Crash Consistent Non-Volatile Memory Express

Xiaojian Liao, Youyou Lu, Zhe Yang, Jiwu Shu*
Department of Computer Science and Technology, Tsinghua University
Beijing National Research Center for Information Science and Technology (BNRist)

Abstract

This paper presents crash consistent Non-Volatile Memory Express (ccNVMe), a novel extension of the NVMe that defines how host software communicates with the non-volatile memory (e.g., solid-state drive) across a PCI Express bus with both crash consistency and performance efficiency. Existing storage systems pay a huge tax on crash consistency, and thus can not fully exploit the multi-queue parallelism and low latency of the NVMe interface. ccNVMe alleviates this major bottleneck by coupling the crash consistency to the data dissemination. This new idea allows the storage system to achieve crash consistency by taking the free rides of the data dissemination mechanism of NVMe, using only two lightweight memory-mapped I/Os (MMIO), unlike traditional systems that use complex update protocol and heavyweight block I/Os. ccNVMe introduces transaction-aware MMIO and doorbell to reduce the PCIe traffic as well as to provide atomicity. We present how to build a high-performance and crash-consistent file system namely MQFS atop ccNVMe. We experimentally show that MQFS increases the IOPS of RocksDB by 36% and 28% compared to a state-of-the-art file system and Ext4 without journaling, respectively.

CCS Concepts: • **Information systems → Information storage systems**.

Keywords: storage protocol, crash consistency, file system, SSD, NVMe

ACM Reference Format:
Xiaojian Liao, Youyou Lu, Zhe Yang, Jiwu Shu. 2021. Crash Consistent Non-Volatile Memory Express. In *ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP '21), October 26–29, 2021, Virtual Event, Germany*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3477132.3483592

*Jiwu Shu is the corresponding author.

SOSP '21, October 26–29, 2021, Virtual Event, Germany

ACM ISBN 978-1-4503-8709-5/21/10…$15.00
https://doi.org/10.1145/3477132.3483592

System	Software overhead	PCIe traffic			
		MMIO	DMA(Q)	Block I/O	IRQ
Ext4/NVMe	High	2(N+2)	2(N+2)	N+2	N+2
HoraeFS/NVMe	Medium	2(N+2)	2(N+2)	N+2	N+2
MQFS/ccNVMe	Low	4	N+1	N+1	N+1
MQFS-A/ccNVMe	Low	2	0	0	0

Table 1. Software overhead and PCIe traffic of different systems for ensuring crash consistency. *The number represents the count of operations needed for ensuring crash consistency of a transaction that consists of N individual 4 KB data blocks. MMIO: memory-mapped I/O over PCIe. DMA(Q): device transfers queue entries from/to host using DMA. Block I/O: 4 KB data blocks transferred via PCIe. IRQ: interrupt request. Existing file systems atop NVMe require (N+2) block I/Os and IRQs over PCIe in the common case; the number 2 indicates the extra journal description and commit record requests. Built atop ccNVMe, MQFS removes the commit record by taking the free rides of the doorbell operations, thereby reducing the number of block I/O and IRQ by 1. ccNVMe also reduces the number of MMIO and DMA(Q) via the transaction-aware MMIO and doorbell techniques. By further decoupling atomicity from durability, MQFS-A atop ccNVMe does not need to wait for the completion of DMAs(Q), block I/Os and IRQs.*

1 Introduction

The storage hardware has improved significantly over the last decade, e.g., an off-the-shelf solid-state drive (SSD) [5] can deliver over 7 GB/s bandwidth and provide 5 microseconds I/O latency. To better utilize high-performance SSDs, the Non-Volatile Memory Express (NVMe) is introduced at the device driver layer to offer fast accesses over PCIe. With these changes, the performance bottlenecks are shifted back to the software stack.

The crash consistency (i.e., consistently update the persistent data structures despite a sudden system crash such as a power outage) is one fundamental and challenging issue faced by the storage systems. Providing crash consistency incurs expensive performance overhead, and further prevents the system software from taking full advantage of the fast storage devices. Responding to this challenge, great efforts [15, 17, 22, 24, 27, 28, 31–33, 38, 39, 45, 47] have been made to enhance the software stack.

Although the hardware and software stack can be continuously advanced, there still remains one critical issue: the inefficiency from the boundary of the hardware and software (i.e., the NVMe driver) prevents the software stack

from further providing higher performance (§3). For example, as presented in Table 1, to guarantee crash consistency of a transaction that consists of N individual 4 KB user data blocks, existing file systems (e.g., Ext4 and HoraeFS [27]) built atop NVMe need to wait for the completion of these data blocks which involve several MMIOs, DMAs, block I/Os and IRQs. This consumes the available bandwidth of the PCIe links and the SSD, and increases the transaction latency, therefore lowering the application performance.

In this paper, we propose ccNVMe, a novel extension of NVMe to define how host software communicates with the SSD across a PCIe bus with both crash consistency and performance efficiency (§4). The key idea of ccNVMe is to couple the crash consistency to the data dissemination; a transaction (a set of requests that must be executed atomically) is guaranteed to be crash-consistent when it is about to be dispatched over PCIe. The data dissemination mechanism of the original NVMe already tracks the life cycle (e.g., submitted or completed) of each request in the hardware queues and doorbells. ccNVMe leverages this feature to submit and complete the transaction in an ordered and atomic fashion, and makes the tracked life cycles persistent for recovery, thereby letting the software ensure crash consistency by taking the free rides of the data dissemination MMIOs. Specifically, a transaction is crash-consistent once ccNVMe rings the submission or completion doorbells.

ccNVMe communicates with the SSD in a transaction-aware fashion, rather than the eager per-request basis of the original NVMe; this reduces the number of MMIOs, block I/Os and interrupt requests (see MQFS/ccNVMe of Table 1), and thus increases the maximum performance that the file system can achieve. By further decoupling atomicity from durability, ccNVMe ensures crash consistency just after ringing (notifying) the SSD's submission queue doorbell, with only two MMIOs (see MQFS-A/ccNVMe of Table 1).

ccNVMe is pluggable and agnostic to storage systems; any storage system demanding crash consistency can enable ccNVMe and explicitly mark the request as an atomic one. Here, we design and implement MQFS to exploit the fast atomicity and high parallelism of ccNVMe (§5). We further introduce a range of techniques including multi-queue journaling and metadata shadow paging to reduce the software overhead.

We implement ccNVMe and MQFS in the Linux kernel (§6). ccNVMe places the submission queues along with its head and tail values on the persistent memory region (PMR) [10] of the NVMe SSDs, and embeds the transaction order in the reserved fields of the NVMe I/O command. As a result, ccNVMe provides failure-atomicity without any logic changes to the hardware, and is compatible with the original NVMe.

We experimentally compare ccNVMe and MQFS against Ext4 [2], HoraeFS [27] which is a state-of-the-art journaling file system, and Ext4-NJ (§7); Ext4-NJ does not perform journaling and we assume it to be the ideal upper bound of the Ext4 on modern NVMe SSDs. We find MQFS performs significantly better than Ext4 and HoraeFS for a range of workloads. MQFS even surpasses Ext4-NJ when the workload is not severely bounded by I/O. In particular, MQFS increases the throughput of RocksDB by 66%, 36% and 28%, compared to Ext4, HoraeFS and Ext4-NJ, respectively. Through the crash consistency test of CrashMonkey [35], we demonstrate that MQFS can recover to a correct state after a crash.

In summary, we make the following contributions:

- We propose ccNVMe to achieve high performance and crash consistency by coupling the crash consistency to the data dissemination, decoupling atomicity from durability and introducing transaction-aware MMIO and doorbell.
- We propose MQFS to fully exploit ccNVMe, along with a range of techniques to reduce software overhead.
- We implement and evaluate ccNVMe and MQFS in the Linux kernel, demonstrating that ccNVMe and MQFS outperform state-of-the-art systems.

2 Non-Volatile Memory Express

Non-Volatile Memory Express (NVMe) [6] is an interface like serial ATA (SATA) for software to communicate with high-performance SSDs. It is much more efficient than legacy interfaces due to its low latency and high parallelism of its high-performance queuing mechanism. The NVMe supports 65535 I/O hardware queues each with 65535 commands (i.e., queue depth). Each hardware queue is mapped to each CPU core to deliver scalable performance. Here, we use Figure 1 to briefly introduce its data dissemination mechanism.

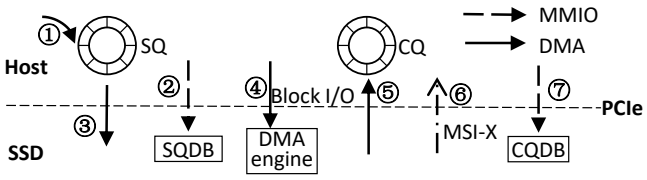

Figure 1. NVMe command processing. *Described in §2.*

Each host CPU core has its own independent submission queue (SQ), completion queue (CQ) and associated doorbells (SQDB and CQDB). The SQ and CQ are essentially circular buffers stored in host memory; the SQDB stores the tail value of SQ while the CQDB stores the head value of CQ. The host first places the I/O command in the free SQ slot (①), followed by updating the SQDB with the new tail value (②) to notify the SSD of the incoming command. The SSD then fetches the command (③) and transfers the data from host (④). After a command has completed execution, the SSD places a CQ entry in the free slot of the CQ (⑤), followed by generating an interrupt to the host (⑥). The host consumes the new CQ entry and then writes the CQDB with a new head value to indicate that the CQ entry has been consumed (⑦). As we can see, an I/O request requires at least 2 MMIOs, 2 DMAs of the queues, 1 block I/O and 1 interrupt request (e.g., MSI-X).

