Write Dependency Disentanglement with HORAE

Xiaojian Liao, Youyou Lu, Erci Xu, and Jiwu Shu, *Tsinghua University*

https://www.usenix.org/conference/osdi20/presentation/liao

This paper is included in the Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation

November 4–6, 2020

978-1-939133-19-9

Write Dependency Disentanglement with Horae

Xiaojian Liao, Youyou Lu[*], Erci Xu, and Jiwu Shu[*]

Tsinghua University

Abstract

Storage systems rely on write dependency to achieve atomicity and consistency. However, enforcing write dependency comes at the expense of performance; it concatenates multiple hardware queues into a single logical queue, disables the concurrency of flash storage and serializes the access to isolated devices. Such serialization prevents the storage system from taking full advantage of high-performance drives (e.g., NVMe SSD) and storage arrays.

In this paper, we propose a new IO stack called Horae to alleviate the write dependency overhead for high-performance drives. Horae separates the dependency control from the data flow, and uses a dedicated interface to maintain the write dependency. Further, Horae introduces the joint flush to enable parallel `FLUSH` commands on individual devices, and write redirection to handle dependency loops and parallelize in-place updates. We implement Horae in Linux kernel and demonstrate its effectiveness through a wide variety of workloads. Evaluations show Horae brings up to 1.8× and 2.1× performance gain in MySQL and BlueStore, respectively.

1 Introduction

The storage system has been under constant and fast evolution in recent years. At the device level, high-performance drives, such as NVMe SSD [15], are pushed onto the market with around 5× higher bandwidth and 6× lower latency against their previous generation (e.g., SATA SSD) [11, 18]. From the system perspective, developers are also proposing new ways of storage array organization to boost performance. For example, in BlueStore [23], a state-of-the-art storage backend of Ceph [45], data, metadata and journal can be separately persisted in different, or even dedicated devices.

With drastic changes from the hardware to the software, maintaining the write dependency without severely impacting the performance becomes increasingly challenging. The write dependency indicates a certain order of data blocks to be persisted in the storage medium, and further underlies a variety of techniques (e.g., journaling [44], database transaction [13]) to provide ordering guarantee in the IO stack. Yet, the write order is achieved through an expensive approach, referred as *exclusive IO processing* in this paper. In the exclusive IO processing, the following IO requests can not be processed until the preceding one has been transferred through PCIe bus, then been processed by the device controller and finally returned with a completion response.

Unfortunately, this one-IO-at-a-time fashion of processing conflicts with the high parallelism of the NVMe stack, and further nullifies the concurrency potentials between multiple devices. First, it concatenates the multiple hardware queues of the NVMe SSD, thereby eliminating the concurrent processing of both host- and device-side cores [50]. Moreover, it serializes the access to physically independent drives, preventing the applications from enjoying the benefits of aggregated devices. In our motivation study, we observe that with the scaling of hardware queues and devices, the performance loss introduced by the write dependency can be up to 87%. Conversely, orderless writes without dependency can easily saturate the high bandwidth of NVMe SSDs (§3).

Therefore, to leverage the high bandwidth of NVMe SSDs while preserving dependency, we propose the *barrier translation* (§4) to convert the ordered writes into orderless data blocks and ordering metadata that describes the write dependency. The key idea of barrier translation is shifting the write dependency maintenance to the ordering metadata during normal execution and crash recovery, while concurrently dispatching the orderless data blocks.

We incarnate this idea by re-architecting modern IO stack with Horae (§5). In a nutshell, Horae bifurcates the traditional IO path into two dedicated ones, namely ordered control path and orderless data path. In the control path, Horae flushes ordering metadata directly into the devices' persistent controller memory buffer (CMB), a region of general-purpose read/write memory on the controller of NVMe SSDs [15, 16], using memory-mapped IO (MMIO). On the other hand, Horae reuses classic IO stack (i.e., block layer to device driver to

[*]Jiwu Shu and Youyou Lu are the corresponding authors.
{shujw, luyouyou}@tsinghua.edu.cn

device) to persist orderless writes. Note that this design is also scaling-friendly as orderless data blocks can be processed in both an asynchronous (to a single device) and pipelined (to multiple devices) manner.

Now, with a bifurcated IO path, we further develop a series of techniques to ensure both high performance and consistency in Horae. First, we design *compact ordering metadata* and efficiently organize them in the CMB (§5.2). Second, the *joint flush* of Horae performs parallel FLUSH commands on dependent devices (§5.3). Next, Horae uses the write redirection to break the dependency loops, *parallelizing in-place updates* with strong consistency guarantee (§5.4). Finally, for crash recovery, Horae reloads the ordering metadata, and further only commits the valid data blocks but discarding invalid ones that violate the write order (§5.5).

To quantify the benefits of Horae, we build a kernel file system called HoraeFS to test applications relying on POSIX interfaces, and a user-space object store called HoraeStore for distributed storage (§5.6). We compare HoraeFS against ext4 and BarrierFS [46], resulting in an up to 2.5× and 1.8× speedup at file system and application (e.g., MySQL [13]) level, respectively (§6). We also evaluate HoraeStore against BlueStore [23], showing the transaction processing performance increases by up to 2.1×.

To sum up, we make the following contributions:

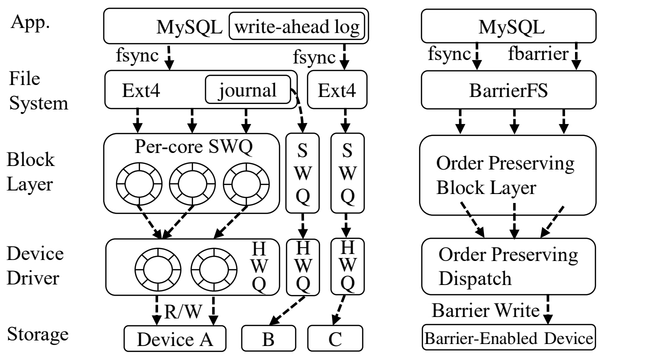
- We perform a study of the write dependency issue on multi-queue drives among both single and multiple devices setup. The results demonstrate considerable overhead of ensuring write dependency.
- We propose the barrier translation to decouple the ordering metadata from the dependent writes to enforce correct write order.
- We present a new IO stack called Horae to disentangle the write dependency of both a single physical device and storage arrays. It introduces a dedicated path to control the write order, and uses joint flush and write redirection to ensure high performance and consistency.
- We adapt a kernel file system and a user-space object store to Horae, and conduct a wide variety of experiments, showing significant performance improvement.

2 Background

This section starts with a brief introduction of enforcing write dependency under current IO stack (§2.1). Then, we illustrate state-of-the-art techniques that alleviate the overhead of enforcing the write dependency (§2.2).

2.1 Basic Ordering Guarantee Approach

In Figure 1(a), we can see that modern IO stack is a combination of software (i.e., the block layer, the device driver) and hardware (i.e., the storage device) layers. Each layer may reorder the write requests for better performance [15, 50] or fairness [4, 29]. Specifically, in the block layer, the host IO scheduler can schedule the requests in the per-core software



(a) Linux IO stack **(b)** Barrier-Enabled IO stack

Figure 1: Existing IO Stacks with Different Order-Preserving Techniques. *SWQ: software queue. In current multi-queue block layer, each core has a software queue. HWQ: hardware queue. The storage device determines the maximum number of HWQs. Emerging NVMe drives usually have multiple HWQs.*

queues based on different algorithms (e.g., deadline). While in the storage device, the controller may fetch and process arbitrary requests due to timeouts and retries.

As a result of this design, the file system must explicitly enforce the storage order. Traditionally, the file system relies on two important steps: *synchronous transfer* and *cache barrier* (e.g., cache FLUSH). First, synchronous transfer requires the file system to process and transfer the dependent data blocks through each layer to the storage interface serially. Then, to further avoid the write reordering by the controller in the storage device layer, the file system issues a cache barrier command (e.g., FLUSH), draining out the data blocks in the volatile embedded buffer to the persistent storage. Afterwards, the file system repeats this processing the next request. Through interleaving dependent requests with exclusive IO processing, the file system ensures the write requests are made durable with the correct order.

The basic approach in guaranteeing the storage order is undoubtedly expensive. It exposes DMA transfer latency (synchronous transfer) and flash block programming delay (cache barrier) to the file system.

2.2 Ordering Guarantee Acceleration

Many techniques [25, 26, 46, 48] improve the basic approach presented in §2.1, by reducing the overhead of synchronous transfer and cache barrier. As barrier-enabled IO stack (BarrierIO [46]) is the closest competitor, we introduce it briefly.

BarrierIO reduces the storage order overhead by preserving the order throughout the entire IO stack. Specifically, the BarrierIO enforces the write dependency mainly using two techniques: the *order-preserving dispatch* to accelerate the synchronous transfer, and the *barrier write* command to improve the cache barrier. First, as shown in Figure 1(b), the order-preserving block layer ensures that the IO scheduler follows the write order specified by the file system. Further, the order-preserving dispatch maintains the write order of

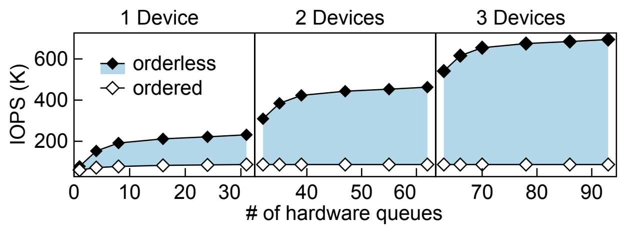


Figure 2: Ordered Write VS. Orderless Write with Varying The Number of Hardware Queues and Devices. *Each device has up to 32 hardware queues. Described in Section 3.1.*

requests queueing in the (single) hardware queue. As a collaborator, the storage controller also fetches and serves the requests in a serialized fashion. Second, BarrierIO replaces the expensive `FLUSH` with a lightweight barrier write command for the storage controller to preserve the write order.

File systems provide the `fsync()` call for applications to order their write requests. Yet, it is still too expensive to preserve order by `fsync()`. Thus, like OptFS [26], BarrierIO separates the ordering from the durability, and further introduces a file system interface, i.e., `fbarrier()`, for ordering guarantee. The `fbarrier()` writes the associated data blocks in order, but returns without durability assurance.

