Barrier-Enabled IO Stack for Flash Storage

Youjip Won, *Hanyang University*; Jaemin Jung, *Texas A&M University*;
Gyeongyeol Choi, Joontaek Oh, and Seongbae Son, *Hanyang University*;
Jooyoung Hwang and Sangyeun Cho, *Samsung Electronics*

https://www.usenix.org/conference/fast18/presentation/won

ISBN 978-1-931971-42-3

Barrier-Enabled IO Stack for Flash Storage

Youjip Won[1] Jaemin Jung[2*] Gyeongyeol Choi[1]
Joontaek Oh[1] Seongbae Son[1] Jooyoung Hwang[3] Sangyeun Cho[3]

[1]*Hanyang University* [2]*Texas A&M University* [3]*Samsung Electronics*

Abstract

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*. 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.

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 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 [47, 26, 35], in filesystem journaling [67, 41, 65, 4], in soft-update [42, 63], or in copy-on-write or log-structure filesystems [61, 35, 60, 31]. Enforcing a storage order is 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 [24, 16].

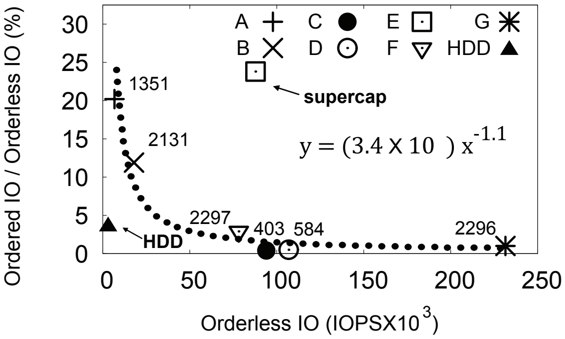


Figure 1: Ordered write vs. Orderless write, Except 'HDD', all are Flash storages; A: (1ch)/eMMC5.0, B: (1ch)/UFS2.0, C: (8ch)/SATA3.0, D: (8ch)/NVMe, E: (8ch)/SATA3.0 (supercap), F: (8ch)/PCIe, G: (32ch) Flash array, The number next to each point is the IOPS of `write()` followed by `fdatasync()`.

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 [73, 6], the large size storage cache [48], and the deep command queue [19, 27, 72]. A state of the art NVMe SSD reportedly exhibits up to 750 KIOPS random read performance [72]. 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) [25, 36], and the multi-bits per cell (MLC, TLC, and QLC) [5, 11]. Meanwhile, the time to program a Flash cell has barely improved, and is even deteriorating in some cases [22].

The Transfer-and-Flush based order-preserving mechanism conflicts with the parallelism and the concurrency in the Flash storage. It disables the parallelism and the concurrency feature of the Flash storage and exposes the raw Flash cell programming latency to the host. The

[*] This work was done while the author was a graduate student at Hanyang University.

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. Fig. 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. In ordered random write, each write request is followed by `fdatasync()`. X-axis denotes the throughput of orderless write which 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 thirty-two 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 [23], `no-barrier` mount option at the EXT4 filesystem [15], and transactional checksum [56, 32, 64]. Efforts such as transactional filesystem [50, 18, 54, 35, 68] and transactional block device [30, 74, 43, 70, 52] save the application from the overhead of enforcing the storage order associated with filesystem journaling. A school of work address more fundamental aspects in controlling the storage order, such as separating the ordering guarantee from durability guarantee [9], providing a programming model to define the ordering dependency among the set of writes [20], 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 [9] relies on Transfer-and-Flush in enforcing the order between the journal commit and the associated checkpoint. Featherstitch [20] 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 upon 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 upon 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 followings;

- 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 multiple transactions concurrently and yet can guarantee the durability of the individual journal commit.

By eliminating the Transfer-and-Flush overhead, the barrier-enabled IO stack successfully exploits the concurrency and the parallelism in modern Flash storage.

2 Background

2.1 Orders in 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.

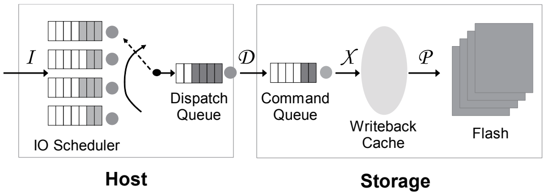


Figure 2: IO stack Organization

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*, $\mathscr{I}$, *Dispatch Order*, $\mathscr{D}$, *Transfer Order*, $\mathscr{X}$, and *Persist Order*, $\mathscr{P}$. The issue order $\mathscr{I} = \{i_1, i_2, \ldots, i_n\}$ 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 $\mathscr{D} = \{d_1, d_2, \ldots, d_n\}$ 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, $\mathscr{X} = \{x_1, x_2, \ldots, x_n\}$, is the set of transfer completions. The persist order, $\mathscr{P} = \{p_1, p_2, \ldots, p_n\}$, 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.

- $\mathscr{I} \neq \mathscr{D}$. The IO scheduler reorders and coalesces the IO requests subject to the scheduling principle, e.g. CFQ, DEADLINE, etc. When there is no scheduling mechanism, e.g. NO-OP scheduler [3] or NVMe [13] interface, the dispatch order may be equal to the issue order.
- $\mathscr{D} \neq \mathscr{X}$. The storage controller can freely schedule the commands in its command queue. In addition, the commands can be serviced out-of-order due to the errors, the time-outs, and the retry.
- $\mathscr{X} \neq \mathscr{P}$. The writeback cache of the storage is not FIFO. In Flash storage, persist order is governed not by the order in which the data blocks are made durable but by the order in which the associated mapping table entries are updated. The two may not coincide.

Due to all these uncertainties, the modern IO stack is said to be *orderless* [8].

