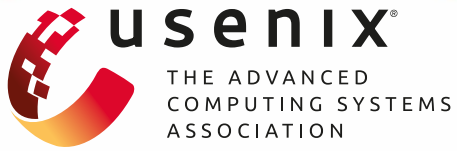


CrossFS: A Cross-layered Direct-Access File System

Yujie Ren, *Rutgers University;* Changwoo Min, *Virginia Tech;*
Sudarsun Kannan, *Rutgers University*

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

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

Open access to the Proceedings of the
14th USENIX Symposium on Operating
Systems Design and Implementation
is sponsored by USENIX

CrossFS: A Cross-layered Direct-Access File System

Yujie Ren
Rutgers University
yujie.ren@rutgers.edu

Changwoo Min
Virginia Tech
changwoo@vt.edu

Sudarsun Kannan
Rutgers University
sudarsun.kannan@rutgers.edu

Abstract

We design CrossFS, a cross-layered direct-access file system disaggregated across user-level, firmware, and kernel layers for scaling I/O performance and improving concurrency. CrossFS is designed to exploit host- and device-level compute capabilities. For concurrency with or without data sharing across threads and processes, CrossFS introduces a file descriptor-based concurrency control that maps each file descriptor to one hardware-level I/O queue. This design allows CrossFS's firmware component to process disjoint access across file descriptors concurrently. CrossFS delegates concurrency control to powerful host-CPUs, which convert the file descriptor synchronization problem into an I/O queue request ordering problem. To guarantee crash consistency in the cross-layered design, CrossFS exploits byte-addressable nonvolatile memory for I/O queue persistence and designs a lightweight firmware-level journaling mechanism. Finally, CrossFS designs a firmware-level I/O scheduler for efficient dispatch of file descriptor requests. Evaluation of emulated CrossFS on storage-class memory shows up to 4.87x concurrent access gains for benchmarks and 2.32x gains for real-world applications over the state-of-the-art kernel, user-level, and firmware file systems.

1 Introduction

We have finally entered an era where storage access latency is transitioning from milliseconds to microseconds [3,52,62,68]. While modern applications strive to increase I/O parallelism, storage software bottlenecks such as system call overheads, coarse-grained concurrency control, and the inability to exploit hardware-level concurrency continues to impact I/O performance. Several kernel-level, user-level, and firmware-level file systems have been designed to benefit from CPU parallelism [65, 66], direct storage access [22, 31, 40], or computational capability in the storage hardware [33, 56]. However, these approaches are designed in isolation and fail to exploit modern, ultra-fast storage hardware.

Kernel-level file systems (Kernel-FS) satisfy *fundamental file system guarantees* such as integrity, consistency, durability, and security. Despite years of research, Kernel-FS designs continue to suffer from **three main bottlenecks**. First, applications must enter and exit the OS for performing I/O, which could increase latency by 1-4μs [31, 68]. Recently found security vulnerabilities have further amplified such costs [25, 39, 47]. Second, even state-of-the-art designs enforce unnecessary serialization (e.g., inode-level read-write lock) when accessing disjoint portions of data in a file leading to high concurrent access overheads [48]. Third, Kernel-FS designs *fail to fully exploit* storage hardware-level capabilities such as compute, thousands of I/O queues, and firmware-level scheduling, ultimately impacting I/O latency, throughput, and concurrency in I/O-intensive applications [5, 8, 9, 45, 69].

As an alternative design point, there is increasing focus towards designing user-level file systems (User-FS) for direct storage access bypassing the OS [17, 22, 31, 40, 52, 65]. However, satisfying the fundamental file system guarantees from untrusted user-level is challenging [33]. While these designs have advanced the state of the art, some designs bypass the OS only for data-plane operations (without data sharing) [17, 31, 52]. In contrast, others provide full direct access by either sidestepping or inheriting coarse-grained and suboptimal concurrency control across threads and processes [22, 40], or even compromise correctness [65]. Importantly, most User-FS designs fail to exploit the hardware capabilities of modern storage.

At the other extreme is the exploration of firmware-level file systems (Firmware-FS) that embed the file system into the device firmware for direct-access [33, 56]. The Firmware-FS acts as a central entity to satisfy fundamental file system properties. Although an important first step towards utilizing storage-level computational capability, current designs miss out on benefiting from host-level multi-core parallelism. Additionally, these designs inherit inode-centric design for request queuing, concurrency control, and scheduling, leading to poor I/O scalability.

In summary, current User-FS, Kernel-FS, and Firmware-FS

designs *lack a synergistic design across the user, the kernel, and the firmware layers*, which is critical for achieving direct storage access and scaling concurrent I/O performance without compromising fundamental file system properties.

Our Approach - Cross-layered File System. To address the aforementioned bottlenecks, we propose **CrossFS**, a cross-layered direct-access file system that provides scalability, high concurrent access throughput, and lower access latency. CrossFS achieves these goals by disaggregating the file system across the user-level, the device firmware, and the OS layer, thereby exploiting the benefits of each layer. The firmware component (FirmFS) is the heart of the file system enabling applications to directly access the storage without compromising fundamental file system properties. The FirmFS taps into storage hardware's I/O queues, computational capability, and I/O scheduling capability for improving I/O performance. The user-level library component (LibFS) provides POSIX compatibility and handles concurrency control and conflict resolution using the host-level CPUs (host-CPUs). The OS component sets up the initial interface between LibFS and FirmFS (e.g., I/O queues) and converts software-level access control to hardware security control.

Scalability. File system disaggregation alone is insufficient for achieving I/O scalability, which demands revisiting file system concurrency control, reducing journaling cost, and designing I/O scheduling that matches the concurrency control. We observe that file descriptors (and not inode) are a natural abstraction of access in most concurrent applications, where threads and processes use independent file descriptors to access/update different regions of shared or private files (for example, RocksDB maintains 3.5K open file descriptors). Hence, for I/O scalability, in CrossFS, we introduce file descriptor-based concurrency control, which allows threads or processes to update or access non-conflicting blocks of a file simultaneously.

Concurrency Control via Queue Ordering. In CrossFS, file descriptors are mapped to dedicated hardware I/O queues to exploit storage hardware parallelism and fine-grained concurrency control. All non-conflicting requests (i.e., requests to different blocks) issued using a file descriptor are added to a file descriptor-specific queue. In contrast, conflicting requests are ordered by using a single queue. This provides an opportunity for device-CPUs and FirmFS to dispatch requests concurrently with almost zero synchronization between host and device-CPUs. For conflict resolution and ordering updates to blocks across file descriptors, CrossFS uses a per-inode interval tree [7], interval tree read-write semaphore (interval tree `rw-lock`), and global timestamps for concurrency control. However, *unlike current file systems that must hold inode-level locks* until request completion, CrossFS only acquires interval tree `rw-lock` for request ordering to FD-queues. In short, CrossFS concurrency design *turns the file synchronization problem into a queue ordering problem*.

CrossFS Challenges. Moving away from an inode-centric to a file descriptor-centric design introduces CrossFS-specific challenges. First, using fewer and wimpier device-CPUs for conflict resolution and concurrency control impacts performance. Second, mapping a file descriptor to an I/O queue (a device-accessible DMA memory buffer) increases the number of queues that CrossFS must manage, potentially leading to data loss after a crash. Finally, overburdening device-CPUs for serving I/O requests across hundreds of file descriptor queues could impact performance, specifically for blocking I/O operations (e.g., `read, fsync`).

Host Delegation. To overcome the challenge of fewer (and wimpier) device-CPUs, CrossFS utilizes the cross-layered design and delegates the responsibility of request ordering to host-CPUs. The host-CPUs order data updates to files they have access to, whereas FirmFS is ultimately responsible for updating and maintaining metadata integrity, consistency, and security with POSIX-level guarantees.

Crash-Consistency and Scheduler. To handle crash consistency and protect data loss across tens and possibly hundreds of FD-queues, CrossFS uses byte-addressable, persistent NVMs as DMA'able and append-only FD-queues from which FirmFS can directly fetch requests or pool responses. CrossFS also designs low-cost data journaling for crash-consistency of firmware file system state (§4.4). Finally, for efficient scheduling of device-CPUs, CrossFS smashes traditional two-level I/O schedulers spread across the host-OS and the firmware into one FirmFS scheduler. CrossFS also equips the scheduler with policies that enhance file descriptor-based concurrency.

Evaluation of our CrossFS prototype implemented as a device-driver and emulated using Intel Optane DC memory [3] shows significant concurrent access performance gains with or without data sharing compared to state-of-the-art Kernel-FS [64, 66], User-FS [31, 40], and Firmware-FS [33] designs. The performance gains stem from reducing system calls, file descriptor-level concurrency, work division across host and device-CPUs, low-overhead journaling, and improved firmware-level scheduling. The concurrent access microbenchmarks with data sharing across threads and processes show up to 4.87X gains. The multithreaded Filebench [59] macrobenchmark workloads without data sharing show up to 3.58X throughput gains. Finally, widely used real-world applications such as RocksDB [9] and Redis [8] show up to 2.32X and 2.35X gains, respectively.

2 Background and Related Work

Modern ultra-fast storage devices provide high bandwidth (8-16 GB/s) and two orders of lower access latency (< 20μsec) [15, 67] compared to HDD storage. The performance benefits can be attributed to innovation in faster storage hardware and access interface (e.g., PCIe support), increase in storage-level compute (4-8 CPUs, 4-8 GB DRAM,

64K I/O queues) [27], fault protection equipped with capacitors [57, 58], and evolving support for storage programmability. In recent years, file systems have evolved to exploit high-performance modern storage devices. However, for applications to truly benefit from modern storage, file system support for scalable concurrency and efficient data sharing is critical [21, 26, 42]. Next, we discuss kernel, user-level, and firmware-level file system innovations and their implications.

Kernel-level File Systems (Kernel-FS). Several new kernel file systems have been designed to exploit the capabilities of modern storage devices [15, 23, 43, 64, 66]. For example, F2FS exploits multiple queues of an SSD and employs a log-structured design [43]. LightNVM moves the FTL firmware code to the host and customizes request scheduling [15]. File systems such as PMFS [23], DAX [64], and NOVA [66] exploit NVM's byte-addressability; they support block-based POSIX interface but replace block operations with byte-level loads and stores. To improve concurrent access performance, file systems such as ext4 introduce inode-level read-write semaphores (`rw-lock`) for improving read concurrency when sharing files [24]. Alternatively, user-level storage access frameworks such as SPDK use NVMe-based I/O command queues to provide direct I/O operations. However, these frameworks only support simple block operations as opposed to a POSIX interface [29].