3 Motivation

Multi-queue. The improvement of storage technologies has been continuously pushing forward the performance of solid-state drives. To meet the large internal parallelism of flash storage, SSDs are often equipped with multiple embedded processors and multiple hardware queues [33, 38]. In the host side, as shown in Figure 1(a), the IO stack employs the multicore friendly design. It statically maps the per-core software queues to the hardware queues for concurrent access.
Multi-device. On the other hand, for higher volume capacity, performance and isolation, applications usually stripe data blocks to multiple devices as in RAID 0, or manually isolate different types of data into multiple devices. For example, as shown in Figure 1(a), the ext4 file system uses a dedicated device for journaling processing. The MySQL database redirects its write-ahead logs to a logging device.

The multi-queue and multi-device bring opportunities to enhance performance for independent write requests. Nevertheless, it still remains unknown how much overhead the write dependency introduces to the multi-queue and multi-device design. Here, we first start with a performance study of the write dependency atop multi-queue and multi-device.

3.1 Write Dependency Overhead

In this subsection, we quantify the overhead of write dependency atop Linux IO stack by comparing the IOPS of ordered writes with orderless ones. We use an NVMe SSD (spec in Table 2 Intel 750) with up to 32 hardware queues and use FIO [9] for testing. During the test, we vary the number of hardware queues and attach more devices. Further, we increase the number of threads issuing 4 KB writes to gain the maximum IOPS.

For orderless random write, we use libaio [3] engine with iodepth of 512. In this setup, the write requests issued by libaio have no ordering constraints and can be freely reordered by the storage controller according to NVMe specification [15]. The results are shown in Figure 2. As we enable more hardware queues, the IOPS of orderless writes increases and gradually saturates the storage devices.

For ordered random write, we use libaio engine but set the iodepth to 1. This setup follows the principle of exclusive IO processing in guaranteeing the ordering. As shown in Figure 2, the IOPS of ordered write can hardly grow, even if we use more devices as RAID 0 to serve the write request.

The gap between orderless and ordered writes (blue area in Figure 2) indicates the overhead of the write dependency. With the increase of hardware queues, the overhead becomes more severe, and reaches up to 87%. We conclude that the Linux IO stack is not efficient in handling ordered writes.

3.2 Write Dependency Analysis

In this subsection, we analyze the write dependency overhead via explaining the behaviors of Linux IO stack.

For a single device, the physically independent hardware queues get logically dependent. The application thread firstly puts the write requests in the software queues through the IO interface (e.g., `io_submit()`). Linux IO stack supports various IO schedulers (e.g., BFQ [4]), which perform requests merging/reordering in the software queues. Next, the requests are dispatched to hardware queues, where IO commands are generated. In the hardware queues, out of consideration for hardware performance (e.g., device-side scheduling) and complexity (e.g., request retries), there are no ordering constraints of storage controller processing the commands [15] [1]. Therefore, due to the orderless feature of both types of queues, to guarantee storage order, the IO stack only processes a single request or a set of independent requests at a time. As a result, the ordered write request keeps most queues idle and leaves the multi-queue drives underutilized.

For multiple devices, physically isolated devices get logically connected. The IO stack employs isolated in-memory data structures for the software environment of multi-device. In the hardware side, a device has its private DMA engine and hardware context. Despite the concurrent execution environment, the ordered writes flow through the multi-device in a serialized fashion; the application can not send a request to a different device until the on-going device finishes execution.

[1]In barrier-enabled devices, the host can queue multiple ordered writes and insert a barrier command in between. Such barrier command is available in a few eMMC products [22] with usually single hardware queue. Multi-queue-based NVMe does not have similar concept.

3.3 Limitation of Existing Work

A straightforward solution to remedy aforementioned issues is to keep the entire IO stack ordered as BarrierIO does, so as to allow more ordered writes to stay in the hardware queue. Recall that in Section 2.2, each layer in BarrierIO stack must preserve the request order spread by the upper layer. Implementing such design is quite simple in old drives with a sole command queue (e.g., mobile UFS, SATA SSD): the requests in the hardware queue are serviced in the FIFO way. However, it is quite challenging to extend this idea to multi-queue drives and multiple devices, without sacrificing the multicore designs of the host IO stack and the core/data parallelism of fast storage devices. We explain the reason in detail.

The key of BarrierIO is that each layer agrees on a specific order. While a single hardware queue structure itself can describe the order, for multi-queue and multi-device, the host IO stack must manually specify the order. Maintaining a global order among multiple queues throughout the entire IO stack potentially ruins the multi-queue and multi-device concurrency, which are vital features to fully exploit the bandwidth of high-performance drives, according to our evaluation (§6.2) and a recent study [50]. Further, the firmware design should comply with the host-specified order, which may introduce synchronization among embedded cores and may neutralize the internal core and data parallelism.

In this paper, instead of keeping the IO stack ordered, we seek a new approach that keeps most parts of current IO stack orderless while preserving correct storage order. We now present our design in the following sections.

4 The Horae Foundation

To efficiently utilize the modern fast storage devices, we wish to keep the orderless and independent property of both the software and hardware intact. We achieve this goal via *barrier translation*, which disentangles the write dependency from original slow and ordered write requests.

4.1 Design

Here, we refer a series of ordered write requests issued by the file system or applications as a write stream. Commonly, a write stream can have multiple sets of data blocks and the inbetween barriers that serve as ordering points between two sets of data blocks. We note a set of data blocks to device A with a monotonic set ID x as A_x, and refer a barrier (write dependency) as $\preceq$. Thus, for a write stream $A_x \preceq B_{x+1}$, it shall be ensured that the data blocks of A_x must be made durable prior to ($\prec$) B_{x+1} or at the same time ($=$) as B_{x+1}.

Next, we move on to remove the dependency (i.e., $\preceq$) between two write requests. We use {} to group a set of independent write requests that can be processed concurrently. Our key issue is to translate the $A_x \preceq B_{x+1}$ into $\{A_x, B_{x+1}\}$. We decouple the indexing of a write stream from its data content. The indexing (i.e., ordering metadata) keeps a minimum set of information to retain the dependency. We refer the ordering metadata as $\tilde{A}_x$ and the data content as $\bar{A}_x$, and thus we have $A_x = \tilde{A}_x \cup \bar{A}_x$. Specifically, if A_x has consecutive data blocks of n length to device A from logical block address (lba) m, $\tilde{A}_x$ = {A, m, n}.

Given a write stream $A_x \preceq B_{x+1}$, the *barrier translation* turns it into $\tilde{A}_x \preceq \tilde{B}_{x+1} \preceq \{\bar{A}_x, \bar{B}_{x+1}\}$. Specifically, the translated write stream guarantees the order in two steps: (1) $\{\tilde{A}_x, \tilde{B}_{x+1}\} \preceq \{\bar{A}_x, \bar{B}_{x+1}\}$ and (2) $\tilde{A}_x \preceq \tilde{B}_{x+1}$. First, we must ensure the ordering metadata is made durable no later than data content. Then, the write dependency of original write stream is extracted as the ordering metadata.

4.2 Proof

Now, we further discuss the correctness of barrier translation via the following proof. Our main point is to show that the translated write stream has the same effect on satisfying the ordering constraints. In other words, the following proves that if the ordering metadata is made durable in specific order, and is made durable no later than the data content, the write dependency of original write stream is maintained.

We formalize the implication as follows:

$$\tilde{A}_x \preceq \tilde{B}_{x+1} \preceq \{\bar{A}_x, \bar{B}_{x+1}\} \implies A_x \preceq B_{x+1} \tag{1}$$

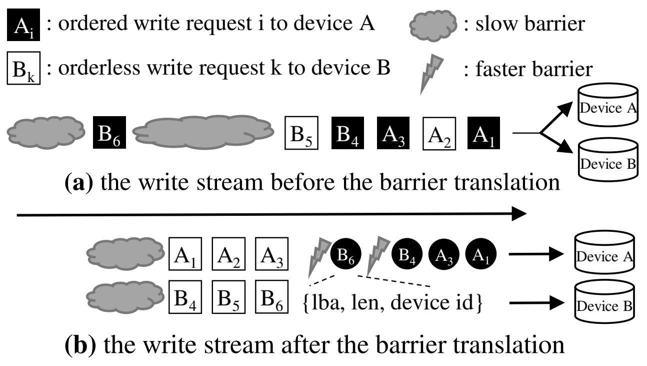
The key lies in the non-deterministic order between $\bar{A}_x$ and $\bar{B}_{x+1}$. We thus discuss the proof in two situations: $\bar{A}_x \preceq \bar{B}_{x+1}$ and $\bar{B}_{x+1} \prec \bar{A}_x$ as follows.

The first case $\bar{A}_x \preceq \bar{B}_{x+1}$. Since we already have $\tilde{A}_x \preceq \tilde{B}_{x+1}$, we have $(\tilde{A}_x \cup \bar{A}_x) \preceq (\tilde{B}_{x+1} \cup \bar{B}_{x+1})$. Because $A_x = \tilde{A}_x \cup \bar{A}_x$ and $B_{x+1} = \tilde{B}_{x+1} \cup \bar{B}_{x+1}$, we have $A_x \preceq B_{x+1}$.

The second case $\bar{B}_{x+1} \prec \bar{A}_x$. This case at first glance may seem to violate the write order. However, since we have $\tilde{B}_{x+1} \preceq \bar{B}_{x+1}$, we can always find and discard the content $\bar{B}_{x+1}$ via the indexing $\tilde{B}_{x+1}$. In this situation, the data content remains empty, i.e., $\emptyset$. The empty result obeys any write dependency, i.e., $\emptyset \in \{A_x \preceq B_{x+1}\} = \{\emptyset, \{A_x\}, \{A_x, B_{x+1}\}\}$.

Two intuitive concerns may be raised from the second case. First, a long write stream may be at a higher risk of losing more data due to discard, although it keeps a consistent state of disk status. However, such scenario is common and acceptable in storage systems. Similar to roll-back and undo log, the discarding time window (e.g., `fsync()` delay) is determined by the file systems or applications. If applications desire data durability rather than ordering, they must synchronously wait for the durability of the write stream, e.g., calling `fsync()`.

