FusionFS: Fusing I/O Operations using CISC$_{Ops}$ in Firmware File Systems

Jian Zhang, Yujie Ren, and Sudarsun Kannan, *Rutgers University*

https://www.usenix.org/conference/fast22/presentation/zhang-jian

This paper is included in the Proceedings of the
20th USENIX Conference on File and Storage Technologies.

February 22–24, 2022 • Santa Clara, CA, USA

978-1-939133-26-7

Open access to the Proceedings
of the 20th USENIX Conference on
File and Storage Technologies
is sponsored by USENIX.

FusionFS: Fusing I/O Operations using $CISC_{Ops}$ in Firmware File Systems

Jian Zhang[*] *Yujie Ren*[*] *Sudarsun Kannan*
Rutgers University

Abstract

We present FusionFS, a direct-access firmware-level in-storage filesystem that exploits near-storage computational capability for fast I/O and data processing, consequently reducing I/O bottlenecks. In FusionFS, we introduce a new abstraction, $CISC_{Ops}$, which combines multiple I/O and data processing operations into one fused operation and offloads them for near-storage processing. By offloading, $CISC_{Ops}$ significantly reduces dominant I/O overheads such as system calls, data movement, communication, and other software overheads. Further, to enhance the use of $CISC_{Ops}$, we introduce MicroTx, a fine-grained crash consistency and fast (automatic) recovery mechanism for both I/O and data processing operations. Finally, we explore efficient and fair use of in-storage compute resources by proposing a novel Completely Fair Scheduler (CFS) for in-storage compute and memory resources across tenants. Evaluation of FusionFS against the state-of-the-art user-level, kernel-level, and firmware-level file systems using microbenchmarks, macrobenchmarks, and real-world applications shows up to 6.12x, 5.09x, and 2.07x performance gains, and 2.65x faster recovery.

1 Introduction

Modern high bandwidth and low-latency storage technologies such as NVMe SSDs [50] and 3D-Xpoint [6] have significantly accelerated I/O performance leading to better application performance. Yet, the combination of software and hardware I/O overheads that include system calls, data movement, and communication cost in the application and the OS, and the storage hardware latency (e.g., PCIe) continue to be an Achilles heel in fully exploiting storage hardware capabilities.

A recent focus is to reduce software indirections by moving filesystems to userspace and avoiding system calls and kernel traps for data and metadata updates [22, 60, 49, 27, 35]. Although effective, the dominating I/O overheads such as data and metadata movement cost, host and device communication cost (e.g., PCIe latency), and indirect costs like polling or interrupts remain. Henceforth, we refer to the combination of above overheads, which includes system calls, as dominating I/O overheads.

Another design point to reduce I/O overheads is the reincarnation of near-storage processing [46]. Vendors are introducing computational storage devices (**CSD**) that embed in-storage processors that range from ARM cores [11], FPGAs [49, 52, 30, 44, 47, 14] to RISC-V processors [51]. To reduce I/O cost and offload computations to CSD, recent research has explored application customization techniques [14], software runtimes [47], system software [49], and databases [46].

More broadly, these techniques can be classified into systems that focus on (1) in-storage compute offloading and (2) in-storage filesystems and key-value stores designed to accelerate I/O and storage management. First, in-storage compute offloading systems (which includes a majority of current CSD solutions) such as the seminal ActiveStore [46] for databases and recent approaches like PolarDB [14] focus on data processing by rewriting application logic to offload computation. While beneficial, these systems either lack storage management or delegate management to the host file system [14]. The former leads to a lack of data and metadata integrity, crash consistency, durability, or managing in-storage resources across tenants. In contrast, the latter incurs high I/O overheads for basic I/O operations and fails to utilize the full potential of CSDs. For example, in key-value stores like LevelDB [5], one could offload data compression to a CSD, but basic I/O operations would still incur system calls, data, metadata (e.g., inode, extents), and journal movement between key-value store, file system, and storage.

In contrast, in-storage management designs like CrossFS [44], DevFS [30], and Insider [47] offload filesystems and key-value stores [49] inside the storage firmware for direct-I/O, bypassing the OS. Unfortunately, these designs lack near-storage processing capability leading to substantial data movement and failing to manage in-storage resources such as device compute and memory or handle multi-tenancy.

We envision an ideal near-storage design that co-designs

[*]The authors contribute equally to this paper.

and combines storage management and data processing by rethinking I/O abstractions to reduce dominant I/O overheads, such as system calls, data and metadata movement, and host to device communication latency. Importantly, the design must ensure (storage) correctness, handle crash consistency, and achieve fairness across tenants.

We propose **FusionFS**, an near-storage file system design to exploit device compute and memory resources for reducing dominant I/O overheads and improving application performance. FusionFS provides fine-grained crash consistency, fast data recovery, and improves system efficiency by providing in-storage compute and memory fairness across tenants.

Towards the above goals, in FusionFS, we revisit I/O abstractions and take inspiration from seminal RISC (reduced instruction set computers) and CISC (complex instruction set computers) architectures. In FusionFS, a RISC operation is a simple POSIX file system operation (e.g., read, write, open) that can be directly offloaded to an in-storage filesystem (StorageFS), bypassing the OS. In contrast, our proposed CISC operations (hereafter referred to as **CISC$_{Ops}$**) are aggregated I/O and data processing operations offloaded as one operation to StorageFS for processing.

For generating CISC$_{Ops}$, we capture frequent I/O (e.g., *file-open-write-close*) and I/O + data processing sequences on a file (e.g., *append-checksum-write*) and combine them to one CISC operation. Intuitively, *aggregating I/O and data processing sequences and offloading them for near-storage processing* significantly reduces dominant overheads (system calls, data movement, and device and host communication costs). Note that CISC$_{Ops}$ support a combination of I/O and data processing operations and differ from traditional POSIX I/O vectors that are homogeneous (e.g., *readv*, *writev*).

Supporting in-storage RISC and CISC$_{Ops}$ introduces new challenges in terms of (1) applications changes, (2) crash consistency, and (3) resource management.

Application Support. FusionFS strives to reduce application changes by requiring minimal changes. First, a user-level library file system (UserLib) enables applications to use POSIX-like extensions for data processing or pack their custom command vectors supported by an in-storage file system (StorageFS). Optionally, FusionFS also provides mechanisms to transparently combine multiple I/O operations (without data processing) into a CISC$_{Ops}$ and offload them for in-storage processing, when feasible.

Fine-grained Crash-Consistency and Fast Recovery. In FusionFS, for traditional filesystem operations, we support journaling inside StorageFS. However, questions arise when packing multiple I/O and data processing operations in a CISC$_{Ops}$: (1) in what granularity should FusionFS support crash consistency? (2) how to exploit in-storage compute to accelerate recovery? For answering these questions, in FusionFS, we explore macro-transactions (MacroTx) and micro-transactions (MicroTx). MacroTx uses an all-or-nothing approach that only commits and recovers an entire CISC$_{Op}$ including the data processing state, whereas MicroTx supports crash consistency of partially committed CISC$_{Ops}$. Further, to reap the benefits of MicroTx, we go a step beyond current filesystems and use in-storage compute to support operational logging and automatic recovery by finishing partially completed CISC$_{Ops}$.

In-storage Resource Fairness. Next, offloading simple I/O operations and CISC$_{Ops}$ across tenants could exceed the in-storage compute (device-CPUs) and memory (device-RAM) resources. Therefore, there is a need for efficient and fair allocation of resources in ways that do not starve operations or tenants. Hence, in FusionFS, we borrow ideas from the Linux CPU scheduler, Completely Fair Scheduler (CFS) [1], to design a device-CPU and device-RAM CFS scheduler for enabling resource fairness and to reduce starvation.

End-to-end Evaluation. We evaluate FusionFS on a wide range of microbenchmarks, macrobenchmarks (Filebench [56]), and applications like LevelDB [5], Snappy compression [19], and Linux file encryption [2]. FusionFS, by using CISC$_{Ops}$ reduces dominant I/O overheads leading to 6.12X gains over the NOVA kernel file system [61], 6.12X over the user-level SplitFS, and 1.65X over the firmware-level CrossFS design. Application workloads like LevelDB [5] and Snappy compression [19] show gains up to 6.12X and 2.43X over user-level SplitFS. To highlight the benefits of CISC$_{Ops}$ as a general principle for kernel file systems, we extend ext4-DAX with CISC$_{Ops}$ and showcase the gains. Next, the proposed fine-grained crash consistency (MicroTx) combined with automatic recovery accelerates filesystem recovery by 2.65X. Further, CISC$_{Ops}$ support for LevelDB's restart-after-failure code accelerates recovery by 3.58X. Finally, the CFS-based device-CPU and RAM management reduce unfairness and improve storage efficiency.

The source code of FusionFS is available at `https://github.com/RutgersCSSystems/FusionFS`

2 Background and Related Work

Hardware Near-storage Processing Advancements. Although modern solid-state and nonvolatile memory storage devices have significantly accelerated I/O performance, software and hardware data access cost continues to be expensive. This has motivated hardware vendors to move away from legacy storage controllers with wimpy device cores for handling firmware functionalities (e.g., FTLs [33]) and support powerful in-storage compute. For example, ARM is introducing CSDs with Cortex-R82 64-bit 16-core processors (yet to be commercially available) [11] . In contrast, products like Newport CSDs with 16GB device-RAM, 1.5GHz 16 core processors, and TCP/IP stack support run Linux OS inside the CSD [20]. Finally, FPGA-based CSDs, such as SmartSSD [21], LSM-FPGA [14], ScaleFlux's CSD [52], implement fixed functions (e.g., filtering, compression, and encryption) and continue to evolve.

Software Innovation and Limitations. Software inno-

Properties	KernFS	UserFS	DeviceFS	Computing Offload	FusionFS
Direct-I/O	✗	Partial	✓	✗	✓
Reduce data copy	✗	Partial	Partial	Partial	✓
Reduce PCIe cost	✗	✗	✗	Partial	✓
In-storage Mgmt.	✗	✗	✓	✗	✓
In-storage process	✗	✗	✗	✓	✓
Durability	Data	Data	Data	Data	Data and compute
Resource Mgmt.	✓	✗	✗	✗	✓
Security	✓	Partial	✓	Partial	Same as KernFS

Table 1: Capabilities and Limitations of State-of-the-art Storage Approaches. The last column shows our proposed FusionFS.

vations for modern storage can be broadly categorized as (a) kernel file system (KernFS) and user-level file system (UserFS), (b) in-storage firmware file systems (DeviceFS) and key-value stores (DeviceKV), and (c) computational offloading (comp. offload) solutions mainly for processing. In Table 1(b), we qualitatively compare these designs.