User-level File Systems (User-FS). There is a renewed focus to bypass the OS and directly access storage hardware from a user-level library. However, managing a file system from an untrusted library introduces a myriad of challenges, which include atomicity, crash consistency, and security challenges [40, 52, 60]. Some prior designs bypass the OS for data plane operations but enter the OS for control plane operations [31, 52, 60]. In contrast, approaches such as Strata [40] and ZoFS [22] provide direct access for control and data plane operations by managing data and metadata in a user-level library. For example, Strata [40] buffers data and metadata updates to a per-process, append-only log, which is periodically committed to a shared area using a trusted file system server. More recently, ZoFS, designed for persistent memory technologies, uses virtual memory protection to secure access to a user-level buffer holding data and metadata updates. Unfortunately, all these approaches require high-overhead concurrency control and use coarse-grained locks that do not scale [31]. For example, in Strata, a process must acquire an inode lease from the trusted file system server before accessing a file [40, 60].

Firmware File Systems (Firmware-FS). After two decades of seminal work on programmable storage [17, 33, 54, 56], prior research takes a radical approach of offloading either the entire file system [33] or a part of it [17, 54] into the device firmware. The firmware approach allows applications to bypass the OS for both control and data plane operations. Firmware-FS acts as a central entity to coordinate updates to file system metadata and data without compromising crash-consistency by using the device-CPUs and the device-RAM for per-inode I/O queues. For security and permission checks, systems such as DevFS [33] rely on the host OS to update a device-level credential table with process permissions. For crash consistency, the power-loss capacitors could be used to flush in-transit updates after a system failure. Insider [56] explores the use of FPGAs for file system design, whereas other efforts have focused on using FPGAs [70] to accelerate key-value stores. Unfortunately, both DevFS and Insider handle concurrency using inode-level locks, limiting file system concurrency and scalability.

ext4-DAX	Strata	DevFS	CrossFS
21.38%	26.57%	27.06%	9.99%

Table 1: Time spent on inode-level lock.

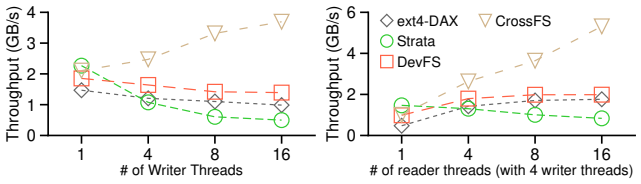
File System Scalability. Several kernel-level file system scalability designs have been explored in the past. SpanFS [32] shares files and directories across cores at a coarse granularity, requiring developers to distribute I/O. ScaleFS [13] decouples the in-memory file system from the on-disk file system and uses per-core operation logs to achieve high concurrency. FLEX [65] moves file operations to a user-level library and modifies the Linux VFS layer. Fine-grained locking mechanisms such as CST-semaphore [36] and range lock [42] are applied to current kernel-level file systems to improve concurrency. However, all the above approaches either lack direct access or require inode-level locks and fail to utilize host and device-level compute capabilities.

3 Motivation

Unfortunately, state-of-the-art Kernel-FS [64, 66], User-FS [31, 40], and Firmware-FS [33] designs suffer from three prominent limitations. First, they lack a synergistic design that could benefit from the combined use of host and device resources, such as host and device-level CPUs, and thousands of hardware I/O queues. Second, the use of an inode-centric design limits concurrent access scalability. Third, for file sharing across processes, applications must trap into an OS for control or data plane or both [60].

To briefly illustrate the poor concurrent access scalability in state-of-the-art file system designs, we conduct an experiment where readers and writers perform random-but-disjoint block accesses on a 12GB file. For our analysis, we compare User-FS Strata [40], Kernel-FS Linux ext4-DAX [64], Firmware-FS DevFS [33], and the proposed CrossFS. Figure 1a shows the aggregate write throughput when multiple writers concurrently update a shared file. The x-axis varies the writer thread count. In Figure 1b, we restrict the number of writers to *4* and increase readers in the x-axis.

Concurrent Write Performance: As shown in Figure 1a, when sharing files across concurrent writers, the throughput substantially reduces for all approaches except CrossFS. ext4-



(a) Concurrent Writers **(b)** Concurrent Readers

Figure 1: Concurrent Write and Read Throughput. (a) shows the aggregated write throughput when concurrent writers update disjoint random blocks of a 12GB file; (b) shows aggregated read throughput when there are 4-concurrent writers.

DAX and NOVA use inode-level `rw-lock`, which prevents writers from updating disjoint blocks of a file. Strata does not scale beyond four threads because it uses an inode-level mutex. Additionally, Strata uses a private NVM log to buffer data and metadata (inode) updates, and the log fills up frequently, consequently stalling write operations.

Sharing across Reader and Writer Threads: Figure 1b and Table 1 show the aggregated read throughput and the execution time spent on an inode-level `rw-lock`. The read performance does not scale (in the presence of writers) even when accessing disjoint blocks, mainly due to the inode-level `rw-lock`. For Strata, by the time readers acquire a mutex for the private log, the log contents could be flushed, forcing readers to access data using Kernel-FS. We see similar performance bottlenecks when using concurrent processes instead of threads for ext4-DAX and NOVA due to their inode-centric concurrency control. For Strata, the performance degrades further; the reader processes starve until writers flush their private log to the shared area. These issues highlight the complexities of scaling concurrent access performance in Kernel-FS and User-FS designs. The observations hold for other user-level file systems such as ZoFS [22]. Finally, as shown in Figure 1a and Figure 1b, CrossFS outperforms other file systems with its fine-grained file descriptor-based concurrency. We will next discuss the design details of CrossFS.

4 Design of CrossFS

CrossFS is a cross-layered design that disaggregates the file system to exploit the capabilities of userspace, firmware, and OS layers. CrossFS also achieves high end-to-end concurrency in the user-level library, I/O request queues, and firmware-level file system, with or without file sharing across threads and processes. Finally, CrossFS reduces system call cost, provides lightweight crash consistency, and efficient device-CPU scheduling. We observe that applications that perform conflicting updates to the same blocks of files, automatically resort to protecting file descriptors with application-level locking [2, 5, 9]. Hence, for concurrency and file sharing, CrossFS uses file descriptors as a fundamental unit of synchronization, enabling threads and processes with separate file descriptors to update disjoint blocks of files concurrently. CrossFS assigns each file descriptor a dedicated I/O queue (**FD-queue**) and adds non-conflicting (disjoint) block requests across descriptors to their independent FD-queue, whereas serializing conflicting block updates or access to a single FD-queue. Consequently, CrossFS converts the file synchronization problem to an I/O request ordering problem, enabling device-CPUs to dispatch requests across FD-queues concurrently. We outline the design principles and then follow it up by describing the details of file descriptor-based concurrency mechanism, crash consistency support, I/O scheduling, and security.

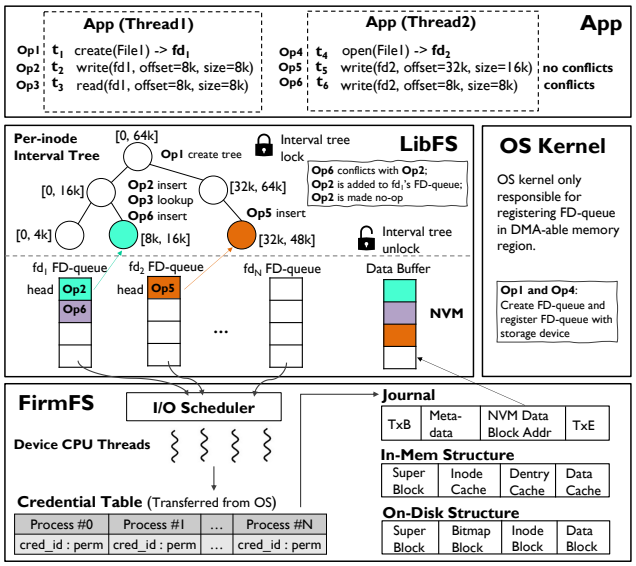


Figure 2: CrossFS Design Overview. The host-level LibFS converts POSIX I/O calls into FirmFS commands, manages FD-queues, and interval tree for checking block conflicts and ordering requests in FD-queues. FirmFS implements the file system, journaling, and scheduler and concurrently processes requests across FD-queues. The OS component is responsible for the FD-queue setup and updates the device credential table with host-level permission information. We show a running example of two instances sharing a file; Op1 to 6 show request execution with global timestamps t1 to t6. Op6 conflicts with Op2, so Op6 is added to the same FD-queue as Op2 using an interval tree.

4.1 CrossFS Design Principles

CrossFS adapts the following key design principles:

Principle 1: For achieving high performant direct-I/O, disaggregate file system across user-level, firmware, and OS layers to exploit host and device-level computing capabilities.

Principle 2: For fine-grained concurrency, align each file descriptor to an independent hardware I/O queue (FD-queue), and design concurrency control focused around the file descriptor abstraction.

Principle 3: To efficiently use device-CPUs, merge software and hardware I/O schedulers into a single firmware scheduler, and design scheduling policies to benefit from the file-descriptor design.

Principle 4: For crash consistency in a cross-layered design,

protect the in-transit user data and the file system state by leveraging persistence provided by byte-addressable NVMs.

4.2 CrossFS Layers

CrossFS enables unmodified POSIX-compatible applications to benefit from direct storage access. As shown in Figure 2, CrossFS comprises of a user-level library (LibFS), a firmware file system component (FirmFS), and an OS component.

User-space Library Component (LibFS). Applications are linked to LibFS, which intercepts POSIX I/O calls and converts them to FirmFS I/O commands. As shown in Figure 2, when opening a file, LibFS creates an FD-queue by requesting the OS to allocate a DMA'able memory region on NVM and registering the region with FirmFS. Each I/O request is added to a file descriptor-specific FD-queue when there are no block-level conflicts. In the presence of block conflicts, the conflicting requests are serialized to the same FD-queue. To identify block conflicts, in CrossFS, we use a per-inode interval tree with range-based locking. The conflict resolution using the interval tree is delegated to the host-CPU when sharing data across multiple threads, and for inter-process file sharing, interval tree updates are offloaded to FirmFS (and the device-CPUs).