Second, a special case of the write dependency, called discarding at a dependency loop, may erase the old but valid data content. For example, consider $A_x \preceq B_{x+1} \preceq A_{x+2}$, where A_x and A_{x+2} operate on the same logical block address. This would occur when storage systems perform in-place updates (IPU). Simply discarding $\bar{A}_x$ in the second case loses the valid data. Our solution is to break the dependency loop by redirecting IPU to another location, i.e., $A_x \preceq B_{x+1} \preceq A'_{x+2}$, where A'_{x+2} targets on a different address from A_x. In this way, we guarantee the correctness of dependency loop as in



A_i : ordered write request i to device A
: slow barrier
B_k : orderless write request k to device B
: faster barrier
B_6 B_5 B_4 A_3 A_2 A_1
Device A
Device B
(a) the write stream before the barrier translation
A_1 A_2 A_3 B_6 B_4 A_3 A_1
B_4 B_5 B_6 {lba, len, device id}
(b) the write stream after the barrier translation

Figure 3: An Example of The Barrier Translation. *The solid circle represents the ordering metadata. The subscript denotes the desired write order of the file system or applications.*

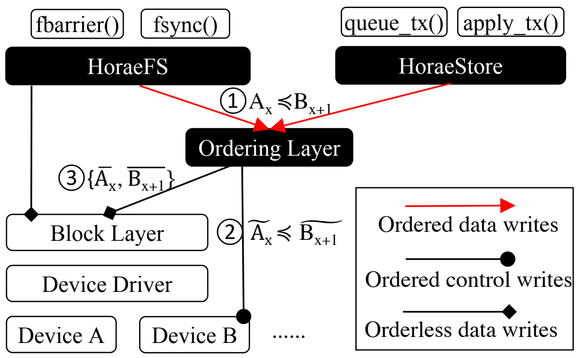


Figure 4: The Horae IO Stack Architecture. Horae *splits the traditional IO path into ordered control path ② and orderless data path ③.* Horae *persists the ordering metadata $\tilde{A}_x$ via the control path before submitting the orderless data blocks $\bar{A}_x$ to the data path.*

normal case, preserving high concurrency as well as the old version of data. We show more details in §5.4.

4.3 Example

We now use an example to go through the entire process of barrier translation. Figure 3(a) presents the original write stream. As we can see, even for different devices, the block-sized (e.g., 4 KB) data is still processed in a serialized fashion. Even worse, the classic approach uses slow and coarse-grained barrier (the cloud shape, e.g., flash `FLUSH`) to enforce the relationship $\preceq$, which may process unrelated data blocks (e.g., the orderless data blocks).

Figure 3(b) shows the output of the barrier translation. Original expensive barriers are replaced with faster and fine-grained barriers (the thunder shape, e.g., memory barrier). Further, the initial ordered writes are translated to orderless ones with no barriers. As a result, after processing the ordering metadata, the devices can process data blocks at their full throttle concurrently.

5 The Horae IO stack

To demonstrate the advantage of the barrier translation, we implement Horae by modifying the classic Linux IO stack. The following sections first give an overview of Horae, and then present the techniques at length.

5.1 Overview

As the high level architecture shown in the Figure 4, Horae extends the generic IO stack with an *ordering layer* targeting ordered writes. Moreover, Horae separates the ordering control from the data flow: the ordering layer translates the ordered data writes (①) into ordering metadata (②) plus the orderless data writes (③). The ordering metadata passes through a dedicated control path (MMIO). The orderless data writes flow through the original block layer and device driver (block IO) as usual. As a result of separation, the data path is no longer bounded by the write dependency, and it thus allows the file systems to dispatch the data blocks to arbitrary hardware queues or storage devices without exclusively occupying the hardware resources.

The key of Horae is the ordering layer, to which the ordering guarantee of the entire IO stack is completely delegated (§5.2). Note that the ordering layer does not need to handle all block IOs, but instead just need to capture the write dependencies of ordered writes. Specifically, Horae stores the ordering metadata in the persistent controller memory buffer (CMB in NVMe spec 1.2 [15], PMR in NVMe spec 1.4 [16]) of the storage device using an ordering queue structure. Horae leverages epochs to group a set of writes with no intra-dependency, and further uses the ordering queue structure itself to reflect the order of each epoch with inter-dependency.

Separating the ordering control path from the data path provides numerous benefits; it saves the block layer, the device driver, and the devices from enforcing write order, which can sacrifice performance or particular property (e.g., scheduling). Further, this design enables Horae to perform parallel `FLUSH`es despite the dependencies among multiple devices (§5.3). Yet, it also faces a challenge, the dependency loop.

As we mentioned in §4.2, the dependency loop occurs when multiple in-place updates (IPU) operate on the same address. As the data path of Horae is totally orderless, multiple ordered in-progress (issued by the file system but the completion response is not returned) IPUs can co-exist and be freely reordered. The dependency loops, if not properly handled, can introduce data version issue (e.g., the former request overwrites the later one) and even the security issue (e.g., unauthorized data access). Horae breaks the dependency loops by write redirection (§5.4). In other words, Horae treats the IPUs as versioned writes, stores their ordering metadata serially, and concurrently redirects them to a pre-reserved disk location. In background, Horae writes the redirected data blocks back to their original destination. In this way, Horae parallels the IPUs while retaining their ordering.

Atop the ordering layer, Horae exports block device abstraction and provides ordering control interface to the upper

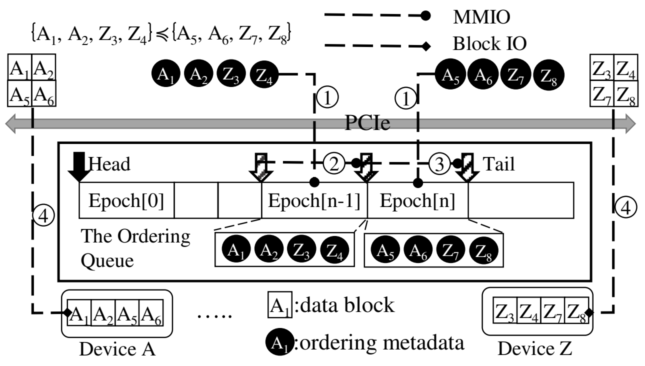


Figure 5: The Circular Ordering Queue Organization. *The ordering queue is located in the persistent controller memory buffer (CMB) of SSD.* Horae *groups a set of independent writes to an epoch, and enforces the ordering between epochs. Described in Section 5.2.*

layer systems (§5.6). We provide APIs (application programming interfaces) and high level porting guidelines for upper layer systems to adapt to Horae. Moreover, we build a kernel file system, HoraeFS, for applications that rely on POSIX file system interfaces, and a user-space object store, HoraeStore, for distributed storage backend.

On top of HoraeFS and HoraeStore, similar to previous works [23,26,46], we provide two interfaces, the ordering and durability interface, for upper layer systems or applications to organize the ordered data flow. The ordering interfaces (i.e., `fbarrier()`, `queue_tx()`) send the data blocks to the storage with ordering guarantee, but return without ensuring durability. The durability interfaces (i.e., `fsync()`, `apply_tx()`) deliver the intact semantics as before.

5.2 Ordering Guarantee

The major role of the ordering layer is guaranteeing the write order. Recall that the write stream $A_x \preceq B_{x+1}$ is translated to $\tilde{A_x} \preceq \tilde{B_{x+1}} \preceq \{\bar{A_x}, \bar{B_{x+1}}\}$, thus the ordering layer enforces the write dependency of the ordering metadata before dispatching the translated data blocks. We first present the organization of the ordering metadata.

Ordering metadata organization. As shown in the center of Figure 5, through the PCIe base address register, Horae uses the persistent Controller Memory Buffer (CMB) as the persistent circular ordering queue. The ordering queue is bounded by the `head` and `tail` pointers, and stores the ordering metadata of one write stream. For an incoming ordered write request, Horae first appends its ordering metadata to the queue via MMIO. As shown in Figure 6, the ordering metadata is compact, mainly consisting of range-based destination (i.e., `lba`, `len`, `devid`). Storing the ordering metadata via control path is a CPU-to-device transfer with byte-addressability; unlike an interrupt-based memory-to-device DMA transfer, it does not transfer a full block nor switch context. Thus, persisting the compact ordering metadata via MMIO is efficient.

Horae leverages the epoch to group a set of independent writes. And, Horae only enforces the write dependency in

Format:	lba	len	devid	etag	dr	plba	rsvd
Size in bits:	32	32	8	1	1	32	22

Figure 6: The Ordering Metadata Format. *lba: logical block address. len: length of continuous data blocks. devid: destination device ID. etag: epoch boundary. dr: is made durable. plba: lba of prepare write. rsvd: reserved bits.*

the unit of epoch. To realize epoch, Horae uses the `etag` to indicate the boundary of epochs. The `etag` implies $\preceq$.

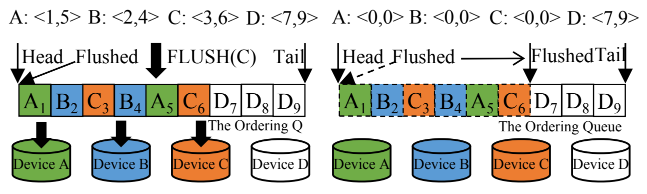
Now, we go through an example presented in Figure 5 to show how data blocks reach the storage with ordering constraints. Suppose that the ordering layer receives two sets of ordered write requests from two threads with ordering constraints $\{A_1, A_2, Z_3, Z_4\} \preceq \{A_5, A_6, Z_7, Z_8\}$. The ordering layer first forms two epochs `N` and `N-1`, and constructs the ordering metadata according to the data blocks and write dependencies. Next, the two threads store the ordering metadata concurrently to the ordering queue via MMIO (①). Then, Horae updates the 8-byte `tail` pointer sequentially to ensure both the update atomicity of each epoch and the write order of associated ordering metadata (②, ③). Finally, the two threads dispatch the orderless writes concurrently via block IO interface (④).

Since the size of available CMB is usually limited (e.g., 2 MB), the ordering queue may exceed the CMB region. Thus, Horae introduces two operations, swap and dequeue, to reclaim free space of the CMB for incoming requests.

Swap. The ordering layer blocks the threads issuing ordered writes, and then invokes swap operation, when the total size of the valid ordering metadata exceeds the queue capacity. It moves the valid ordering metadata to a checkpoint region with larger capacity, e.g., a portion of flash storage. It first waits for the on-going append operations on the ordering queue to complete. Next, it copies the whole valid ordering metadata to the checkpoint region. Finally, it updates the `head` and `tail` pointers atomically with a lightweight journal.

Dequeue. The ordering layer dequeues the expired ordering metadata when its associated data blocks and the preceding ones are durable. The dequeue operation moves the 8-byte `head` pointer.