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, $\mathscr{I}$, and the order in which the associated data blocks are made durable, $\mathscr{P}$. It is equivalent to collectively enforcing the partial orders between the pair of the orders in the adjacent layers in Fig. 2. It can be formally represented as in Eq. 1.

$$(\mathscr{I} = \mathscr{P}) \equiv (\mathscr{I} = \mathscr{D}) \wedge (\mathscr{D} = \mathscr{X}) \wedge (\mathscr{X} = \mathscr{P}) \quad (1)$$

The modern IO stack has evolved under the assumption that the host cannot control the persist order, i.e. $\mathscr{X} \neq \mathscr{P}$. This is due to 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 which is hidden from outside [21]. When the host blindly enforces a certain persist order, it may experience anomalous delay in IO service. Due to this constraint of $\mathscr{X} \neq \mathscr{P}$, Eq. 1 is unsatisfiable. The constraint that the host cannot control the persist order is a fundamental limitation in modern IO stack design.

The block device layer adopts the indirect and the expensive approach to control the storage order in spite of the constraint $\mathscr{X} \neq \mathscr{P}$. 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 $\mathscr{D} = \mathscr{X}$. Wait-on-Transfer is expensive; it blocks the caller and interleaves the requests with DMA transfer. Second, when the preceding command is serviced, the caller issues the flush command and waits for its completion. 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 and to satisfy $\mathscr{X} = \mathscr{P}$. 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*.

The cost of Transfer-and-Flush is prohibitive. It neutralizes 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 data blocks are persisted before the journal

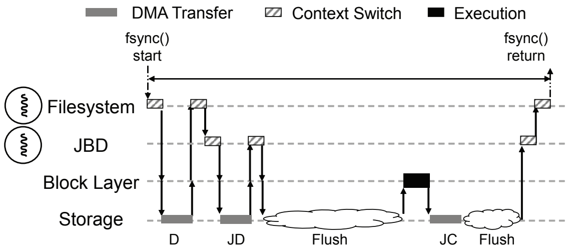


Figure 3: DMA, flush, and context switches in `fsync()`, ‘D’, ‘JC’ and ‘JC’ denote the DMA transfer time for *D*, *JD* and *JC*, respectively. ‘Flush’ denotes the time to service the flush request.

transaction. Fig. 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 DMA transfer. When the DMA transfer completes, 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 storage controller may persist the journal transaction 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 paper, we will use *JD* and *JC* to denote the coalesced chunk of the journal descriptor 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 [67, 9]. For the storage order within a transaction, JBD interleaves the write request for *JD* and the write request for *JC* with Transfer-and-Flush. To control the storage order between the transactions, JBD thread waits for *JC* to become durable before it starts committing the following transaction. JBD uses Transfer-and-Flush mechanism in enforcing both intra-transaction and inter-transaction storage order.

In earlier days of Linux, the block device layer explicitly issued a flush command in committing a journal transaction [15]. In this approach, the flush command blocks not only the caller but also the other requests in the same dispatch queue. Since Linux 2.6.37, the filesystem (JBD) implicitly issues a flush command [16]. In writing *JC*, JBD tags the write request with `REQ_FLUSH` and `REQ_FUA`. Most storage controllers have evolved to support these two flags; with these two flags, the storage controller flushes the writeback cache before servicing the command and in servicing the command it directly persists *JC* to storage surface bypassing the writeback cache. In this approach, only the JBD thread blocks and the other threads that share the same dispatch queue can proceed. Our effort can be thought as a continuation to this evolution of the IO stack. We mitigate the Transfer-and-Flush overhead by making the storage device more capable: supporting a barrier command and by redesigning the host side IO stack accordingly.

3 Order-Preserving Block Device Layer

3.1 Design

The order-preserving block device layer consists of the newly defined barrier write command, order-preserving dispatch module, and 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, $\mathscr{I} = \mathscr{D}$, $\mathscr{D} = \mathscr{X}$, and $\mathscr{X} = \mathscr{P}$, 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 $\mathscr{I}$ and the persist order $\mathscr{P}$ without Transfer-and-Flush.

The order-preserving block device layer categorizes the write requests into two categories, *orderless* write and *order-preserving* write. The order-preserving requests are the ones that are subject to the storage ordering constraint. Orderless request is the one which is irrelevant to the ordering dependency and which can be scheduled freely. We distinguish the two to avoid imposing unnecessary ordering constraint in scheduling the requests. The details are to come shortly. We refer to a set of the order-preserving requests that can

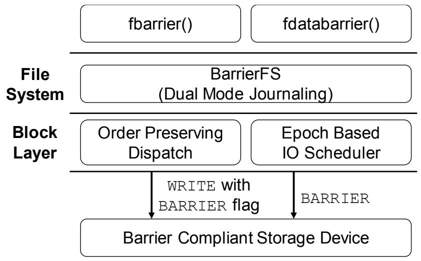


Figure 4: Organization of the barrier-enabled IO stack

be reordered with each other as an *epoch* [14]. 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 the `bio` object and the `request` object to represent an order-preserving write and a barrier write. `REQ_ORDERED` attribute is used to specify the order-preserving write. Barrier write request has both `REQ_ORDERED` and `REQ_BARRIER` attributes. The order-preserving block device layer handles the request differently based upon its category. Fig. 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 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 [1, 2]. The barrier command can satisfy the condition $\mathscr{X} = \mathscr{P}$ in Eq. 1 which has been unsatisfiable for several decades due to the mechanical characteristics of the rotating media. The naive way of using barrier is to replace the existing flush operation [66]. This simple replacement still leaves the caller under the 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 write command.

With reasonable complexity, the Flash storage can be made to support a barrier write command [30, 57, 39]. When the Flash storage has Power Loss Protection (PLP) feature, e.g. a supercapacitor, 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 [57, 39]. 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. When multiple controller cores concurrently write the data blocks to multiple channels, one may have to use sophisticated crash recovery protocol such as ARIES [46] to recover the storage to consistent state. If the entire Flash storage is treated as a single log device, we can use simple crash recovery algorithm used in LFS [61]. Since the persist order is enforced by the crash recovery logic, the storage controller can flush the writeback cache at the full throttle 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 use 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. It 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. In crash recovery, the UFS controller locates the beginning of the most recently flushed segment. It scans the pages in the segment from the beginning till it encounters the page that has not been programmed successfully. The storage controller discards the rest of the pages including the incomplete one.