Firmware File System Component (FirmFS). FirmFS is responsible for fetching and processing I/O requests from FD-queues. Internally, FirmFS's design is similar to a traditional file system with in-memory and on-disk metadata structures, including a superblock, bitmap blocks, and inode and data blocks. FirmFS also supports data and metadata journaling using a dedicated journal space on the device, as shown in Figure 2. FirmFS fetches I/O requests from FD-queues and updates in-memory metadata structures (stored in the device-level RAM). For crash-consistency, FirmFS journals the updates to the storage and then checkpoints them (§4.4). The mounting process for FirmFS is similar to a traditional file system: finding the superblock, followed by the root directory. To schedule requests from FD-queues, FirmFS also implements a scheduler (§4.5). Finally, FirmFS implements a simple slab allocator for managing device memory.

OS Component. The OS component is mainly used for setting up FD-queues by allocating DMA'able memory regions, mounting of CrossFS, and garbage collecting resources. The OS component also converts process-level access controls to device-level credentials for I/O permission checks without requiring applications to trap into the OS (§4.6).

4.3 Scaling Concurrent Access

We next discuss CrossFS's file descriptor-based concurrency design that scales without compromising correctness and sharply contrasts with prior inode-centric designs.

4.3.1 Per-File Descriptor I/O Queues

Modern NVMe devices support up to 64K I/O queues, which can be used by applications to parallelize I/O requests. To exploit this hardware-level I/O parallelism feature, CrossFS aligns each file descriptor with a dedicated I/O queue (with a configurable limit on the maximum FD-queues per inode). As shown in Figure 2, during file open (*Op1*), LibFS creates an FD-queue (I/O request + data buffer) by issuing an IOCTL call to the OS component, which reserves memory for an FD-queue, and registers the FD-queue's address with FirmFS. For handling uncommon scenarios where the number of open file descriptors could exceed the available I/O queues (e.g., 64K I/O queues in NVMe), supporting FD-queue reuse and multiplexing could be useful. For reuse, CrossFS must service pending requests in an FD-queue, and clear its data buffers. For multiplexing, CrossFS must implement a fair queue sharing policy. While CrossFS currently supports queue reuse after a file descriptor is closed, our future work will address the support for FD-queue multiplexing.

4.3.2 Concurrency Constraints

CrossFS provides correctness and consistency guarantees of a traditional kernel-level file system (e.g., ext4). We first define the constraints that arise as a part of CrossFS's file-descriptor design, and then describe how CrossFS satisfies these constraints.

- Constraint 1: Read requests (commands) entering a device at timestamp T must fetch the most recent version of data blocks, which are either buffered in FD-queues or stored in the storage. The *timestamp* refers to a globally incrementing atomic counter added to an I/O command in the FD-queues.
- Constraint 2: For concurrent writes to conflicting (same) blocks across file descriptors, the most recent write at T_j can overwrite a preceding write at T_i where $i < j$.

4.3.3 Delegating Concurrency Control to Host-CPUs

Handling conflict resolution across several threads that concurrently update a shared file could be compute-heavy. As a consequence, using fewer (or wimpier) device-CPUs could pose a scalability challenge. Hence, in CrossFS, for data sharing across multiple threads (a common scenario in most applications), we exploit our cross-layered design and delegate concurrency control to host-CPUs without impacting file system correctness. In the case of data sharing across processes, for simplicity and security, we offload concurrency control to FirmFS (discussed in §4.6). We first discuss the data structure support and then discuss CrossFS support for concurrent reader and writer threads with host-delegated concurrency control.

Request Timestamp. Before adding a request to an FD-queue, LibFS tags each I/O request with a globally incrementing hardware TSC-based timestamp. The timestamp is used to resolve ordering conflicts across writers and fetch the

most recent data for readers. The use of TSC-based timestamp for synchronization has been widely studied by prior research [35, 38, 46, 55].

Per-Inode Interval Tree. To resolve conflicts across concurrent writers and readers, we use a per-inode interval tree. An interval tree is an augmented red-black tree, indexed by interval ranges (low, high) and allows quick lookup for any overlapping or exact matching block range [7, 28].

In CrossFS, an inode-level interval tree is used for conflict resolution, i.e., to identify pending I/O requests in FD-queues that conflict with a newly issued I/O request. The OS allocates the interval tree in a DMA'able region during file creation and returns interval tree's address to LibFS. For each update request, LibFS adds a new interval tree node indexed by the request's block range. The nodes store a timestamp and a pointer to the newly added I/O request in the FD-queue. The interval tree's timestamp (TSC) is checked for following cases: (1) for writes to identify conflicting updates to block(s) across different FD-queues; (2) for reads to fetch the latest uncommitted (in-transit) updates from an FD-queue; and (3) for file commits (`fsync`). The interval tree items are deleted after FD-queue entries are committed to the storage blocks (discussed in 4.3.6).

Threads that share a file also share inode-level interval tree. For files shared within a process, interval tree is updated and accessed just by LibFS. The updates to interval tree are protected using a read-write lock (`rw-lock`), held only at the time request insertion to FD-queue.

4.3.4 Supporting Concurrent Readers and a Writer

CrossFS converts a file synchronization problem to a queue ordering problem. Hence, the first step towards concurrent write and read support for a file is to identify a file descriptor queue to which a request must be added. To allow concurrent readers and one writer to share a file, requests are ordered using a global timestamp.

For each write request, LibFS acquires an inode's interval tree `rw-lock` and atomically checks the interval tree for conflicts. A conflict exists if there are unprocessed prior FD-queue write requests that update the same block(s). After detecting a conflict, LibFS adds a global timestamp to the write request and orders it in the same FD-queue of the prior request. LibFS also updates the interval tree's node to point to the current request and releases the interval tree's `rw-lock`. For example, in Figure 2, for *Op6*, Thread 2's write to offset 8k conflicts with Thread 1's *Op2* buffered in *fd1*'s queue with an earlier timestamp. Hence, Thread 2's request is added to *fd1's* FD-queue for ordering the updates.

Note that, while prior systems acquire the inode `rw-lock` until request completion, *CrossFS acquires inode-level interval tree `rw-lock` only until a request is ordered to the FD-queue*. This substantially reduces the time to acquire interval tree `rw-lock` compared to prior inode-centric designs (see §6). To reduce the latency when reading block(s) from a file, LibFS uses the interval tree to identify conflicting writes to the same block(s) buffered in the FD-queue. If a conflict exists, LibFS (i.e., the host-CPU) returns the requested blocks from FD-queue, thereby reducing FirmFS work. For example, as shown in Figure 2, for *Thread 1's Op3*, LibFS checks the file descriptors *fd1*'s FD-queue for a preceding write to the same block and returns the read from the FD-queue. Read operations also acquire interval tree `rw-lock` before lookup, and do not race with an on-going write or a FD-queue cleanup (which also acquires interval tree `rw-lock`).

4.3.5 Supporting Concurrent Writers

CrossFS supports concurrent writers to share files without enforcing synchronization between host and device-CPUs.

Non-conflicting Concurrent Writes. When concurrent writers share a file, non-conflicting (disjoint) writes to blocks across file descriptors are added to their respective FD-queues, tagged with a global timestamp, and added to an interval tree node for the corresponding block range. In Figure 2, non-conflicting *Op2* in *fd1* and *Op5* in *fd2* are added to separate FD-queues and can be concurrently processed by FirmFS.

Conflicting Concurrent Writes. The number of conflicting blocks across concurrent writers could vary.

(1) Single-block conflict. A single-block conflict refers to a condition where a conflict occurs for one block. When a current writer updates one block (say block k) at a timestamp (say j), the write request (W_jB_k) is atomically checked against the interval tree for preceding non-committed writes. If an earlier write (say W_iB_k, where $i < j$) exists, CrossFS adds the request (W_jB_k) to the queue, marks the earlier request (W_iB_k) as a *no-op*, and updates the interval tree to point to the new request (W_jB_k), thereby avoiding an extra write operation. For example, in Figure 2, the later request *Op6* is added to the FD-queue of *Op2*, and *Op2* is made a no-op.

(2) Multi-block Write conflict. CrossFS must handle multi-block conflicts for correctness, although uncommon in real-world applications. First, when a writer's request to update one block ($W_{i+1}B_k$) conflicts with an earlier write that updates multiple blocks (say $W_iB_kB_{k+1}B_{k+2}$), the prior request cannot be made a *no-op*. Hence, LibFS adds the new request ($W_{i+1}B_k$) to the same FD-queue of the prior request ($W_iB_kB_{k+1}B_{k+2}$) and inserts a child node (sub-tree) to the interval tree (B_k, B_{k+2} range) with a pointer to the newly added FD-queue request. This technique is used even for a multi-block request that conflicts with a prior single or multi-block request or when conflicting small writes update different ranges in the single block. Although this approach would incur multiple writes for some blocks, it simplifies handling multi-block conflicts that are uncommon. For rare cases, where multiple blocks of the new request ($W_iB_kB_{k+1}B_{k+2}$) conflicts with blocks across multiple FD-queues (say W_jB_k and W_nB_k+2 with j and n in different queues), we treat these as an inode-level barrier operation, which we discuss next.

(3) Multi-block Concurrent Reads. Concurrent readers always fetch the most recent blocks for each update using the interval tree. For blocks with partial updates across requests, LibFS patches these blocks by walking through the sub-tree of an interval tree range.

Support for File Appends. Several classes of applications (e.g., RocksDB [9] evaluated in this paper) extensively use file appends for logging. In CrossFS, file appends are atomic operation. To process appends, FirmFS uses an atomic transaction to allocate data blocks and update metadata blocks (i.e., inode). We use and extend atomic transaction support for `O_APPEND` from the prior NVM-based file system [23].

4.3.6 File Commit (fsync) as a Barrier

File commit (`fsync`) operations in POSIX guarantee that a successful commit operation to a file descriptor commits all updates to a file preceding the `fsync` across all file descriptors. However, in CrossFS, requests across file descriptors are added to their own FD-queue and processed in parallel by FirmFS, which could break POSIX's `fsync` guarantee. Consider a scenario where a `fsync` request added to an empty FD-queue getting dispatched before earlier pending writes in other FD-queues. To avoid these issues, CrossFS treats file commit requests as a special **barrier** operation. The `fsync` request is tagged as a barrier operation and atomically added to *all FD-queues of an inode* by acquiring an interval tree `rw-lock`. The non-conflicting requests in FD-queues are concurrently dispatched by multiple device-CPUs to reduce the cost of a barrier. We study the performance impact of `fsync` operations in §6.