Optimization. We observe that the MMIO write latency through PCIe is acceptable, but MMIO read can be extremely slow (8 B read costs 0.9us, 4 KB read costs 113us). This is because MMIO read is split into small-sized (determined by CPU) non-posted read transactions to guarantee atomicity [17]. Yet, MMIO reads can be abundant on the CMB. For example, Horae allocates free locations from the ordering queue before sending the ordering metadata, which requires frequent `head` and `tail` pointer access. Also, Horae needs to read the whole ordering queue for a swap operation. Horae avoids slow MMIO reads by maintaining an in-memory write-through cache for the entire CMB. The cache serves all read operations in memory. Write operations are performed to the cache, and persisted to the device CMB simultaneously via



(a) Pre-FLUSH states **(b)** Post-FLUSH states

Figure 7: The Durability and Joint FLUSH of Horae. *A FLUSH command to a single device also triggers flushings to other devices whose write requests must be made durable in advance.*

MMIO. The extra memory consumption (2 MB) is negligible.

5.3 Durability Guarantee

Applications may also require instant durability. Linux IO stack uses the FLUSH command (e.g., calling `fsync()`) to enforce durability and ordering of updates in a storage device. Further, the FLUSH serves as a barrier between multiple devices, so as to ensure that the post-FLUSH requests are not made durable prior to the pre-FLUSH ones. For example, to guarantee the durability of $A_x \preceq B_{x+1}$, Linux IO stack issues two FLUSHes. The first one on device A ensures the write dependency ($\preceq$) as well as the durability of A_x, and the second one on device B for durability of B_{x+1}.

The FLUSH of Horae no longer serves as a barrier. Thus, Horae eliminates intermediate FLUSHes and invokes an eventual FLUSH to ensure durability. Unlike the legacy FLUSH targeting on sole device, the FLUSH of Horae, called joint FLUSH, automatically flushes related devices whose data blocks should be persisted in advance.

Figure 7 shows an example of the joint FLUSH in detail. The states of the ordering queue are in Figure 7a, and suppose we decide to flush device C. The ordering layer firstly finds the flush point (6), the last position of the flushed device in the ordering queue. Next, it identifies the devices that need to be flushed simultaneously. A device should be flushed if it has requests prior to the flush point. To quickly locate the flush candidates, Horae keeps the first and last position of each device (device range) in the ordering queue (e.g., $\langle 1,5 \rangle$ of device A), as shown in the top of Figure 7a. By checking the existence of the intersection of the device range and head-to-flush range ($\langle 1,6 \rangle$), Horae selects the flush candidates. Then, Horae sends FLUSH commands to the candidate devices (A and B) simultaneously. When the joint FLUSH completes, Horae moves the `flushed` pointer and resets the device range, as shown in Figure 7b.

With the `flushed` pointer, Horae ensures the durability of the write requests between the `head` and `flushed` position. However, on SSDs with power loss protection (PLP), data blocks are guaranteed to durable when they reach the storage buffer, prior to a FLUSH command. Therefore, Horae uses the `dr` bit (shown in Figure 6) to indicate the durability of the requests after the `flushed` position. For SSD with PLP, Horae sets the `dr` bit, once the ordered write requests are completed via interrupt handler or polling.

While the legacy FLUSH is always performed in a serialized and synchronous fashion, the joint FLUSH enables Horae to flush concurrently and asynchronously, for devices without PLP. These devices expose extremely long flash programming latency, so async flushings on them exploit the potential parallelism. However, Horae remains the sync flushing on the other type of devices with PLP. Since flushing such devices is returned from the block layer directly, async flushing incurs unnecessary context switches from the wakeup mechanism.

5.4 Handling Dependency Loops

To understand the motivation for resolving the dependency loops, we show two examples. First, consider a data block is overwritten repeatedly. Reordering of two overwrite operations may cause the later one to be overwritten by the former one, and results in a data version issue. Second, consider at the file system level. The file system frees a data block from owner A, and then reallocates it to owner B. Reordering of reallocating and freeing upon a sudden crash can cause a security issue: owner B can see the data content of owner A.

Classic IO stacks handle dependency loops by prohibiting multiple in-progress IPUs on the same address. IPUs on the same address are operated exclusively, where the next IPU can not be submitted until the preceding one is completely durable. However, this approach serializes the access to the same address, leaving the device underutilized. Horae allows multiple in-progress parallel IPUs on the same address, and resolves dependency loops through IPU detection, prepare and commit write.

IPU detection. The foremost issue is to detect IPUs. In Horae, the upper layer systems must specify the IPU explicitly because of the awareness of IPU.

Prepare write. In receiving an IPU, Horae first allocates an available location from the preparatory area (p-area), a pre-reserved area of each device for handling dependency loops. It stores the location in the `plba` region (shown in Figure 6) of the ordering metadata. Next, it dispatches the IPU to the `plba` position of p-area, via the same routine as the classic orderless write. Further, Horae uses an in-memory IPU radix tree to record the `plba` for following read operations to retrieve the latest data. The IPU tree accepts the logical block address (`lba`) as the input and outputs the newest `plba`. Compared to the IO processing, the modification on the IPU tree is performed in memory, and its overhead is thus negligible.

Commit write. Horae applies the effects of IPUs via the commit write, once the durability of the prepare write is satisfied. Horae firstly scans the ordering queue, and merges the overlapping data blocks of the p-area. Then, it writes the merged data blocks back to their original destination concurrently. When the commit write completes, Horae removes associated entries of the IPU tree, and dequeues the entries between the `head` and `flushed` pointer of the ordering queue.

Horae introduces two commit write policies, lazy and ea-

API	Explanation
olayer_init_stream(sid, param)	Register an ordered write stream with ID `sid` and parameters `param`
olayer_submit_bio(bio, sid)	Submit an ordered block IO `bio` to stream `sid`
olayer_submit_bh(bh, op, opflags, sid)	Submit a buffer head `bh` to stream `sid` with specific flags `opflags` and `op`
olayer_submit_bio_ipu(bio, sid)	Submit an ordered in-place update block IO `bio` to stream `sid`
olayer_blkdev_issue_flush(bdev, gfp_mask, error_sector, sid)	Issue a joint `FLUSH` to device `bdev` and stream `sid`
fbarrier(fd)	Write the data blocks and file system metadata of file `fd` in order
fdatabarrier(fd)	Write the data blocks of file `fd` in order
io_setup(nr_events, io_ctx, sids)	Create an asynchronous I/O context `io_ctx` and a set of streams `sids`
io_submit_order(io_ctx, nr, iocb, sid)	Write `nr` data blocks defined in `iocb` to stream `sid`

Table 1: The APIs of Horae. Horae *provides the **stream** (or sequencer) abstraction for upper layer systems. Each stream is identified by a unique* `sid`*, and represents a sequence of ordered IO requests. To realize multiple streams,* Horae *evenly partitions the CMB area and p-area, and assigns a portion to each stream.*

ger commit write. The lazy commit write is performed in background when the IO stack is idle. When Horae runs out of p-area space, it triggers eager commit write.

Read. As the ordered IPUs are redirected to p-area, the following read operation retrieves the latest data blocks from p-area first. Horae searches the IPU tree for the `plba`, and reads the data from the `plba` position of p-area.

With write redirection for IPUs, Horae provides higher consistency than the data consistency, which matches the default ordered mode of ext4. The data consistency requires that (1) the file system metadata does not point to garbage data, and (2) the old data blocks do not overwrite new ones. Horae is able to preserve the order between data and file system metadata, and thus the file system metadata always points to valid data. Also, Horae controls the order of parallel IPUs. Therefore, Horae supports data consistency.

However, the data consistency does not provide request atomicity. Under data consistency, the IPUs directly overwrite valid data blocks, and can sometimes leave the file system a combination of old and new data, which brings inconvenience to maintain application-level consistency [2]. This is because commodity storage does not provide atomic write operations. For a write request containing a 4 KB data block, it may be split into multiple 512 B (determined partially by `Max_Payload_Size`) PCIe atomic ops (i.e., transactions) [17]. A crash may result in partially updated 4 KB data block.

Horae enhances data consistency with request atomicity; continuous data blocks of each write request sent to the Horae are made durably visible atomically due to double write. Once the durability of the prepare write is satisfied, Horae ensures request atomicity. Otherwise, the prepare write may be partially completed and its effect is thus ignored.

5.5 Crash Consistency

In the face of a sudden crash, Horae must be able to recover the system to a correct state that the data blocks are persisted in the correct order and the already durable data blocks with a completion response are not lost. This subsection discusses the crash consistency of Horae, including the ordering queue consistency and the data block consistency.

The ordering queue. As stated in §5.2, Horae writes the ordering metadata via MMIO. But MMIO writes are not guaranteed to be durable because they are posted transactions without completions. After each MMIO write, Horae issues a non-posted MMIO read of zero byte length to ensure prior writes are made durable [24].

Upon a power outage, the ordering queue, the `head`, `flushed` and `tail` pointers, are saved to a backup region of flash memory, with the assistance of capacitors inside the SSD. When power resumes, Horae loads the backup region into the CMB.

The data block. Horae recovers the storage to a correct state with the support of the ordering queue. Horae scans the ordering queue from the `head` position to the `flushed` position, in case of the data blocks are made durable with the `FLUSH` response but expired entries are not dequeued. Horae commits the data blocks of the p-area that obey the order.

Further, Horae recovers the durable yet not flushed data blocks (e.g., in SSD with PLP). Starting from the `flushed` position, Horae sequentially scans the ordering queue, and commits the prepare writes until it finds a non-durable data block (i.e., `dr` bit is not set). After that, Horae discards the following data blocks through filling the blocks with "zeros", because they violate the write order.

5.6 The API of Horae and Use Cases

This subsection first describes the API of Horae, and then presents two use cases: a file system leveraging a single write stream (i.e., a single ordering queue), and a user-space object store running with multiple write streams.

5.6.1 The API of Horae

To enable the developers to leverage the efficient ordering control of Horae, we provide three levels of functionalities/APIs shown in Table 1: the kernel block device, the file system and the asynchronous IO interface (i.e., libaio [3]).

Kernel block device interface. The kernel systems (e.g., file systems) can use `olayer_init_stream` to initialize an ordered write stream for further use. `olayer_submit_bio` and `olayer_submit_bh` deliver the same block IO submission function as classic interfaces (i.e., `submit_bio` and `submit_bh`); but they return when associated ordering metadata are persisted in the `sid` ordering queue (§5.2). `olayer_submit_bio_ipu` is for in-place updates (§5.4). `olayer_blkdev_issue_flush` is extended from classic `FLUSH` interface (i.e., `blkdev_issue_flush`); it keeps the same arguments and performs joint `FLUSH` on a given stream and target devices (§5.3).