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

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

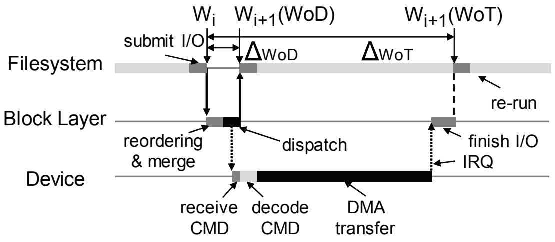


Figure 5: Wait-on-Dispatch vs Wait-on-Transfer, W_i: i^{th} write request, W_{i+1}(WoD): $(i+1)^{th}$ write request under Wait-on-Dispatch, W_{i+1}(WoT): $(i+1)^{th}$ write request under Wait-on-Transfer

layer dispatches the following command immediately after it dispatches the preceding one (Fig. 5) and yet the host can 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 $\mathscr{D} = \mathscr{X}$ in Eq. 1 without Wait-on-Transfer overhead.

The dispatch module constructs a command from the requests. The dispatch module constructs the barrier write command when it encounters the barrier write request, i.e. the write request with `REQ_ORDERED` and `REQ_BARRIER` flags. For the other requests, it constructs the commands as it used to do in the legacy block device.

Implementing an order-preserving dispatch is rather simple; 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. The following is the reason. SCSI standard defines three command priority levels: *head of the queue*, *ordered*, and *simple* [59]. With each, the storage controller puts the incoming command at the head of the command queue, at the tail of the command queue or at an arbitrary position determined at its disposal, respectively. The default priority is *simple*. The command with *simple* priority cannot be inserted in front of the existing *ordered* or *head of the queue* command. 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 device can temporarily be unavailable or the caller can be switched out involuntarily after dispatching a write request. 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 request after a certain time interval, e.g. 3 msec for SCSI devices [59]. The *ordered* priority command has rarely been used in the existing block device implementations. This is because when the host cannot control the persist order, enforcing a transfer order with *ordered* priority command barely carries any meaning from the perspective of ensuring the storage order. In the emergence of the barrier write, the ordered priority plays an essential 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 Fig. 5 illustrate the delays between the consecutive requests in Wait-on-Transfer and Wait-on-Dispatch, 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 $\mathscr{I} = \mathscr{D}$. 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, 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, all order-preserving requests in the queue belong to the same epoch. The requests in the queue can be freely re-ordered and merged with each other. The IO scheduler uses the existing scheduling discipline, e.g. CFQ. The merged request will be order-preserving if one of the components is 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 not any order-preserving requests in the queue, the IO scheduler starts accepting the IO requests again. 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.

Fig. 6 illustrates an example. The circle and the rectangle that enclose the write request denote the order-preserving flag and barrier flag, respectively. An

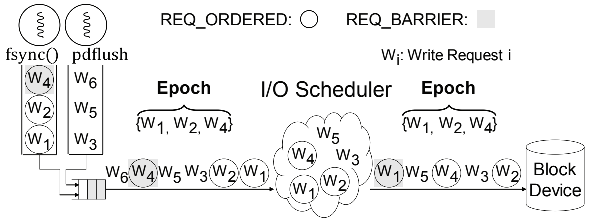


Figure 6: Epoch Based Barrier Reassignment

`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 writes. 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 stops accepting the new request. 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, $\mathscr{I} = \mathscr{D}, \mathscr{D} = \mathscr{X}$ and $\mathscr{X} = \mathscr{P}$ in Eq. 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

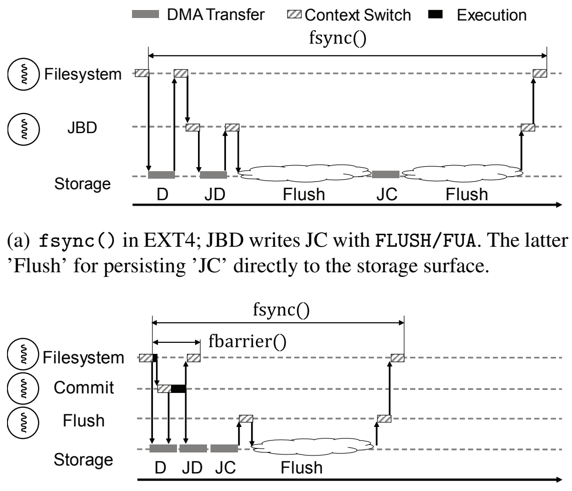
The barrier-enabled IO stack offers four synchronization primitives: `fsync()`, `fdatasync()`, `fbarrier()`, and `fdatabarrier()`. We propose two new filesystem interfaces, `fbarrier()` and `fdatabarrier()`, to separately support ordering guarantee. `fbarrier()` and `fdatabarrier()` synchronize the same 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 [9] in that it writes the data blocks and the journal transactions in order but returns without ensuring that they become durable.