4.3.7 Metadata-heavy Operations

CrossFS handles metadata-heavy operations with a file descriptor (e.g., close, unlink, file rename) and without a file descriptor (e.g., mkdir, directory rename) differently. We next discuss the details.

Metadata Operations with a File Descriptor. Metadata-heavy operations include inode-level changes, so adding these requests to multiple FD-queues and concurrently processing them could impact correctness. Additionally, concurrent processing is prone to crash consistency bugs [18, 31, 53]. To avoid such issues, CrossFS maintains a LibFS-level pathname resolution cache (with files that were opened by LibFS). The cache maintains a mapping between file names and the list of open file descriptors and FD-queue addresses. CrossFS treats these metadata-heavy operations as barrier operations and adds them to all FD-queues of an inode after acquiring an interval tree `rw-lock`. The requests in FD-queues preceding the barrier are processed first, followed by atomically processing a metadata request in one queue and removing others.

Metadata Operations without a File Descriptor. For metadata operations without a file descriptor (e.g., mkdir), CrossFS uses a global queue. Requests in the global queue are processed in parallel with FD-queue requests (barring some operations). Similar to kernel file systems, FirmFS concurrently processes non-dependent requests fetched from the global queue without compromising ordering. However, for complex operations such as a directory rename prone to atomicity and crash consistency bugs, CrossFS uses a system-wide ordering as used in traditional file systems [48]. Our current approach is to add a barrier request across all open FD-queues in the system. However, this could incur high latency when there are hundreds of FD-queues. Thus, we only add the rename request to the global queue and maintain a list of global barrier timestamps (`barrier_TSC`). Before dispatching a request, a device-CPU checks if request's `TSC` $<$ `barrier_TSC`; otherwise, delays processing the request until all prior requests before the barrier are processed and committed. CrossFS is currently designed to scale data plane operations and our future work will explore techniques to parallelize non-dependent metadata-heavy operations.

4.3.8 Block Cache and Memory-map Support.

For in-memory caching, CrossFS uses FD-queue buffers (in NVM) as data cache for accessing recent blocks but does not yet implement a shared main memory (DRAM) cache. We evaluate the benefits of using FD-queue as data buffer in §6. Implementing a shared page cache could be beneficial for slow storage devices [23, 37]. Similarly, our future work will focus on providing memory-map (`mmap`) support in CrossFS.

4.4 Cross-Layered Crash Consistency

A cross-layered file system requires a synergistic lightweight crash consistency support at the host (LibFS) and the device (FirmFS) layers. We describe the details next.

FD-queue Crash Consistency. The FD-queues buffer I/O requests and data. For I/O-intensive applications with tens of open file descriptors, providing persistence and crash consistency for LibFS-managed FD-queues could be important. Therefore, CrossFS utilizes system-level hardware capability and uses byte-addressable persistent memory (Intel Optane DC Persistent Memory [3]) for storing FD-queues. The FD-queues are allocated in NVM as a DMA'able memory. LibFS adds requests to persistent FD-queues using a well-known append-only logging protocol for the crash consistency [55, 61], and to prevent data loss from volatile processor caches, issues persistent writes using CLWB and memory fence. A commit flag is set after a request is added to the FD-queue. After a failure, during recovery, requests with an unset commit flag are discarded. Note that the interval tree is stored in the host or device RAM and is rebuilt using the FD-queues after a crash. In the absence of NVM, CrossFS uses DRAM for FD-queues, providing the guarantees of traditional kernel file systems that can lose uncommitted writes.

Low-overhead FirmFS Crash Consistency. The FirmFS, which is the heart of the CrossFS design, provides crash consistency for the file system data and metadata state in the device memory. FirmFS implements journaling using a

REDO journal maintained as a circular log buffer to hold data and metadata log entries [23, 31]. When FirmFS dispatches an `update` request, FirmFS initiates a new transaction, appends both data and metadata (e.g., inode) entries to the log buffer, and commits the transaction. However, data journaling is expensive; hence most prior NVM file systems only enable metadata journaling [18, 23, 31]. In contrast, CrossFS provides lightweight journaling by exploiting the persistent FD-queues. Intuitively, because the I/O requests and data are buffered in persistent FD-queues, the FirmFS journal can avoid appending data to the journal and provide a reference to the FD-queue buffer. Therefore, CrossFS dedicates a physical region in NVM for storing all FD-queues and buffers using our in-house persistent memory allocator. LibFS, when adding requests (e.g., read or write) to a persistent FD-queue, tags the request with the virtual address and the relative offset from the mapped FD-queue NVM buffer. FirmFS uses the offset to find the NVM physical address and stores it alongside the journal metadata. For recovery, the physical address can be used to recover the data. Consequently, CrossFS reaps the benefits of *data+metadata* journaling at the cost of metadata-only journaling. The journal entries are checkpointed when the journal is full or after a two-second (configurable) interval similar to prior file systems [23]. After a crash, during recovery, the FirmFS metadata journal is first recovered, followed by the data in the NVM FD-queues.

4.5 Multi-Queue File-Descriptor Scheduler

In traditional systems, applications are supported by an I/O scheduler in the OS and the storage firmware. In CrossFS, applications bypass the OS and lack the OS-level I/O scheduler support. As a consequence, when the number of FD-queues increase, non-blocking operations (e.g., `write`) could bottleneck blocking operations (e.g., `read, fsync`), further exacerbated by the limited device-CPU count. To address these challenges, CrossFS exploits its cross-layered design and merges two-level I/O schedulers into a single multi-queue firmware-level scheduler (FD-queue-scheduler). The FD-queue-scheduler's design is inspired by the state-of-the-art Linux *blk-queue* scheduler that separates software and hardware queues [14]. However, unlike the *blk-queue* scheduler, the FD-queue-scheduler (and FirmFS in general) is agnostic of process and thread abstractions in the host. Therefore, FD-queue-scheduler uses FD-queues as a basic scheduling entity and builds scheduling policies that decide how to map device-CPUs to serve requests from FD-queues.

4.5.1 Scheduling Policies.

The FD-queue-scheduler currently supports a simple round-robin policy and an urgency-aware scheduling policy.

Round-robin Scheduling. The round-robin scheduling aims to provide fairness. We maintain a global list of FD-queues, and the device-CPUs iterate through the global list to pick a queue currently not being serviced and schedule an I/O request from the FD-queue's head. To reduce scheduling unfairness towards files with higher FD-queue count, the round-robin scheduler performs two-level scheduling: first to pick an unserviced inode, and then across the FD-queues of the inode.

Urgency-aware Scheduling. While the round-robin scheduler increases fairness, it cannot differentiate non-blocking (e.g., POSIX `write, pwrite` return before persisting to disk) and latency-sensitive blocking (e.g., `read, fsync`) I/O operations. In particular, non-blocking operations such as `write` incur additional block writes to update metadata and data journals in contrast to read operations. Hence, in FirmFS, we implement an *urgency-aware* scheduling policy that prioritizes blocking operations without starving non-blocking operations. The device-CPUs when iterating the FD-queue list, pick and schedule blocking requests at the head. Optionally, LibFS could also tag requests as blocking and non-blocking. To avoid starving non-blocking operations, non-blocking I/O requests from FD-queues that are either full or delayed beyond a threshold (100μsec by default, but configurable) are dispatched. Our evaluation in §6 shows that urgency-aware policy improves performance for read-heavy workloads without significantly impacting write performance.

4.6 Security and Permission Checking

CrossFS matches the security guarantees provided by traditional and state-of-the-art user-level file systems [31, 40] by satisfying the following three properties. First, in CrossFS, the filesystem's metadata is always updated by the trusted FirmFS. Second, data corruptions are restricted to files for which threads and processes have write permission. Third, for files shared across processes, CrossFS allows only legal writers to update a file's data or the user-level interval tree.

File System Permissions. CrossFS aims to perform file permission checks without trapping into the OS for data plane operations. For permission management, CrossFS relies on trusted OS and FirmFS. The FirmFS maintains a credential table that maps a unique process ID to its credentials. During the initialization of per-process LibFS, the OS generates a random (128-bit) unique ID [1, 6] for each process and updates the firmware credential table with the unique ID and the process credentials [4] and returns the unique ID to LibFS. FirmFS also maintains a FD-queue to per-process unique ID mapping internally. When LibFS adds a request to its private FD-queue, it also adds the request's unique ID. Before processing a request, FirmFS checks if a request's unique ID matches FD-queue's unique ID (stored internally in FirmFS), and if they match, the I/O request's permission (e.g., write access) is checked using the credentials in the firmware table. Any mismatch between an I/O request and FD-queue IDs are identified and not processed by FirmFS. As a consequence, accidental (unintended) writes to FD-queue could be identified by FirmFS. Further, a malicious process that succeeds in forging another process's unique ID cannot use the ID in its

own FD-queue.

FD-queue Protection. First, FD-queues are privately mapped to a process address space and cannot be updated by other processes. Second, an untrusted LibFS with access permission to a file could reorder or corrupt FD-queue requests, but cannot compromise metadata. Finally, a malicious reader could add a file update request to FD-queue, but would not be processed by FirmFS.

Interval Tree. To reduce work at the device-CPUs, CrossFS uses LibFS (and host-CPUs) to manage inode-level interval tree and concurrency control. Because the interval tree is shared across writers and readers, an issue arises where a reader process (with read-only permission to a file) could accidentally or maliciously corrupt updates in the interval tree by legal writers (e.g., inserting a truncate("file", 0)).

To overcome such security complications, for simplicity, CrossFS provides two modes of interval tree updates: (a) FirmFS mode, and (b) LibFS delegation mode. In the FirmFS mode (enabled by default), the host-CPUs add I/O requests to FD-queues, but FirmFS and device-CPUs are responsible for updating the per-inode interval tree and managing concurrency control. This mode is specifically useful for cases where files are shared across multiple processes. As a consequence, file updates (e.g., truncate("file", 0)) by a malicious process with read-only access would be invalidated by the trusted FirmFS. This approach marginally impacts performance without compromising direct-I/O. In contrast, the LibFS delegation mode (an optional mode, which can be enabled when mounting CrossFS) allows the use of host-CPUs for interval tree updates but does not allow inter-process file sharing. We observe that most I/O-intensive applications (including RocksDB analyzed in the paper) without file sharing could benefit from using host-CPUs. We evaluate the trade-offs of both designs in §6.

Security Limitations. We discuss security limitations common to most User-FS designs [22, 33], and CrossFS.

