, Qirong Ho, Jin Kyu Kim, Seunghak Lee, Abhimanu Kumar, Jinliang Wei, Wei Dai, Gregory R. Ganger, Phillip B. Gibbons, and others. 2014. Exploiting bounded staleness to speed up big data analytics. In *Proceedings of the USENIX Annual Technical Conference (ATC'14)*.
[16] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. 2001. Wide-area cooperative storage with CFS. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP'01)*.
[17] Brian Dees. 2005. Native command queuing-advanced performance in desktop storage. *IEEE Potent. Mag.* 24, 4 (2005), 4–7.
[18] Ramez Elmasri. 2008. *Fundamentals of Database Systems*. Pearson Education India, 815–817.
[19] Christopher Frost, Mike Mammarella, Eddie Kohler, Andrew de los Reyes, Shant Hovsepian, Andrew Matsuoka, and Lei Zhang. 2007. Generalized file system dependencies. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP'07)*.
[20] Jongmin Gim and Youjip Won. 2010. Extract and infer quickly: Obtaining sector geometry of modern hard disk drives. *ACM Trans. Stor.* 6, 2 (2010).
[21] Laura M. Grupp, John D. Davis, and Steven Swanson. 2012. The bleak future of NAND flash memory. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST'12)*. 1.
[22] Jie Guo, Jun Yang, Youtao Zhang, and Yiran Chen. 2013. Low cost power failure protection for MLC NAND flash storage systems with PRAM/DRAM hybrid buffer. In *Proceedings of the Design, Automation and Test Conference (DATE'13)*. 859–864.
[23] Christoph Hellwig. Patchwork Block: Update Documentation for REQ_FLUSH/REQ_FUA. Retrieved from https://patchwork.kernel.org/patch/134161/.

[24] Mark Helm, Jae-Kwan Park, Ali Ghalam, Jason Guo, Chang wan Ha, Cairong Hu, Heonwook Kim, Kalyan Kavalipurapu, Eric Lee, Ali Mohammadzadeh, and others. 2014. 19.1 A 128Gb MLC NAND-flash device using 16nm planar cell. In *Proceedings of the IEEE International Solid-State Circuits Conference (ISSCC'14)*.

[25] SK hynix. 2015. eMMC5.1 solution in SK hynix. Retrieved from https://www.skhynix.com/kor/product/nandEMMC.jsp.

[26] Sooman Jeong, Kisung Lee, Seongjin Lee, Seoungbum Son, and Youjip Won. 2013. I/O stack optimization for smartphones. In *Proceedings of the USENIX Annual Technical Conference (ATC'13)*. Berkeley, CA.

[27] JEDEC Standard JESD220C. 2016. Universal flash storage(UFS) version 2.1.

[28] JEDEC Standard JESD84-B51. 2015. Embedded multi-media card(eMMC) electrical standard (5.1).

[29] Junbin Kang, Benlong Zhang, Tianyu Wo, Weiren Yu, Lian Du, Shuai Ma, and Jinpeng Huai. 2015. SpanFS: A scalable file system on fast storage devices. In *Proceedings of the USENIX Annual Technical Conference (ATC'15)*. Berkeley, CA.

[30] Woon-Hak Kang, Sang-Won Lee, Bongki Moon, Gi-Hwan Oh, and Changwoo Min. 2013. X-FTL: Transactional FTL for SQLite databases. In *Proceedings of the ACM Special Interest Group on Management of Data (SIGMOD'13)*.

[31] Ram Kesavan, Rohit Singh, Travis Grusecki, and Yuvraj Patel. 2017. Algorithms and data structures for efficient free space reclamation in WAFL. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST'17)*. USENIX Association, Berkeley, CA, 1–14.

[32] Hyeong-Jun Kim and Jin-Soo Kim. 2011. Tuning the Ext4 filesystem performance for android-based smartphones. In *Proceedings of the 2011 International Conference on Frontiers in Computer Education (ICFCE'11)*, Sabo Sambath and Egui Zhu (Eds.), Vol. 133. Springer, 745–752.

[33] Youngjae Kim. 2015. An empirical study of redundant array of independent solid-state drives (RAIS). *Cluster Comput.* 18, 2 (2015), 963–977.

[34] Alexey Kopytov. 2004. SysBench Manual. Retrieved from http://imysql.com/wp-content/uploads/2014/10/sysbench-manual.pdf.

[35] Changman Lee, Dongho Sim, Jooyoung Hwang, and Sangyeun Cho. 2015. F2FS: A new file system for flash storage. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST'15)*. Berkeley, CA.