`fdatabarrier()` synchronizes the modified blocks, but not the journal transaction. 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.

```
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.

4.2 Dual Mode Journaling



(a) `fsync()` in EXT4; JBD writes JC with `FLUSH/FUA`. The latter ’Flush’ for persisting ’JC’ directly to the storage surface.

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

Figure 7: Details of `fsync()` and `fbarrier()`

Committing a journal transaction essentially consists of two saparate 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 requests) and the data plane activity (persisting the associated data blocks and journal transaction) of a journal commit operation. Further, we allocate the separate threads to each task so that the two activities can proceed in parallel with minimum dependency. The two threads are called as *commit* thread and *flush* thread, respectively. We refer to this mechanism as *Dual Mode Journaling*. Dual Mode Journaling mechanism can support two journaling modes, durability guarantee mode and ordering guarantee mode, in versatile manner.

The commit thread is responsible for dispatching the write requests for *JD* and *JC*. The commit thread writes each of the two 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 (Fig. 7(b)). In EXT4, JBD thread interleaves the write request for *JC* and *JD* with Transfer-and-Flush (Fig. 7(a)). 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()`, the flush thread returns after removing the transaction from the committing transaction list. It returns without issuing the flush command. If the journal commit is triggered by `fsync()`, 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 (Fig. 7(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 forms 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. Order-preserving block device satisfies prefix constraint [69]. 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 (Fig. 7).

In `fdatabarrier()` and `fdatasync()`, BarrierFS writes *D* with a barrier write. If there are more than one write requests in writing *D*, 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 light-weight storage barrier.

An `fdatabarrier()` (or `fdatasync()`) may not find any dirty pages to synchronize upon 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 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 [67]. If the filesystem finds a conflict, the thread delegates the insertion to the JBD thread and blocks. 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.

In BarrierFS, there can be more than one committing transactions. The conflict pages may be associated with different committing transactions. We refer to this situation as *multi-transaction page conflict*. As in EXT4, BarrierFS inserts the conflict pages to the running transaction when it makes a committing transaction durable. However, to commit a running transaction, BarrierFS has to scan all buffer pages in the committing transactions for page conflicts and ensure that it is free from any page conflicts. When there exists large number of committing transactions, the scanning overhead to check for the page conflict can be prohibitive in BarrierFS.

To reduce this overhead, 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 conflict. When the filesystem 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.

4.5 Concurrency in Journaling

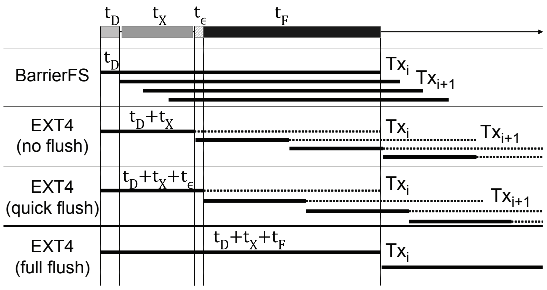


Figure 8: Concurrency in filesystem journaling under varying storage order guarantee mechanisms, t_D: dispatch latency, t_X: transfer latency, t_ϵ: flush latency in supercap SSD, t_F: flush latency

We examine the degree of concurrency in journal commit operation under different storage order guarantee mechanisms: BarrierFS, EXT4 with `no-barrier` option (EXT4 (no flush)), EXT4 with supercap SSD (EXT4 (quick flush)), and plain EXT4 (EXT4 (full flush)). With `no-barrier` mount option, the JBD thread omits the flush command in committing a journal transaction. With this option, the EXT4 guarantees neither durability nor ordering in journal commit operation since the storage controller may make the data blocks durable out-of-order. We examine this configuration to illustrate the filesystem journaling behavior when the flush command is removed in the journal commit operation.

In Fig. 8, each horizontal line segment represents a journal commit activity. It consists of the solid line segment and the dashed 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, it means that the journal commit finishes before the transaction reaches the disk surface. In EXT4 (full flush), EXT4 (quick flush), and EXT4 (no flush), the filesystem commits a new transaction only after preceding journal commit finishes. The journal commit is a serial activity. In EXT4 (full flush), the journal commit finishes only after all associated blocks are made durable. In EXT4 (quick flush), the journal commit finishes more quickly than in EXT4 (full flush) since the SSD returns the flush command without persisting the data blocks. In EXT4 (no flush), the journal commit finishes more quickly than EXT (quick flush) since it does not issue the flush command. In journaling throughput, BarrierFS prevails the remainders by far since the interval between the consecutive journal commits is as small as the dispatch latency, t_D.