Shared Data Structure Corruption. Sharing data and data-structures using untrusted user-level library makes CrossFS and other user-level file systems vulnerable to malicious/buggy updates or even DoS attacks. For example, in CrossFS, a user-level thread could corrupt the inode interval tree, impacting data correctness. Prior byte-addressable NVM designs such as ZoFS use hardware support such as Intel MPK ([22, 50]) to isolate memory regions across the file system library and application threads. Unfortunately, the isolation only protects against accidental corruption but not malicious MPK use [19].

Denial of Service (DoS) Attacks. CrossFS and most user-level designs do not handle DoS attacks, where a malicious LibFS could lock all intervals in the interval tree, perform many small updates to blocks, or force long interval tree merges for reads to those ranges. Recent studies have shown that lock-based attacks are possible for kernel file systems too [51]. One approach in CrossFS could be to use OS-level monitors to revoke threads that hold interval tree locks beyond a threshold (e.g., Linux hard/soft-lockup detector). However, a more thorough analysis is required, which will be the focus of our future work.

5 Implementation

Due to the lack of programmable storage hardware, we implement and emulate CrossFS as a device driver. CrossFS is implemented in about 13K lines of code spread across LibFS (2K LOC), FirmFS with scheduler and journaling (9K LOC), and the OS (300 LOC) components. For storage, we use Intel Optane DC Persistent Memory attached to the memory bus. To emulate device-CPUs, we use dedicated CPU cores, but also consider related hardware metrics such as device-level CPU speed, PCIe latency, and storage bandwidth (see §6). For direct-I/O, unlike prior systems that use high-overhead IOCTLs [33], the persistent FD-queues are mapped as shared memory between LibFS and the driver's address space.

LibFS. In addition to the details discussed in §4, LibFS uses a shim library to intercept POSIX operations and convert them to FirmFS compliant commands. We extend the NVMe command format that supports simple block (read and write) operations to support file system operations (e.g., open, read, write). The I/O commands are allocated using NVM, and LibFS issues the commands and sets a doorbell flag for FirmFS to begin processing. FirmFS processes a command and sets an acknowledgment bit to indicate request completion. Finally, for each file descriptor, LibFS maintains a user-level file pointer structure with a reference to the corresponding FD-queue, the file name, the interval tree, and information such as file descriptor offset. For persisting FD-queues, as described in §4.4, data updates are appended to NVM log buffers and persisted using CLWB and memory fence instructions.

FirmFS. The device-CPUs are emulated using dedicated (kernel) threads pinned to specific CPUs, whereas FirmFS block management (superblock, bitmaps, inodes, and data block management) and journaling structures implemented by extending PMFS [23] ported to Linux 4.8.2 kernel. FirmFS extends PMFS's inode and dentry with a simplified dentry structure and dentry cache for path resolution. In our file descriptor-based design, path resolution involves mapping a path to an inode containing a list of all FD-queues. For FirmFS data + metadata journaling, we use REDO logs and substitute actual data with the physical address of data in NVM FD-queues.

OS Component. We primarily extend the OS for FD-queue setup and credential management. For FD-queue setup, we implement an IOCTL in the OS to allocate contiguous DMA-able memory pages using NVM memory, mapping them to process and FirmFS device driver address spaces. The OS

component also cleans up FD-queues after a file is closed. Next, for credential management, we modify Linux process creation mechanism in the OS to generate a per-process unique ID. The OS also updates the firmware-level credential table with this information.

6 Evaluation

We compare CrossFS with state-of-the-art User-FS, Kernel-FS, and Firmware-FS using microbenchmarks, macrobenchmarks, and real-world applications to answer the following questions.

- How effective is CrossFS in exploiting the cross-layered design and file descriptor-based concurrency?
- How effective is CrossFS's FD-queue scheduler?
- What is the impact of host configuration such as FD-queue depth and device configurations such as storage bandwidth, device-CPU frequency, and PCIe latency?
- Can CrossFS's cross-layered design scale for metadata-intensive workloads with no data sharing across threads?
- Does CrossFS benefit real-world applications?

6.1 Experimental Setup

We use a dual-socket, 64-core, 2.7GHz Intel(R) Xeon(R) Gold platform with 32GB memory and a 512GB SSD. For storage and FD-queues, we use a 512GB (4x128GB) Optane DC persistent memory with 8GB/sec read and 3.8GB/sec rand-write bandwidth [30]. To emulate the proposed cross-layered file system, we reserve and use 2GB of DRAM for maintaining FirmFS in-memory state.

Besides, to study the implications of CrossFS for different storage bandwidths, PCIe latency, and device-CPU speeds, we use a CloudLab-based Intel(R) Xeon(R) CPU E5-2660 server that allows memory bandwidth throttling. We reserve 80GB memory mounted in a NUMA node for an emulated NVDIMM and vary bandwidth between 500MB/s to 10GB/s. To emulate PCIe latency, we add a 900ns software delay [49] between the time a request is added to the host's FD-queue and the time a request is marked ready for FirmFS processing. To emulate and vary device-CPU speeds, we apply dynamic voltage frequency scaling (DVFS). We enable persistence of NVM-based FD-queues and our proposed FirmFS data + metadata journaling that uses REDO logging.

For analysis, we compare CrossFS against state-of-the-art file systems in three different categories: User-FS Strata [40] and SplitFS [31], Kernel-FS ext4-DAX [64] and NOVA (a log structure NVM file system [66]), and finally, Firmware-FS DevFS [33]. To understand the benefits of avoiding system calls, for CrossFS, we compare an IOCTL-mode implementation with kernel traps but without VFS cost (CrossFS-ioctl) and a direct-mode without both kernel traps and VFS cost (CrossFS-direct).

6.2 Microbenchmarks

We first evaluate CrossFS using two microbenchmarks and then provide an in-depth analysis of how the host-side and the device-side configurations affect CrossFS performance.

6.2.1 Concurrency Analysis

In Figure 3a and Figure 3b, we vary the number of concurrent readers in the x-axis setting the concurrent writer count to four threads. For this multithreaded micro-benchmark, we use LibFS-level interval tree updates. We compare CrossFS-ioctl and CrossFS-direct against ext4-DAX, NOVA, DevFS, SplitFS, and Strata, all using Intel Optane DC persistent memory for storage.

ext4-DAX. First, ext4-DAX, a Kernel-FS, suffers from high system call and VFS cost, and locking overheads [31,63]. The inode `rw-lock` (read-write semaphore) contention between writers and readers increases with higher thread count. For the four concurrent reader-writer configuration, the time spent on locks is around 21.38%. Consequently, reader and writer throughputs are significantly impacted (see Figure 3b).

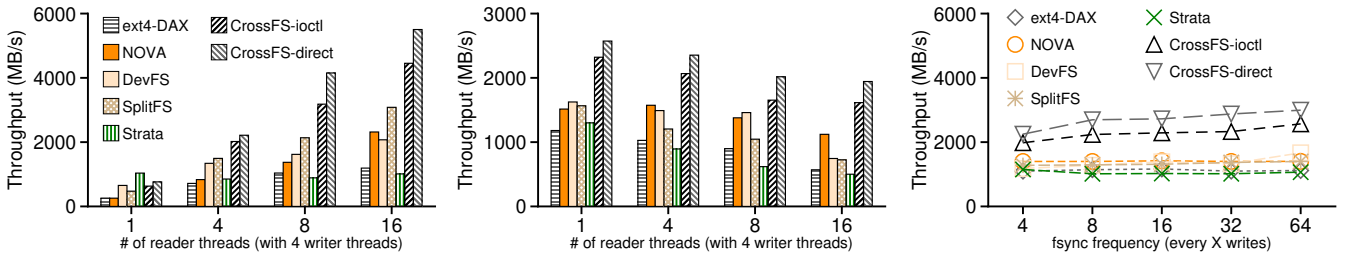
NOVA. Second, NOVA, also a Kernel-FS, exploits per-CPU file system data structures and log-structured design to improve aggregated write throughput compared to ext4-DAX. However, the use of inode-level `rw-lock` and system call costs impact write and read scalability.

DevFS. DevFS, a Firmware-FS, provides direct-access and avoids system calls, but uses per-inode I/O queues and inode-level `rw-lock`. Further, DevFS uses two levels of inode synchronization: first, when adding requests to an inode's I/O queue; second, when dispatching requests from the I/O queue for Firmware-FS processing. Consequently, the throughput does not scale with increasing thread count.

SplitFS. SplitFS, a User-FS, maps staging files to a process address space and converts data plane operations like `read()`, `write()` in the user-level library to memory loads and stores. Therefore, SplitFS avoids kernel traps for data plane operation (although metadata is updated in the kernel). As a result, SplitFS shows higher read and write throughputs over ext4-DAX, NOVA, and DevFS. However, for concurrent readers and writers, SplitFS gains are limited by inode-level locks.

Strata. Strata's low throughput is because of the following reasons: first, Strata uses an inode-level `mutex`; second, Strata uses a per-process private log (4GB by default as used in [40]). When using concurrent writers, the private log frequently fills up, *starving* readers to wait until the logs are digested to a shared area. The starvation amplifies for concurrent reader processes that cannot access private writer logs and must wait for the logs to be digested to a shared area.

CrossFS. In contrast, the proposed CrossFS-ioctl and CrossFS-direct's cross-layered designs with file descriptor concurrency allow concurrent read and write across FD-queues. Even though fewer than 1% of reads are fetched



(a) Aggregated Read Throughput **(b)** Aggregated Write Throughput **(c)** fsync Efficiency

Figure 3: Microbenchmark. Throughput of concurrent readers and 4 writers sharing a 12GB file. For CrossFS and DevFS, 4 device-CPUs are used. CrossFS-ioctl uses IOCTL commands bypassing VFS but with OS traps, whereas CrossFS-direct avoids OS traps and VFS. Figure 3a and Figure 3b are random accesses. Figure 3c shows the impact on performance when varying the commit frequency with 4 concurrent writers.

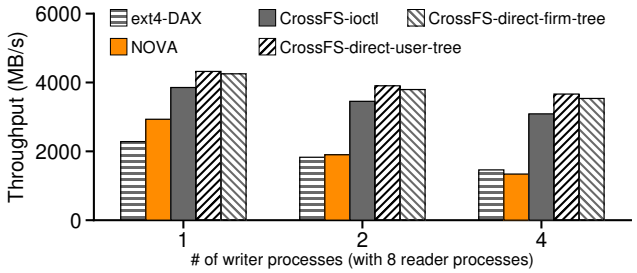


Figure 4: Process Sharing. Results show aggregated read throughput (MB/s) of 8 reader processes when sharing a file varying number of writer processes. *CrossFS-direct-user-tree* refers to using user-level interval tree, and *CrossFS-direct-firm-tree* refers to FirmFS managed interval tree, as discussed in §4.6.