[36] Seungjae Lee, Jin-yub Lee, Il-han Park, Jongyeol Park, Sung-won Yun, Min-su Kim, Jong-hoon Lee, Minseok Kim, Kangbin Lee, Taeeun Kim, and others. 2016. 7.5 A 128Gb 2b/cell NAND flash memory in 14nm technology with tPROG=640us and 800MB/s I/O rate. In *Proceedings of the IEEE International Solid-State Circuits Conference (ISSC'16)*.

[37] Wongun Lee, Keonwoo Lee, Hankeun Son, Wook-Hee Kim, Beomseok Nam, and Youjip Won. 2015. WALDIO: Eliminating the filesystem journaling in resolving the journaling of journal anomaly. In *Proceedings of the USENIX Annual Technical Conference (ATC'15)*. Berkeley, CA.

[38] Lanyue Lu, Yupu Zhang, Thanh Do, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2014. Physical disentanglement in a container-based file system. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)*.

[39] Youyou Lu, Jiwu Shu, Jia Guo, Shuai Li, and Onur Mutlu. LightTx: A lightweight transactional design in flash-based SSDs to support flexible transactions. In *Proceedings of the IEEE IEEE International Conference on Computer Design (ICCD'13)*.

[40] Ashlie Martinez and Vijay Chidambaram. 2017. CrashMonkey: A framework to automatically test file-system crash consistency. In *Proceedings of the 9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage'17)*.

[41] Avantika Mathur, Mingming Cao, Suparna Bhattacharya, Andreas Dilger, Alex Tomas, and Laurent Vivier. 2007. The new ext4 filesystem: Current status and future plans. In *Proceedings of the Linux Symposium 2007*.

[42] Marshall K. McKusick, Gregory R. Ganger, and others. 1999. Soft updates: A technique for eliminating most synchronous writes in the fast filesystem. In *Proceedings of the USENIX Annual Technical Conference (ATC'99)*.

[43] Changwoo Min, Woon-Hak Kang, Taesoo Kim, Sang-Won Lee, and Young Ik Eom. 2015. Lightweight application-level crash consistency on transactional flash storage. In *Proceedings of the USENIX Annual Technical Conference (ATC'15)*. Berkeley, CA.

[44] Changwoo Min, Sanidhya Kashyap, Steffen Maass, and Taesoo Kim. 2016. Understanding manycore scalability of file systems. In *Proceedings of the USENIX Annual Technical Conference (ATC'16)*.

[45] Changwoo Min, Sanidhya Kashyap, Steffen Maass, and Taesoo Kim. 2016. Understanding manycore scalability of file systems. In *Proceedings of the USENIX Annual Technical Conference (ATC'16)*. 71–85.

[46] C Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. 1992. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst.* 17, 1 (1992), 94–162.

[47] AB MySQL. 2007. *Mysql 5.1 Reference Manual*. Sun Microsystems.

[48] Dushyanth Narayanan, Austin Donnelly, and Antony Rowstron. 2008. Write off-loading: Practical power management for enterprise storage. *ACM Trans. Stor.* 4, 3 (2008), 10:1–10:23.

[49] Edmund B. Nightingale, Kaushik Veeraraghavan, Peter M. Chen, and Jason Flinn. 2006. Rethink the sync. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI'06)*.

[50] M. Okun and A. Barak. 2002. Atomic writes for data integrity and consistency in shared storage devices for clusters. In *Proceedings of the International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP'02)*.

[51] Jiaxin Ou, Jiwu Shu, and Youyou Lu. 2016. A high performance file system for non-volatile main memory. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys'16)*.

[52] Xiangyong Ouyang, David Nellans, Robert Wipfel, David Flynn, and Dhabaleswar K Panda. 2011. Beyond block I/O: Rethinking traditional storage primitives. In *Proceedings of the IEEE Symposium on High Performance Computer Architecture (HPCA'11)*.

[53] Salvador Palanca, Stephen A. Fischer, Subramaniam Maiyuran, and Shekoufeh Qawami. 2016. MFENCE and LFENCE micro-architectural implementation method and system. (July 5 2016). US Patent 9,383,998.

[54] Stan Park, Terence Kelly, and Kai Shen. 2013. Failure-atomic msync(): A simple and efficient mechanism for preserving the integrity of durable data. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys'13)*.

[55] Thanumalayan Sankaranarayana Pillai, Ramnatthan Alagappan, Lanyue Lu, Vijay Chidambaram, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2017. Application crash consistency and performance with CCFS. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST'17)*. Berkeley, CA, 181–196.

[56] Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2005. IRON file systems. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP'05)*.