Host-level KernFS and UserFS: Modern KernFS designs for fast storage devices reduce software indirections (e.g., page cache) and guarantee fundamental properties like crash consistency, security, and data sharing [58, 61]. Yet, the I/O overheads such as system calls, data movement, communication latency, and concurrency bottleneck continue to be a problem. An alternative trend is the re-introduction of UserFS designs aimed to bypass the OS (e.g., Strata [35], SplitFS [28], and others [42, 57, 37, 38]). While effective for applications that execute in isolation, a lack of a trusted computing base (e.g., OS) makes it challenging to handle security, data sharing, or multitenancy [30, 37, 38]. In contrast, hybrid designs like SplitFS [28], depend on the OS for metadata management, which could increase I/O overheads. Importantly, most UserFS designs fail to reduce data movement between the host and the storage and do not utilize in-storage compute.

Device-level File Systems (DeviceFS): As an alternative design point, prior work explored offloading file systems [30, 44, 45] and key-value stores [29, 49] inside CSDs and providing applications with direct-I/O. However, these designs generally lack data processing capability. DevFS [30] offloads file system into the firmware, whereas CrossFS [44] exploits parallel I/O queues for I/O scaling. CrossFS and DevFS reduce system calls, but data movement and communication costs remain. Prior solutions have also explored offloading key-value stores inside CSDs [49, 29, 13], which could benefit a specific class of applications that do not require file systems. Unfortunately, issues like high I/O overheads (e.g., data movement) and lack of near-storage processing and resources fairness remain in these designs.

In-storage Computation: In-storage computation systems primarily offload specific functions to the CSD. For example, seminal systems such as CASSM [54], RARES [36], and Active-Storage [46] offloaded database search and scan operations on slow hard drives. Recently, to benefit from fast storage, runtimes like LSM-FPGA [62], PINK [25], KEVIN [34] and others [11, 30, 44, 48] redesign and offload database compaction to FPGA-based CSD, whereas Newport OS deploys a specialized OS for offloading functions [20]. However, these systems lack storage management, do not handle critical storage properties like data durability, security, and sharing, and depend on the host-level file system.

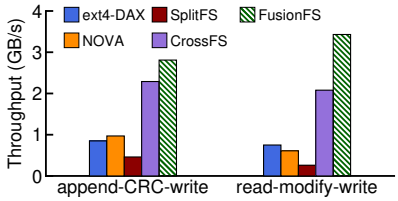


Figure 1: Analysis. Y-axis shows cumulative throughput when using 16 applicaiton threads.

Batching Operations. To reduce I/O cost, several I/O batching strategies have been proposed. Traditional file systems support vectored I/O, but it is restrictive and only supports data plane operations (e.g., writev, readv). Notably, all operations packed in a vector must be the same. Further, vectored I/O only provides coarse-grained all-or-nothing durability. Next, Chen et al. propose NFSv batch remote NFS I/O operations of the same type to reduce network latency [15], whereas TC-NFS [55] extended NSFv to support compound transactions. In contrast, FusionFS designs $CISC_{Ops}$ to reduce dominating I/O overheads by organically aggregating non-similar I/O sequences that could contain data-plane, control-plane, and processing operations. FusionFS also provides fine-grained durability and recovery without compromising in-storage resource fairness. We also showcase the benefits of $CISC_{Ops}$ for traditional file systems (§ 5).

3 Motivation

To motivate the need for reducing dominating I/O overheads like kernel/userspace crossing, data movement cost, and communication cost between host and device, we study the performance of state-of-the-art designs: KernFS ext4-DAX [58] and NOVA [61] (a log-structured design) designed for fast NVMs; hybrid UserFS SplitFS [27]; DeviceFS CrossFS [44]; our proposed FusionFS.

We use two workloads modeled after real-world applications: (1) an I/O-intensive *read-modify-write* that opens a 12GB file, continuously reads 4K blocks, updates, and writes them back depicting databases, key-value stores, and others; (2) an I/O + processing-intensive *append-checksum-write* (hereafter referred to as *append-CRC-write*) workload that appends data, computes checksum, and writes the data, replicating the behavior of several applications like key-value stores (LevelDB [7]), web-servers [56]. For brevity, we show results for 16 thread configuration of the benchmarks and show thread sensitivity in § 5. Because state-of-the-art systems use NVM as storage, we use a machine with 512 GB DC Optane NVM for storage, 64 CPUs, and 32 GB DRAM [6].

In Figure 1, the y-axis shows the throughput. First, kernel-level ext4-DAX provides direct access without data copies to page cache but incurs significant system call and data

copy cost for *read-modify-write* workload leading to substantially lower throughput. With its multicore-friendly and log-structured design, NOVA reduces some I/O overheads (e.g., avoiding double writes for a journal), but other overheads remain. Next, hybrid user-level SplitFS memory-maps storage to userspace and replaces reads/writes with loads/store operations. SplitFS reduces system calls, but metadata updates require frequent OS interaction. We also observe high OS locking and pre-paging costs for supporting user-level memory-map for 16 threads, leading to poor performance in both workloads. CrossFS, an emulated firmware-level file system design, reduces system calls and only metadata movement between filesystem and storage, resulting in higher performance. In contrast, FusionFS eliminates data movement significantly as well as host and device interaction by offloading both I/O and data processing. In § 5, we show the breakdown of FusionFS benefits for these workloads.

4 FusionFS Design

We next discuss FusionFS's design principles, followed by system architecture, mechanics for supporting $CISC_{Ops}$, support for fine-grained durability and fast recovery, permission management, and in-storage resource management.

4.1 Principles

1. Co-design in-storage management and data processing to eliminate dominating I/O overheads. We design a near-storage file system that combines storage management and data processing to reduce dominating I/O overheads such as system calls, data movement, and communication costs.

2. Design abstractions to reduce host and device interactions. We design $CISC_{Ops}$, a novel approach to fuse identical and nonidentical I/O and data processing operations. $CISC_{Ops}$ aggregate a sequence of I/O and processing operations and utilize device-CPUs to reduce data movement and communication between the host and the storage. We also explore an application-explicit and transparent approach (without data processing).

3. Exploit in-storage compute for fine-grained crash consistency and faster recovery. We design fine-grained crash consistency, micro-transactions (MicroTx), which persist all operations (including intermediate processing state) and reduce data loss in case after a failure. MicroTx uses an operational log and device-CPUs for automatic recovery by completing unfinished $CISC_{Ops}$ for faster recovery.

4. Manage in-storage resources for fairness and performance efficiency across tenants. We design a completely fair device-CPU and device-RAM scheduler (CFS) for fairness and for avoiding starvation across tenants.

4.2 System Architecture

To support direct-I/O design, FusionFS designs a user-level library (UserLib) and in-storage (StorageFS) components that work in tandem to offload I/O and computation without compromising correctness, crash consistency, recovery, security, and resource fairness. Figure 2 shows the high-level design of FusionFS. Applications issue traditional POSIX I/O requests or $CISC_{Ops}$ adding them to an inode-queue. StorageFS checks permission for each I/O request, dequeues, and schedules for processing, but importantly also provides fine-grained durability and recovery.

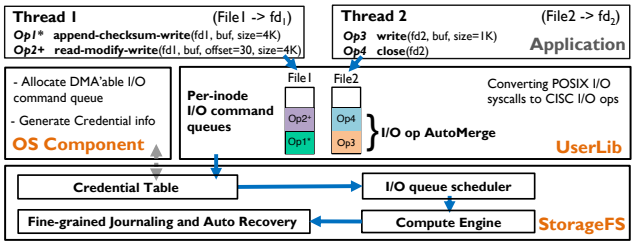


Figure 2: FusionFS High-level Design. Figure shows the high-level design of FusionFS with the UserLib, the StorageFS, and the OS components. For thread1, Op1 and Op2 show a $CISC_{Op}$ with data processing, whereas Op3 and Op4 show simple I/O. StorageFS shows the in-device structure with durability, permission, and scheduling components.

4.2.1 User and Device Components

We next discuss the details of user-level UserLib and in-storage StorageFS layers.

UserLib. This is an untrusted per-process user library to interface applications to the in-storage file system using host-CPUs. Beyond supporting POSIX I/O through interception and $CISC_{Ops}$, UserLib also sets up I/O queues, prepares requests, and handles errors.

I/O Commands. For regular POSIX I/O, application changes are not required, whereas $CISC_{Ops}$ require some changes to either use pre-defined UserLib $CISC_{Ops}$ or construct one. Table 2 shows select pre-defined Compute + I/O and I/O-only $CISC_{Ops}$. In § 4.3, we discuss principles and mechanics of constructing $CISC_{Ops}$ and the limited support for application-transparent I/O-only $CISC_{Ops}$.

inode-queue. To exploit modern NVMe's 64K hardware I/O queues and increase parallelism across files, we use a dedicated per-file I/O queue, referred to as inode-queue. The inode-queues buffer requests and intermediate data to be processed by StorageFS. The queues are created using a DMA'able memory region when opening a file [30, 44]. The OS maps the DMA regions, which can be accessed by the host and the device layers. By default, each inode-queue has 32 entries for inserting I/O commands and 2MB of data buffers, but the number of entries is configurable depending on host DRAM availability.