File system interface. We offer two ordering file system interfaces, the `fbarrier` and `fdatabarrier`, for upper layer systems or applications to organize their ordered data flow. The `fbarrier` bears the same semantics as the `osync` of OptFS and `fbarrier` of BarrierFS; it writes the data blocks and journaled metadata blocks in order but returns without ensuring durability. The `fdatabarrier` is the same as that of BarrierFS; it only ensures the ordering of the application data blocks but not the journaled blocks. We further show the internals of these interfaces in the following §5.6.2.

Libaio interface. Libaio provides wrapper functions for async IO system calls, which are used for some systems (e.g., BlueStore and KVell [36]) designed on high-performance drives. We expose the ordering control path of Horae via two new interfaces on libaio. `io_setup` allows developers to allocate a set of streams defined in `sids`. `io_submit_order` performs ordered IO submission; it bears the same semantics of `fdatabarrier`. We further show the usage of these interfaces in boosting BlueStore in §5.6.3.

Porting guidelines. We provide three guidelines. First, upper layer systems can distinguish the ordered writes from the orderless ones based on the categories of the request, and send them using Horae's APIs. The requests that contain metadata, the write-ahead log and the data of eager persistence (e.g., data specified by the `fsync()` thread) are treated as ordered writes. Second, due to the separation of ordering control path, upper layer system can design and implement the ordering and durability logic individually. In the ordering logic, they can use the ordering control interface (e.g., `olayer_submit_bio`) to dispatch the following ordered writes immediately after the previous one returns from the control path. This allows the ordered data blocks to be transferred in an asynchronous manner without waiting for the completion of DMA. Third, they can remove all `FLUSH`es that serve as ordering points in the ordering logic, and invoke an eventual joint `FLUSH` in the durability logic to guarantee durability.

5.6.2 The HoraeFS File System

We build HoraeFS atop the Horae with a set of modifications of BarrierFS [46]. BarrierFS builds on Ext4, and divides the journaling thread (i.e., JBD2 in Figure 8) into submit thread and flush thread.

HoraeFS inherits this design, and the major changes are that (1) we submit ordered writes and reads to the ordering layer first, (2) we remove the `FLUSH` to coordinate the data and journal device and (3) we detect IPUs through inspecting

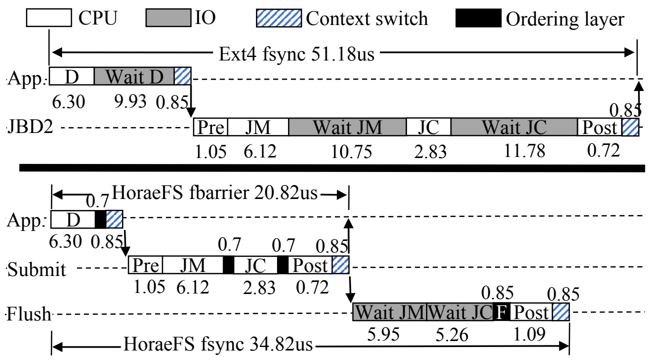


Figure 8: The `fsync()` and `fbarrier()`. *4 KB data size. The number shows the latency of each operation in microseconds. D: application data blocks. Pre: prepare the journaled metadata. JM: journaled file system metadata. JC: journaled commit record. Post: change transaction state, calculate journal stats, etc.*

the `BH_New` states of each write. Although the journal area is repeatedly overwritten, we do not treat the journaled writes as IPUs, as the journal area is always cleaned before reused.

Figure 8 shows a side-by-side comparison between Ext4 and Horae on a NVMe SSD. For each 4 KB allocating write in most cases, both file systems issue three data blocks, namely the data block (*D*), the journaled metadata block (*JM*) and the journaled commit block (*JC*). Further, both file systems order the data blocks with $\{D, JM\} \leq JC$. Ext4 enforces the ordering constraints through the exclusive IO processing. It waits for the completion of preceding data blocks (e.g., Wait JM) and issues a `FLUSH` (not shown in the figure because it is returned by the block layer quickly). HoraeFS eliminates the exclusive IO processing, and preserves the order through the ordering layer, as shown in the black rectangle. HoraeFS waits for the durability of the associated blocks in flush thread, and finally issues a `FLUSH` to the ordering layer for durability.

HoraeFS differs from BarrierFS in the IO dispatching (i.e., the white rectangle) and IO waiting (i.e., the gray rectangle) phase. During IO dispatching phase, BarrierFS passes through the entire order-preserving SCSI software stack. Besides, BarrierFS experiences extra waiting time due to the order-preserving hardware write, i.e., the barrier write.

5.6.3 The HoraeStore Distributed Storage Backend

We build HoraeStore atop the Horae with a set of modifications of BlueStore [23], an object store engine of Ceph.

BlueStore directly manages the block device by async IO interfaces (e.g., libaio), providing transaction interfaces for distributed object storage (i.e., RADOS). Inside each write transaction, it first persists the aligned write to data storage, followed by storing the unaligned small writes and metadata in a RocksDB [1] KV transaction (KVTXN). The RocksDB first writes the write-ahead log (WAL), then applies the updates to the KV pairs. For inter-transaction ordering, it uses sequencers; the next transaction can not start a KVTXN until the preceding one has made the aligned write durable and

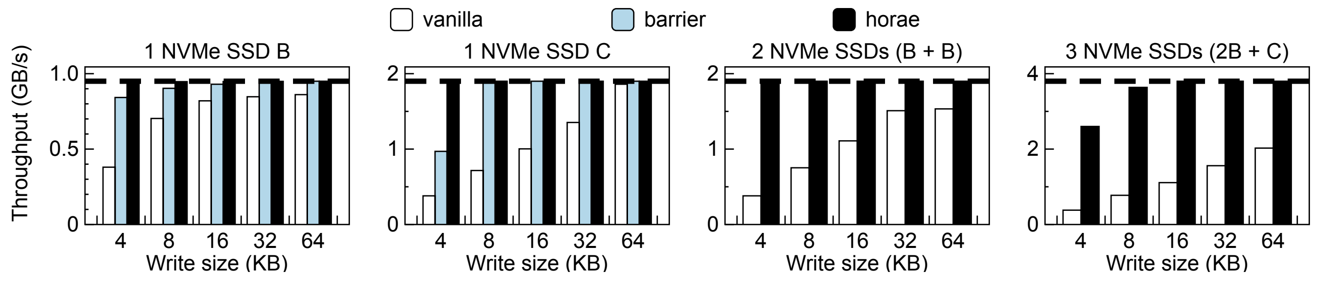


Figure 9: Ordered Write Performance. *The throughput of randomly writing 1 GB drive space by 1 write stream (i.e., 1 thread). vanilla: native Linux NVMe IO stack. horae:* Horae *IO stack. barrier: Barrier-enabled IO stack. Horizontal dotted lines: maximum device bandwidth.*

started its KVTXN. Besides, BlueStore uses a single KV thread to serially perform KVTXNs, which blocks and waits for in-progress KVTXNs to become durable.

HoraeStore accelerates the transaction ordering guarantee, while exporting the same transaction interfaces. For intra-transaction ordering, HoraeStore uses the new async IO interface of Horae, `io_submit_order()`, to control the write order of data, WAL and KV pairs. With this new interface, for each transaction, HoraeStore processes the data, WAL and KV pairs concurrently.

For inter-transaction ordering, HoraeStore extends the overlapping range of dependent transactions, and improves the concurrency of KVTXNs' submission. First, in HoraeStore, the following transaction starts the KVTXN immediately after the preceding one has satisfied the ordering of its aligned writes (D) and KVTXN. In other words, Horae can process two dependent transactions concurrently, once the ordering between them is satisfied, i.e., $\{D_1, D_2\} \leq KVTXN_1 \leq KVTXN_2$. Second, HoraeStore separates the ordering of KVTXN from durability; it starts the KVTXNs in a KV submit thread for ordering, and ensures the durability in a KV flush thread. Hence, more KVTXNs can be queued in the KV submit thread, and can further be dispatched to RocksDB for processing.

5.7 Implementation Details and Discussion

We implement Horae in Linux kernel as a pluggable kernel module (i.e., the ordering layer), consisting of 1288 lines of code (LOC); no changes are needed for traditional IO stack (i.e., the block layer, NVMe and SCSI driver). HoraeFS is implemented based on BarrierFS with approximately 100 LOC changes. HoraeStore is implemented based on BlueStore with around 200 LOC change.

Horae needs a region of byte-addressable persistent memory for efficient ordering control path. We realize this by remapping the CMB region of a capacitor-backed CMB-enabled SSD from StarBlaze [20] using `ioremap_wc()`.

Currently, CMB-enabled SSDs are already available in the market [14, 18]. Moreover, many SSDs have enabled power loss protection [10, 12, 18, 32]. Therefore, the persistent CMB (or PMR) requirement of Horae can be achieved easily. We further discuss the alternatives of the CMB in §7.

6 Evaluation

This section evaluates the Horae, HoraeFS and HoraeStore by answering the following questions:

- What is the performance of Horae in guaranteeing the ordering? (§6.2, §5.2)
- What is the performance of Horae in guaranteeing the durability? (§6.3, §5.3)
- How does Horae perform under in-place updates with different consistency level? (§6.4, §5.4)
- Can Horae recover correctly after a crash and how much overhead does recovery introduce? (§6.5, §5.5)
- How much improvement does Horae bring to applications? (§6.6, §5.6)

6.1 Experimental Setup

Hardware. We conduct all experiments with a 12-core machine running at 2.50 GHZ. Table 2 shows the specification of the candidate SSDs. We use three broadly categorized SSDs: the SATA SSD (labelled as A), the consumer-grade NVMe SSD (B), and the high-performance datacenter-grade NVMe SSD (C). The NVMe SSD B and C are with PLP. The size of CMB used by Horae is 2 MB.