The concurrencies in journaling in EXT4 (no flush) and in EXT4 (quick flush) have their price. EXT4 (quick flush) requires the additional hardware component, 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 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 IO stack 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 the 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) and via making the disk access sequential (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 upon 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.

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 [7]. 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 [12, 17]. 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), PC server (4 cores, Linux 3.10.61) and 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 compare the BarrierFS against EXT4 and OptFS [9]. We refer to each of these as supercap-SSD, plain-SSD, and UFS, respectively. We implement barrier write command in UFS device. In plain-SSD and supercap SSD, we assume that the performance overhead of barrier write is 5% and none, repsectively.

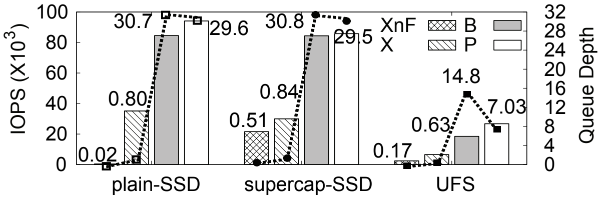
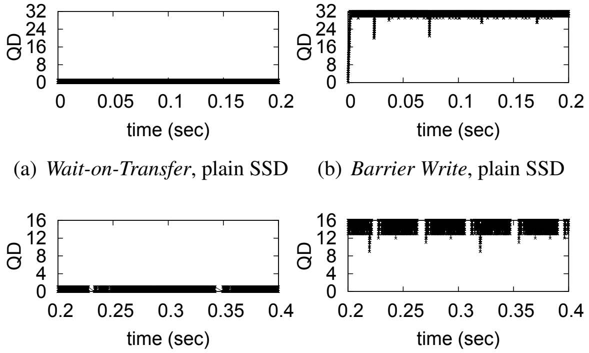


Figure 9: 4KB Randwom Write; XnF: `write()` followed by `fdatasync()`, X: `write()` followed by `fdatasync()(no-barrier option)`, B: `write()` followed by `fdatabarrier()`, P: orderless `write()`



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

(b) *Barrier Write*, plain SSD

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

(d) *Barrier Write*, UFS

Figure 10: Queue Depth, 4KB Random Write

6.1 Order-Preserving Block Layer

We examine the performance of 4 KByte 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). Fig. 9 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 UFS. The overhead of DMA transfer is significant. When we interleave the write requests with DMA transfer, the IO performance is less than 40% of the orderless write in each of the three storage devices.

The overhead of barrier write is negligible. When using 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. 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 (Fig. 10(a) and Fig. 10(c)). In barrier write, the queue depth grows near to its maximum in all storage devices (Fig. 10(b) and Fig. 10(d)).

[2] QD: queue depth

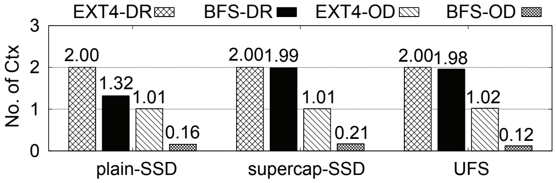


Figure 11: Average Number of Context Switches, EXT4-DR: `fsync()`, BFS-DR: `fsync()`, EXT-OD: `fsync()` with `no-barrier`, BFS-OD: `fbarrier()`, 'DR' = durability guarantee, 'OD' = ordering guarantee, 'EXT4-OD' guarantees only the transfer order, but not storage order.

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

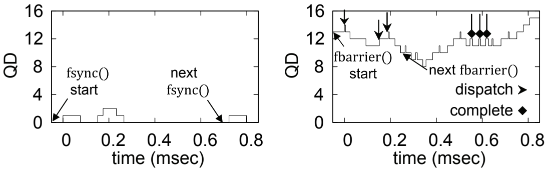
Latency: In plain-SSD and supercap-SSD, the average `fsync()` latency decreases by 40% when we use BarrierFS against when we use EXT4 (Table 2). In UFS, the `fsync()` latency decreases by 60% in BarrierFS compared against EXT4. UFS experiences more significant reduction in `fsync()` latency than the other SSDs do.

BarrierFS makes the `fsync()` latency less variable. In supercap-SSD and UFS, the `fsync()` latencies at the 99.99^{th} percentile are 30× of the average `fsync()` latency (Table 2). In BarrierFS, the tail latencies at 99.99^{th} percentile decrease by 50%, 20%, and 70% in UFS, plain-SSD, and supercap-SSD, respectively, against EXT4.

(%)	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
99^{th}	4.15	3.51	11.41	8.96	0.16	0.10
99.9^{th}	22.83	9.02	16.09	9.30	0.28	0.24
99.99^{th}	33.10	17.60	17.26	14.19	4.14	1.35

Table 1: `fsync()` latency statistics (msec)

Context Switches: We examine the number of application level context switches in different journaling modes (Fig. 11). 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 successive write requests are much smaller than the timer interrupt interval due to small flush latency. 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

Figure 12: Queue Depth in BarrierFS: `fsync()` and `fbarrier()`

BFS-OD manifests the benefits of BarrierFS. The `fbarrier()` rarely finds 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()` (Fig. 12(a)). This is due to the context switch between the application thread and the commit thread. Writing *D* and writing *JD* are 160 μsec apart, but it takes $70 \mu sec$ to service the write request for *D*. In `fbarrier()`, BarrierFS successfully drives the command queue to its full capacity (Fig. 12(b)).

Throughput and Scalability: The filesystem journaling is a main obstacle against building an 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 4-Kbyte allocating write followed by `fsync()`. Each thread operates on its own file. BarrierFS exhibits much more scalable behavior than EXT4 (Fig. 13). In plain-SSD, BarrierFS exhibits 2× performance against EXT4 in all numbers of cores (Fig. 13(a)). In supercap-

SSD, the performance saturates with six cores in both EXT4 and BarrierFS. BarrierFS exhibits 1.3× journaling throughput against EXT4 (Fig. 13(b)).

6.3 Server Workload

We run two workloads: `varmail` [71] 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. 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. Fig. 14 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 .

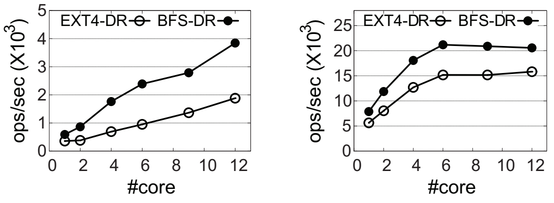
In MySQL, BFS-OD prevails EXT4-OD by 12%. Compared to the baseline, EXT4-DR, BarrierFS achieves 43× performance (1.3 vs. 56.0 IOPS) when we enforce only ordering guarantee with BarrierFS (BFS-OD) in plain SSD.