StorageFS. It is the heart of FusionFS design responsible for traditional file system functionalities like metadata and data management and permission control (§ 4.5). StorageFS houses compute engine (§ 4.3), supports traditional and fine-grained data and metadata journaling (§ 4.4) and recovery (4.4.1), and implements resource schedulers (§ 4.6). StorageFS is designed for general-purpose device-CPUs (e.g., ARM CSD) [11] and implements simple in-memory and on-disk filesystem structures such as a super-block, bitmap blocks, inode blocks, and data blocks (see Figure 2). For in-

Type	Ops. Sequence	Overheads			
		Data move	Syscall	Comm.	Resource
I/O-only	open, read, write	Hi	Hi	Med	Lo
I/O-only	open, read, close	Med	Hi	Hi	Lo
I/O-only	write, fsync	Med	Hi	Med	Lo
I/O-only	read, update, write	Med	Hi	Hi	Lo
Compute+I/O	write, CRC, write	Med	Hi	Hi	Med
Compute+I/O	read, CRC	Med	Med	Med	Med
Compute+I/O	read, compress, write	Hi	Med	Hi	Hi
Compute+I/O	read, encrypt, write	Hi	Med	Med	Hi

Table 2: Frequently used select I/O-only and I/O+Compute operation sequences and their overheads. **Hi**gh, **Med**ium, and **Lo**w indicate the relative magnitude of overheads. The columns, Data move, Syscall, Comm., and Resource denote data movement, system call, communication between host and device, and compute and memory usage, respectively.

storage computation, StorageFS parses through all operations in a $CISC_{Op}$ vector, executes, and returns operation-specific return codes. Internally, the StorageFS compute component currently supports several data processing operations like checksum, compression, encryption, and decryption, beyond just parsing and sorting operations. To address the lack of programmable hardware, StorageFS prototype is currently implemented as a device driver with separate CPUs, memory regions, and disk with carefully emulated hardware parameters. We next discuss the details of each StorageFS component.

4.3 Operation Types and $CISC_{Ops}$ Mechanics

We first discuss operation types to offload, followed by the mechanics for offloading.

4.3.1 Operation Types

Applications generally access storage using (1) simple I/O operations to store or read state, (2) issue a sequence of I/O operations, or (3) access, process, and store data. The processing could vary from operations like compression, encryption, checksumming, or complex transformations that search, sort, or even execute ML operations (e.g., add, multiply, XOR). Reducing I/O overheads across all such operation types is critical.

Offloading Simple I/O Operations. For basic POSIX I/O, UserLib intercepts system calls and adds them to inode-queues. We extend the NVMe commands to add new operation codes for representing filesystem operations (e.g., read, write, open, close). After adding a request (command), UserLib sets a doorbell for the StorageFS to start processing, which sets the request's commit flag after completion. All blocking (e.g., *read*) and non-blocking (*write*) data plane operations are added to inode-queues, whereas metadata-intensive operations (e.g., *mkdir*) use a separate global metadata I/O queue. For error-prone operations like file and directory rename [12], FusionFS uses global file system locks.

$CISC_{Op}$ Operations. We next discuss UserLib support to identify and aggregate identical and non-identical I/O and data processing operations.

Identical and Non-identical I/O Operations. We observe that in several applications, I/O operations are executed in sequence or pairs. For example, Figure 3(a) shows a widely-used NoSQL database and webserver sequence that opens, writes, syncs, and closes the file when inserting values (i.e., open()->write()->sync()->close()) or when reading data [5]. The figure also shows overheads for each operation, which includes system calls between application and the OS (S), data movement (D), metadata movement (M) such as inode, and host-storage communication (PCIe or memory bus) cost (C). Note that the direct-access (DAX) filesystems for NVM incur one less data copy compared to the block-level file system. We observe several such sequences contributing to I/O overheads as listed in Table 2. In contrast, $CISC_{Ops}$ aggregates and offloads such sequences to StorageFS reducing I/O overheads (see Figure 3(b)).

I/O and Data Processing Operations. For supporting data processing + I/O operations, as opposed to full application redesign [14] for CSDs, we aim to reduce application changes for organically supporting I/O and their related pre and post-processing to reduce I/O overheads. Specifically, as shown in Table 2, we observe that applications frequently fetch I/O data to perform operations like checksum generation (CRC), compression/decompression, encryption, search, sort, and ML operation pairs (e.g., XOR, multiplication). For example, as shown in the code snippet in Figure 3(c), NoSQL persistent stores like LevelDB avoid expensive file commits or propagation of corrupted data by adding CRC for integrity check. After each file system append() system call, the CRC is computed and appended to the actual data. This sequence incurs 2 system calls, 4 data and metadata copies (2 for DAX file systems) , and 2 PCIe operations in OS file systems (see Figure 3(a)). The above sequence repeats for all data reads or replication to other nodes to check data integrity. In contrast, an *append-CRC-write* $CISC_{Ops}$ offloaded to StorageFS significantly reduces I/O overheads to 1 data movement and a PCIe operation without incurring system calls (see Figure 3(b)).

4.3.2 Mechanics for Application-explicit Support

With the explicit approach, applications can use $CISC_{Ops}$ pre-constructed by UserLib or construct custom $CISC_{Ops}$.

$CISC_{Op}$ Command Structure: Each $CISC_{Op}$ is a vector of commands in an extended NVMe format [59] for supporting multiple operations and added to inode-queue for processing. For example, Figure 3(d) shows the code snippet for packing a *append-CRC-write* $CISC_{Op}$ in LevelDB. Each $CISC_{Op}$ vector element contains an operation code (*opc*), input and output address to specify DMA'able address from which data must be fetched or stored (*in*), I/O offset (*slba*), block count (*nlb*), and return code (*retcode*), and a journal commit flag (*commit*). The number of elements in the $CISC_{Op}$ is configurable, and by default, can pack *32* operations to fit in a DMA-able page. Furthermore, FusionFS could also combine multiple $CISC_{Ops}$ to batch operations.

Specifying Dependency. Applications or UserLib developers can specify inter-dependencies across operations in a $CISC_{Op}$. For example, as shown in Figure 3(e), for the *append-CRC-write*, the CRC depends on the input of previous *append* and the bytes actually written to storage, which is unknown until *append* completes. The input dependencies can be specified

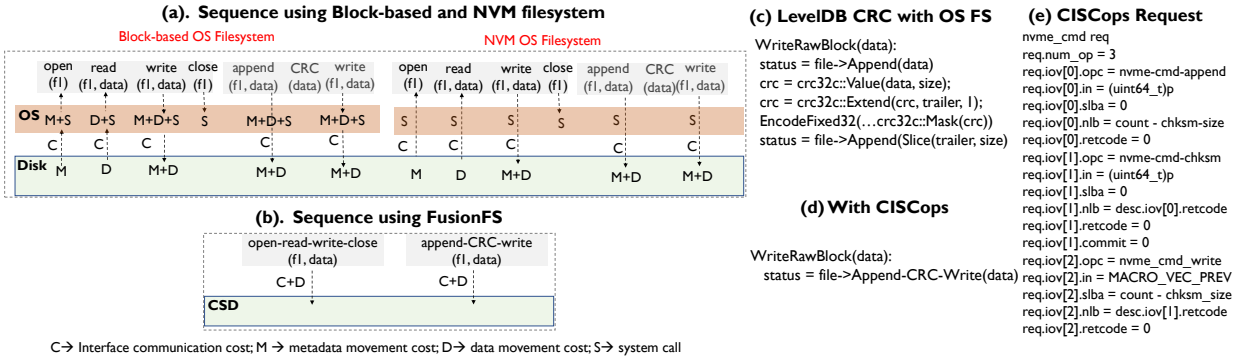


Figure 3: Comparison of I/O Overheads. (a) and (b) compare data movement (D), communication cost (C), and system call (S) using traditional storage (top) and envisioned $CISC_{Ops}$ design that bypasses OS (bottom); (c) and (d) show CRC (checksum) code sequence in vanilla LevelDB (top) and proposed ($CISC_{Ops}$); (e) shows packing a $CISC_{Ops}$ request. For NVM direct-access filesystems, the data copies are reduced to one per operation. Metadata caching could reduce I/O.

using the DMA address (e.g., *req.iov[1].in = (uint64_t)p* for checksum input) and the bytes to process (*req.iov[1].nlb = desc.iov[0].retcode*).

Concurrency and Ordering. FusionFS supports out-of-order processing for concurrency by default (e.g., vectored writes or combination of reads and writes without conflicts) as well as in-order processing (e.g., *append-CRC-write*). To prevent out-of-ordering, the $CISC_{Op}$ structure allows specifying an "order" field to execute the operation sequentially (e.g., *req.iov[1].order = MACRO_VEC_PREV*), which are otherwise parallelized in the presence of free device cores.

4.3.3 StorageFS Compute Engine

Inside the storage, StorageFS implements a generic parser to disassemble $CISC_{Ops}$ and either execute them with file system logic for basic I/O or use the compute engine for data processing. We have currently added new data processing functionalities to the compute engine, which would require firmware upgrade [53]. However, we will explore alternative dynamic code injection techniques (e.g., eBPF [4]).

4.3.4 Partial Support for Automatic Offloading

FusionFS enables a partial support for transparently generating, merging, and offloading $CISC_{Ops}$ for a group of I/O-only operations without data processing, referred to as *AutoMerge*. This is useful when application changes are not feasible. AutoMerge primarily reduces system calls, host-to-device interaction, and overheads such as command generation, I/O queuing, and scheduling but not necessarily data movement. AutoMerge can either merge non-dependent operations on the same file by different threads or asynchronous operations. UserLib parses all pending operations in a file's inode-queue to generate $CISC_{Ops}$. For example, consider multiple *append-CRC-write* to different blocks of the same log file across threads or a sequence of asynchronous writes. AutoMerge can aggregate two non-dependent operations – *append* in a *append-CRC-write* with *write* in previous *append-CRC-write* – to generate $CISC_{Ops}$. We study the benefits and implications of AutoMerge in § 5.

Limitations. While our AutoMerge provides simple batching, it currently lacks support for offloading data processing operations (a harder problem). Similarly, it is ineffective for single-threaded applications with blocking I/O. We plan to explore automatic data processing offloading (a harder challenge) and other limitations in our future work.

4.4 Supporting Durability and Fast Recovery

We next discuss the support for basic journaling and fine-grained crash-consistency support for offloaded POSIX I/O and $CISC_{Ops}$, respectively, followed by the support for automatic recovery after failure by utilizing in-storage compute.

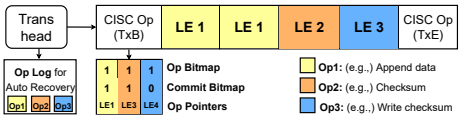


Figure 4: Micro-transaction (MicroTx) Design. Figure shows an example of *append-CRC-write*.

