CompoundFS: Compounding I/O Operations in Firmware File Systems

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

Abstract

We introduce CompoundFS, a firmware-level file system that combines multiple filesystem I/O operations into a single compound operation to reduce software overheads. The overheads include frequent interaction (e.g., system calls), data copy, and the VFS overheads between user-level application and the storage stack. Further, to exploit the compute capability of modern storage, CompoundFS also provides a capability to offload simple I/O data processing operations to the device-level CPUs, which further provides an opportunity to reduce interaction with the filesystem, move data, and free-up host CPU for other operations. Preliminary evaluation of CompoundFS against the state-of-the-art user-level, kernel-level, and firmware-level file systems using microbenchmarks and a real-world application shows up to 178% and 75% performance gains, respectively.

1 Introduction

With the advent of ultra-fast storage technologies such as NVMe SSD and 3D-Xpoint, the software bottlenecks are slowly dominating the hardware cost [2, 4, 17, 35]. To reduce the software overheads, there has been a renewed interest across industry and academia, developing solutions that aim at thinning the software storage stack. Most solutions (software and firmware) aim to reduce OS interaction across the data and control plane without compromising correctness, consistency, crash-consistency, or security guarantees [12,21,25,31,32,36].

Prior software-level solutions range from application-customized storage stack to reduce generic filesystem overheads [7], full user-level filesystems [12,25,36], and hybrid user and kernel-level filesystems [20,23]. Hardware solutions include firmware file systems [21], key-value stores (e.g., KV-SSDs) [32], and compute accelerated (e.g., FPGA) in-device storage [31] allowing applications to bypass the kernel for direct-access. Several nonvolatile memory (NVM) specific hybrid user and kernel-level filesystems such as ZoFS [12], FLEX [36], and SplitFS [20] for memory-based storage have been proposed.

Limitations of Current Work. While current proposals reduce OS interaction and system call costs, they either do not eliminate major software bottlenecks or fully exploit the compute capability of modern storage hardware. The software bottlenecks include: (1) frequently crossing application and storage stack boundaries, and (2) high data movement overhead between application and filesystem. The boundary-crossings include application core switching to the file system running as a separate server process [12,23,25] or inside the kernel [20,23,28], or in the firmware [21]. Some user-level filesystems must trap into the OS for concurrent filesystem access across applications [23, 28]. Finally, most prior approaches cannot exploit storage-level compute capability, or require applications to be fully redesigned [31].

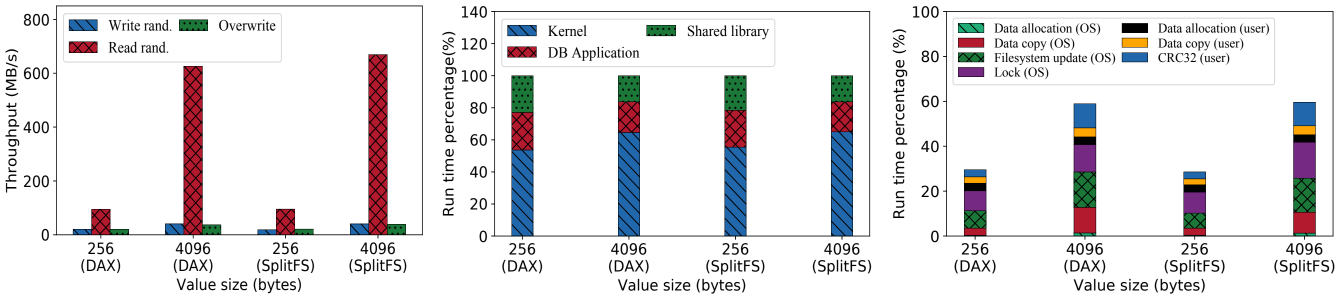
Contribution. To overcome these challenges, we design **CompoundFS**, a direct-access firmware-level filesystem that reduces interaction and data copy overheads between applications and the filesystem and utilizes the device-level compute capabilities. For direct-access, CompoundFS adopts the design principle of prior firmware filesystems [21]. To reduce overheads, CompoundFS introduces Compound Operations (hereafter referred to as **CompoundOps**). CompoundOps combine multiple POSIX-style I/O operations into a single *enhanced* I/O operation (e.g., file open-and-read, read-modify-write). The CompoundOps are issued by the application and executed as one operation inside storage firmware. Consequently, CompoundOps reduce interaction and data copy overheads between application and filesystem. Further, in CompoundFS, we extend CompoundOps with simple device-level compute operations such as checksum and compression that reduce host CPU use and also avoid repeated data movement (e.g., write-and-checksum). Incorporating CompoundOps requires simple and intuitive changes to POSIX-based applications.