Compared Systems. We mainly compare with two types of IO stacks, Vanilla and BarrierIO [46]. Vanilla is the default Linux IO stack. Ext4 [7] is a journaling file system running upon vanilla. We setup ext4 with default options in ordered journaling mode (denoted as ext4-DR). We disable the barriers in ext4-DR (`nobarrier` option) with ext4-OD, which only guarantees the dispatch order reaching in storage buffer (not storage medium). Similar to ext4, we test BarrierFS [46] upon BarrierIO stack with durability guarantee (denoted as BFS-DR) and ordering guarantee (denoted as BFS-OD). Since we do not have barrier compliant storage

	Model	Seq. Bandwidth	Rand. IOPS (8GB span)
A	Samsung 860 PRO SATA	Read: 560MB/s Write: 530MB/s	Read: 100k Write: 90k
B	Intel 750 NVMe	Read: 2200MB/s Write: 950MB/s	Read: 430K Write: 230K
C	Intel DC P3700 NVMe	Read: 2800MB/s Write: 1900MB/s	Read: 640K Write: 475K

Table 2: SSD Specifications.

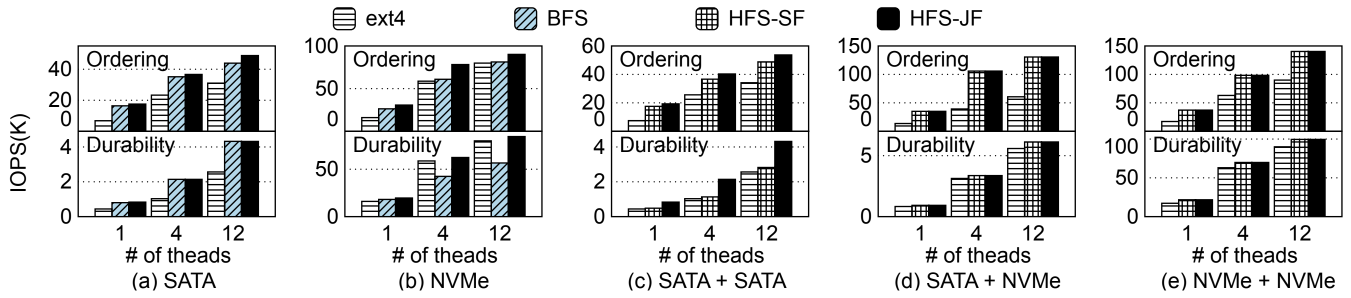


Figure 10: File System Performance under FIO Allocating Write Workload. *Ordering:* `nobarrier` *option of ext4, but only guarantees a relaxed ordering of reaching the storage write buffer.* `fbarrier()` *of BarrierFS and* HoraeFS. *Durability:* `fsync()`. *SF: serialized* `FLUSH`. *JF: joint* `FLUSH`.

devices, we reuse the software of BarrierFS and add extra 5% overhead of the hardware barrier write, following the assumption of the BarrierFS paper. To run BarrierFS correctly in multi-queue drives, we modify the NVMe driver to setup only one IO command queue. HoraeFS (abbreviated as HFS) uses just one ordering queue.

6.2 Basic Performance Evaluation

First, we demonstrate the effectiveness of Horae in guarantee the ordering (§5.2), through measuring the throughput of three IO stacks on block devices. We vary the number of devices and organize them as soft-RAID 0. Specifically, we evenly distribute X KB random writes to different devices in a round-robin fashion. Figure 9 shows the overall throughput of ordering guarantee with varying the write size. Note that we only have the result of BarrierIO on a single device, because it does not support multiple drives.

Result. From the result, we find that Horae outperforms vanilla and BarrerIO IO stack by 4.1× and 2× respectively in the case of a single device and 4 KB write unit. On multiple devices, Horae achieves up to 6.8× throughput than vanilla. Further, we observe that Horae can easily saturate the device bandwidth with small write units (e.g., 4 KB).

Analysis. We now decompose the IO path to better understand the performance. The overall IO path can be broken down into four parts, and we measure the overhead of each part when issuing 4 KB data blocks as follows: (1) data page preparation costs 0.8 us in our test; (2) the ordering layer processes and writes ordering metadata within 0.7 us; (3) about 1.0 us is spent on block layer, which performs request merging and `bio`(block IO data structure)-to-`request`(NVMe driver data structure) transmission; (4) data DMA, device-side processing and interrupt handler consume 8.7 us. The classic approach experiences all parts except (2), which counts up to 10.5 us in total. Thus, the maximum IOPS and throughput that a single write stream can achieve in classic IO stack are 95K and 380 MB/s. Horae experiences (1) and (2) in most cases. The ordering layer delegates the submission of orderless block IO to per-CPU background submitter threads, so as to hide the overhead of block layer for foreground ordering calls. Thus, Horae can achieve up to 2.6 GB/s in the case of 4 KB writes. BarrierIO eliminates (4) in ordering guarantee, and thus can achieve up to 2.2 GB/s in NVMe stack theoretically. However, we observe that configuring the drive to a single IO command queue considerably decreases the available bandwidth of high-performance storage.

6.3 File System Evaluation

In this subsection, we evaluate the performance of POSIX file systems atop different IO stacks with varying the number and type of storage devices. Moreover, we demonstrate the effectiveness of Horae in guaranteeing the durability (§5.3). We perform allocating write so that the file system always finds the updated metadata in the journal. We set up the file system with two modes: the internal journal that mixes data and journal blocks in a single device, and the external journal that uses a dedicated device for locating journal. The result is shown in Figure 10, and we make the following observations.

Effect of removing flush. As shown in Figure 10(a), on SATA SSD, HFS achieves 80% higher IOPS averagely against ext4, and exhibits comparable performance compared to BFS. Here, the major overhead is two `FLUSH`es. The former one is used to control the write order of the data blocks and the commit record, and the later one is to ensure durability. The `FLUSH` of SATA SSD exposes raw flash programming delay (1 ms). In BFS and HFS, the former `FLUSH` is eliminated.

Effect of async DMA. As shown in Figure 10(b), on NVMe SSD, HFS achieves 22% higher IOPS than ext4. Compared to the durability of HFS, the ordering guarantee of HFS can further boost IOPS by 57%. The major overhead here is shifted to the DMA transfer. Figure 8 shows the source of improvement, and the number next to each rectangle shows the single thread latency of each operation. Due to the separation of the ordering and durability, HFS and BFS can overlap the DMA transfer with CPU processing, and thus partly hide the DMA delay. BFS does not perform well on NVMe SSD due to the restriction of the single IO command queue, especially when we increase the number of threads.

Effect of joint flush. As shown in Figure 10(c), HFS outperforms ext4 by 88% and 90% on average in durability and ordering respectively. Comparing HFS-PF to HFS-SF, we find the joint flush improves the overall performance by up to

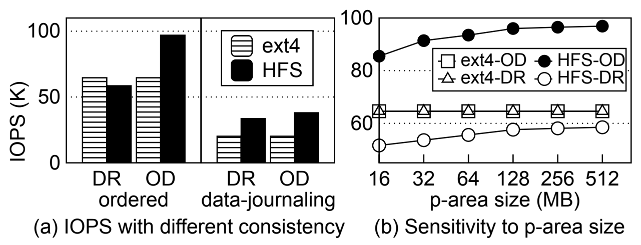


Figure 11: In-Place Update Performance. *OD:* `nobarrier` *option of ext4,* `fbarrier()` *of HFS. DR:* `fsync()`*. Test on SSD C.*

70%. This is because joint flush allows physically independent devices to perform flushing concurrently.

Effect of parallel device access. Figure 10(d-e) plots the result when we use an NVMe SSD to accelerate the journal. In ordering, HFS improves ext4 by 150% and 76% respectively. The major contributor here is the parallel device access. In HFS, once the associated ordering metadata is processed serially by the control path, the data blocks can be transferred and processed by individual devices concurrently. While in ext4, this is done in a serialized manner.

6.4 In-Place Update Evaluation

In this subsection, we evaluate the performance of in-place update under different consistency and with varying the size of the preparatory area (p-area) (§5.4).

As shown in the X title of Figure 11(a), we first setup the file system with two modes, the ordered and data-journaling mode, representing data and version consistency, respectively. Then, we issue 10 GB overwrites to a 10 GB file. The ordered mode performs metadata journal. The data-journaling mode performs data journal to achieve the version consistency that the version of data matches that of metadata.

In the ordered mode, the double write of eager commit write enhances the data consistency at the cost of 10% IOPS loss in durability. In ordering, HFS exhibits 50% higher IOPS compared to ext4, because HFS can emit multiple IPUs simultaneously without interleaving each IPU with DMA transfer and `FLUSH` command.

In the data-journaling mode, both file systems first put the IPU to journal area. When performing journal, HFS dispatches the commit record immediately after the journaled data, and thus provides 60% higher IOPS on average.

When Horae runs out of p-area space, Horae blocks incoming requests and triggers eager commit write. To investigate the performance of Horae in such a situation, we run the same IPU workload with the scaling of p-area size. The results are shown in Figure 11(b). We find that HFS-OD performs dramatically better than ext4-OD even with small p-area. To provide request atomicity, HFS-DR delivers less IOPS than ext4-DR. As we enlarge the p-area, the IOPS gap narrows.

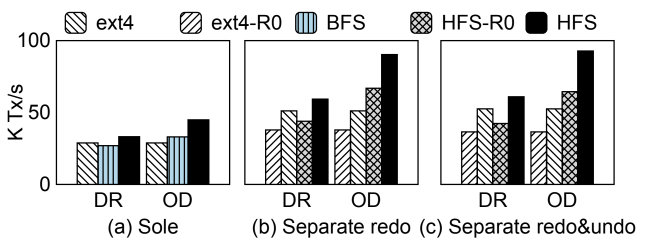


Figure 12: MySQL under OLTP-insert Workload. *(a) Sole: mix data, redo log and undo log in device B. (b) Separate redo: data and undo log to device B, redo log to device C. (c) Separate redo & undo: data to device B, redo log to device C, undo log to another device B. DR: The* `fsync()` *used to control the write order of transactions is replaced with* `fbarrier()`*. OD: All* `fsync()`*s are replaced with* `fbarrier()`*s.* `Sync()` *the database every 1 second. R0: organize underlying devices as logical volumes using RAID 0.*

6.5 Crash Recovery Evaluation

To verify the consistency guarantees of Horae (§5.5), we run workloads, and forcibly shut down the machine at random. We restart the machine and measure the recovery performance. We choose Varmail workload of Filebench [8] for its intensive `fsync()`. Varmail contains two `fsync()`s in each flow loop, and we replace the first one with `fbarrier()`.

We repeat the test 30 times, and observe HoraeFS can always recover to a consistent state. The recovery time comes from two main parts: the ordering queue load time and commit write time. First, Horae loads the pointers and the ordering metadata into host DRAM, which consumes 29.8 ms on average. Next, the commit write requires "read-merge-write", and costs 497.6 ms on average.