4.4.1 Traditional Journaling

For POSIX I/O, StorageFS supports traditional REDO journaling. The journal log-entry headers contain a unique transaction ID followed by the metadata log (with a pointer to actual data) and a transaction commit bit. Before a transaction commits, first the data is updated in place, followed by metadata logging.

To support crash consistency for $CISC_{Ops}$, FusionFS first provides an "all-or-nothing" macro-transaction (MacroTx), which wraps an entire $CISC_{Op}$ into one transaction and recovers only if the entire $CISC_{Op}$ is committed (to the log) before the failure. This simple approach resembles crash-consistency support in today's vectorized I/O. However, this approach risks losing I/O data and computation state when $CISC_{Ops}$ do not complete.

4.4.2 Durability & Automatic Recovery with MicroTx

We overcome the limitations of MacroTx by designing MicroTx that provides fine-grained durability of all I/O and processing operations of a $CISC_{Op}$, which we refer to as micro-ops. Each micro-op can be independently committed and recovered after a system failure. Figure 4 illustrates a MicroTx structure with *append-CRC-write*, where a log-entry for append, checksum, and write micro-ops are independently committed. Each $CISC_{Op}$ log entry uses a bitmap in *TxB* to represent the number of micro-ops (micro-op bitmaps) and the address offset of log entries for each micro-op. *TxB* also

contains a commit bitmap to mark and track committed micro-ops, and *TxE* represents a bit to indicate the completion of an entire $CISC_{Op}$. Because some compute operations (e.g., compression) could have a state larger than the available log entry size (e.g., 48 bytes by default), users can configure log-entry size during filesystem mount. We will explore dynamic log entry sizes in our future work.

Next, we utilize MicroTx and device-CPUs to design an automatic recovery mechanism and redo incomplete $CISC_{Ops}$. This is in contrast to current OS file systems that rely on applications to redo failed operations, which increases recovery time and developer efforts. For auto-recovery (optionally enabled during filesystem mount), MicroTx additionally uses operational log (shown in Figure 4) to write $CISC_{Ops}$ and the input data similar to data logging before processing a request. After logging, a "commit" flag is set and used as a receipt by an application. In case of a failure, MicroTx's recovery first recovers all committed micro-ops, followed by recovering $CISC_{Op}$ and input data using the operational log and then executes all incomplete micro-ops in $CISC_{Ops}$. In Table 10c, we show the benefits of automatic recovery in reducing recovery and restart time after a system crash or failure by reducing I/O costs. Importantly, MicroTx and automatic recovery could reduce application/developer effort to check and redo incomplete I/O operations.

4.4.3 Error Handling

To handle errors (e.g., insufficient disk space), for application-explicit $CISC_{Ops}$, when using an all-or-nothing MacroTx, FusionFS aborts the entire sequence and also updates the return code for the operation that caused the failure. In contrast, when using MicroTx, all operations starting from the erroneous micro-op are aborted with an error return code. However, FusionFS could potentially execute all subsequent non-dependent operations in the sequence. For example, in a $CISC_{Op}$ with 10 writes, a large write (say, the 6th write) could fail due to the lack of disk space, but subsequent smaller writes (7 to 10) could succeed. For the transparent AutoMerge, because applications expect independent execution of micro-ops, we allow execution of all I/O operations.

Potential (infrequent) errors could occur during automatic recovery (say after a system crash and restart). With MicroTx and operational log enabled, we retain the journaled micro-ops, abort the erroneous operation, and use operational log to report the failure with file name, operation type, and error type, which is later checked by UserLib to identify errors. Beyond our current design, a careful exploration of opportunities to reduce $CISC_{Op}$ aborts and failures, and correctness issues is critical.

4.5 Permission Checking and Data Sharing

We aim to match the security guarantees of OS file systems for both POSIX I/O and $CISC_{Ops}$ by satisfying the following assertions: (1) only processes with the right permission can access a file or directory; (2) the file system metadata is updated only by a trusted entity; (3) for inter-processes file sharing, only legal writers can update the data.

FusionFS achieves these goals by utilizing trusted OS but without compromising direct I/O. First, the OS shares credential information of a process with StorageFS during process initialization, and StorageFS maintains a per-process credential table similar to prior designs [44]. Second, inode-queues and their DMA'able memory regions (e.g., when opening files) are created by the OS only when requested by a process with the right credentials. Third, access to inode-queues (i.e., DMA'able memory pages) is protected by virtual memory protection, preventing illegal access by a malicious process. Finally, the OS shares credentials with StorageFS. For all requests in the inode-queue, StorageFS checks permission for operations packed in a $CISC_{Op}$ by comparing against the credential list before processing, thereby avoiding partial execution. For example, in a *read-compress-write* $CISC_{Op}$ issued to a read-only file, FusionFS' permission manager does not allow partial $CISC_{Ops}$ execution.

Data Sharing. Supporting direct-I/O via user-level library and inode-queues complicates secure inter-processes file sharing. When a file is shared and accessed across readers and writers, the inode-queues used for dispatching requests are also shared. Unfortunately, a reader process (with read-only permission) could accidentally or maliciously corrupt I/O or data processing requests issued by writers. To overcome these complexities, we employ the following design. First, all legal writers (without readers) can concurrently access and update a file, similar to OS file systems. Second, similar to KernFS and UserFS designs, applications are responsible for ordering updates to an inode-queue (e.g., using lease-based locks [35]). However, in the presence of readers (a file opened with read-only permission), to prevent corruption of writer requests in the inode-queue, FusionFS detects file opened with read-only permission and delegates the trusted OS to add I/O commands from both writers and readers after permission checks.

4.6 Resource Management

We introduce in-storage compute and memory-centric scheduling to enable in-storage resource fairness across tenants, avoid starvation, and improve performance efficiency.

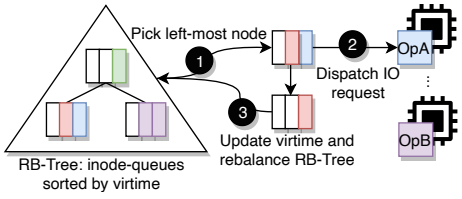


Figure 5: FusionFS scheduler high-level overview

4.6.1 FusionFS Compute-Centric Scheduling

The number of application threads that issue POSIX I/O and $CISC_{Ops}$ could exceed available CSD compute cores. Specifically, compute-intensive data processing (e.g., checksum, compression) will increase in-storage compute use, leading to workload imbalance, starvation, and impacting the

performance of POSIX I/O across tenants. Unfortunately, conventional I/O schedulers and prior in-storage schedulers are I/O centric (e.g., Linux *blk-queue*) and fail to consider device-CPU usage.

In-storage CFS Compute Scheduler. In FusionFS, we take inspiration from the OS CFS CPU scheduler and explore its use for device-CPU scheduling [1]. We account for I/O and data processing operations that use device-CPUs. Figure 5 provides a high-level overview. At a high level, for CPU fairness, StorageFS scheduler selects and dispatches requests from inode-queues (of processes) with the least CPU usage (i.e., virtual CPU runtime). However, unlike OS schedulers, in-storage StorageFS lacks process state, which makes bookkeeping challenging, specifically for keeping track of virtual runtime (*virtime*) usage of device-CPUs by each process. We overcome this by using the inode-queues of an application that buffers I/O requests to track and bookkeep device-CPU usage. For each I/O or data processing request dispatched from an inode-queue, FusionFS increments the *virtime*, and selecting a request from an inode-queue with the least *virtime*.

Internally the scheduler maintains a global red-black tree (RB-tree) that stores reference to inode-queues, sorted by their *virtime* (see Figure 5). Initially (after mount), the scheduler uses a round-robin approach to pick an inode-queue, but once requests are dispatched, their *virtime* are updated, and the tree is rebalanced. Note that the scheduler always dispatches requests from the left-most RB-tree node that points to inode-queue with the least *virtime*. While we currently implement a simple RB-tree, we will explore alternative device-optimized (i.e., firmware) data structures [47].

Fairness Across Tenants. We employ a two-pronged approach to prevent a greedy process (tenant) from increasing compute share by increasing inode-queues and starving other tenants. First, the trusted OS tags each inode-queue with a process ID, and StorageFS maintains the overall *virtime* of each process using the sum of all its inode-queue's *virtime*. The scheduler always selects a request from an inode-queue with the smallest process-level and inode-queue-level *virtime*. Second, an administrator can limit the number of inode-queues per process.

Request Termination and Preemption. To handle misbehaving or long-running requests with large inputs, StorageFS first terminates the request for avoiding starvation, then sets error codes for the request, finally clearing the transaction states (logs). Our ongoing work is exploring request preemption, which requires committing intermediate state of CISC_{Ops} using MicroTx and switching to other operations. However, preemption introduces correctness challenges. For example, preempting an *append-compress-write* after the compress operation could lead to incorrect read operations on the same file other threads.

4.6.2 Memory-Centric Scheduling

Beyond device-CPUs, efficient management of device-RAM is critical for fairness. Although modern CSDs are equipped with 4-16GB of memory, a combination of in-storage data processing, filesystem operations, and FTL's logical to physical block translations could increase memory contention and starvation across clients [16, 43]. For example, offloading memory-hungry file compression with large inputs could starve or block other CISC_{Ops} (e.g., *append-CRC-write*) or POSIX I/O from other tenants even when free compute cores are available.

We overcome the above challenges by extending the CFS to share device memory capacity efficiently. First, we implement a simple slab allocator for allocation and deallocation. Next, we enhance the CFS scheduler with memory usage (*memuse*) accounting for each process and inode-queues and maintaining a memory-specific RB-tree with per-inode *memuse*. When device CPUs are not a bottleneck, the scheduler selects a process and inode-queue with the least *memuse*; this avoids blocking or failing other requests. Finally, in § 5, we evaluate the benefits of FusionFS' memory-centric scheduling with the multitenant workload. An ideal scheduler must provide multi-resource fairness across compute, memory, and other resources (e.g., using Dominant Resource Fairness [23]), which we will explore in our future work.