CompoundFS currently supports a simple scheduling and all-or-nothing model for crash-consistency (all operations in a CompoundOps either succeed or fail). Further, CompoundFS support for CompoundOps is currently limited to a single inode. Our ongoing work is addressing challenges for ordering CompoundOps across application(s) threads and crash-consistency, a better I/O scheduling mechanisms for efficient management of device-CPUs (discussed in Section 3), and support for multi-inode CompoundOps.

Preliminary evaluation of CompoundFS with CompoundOps on an emulated infrastructure using Intel DC Optane memory shows substantial reduction in application and file system interaction, system call cost, data overheads, and device-level computation benefits, all leading to performance gains of 178% and 75% in microbenchmarks and real-world LevelDB application, respectively.

2 Background and Motivation

Next, we present a brief background on direct-access filesystem approaches and other efforts to offload computation to storage. We then discuss the limitations and provide empirical evidence of these limitations.



(a) Throughput vs. Value size

(b) Runtime Breakdown

(c) Function Cost Split (Value Size)

Figure 1: OS Overhead Analysis on LevelDB. Figures show (a) throughput (compression disabled), (b) user-level vs. OS time breakdown (in %), and (c) time consumption of dominant user-level and kernel functions.

2.1 Direct-access Filesystem (DirectFS).

Several DirectFS systems have been proposed to reduce system call overheads, provide partial or full crash-consistency guarantees, and POSIX compatibility. For example, state-of-the-art approaches such as Strata [23], and more recently SplitFS [20], divide the filesystem across the userspace and the OS. Approaches such as Moneta-D [9] and Arrakis [28] split filesystem across the library, kernel, and firmware. User-level approaches like [12, 25] deploy a microkernel-like trusted server reducing OS interaction. An alternative approach to achieving direct-access is by deploying a filesystem inside storage firmware as used by prior hardware-centric approaches such as DevFS [21] and Insider [31]. Apart from providing direct access, firmware filesystems can exploit hardware capabilities such as device-level CPUs, power-loss fail-safe capacitors, device-level I/O queues for parallelism, and others [21].

2.2 Storage-level Computation

Exploiting storage-level compute capability has been explored for the past four decades. Seminal systems such as CASSM [33], RARES [24], and Active-Storage [6, 30] proposed adding one or more processors to a disk for operations such as database scan and search. Recent systems such as Smart-SSD [11], BlueDBM [19], and Samsung's KV-SSD [32] deploy query processing engine, big data applications, and key-value store engines inside SSDs. Alternatively, the use of low-power FPGAs to accelerate storage performance is gaining traction. LSM-FPGA [37] offloads LSM store's compaction engine to the storage. Quero et al. offload sorting operations to SSD to improve application performance and SSD lifetime [29], whereas Biscuit allows developers to write custom applications for processing inside a raw storage device. [14]. To accelerate persistent key-value stores, POLARDB [8] offloads and distributes table scan tasks from host CPU to an FPGA-centric smart storage device. Caribou [16] explores the design of near-data processing that supports key-value engines. YourSQL [18] filters the data by offloading data scanning of a query to user-programmable solid-state drives.

2.3 Limitations of Prior Systems.

DirectFS: While DirectFS reduce system call costs (between application and OS), these approaches are limited by one of the following. First, prior approaches do not necessarily reduce boundary-crossings between application address space and the storage stack. The storage stack could be running as a separate server process [12, 23, 25], or inside the kernel [20, 23, 28] for control plane operations (e.g., metadata update), or even inside the device firmware [21]. Second, data movement between application and storage stack is not effectively reduced.