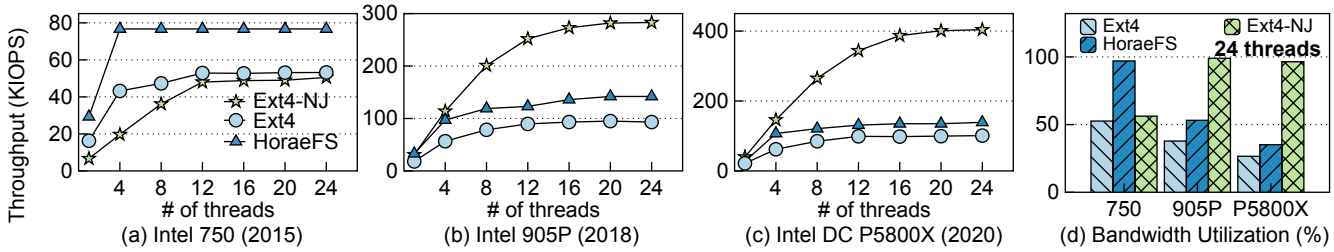

Figure 2. Motivation test. *Throughput of 4 KB append write followed by* `fsync`. *Described in §3.*

PMR (persistent memory region) is a new feature of NVMe released in the 1.4 spec circa June 2019 [8]. It is a region of general purpose read/write persistent memory of the SSD. The SSD can enable this feature by exposing a portion of persistent memory (e.g., capacitor-backed DRAM or Optane Memory) which can be accessed by the CPU load and store instructions. In this paper, to implement and evaluate ccNVMe on a variety of commercial SSDs that do not support PMR, we use 2 MB in-SSD capacitor-backed DRAM to package the tested SSDs as PMR-enabled ones (details in §6).

3 Motivation

In this section, we revisit the crash consistency on modern NVMe SSDs. Journaling (a.k.a., write-ahead logging) is a popular solution used by many file systems including Linux Ext4 [2], IBM's JFS [14], SGI's XFS [44] and Windows NTFS [34] to solve crash consistency issue. Hence, we perform experiments on journaling file systems in particular Ext4, Ext4 without journaling (Ext4-NJ) and a recently proposed HoraeFS [27] to understand the crash consistency overhead. In the Ext4-NJ setup, we disable journaling of Ext4, and assume it to be the ideal upper bound of Ext4 on modern NVMe SSDs. Using the `FIO` [4] tool, we launch up to 24 threads, and each performs 4 KB append `writes` to its private file followed by `fsync` independently. We choose this workload as the massive small synchronous updates can stress the crash consistency machinery (i.e., journaling). We consider three NVMe SSDs that were introduced over the last 6 years, including flash and Optane SSDs; the performance matrix of these SSDs is presented in Table 3; the other configurations of the testbed are described in §7.1. Figure 2 shows the overall results. The gap between Ext4-NJ and Ext4 (or HoraeFS) quantifies the crash consistency overhead.

In the older NVMe drive, as shown in Figure 2(a), the journaling setups (i.e., Ext4 and HoraeFS) perform comparably against the no-journaling setup (i.e., Ext4-NJ), and even outperform Ext4-NJ. Using journaling to take advantage of the higher sequential bandwidth of the SSD, and optimizing journaling as in HoraeFS to reduce the software overhead, delivers significant improvements on throughput; the SSD's bandwidth is therefore saturated (see Figure 2(d)).

However, as NVMe SSDs evolve, the crash consistency overhead becomes significant and tends to be more severe, as presented in Figure 2(b)-(c). Notably in the 24-core case of Figure 2(c), the crash consistency overhead (i.e., the ratio of (Ext4-NJ - HoraeFS) to HoraeFS) is nearly 66%. Except for Ext-NJ, all file systems fail to fully exploit the available bandwidth. Further analyses suggest that the inefficiency comes from the software overhead and PCIe traffic.

Software overhead. Many Ext4-based file systems including HoraeFS and BarrierFS [45] use a separate thread to dispatch the journal blocks for ordering and consistency. The computing power of a single CPU core is sufficient for old drives, but is inadequate for newer fast drives. Moreover, the context switches between the application and journaling thread introduce non-negligible CPU overhead. Efficiently utilizing multi-cores to perform journaling in the application's context becomes important, as we will show in MQFS.

PCIe traffic. To achieve atomicity of N 4 KB data blocks, the journaling generates two extra blocks (i.e., the journal description and commit block) for a single transaction. This approach requires 2 × (N+2) MMIOs, 2 × (N+2) DMAs from/to the queue entries, (N+2) blocks I/Os and (N+2) interrupt requests if block merging is disabled. When the SSD is fully driven by the software stack with enough CPU cores, the application performance is instead bottlenecked by the boundary of the software and hardware. With a large bandwidth consumed at the device driver, the available bandwidth provided to the file system is therefore limited. Moreover, the file system needs to wait for the completion of these I/Os and requests to ensure the atomicity of a transaction. This increases the transaction latency, leaves the CPU in an idle state and thus lowers the throughput.

4 ccNVMe: Design and Implementation

To reduce the PCIe traffic and improve the performance efficiency for crash-consistent storage systems, we propose ccNVMe to provide efficient atomicity and ordering.

The **key idea** of ccNVMe is to couple the crash consistency to the data dissemination. The original NVMe already records the requests in the submission queues and their states in the doorbells; ringing the submission queue doorbell (SQDB) indicates that the requests are about to (but not

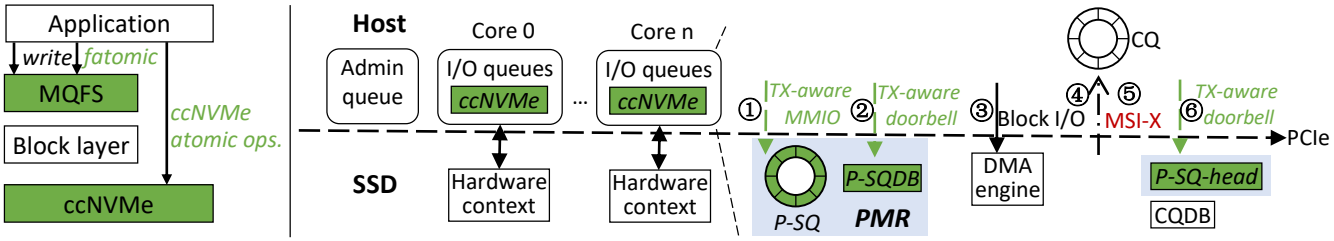

Figure 3. ccNVMe design overview. *To guarantee atomicity, ccNVMe needs only step ① and ② despite the size of a transaction. The green parts highlight the differences between ccNVMe and the original NVMe. Described in §4.*

yet) be submitted to the SSD while ringing the completion queue doorbell (CQDB) suggests that these requests are completed. These two doorbells (states) naturally represent the "0" (nothing) and "1" (all) states of the atomicity.

Based on this observation, we extend NVMe to be atomic and further crash-consistent. ccNVMe makes the submission queues durable in case of a sudden crash, and rings the doorbells in the unit of a transaction rather than a request, to let the requests of a transaction reach the same state (e.g., all or nothing), thereby achieving atomicity. We show how to provide the transaction abstraction atop the original request notion in §4.2. However, as NVMe does not prescribe any ordering constraint nor persistence of the submission queue, it is non-trivial to persist the submission queues entries over the PCIe link and ring the doorbell efficiently and correctly. We introduce transaction-aware MMIO and doorbell for efficient persistence in §4.3, and present how to ring the doorbell to enforce ordering guarantees in §4.4.

4.1 Overview

Figure 3 presents an overview of ccNVMe. In the leftmost of the figure, ccNVMe is a device driver that sits between the block layer and the storage devices. But unlike traditional NVMe, ccNVMe moves further by providing the atomicity guarantees at the boundary of the hardware and software layers. This design has two major advantages. First, ccNVMe lets the atomicity take the free rides of fast NVMe queuing and doorbell operations, and thus accelerates the crash consistency guarantees. Second, ccNVMe provides generic atomic primitives, which can free the upper layer systems from the need to implement a complex update protocol and to reason about correctness and consistency. For example, the applications can directly issue atomic operations, or use the classic file systems APIs (e.g., `write`) followed by `fsync` or a new file system primitive `fatomic` proposed by this work to ccNVMe to ensure failure atomicity.

In the right part of Figure 3, ccNVMe keeps the multi-queue design of the original NVMe intact; each CPU core has its own independent I/O queues (i.e., SQs and a CQ), doorbells and hardware context (e.g., interrupt handler) if the underlying SSD offers enough hardware resources. The only difference is the (optional) ccNVMe extension added to each core. In particular, ccNVMe creates persistent submission queues (P-SQ) and corresponding doorbells (P-SQDB) in the persistent memory region (PMR) of the NVMe SSD. When receiving atomic operations, ccNVMe generates ccNVMe I/O commands to the P-SQ and rings the P-SQDB. Now, the atomicity is achieved by only two MMIOs (i.e., ①-②) in the common case. We design the ccNVMe I/O command by using the reserved fields of the NVMe common command; this makes ccNVMe compatible with NVMe. Consequently, the storage device can directly fetch the I/O commands from the P-SQ without any logic changes.

Dword	Bits	NVMe	ccNVMe
2-3	00:63	reserved	transaction ID
12	16:19	reserved	`REQ_TX` or `REQ_TX_COMMIT`

Table 2. ccNVMe command format. *Each command is 64 Bytes. Dword: 4 Bytes. Described in §4.2.*

The other procedures of an I/O command of ccNVMe, including the data transfer (③), interrupt (④ and ⑤) and command completion (⑥), is almost similar to NVMe, except that the basic operational unit is the *transaction* (a set of operations that need to be executed atomically) rather than the request from each slot of the queues.

4.2 Transaction: the Basic Operational Unit

Each entry of an SQ represents a request to a continuous range of logical block addresses. ccNVMe distinguishes the atomic request from the non-atomic request via a special attribute `REQ_TX`. A special atomic request with `REQ_TX_COMMIT` serves as a commit point for a transaction. Hence, the commit request implicitly flushes the device to ensure durability, by issuing a flush command first and setting the `FUA` bit in the I/O command, if the volatile write cache is present in the SSD. ccNVMe embeds these attributes in a reserved field in the I/O command (Table 2), and handles these atomic requests differently based on their category (§4.3, §4.4).

ccNVMe groups a set of requests as a transaction and assigns each transaction a unique transaction ID. The transaction ID can be generated by the applications or file systems

by a logical or physical timestamp (e.g., RDTSCP instruction). This ID is used for the unique identification of a transaction as well as deciding the persistence order across multiple submission queues. The transaction ID is stored in a reserved field of the command (Table 2).