4.7 Emulation and Application Changes.

Due to a lack of a programmable storage device, FusionFS is implemented as a device driver. We us Intel Optane Memory [6] for storage similar to prior work [30, 44, 26]. We emulate device-CPU and device-RAM by using dedicated CPUs and memory managed by StorageFS. To emulate PCIe latency, we add 900-1000ns delays [40] for all interactions between the host and the device. Finally, the storage and the device-RAM bandwidth could vary across vendors. To understand the implications, we use DRAM thermal throttling and study the impact [32, 24]. Note that bandwidth throttling works only for DRAM technology and in older Intel Haswell architectures. Therefore, we use a multi-socket DRAM-based system to emulate and vary memory and storage bandwidths.

We built FusionFS by extending CrossFS' direct-I/O support and adding CISC_{Ops}, fine-grained journaling, fast and automatic recovery, efficient and fair in-storage resource management, and optimizations to improve device-compute scalability. UserLib and StorageFS components add 3K and 11K LOC, and the data processing operations like compression, checksum, and decryption functions add 2.4K lines of code. Finally, to use CISC_{Ops}, LevelDB requires < 38 LOC changes replacing the CRC logic, whereas file encryption and snappy compression require < 21 LOC changes.

5 Evaluation

Our evaluation answers the following questions:

- How effective is FusionFS and its CISC_{Ops} abstraction in reducing I/O overheads across microbenchmarks and macrobenchmarks?
- How sensitive are FusionFS gains towards device-CPU frequency and memory and storage bandwidth?

CPU	Intel Xeon(R) Gold 3.1GHz, dual-socket, 64-core
DRAM	96GB DDR4 2666 MT/s
NVM	256GB Intel Optane DC PMM (2*128GB)
Device-CPU	4-cores, Fast (2.7GHz) and Slow (1.2GHz) CPUs
Device-RAM	2GB dedicated for device operations
PCIe Latency	900us

Table 3: Experiment Platform and Setup. The device-CPUs and device-RAM are emulated through DVFS and thermal throttling.

$CISC_{Ops}$	**ext4-DAX**	**ext4-DAX-$CISC_{Ops}$**	**FusionFS**
append-CRC-write	0.85 GB/s	1.18 GB/s	2.81 GB/s
read-modify-write	0.75 GB/s	0.84 GB/s	3.43 GB/s

Table 4: $CISC_{Ops}$ under traditional ext4 file systems.

- Is $CISC_{Ops}$ effective for traditional OS filesystems?
- Can MicroTx and auto-recovery improve durability and accelerate recovery?
- How effective is the CFS-based compute and memory scheduler in improving resource fairness across tenants?
- What is the overall impact of FusionFS on real-world applications?

5.1 Experimental Setup and Methodology

Due to the lack of programmable CSD, we carefully emulate our FusionFS with the parameters shown in Table 3. Our Optane NVM storage provides 8 GB/sec read and 3.8 GB/sec write bandwidth. We compare. For StorageFS processing, we reserve 4 CPUs [50]. We also study the impact of varying CPU speeds using fast (2.7GHz and default) and slow (1.2GHz) CPUs resembling ARM-based CSDs [11]. For device-RAM, we reserve 2 GB memory managed by StorageFS. We study the impact of device-RAM bandwidth using a 64-core CloudLab machine. For PCIe latency, we add 900ns [40] delay before a request is processed.

Methodology. We compare FusionFS against the state-of-the-art KernFS – ext4-DAX [58] and NOVA [61], hybrid UserFS – SplitFS [28], and DeviceFS – CrossFS [44]. Note that some file systems do not support macro-benchmarks and applications evaluated in this paper. The throughput shown for workloads (in GB/s) combines data (payload) I/O and processing times.

5.2 Benchmark Analysis

We evaluate microbenchmarks and Filebench macrobenchmark [56] to understand $CISC_{Ops}$ benefits and implications.

5.2.1 Microbenchmark

We evaluate I/O data and metadata intensive *file-open-write-close* in Figure 6a, I/O and data processing intensive *append-CRC-write* in Figure 6b, and data plane heavy *read-modify-write* benchmark in Figure 6c. The *file-open-write-close* is modeled after NoSQL databases, file-servers, and web-servers that operate on several files. The workload opens a file, performs a 2MB write, and closes the file, repeating this for 10K files. Next, the *append-CRC-write* benchmark (as discussed extensively in this paper) is used for providing integrity in NoSQL databases [5, 3], key-value stores [10], and others. Each thread appends a 4KB block, computes the checksum, and writes the checksum on a 12GB file. The data plane-intensive *read-modify-write* mimics several widely-used applications [5, 3, 10, 56] by reading a random 4KB block, modifying with random text, and writing back data blocks on a 12GB file.

Workload	**Syscall I/O**	**Direct-I/O**	**Direct-I/O + $CISC_{Ops}$**	**Direct-I/O + $CISC_{Ops}$ + CFS-sched**
append-CRC-write	1.15 GB/s	2.23 GB/s	2.74 GB/s	2.81 GB/s
read-modify-write	0.75 GB/s	1.91 GB/s	2.85 GB/s	2.98 GB/s

Table 5: Breakdown of FusionFS incremental gains.

# of threads	**1**	**2**	**8**	**16**
ext4-DAX	0.26 GB/s	0.66 GB/s	0.99 GB/s	0.85 GB/s
FusionFS-AutoMerge	0.26 GB/s	0.93 GB/s	1.66 GB/s	2.70 GB/s
FusionFS	0.27 GB/s	0.95 GB/s	1.94 GB/s	2.81 GB/s
FusionFS-Batch	0.29 GB/s	1.05 GB/s	2.06 GB/s	2.98 GB/s

Table 6: FusionFS Optimizations. (append-CRC-write).

Methodology. We vary the number of benchmark threads in the x-axis, and the y-axis shows the throughput (GB/sec), and the threads use separate files. We compare ext4-DAX, NOVA, SplitFS, CrossFS, and FusionFS. Additionally, to understand the impact of slower device-CPUs, we also evaluate in-storage StorageFS to use 1.2GHz device-CPUs (*CrossFS-slow-device-cpu* and *FusionFS-slow-device-cpu*).

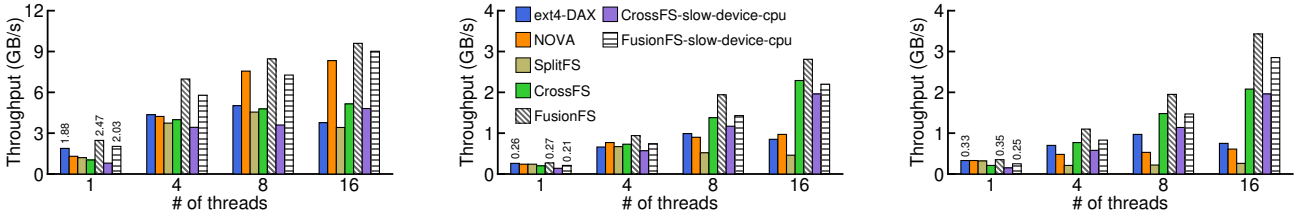
Observation. As shown in Figure 6a, in KernFS designs ext4-DAX and NOVA, each I/O operation incurs system call, data copy, and the device communication latencies, resulting in high I/O overheads. However, NOVA performs better than ext4-DAX due to its log-structured design and per-CPU (multicore-friendly) block management. Next, SplitFS, a hybrid UserFS, memory-maps staging files to userspace and performs load and store operations. However, SplitFS uses the OS for metadata operations. Specifically, we observe increased kernel overheads that increase with workload size and thread count from metadata operations, internal data copies, and block lookup and pre-paging (MAP_POPULATE) cost for the userspace mmap'ed files that stage I/O.

In contrast, in-storage CrossFS bypasses the OS and avoids system calls but suffers from data copies between the host and the device (for read and write I/O), PCIe latency (hardware), and the software cost to allocate, enqueue, and dequeue requests. The blocking *reads()* also stall the host CPUs.

Finally, FusionFS merges the open->write->close into a *file-open-write-close* $CISC_{Ops}$, avoids system calls, reduces a data copy between the application and the OS, and the PCIe latency, all leading to up to 4.58X gains over ext4-DAX, 6.12X over SplitFS, and 1.65X over CrossFS. Table 5 shows the incremental benefits of FusionFS' design optimizations.

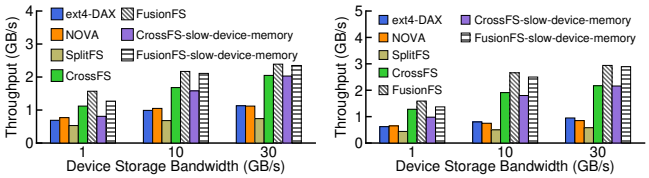
Next, as shown in Figure 6b, for *append-CRC-write*, similar to *file-open-write-close*, prior approaches lack in-storage compute capability for CRC, resulting in two system calls (except CrossFS) and a data copy. In contrast, FusionFS improves performance over ext4-DAX, SplitFS, and CrossFS by up to 3.3X, 6.1X, and 1.3X, respectively. Finally, for *read-modify-write*, FusionFS outperforms all other systems.

Device Compute Speed. In Figure 6, we show *FusionFS-slow-device-cpu* and *CrossFS-slow-device-cpu* configurations using slower device compute by throttling them to 1.2GHz. FusionFS outperforms CrossFS and, importantly,



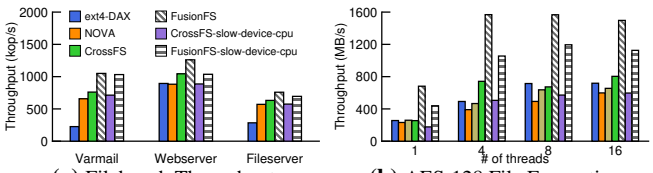
(a) Open-Write-Close, 10K files, 2MB each file **(b)** Append-CRC-Write **(c)** Read-Modify-Write

Figure 6: Microbenchmarks. Shows aggregated throughput. Threads operate on separate files. CrossFS and FusionFS use 4 device cores.



(a) Append-CRC-Write **(b)** Read-Modify-Write

Figure 7: Sensitivity to Device Bandwidth. The x-axis shows the device storage bandwidth. For CrossFS and FusionFS, we show results when using slow device memory throttled to same bandwidth as storage.



(a) Filebench Throughput. **(b)** AES-128 File Encryption

Figure 8: Macrobenchmark. (a) shows the Filebench performance with three workloads. (b) varies the number of threads when encrypting target files

other filesystems that use 2.5× faster host CPUs, highlighting the importance and benefits of reducing I/O overheads.