Storage-level Compute: Most prior storage-level compute research has been designed purely for data processing customized to an application without using filesystems. They lack crash-consistency capabilities, support for POSIX, or require OS interaction [14, 29]. Approaches like DevFS [21] lack storage-level compute capability and may even require a full redesign of the storage stack [31]. In contrast, **CompoundFS**, in addition to supporting filesystems, provides a generic, simple extension to POSIX for exploiting to storage-level compute. Further, CompoundFS compounds I/O operations and reduces data movement cost, substantially reducing latency and improving throughput even with simple device-CPUs. Using accelerators for scaling specific operations (e.g., scan) in the above state-of-the-art-systems [8, 16, 18] could possibly improve the performance of CompoundFS further.

2.4 Analysis

To understand the software overheads, in Figure 1, we briefly analyze widely-used LevelDB, which is a persistent key-value store application. We study the throughput (Figure 1a), the runtime breakdown (Figure 1b), and the cost of dominant operations across the user and kernel-levels (Figure 1c). For our study, we use the popular *db_bench* [3] benchmark, and evaluate random write, random read, and overwrite operations. We use 4-client threads and vary the value sizes from 256B to 4KB. The key size (16 bytes) and the number of key-value pairs (100K) are kept constant. We use a 512GB DC Optane persistent memory with 64 cores and 32 GB DRAM to analyze state-of-the-art ext4-DAX (a kernel filesystem) and SplitFS [20] (a hybrid user- and kernel-level filesystem).

First, as shown in Figure 1a, the write throughput is significantly lower than the read throughput because write suffers from high compaction cost, a behavior well studied in the past [22]. Next, as shown in Figure 1b, LevelDB suffers significant kernel-level overhead, spending close to 65% of the runtime when using 4K value size. To understand the overheads inside the OS, Figure 1c shows the breakdown of time-consuming user- and kernel-level functionalities. As shown in Figure 1c, the data copy overheads (between the user and OS buffers) and across data structures inside the OS consume 9% of the time. In contrast, filesystem metadata updates and locking consume 15% and 16%, respectively. Surprisingly, these overheads are high in SplitFS (implemented over ext4-DAX), which converts data plane operations to NVM load and store operations.

We observe that LevelDB performs several metadata-heavy operations, such as file creation, rename, close, and sync operations, thereby increasing user-to-kernel data movement. We also notice that SplitFS suffers from high kernel-level locking and kernel-level allocation overheads due to its use of pre-paging. In the user-level, the checksum (CRC) overheads followed by data copy costs are high. LevelDB (and several other applications) use CRC for application-level crash-consistency during logging and compaction to avoid frequent *fsync* for each key by first writing the payload and then the CRC of the payload.

Based on these observations, we posit the following: (1) system call and data copy overheads between application and storage stack in both control and data plane impacts performance and must be reduced; (2) applications spend a significant time pre- or post-processing I/O data (e.g., CRC) before read and write operation that could be offloaded to a storage device with computation capability.

3 Design of CompoundFS

We present the general architecture of CompoundFS, a firmware filesystem to reduce system call and data copy overheads by combining multiple operations and creating a compound operation. Additionally, CompoundFS can also offload I/O data pre- and post-processing to storage hardware freeing up the host CPUs. We also discuss the crash-consistency and scheduling challenges.

Figure 2 shows the high-level design of CompoundFS, which consists of a user-level library (UserLib) and a firmware-level filesystem component (StorageFS). For traditional use (without CompoundOps), CompoundFS allows unmodified POSIX-based applications to benefit from direct storage access similar to prior filesystems such as DevFS. CompoundFS adds simple extensions to traditional POSIX APIs to (1) combine two or more POSIX operations into one CompoundFS operation, and (2) offload some pre- and post-processing computations to StorageFS. We first provide a brief overview of UserLib and StorageFS and then discuss our current design for combining multiple operations and offloading computation.