4.3 Transaction-Aware MMIO and Doorbell

ccNVMe uses persistent MMIO writes to insert atomic requests to P-SQ and ring P-SQDB, which is different from NVMe that uses non-persistent MMIO writes. Figure 4(a) illustrates the persistent MMIO write. MMIO write is performed directly by the CPU store instruction. Here, since the P-SQ structure is organized in a circular log, ccNVMe leverages the write combining (WC) buffer mode of the CPU to consolidate consecutive writes into a larger write burst, thereby improving the memory and PCIe accesses efficiency. To ensure persistence, ccNVMe uses MMIO flushing via two steps. First, `clflush` followed by `mfence` is used to flush the MMIO writes to the PCIe Root Complex. Second, exploiting the PCIe ordering that a read request must not pass a posted request (e.g., write) (Table 2-39 in PCIe 3.1a spec [11]), ccNVMe issues an extra MMIO read request of zero-byte length to ensure that the MMIO writes finally reach PMR.

Unfortunately, persistent MMIO write is significantly slower than the non-persistent one. As shown in Figure 5, when issuing 64 bytes payloads, the latency of persistent write (i.e., write+sync) is 2.5× higher than that of non-persistent write (i.e., write). We also notice that the bandwidth and latency of the persistent write are approaching non-persistent write, especially when the MMIO size is larger than 512 bytes.

The original NVMe uses non-persistent MMIOs and can place the submission queues in the host memory. As a result, it updates the submission queues and rings the doorbells in a relatively eager fashion: whenever a request is inserted into the NVMe submission queue, it rings the doorbell immediately. However, in ccNVMe that requires persistent MMIOs and operates at the unit of a transaction, ringing the doorbell on a per-request basis results in considerable overhead, for two reasons. First, issuing persistent MMIO writes without batching prevents the CPU from exploiting the coalescing potentials in the WC buffer, lowering the performance. Second, per-request doorbell incurs unnecessary MMIOs over PCIe, as a transaction is completed only if all of its requests are finished; only one doorbell operation is needed.

ccNVMe introduces *transaction-aware MMIO and doorbell* for dispatching the requests and ringing the doorbell. The key idea here is to postpone the MMIO flushing (i.e., ② and ③ in Figure 4(a)) and doorbell until a transaction is being committed. Figure 4(b) depicts an example. Suppose a transaction consists of two requests, W_{x-1} and W_{x-2}. In step 1, W_{x-1} is a normal atomic request with `REQ_TX` that comes first, and ccNVMe stores it using the CPU store instruction. When receiving a commit request with `REQ_TX_COMMIT`, ccNVMe triggers MMIO flushing. In step 2, ccNVMe uses cache

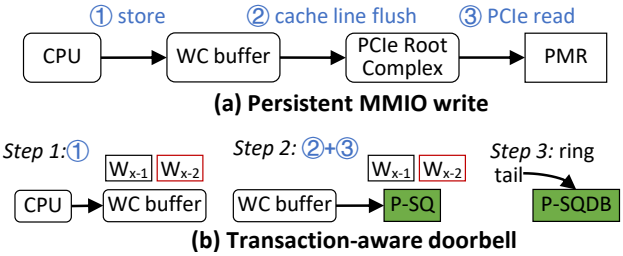

Figure 4. Transaction-aware MMIO and doorbell. *Described in §4.3.*

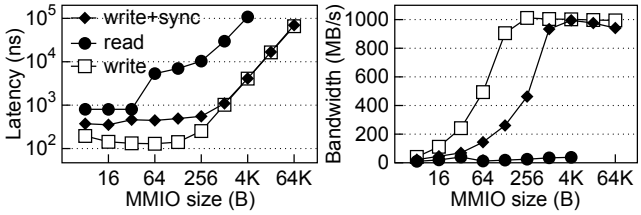

Figure 5. PMR performance. *One kernel thread performs sequential accesses to 2 MB PMR space. Described in §4.3.*

line flush and PCIe read as presented in Figure 4(a) to persist the queue entries to P-SQ. Finally in step 3, ccNVMe rings the P-SQDB by setting the tail pointer of the P-SQ. Compared to naïve approach which requires N MMIO flushings and N doorbell operations for a transaction that contains N requests, ccNVMe only requires 1 MMIO flushing and 1 doorbell operation, regardless of the size of the transaction.

4.4 Correctness and Crash Recovery

Crash consistency includes atomicity and ordering guarantees. We present how ccNVMe provides these two guarantees during normal execution and after a crash. The key idea here is to complete the dependent transaction atomically and in order and track the life cycle of transactions in the face of a sudden crash. During crash recovery, ccNVMe uses the life cycles from PMR to redo the completed transactions while dropping non-atomic or out-of-order transactions.

Normal execution. In ccNVMe, the requests are submitted and completed in the unit of transaction. This is achieved via the transaction-aware doorbell mechanism. In particular, ccNVMe holds the same assumption that ringing the doorbell (i.e., writing a value to a 4 B address) itself is an atomic operation as in the original NVMe. ccNVMe rings the doorbell atomically (i.e., steps ② and ⑥ of Figure 3) after updating the entries of P-SQ and CQ. Consequently, the transactions are submitted and completed atomically.

The ordering here means the completion order of dependent transactions during normal execution. ccNVMe maintains only the "first-come-first-complete" order of each hardware queue although the original NVMe does not prescribe

any ordering constraint. ccNVMe allows the device controller to process the I/O commands from the submission queue in any order, the same as the original NVMe. Yet, ccNVMe completes the I/O commands in order by chaining the completion doorbell (i.e., updating P-SQ-head and ringing the CQDB sequentially). This ensures that a transaction is made complete only when its preceding ones finish; the upper layer systems thus see the completion states of dependent transactions in order.

Crash recovery. During crash recovery, ccNVMe finds the unfinished transactions and leaves the specific recovery algorithms (e.g., rollback) to upper layer systems. In particular, in the face of a sudden power outage, the data of PMR including P-SQ, P-SQDB and P-SQ-head are saved to a backup region of the persistent media (e.g., flash memory) of the SSD. When power resumes, the data is loaded back onto the PMR. Then, ccNVMe performs crash recovery during the NVMe probe; it provides the upper layer system with the unfinished transactions for recovery. Specifically, the transactions of the P-SQ that range from the P-SQ-head to P-SQDB are unfinished ones. ccNVMe makes an in-memory copy of these unfinished transactions; the upper layer systems can thus use this copy for recovery logic (e.g., replay finished transactions and discard unfinished ones). As ccNVMe always completes the transactions atomically and in order, it keeps the correct persistence order after crash recovery.

ccNVMe does not guarantee any global order across multiple hardware queues; it only assists the upper layer system with the global order by providing the persistent transaction ID field. Upper layer systems can embed the global order in this field to decide the persistence order during recovery. We further show a file system crash recovery in §5.5 and experimentally study its correctness in §7.6.

4.5 Programming Model

ccNVMe is a generic device driver that does not change the interfaces for upper layer systems. Specifically, the kernel file system can use the intact `submit_bio` function to submit the write requests that require crash consistency; the application can use the original `nvme` command or the `ioctl` system call to submit raw ccNVMe commands. The only exception is that upper layer systems must explicitly mark the request (e.g., tag the `bio` structure with `REQ_TX`) and control the ordering across multiple hardware queues (e.g., write the transaction ID to a new field of the original `bio`).

In the current design, ccNVMe does not control the ordering and atomicity across multiple hardware queues for the consideration of CPU and I/O concurrency. Therefore, ccNVMe does not allow the requests of a transaction to be distributed to different hardware queues. This requires that the thread queuing atomic requests to ccNVMe can not change its running core until it commits the transaction (i.e., marks a request with `REQ_TX_COMMIT`), as the current NVMe storage stack follows the principle that assigning a dedicated hardware queue to each core as much as possible. As we will show in §7.5.2, queuing a transaction consumes only μs-scale latency, and this is not a big limitation. We leave the solution to this limitation for future work.

4.6 Discussion

ccNVMe does not need any change in hardware logic, though, additional changes in the NVMe SSD controller are highly likely to significantly boost the performance.

Transaction-aware scheduling. The responsiveness of a transaction is determined by its slowest request. The NVMe SSD controller can leverage the transaction notion of ccNVMe to dispatch and schedule requests to different channels and chips, to achieve low transaction latency.

Transaction-aware interrupt coalescing. The NVMe has standardized interrupt coalescing to mitigate host interrupt overhead by reducing the rate at which interrupt requests (e.g., MSI-X) are generated by a controller. Nonetheless, the suitable aggregation granularity of the interrupt coalescing is hard to decide due to the semantic gap and workload change. Using ccNVMe, the controller can send only one interrupt to the host only when a transaction is completed.

5 MQFS: The Multi-Queue File System

ccNVMe is file system and application agnostic; any file system or application desiring crash consistency can be adopted to ccNVMe, by explicitly marking the atomic requests and assigning the same transaction ID to the requests from a transaction. Recall that our study in §3 shows modern Linux file systems still suffer from software overhead and thus are unable to take full advantage of ccNVMe. In this work, we introduce multi-queue file system (MQFS) to fully exploit the atomicity guarantee and multi-queue parallelism of ccNVMe.

5.1 Overview

We develop MQFS based on Ext4 [2], reusing some of its techniques including in-memory indexing and directory/file structure. The major difference is the *multi-queue journaling* introduced to replace the traditional journaling module (i.e., JBD2), along with a range of techniques to ensure both high performance and strong consistency. Here, we present how the critical functions of the crash consistency perform and interact with ccNVMe, followed by introducing each technique at length in the next subsections.