Impact of Memory and Storage Bandwidth. The bandwidth of device storage and device-RAM could vary across vendors. To understand the sensitivity, in Figure 7, we use DRAM thermal throttling to study the impact. As discussed earlier, we use a 64-core dual-socket CloudLab machine [17] that uses DRAM as storage. In Figure 7, we vary the storage bandwidth from 1GB to 30GB (maximum without throttling) along the x-axis. The bars *CrossFS-slow-device-memory* and *FusionFS-slow-device-memory* represent the case where both CrossFS and FusionFS use slower but the same device-RAM and storage bandwidth by pinning them to a slower NUMA socket.

Observation. FusionFS consistently delivers higher gains. For example, when varying the storage bandwidth without slower device-RAM results in 1.35x and 3.09x gains over CrossFS and ext4-DAX. Next, when the device-RAM bandwidth is also reduced, FusionFS' throughput reduces but is still higher than other systems. *The results show that dominating I/O overheads add more constraints for storage-intensive workloads, and reducing them is critical.*

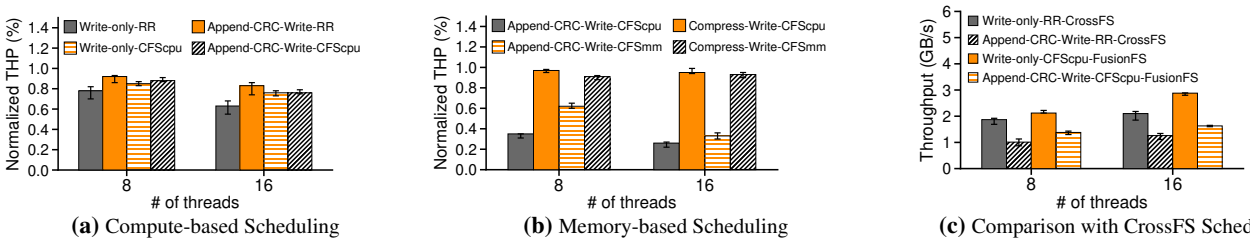
Effectiveness of CISC$_{Ops}$ for ext4-DAX. To understand the effectiveness of CISC$_{Ops}$ as a general principle for all file systems, we extend ext4-DAX to support *append-CRC-write* and *read-modify-write* CISC$_{Ops}$ and compare the throughput against vanilla ext4-DAX and FusionFS in Table 4. The vanilla ext4-DAX incur two system calls and two data moves (see Figure 3a) for both workloads. Next, ext4-DAX-CISC$_{Ops}$ reduces system calls by half with one data copy leading to better throughput. In contrast, FusionFS eliminates all system calls and one data copy leading to 4.08x higher throughput.

Optimizations: Application Explicit Batching and Transparent CISC$_{Ops}$. First, FusionFS can batch multiple, non-dependent CISC$_{Ops}$ as a vector to eliminate data copy, system call, and other software overheads, such as enqueuing and dequeueing requests. The number of requests to batch depends on the available DMA memory used as a data buffer for each inode-queue (a configurable parameter in FusionFS). Table 6 shows the performance for explicitly batching 10 CISC$_{Ops}$ (*FusionFS-Batch*) and varying the number of threads. As shown, FusionFS with batching shows 5.02x and 1.45x gains, respectively, compared to not batching.

Next, as discussed in §4.3.4, FusionFS provides partial support for transparently generating and offloading CISC$_{Ops}$ by aggregating non-dependent and pending I/O requests in an inode-queue, mainly for asynchronous I/O or requests across multiple threads but without offloading data processing. As shown in Table 6, *FusionFS-AutoMerge* provides gains over ext4-DAX for higher thread counts, but as expected, the application-explicit approach outperforms all cases.

5.2.2 Macrobenchmark - Filebench

To validate the microbenchmark gains, we next evaluate FusionFS for the widely-used Filebench [56] in Figure 8a. We use the *fileserver*, the *webserver*, and the *varmail* workloads. The fileserver opens a file, randomly appends 16K bytes, and closes the file. In FusionFS, these operations are aggregated to one *file-open-write-close* and offloaded using a temporary file's inode-queue. The webserver opens a file, reads the whole file, and closes it, which is aggregated to *open-read-close*. Finally, varmail issues a combination of file create, write, sync, and close operations and open file, read, and close file operations, which are aggregated to *open-write-close* and *open-read-close*, respectively. The I/O sequences are repeated on thousands of files by 16 worker threads. The workloads are metadata-heavy issuing file create, delete, directory update operations that contribute to 69%, 63%, and 64% of the overall I/O in the varmail, the fileserver, and the webserver workloads, respectively. FusionFS and CrossFS outperform other file systems by eliminating system calls, reducing data movement, communication, and software costs such as queuing delays, delivering 36% gains for webserver workload over ext4-DAX. Furthermore, despite NOVA's multicore parallelism friendly design, reducing I/O overheads is



(a) Compute-based Scheduling **(b)** Memory-based Scheduling **(c)** Comparison with CrossFS Scheduler

Figure 9: Scheduler. (a) shows the normalized throughput (THP) for each workload relative to no-sharing of device-CPU resources. (b) shows the throughput factor relative to the no-sharing of device-memory resources. The device memory budget is set to 2GB. (c) compares CPU scheduling against CrossFS.

critical, as showcased by *FusionFS-slow-device-cpu* gains.

5.2.3 Macrobenchmark - File Encryption

We next study FusionFS on widely used Linux encryption and decryption service, Cryptsetup [2]), which is used by several applications (e.g., OpenSSL [8]). Cryptsetup uses widely-used AES-128 [39] symmetric block cipher to encrypt files. The application threads read data from a 4GB file, encrypt, and write back the encrypted data. In Figure 8b's x-axis, we vary the application's thread count. For FusionFS, we replace the read, encrypt, write sequence with *read-encrypt-write* CISC_{Ops}. FusionFS, by aggregating operations, reduces I/O cost and outperforms its counterparts achieving up to 2.48X gains over NOVA.

5.3 Crash Consistency and Recovery

We next study FusionFS' crash consistency and recovery capabilities. We use (1) a *append-CRC-write* workload and (2) a *vector-write* (ten writes) benchmark. For both, we inject failures at different points to test durability, as shown in Table 10a. For the *append-CRC-write*, we add three failure states (*F1*, *F2*, and *F3*), whereas for the vectored write, we inject a failure before any writes (*F4*), a failure between writes 4 and 5 (*F5*), and a failure after all the writes complete (*F6*). In Table 10b, we show crash-consistency correctness of committed and uncommitted operations for different file systems. Note that FusionFS offers the basic all-or-nothing (*MacroTx*), and an optional (and optimistic) auto-recovery, *AutoRec* that uses *MicroTx* and operational log for fast recovery.

First, as shown in Table 10a, for the *append-CRC-write*, for the case when the failure happens before *append* commits (*F1*), ext4-DAX, NOVA, SplitFS, CrossFS, and FusionFS's MacroTx provide crash consistency excluding the uncommitted append (C). In contrast, FusionFS's AutoRec provides operational logging; hence during restart, it can recover and re-execute *append-CRC-write* to completion if a valid operational log entry exists, as indicated by the (C/R) state. Similarly, for a failure after checksum (*F2*), AutoRec recovers both append and checksum's state from MicroTx, and finishes writing, providing better recoverability after failure. Next, *F4-F6* shows the failure points for the vectored 4KB write workload. When a failure occurs at *F5* (partial writes of a vector), all file system approaches, excluding FusionFS' AutoRec, do not recover due to their all-or-nothing approach restoring the file system to a consistent state. In contrast, AutoRec uses MicroTx with fine-grained commits, recovers partially completed writes in a vector (C/R), and finishes the vectored write.

Recovery Time. To study the impact on recovery time, in Table 10c, we run the *append-CRC-write* workload with 16 threads that issue 16MB appends and inject failures at crash points *F1* (before append) and *F2* (after checksum). For ext4-DAX without data atomicity, applications must re-execute the entire operation sequence and incur system calls and data movement costs, also increasing recovery time. In contrast, NOVA and SplitFS provide atomic appends. We assume applications when using NOVA file system keep a record of appends and only re-execute checksum and write operations during the restart. This reduces data movement and system call costs. Next, MacroTx must re-execute the entire sequence, but offloading as CISC_{Ops} reduces cost. Finally, AutoRec uses MicroTx to automatically recover state at *F1* and *F2* and uses the operational log to re-execute and complete the CISC_{Op} without application interaction. Consequently, this provides up to 2.65X gains over ext4-DAX.

Latency Impact. To understand the performance impact of AutoRec, in Table 10d, we compare the average latency of *append-CRC-write* and *vector-write*. First, MacroTx reduces the average latency of all operations, including a substantial reduction for vectored writes by reducing I/O overheads. Next, MicroTx provides fine-grained durability (i.e., commit for each operation of a CISC_{Op}), but the latency increase over MacroTx is negligible because it reuses the same journal blocks, reducing the block allocation cost. In contrast, the optional AutoRec's operational logging with request commands and input, marginally increases the latency.

5.4 Device Compute and Memory Fairness

We next evaluate the effectiveness of FusionFS CFS-based scheduling in providing storage compute and memory fairness across tenants using workloads that are bottlenecked by (a) device-CPU and (b) device-RAM. We consider I/O intensive random *write-only* benchmark, compute-intensive *append-CRC-write*, and compute + memory-intensive *read-compress-write* workloads. We compare FusionFS CFS schedulers against round-robin I/O scheduler (*RR-scheduler*) as proposed in recent studies [44].

Device Compute + I/O Scheduling We first analyze the effectiveness of FusionFS's compute-centric CFS by co-running *append-CRC-write* with I/O-intensive *write-only* workload performing 4KB writes. In Figure 9a, the x-axis

$CISC_{Ops}$	Crash Condition
append-CRC-write	Before append completes (**F1**), after checksum calculation (**F2**), after checksum write (**F3**)
vector-write (10 writes)	before first write completes (**F4**), between writes 4 and 5 (**F5**), after all writes complete (**F6**)

(a) $CISC_{Ops}$ Failure (F) Condition.