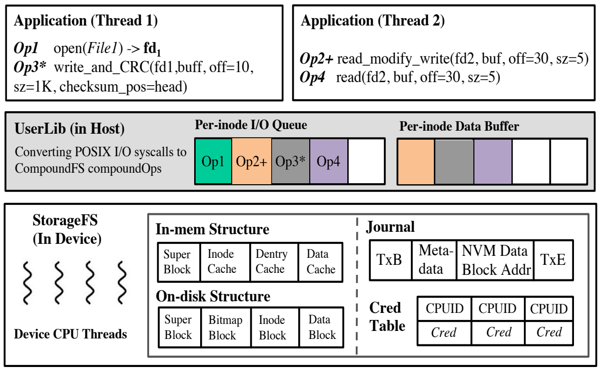


Figure 2: CompoundFS High-level Design. The filesystem data structure is partitioned into global and per-file structures. The per-file structures are created during file setup. CompoundFS metadata structures are similar to other kernel-level filesystems. *Op2+* shows a CompoundOps, *Op3** shows a CompoundOps with processing.

User-space Library (UserLib). UserLib intercepts POSIX as well as our extended CompoundOps and converts them to StorageFS understandable (NVMe-like) I/O commands [35]. To exploit the hardware-level parallelism in modern I/O devices that can support and process requests from 64K I/O queues, during file open, UserLib requests the OS for a DMA memory region for creating inode-queues, and registers them with StorageFS. For subsequent data plane operations, such as read, write, and fsync, UserLib adds these commands to the inode-queues and rings a doorbell, which is then processed by StorageFS.

Storage File System (StorageFS). The high-level design of StorageFS is similar to other firmware-level file systems [21]. StorageFS provides a simpler filesystem with in-memory and on-disk metadata structures such as super-block, bitmap blocks, inode, and data blocks. StorageFS also supports data and metadata journaling using a dedicated journal space on the device as shown in Figure 2. Our current design adapts and significantly extends the PMFS [13] filesystem.

PMFS is a kernel-level file system designed for NVMs to bypass the page cache. It relies on the VFS layer using traditional system calls that incur data movement between user and kernel space, and the host-CPUs for file system processing. While the page cache bypassing design aligns with the goal of CompoundFS, PMFS must be extended to avoid dependence on the VFS, system calls, and crash-consistency techniques that are tailored for StorageFS.

We modify PMFS to a command-based architecture. The I/O operations are packed as commands, and the commands encapsulate a command ID (e.g., read/write/append) and other related parameters of a command (e.g., I/O buffer, size). To eliminate system call and the VFS dependency, UserLib registers a shared circular buffer with StorageFS during an application's initialization from which StorageFS can directly process. One key low-level point is that UserLib uses a submission head to point to the next available entry in the command buffer. StorageFS maintains a completion head pointing to the current entry in the circular buffer under process; this is similar to the new IO_Uring [5] interface recently added to the Linux kernel. StorageFS fetches I/O commands from inode-queues, processes request by updating data and metadata updates in device memory, followed by on-disk journaling for crash-consistency, and finally, checkpointing them (see §4.2).

Emulation. Due to a lack of programmable storage, we emulate CompoundFS as a device driver with dedicated cores that use kernel threads to process requests. During the CompoundFS mounting, StorageFS finds the superblock, followed by the root directory. In addition, StorageFS also reserves a region of firmware memory for performing I/O. For security, CompoundFS uses a model similar to DevFS [21] by maintaining a host CPUID to credential mapping updated by the host OS and checked during each I/O operation.

Avoiding System Calls and Data Copy. For avoiding system calls, UserLib writes I/O commands and input/output buffer to the per-inode DMA buffer. StorageFS uses the DMA buffer to perform I/O operations directly, thereby avoiding system call overheads. However, in addition to system call overheads, reducing data copy overheads (moving data back-and-forth between the filesystem and user-space buffers) is critical for I/O bound applications. We next discuss the design of CompoundOps that combine multiple I/O operations into one compound operation.

4 Realizing CompoundFS Operations

We envision and currently support two forms of CompoundOps: (1) I/O-only CompoundOps that combine two or more traditional POSIX operations into one data-plane operation; (2) data pre- and post-processing CompoundOps. We first briefly discuss the mechanics of supporting CompoundOps followed by the details of currently supported operations.