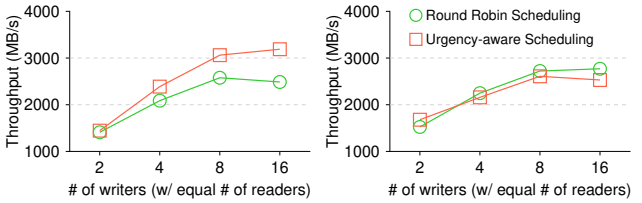
directly from the FD-queue, the performance gains are high. CrossFS-ioctl incurs kernel traps but avoids VFS overheads and low overhead journaling, resulting in 3.64X and 2.34X read performance gains over DAX and Strata, respectively. CrossFS-direct also avoids kernel traps achieving 3.38X and 4.87X write and read throughput gains over ext4-DAX, respectively. The read gains are higher because, for writes, the inode-level interval tree's `rw-lock` must be acquired (only) for FD-queue ordering. In our experiments, this accounts for only 9.9% of the execution time. Finally, as shown in Table 2, CrossFS not only improves throughput but *also reduces latency* for concurrent access.

6.2.2 Multi-Process Performance

Figure 4 shows CrossFS performance in the presence of multiple writers and readers processes. We use the same workload used earlier in Figure 3. For CrossFS-direct approach, we evaluate two cases: first, as discussed in §4.6, with CrossFS-direct-user-tree, we maintain a shared interval tree in the user-level, and LibFS updates the interval tree. While this approach reduces work for device-CPUs, a buggy or corrupted reader could accidentally or maliciously corrupt interval tree updates. In contrast, the CrossFS-direct-firm-tree approach avoids these issues by using FirmFS to update the interval tree, without impacting direct-I/O. As the figure shows, both approaches provide significant performance

	ext4-DAX	NOVA	DevFS	CrossFS-ioctl	CrossFS-direct
Readers	5.72	4.68	2.91	1.93	1.27
Writers	3.86	2.48	2.31	1.89	1.15

Table 2: Latency. Average per-thread random write and read latency (μs) with 4 concurrent readers and writers.



(a) Aggregated read throughput. **(b)** Aggregated write throughput.

Figure 5: Urgency-aware Scheduler Impact. Results show aggregated throughput for reader and writer threads. The x-axis varies the number of reader and writer threads.

gains compared to other state-of-the-art designs. Besides, CrossFS-direct-firm-tree shows only a marginal reduction in performance due to an increase in device-CPU work.

6.2.3 Commit Frequency

To study the performance of CrossFS's barrier-based commits, in Figure 3c, we evaluate `fsync` performance by running the random write benchmark with 4 concurrent writers. In the x-axis, we gradually increase the interval between successive `fsyncs`. As shown, compared to ext4-DAX, CrossFS delivers 2.05X and 2.68X performance gains for `fsyncs` issued at 4 write (worst-case) and 16 write (best-case) intervals, respectively. Although CrossFS adds a commit barrier to all FD-queues of an inode, device-CPUs can concurrently dispatch requests across FD-queues without synchronizing until the barrier completion. Additionally, CrossFS avoids system call cost for both `fsync` and write operations.

6.2.4 Urgency-aware Scheduler

CrossFS's cross-layered design smashes traditional OS-level I/O and firmware scheduler into a single firmware-level scheduler. To understand the implication of a scheduler, we evaluate two scheduling policies: (a) round-robin, which provides fairness, and (b) urgency-aware, which prioritizes

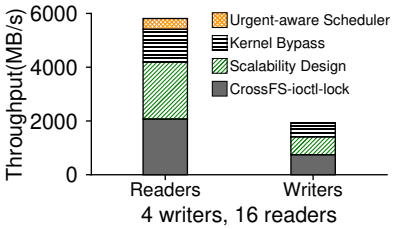


Figure 6: CrossFS Incremental Performance Breakdown. Results show aggregated throughput breakdown for 16 readers and 4 writers performing random access on a 12GB file. The baseline CrossFS approach (CrossFS-ioctl-lock) uses IOCTLs and coarse-grained inode-level lock.

blocking operations (e.g., read). We use the random-access micro-benchmark (discussed earlier) with an equal number of readers and writers performing random access on a shared 12GB file. Figure 5a and 5b show aggregated throughput for concurrent writers and readers, while varying the reader and writer count on the x-axis. First, the round-robin policy does not differentiate blocking reads and non-blocking writes; hence, it dispatches read and write requests with equal priority. Consequently, with 4 device-CPUs and the use of interval tree `rw-lock`, blocking reads are delayed. The write throughput does not scale beyond 8 threads. In contrast, CrossFS's urgency-aware scheduling prioritizes blocking reads without starving writes beyond a threshold, thereby accelerating reads by 1.22x.

6.2.5 CrossFS Performance Breakdown.

To decipher the source of CrossFS-direct performance gains, in Figure 6, we show the throughput breakdown of 4 writers, 16 readers configuration. CrossFS-ioctl-lock represents the CrossFS baseline that suffers from IOCTL (system call) cost and uses a coarse-grained inode `rw-lock`. Replacing the inode-level lock with fine-grained FD-queue concurrency (shown as Scalability Design) and eliminating system call cost (Kernel Bypass) provides significant performance benefits over the baseline. The urgency-aware scheduler improves the read throughput further.

6.2.6 Sensitivity Analysis - Host-side Configuration

We next study the performance impact of host-side configurations, which includes: (a) FD-queue depth (i.e., queue length), (b) read hits that directly fetch data from FD-queue, and (c) write conflicts with in-transit FD-queues requests. The values over the bars in Figure 7a and 7b show read hits and write conflicts (in percentage), respectively.

Read Hits. To understand the performance impact of FD-queue read hits, we mimic sequential producer-consumer I/O access patterns exhibited in multithreaded I/O-intensive applications such as Map-Reduce [20] and HPC "Cosmic Microwave Background (CMB)" [16]. The concurrent writers (producers) sequentially update specific ranges of blocks in a file, whereas the consumers sequentially read from specific ranges. Note that both producers and consumers use separate file descriptors. The consumers start at the same time as producers. In Figure 7a, for CrossFS, we vary FD-queue depths to 32 (CrossFS-direct-QD32, the default depth) and 256 (CrossFS-direct-QD256) entries. For simplicity, we only show the results for CrossFS-direct, which outperforms CrossFS-ioctl. As expected, increasing the queue depth from 32 to 256 increases the read hit rate, enabling host threads to fetch updates from FD-queues directly. Consequently, read throughput for CrossFS-direct-QD256 improves by 1.12x over CrossFS-direct-QD32, outperforming other approaches.

Write Conflicts. A consequence of increasing the queue depth is the increase in write conflicts across concurrent writers. To illustrate this, we only use concurrent writers in the above benchmark, and the writers sequentially update all blocks in a file. As shown in Figure 7b, an increase in queue-depth increases write conflicts (27% for CrossFS-direct-QD256 with 8 threads), forcing some requests to be ordered to the same FD-queue. However, this does not adversely impact the performance because of our optimized conflict resolution and fewer host-CPU stalls with 256 FD-queue entries.

6.2.7 Sensitivity Analysis - Device Configuration

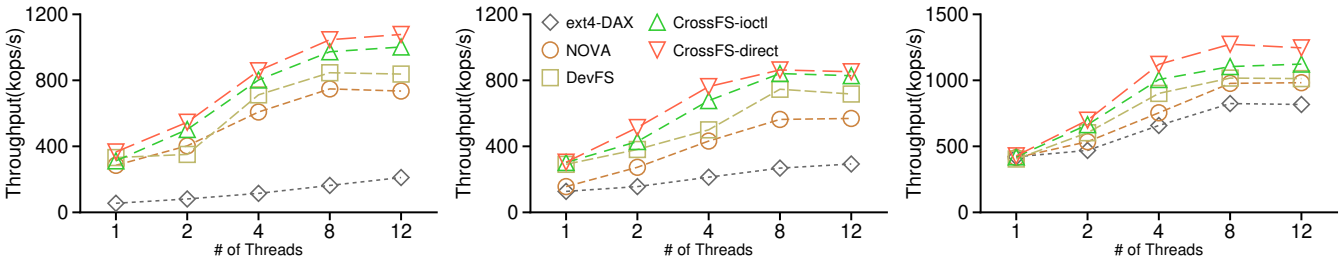
A cross-layered file system could be deployed in storage devices with different bandwidths, incur PCIe latency for host and device interaction, and use wimpier CPUs. To understand CrossFS's performance sensitivity towards these device configurations, we decipher the performance by varying the storage bandwidth, adding PCIe latency, and reducing the frequency of device-CPUs.

For varying the storage bandwidth, we use DRAM as a storage device and vary the storage bandwidth between 0.5GB/s to 10GB/s using thermal throttling. We use DRAM because Optane NVM cannot be thermal-throttled, and its bandwidth cannot be changed. In Figure 7c, we compare three CrossFS approaches: (1) CrossFS-noPCIeLat-HighFreq - the default approach without PCIe latency and high device-CPU frequency (2.7GHz); (2) CrossFS-PCIeLat-HighFreq - an approach that emulates *Gen 3 x8 PCIe's* latency of 900ns [49] by adding software delays between the time a request is added to a FD-queue and the time when the request is processed by FirmFS; and finally, (3) CrossFS-PCIeLat-LowFreq - an approach that reduces device-CPU frequency to 1.2GHz (the minimum frequency possible) using DVFS [41] in addition to added PCIe latency. The results show random write throughput when four concurrent writers and readers share a 12GB file. We also compare ext4-DAX, NOVA, and DevFS *without reducing CPU frequency or PCIe latency*.

At lower storage bandwidths (e.g., 500MB/s), as expected, CrossFS's gains are limited by storage bandwidth. However, even at 2GB/s bandwidth, CrossFS-noPCIeLat-HighFreq shows 1.96x gains over ext4-DAX. Next, the impact of 900ns PCIe latency (CrossFS-PCIeLat-HighFreq) is overpowered by other software overheads such as file system metadata and

(a) Read Hit Rate Impact **(b)** Write Conflicts Impact **(c)** Sensitivity to Device Configurations.