MQFS divides the logical address space of the device into a file system area and several journal areas; the file system area remains intact as in Ext4; MQFS partitions the journal area into multiple portions, and each portion is mapped to a hardware queue. By tagging persistent updates as atomic ones, each core performs journaling on its own hardware queue and journal area, and thus reduces the synchronization from multiple CPU cores.

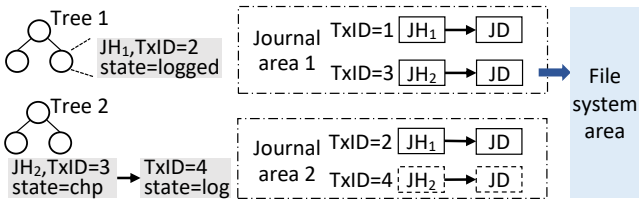

Figure 6. Multi-Queue Journaling. *Described in §5.2.*

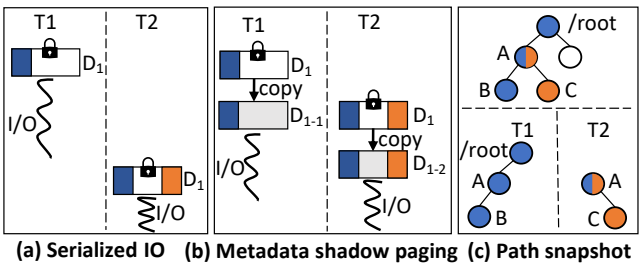

Figure 7. MQFS introduces metadata shadow paging to construct journal entries in parallel. *Described in §5.3.*

Synchronization primitives. The `fsync` of MQFS guarantees both atomicity and durability. Every time a transaction is needed (e.g., `create`, `fsync`), the linearization point is incremented atomically and assigned as the transaction ID. When `fsync` is called, MQFS tags the updates with `REQ_TX` and the final journal description block with `REQ_TX_COMMIT`, followed by sending these blocks to the journal area for atomicity and recovery. Note that compared to JBD2, MQFS eliminates the commit block and removes the ordering points (e.g., `FLUSH`), thereby reducing the write traffic and boosting performance; ringing the P-SQDB actually plays the same role as the commit block. The `fsync` returns successfully until the transaction is made durable, i.e., all updates have experienced steps ① to ⑥ of Figure 3.

Atomicity primitives. MQFS decouples the atomicity from durability, and introduces two new interfaces, `fatomic` and `fdataatomic`, to separately support the atomicity guarantee. `fatomic` synchronizes the same set of blocks of `fsync`, but returns without ensuring durability, i.e., all updates have experienced steps ① to ② of Figure 3. `fdataatomic` is similar to `fatomic`, except that it does not flush the file metadata (e.g., timestamps) if the file size is unchanged. Refer to the following code. `write(file1, "Hello"); write(file1, "SOSP"); fatomic(file1);` using `fatomic`, the application can ensure that the file content is either empty or "Hello SOSP"; no intermediate result (e.g., "Hello") will be persisted.

We present the I/O path of the synchronization and atomicity primitives, and the advantages of separation of atomicity from durability using detailed graphs and numbers in §7.5.2.

5.2 Multi-Queue Journaling

Each core writes the journaled data blocks to its dedicated journal area with less coordination between other cores at runtime. Conflicts are resolved by using the global transaction ID among transactions during checkpointing. A simple way of checkpointing is to suspend all logging requests to each hardware queue, and then checkpoint the journaled data from the journal areas in the order determined by the transaction ID. MQFS instead introduces multi-queue journaling to allow one core to perform checkpointing without suspending the logging requests of other cores.

The key idea of multi-queue journaling is to use per-core in-memory indexes to coordinate the logging and checkpointing, while using on-disk transaction IDs to decide the persistence order. Specifically, the index is a radix tree, which manages the state and newest version of a portion of journaled data blocks (Figure 6). MQFS distributes the journaled blocks to the radix trees with different strategies based on the journaling mode. In data journaling, MQFS distributes the journaled blocks by hashing the final location of the journaled data, e.g., logical block address % the number of trees. In metadata journaling mode, as only the metadata is journaled and the metadata is scattered over multiple block groups (a portion of file system area), MQFS finds the radix tree by hashing the block group ID of the journaled metadata. Each radix tree takes the logical block address of the journaled block as the key, and outputs the journal description entry (JH) recording the mapping from journal block address to final block address along with the transaction ID (TxID) and its current state (state).

MQFS uses these indexes to checkpoint the newest data block and append (but not suspend) the incoming conflicting logging requests. In Figure 6, suppose journal area 1 runs out of space and checkpointing is triggered; note that JH with the same subscript indicates a data block written to the same logical block address. MQFS replays the log sequentially; for JH_1, it finds its TxID is lower than the newest one from tree 1, i.e., another journal area contains a newer block, and thus skips the journaled data (JD). JH_2 in its log is the newest one and therefore can be checkpointed; before checkpointing, the state field is set to `chp`, indicating that this block is being checkpointed. Now, suppose journal area 2 receives a new JH_2. By searching tree 2, MQFS finds that another journal area is checkpointing this block. It then appends the new JH_2 after the old JH_2 of tree 2, marks this entry as `log` and continues to write JH_2 and JD. Using the in-memory indexes to carefully control the concurrency, and the transaction ID to correctly enforce the checkpointing order, MQFS can process logging and checkpointing with higher run-time concurrency.

5.3 Metadata Shadow Paging

The metadata is small (e.g., 256 B) and the file system usually stitches metadata from different files into a single shared

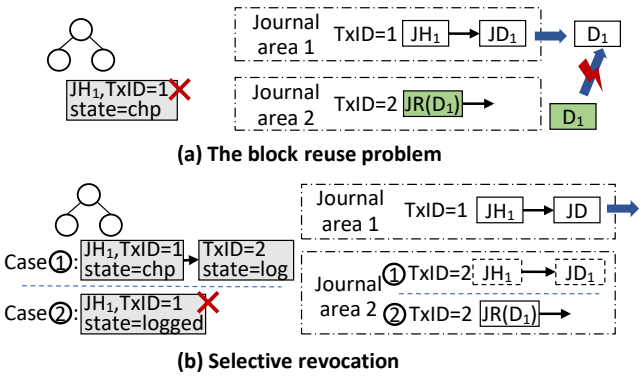

Figure 8. The block reuse problem and our solution selective revocation. *Described in §5.4.*

metadata block. Though accessing different parts of the metadata block, operations from different threads are executed serially. For example, as shown in Figure 7(a), two threads T1 and T2 update the same block D_1. Although the two threads update disjoint parts, they are serialized by the page lock, due to the access granularity of the virtual memory subsystem.

To ease this overhead and to construct the journal entries in parallel, we introduce *metadata shadow paging* to further parallelize I/O operations. The main idea is to update the metadata page sequentially while making a local copy for journaling. MQFS uses this technique to fully exploit the concurrent logging of the multi-queue journaling.

For example in Figure 7(b), T1 updates the in-memory D_1, makes a copy and then journals that copy D_{1-1}. Immediately after T1 has made a copy, T2 can start processing D_1 with the same procedure, i.e., copy and journal D_{1-2}.

In MQFS, only the metadata blocks use shadow paging because only a few metadata blocks (e.g., 1-3) are needed for a `fsync` call in the common case. Data blocks still use the typical lock-based approach because (1) the data blocks are aligned with page granularity without page-level contention from different files, and (2) the number of data blocks is usually non-deterministic; a request with enormous user data blocks can consume a large portion of memory space.

MQFS uses metadata shadow paging to journal the file system directory tree. Specifically, MQFS first takes a snapshot of the updated metadata, and then journals the read-only snapshot. For example in Figure 7(c), T1 creates a new file `B` and persists it to the storage. Meanwhile, T2 creates and syncs a new file `C`. Assume `A` is new to `/root` and T1 goes first. T1 performs shadow paging on path `/root/A/B`, and then releases the lock on the directory entries of `A` and `/root`. After that, T2 performs shadow paging on path `A/C`. Finally, T1 and T2 journal individual path snapshots in parallel, thereby increasing concurrency; MQFS merges the two paths at the checkpoint phase, applying the newest file system directory tree to the file system area.

5.4 Handling Block Reuse across Multi-Queue

A challenge of MQFS is handling block reuse. Like Ext4, MQFS supports both data and ordered metadata journaling. The tricky block reuse case, becomes more challenging in MQFS. In metadata journaling, the file system journals only the file system metadata and lets the user data blocks bypass the journal area. Problems arise from the bifurcated paths.

For example, the file system first journals a directory entry (the content of directories is considered metadata) JD_1. Then the user deletes the directory, freeing the block of D_1. Later, the file system reuses D_1 and writes some user data to it, bypassing journal. Assume a crash happens at this time. The further recovery replays the log which overwrites D_1 with the old directory entry JD_1. As a result, the user data is filled with the content of the directory entry and is thus corrupt. To address this block reuse problem, classic journaling adds revocation record JR to avoid replaying the revoked JD_1.

Unfortunately, directly applying the JR to MQFS can not solve this problem. Using the same example, as shown in Figure 8(a), suppose journal area 1 runs out of space and performs a checkpoint on JD_1. At this time, journal area 2 receives a JR on D_1, indicating the previous JD_1 of journal area 1 can not be replayed. After that, the file system reuses the D_1 and submits a user block directly to the file system area, bypassing the journal. Assume the JD_1 is successfully checkpointed and a crash happens before the persistence of the later D_1. During crash recovery, the JR does not take any effect because the JD_1 is already written back to the file system area. As a result, the user still sees the incorrect data block (i.e., the old directory entry).

The root cause of this issue is: though the JR synchronizes the journal area and file system area, it is unable to coordinate the journal areas across multiple queues. Hence, MQFS uses the per-core radix trees for synchronization, writing the JR record selectively.