6.4 Mobile Workload: SQLite

We examine the performances of the libarary 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()` four 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 one. In mobile storage, BarrierFS achieves 75% performance improvement against EXT4 in default PERSIST journal mode under durability guarantee (Fig. 15). 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. 5300 ins/sec).

Notes on OptFS: OptFS does not perform well in our experiment (Fig. 14 and Fig. 15), unlike that in [9]. 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 impacts the performance of `osync()`. The `osync()` scans all dirty pages for the checkpoint at its beginning. Selective data mode journaling inserts the updated data blocks to the journal transaction. Delayed checkpoint prohibits the data blocks in the journal transaction from being checkpointed until the associated ADN arrives. As a result, `osync()` checkpoints only a small fraction of dirty pages each time it is called. The dirty pages in the journal transactions are scanned multiple times before they are checkpointed. The `osync()` shows particularly poor performance in OLTP workload (Fig. 14), where most



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

Figure 13: fxmark: scalability of filesystem journaling

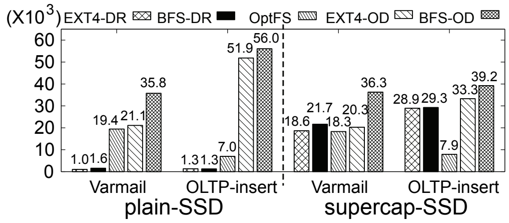
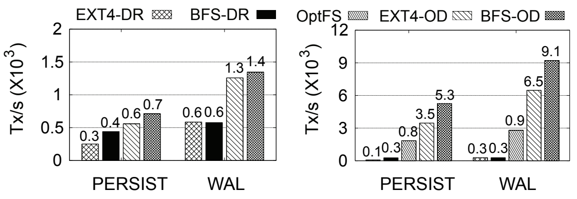


Figure 14: `varmail` (ops/s) and OLTP-insert (Tx/s)



(a) UFS (b) plain-SSD

Figure 15: SQLite Performance: ins/sec, 100K inserts

updates are subject 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`. 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 orders nor the persist orders in committing the filesystem journal transaction.

Scenario	-	EXT4-DR	BFS-DR	EXT4-OD	BFS-OD
A	clean	1000	1000	547	1000
	fixed	0	0	0	0
	failed	0	0	453	0
B	clean	1000	1000	109	1000
	fixed	0	0	891	0
	failed	0	0	0	0

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

7 Related Work

Featherstitch [20] 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 `xsyncfs` [49] mitigates the overhead of `fsync()`, it needs to maintain complex causal dependencies among buffered updates. `NoFS` (no order file system) [10] 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 [38, 29, 55]. IceFS [38] allocates 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 [18, 35, 54, 68] 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 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

The Flash storage provides the cache barrier command to allow the host to control the persist order. HDD cannot provide this feature. 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 upon 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 without interleaving the requests with DMA transfer.

We hope that this work provides a useful foundation in designing a new IO stack for the Flash storage[3].

9 Acknowledgement

We would like to thank our shepherd Vijay Chidambaram and the anonymous reviewers for their valuable feedback. We also would like to thank Jayashree Mohan for her help in CrashMonkey. 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 Samsung Elec.

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

References

[1] emmc5.1 solution in sk hynix. https://www.skhynix.com/kor/product/nandEMMC.jsp.

[2] Toshiba expands line-up of e-mmc version 5.1 compliant embedded nand flash memory modules. http://toshiba.semicon-storage.com/us/company/taec/news/2015/03/memory-20150323-1.html.

[3] AXBOE, J. Linux block IO present and future. In *Proc. of Ottawa Linux Symposium* (Ottawa, Ontario, Canada, Jul 2004).

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

[5] CHANG, Y.-M., CHANG, Y.-H., KUO, T.-W., LI, Y.-C., AND LI, H.-P. Achieving SLC Performance with MLC Flash Memory. In *Proc. of DAC 2015* (San Francisco, CA, USA, 2015).

[6] CHEN, F., LEE, R., AND ZHANG, X. Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing. In *Proc. of IEEE HPCA 2011* (San Antonio, TX, USA, Feb 2011).

[7] CHEN, Q., LIANG, L., XIA, Y., CHEN, H., AND KIM, H. Mitigating sync amplification for copy-on-write virtual disk. In *Proc. of USENIX FAST 2016* (Santa Clara, CA, 2016), pp. 241–247.

[8] CHIDAMBARAM, V. *Orderless and Eventually Durable File Systems*. PhD thesis, UNIVIRSITY OF WISCONSIN–MADISON, 2015.

[9] CHIDAMBARAM, V., PILLAI, T. S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Optimistic Crash Consistency. In *Proc. of ACM SOSP 2013* (Farmington, PA, USA, Nov 2013). https://github.com/utsaslab/optfs.

[10] CHIDAMBARAM, V., SHARMA, T., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Consistency Without Ordering. In *Proc. of USENIX FAST 2012* (San Jose, CA, USA, Feb 2012).

[11] CHO, Y. S., PARK, I. H., YOON, S. Y., LEE, N. H., JOO, S. H., SONG, K.-W., CHOI, K., HAN, J.-M., KYUNG, K. H., AND JUN, Y.-H. Adaptive multi-pulse program scheme based on tunneling speed classification for next generation multi-bit/cell NAND flash. *IEEE Journal of Solid-State Circuits(JSSC) 48*, 4 (2013), 948–959.

[12] CIPAR, J., GANGER, G., KEETON, K., MORREY III, C. B., SOULES, C. A., AND VEITCH, A. LazyBase: trading freshness for performance in a scalable database. In *Proc. of ACM EuroSys 2012* (Bern, Switzerland, Apr 2012).

[13] COBB, D., AND HUFFMAN, A. NVM express and the PCI express SSD Revolution. In *Proc. of Intel Developer Forum* (San Francisco, CA, USA, 2012).