Figure 7: Performance Sensitivity towards Host and Device Configuration. Figure 7a and Figure 7b host configuration (FD-queue depth) sensitivity. Figure 7a shows read throughput with increasing in read hit rate for 32 and 256 entry FD-queue depth. Figure 7b shows write throughput and write conflict (%) when varying FD-queue depth across concurrent writers sequentially updating a file. Figure 7c shows device configuration sensitivity. The x-axis varies the storage bandwidth, and the graph shows write throughput for 4 concurrent writers when adding PCIe latency and reducing device-CPU frequency.



(a) Varmail **(b)** Fileserver **(c)** Webserver

Figure 8: Filebench Throughput. The evaluation uses 4 device-CPUs.

data management, journaling, and scheduling. While higher CPU frequency, storage bandwidth, and low PCIe latency would maximize gains when using a programmable storage, even when using 2.5x slower CPUs and 900ns PCIe latency, our cross-layered and concurrency-centric design provides 1.87x, 1.97x, 1.35x over ext4-DAX, NOVA, and DevFS that use high-frequency CPUs without PCIe latency, respectively.

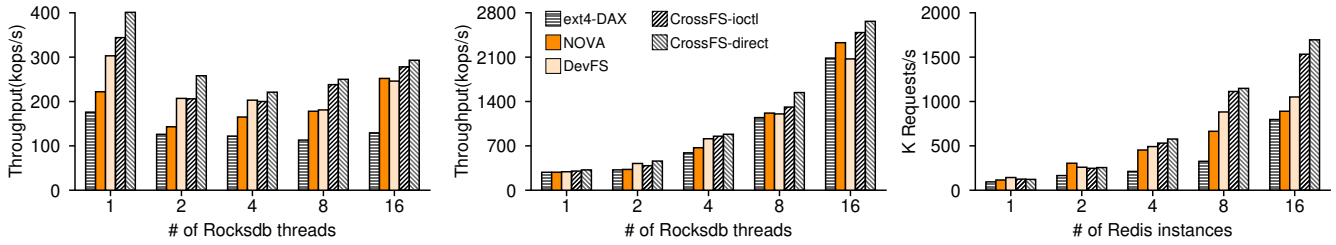
6.3 Macrobenchmark: Filebench

Does CrossFS's cross-layered design benefit multithreaded workloads without file sharing? To understand the performance, we evaluate well-known Filebench's *Varmail* (metadata-heavy), *Fileserver* (write-heavy), and *Webserver* (read-heavy) workloads that represent real-world workloads [22, 23, 66]. The workloads are metadata-intensive and perform operations such as file create, delete, directory update, which contributes to 69%, 63%, and 64% in Varmail, Fileserver, and Webserver, respectively, of the overall I/O. The data read to write ratios are 1:1, 1:2, and 10:1 in Varmail, Fileserver, and Webserver, respectively.

We compare CrossFS-ioctl and CrossFS-direct against ext4-DAX, NOVA, and DevFS. SplitFS does not yet support Filebench and RocksDB. Figure 8 shows the throughput for three workloads. The x-axis shows the throughput when increasing Filebench's thread count. Without file sharing across threads, DevFS (without system calls) performs better than NOVA and ext4-DAX. In contrast, CrossFS-direct, for *Varmail* and *Fileserver*, outperforms other approaches and provides 1.47x and 1.77x gains over NOVA, respectively. *Varmail* is metadata-intensive and (with 1:1 read-write ratio for data operations), and *fileserver* is highly write-intensive. For both these workloads, CrossFS-direct avoids system calls, reduces VFS cost, and provides fast metadata journaling at the cost of data journaling (aided by NVM-based FD-queues). With just 4 device-CPUs and a metadata I/O-queue for operations without file descriptors, CrossFS-direct gains flatten. Finally, *Webserver* has a significantly higher read ratio. While CrossFS-direct improves performance, the throughput gains (1.71x) are restricted. First, blocking reads stress the available four device-CPUs. Second, we notice OS scheduler overheads as a side-effect of emulating device-CPUs with Linux kernel threads. The kernel threads in Linux periodically check and yield to the OS scheduler if necessary (i.e., `need_resched()`); the periodic yields negatively impact blocking random read operations for CrossFS and DevFS.

6.4 Real-World Applications

RocksDB. RocksDB [9] is a widely used LSM-based NoSQL database used as a backend in several production systems and frameworks [11, 12, 44]. RocksDB is designed to exploit SSD's parallel bandwidth and multicore parallelism. As discussed earlier, we observe around 40% of I/O accesses to shared files across RocksDB's foreground threads that share



(a) Rocksdb random writes **(b)** Rocksdb random reads **(c)** Redis SET benchmark

Figure 9: Application throughput. Figure 9a and Figure 9b show random write and read performance for RocksDB by varying threads in *DBbench* workload. RocksDB internally creates background compaction threads. Figure 9c shows the *SET* benchmark for Redis by varying the number of Redis instances.

log files and background threads that compact in-memory data across LSM levels in the SST (string sorted) files [34]. However, conflicting block updates are negligible. In Figure 9a and Figure 9b, we vary the number of application (foreground) threads along the x-axis and set the device-CPU count to four. We compare ext4-DAX, NOVA, DevFS, CrossFS-ioctl, and CrossFS-direct's throughput for random write and read operations. Note that RocksDB, by default, uses three background parallel compaction threads, which increases with increasing SST levels [10]. We use widely used *DBbench* [2], with 100B keys, 1KB values, and a 32GB database size. We set RocksDB's memory buffer to 128MB [44].

CrossFS-direct and CrossFS-ioctl show up to 2.32x and 1.15x write and read throughput gains over ext4-DAX, respectively. The read and write performance benefits over DevFS are 1.33x and 1.21x, respectively. We attribute these gains to the following reasons. First, RocksDB threads do not update the same block (lower than 0.1% conflicts); hence, unlike other approaches, CrossFS-ioctl and CrossFS-direct avoid inode-level locks. Second, both CrossFS-ioctl and CrossFS-direct avoid VFS overheads. Additionally, CrossFS-direct also avoids system call costs. When increasing application threads, the burden on four device-CPUs increases, impacting performance for the 16 application thread configuration. The blocking reads are impacted due to the use of kernel threads for device-CPU emulation (see §6.3).

Redis. Redis is a widely used storage-backed in-memory key-value store [8], which logs operations to append-only-files (AOF) and checkpoints in-memory key-values asynchronously to backup files called RDB [40]. We run multiple Redis instances and the instances do not share AOF or RDB files. We use background write mode for Redis instances that immediately persist key-value updates to the disk. Figure 9c shows the Redis performance.

First, when increasing Redis instances, the number of concurrent writers increases. The server and the client (benchmark) instances run as separate processes, which increases inter-process communication, system call, and VFS costs. Although instances do not share files, CrossFS-direct provides considerable performance gains mostly stemming from direct storage access, avoiding VFS overheads, and lowering journaling cost. Consequently, CrossFS provides 2.35x higher throughput over ext4-DAX.

7 Conclusion

This paper proposes CrossFS, a cross-layered file system design that provides high-performance direct-I/O and concurrent access across threads and applications with or without data sharing. Four key ingredients contribute to CrossFS's high-performant design. First, our cross-layered approach exploits hardware and software resources spread across the untrusted user-level library, the trusted OS, and the trusted device firmware. Second, the fine-grained file descriptor concurrency design converts a file synchronization problem to the I/O queue ordering problem, which ultimately scales concurrent access. Third, our lightweight data + metadata journaling aided by NVM reduces crash consistency overheads. Finally, our unified firmware-level scheduler complements the file descriptor design, reducing I/O latency for blocking operations. Our detailed evaluation of CrossFS against state-of-the-art kernel-level, user-level, and firmware-level file system shows up to 4.87x, 3.58x, 2.32x gains on microbenchmarks, macrobenchmarks, and applications.

Acknowledgements

We thank the anonymous reviewers and Emmett Witchel (our shepherd) for their insightful comments and feedback. We thank the members of the Rutgers Systems Lab for their valuable input. This material was supported by funding from NSF grant CNS-1910593. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and may not reflect the views of NSF.

References

[1] A Universally Unique IDentifier (UUID) URN Namespace. https://www.ietf.org/rfc/rfc4122.txt.

[2] Google LevelDB . http://tinyurl.com/osqd7c8.

[3] Intel-Micron Memory 3D XPoint. http://intel.ly/1eICR0a.

[4] Linux Credentials. https://www.kernel.org/doc/html/latest/security/credentials.html.

[5] MySQL. https://www.mysql.com/.

[6] OpenSSL. https://www.openssl.org/docs/man1.1.1/man1/openssl-genrsa.html.

[7] rbtree based interval tree as a prio_tree replacement. https://lwn.net/Articles/509994/.

[8] Redis. http://redis.io/.

[9] RocksDB. http://rocksdb.org/.

[10] RocksDB Tuning Guide. https://github.com/facebook/rocksdb/wiki/RocksDB-Tuning-Guide/.

[11] Tensor Flow. https://www.tensorflow.org/.

[12] Abutalib Aghayev, Sage Weil, Michael Kuchnik, Mark Nelson, Gregory R. Ganger, and George 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, pages 353–369, New York, NY, USA, 2019. Association for Computing Machinery.

[13] Srivatsa S. Bhat, Rasha Eqbal, Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. Scaling a File System to Many Cores Using an Operation Log. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 69–86, New York, NY, USA, 2017. ACM.

[14] Matias Bjørling, Jens Axboe, David Nellans, and Philippe Bonnet. Linux Block IO: Introducing Multi-queue SSD Access on Multi-core Systems. In *Proceedings of the 6th International Systems and Storage Conference*, SYSTOR '13, pages 22:1–22:10, New York, NY, USA, 2013. ACM.

[15] Matias Bjørling, Javier González, and Philippe Bonnet. LightNVM: The Linux Open-channel SSD Subsystem. In *Proceedings of the 15th Usenix Conference on File and Storage Technologies*, FAST'17, Santa clara, CA, USA, 2017.

[16] J. Borrill, J. Carter, L. Oliker, and D. Skinner. Itegrated performance monitoring of a cosmology application on leading HEC platform. In *Proceedings of 2005 International Conference on Parallel Processing (ICPP'05)*, pages 119–128, 2005.

[17] Adrian M. Caulfield, Todor I. Mollov, Louis Alex Eisner, Arup De, Joel Coburn, and Steven Swanson. Providing Safe, User Space Access to Fast, Solid State Disks. *SIGARCH Comput. Archit. News*, 40(1), March 2012.

[18] Vijay Chidambaram, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi 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.