Specifically, as shown in Figure 8(b), there are two cases when the file system is about to submit a JR record: (1) the reused block is being checkpointed by the journal area 1 and (2) the reused block is not yet checkpointed. In the first case, the JR record is cancelled and MQFS regresses to data journaling mode for JD_1 and journals the JD_1 for correctness, even D_1 is a user data block. In the second case, the JR record is accepted by journal area 2; the radix tree removes associated JH that is older than the JR. The next checkpoint of journal area 1 therefore ignores JD_1.

5.5 Crash Recovery

Graceful shutdown. At a graceful shutdown (e.g., `umount`), MQFS waits for the completion of all in-progress transactions before detaching from ccNVMe. This ensures that MQFS does not rely on any information from ccNVMe for replaying the journal and ensuring crash consistency.

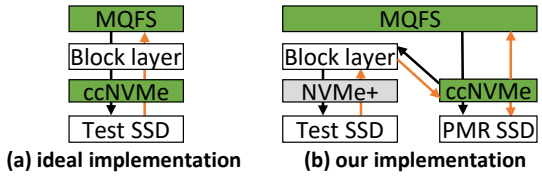

Figure 9. Ideal implementation vs. ours. *NVMe+: original NVMe + transaction-aware doorbell. Described in §6.*

Sudden crash. MQFS performs crash recovery in the unit of transaction. It first reads the P-SQ from ccNVMe to find the unfinished transactions; MQFS discards these transactions. For committed transactions, MQFS links the transactions ordered by the transaction ID from all journal areas, and replays them sequentially, the same as in the classic single-compound journaling of Ext4.

6 Implementation Details

We implement ccNVMe in the Linux kernel 4.18.20 as a loadable kernel module by extending the `nvme` module, which is based on the NVMe 1.2 spec [7] (circa Nov. 2014). We look at the newest 1.4c spec [9] (circa Mar. 2021); there is no change in the command processing steps, common command fields nor the specific write command in this version, and thus the ccNVMe design can be also applied to the newest NVMe.

Using `ioremap_wc`, ccNVMe remaps the PMR region from a PMR-enabled SSD to enable the write combining feature of the CPU on this region. ccNVMe uses `memcpy_fromio` for MMIO read and `memcpy_toio` for MMIO write. As our tested SSDs have not enabled PMR yet, we use an indirect approach to evaluate ccNVMe, as depicted in Figure 9.

In the ideal implementation (Figure 9(a)), a request requires one round trip to the Test SSD. Our implementation (Figure 9(b)) uses a PMR SSD to wrap a Test NVMe SSD as a PMR-enabled one. In particular, MQFS first submits the request to ccNVMe. ccNVMe then forwards the request to the Test SSD through the block layer, after it performs queue and doorbell operations on the PMR SSD. Upon completion, ccNVMe rings the doorbell (if desired) on the PMR SSD before returning to MQFS. In our implementation, the MMIO operations (i.e., ①, ② and ⑥ of Figure 3) are duplicated; one to PMR SSD and another to the Test SSD. The block I/O and MSI-X (i.e., ③, ④ and ⑤ of Figure 3) remains one from the Test SSD. Therefore, the evaluation atop our implementation can reflect the least performance and the same consistency of the ideal implementation.

7 Evaluation

In this section, we first describe the setups of our test environment (§7.1). Next, we examine the performance of transaction processing of ccNVMe (§7.2) and evaluate MQFS against the state-of-the-art journaling file systems through microbenchmark (§7.3) and macrobenchmark (§7.4). Then, we perform a deep dive into understanding how different aspects of ccNVMe and MQFS contribute to its performance gains (§7.5). Finally, we verify the crash consistency of MQFS in the face of a series of complex crash scenarios (§7.6).

Name	Seq. Bandwidth	Rand. IOPS	4KB Latency
Intel flash	Read: 2.2 GB/s	Read: 430K	Read: 20 us
750 NVMe	Write: 0.95 GB/s	Write: 230K	Write: 20 us
Intel Optane	Read: 2.6 GB/s	Read: 575K	Read: 10 us
905P NVMe	Write: 2.2 GB/s	Write: 550K	Write: 10 us
Intel Optane	Read: 7.2 GB/s	Read: 1.5M	Read: 5 us
DC P5800X[1]	Write: 6.2 GB/s	Write: 1.5M	Write: 5 us

[1] This is a PCIe 4.0 SSD. On our PCIe 3.0 server, its sequential read/write bandwidth and random read/write IOPS are 3.3 GB/s, 3.3 GB/s, 850K and 820K, respectively. Its 4 KB random read/write latency through the Linux kernel NVMe stack is 8 us and 9 us, respectively.

Table 3. NVMe SSDs performance.

7.1 Experimental Setup

Hardware. We conduct all experiments in a server with 2 Intel E5-2680 V3 CPUs; each CPU has 12 physical cores and runs at 2.50 GHZ. We use three SSDs; their performance is presented in Table 3. The PMR SSD has 2 MB PMR and its PMR performance is presented in Figure 5.

Compared systems. For the performance of atomicity guarantees, we compare ccNVMe against the classic approach (e.g., JBD2) and Horae's approach [27]. For file system and application performance, we compare MQFS against Ext4, Ext4-NJ and HoraeFS [27], a state-of-the-art journaling file system optimized for NVMe SSDs. Ext4 is mounted with default options. To show the ideal performance upper bound, we disable the journaling in Ext4 and refer this setup to Ext4-NJ. Note that all the tested file systems are based on the same codebase of the Ext4, share the same OS, use metadata journaling and use 1 GB journal space in total.

7.2 Transaction Performance

This section evaluates the transaction performance of different approaches: the classic that writes a journal description block and journaled blocks followed by writing a commit record; the Horae one that removes the ordering points of the classic one; the ccNVMe one that packs the journal description block and the journaled blocks as a single transaction. During the test, we vary the number of threads and the size of a transaction. Each transaction consists of several random 4 KB requests. Each thread performs its own transactions independently. Figure 10 reports the results.

Single-core performance. ccNVMe-atomic outperforms the classic and Horae by 3× and 2.2× averagely in a single core, as shown in Figure 10(a). Compared to the classic and Horae, ccNVMe achieves 1.5× and 1.2× throughput when we

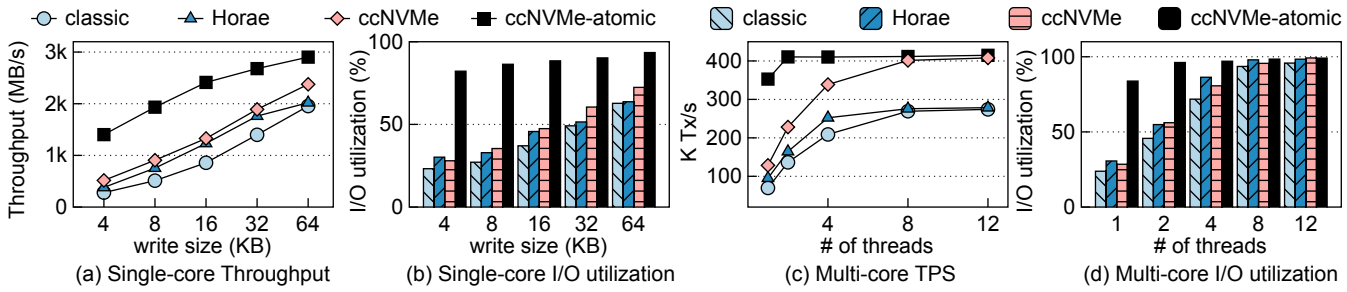

Figure 10. Atomic transaction performance. *Tested SSD: Intel P5800X. Throughput: transactions per second × write size. I/O utilization: used bandwidth ÷ maximum bandwidth. TPS: transactions per second. ccNVMe: atomicity and durability guarantee. ccNVMe-atomic: atomicity guarantee. Described in §7.2.*

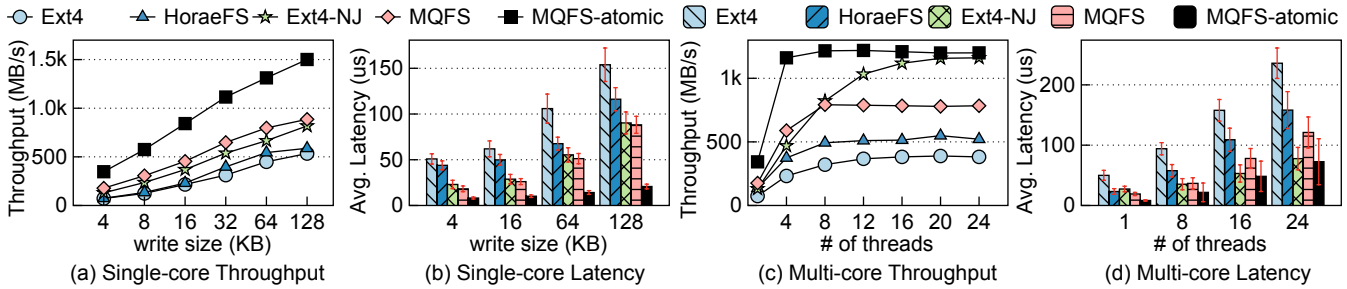

Figure 11. File system performance. *Tested SSD: Intel 905P. MQFS: atomicity and durability guarantees (i.e.,* `fsync`*). MQFS-atomic: atomicity guarantee (i.e.,* `fdataatomic`*). Error bar: standard deviation of the latency. Described in §7.3.*

wait for the durability of the transactions. Through inspecting the I/O utilization via the `iostat` tool, we observe that even with a 64 KB write size, the classic and Horae drives only 62% and 63% the bandwidth while ccNVMe achieves 93% (Figure 10(b)). ccNVMe-atomic does not expose the latency of block I/O and interrupt handler to the transaction processing, thus achieving higher throughput and I/O utilization. Moreover, ccNVMe removes the ordering points in transaction processing as in Horae, and reduces the traffic (i.e., the commit block and doorbell MMIOs) over PCIe, therefore outperforming its peers.