Systems	Crash No.					
	F1	**F2**	**F3**	**F4**	**F5**	**F6**
ext4-DAX, NOVA, SplitFS, CrossFS	*C*	*C*	*C/R*	*C*	*C*	*C/R*
FusionFS-MacroTx	*C*	*C*	*C/R*	*C*	*C*	*C/R*
FusionFS-AutoRec	*C/R*	*C/R*	*C/R*	*C/R*	*C/R*	*C/R*

(b) Consistency (C) and Recovery (R) after crash. **C** and **R** denotes successful crash-consistency and recovery after failure.

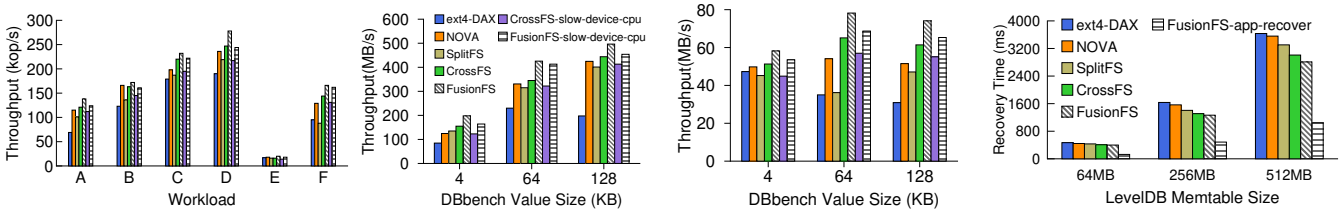
No.	ext4-DAX	NOVA	SplitFS	CrossFS	MacroTx	AutoRec
F1	75.5	55.6	23.9	43.7	30.4	28.5
F2	74.3	55.1	24.6	41.3	29.4	8.3

(c) Recovery time (ms) *append-CRC-write* running 16 threads with 16MB I/O size.

Operation	ext4-DAX	NOVA	SplitFS	CrossFS	MacroTx	AutoRec
append-CRC-write	18.4	17.3	16.5	16.9	15.6	23.4
vector-write	44.6	41.1	35.3	39.1	29.2	46.6

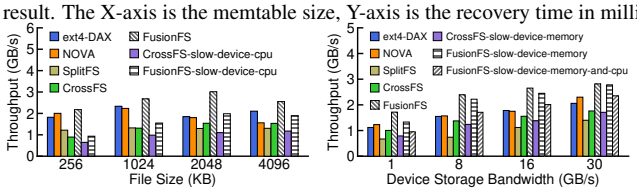
(d) Average latency (μs) for each $CISC_{Op}$.

Figure 10: Crash consistency and Fast Recovery.



(a) *YCSB* benchmark **(b)** Random Reads **(c)** Random Writes **(d)** Recovery benchmark.

Figure 11: LevelDB Evaluation. (a) shows *YCSB* benchmark result, (b) and (c) show *db_bench* benchmark results, (d) shows LevelDB Recovery benchmark result. The X-axis is the memtable size, Y-axis is the recovery time in milliseconds.



(a) Snappy Throughput **(b)** Sensitivity to Device Bandwidth

Figure 12: Snappy Compression Throughput. (a) varies file size and use 16 threads and 100K files. (b) varies storage bandwidth and use 16 threads, 100K files and 2MB each file.

varies application thread count for all workloads, the y-axis shows normalized throughput, and errors lines on the bars show max and min throughput variation across threads. The application threads operate on separate files. Further, we also compare the performance against the state-of-the-art CrossFS with round-robin scheduling in Figure 9c.

Observations. First, when using the baseline round-robin *RR-scheduler*, StorageFS picks a request from either the inode-queues of the *append-CRC-write* or the *write-only* workloads and dispatches them for processing. When using RR-scheduler, *append-CRC-write-RR* with higher compute needs could unfairly delay or starve I/O-intensive *write-only-RR* requiring short bursts of CPU for executing the file system logic, journals, and checkpointing; this impacts the throughput. In contrast, FusionFS' CFS scheduler accounts for both workloads' virtual device CPU usage time, equally prioritizes the *write-only* and the *append-CRC-write* workloads, consequently improving the throughput of *write-only-CFScpu* workload by 1.34x. The throughput of *append-CRC-write-CFScpu* reduces marginally. Finally, our CFS scheduler achieves higher gains over CrossFS for 8 and 16 thread configurations.

Memory-based Scheduling To understand the effectiveness of device memory-centric scheduling, we study capacity-intensive *read-compress-write* (shown as Compress-Write) co-scheduled with *append-CRC-write* workloads in Figure 9b. We limit the device-RAM budget for in-storage processing to 2*GB*. We compare *append-CRC-write-CFScpu* and *compress-write-CFScpu* against *append-CRC-write-CFSmm*, and *compress-write-CFSmm* scheduler that treats memory capacity as a first-class citizen towards $CISC_{Ops}$ scheduling.

Observations. The CFS CPU scheduler lacks memory capacity awareness. Consequently, the memory-intensive compression workload often stalls *append-CRC-write* despite the availability of device-CPUs. Moreover, stalling leads to side-effects such as frequent polling to check for free device memory availability. In contrast, FusionFS's CFS enables fairness based on each workload's memory usage, thereby equally prioritizing *append-CRC-write* and *read-compress-write* workloads. As shown, *append-CRC-write-CFSmm*'s throughput improves by 1.76x, whereas *compress-write-CFSmm*'s throughput only reduces by 15%.

5.5 Real-World Applications

We next study the benefits of FusionFS for real-world applications, LevelDB [7] and Snappy Compression [19]. Beyond performance, we also explore recovery and restart performance through simple changes to LevelDB's recovery and restart code.

For LevelDB, we modify the *append->checksum->write* sequence designed to avoid frequent commits (*fsync*) for SST files and WAL and replace them with *append-CRC-write* $CISC_{Ops}$. Similarly, we offload read operation using *read-checksum*. We evaluate the random write workload in Figure 11c and the random read workload in Figure 11b using the widely-used *db_bench* for 1 million key-value pairs and 16 application threads. The value size is varied from 4KB to 128KB. Note that recent LevelDB versions use 64K internal buffer for smaller appends. Further, we also evaluate YCSB [18] cloud benchmark using workloads A-F with varying read/write ratios issued with Zipfian distribution [31].

Observations. First, FusionFS provides considerable gains for both random write and read workloads. For random writes, smaller appends in *append-CRC-write* (e.g., 4K) are buffered, resulting in one system call instead of two calls for the larger values (64K). FusionFS gains stem from a combination of reduced system call, data movement, and communication costs. Offloading CRC to the device helps in the better utilization of host-CPUs for other work across application threads. We also observe that, in contrast to CrossFS, FusionFS CFS is more effective in multiplexing 4 device cores compared to CrossFS, which uses linked lists to schedule requests. For random reads, beyond offloading CRC to the device-CPUs and efficient scheduling, FusionFS reads only data without CRC bytes. In summary, FusionFS achieves gains between 1.23x-2.23x and 1.81x- 2.51x, respectively, over ext4-DAX.

Next, YCSB uses Zipfian access pattern, and therefore, the application-level caching is beneficial for all approaches. Despite caching, FusionFS provides high gains for write-intensive C, D, and F workloads. Furthermore, we believe adding more $CISC_{Ops}$ to other parts of the application (e.g., SST compaction) [14] would further increase the gains.

Restarts using Application-Customized $CISC_{Ops}$. We next study the benefits of application-customized $CISC_{Ops}$ in reducing restart cost using LevelDB. We observe that LSMs such as LevelDB [5], and others (e.g., Redis [10]), persist in-memory state to a write-ahead log (WAL) before updating the data file (e.g., SST files). LevelDB reads and checks the integrity of key-value pairs during restarts using the checksum and then sorts and writes them to disk files (SST files). All of these operations consume high I/O costs and data movement. Notably, the restart cost increases with the memory buffer (i.e., memtable size) and the WAL size.

Observations. Figure 11d shows the recovery cost for all prior systems, FusionFS (without application-customized restart), and FusionFS-app-recover with customized recovery. FusionFS offloads just the *read-checksum* achieving up to 1.17x faster restarts. In contrast, for *FusionFS-app-recover*, we reduce restart costs by enabling developers to construct and offload a custom *read-checksum-sort-write* $CISC_{Ops}$. For each key-value pair, the offloaded operation validates checksum, sorts them using a RB-tree in StorageFS, and writes them directly to the SST file. This results in up to 2.69x and 3.58x faster recovery over FusionFS and ext4-DAX, respectively.

Snappy Compression. Next, we evaluate FusionFS on the widely-used snappy file compression [19]. Figure 12a shows results for compressing 100K files using 16 threads. We vary the file sizes along the x-axis, and the y-axis shows the throughput in terms of bytes compressed. For FusionFS, we add a *open-read-compress-write* $CISC_{Ops}$.

Observations. First, we observe that NOVA performs well with its multi-core friendly structures and log-structured file system. Second, SplitFS avoids system calls but suffers from high kernel activity and pre-paging cost for staged mmap() files when opening 100K files. Third, CrossFS lacks $CISC_{Ops}$ and incurs higher overhead from creating file descriptor queues and data movement. In contrast, FusionFS avoids data copy overheads by aggregating I/O operations and offloading compression to device-CPUs resulting in 1.63x and 1.67x gains over ext4DAX and NOVA, respectively. However, for large 4MB files, the high compression cost dominates I/O costs, thereby reducing throughput for all cases.

Sensitivity to Device-CPU Speeds. We evaluate LevelDB (in Figure 11) and snappy compression (in Figure 12a) using FusionFS and CrossFS that use slower device-CPU (*slow-device-cpu*). Despite using slower CPUs, FusionFS gains significantly over other designs, specifically for I/O-intensive workloads. For example, LevelDB's *FusionFS-slow-device-cpu* provides 1.79x gains over ext4-DAX.

Sensitivity to Storage and Device Memory Bandwidth. Finally, in Figure 12b, we study the impact of device-RAM and storage bandwidth for memory and I/O-intensive snappy compression by throttling memory on a dual-socket CloudLab machine that uses DRAM for storage. In the x-axis, we vary the storage bandwidth from 1GB/s to 30GB/s, and the values 8GB and 16GB emulate the bandwidth of PCIe Gen4 and Gen5 SSDs [9]. Further, for CrossFS and FusionFS, we also study the impact of memory bandwidth (*CrossFS-slow-device-memory* and *FusionFS-slow-device-memory*) and the impact of slow CPU and memory bandwidth (*FusionFS-slow-device-memory-and-cpu*).