[19] R. Joseph Connor, Tyler McDaniel, Jared M. Smith, and Max Schuchard. PKU Pitfalls: Attacks on PKU-based Memory Isolation Systems. In Srdjan Capkun and Franziska Roesner, editors, *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, pages 1409–1426. USENIX Association, 2020.

[20] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, pages 137–150, San Francisco, CA, 2004.

[21] Yang Deng, Arun Ravindran, and Tao Han. Edge Datastore for Distributed Vision Analytics: Poster. In *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*, SEC '17, pages 29:1–29:2, New York, NY, USA, 2017. ACM.

[22] Mingkai Dong, Heng Bu, Jifei Yi, Benchao Dong, and Haibo Chen. Performance and Protection in the ZoFS User-Space NVM File System. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, pages 478–493, New York, NY, USA, 2019. Association for Computing Machinery.

[23] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. System Software for Persistent Memory. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 15:1–15:15, New York, NY, USA, 2014. ACM.

[24] Jake Edge. VFS parallel lookups. https://lwn.net/Articles/685108/.

[25] Brendan Gregg. KPTI/KAISER Meltdown Initial Performance Regressions. http://www.brendangregg.com/blog/2018-02-09/kpti-kaiser-meltdown-performance.html.

[26] Tyler Harter, Chris Dragga, Michael Vaughn, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. A File is Not a File: Understanding the I/O Behavior of Apple Desktop Applications. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 71–83, New York, NY, USA, 2011. ACM.

[27] Jun He, Sudarsun Kannan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. The Unwritten Contract of Solid State Drives. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 127–144, New York, NY, USA, 2017. ACM.

[28] Mohammad Hedayati, Kai Shen, Michael L. Scott, and Mike Marty. Multi-Queue Fair Queuing. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 301–314, Renton, WA, July 2019. USENIX Association.

[29] Intel. Storage Performance Development Kit. http://www.spdk.io/.

[30] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module, 2019.

[31] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. SplitFS: Reducing Software Overhead in File Systems for Persistent Memory. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, pages 494âĂŞ–508, New York, NY, USA, 2019. Association for Computing Machinery.

[32] Junbin Kang, Benlong Zhang, Tianyu Wo, Weiren Yu, Lian Du, Shuai Ma, and Jinpeng 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.

[33] Sudarsun Kannan, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Yuangang Wang, Jun Xu, and Gopinath Palani. Designing a True Direct-access File System with DevFS. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies*, FAST'18, pages 241–255, Berkeley, CA, USA, 2018. USENIX Association.

[34] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Redesigning LSMs for Nonvolatile Memory with NoveLSM. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 993–1005, Boston, MA, July 2018. USENIX Association.

[35] Sanidhya Kashyap, Changwoo Min, Kangnyeon Kim, and Taesoo Kim. A Scalable Ordering Primitive for Multicore Machines. In *Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal, April 23-26, 2018*, pages 34:1–34:15. ACM, 2018.

[36] Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. Scalable NUMA-aware Blocking Synchronization Primitives. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '17, pages 603–615, Berkeley, CA, USA, 2017. USENIX Association.

[37] Hyeong-Jun Kim, Young-Sik Lee, and Jin-Soo Kim. NVMeDirect: A User-space I/O Framework for Application-specific Optimization on NVMe SSDs. In *8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 16)*, Denver, CO, 2016. USENIX Association.

[38] Jaeho Kim, Ajit Mathew, Sanidhya Kashyap, Madhava Krishnan Ramanathan, and Changwoo Min. MV-RLU: Scaling Read-Log-Update with Multi-Versioning. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*, pages 779–792. ACM, 2019.

[39] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[40] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. Strata: A Cross Media File System. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, 2017.

[41] Etienne Le Sueur and Gernot Heiser. Dynamic Voltage and Frequency Scaling: The Laws of Diminishing Returns. In *Proceedings of the 2010 International Conference on Power Aware Computing and Systems*, HotPower'10, page 1âĂŞ8, USA, 2010. USENIX Association.

[42] Chang-Gyu Lee, Hyunki Byun, Sunghyun Noh, Hyeongu Kang, and Youngjae Kim. Write Optimization of Log-structured Flash File System for Parallel I/O on Manycore Servers. In *Proceedings of the 12th ACM International Conference on Systems and Storage*, SYSTOR '19, pages 21–32, New York, NY, USA, 2019. ACM.

[43] Changman Lee, Dongho Sim, Joo-Young Hwang, and Sangyeun Cho. F2FS: A New File System for Flash Storage. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, FAST'15, Santa Clara, CA, 2015.

[44] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy 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, pages 447–461, New York, NY, USA, 2019. Association for Computing Machinery.

[45] Jianwei Li, Wei-keng Liao, Alok Choudhary, Robert Ross, Rajeev Thakur, William Gropp, Rob Latham, Andrew Siegel, Brad Gallagher, and Michael Zingale. Parallel netCDF: A High-Performance Scientific I/O Interface. In *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*, SC '03, pages 39–, New York, NY, USA, 2003. ACM.

[46] Hyeontaek Lim, Michael Kaminsky, and David G. Andersen. Cicada: Dependably Fast Multi-Core In-Memory Transactions. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pages 21–35. ACM, 2017.

[47] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading Kernel Memory from User Space. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 973–990, Baltimore, MD, August 2018. USENIX Association.

[48] Changwoo Min, Sanidhya Kashyap, Steffen Maass, and Taesoo Kim. Understanding Manycore Scalability of File Systems. In Ajay Gulati and Hakim Weatherspoon, editors, *2016 USENIX Annual Technical Conference, USENIX ATC 2016, Denver, CO, USA, June 22-24, 2016*, pages 71–85. USENIX Association, 2016.

[49] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W. Moore. Understanding PCIe Performance for End Host Networking. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM'18, pages 327–341, New York, NY, USA, 2018. Association for Computing Machinery.

[50] Soyeon Park, Sangho Lee, Wen Xu, HyunGon Moon, and Taesoo Kim. libmpk: Software Abstraction for Intel Memory Protection Keys (Intel MPK). In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 241–254, Renton, WA, July 2019. USENIX Association.

[51] Yuvraj Patel, Leon Yang, Leo Arulraj, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Michael M. Swift. Avoiding Scheduler Subversion Using Scheduler-Cooperative Locks. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[52] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The Operating System is the Control Plane. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, Broomfield, CO, 2014.

[53] Thanumalayan Sankaranarayana Pillai, Ramnatthan Alagappan, Lanyue Lu, Vijay Chidambaram, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Application Crash Consistency and Performance with CCFS. In *Proceedings of the 15th Usenix Conference on File and Storage Technologies*, FAST'17, Santa clara, CA, USA, 2017.

[54] Vijayan Prabhakaran, Thomas L. Rodeheffer, and Lidong Zhou. Transactional Flash. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, San Diego, California, 2008.

[55] Madhava Krishnan Ramanathan, Jaeho Kim, Ajit Mathew, Xinwei Fu, Anthony Demeri, Changwoo Min, and Sudarsun Kannan. Durable Transactional Memory Can Scale with Timestone. In *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020 [ASPLOS 2020 was canceled because of COVID-19]*, pages 335–349. ACM, 2020.

[56] Zhenyuan Ruan, Tong He, and Jason Cong. INSIDER: Designing In-Storage Computing System for Emerging High-Performance Drive. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 379–394, Renton, WA, July 2019. USENIX Association.

[57] Samsung. NVMe SSD 960 Polaris Controller. http://www.samsung.com/semiconductor/minisite/ssd/downloads/document/NVMe_SSD_960_PRO_EVO_Brochure.pdf.

[58] Y. Son, J. Choi, J. Jeon, C. Min, S. Kim, H. Y. Yeom, and H. Han. SSD-Assisted Backup and Recovery for Database Systems. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, pages 285–296, April 2017.

[59] Tarasov Vasily. Filebench. https://github.com/filebench/filebench.

[60] Haris Volos, Sanketh Nalli, Sankarlingam Panneerselvam, Venkatanathan Varadarajan, Prashant Saxena, and Michael M. Swift. Aerie: Flexible File-system Interfaces to Storage-class Memory. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, Amsterdam, The Netherlands, 2014.

[61] Haris Volos, Andres Jaan Tack, and Michael M. Swift. Mnemosyne: Lightweight Persistent Memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, Newport Beach, California, USA, 2011.

[62] Michael Wei, Matias Bjørling, Philippe Bonnet, and Steven Swanson. I/O Speculation for the Microsecond Era. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, Philadelphia, PA, 2014.

[63] Zev Weiss, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. DenseFS: a Cache-Compact Filesystem. In *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*, Boston, MA, July 2018. USENIX Association.

[64] Matthew Wilcox and Ross Zwisler. Linux DAX. https://www.kernel.org/doc/Documentation/filesystems/dax.txt.

[65] Jian Xu, Juno Kim, Amirsaman Memaripour, and Steven Swanson. Finding and Fixing Performance Pathologies in Persistent Memory Software Stacks. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, pages 427–439, New York, NY, USA, 2019. Association for Computing Machinery.

[66] Jian Xu and Steven Swanson. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. In *Proceedings of the 14th Usenix Conference on File and Storage Technologies*, FAST'16, Santa Clara, CA, 2016.

[67] Qiumin Xu, Huzefa Siyamwala, Mrinmoy Ghosh, Tameesh Suri, Manu Awasthi, Zvika Guz, Anahita Shayesteh, and Vijay Balakrishnan. Performance Analysis of NVMe SSDs and Their Implication on Real World Databases. In *Proceedings of the 8th ACM International Systems and Storage Conference*, SYSTOR '15, Haifa, Israel, 2015.

[68] Jisoo Yang, Dave B. Minturn, and Frank Hady. When Poll is Better Than Interrupt. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, FAST'12, San Jose, CA, 2012.

[69] Yongen Yu, Douglas H. Rudd, Zhiling Lan, Nickolay Y. Gnedin, Andrey V. Kravtsov, and Jingjin Wu. Improving Parallel IO Performance of Cell-based AMR Cosmology Applications. *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, pages 933–944, 2012.

[70] Teng Zhang, Jianying Wang, Xuntao Cheng, Hao Xu, Nanlong Yu, Gui Huang, Tieying Zhang, Dengcheng He, Feifei Li, Wei Cao, Zhongdong Huang, and Jianling Sun. FPGA-Accelerated Compactions for LSM-based Key-Value Store. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 225–237, Santa Clara, CA, February 2020. USENIX Association.