Multi-core performance. We extend the single-core measurements to use up to 12 threads, and each issues 4 KB atomic writes. Figure 10(b) presents the results. We highlight two takeaways here. First, by decoupling atomicity from durability, ccNVMe-atomic saturates the bandwidth using only two cores, while others need at least 8 cores (Figure 10(d)). Second, when the load is high (i.e., over 8 cores), all approaches are able to saturate the bandwidth by issuing independent transactions. However, as ccNVMe eliminates the commit block and reduces the MMIOs, ccNVMe still brings 50% TPS gain against the classic and Horae (Figure 10(c)).

7.3 File System Performance

We examine the throughput and latency of the file systems. Here, we use FIO [4] to issue append `write` followed by `fsync` or `fdataatomic`, which always trigger metadata journaling. During the test, we vary the size of each write request and the number of threads. Figure 11 shows the results.

Single-core performance. From Figure 11(a), we observe that MQFS exhibits 2.1×, 1.9× and 1.2× throughput averagely against Ext4, HoraeFS and Ext4-NJ respectively in a single core. As presented in Figure 11(b), the `fsync` latency decreases by 56%, 41% and 24% on average, when we use MQFS against when we use Ext4, HoraeFS and Ext4-NJ, respectively. From the error bar, we find that MQFS delivers more stable latency. Here, we find that the SSD's bandwidth is not fully saturated by the single thread. Unlike HoraeFS that uses a dedicated thread to perform journaling, MQFS performs journaling at the application's context to avoid context switch, and scales the journaling to multi-queue, thereby increasing the throughput and decreasing the latency. Due to the elimination of the ordering points in journaling, MQFS overlaps the CPU and I/O processing, and thus prevails Ext4-NJ.

Multi-core performance. In Figure 11(c), when the number of threads is lower than 12, MQFS exhibits up to 2.4×, 1.5× and 1.1× throughput gain against Ext4, HoraeFS and Ext4-NJ respectively. As shown in Figure 11(d), MQFS reduces the average `fsync` latency by 55.6% and 28% averagely when compared to Ext4 and HoraeFS respectively. Here, the reasons are a little different. First, `fsync` calls from different threads are likely to contend for the same metadata block. In

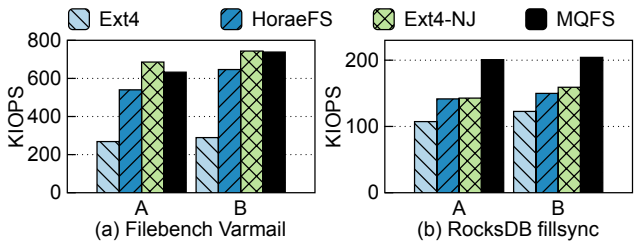

Figure 12. Macrobenchmark. *A: Intel Optane 905P SSD. B: Intel Optane P5800X SSD. Described in §7.4.*

Ext4 and HoraeFS, the accesses to the same block are serialized by the block-level lock. MQFS copies out the metadata block for journaling and thus improves the I/O concurrency. Second, in MQFS, when a thread performs checkpointing, except for necessary version comparison of its local transaction ID with the global one on the global radix trees, it does not block other threads. When the number of threads grows over 12, MQFS saturates the throughput; it achieves 68% the throughput of Ext4-NJ, and outperforms Ext4 and HoraeFS by 2× and 1.5×, respectively. The major bottleneck here is shifted to the write traffic over PCIe. As MQFS does not need the journal commit block and reduces the MMIOs using write combining, it provides higher throughput.

Decoupling atomicity from durability. From Figure 11, we also observe MQFS-atomic further improves performance over MQFS and Ext4-NJ. The improvements come from two aspects. First, ccNVMe itself decouples atomicity from durability; built atop ccNVMe, MQFS guarantees atomicity once the atomic requests are inserted into the hardware queue (i.e., ① and ② of Figure 3), which is very fast (more details in §7.5.2). Second, compared to Ext4-NJ, the threads of MQFS need not synchronize on the shared page, and thus insert the requests independently, efficiently using the CPU cycles.

7.4 Application Performance

We now evaluate MQFS performance over the I/O intensive Varmail [3], and both CPU and I/O intensive RocksDB [1].

Varmail. Varmail is a metadata and `fsync` intensive workload from Filebench [3]. Here, we use the default configuration of Varmail. Figure 12(a) plots the results.

In SSD A, MQFS achieves 2.4×, 1.2× and 0.9× the throughput of Ext4, HoraeFS and Ext4-NJ respectively. In the faster SSD B, MQFS outperforms Ext4 and HoraeFS by 2.6× and 1.1× respectively; MQFS achieves comparable throughput compared to Ext4-NJ. The improvement of MQFS comes from the following aspects. First, in SSD A, all HoraeFS, Ext4-NJ and MQFS are bounded by the I/O. Compared to HoraeFS, MQFS eliminates the journaling commit block and reduces the persistent MMIOs and thus provides higher throughput. Second, in the faster SSD B, I/O is no longer the bottleneck for HoraeFS and Ext4-NJ. Varmail contains many persistent metadata operations such as `creat` and `unlink` followed by `fsync`. MQFS parallelizes the I/O processing of the metadata blocks by metadata shadow paging while Ext4-NJ and HoraeFS serialize the accesses to the shared metadata blocks. Consequently, MQFS utilizes the CPU more efficiently to fully drive the SSD and thus provides higher throughput.

RocksDB. RocksDB is a popular key-value store deployed in several production clusters [1]. We deploy RocksDB atop the tested file systems and measure the throughput of the user requests. Here, we use `db_bench`, a benchmark tool of RocksDB to evaluate the file system performance under the `fillsync` workload, which represents the random write-dominant case. During the test, the benchmark launches 24 threads, and each issues 16-byte key and 1024-byte value to a 20 GB dataset. Figure 12(b) shows the result.

In SSD A, MQFS prevails EXT4-NJ and HoraeFS by 40%. In SSD B, the throughput increases by 66%, 36% and 28% when we use MQFS against when we use Ext4, HoraeFS and Ext4-NJ respectively. MQFS overlaps the I/O processing of the data, metadata and journaled blocks, and reduces the cache line flushings over the PCIe. Therefore, MQFS significantly reduces the CPU cycles spent on idle-waiting (i.e., block I/O) or busy-waiting (i.e., MMIO) for I/O completion. This in turn reserves more CPU cycles for RocksDB and file system logic. During the test, we observe that MQFS has 5× higher CPU utilization (i.e., the CPU cycles consumed in kernel space) and RocksDB atop MQFS has 2× higher CPU utilization (i.e., the CPU cycles consumed in user space). Moreover, the MQFS does not need a commit record, which not only reduces the number of block I/Os that need extra CPU operations (e.g., memory allocation), but also removes the context switches introduced by the interrupt handler. As a result of higher CPU and I/O efficiency, MQFS outperforms its peers on RocksDB which is both CPU and I/O intensive.

7.5 Understanding the Performance

In this section, we evaluate how various design techniques of MQFS contribute to its performance improvement.

7.5.1 Performance Contribution. Now, we show that each of the design techniques of MQFS, i.e., ccNVMe (§4), the multi-queue journaling (§5.2) and the metadata shadow paging (§5.3), are essential to improve the performance. The test increases the number of threads, and each issues 4 KB `write` followed by `fsync` on a private file. We choose Ext4 as the baseline because MQFS is implemented atop Ext4.

Figure 13(a) shows the result on an Optane 905P SSD. The ccNVMe (+ccNVMe) contributes to performance significantly; it achieves approximately 1.4× the throughput of the baseline. Atop the ccNVMe, the multi-queue journaling (+MQJournal) further makes around 47% gains averagely. The metadata shadow paging (+MetaPaging) shows a further 23% throughput improvement. The above results therefore indicate that all of the three building blocks are indeed necessary to improve the performance.

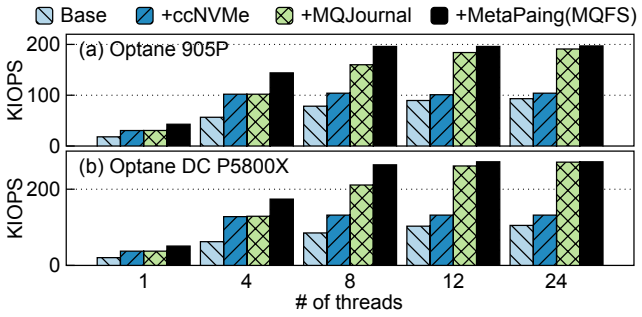

Figure 13. Performance contribution. *Base: Ext4. +ccNVMe: use ccNVMe (§4) to perform journaling. +MQJournal: multi-queue journaling (§5.2). +MetaPaging: metadata shadow paging (§5.3). Described in §7.5.1.*

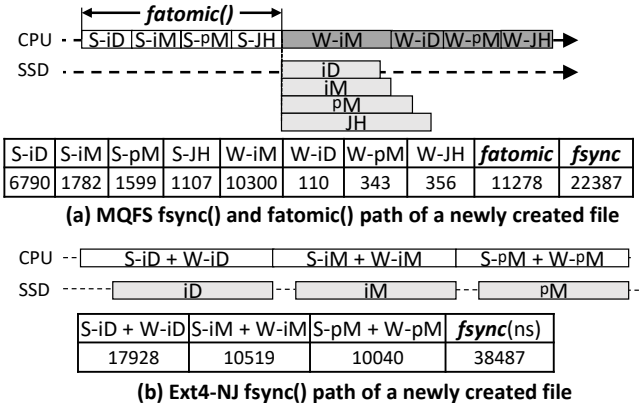

Figure 14. Latency breakdown. *S: submit; i: this file; D: data; M: metadata; p: parent directory; JH: journal description block; W: wait for I/O completion. The number in the tables shows the time (in nanoseconds) spent on this function. Due to power loss protection of the Optane SSD, the* FLUSH *is ignored by the block layer and is thus not shown. Described in §7.5.2.*

We further quantify the effect of each technique in a faster SSD. The results are shown in Figure 13(b). We find that the advantages of ccNVMe become more obvious; the ccNVMe increases the throughput by up to 2.1×. This is because when I/O becomes faster, both the CPU and I/O efficiency become dominant factors affecting performance. ccNVMe removes context switches and decouples atomicity from the durability, thereby accelerating the rate at which the CPU dispatches requests to the device. Moreover, ccNVMe reduces the block I/O and MMIO traffic over PCIe, making more bandwidth for file system usage. The MQJournal also boosts the throughput by 53% averagely with varying the threads. When enabling MetaPaging, the throughput increases by 20% on average. This suggests that scaling the I/Os of the journaling and decoupling the atomicity from the durability to parallelize CPU and I/Os bring significant performance improvement. We also notice that the benefit of ccNVMe becomes narrow beyond 8 threads. This is because when the number of threads exceeds 8, the multicore scalability of the traditional journaling becomes the major performance bottleneck that hides the benefits of ccNVMe.