[14] CONDIT, J., NIGHTINGALE, E. B., FROST, C., IPEK, E., LEE, B., BURGER, D., AND COETZEE, D. Better I/O through byte-addressable, persistent memory. In *Proc. of ACM SOSP 2009* (Big Sky, MT, USA, Oct 2009).

[15] CORBET, J. Barriers and journaling filesystems. http://lwn.net/Articles/283161/, August 2010.

[16] CORBET, J. The end of block barriers. https://lwn.net/Articles/400541/, August 2010.

[17] CUI, H., CIPAR, J., HO, Q., KIM, J. K., LEE, S., KUMAR, A., WEI, J., DAI, W., GANGER, G. R., GIBBONS, P. B., ET AL. Exploiting bounded staleness to speed up big data analytics. In *Proc. of USENIX ATC 2014* (Philadelihia, PA, USA, Jun 2014).

[18] DABEK, F., KAASHOEK, M. F., KARGER, D., MORRIS, R., AND STOICA, I. Wide-area Cooperative Storage with CFS. In *Proc. of ACM SOSP 2001* (Banff, Canada, Oct 2001).

[19] DEES, B. Native command queuing-advanced performance in desktop storage. *IEEE Potentials Magazine 24*, 4 (2005), 4–7.

[20] FROST, C., MAMMARELLA, M., KOHLER, E., DE LOS REYES, A., HOVSEPIAN, S., MATSUOKA, A., AND ZHANG, L. Generalized File System Dependencies. In *Proc. of ACM SOSP 2007* (Stevenson, WA, USA, Oct 2007).

[21] GIM, J., AND WON, Y. Extract and infer quickly: Obtaining sector geometry of modern hard disk drives. *ACM Transactions on Storage (TOS) 6*, 2 (2010).

[22] GRUPP, L. M., DAVIS, J. D., AND SWANSON, S. The bleak future of nand flash memory. In *Proc.of USENIX FAST 2012* (Berkeley, CA, USA, 2012).

[23] GUO, J., YANG, J., ZHANG, Y., AND CHEN, Y. Low cost power failure protection for mlc nand flash storage systems with pram/dram hybrid buffer. In *Proc. of DATE 2013* (Alpexpo Grenoble, France, 2013), pp. 859–864.

[24] HELLWIG, C. Patchwork block: update documentation for req_flush / req_fua. https://patchwork.kernel.org/patch/134161/.

[25] HELM, M., PARK, J.-K., GHALAM, A., GUO, J., WAN HA, C., HU, C., KIM, H., KAVALIPURAPU, K., LEE, E., MOHAMMADZADEH, A., ET AL. 19.1 A 128Gb MLC NAND-Flash device using 16nm planar cell. In *Proc. of IEEE ISSCC 2014* (San Francisco, CA, USA, Feb 2014).

[26] JEONG, S., LEE, K., LEE, S., SON, S., AND WON, Y. I/O Stack Optimization for Smartphones. In *Proc. of USENIX ATC 2013* (San Jose, CA, USA, Jun 2013).

[27] JESD220C, J. S. Universal Flash Storage(UFS) Version 2.1.

[28] JESD84-B51, J. S. Embedded Multi-Media Card(eMMC) Electrical Standard (5.1).

[29] KANG, J., ZHANG, B., WO, T., YU, W., DU, L., MA, S., AND HUAI, J. SpanFS: A Scalable File System on Fast Storage Devices. In *Proc. of USENIX ATC 2015* (Santa Clara, CA, USA, Jul 2015).

[30] KANG, W.-H., LEE, S.-W., MOON, B., OH, G.-H., AND MIN, C. X-FTL: Transactional FTL for SQLite Databases. In *Proc. of ACM SIGMOD 2013* (New York, NY, USA, Jun 2013).

[31] KESAVAN, R., SINGH, R., GRUSECKI, T., AND PATEL, Y. Algorithms and data structures for efficient free space reclamation in wafl. In *Proc. of USENIX FAST 2017* (Santa Clara, CA, 2017), USENIX Association, pp. 1–14.

[32] KIM, H.-J., AND KIM, J.-S. Tuning the ext4 filesystem performance for android-based smartphones. In *Proc. of ICFCE 2011* (2011), S. Sambath and E. Zhu, Eds., vol. 133 of *Advances in Intelligent and Soft Computing*, Springer, pp. 745–752.

[33] KIM, Y. An empirical study of redundant array of independent solid-state drives (RAIS). *Springer Cluster Computing 18*, 2 (2015), 963–977.

[34] KOPYTOV, A. SysBench manual. `http://imysql.com/wp-content/uploads/2014/10/sysbench-manual.pdf`, 2004.

[35] LEE, C., SIM, D., HWANG, J., AND CHO, S. F2FS: A New File System for Flash Storage. In *Proc. of USENIX FAST 2015* (Santa Clara, CA, USA, Feb 2015).

[36] LEE, S., LEE, J.-Y., PARK, I.-H., PARK, J., YUN, S.-W., KIM, M.-S., LEE, J.-H., KIM, M., LEE, K., KIM, T., ET AL. 7.5 A 128Gb 2b/cell NAND flash memory in 14nm technology with tPROG=640us and 800MB/s I/O rate. In *Proc. of IEEE ISSCC 2016* (San Francisco, CA, USA, Feb 2016).

[37] LEE, W., LEE, K., SON, H., KIM, W.-H., NAM, B., AND WON, Y. WALDIO: eliminating the filesystem journaling in resolving the journaling of journal anomaly. In *Proc. of USENIX ATC 2015* (Santa Clara, CA, USA, Jul 2015).

[38] LU, L., ZHANG, Y., DO, T., AL-KISWANY, S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Physical Disentanglement in a Container-Based File System. In *Proc. of USENIX OSDI 2014* (Broomfield, CO, USA, Oct 2014).

[39] LU, Y., SHU, J., GUO, J., LI, S., AND MUTLU, O. Lighttx: A lightweight transactional design in flash-based ssds to support flexible transactions. In *Proc. of IEEE ICCD 2013*.

[40] MARTINEZ, A., AND CHIDAMBARAM, V. Crashmonkey: A framework to automatically test file-system crash consistency. In *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 17)* (Santa Clara, CA, 2017). `https://github.com/utsaslab/crashmonkey`.

[41] MATHUR, A., CAO, M., BHATTACHARYA, S., DILGER, A., TOMAS, A., AND VIVIER, L. The new ext4 filesystem: current status and future plans. In *Proc. of Linux symposium 2007* (Ottawa, Ontario, Canada, Jun 2007).

[42] MCKUSICK, M. K., GANGER, G. R., ET AL. Soft Updates: A Technique for Eliminating Most Synchronous Writes in the Fast Filesystem. In *Proc. of USENIX ATC 1999* (Monterey, CA, USA, Jun 1999).

[43] MIN, C., KANG, W.-H., KIM, T., LEE, S.-W., AND EOM, Y. I. Lightweight application-level crash consistency on transactional flash storage. In *Proc. of USENIX ATC 2015* (Santa Clara, CA, USA, Jul 2015).