Observations. For all storage bandwidths, FusionFS provides considerable gains. For extremely low device-memory bandwidth (say, 1GB), FusionFS throughput is similar to KernFS and UserFS that use faster host DRAM. However, real devices are expected to have higher bandwidth (8GB and above) [11, 41], for which FusionFS shows considerable gains.

6 Conclusion

We designed and evaluated FusionFS, an in-storage firmware design that combines file system and data processing capabilities in modern CSDs. To reduce the impact of system calls, data copy, and other software and hardware overheads, we introduced $CISC_{Ops}$, which fuses multiple I/O and compute operations and offloads them to storage. Evaluation of FusionFS with several benchmarks and applications show significant performance benefits.

Acknowledgements

We thank Joo-young Hwang (our shepherd) for insightful comments to improve the quality of this paper. We also thank anonymous reviewers and the members of RSRL for their valuable feedback. We are grateful to Rutgers Panic Lab for the infrastructure support. This research was supported by funding from NSF grant CNS-1910593. This work was partially carried out on the experimental platform funded by NSF grant CNS-1730043.

References

[1] Completely Fair Scheduler. https://www.kernel.org/doc/html/latest/scheduler/sched-design-CFS.html.

[2] Cryptsetup. https://linux.die.net/man/8/cryptsetup.

[3] Facebook RocksDB. http://rocksdb.org/.

[4] Filesystem sandboxing with eBPF. https://lwn.net/Articles/803890/.

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

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

[7] LevelDB Source Code. https://github.com/google/leveldb.

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

[9] PCI Express. https://en.wikipedia.org/wiki/PCI_Express.

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

[11] ARM. https://www.arm.com/solutions/storage/computational-storage.

[12] 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, page 69–86, New York, NY, USA, 2017. Association for Computing Machinery.

[13] Tim Bisson, Ke Chen, Changho Choi, Vijay Balakrishnan, and Yang-suk Kee. Crail-kv: A high-performance distributed key-value store leveraging native kv-ssds over nvme-of. In *2018 IEEE 37th International Performance Computing and Communications Conference (IPCCC)*, pages 1–8, 2018.

[14] Wei Cao, Yang Liu, Zhushi Cheng, Ning Zheng, Wei Li, Wenjie Wu, Linqiang Ouyang, Peng Wang, Yijing Wang, Ray Kuan, Zhenjun Liu, Feng Zhu, and Tong Zhang. POLARDB Meets Computational Storage: Efficiently Support Analytical Workloads in Cloud-Native Relational Database. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 29–41, Santa Clara, CA, February 2020. USENIX Association.

[15] Ming Chen, Dean Hildebrand, Henry Nelson, Jasmit Saluja, Ashok Subramony, and Erez Zadok. vNFS: Maximizing NFS performance with compounds and vectorized I/O. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST)*, pages 301–314, Santa Clara, CA, February-March 2017. USENIX Association.

[16] Tae-Sun Chung, Dong-Joo Park, Sangwon Park, Dong-Ho Lee, Sang-Won Lee, and Ha-Joo Song. A survey of flash translation layer. *J. Syst. Archit.*, 55(5-6):332–343, May 2009.

[17] Cloudlab. https://docs.cloudlab.us/cloudlab-manual.html.

[18] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, 2010.

[19] Jeff Dean, Sanjay Ghemawat, and Steinar H. Gunderson. Snappy Compession. https://github.com/google/snappy.

[20] Jaeyoung Do, Victor C. Ferreira, Hossein Bobarshad, Mahdi Torabzadehkashi, Siavash Rezaei, Ali Heydarigorji, Diego Souza, Brunno F. Goldstein, Leandro Santiago, Min Soo Kim, Priscila M. V. Lima, Felipe M. G. França, and Vladimir Alves. Cost-effective, energy-efficient, and scalable storage computing for large-scale ai applications. *ACM Trans. Storage*, 16(4), October 2020.

[21] Jaeyoung Do, Yang-Suk Kee, Jignesh M. Patel, Chanik Park, Kwanghyun Park, and David J. DeWitt. Query Processing on Smart SSDs: Opportunities and Challenges. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 1221–1230, New York, NY, USA, 2013. 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, page 478–493, New York, NY, USA, 2019. Association for Computing Machinery.

[23] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, Boston, MA, 2011.

[24] HewlettPackard Quartz. https://github.com/HewlettPackard/quartz.

[25] Junsu Im, Jinwook Bae, Chanwoo Chung, Arvind, and Sungjin Lee. PinK: High-speed In-storage Key-value Store with Bounded Tails. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 173–187. USENIX Association, July 2020.

[26] Insoon Jo, Duck-Ho Bae, Andre S. Yoon, Jeong-Uk Kang, Sangyeun Cho, Daniel D. G. Lee, and Jaeheon Jeong. YourSQL: A High-Performance Database System Leveraging in-Storage Computing. *Proc. VLDB Endow.*, 9(12):924–935, August 2016.

[27] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. SplitFS: Reducing Software Overhead in File Systems for Persistent Memory. SOSP '19: Symposium on Operating Systems Principles, New York, NY, USA, 2019. ACM.

[28] 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, page 494–508, New York, NY, USA, 2019. Association for Computing Machinery.

[29] Yangwook Kang, Rekha Pitchumani, Pratik Mishra, Yang-suk Kee, Francisco Londono, Sangyoon Oh, Jongyeol Lee, and Daniel D. G. Lee. Towards building a high-performance, scale-in key-value storage system. In *Proceedings of the 12th ACM International Conference on Systems and Storage*, SYSTOR '19, page 144–154, New York, NY, USA, 2019. Association for Computing Machinery.

[30] 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.

[31] 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.

[32] Sudarsun Kannan, Ada Gavrilovska, Vishal Gupta, and Karsten Schwan. Heteroos: Os design for heterogeneous memory management in datacenter. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pages 521–534, New York, NY, USA, 2017. ACM.

[33] Jesung Kim, Jong Min Kim, S. H. Noh, Sang Lyul Min, and Yookun Cho. A Space-Efficient Flash Translation Layer for CompactFlash Systems. *IEEE Transactions on Consumer Electronics*, 48(2):366–375, May 2002.

[34] Jinhyung Koo, Junsu Im, Jooyoung Song, Juhyung Park, Eunji Lee, Bryan S. Kim, and Sungjin Lee. Modernizing file system through in-storage indexing. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 75–92. USENIX Association, July 2021.

[35] 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.

[36] Chyuan Shiun Lin, Diane CP Smith, and John Miles Smith. The Design of a Rotating Associative Memory for Relational Database Applications. *ACM Transactions on Database Systems (TODS)*, 1(1):53–65, 1976.

[37] Jing Liu, Andrea C Arpaci-Dusseau, Remzi H Arpaci-Dusseau, and Sudarsun Kannan. File Systems as Processes. In *11th {USENIX} Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*, 2019.

[38] Jing Liu, Anthony Rebello, Yifan Dai, Chenhao Ye, Sudarsun Kannan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Scale and Performance in a Filesystem Semi-Microkernel. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 819–835, New York, NY, USA, 2021. Association for Computing Machinery.

[39] Frederic P. Miller, Agnes F. Vandome, and John McBrewster. *Advanced Encryption Standard*. Alpha Press, 2009.

[40] 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, page 327–341, New York, NY, USA, 2018. Association for Computing Machinery.

[41] NVIDIA Mellanox BlueField DPU. https://www.mellanox.com/files/doc-2020/pb-bluefield-smart-nic.pdf.

[42] Simon Peter, Thomas Anderson, and Timothy Roscoe. Arrakis: The operating system as control plane. In *Proc. 11th USENIX Conf. Oper. Syst. Des. Implement*, volume 38, pages 44–47, 2013.

[43] Moinuddin K. Qureshi and Gabe H. Loh. Fundamental latency trade-off in architecting dram caches: Outperforming impractical sram-tags with a simple and practical design. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, pages 235–246, Washington, DC, USA, 2012. IEEE Computer Society.

[44] Yujie Ren, Changwoo Min, and Sudarsun Kannan. Crossfs: A cross-layered direct-access file system. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 137–154. USENIX Association, November 2020.

[45] Yujie Ren, Jian Zhang, and Sudarsun Kannan. CompoundFS: Compounding I/O Operations in Firmware File Systems. In *12th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 20)*. USENIX Association, July 2020.

[46] Erik Riedel, Garth A. Gibson, and Christos Faloutsos. Active Storage for Large-Scale Data Mining and Multimedia. In *Proceedings of the 24rd International Conference on Very Large Data Bases*, VLDB '98, pages 62–73, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc.

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

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

[49] Samsung. Samsung Key Value SSD. https://www.samsung.com/semiconductor/global.semi.staticSamsung_Key_Value_SSD_enables_High_Performance_Scaling-0.pdf.

[50] Samsung. Samsung NVMe SSD 960 Data Sheet. http://www.samsung.com/semiconductor/minisite/ssd/downloads/document/Samsung_SSD_960_PRO_Data_Sheet_Rev_1_1.pdf.

[51] Seagate RISC-V storage solution. https://www.seagate.com/innovation/risc-v/.

[52] SNIA. SNIA Computational Storage Technical Work Group (TWG).

[53] StorageReview.com. Firmware Upgrade. http://www.storagereview.com/how_upgrade_ssd_firmware.

[54] Stanley Y. W. Su and G. Jack Lipovski. CASSM: A Cellular System for Very Large Data Bases. In *Proceedings of the 1st International Conference on Very Large Data Bases*, VLDB '75, Framingham, Massachusetts, 1975.

[55] Wei Su, Akshay Auror, Ming Chen, and Erez Zadok. Supporting transactions for bulk NFSv4 compounds. In *Proceedings of the 13th ACM International Systems and Storage Conference (SYSTOR '20)*, Haifa, Israel, June 2020. ACM.

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

[57] 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.

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

[59] NVM Express Workgroup. NVMExpress Specification. https://nvmexpress.org/resources/specifications/.

[60] 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. ACM.

[61] 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, 2016.

[62] 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.