[57] Vijayan Prabhakaran, Thomas L. Rodeheffer, and Lidong Zhou. 2008. Transactional flash. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI'08)*. Berkeley, CA, 147–160. http://dl.acm.org/citation.cfm?id=1855741.1855752

[58] Dhathri Purohith, Jayashree Mohan, and Vijay Chidambaram. 2017. The dangers and complexities of SQLite benchmarking. In *Proceedings of the 8th Asia-Pacific Workshop on Systems (APSys'17)*. ACM, New York, NY. DOI: http://dx.doi.org/10.1145/3124680.3124719

[59] H. Rev. 2014. *SCSI Commands Reference Manual*. Seagate.

[60] Ohad Rodeh, Josef Bacik, and Chris Mason. 2013. BTRFS: The linux B-tree filesystem. *ACM Trans. Stor.* 9, 3 (2013).

[61] Mendel Rosenblum and John K. Ousterhout. 1992. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst.* 10, 1 (Feb. 1992), 26–52. DOI: http://dx.doi.org/10.1145/146941.146943

[62] Priya Sehgal, Vasily Tarasov, and Erez Zadok. 2010. Evaluating performance and energy in file system server workloads. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST'10)*. Berkeley, CA.

[63] Margo I. Seltzer, Gregory R. Ganger, Marshall K. McKusick, Keith A. Smith, Craig A. N. Soules, and Christopher A. Stein. 2000. Journaling versus soft updates: Asynchronous meta-data protection in file systems. In *Proceedings of the USENIX Annual Technical Conference (ATC'00)*. Berkeley, CA.

[64] Girish Shilamkar. 2007. Journal Checksums. Retrieved from http://wiki.old.lustre.org/images/4/44/Journal-\checksums.pdf.

[65] SQLite. 2018. Well-known Users of SQLite. Retrieved from https://www.sqlite.org/famous.html.

[66] Adam Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. 1996. Scalability in the XFS file system. In *Proceedings of the USENIX Annual Technical Conference (ATC'96)*. Berkeley, CA, 1. http://dl.acm.org/citation.cfm?id=1268299.1268300

[67] Toshiba. 2015. Toshiba Expands Line-up of e-MMC Version 5.1 Compliant Embedded NAND Flash Memory Modules. Retrieved from http://toshiba.semicon-storage.com/us/company/taec/news/2015/03/memory-20150323-1.html.

[68] Theodore Ts'o. 2015. Using Cache barrier in liue of REQ_FLUSH. Retrieved from http://www.spinics.net/lists/linux-ext4/msg49018.html.

[69] Stephen C. Tweedie. 1998. Journaling the linux ext2fs filesystem. In *Proceedings of the 4th Annual Linux Expo*.

[70] Rajat Verma, Anton Ajay Mendez, Stan Park, Sandya Mannarswamy, Terence Kelly, and Charles Morrey. 2015. Failure-atomic updates of application data in a linux file system. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST'15)*. Berkeley, CA

[71] Yang Wang, Manos Kapritsos, Zuocheng Ren, Prince Mahajan, Jeevitha Kirubanandam, Lorenzo Alvisi, and Mike Dahlin. 2013. Robustness in the salus scalable block store. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation (NSDI'13)*. USENIX Association, Berkeley, CA, 357–370. http://dl.acm.org/citation.cfm?id=2482626.2482661

[72] Zev Weiss, Sriram Subramanian, Swaminathan Sundararaman, Nisha Talagala, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2015. ANViL: Advanced virtualization for modern non-volatile memory devices. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST'15)*. Berkeley, CA.

[73] Andrew Wilson. 2008. The new and improved filebench. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST'08)*. Berkeley, CA.

[74] Qiumin Xu, Huzefa Siyamwala, Mrinmoy Ghosh, Tameesh Suri, Manu Awasthi, Zvika Guz, Anahita Shayesteh, and Vijay Balakrishnan. 2015. Performance analysis of NVMe SSDs and their implication on real world databases. In *Proceedings of the ACM International Systems and Storage Conference (SYSTOR'15)*. Haifa, Israel.

[75] S. y. Park, E. Seo, J. Y. Shin, S. Maeng, and J. Lee. 2010. Exploiting internal parallelism of flash-based SSDs. *IEEE Comput. Arch. Lett.* 9, 1 (2010), 9–12.

[76] C. Zhang, Y. Wang, T. Wang, R. Chen, D. Liu, and Z. Shao. 2014. Deterministic crash recovery for NAND flash based storage systems. In *Proceedings of the ACM/EDAC/IEEE Design Automation Conference (DAC'14)*.

Received June 2018; accepted July 2018