7.5.2 Decomposing the Latency. In this section, we investigate the file system internal procedure to understand the performance of MQFS against Ext4-NJ. The test initiates one thread and repeats the following operations: it first creates a file, and then `writes` 4 KB data to the file, ending with calling `fsync` on the file. As shown in the topmost of Figure 14(a), for each `fsync`, MQFS starts a transaction, searches the dirty data blocks and allocates space for them (S-iD), followed by sending the blocks to ccNVMe. After that, MQFS processes the file metadata (S-iM) and the parent directory (S-pM) with similar procedures. Next, it constructs and submits the journal description block that contains the transaction ID and the mapping from the home logical block address to the journal logical block address of the journaled data (S-JH). It finally waits for the completion and durability of these blocks (W-x). The table below presents the average time (in nanoseconds) spent on each function. Similarly, Figure 14(b) presents the `fsync` path of Ext4-NJ, which synchronously processes each type of data block without journaling.

MQFS decreases the overall latency by 42% compared to Ext4-NJ. This improvement comes from two aspects: higher CPU and I/O efficiency. First, the CPU is used more efficiently in MQFS. With ccNVMe, MQFS continuously submits the iD, iM, pM and JH, without leaving the CPU in an idle state waiting for the I/O like Ext4-NJ. From the figure, we can see that atomicity guarantee (i.e., `fatomic`) costs only around 10 μs. Second, the I/O in MQFS is performed with higher efficiency. MQFS queues more I/Os to the storage, taking full advantage of the internal data parallelism of the SSD. Nevertheless, the `fatomic` and `fsync` can be further improved according to our analysis. The first is the block layer which is still relatively heavyweight for today's ultra-low latency SSD; for example, S-iM still costs more than 1 μs to pass the request. The second is in S-iD, where Ext4 introduces non-negligible overhead for searching the dirty blocks and allocating space.

Workload	Brief Description	Crash Points	
		Total	Passed
create_delete	`create()` and `remove()` on files.	1000	1000
generic 035	`rename()` overwrite on existing files and directories. From xfstest 035.	1000	1000
generic 106	`link()` and `unlink()` on files, `remove()` directory. From xfstest 106.	1000	1000
generic 321	Various directory `fsync()` tests. From xfstest 321.	1000	1000

Table 4. Crash consistency test. *Described in §7.6.*

7.6 Crash Consistency

We test if MQFS recovers correctly in the face of unexpected system failures. We use CrashMonkey [35], a black-box crash test tool, to automatically generate and perform 1000 tests for each workload. We run four workloads to cover several error-prone file system calls including `rename`; the generic workloads are from xfstest. Table 4 reports the results. As MQFS always packs the target files of a file operation into a single transaction for atomicity, it passes all 1000 test cases.

8 Related Work

Crash consistent file systems. Many researches optimized crash-consistent storage systems in particular the journaling file systems [15, 17–20, 22, 25–27, 29, 36, 38, 43, 45]. These systems rely on the classic journaling over NVMe to provide failure atomicity, which waits for the completion of several PCIe round trips. Built atop ccNVMe, MQFS however achieves the atomicity guarantee by using only two persistent MMIOs. This increases the throughput as well as decreases the latency, since ccNVMe conceals the PCIe transfer and interrupt handler overhead to the file system for atomicity, and reduces the traffic (e.g., the commit record) over PCIe. We next discuss the comparison of the techniques (i.e., multi-queue journaling and metadata shadow paging) against these journaling file systems at length.

One category of these works [22, 29, 38, 39] is to improve the multicore scalability by partitioning the journal into multiple micro journals, which is similar to multi-queue journaling. The differences lie in the control flow of each micro journal and the coordination among micro journals.

First, IceFS [29], SpanFS [22], CCFS [39] and iJournaling [38] introduce extra write traffic (e.g., the commit record) and expensive ordering points (e.g., the `FLUSH`). Partitioning amplifies the extra write traffic, as it prevents multiple transactions from sharing a commit record to amortize the write traffic. MQFS however does not need ordering points and extra write traffic, by taking the free rides of the NVMe doorbell operations.

Second, the virtual journals of IceFS share a single physical log and need to suspend logging and serialize checkpointing when making free space. SpanFS allows each domain to perform checkpointing in parallel, but needs to maintain a global consistency over multiple domains (i.e., building connections across domains) at logging phase; this introduces extra synchronization overhead and write traffic. iJournaling preserves the legacy single compound journal, and may need to synchronize the compound journal and the file-level journals during checkpointing. MQFS instead uses scalable in-memory indexes for higher run-time scalability, and detects conflicts during checkpointing and recovery.

Another category [17, 27, 45] is to decouple the ordering from durability, thereby removing the ordering points of journaling. ccNVMe naturally decouples the transaction ordering from durability when queuing requests (the in-order completion in §4.4). ccNVMe further decouples a stronger property, the atomicity, from the durability, providing a clearer post-crash state. In addition, MQFS differs from them in the multicore scalability and the handling of page conflicts.

First, OptFS, BarrierFS and HoraeFS use only one thread to commit transactions. Hence, the throughput is bounded by the single thread and the latency increases due to communication (e.g., context switches) between the application and JBD2 thread. In contrast, MQFS scales the journaling to multiple hardware queues and performs it in the applications' context, to fully exploit the concurrency provided by the SSD and the multi-core CPUs of modern computers.

Second, in BarrierFS and HoraeFS, a running transaction with a conflict page can not be committed until the dependent transactions have made this page durable. This serializes the committing phase on transactions sharing the same page. MQFS uses metadata shadow paging to handle page conflict, thereby increasing the I/O concurrency of committing phase.

ScaleFS [15] logically records the directory changes in per-core operation logs for running transactions and merges these logs during committing. The hybrid-granularity journaling of CCFS associates byte-range changes with the running transaction and super-imposes the delta on the global block when transaction committing starts. These designs are orthogonal to the metadata shadow paging, and can be applied to MQFS to concurrently buffer the in-memory changes for the running transaction before committing.

ccNVMe does not provide isolation as in TxOS [40]. Instead, we leave the isolation to upper layer systems since there are various levels of isolation and different systems have their own demand for isolation. Providing isolation at upper layer systems is orthogonal to ccNVMe's design.

Transactional storage. A school of works provide atomicity interfaces at disk level [16, 21, 23, 30, 37, 41, 42]. They can achieve higher performance than ccNVMe by leveraging the features of storage media (e.g., copy-on-write of NAND flash). Yet, they require extensive hardware changes and it remains unclear whether similar designs can be applied to emerging Optane memory-based SSDs. ccNVMe requires the storage to enable the standard PMR, which is relatively simple (by using capacitor-backed DRAM or directly exposing a portion of the persistent Optane memory). ccNVMe does not limit the number of concurrent atomic writes as long as the hardware queue is available, while in transactional SSD this is limited by the internal resources (e.g., device-side CPU).

Byte-addressable SSD. Flatflash [12] exploits the byte addressability of SSD for a unified memory-storage hierarchy. Bae *et.al* [13] design an SSD with dual byte and block interfaces and simply put the database logging atop its SSD. Coinpurse [46] uses the PMR to expedite non-aligned writes from the file systems. Horae [27] builds dedicated ordering interfaces atop PMR. In this work, we use PMR to extend NVMe for efficient crash consistency.

9 Conclusion

We present ccNVMe, a new approach to achieve high performance and crash consistency simultaneously in storage systems. By coupling the crash consistency to the data dissemination and decoupling atomicity from durability, ccNVMe ensures atomicity guarantee with only two lightweight MMIOs and therefore improves the performance. We introduce MQFS to fully exploit the ccNVMe, showing that MQFS successfully saturates the SSD's bandwidth with fewer CPU cores and outperforms state-of-the-art file systems.

Acknowledgments

We sincerely thank our shepherd Jinyang Li and the anonymous reviewers for their valuable feedback. This work is supported by the National Key Research & Development Program of China (Grant No. 2018YFB1003301), the National Natural Science Foundation of China (Grant No. 61832011), Huawei Innovation Project (Grant No. YBN2019125112), and Sponsored by Zhejiang Lab (Grant No. 2020KC0AB03).

References

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

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

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

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

[5] [n.d.]. Intel Optane SSD DC P5800X Series. https://ark.intel.com/content/www/us/en/ark/products/201859/intel-optane-ssd-dc-p5800x-series-1-6tb-2-5in-pcie-x4-3d-xpoint.html.

[6] [n.d.]. Non-Volatile Memory express. https://nvmexpress.org.

[7] [n.d.]. NVMe 1.2 Spec. https://nvmexpress.org/wp-content/uploads/NVM_Express_1_2_Gold_20141209.pdf.

[8] [n.d.]. NVMe 1.4 Spec. https://nvmexpress.org/wp-content/uploads/NVM-Express-1_4-2019.06.10-Ratified.pdf.

[9] [n.d.]. NVMe 1.4 Spec Revision 1.4c. https://nvmexpress.org/wp-content/uploads/NVM-Express-1_4c-2021.06.28-Ratified.pdf.