6.6 Application Evaluation

6.6.1 MySQL

We evaluate MySQL with OLTP-insert workload [21]. The setups are described in the caption of Figure 12.

In sole configuration, HFS-DR outperforms ext4-DR and BFS-DR by 15% and 23% respectively. In ordering, HFS-OD prevails ext4-OD by 56% and achieves 36% higher TPS than BFS-OD. This evidences that HFS is more efficient in controlling the write order.

When using dedicated devices to store redo and undo logs (i.e., Figure 12(b)), HFS-DR outperforms ext4-DR by 16%, and HFS-OD performs 76% better than ext4-OD. This is because HFS can parallelize the IOs to individual devices.

Comparing Figure 12(b) with (c), we find that separating undo logs does not bring much improvement in both ext4 and HFS. Undo logs perform logical logging to retain the old version of database tables, which incurs less writes compared to physical logging (redo log). MySQL tightly embeds the undo logs in the table files, thus separating undo logs does not alleviate the write traffic to the data device.

Comparing HFS with HFS-R0, we observe that, from the performance aspect, manually distributing data flows to de-

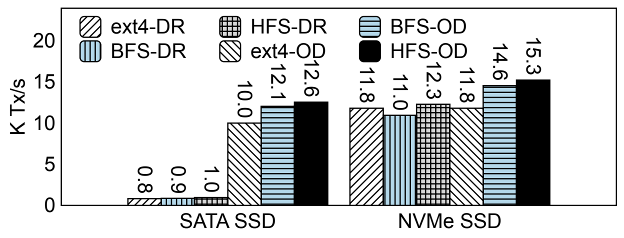


Figure 13: SQLite Random Insert Performance. *SQLite runs at WAL mode. 1M inserts in random key order. Key size 16 bytes, value size 100 bytes. DR: The first three* `fdatasync()`*s used to control storage order of transactions are replaced with* `fdatabarrier()`*s, but the last one remains intact. OD: All* `fdatasync()`*s are replaced with* `fdatabarrier()`*s.*

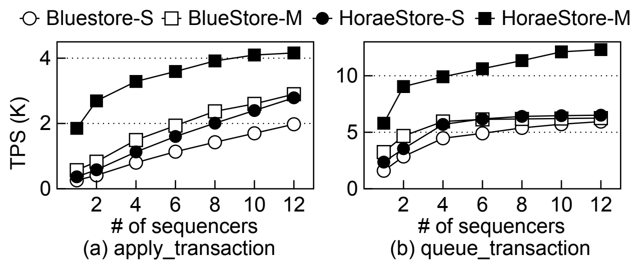


Figure 14: Object Store Performance. *Store-S: mix data, metadata, WAL to device A. Store-M: data to device A, metadata to device B, WAL to device C. apply_transaction: durability guarantees. queue_transaction: ordering guarantee.*

vices of particular usage is better than the automatic dispersion of logical volumes. A naive implementation of RAID 0 treats the devices equally. However, the data flows of application usually have different write traffic and locality. Therefore, uniform distribution potentially bounds the better devices and ruins the data locality.

6.6.2 SQLite

This subsection focuses on the performance of SQLite [19]. The detailed setups are presented in the caption of Figure 13.

On SATA SSD, the ordering setups (OD) outperform the durability ones (DR) by an order of magnitude due to the reduction of the prohibitive `FLUSH`. In ordering, both BFS and HFS exhibit over 20% performance gain against ext4 due to the separation of ordering and durability that brings chances of overlapping CPU with IOs.

On NVMe SSD, as the `FLUSH` becomes inexpensive, ext4-OD achieves almost the same performance as ext4-DR. HFS separates the control path from the data path, and thus SQLite can order the table files and logs through `fbarrier()`. Therefore, more IOs can be processed at the same time.

6.6.3 BlueStore

This subsection evaluates the transaction processing of object store with default options. We use the built-in object store benchmark [5] of Ceph with varying the number of sequencers. Each sequencer serializes the transactions, and transactions of different sequencers do not have dependency. Each transaction issues 20 KB write, which is split into 16 KB aligned write to data device and 4 KB small write to RocksDB. Two interfaces are evaluated, `apply_transaction()` and `queue_transaction()`, representing ordering and durability guarantee, respectively. Figure 14 shows the results.

In Figure 14(a), HoraeStore exhibits 1.4× and 2.1× TPS gain against BlueStore, in S and M setup, respectively. To preserve order, BlueStore does not submit the small write and metadata to RocksDB until the aligned write has been completed. While in HoraeStore, once the aligned write and KV transaction have been processed serially via the control path, associated data blocks can be processed concurrently.

The `queue_transaction()` brings opportunities to apply multiple transactions. As shown in Figure 14(b), HoraeStore outperforms BlueStore averagely by 23% and 83% in S and M setup, respectively. Due to the write dependency over multiple devices, the slower data device burdens the faster metadata and WAL devices. Hence, BlueStore-M delivers similar TPS compared to BlueStore-S. In HoraeStore-M, as the control path guarantees the ordering, the synchronization between aligned write and KV transaction is alleviated. Further, HoraeStore enables more KV transactions to continuously fulfill the metadata and WAL storage.

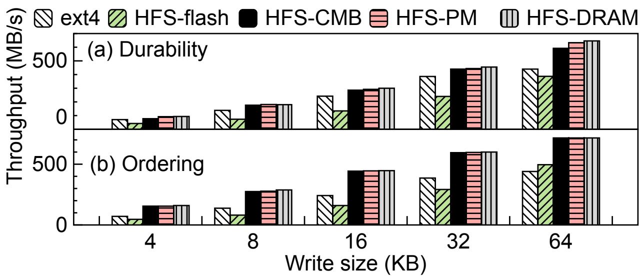


Figure 15: Comparison of The Media of The Ordering Queue. *flash: block IO to SSD. CMB: MMIO to SSD's memory buffer. PM: Intel Optane persistent memory. DRAM: capacitor-back memory.*

7 Discussion

CMB Alternatives. Recall that Horae persists the ordering metadata in the CMB for efficiency. Nevertheless, several off-the-shelf non-volatile media are capable of storing the ordering metadata: SSD (flash), Intel persistent memory (PM) and capacitor-backed DRAM. We locate the ordering queue in these media and measure the single-threaded throughput of the ordered writes, as shown in Figure 15. We find that storing the ordering metadata directly through the block-based interface to SSD (i.e., HFS-flash) significantly decreases the throughput. This is because, even the ordering metadata is 16 B, it must be padded to 4 KB, where the 4 KB synchronous PCIe transfer masks the concurrency of translated orderless writes. When the write size increases (over 64 KB), HFS-flash gradually outperforms ext4. We also find the PM and

DRAM are also satisfiable alternatives of the CMB.

8 Related Work

Storage order. Many researchers [25–28, 40, 46] have studied and mitigated the overhead of storage order guarantee. Among these studies, the closest ones are OptFS [26] and BarrierFS [46] that separate the ordering guarantee from durability guarantee and provide similar ordering primitive. OptFS, BarrierFS and HoraeFS are proposed under different storage technologies (HDD, SATA SSD, NVMe SSD and storage arrays), thereby mainly differing in the architecture, the scope of application and hardware requirements. First, OptFS embeds the transaction checksum in the journal commit block and detects the ordering violation during recovery, which reduces the rotational disk FLUSH overhead at runtime. BarrierFS preserves the order layer by layer, thereby aligning with the single queue structure of SCSI stack and devices. They are difficult to be extended to multiple queues or multiple devices. In contrast, Horae stores the ordering metadata via a dedicated control path to maintain the write order. This design aims to let the ordering bypass the traditional stack to enable high throughput and easy scaling to multiple devices. Second, the checksum-based ordering approach of OptFS is limited to continuous address space (e.g., file system journaling), because the checksum can be only used to detect the ordering violation of data blocks in pre-determined locations. Alternatively, Horae builds a more generic ordering layer which can spread data blocks to arbitrary logical locations of any device. Third, OptFS requires the disk to support asynchronous durability notification. BarrierIO requires barrier compliant storage device which is only available in a few eMMC (embedded multimedia card) products. Horae can run on the standard NVMe devices with exposed CMB. The CMB feature is already defined in NVMe spec, and is under increasing promotion and recognition by NVMe and SPDK communities [6].

Dependency tracking. Some works use dependency tracking techniques to handle storage order. Soft updates [39] directly tracks the dependencies of the file system structures in a per-pointer basis. Similarly, Featherstitch [28] introduces the *patch* to specify how a range of bytes should be changed. Horae also tracks the write dependencies in the ordering queue. The tracking unit of Horae is different from prior works; each entry in Horae describes how a range of data blocks (e.g., 4 KB) should be ordered. The block-aligned tracking introduces less complexity of both dependency tracking and file system modifications, and it is more generic in the context of block device. In addition, due to the disability of telling data versions, soft updates does not support version consistency. Featherstitch assumes single in-progress write to the same block address, and treats dependency loop as errors. Thus, the in-place updates of Featherstitch must wait for the completion of preceding one. Instead, Horae saves the in-place updates from long DMA transfer through write redirection with enhanced consistency.

Storage IO stack. A school of works [30,31,34,35,37,40–43, 49,51] improve the storage IO stack. Xsyncfs [40] uses output-triggered commits to persist a data block only when the result needs to be externally visible. IceFS [37] allocates separate journals for each container for isolation. SpanFS [31] partitions the file system at *domain* granularity for performance scalability. Built atop F2FS [34], ParaFS [51] co-designs the file system with the SSD's FTL to bridge the semantics gap between the hardware and software. iJournaling [41] designs fine-grained journaling for each file, and thus mitigates the interference between `fsync()` threads. CCFS [42] provides similar *stream* abstraction at file system level for applications to implement correct crash consistency. Its stream is designed on individual journals and still relies on exclusive IO processing to preserve the order. Horae exports stream at block level via the dedicated control path, not relying on exclusive IO processing nor journal. TxFS [30] leverages the file system journaling to provide transactional interface. Son *et al.* [43] propose a high-performance and parallel journal scheme. FlashShare [49] punches through the IO stack to device firmware to optimize the latency for ultra-low latency SSDs. AsyncIO [35] overlaps the CPU execution with IO processing, so as to reduce the `fsync()` latency. CoinPurse leverages the byte interface and device-assisted logic to expedite non-aligned writes [47]. However, these works still rely on exclusive IO processing to control the internal order (e.g., the order between data blocks and metadata blocks) and external order (e.g., the order of applications' data).