[44] MIN, C., KASHYAP, S., MAASS, S., AND KIM, T. Understanding Manycore Scalability of File Systems. In *Proc. of USENIX ATC 2016* (Denver, CO, USA, Jun 2016).

[45] MIN, C., KASHYAP, S., MAASS, S., AND KIM, T. Understanding manycore scalability of file systems. In *Proc.of USENIX ATC 2016* (Denver, CO, 2016), pp. 71–85.

[46] MOHAN, C., HADERLE, D., LINDSAY, B., PIRAHESH, H., AND SCHWARZ, P. ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems(TODS) 17*, 1 (1992), 94–162.

[47] MYSQL, A. Mysql 5.1 reference manual. *Sun Microsystems* (2007).

[48] NARAYANAN, D., DONNELLY, A., AND ROWSTRON, A. Write Off-loading: Practical Power Management for Enterprise Storage. *ACM Transactions on Storage(TOS) 4*, 3 (2008), 10:1–10:23.

[49] NIGHTINGALE, E. B., VEERARAGHAVAN, K., CHEN, P. M., AND FLINN, J. Rethink the Sync. In *Proc. of USENIX OSDI 2006* (Seattle, WA, USA, Nov 2006).

[50] OKUN, M., AND BARAK, A. Atomic writes for data integrity and consistency in shared storage devices for clusters. In *Proc. of ICA3PP 2002* (Beijing, China, Oct 2002).

[51] OU, J., SHU, J., AND LU, Y. A high performance file system for non-volatile main memory. In *Proc. of ACM EuroSys 2016* (London, UK, Apr 2016).

[52] OUYANG, X., NELLANS, D., WIPFEL, R., FLYNN, D., AND PANDA, D. K. Beyond block I/O: Rethinking traditional storage primitives. In *Proc. of IEEE HPCA 2011* (San Antonio, TX, USA, Feb 2011).

[53] PALANCA, S., FISCHER, S. A., MAIYURAN, S., AND QAWAMI, S. Mfence and lfence micro-architectural implementation method and system, July 5 2016. US Patent 9,383,998.

[54] PARK, S., KELLY, T., AND SHEN, K. Failure-atomic Msync(): A Simple and Efficient Mechanism for Preserving the Integrity of Durable Data. In *Proc. of ACM EuroSys 2013* (Prague, Czech Republic, Apr 2013).

[55] PILLAI, T. S., ALAGAPPAN, R., LU, L., CHIDAMBARAM, V., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Application crash consistency and performance with ccfs. In *Proc.of USENIX FAST 2017* (Santa Clara, CA, 2017), pp. 181–196.

[56] PRABHAKARAN, V., BAIRAVASUNDARAM, L. N., AGRAWAL, N., GUNAWI, H. S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. IRON File Systems. In *Proc. of ACM SOSP 2005* (Brighton, UK, Oct 2005).

[57] PRABHAKARAN, V., RODEHEFFER, T. L., AND ZHOU, L. Transactional flash. In *Proc. of USENIX OSDI 2008* (Berkeley, CA, USA, 2008), pp. 147–160.

[58] PUROHITH, D., MOHAN, J., AND CHIDAMBARAM, V. The dangers and complexities of sqlite benchmarking. In *Proceedings of the 8th Asia-Pacific Workshop on Systems* (New York, NY, USA, 2017), APSys '17, ACM, pp. 3:1–3:6.

[59] REV, H. SCSI Commands Reference Manual. `http://www.seagate.com/files/staticfiles/support/docs/manual/Interface%20manuals/100293068h.pdf/`, Jul 2014. Seagate.

[60] RODEH, O., BACIK, J., AND MASON, C. Btrfs: The linux b-tree filesystem. *ACM Transactions on Storage (TOS) 9*, 3 (2013).

[61] ROSENBLUM, M., AND OUSTERHOUT, J. K. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems (TOCS) 10*, 1 (Feb. 1992), 26–52.

[62] SEHGAL, P., TARASOV, V., AND ZADOK, E. Evaluating Performance and Energy in File System Server Workloads. In *Proc. of USENIX FAST 2010* (San Jose, CA, USA, Feb 2010).

[63] SELTZER, M. I., GANGER, G. R., MCKUSICK, M. K., SMITH, K. A., SOULES, C. A., AND STEIN, C. A. Journaling Versus Soft Updates: Asynchronous Meta-data Protection in File Systems. In *Proc. of USENIX ATC 2000* (San Diego, CA, USA, Jun 2000).

[64] SHILAMKAR, G. Journal Checksums. `http://wiki.old.lustre.org/images/4/44/Journal-\checksums.pdf`, May 2007.

[65] SWEENEY, A., DOUCETTE, D., HU, W., ANDERSON, C., NISHIMOTO, M., AND PECK, G. Scalability in the xfs file system. In *Proc. of USENIX ATC 1996* (Berkeley, CA, USA, 1996).

[66] TS'O, T. Using Cache barrier in liue of `REQ_FLUSH`. `http://www.spinics.net/lists/linux-ext4/msg49018.html`, September 2015.

[67] TWEEDIE, S. C. Journaling the linux ext2fs filesystem. In *Proc.of The Fourth Annual Linux Expo* (Durham, NC, USA, May 1998).

[68] VERMA, R., MENDEZ, A. A., PARK, S., MANNARSWAMY, S., KELLY, T., AND MORREY, C. Failure-Atomic Updates of Application Data in a Linux File System. In *Proc. of USENIX FAST 2015* (Santa Clara, CA, USA, Feb 2015).

[69] WANG, Y., KAPRITSOS, M., REN, Z., MAHAJAN, P., KIRUBANANDAM, J., ALVISI, L., AND DAHLIN, M. Robustness in the salus scalable block store. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2013), nsdi'13, USENIX Association, pp. 357–370.

[70] WEISS, Z., SUBRAMANIAN, S., SUNDARARAMAN, S., TALAGALA, N., ARPACI-DUSSEAU, A., AND ARPACI-DUSSEAU, R. ANViL: Advanced Virtualization for Modern Non-Volatile Memory Devices. In *Proc. of USENIX FAST 2015* (Santa Clara, CA, USA, Feb 2015).

[71] WILSON, A. The new and improved FileBench. In *Proc. of USENIX FAST 2008* (San Jose, CA, USA, Feb 2008).

[72] XU, Q., SIYAMWALA, H., GHOSH, M., SURI, T., AWASTHI, M., GUZ, Z., SHAYESTEH, A., AND BALAKRISHNAN, V. Performance Analysis of NVMe SSDs and Their Implication on Real World Databases. In *Proc. of ACM SYSTOR 2015* (Haifa, Israel, May 2015).

[73] Y. PARK, S., SEO, E., SHIN, J. Y., MAENG, S., AND LEE, J. Exploiting Internal Parallelism of Flash-based SSDs. *IEEE Computer Architecture Letters(CAL) 9*, 1 (2010), 9–12.

[74] ZHANG, C., WANG, Y., WANG, T., CHEN, R., LIU, D., AND SHAO, Z. Deterministic crash recovery for NAND flash based storage systems. In *Proc. of ACM/EDAC/IEEE DAC 2014* (San Francisco, CA, USA, Jun 2014).