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

[11] [n.d.]. PCI Express Base Specification Revision 3.1. https://pcisig.com/specifications/.

[12] Ahmed Abulila, Vikram Sharma Mailthody, Zaid Qureshi, Jian Huang, Nam Sung Kim, Jinjun Xiong, and Wen-mei Hwu. 2019. FlatFlash: Exploiting the Byte-Accessibility of SSDs within a Unified Memory-Storage Hierarchy. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 971–985. https://doi.org/10.1145/3297858.3304061

[13] Duck-Ho Bae, Insoon Jo, Youra Adel Choi, Joo-Young Hwang, Sangyeun Cho, Dong-Gi Lee, and Jaeheon Jeong. 2018. 2B-SSD: The Case for Dual, Byte- and Block-Addressable Solid-State Drives. In *Proceedings of the 45th Annual International Symposium on Computer Architecture* (Los Angeles, California) *(ISCA '18)*. IEEE Press, 425–438. https://doi.org/10.1109/ISCA.2018.00043

[14] Steve Best. 2000. JFS Log: How the Journaled File System Performs Logging.. In *Annual Linux Showcase & Conference*.

[15] Srivatsa S. Bhat, Rasha Eqbal, Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. 2017. Scaling a File System to Many Cores Using an Operation Log. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 69–86. https://doi.org/10.1145/3132747.3132779

[16] C. Chao, R. M. English, D. Jacobson, A. Stepanov, and J. Wilkes. 1997. Mime: a high performance parallel storage device with strong recovery guarantees.

[17] Vijay Chidambaram, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2013. Optimistic Crash Consistency. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. ACM, New York, NY, USA, 228–243. https://doi.org/10.1145/2517349.2522726

[18] Vijay Chidambaram, Tushar Sharma, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2012. Consistency without Ordering. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies* (San Jose, CA) *(FAST'12)*. USENIX Association, USA, 9.

[19] Christopher Frost, Mike Mammarella, Eddie Kohler, Andrew de los Reyes, Shant Hovsepian, Andrew Matsuoka, and Lei Zhang. 2007. Generalized File System Dependencies. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles* (Stevenson, Washington, USA) *(SOSP '07)*. Association for Computing Machinery, New York, NY, USA, 307–320. https://doi.org/10.1145/1294261.1294291

[20] Yige Hu, Zhiting Zhu, Ian Neal, Youngjin Kwon, Tianyu Cheng, Vijay Chidambaram, and Emmett Witchel. 2018. TxFS: Leveraging File-System Crash Consistency to Provide ACID Transactions. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 879–891. https://www.usenix.org/conference/atc18/presentation/hu

[21] M. Kaashoek and Wilson Hsieh. 2001. Logical Disk: A Simple New Approach to Improving File System Performance. (03 2001).

[22] Junbin Kang, Benlong Zhang, Tianyu Wo, Weiren Yu, Lian Du, Shuai Ma, and Jinpeng Huai. 2015. SpanFS: A Scalable File System on Fast Storage Devices. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference* (Santa Clara, CA) *(USENIX ATC '15)*. USENIX Association, Berkeley, CA, USA, 249–261. http://dl.acm.org/citation.cfm?id=2813767.2813786

[23] Woon-Hak Kang, Sang-Won Lee, Bongki Moon, Gi-Hwan Oh, and Changwoo Min. 2013. X-FTL: Transactional FTL for SQLite Databases. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (New York, New York, USA) *(SIGMOD '13)*. Association for Computing Machinery, New York, NY, USA, 97–108. https://doi.org/10.1145/2463676.2465326

[24] Jongseok Kim, Cassiano Campes, Joo-Young Hwang, Jinkyu Jeong, and Euiseong Seo. 2021. Z-Journal: Scalable Per-Core Journaling. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, 893–906. https://www.usenix.org/conference/atc21/presentation/kim-jongseok

[25] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. 2017. Strata: A Cross Media File System. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 460–477. https://doi.org/10.1145/3132747.3132770

[26] Gyusun Lee, Seokha Shin, Wonsuk Song, Tae Jun Ham, Jae W. Lee, and Jinkyu Jeong. 2019. 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)*. USENIX Association, Renton, WA, 603–616. https://www.usenix.org/conference/atc19/presentation/lee-gyusun

[27] Xiaojian Liao, Youyou Lu, Erci Xu, and Jiwu Shu. 2020. Write Dependency Disentanglement with HORAE. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 549–565. https://www.usenix.org/conference/osdi20/presentation/liao

[28] Xiaojian Liao, Youyou Lu, Erci Xu, and Jiwu Shu. 2021. Max: A Multicore-Accelerated File System for Flash Storage. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, 877–891. https://www.usenix.org/conference/atc21/presentation/liao

[29] Lanyue Lu, Yupu Zhang, Thanh Do, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2014. Physical Disentanglement in a Container-based File System. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (Broomfield, CO) *(OSDI'14)*. USENIX Association, Berkeley, CA, USA, 81–96. http://dl.acm.org/citation.cfm?id=2685048.2685056

[30] Youyou Lu, Jiwu Shu, Jia Guo, Shuai Li, and Onur Mutlu. 2013. LightTx: A lightweight transactional design in flash-based SSDs to support flexible transactions. In *2013 IEEE 31st International Conference on Computer Design (ICCD)*. 115–122. https://doi.org/10.1109/ICCD.2013.6657033

[31] Youyou Lu, Jiwu Shu, and Wei Wang. 2014. ReconFS: A Reconstructable File System on Flash Storage. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies* (Santa Clara, CA) *(FAST'14)*. USENIX Association, USA, 75–88.

[32] Youyou Lu, Jiwu Shu, and Jiacheng Zhang. 2019. Mitigating Synchronous I/O Overhead in File Systems on Open-Channel SSDs. *ACM Trans. Storage* 15, 3, Article 17 (May 2019), 25 pages. https://doi.org/10.1145/3319369

[33] Youyou Lu, Jiwu Shu, and Weimin Zheng. 2013. Extending the Lifetime of Flash-based Storage through Reducing Write Amplification from File Systems. In *11th USENIX Conference on File and Storage Technologies (FAST 13)*. USENIX Association, San Jose, CA, 257–270. https://www.usenix.org/conference/fast13/technical-sessions/presentation/lu_youyou

[34] Microsoft. [n.d.]. Windows NTFS. https://en.wikipedia.org/wiki/NTFS.

[35] Jayashree Mohan, Ashlie Martinez, Soujanya Ponnapalli, Pandian Raju, and Vijay Chidambaram. 2018. Finding Crash-Consistency Bugs with Bounded Black-Box Crash Testing. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 33–50. https://www.usenix.org/conference/osdi18/presentation/mohan

[36] Edmund B. Nightingale, Kaushik Veeraraghavan, Peter M. Chen, and Jason Flinn. 2006. Rethink the Sync. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation* (Seattle, Washington) *(OSDI '06)*. USENIX Association, USA, 1–14.

[37] Xiangyong Ouyang, David Nellans, Robert Wipfel, David Flynn, and Dhabaleswar K. Panda. 2011. Beyond Block I/O: Rethinking Traditional Storage Primitives. In *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture (HPCA '11)*. IEEE Computer Society, USA, 301–311.

[38] Daejun Park and Dongkun Shin. 2017. iJournaling: Fine-grained Journaling for Improving the Latency of Fsync System Call. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference* (Santa Clara, CA, USA) *(USENIX ATC '17)*. USENIX Association, Berkeley, CA, USA, 787–798. http://dl.acm.org/citation.cfm?id=3154690.3154764

[39] Thanumalayan Sankaranarayana Pillai, Ramnatthan Alagappan, Lanyue Lu, Vijay Chidambaram, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2017. Application Crash Consistency and Performance with CCFS. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*. USENIX Association, Santa Clara, CA, 181–196. https://www.usenix.org/conference/fast17/technical-sessions/presentation/pillai

[40] Donald E. Porter, Owen S. Hofmann, Christopher J. Rossbach, Alexander Benn, and Emmett Witchel. 2009. Operating System Transactions. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (Big Sky, Montana, USA) *(SOSP '09)*. Association for Computing Machinery, New York, NY, USA, 161–176. https://doi.org/10.1145/1629575.1629591

[41] Vijayan Prabhakaran, Thomas L. Rodeheffer, and Lidong Zhou. 2008. Transactional Flash. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (San Diego, California) *(OSDI'08)*. USENIX Association, USA, 147–160.

[42] Russell Sears and Eric Brewer. 2006. Stasis: Flexible Transactional Storage. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation* (Seattle, Washington) *(OSDI '06)*. USENIX Association, USA, 29–44.

[43] Yongseok Son, Sunggon Kim, Heon Young Yeom, and Hyuck Han. 2018. High-performance Transaction Processing in Journaling File Systems. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies* (Oakland, CA, USA) *(FAST'18)*. USENIX Association, Berkeley, CA, USA, 227–240. http://dl.acm.org/citation.cfm?id=3189759.3189781

[44] Adam Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. 1996. Scalability in the XFS File System.. In *USENIX Annual Technical Conference*, Vol. 15.

[45] Youjip Won, Jaemin Jung, Gyeongyeol Choi, Joontaek Oh, Seongbae Son, Jooyoung Hwang, and Sangyeun Cho. 2018. Barrier-enabled IO Stack for Flash Storage. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies* (Oakland, CA, USA) *(FAST'18)*. USENIX Association, Berkeley, CA, USA, 211–226. http://dl.acm.org/citation.cfm?id=3189759.3189779

[46] Z. Yang, Y. Lu, E. Xu, and J. Shu. 2020. CoinPurse: A Device-Assisted File System with Dual Interfaces. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*. 1–6.

[47] Jiacheng Zhang, Jiwu Shu, and Youyou Lu. 2016. ParaFS: A Log-Structured File System to Exploit the Internal Parallelism of Flash Devices. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. USENIX Association, Denver, CO, 87–100. https://www.usenix.org/conference/atc16/technical-sessions/presentation/zhang