NoFS [27] introduces backpointer-based consistency to remove the ordering point between two dependent data blocks. Due to the lack of ordering updates, NoFS does not support atomic operations (e.g., `rename()`).

9 Conclusion

In this paper, we revisit the write dependency issue on high-performance storage and storage arrays. Through a performance study, we notice that with the growth of performance of storage arrays, the performance loss induced by the write dependency becomes more severe. Classic IO stack is not efficient in resolving this issue. We thus propose a new IO stack called Horae. Horae separates the ordering control from the data flow, and uses a range of techniques to ensure both high performance and strong consistency. Evaluations show that Horae outperforms existing IO stacks.

10 Acknowledgement

We sincerely thank our shepherd Vijay Chidambaram and the anonymous reviewers for their valuable feedback. We also thank Qing Wang and Zhe Yang for the discussion on this work. This work is supported by National Key Research & Development Program of China (Grant No. 2018YFB1003301), the National Natural Science Foundation of China (Grant No. 61832011, 61772300).

References

[1] A Persistent Key-Value Store for Fast Storage. https://rocksdb.org/.

[2] A way to do atomic writes. https://lwn.net/Articles/789600/.

[3] An async IO implementation for Linux. https://elixir.bootlin.com/linux/v4.18.20/source/fs/aio.c.

[4] BFQ (Budget Fair Queueing). https://www.kernel.org/doc/html/latest/block/bfq-iosched.html.

[5] Ceph Objectstore benchmark. https://github.com/ceph/ceph/blob/master/src/test/objectstore_bench.cc.

[6] Enabling the NVMe^{TM} CMB and PMR Ecosystem. https://nvmexpress.org/wp-content/uploads/Session-2-Enabling-the-NVMe-CMB-and-PMR-Ecosystem-Eideticom-and-Mell....pdf.

[7] ext4 Data Structures and Algorithms. https://www.kernel.org/doc/html/latest/filesystems/ext4/index.html.

[8] Filebench - A Model Based File System Workload Generator. https://github.com/filebench/filebench.

[9] fio - Flexible I/O tester. https://fio.readthedocs.io/en/latest/fio_doc.html.

[10] Intel Solid State Drive 750 Series Datasheet. https://www.intel.com/content/dam/www/public/us/en/documents/product-specifications/ssd-750-spec.pdf.

[11] Intel® SSD 545s Series. https://www.intel.com/content/www/us/en/products/memory-storage/solid-state-drives/consumer-ssds/5-series/ssd-545s-series/545s-256gb-2-5inch-6gbps-3d2.html.

[12] Intel® SSD DC P3700 Series. https://ark.intel.com/content/www/us/en/ark/products/79621/intel-ssd-dc-p3700-series-2-0tb-2-5in-pcie-3-0-20nm-mlc.html.

[13] MySQL reference manual. https://dev.mysql.com/doc/refman/8.0/en/.

[14] NoLoad U.2 Computational Storage Processor. https://www.eideticom.com/uploads/attachments/2019/07/31/noload_csp_u2_product_brief.pdf.

[15] NVMe specifications. https://nvmexpress.org/resources/specifications/.

[16] NVMe SSD with Persistent Memory Region. https://www.flashmemorysummit.com/English/Collaterals/Proceedings/2017/20170810_FM31_Chanda.pdf.

[17] PCI Express Base Specification Revision 3.0. http://www.lttconn.com/res/lttconn/pdres/201402/20140218105502619.pdf.

[18] Product Brief: Intel® Optane^{TM} SSD DC D4800X Series. https://www.intel.com/content/www/us/en/products/docs/memory-storage/solid-state-drives/data-center-ssds/optane-ssd-dc-d4800x-series-brief.html.

[19] SQLite. https://www.sqlite.org/index.html.

[20] Starblaze OC SSD. http://www.starblaze-tech.com/en/lists/content/id/137.html.

[21] SysBench manual. https://imysql.com/wp-content/uploads/2014/10/sysbench-manual.pdf.

[22] Embedded Multimedia Card. http://www.konkurel.ru/delson/pdf/D93C16GM525(3).pdf, 2018.

[23] A. Aghayev, S. Weil, M. Kuchnik, M. Nelson, G. R. Ganger, and G. Amvrosiadis. File systems unfit as distributed storage backends: Lessons from 10 years of ceph evolution. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 353–369, New York, NY, USA, 2019. Association for Computing Machinery.

[24] D.-H. Bae, I. Jo, Y. A. Choi, J.-Y. Hwang, S. Cho, D.-G. Lee, and J. Jeong. 2b-ssd: The case for dual, byte- and block-addressable solid-state drives. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ISCA '18, page 425–438. IEEE Press, 2018.

[25] Y.-S. Chang and R.-S. Liu. Optr: Order-preserving translation and recovery design for ssds with a standard block device interface. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '19, pages 1009–1023, Berkeley, CA, USA, 2019. USENIX Association.

[26] V. Chidambaram, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Optimistic crash consistency. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 228–243, New York, NY, USA, 2013. ACM.

[27] V. Chidambaram, T. Sharma, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Consistency without ordering. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, FAST'12, page 9, USA, 2012. USENIX Association.

[28] C. Frost, M. Mammarella, E. Kohler, A. de los Reyes, S. Hovsepian, A. Matsuoka, and L. Zhang. Generalized

file system dependencies. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, page 307–320, New York, NY, USA, 2007. Association for Computing Machinery.

[29] M. Hedayati, K. Shen, M. L. Scott, and M. Marty. Multi-queue fair queuing. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 301–314, Renton, WA, July 2019. USENIX Association.

[30] Y. Hu, Z. Zhu, I. Neal, Y. Kwon, T. Cheng, V. Chidambaram, and E. Witchel. Txfs: Leveraging file-system crash consistency to provide ACID transactions. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 879–891, Boston, MA, July 2018. USENIX Association.

[31] J. Kang, B. Zhang, T. Wo, W. Yu, L. Du, S. Ma, and J. Huai. Spanfs: A scalable file system on fast storage devices. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '15, pages 249–261, Berkeley, CA, USA, 2015. USENIX Association.

[32] W.-H. Kang, S.-W. Lee, B. Moon, Y.-S. Kee, and M. Oh. Durable write cache in flash memory ssd for relational and nosql databases. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 529–540, New York, NY, USA, 2014. ACM.

[33] N. Kirsch. Phison E12 High-Performance SSD Controller. `https://www.legitreviews.com/sneak-peek-phison-e12-high-performance-ssd-controller_206361`, 2018.

[34] C. Lee, D. Sim, J.-Y. Hwang, and S. Cho. F2fs: A new file system for flash storage. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, FAST'15, pages 273–286, Berkeley, CA, USA, 2015. USENIX Association.

[35] G. Lee, S. Shin, W. Song, T. J. Ham, J. W. Lee, and J. Jeong. Asynchronous i/o stack: A low-latency kernel i/o stack for ultra-low latency ssds. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 603–616, Renton, WA, July 2019. USENIX Association.

[36] B. Lepers, O. Balmau, K. Gupta, and W. Zwaenepoel. Kvell: The design and implementation of a fast persistent key-value store. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 447–461, New York, NY, USA, 2019. Association for Computing Machinery.

[37] L. Lu, Y. Zhang, T. Do, S. Al-Kiswany, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Physical disentanglement in a container-based file system. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 81–96, Berkeley, CA, USA, 2014. USENIX Association.

[38] Marvell. Marvell 88SS1093 Flash Memory Controller. `https://www.marvell.com/content/dam/marvell/en/public-collateral/storage/marvell-storage-88ss1093-product-brief-2017-03.pdf`, 2017.

[39] M. K. McKusick and G. R. Ganger. Soft updates: A technique for eliminating most synchronous writes in the fast filesystem. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '99, page 24, USA, 1999. USENIX Association.

[40] E. B. Nightingale, K. Veeraraghavan, P. M. Chen, and J. Flinn. Rethink the sync. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, page 1–14, USA, 2006. USENIX Association.

[41] D. Park and D. Shin. ijournaling: Fine-grained journaling for improving the latency of fsync system call. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '17, pages 787–798, Berkeley, CA, USA, 2017. USENIX Association.

[42] T. S. Pillai, R. Alagappan, L. Lu, V. Chidambaram, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Application crash consistency and performance with CCFS. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 181–196, Santa Clara, CA, Feb. 2017. USENIX Association.

[43] Y. Son, S. Kim, H. Y. Yeom, and H. Han. High-performance transaction processing in journaling file systems. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies*, FAST'18, pages 227–240, Berkeley, CA, USA, 2018. USENIX Association.

[44] S. C. Tweedie. Journaling the linux ext2fs filesystem. In *In LinuxExpo'98: Proceedings of The 4th Annual Linux Expo*, 1998.

[45] S. Weil, S. Brandt, E. Miller, D. Long, and C. Maltzahn. Ceph: A scalable, high-performance distributed file system. pages 307–320, 11 2006.

[46] Y. Won, J. Jung, G. Choi, J. Oh, S. Son, J. Hwang, and S. Cho. Barrier-enabled io stack for flash storage. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies*, FAST'18, pages 211–226, Berkeley, CA, USA, 2018. USENIX Association.

[47] Z. Yang, Y. Lu, E. Xu, and J. Shu. Coinpurse: A device-assisted file system with dual interfaces. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2020.

[48] J. Yeon, M. Jeong, S. Lee, and E. Lee. Rflush: Rethink the flush. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies*, FAST'18, pages 201–209, Berkeley, CA, USA, 2018. USENIX Association.

[49] J. Zhang, M. Kwon, D. Gouk, S. Koh, C. Lee, M. Alian, M. Chun, M. T. Kandemir, N. S. Kim, J. Kim, and M. Jung. Flashshare: Punching through server storage stack from kernel to firmware for ultra-low latency ssds. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 477–492, Carlsbad, CA, Oct. 2018. USENIX Association.

[50] J. Zhang, M. Kwon, M. Swift, and M. Jung. Scalable parallel flash firmware for many-core architectures. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 121–136, Santa Clara, CA, Feb. 2020. USENIX Association.

[51] J. Zhang, J. Shu, and Y. Lu. Parafs: A log-structured file system to exploit the internal parallelism of flash devices. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 87–100, Denver, CO, June 2016. USENIX Association.