4.1 Mechanics

Unlike vectored-I/O, CompoundOps could have one or more different POSIX (micro) operations that can be combined together and also support simple data pre- and post-processing. First, we start by extending the NVMe command structures. Current NVMe commands support simple block operations. Prior work such as DevFS extended the simple NVMe commands with POSIX-like filesystem operations (e.g., read, write, open, close). CompoundFS goes one step beyond to extend the NVMe commands to support multiple operations. We extend the *opcode* (operation code), *return code*, the I/O buffer pointers to support a list (using array) of operations, in addition to an additional field on the number of operations. In our current design, we restrict all the operations to a single inode, and our ongoing research is exploring the use of CompoundOps across different files.

4.1.1 Compound I/O Operations.

First, we aim to create simple CompoundOps that combine two or more traditional POSIX operations into a compound operation, thereby reducing interaction between applications and StorageFS. We compound operations based on two principles. The first principle involves combining operations that applications and developers use in pairs. Combining these operations can substantially reduce system call overheads. The second principle involves I/O operations that require simple data manipulation (e.g., compression). Offloading such operations to device-level CPU reduces data movement across host and device and the system call cost, also freeing up host-level CPUs.

We target operations that are generally used in pairs in applications; for example, open-write, open-read (open a file and read data blocks), open-write (open a file and write data blocks), write-close (write and close a file), and read-modify-write (discussed in this paper). Providing a simple API to programmers, one that does not require extensive changes to the application's logic or complex input argument is critical. Take the example of a read-modify-write operation that can be used for implementing overwrite operations (e.g., in LSM-based key-value stores). The read-modify-write is used by an application, which combines the arguments of a read and write operation as shown in Figure 2. Upon successful execution, the StorageFS returns the number of bytes written. To implement a read-modify-write, a combined NVMe command is added to the inode-queue buffer, and StorageFS is notified. StorageFS acquires an inode-level lock, iterates through the opcode array, and the corresponding input or output buffers, and the I/O size. Note that, before starting to execute a CompoundOps, CompoundFS must also check the permission of each operation. Our current simplistic design returns an error if the permission check of even one operation fails, and each device-CPU must complete the CompoundOps before switching to other operations.

4.1.2 Compound Operations with Processing.

To utilize the compute capability in modern storage devices (with 4 to 8 wimpy CPUs), we extend the CompoundOps to support simple data pre- and post-processing. Enabling storage-level compute not only reduces host CPU involvement but also reduces interaction between application and storage stack, and data transfer. For example, persistent key-value stores such as LevelDB, RocksDB, Redis, and several others, store data and the checksum when writing new updates to log and during compaction [26] mainly to avoid expensive (`fsync`) operations for each key-value pair. Unlike `fsync`, using checksums provides an optimistic application-level crash-consistency for user data by writing data and its checksum and not requiring an immediate page cache flush or enforcing strict data ordering [10]. In the case of system failure, the stored checksum and the newly computed checksum of persisted data is compared to identify data corruption, which is a critical part of the filesystem crash-consistency mechanism.

However, with checksums, the data (payload) and CRC is written as separate write operations [3], because: (1) a write may store fewer than the bytes it was issued for, in which case the checksums do not match during read, (2) the CRC is used as a commit for the preceding write, and (3) the CRC is written at different locations. CompoundFS, to reduce such overheads combines these operations into one operation. Currently, CompoundFS supports *write_and_checksum*, *read_and_checksum*, *compress_and_write*, and *compress_and_read* (used for reducing the storage space). Applications explicitly issue a *compress_and_write* using UserLib, which adds the request to the inode-queue and is processed by StorageFS similar to simple CompoundOps described earlier. CompoundFS returns an error or return code for each operation along with the return code list.

4.2 Atomicity and Crash-Consistency

Traditional OS-level file systems guarantee atomicity and crash-consistent durability properties. While file systems such as DevFS are designed to satisfy these properties for simple POSIX style operations, supporting them for CompoundOps introduces a new spectrum of challenges. We discuss the challenges and our initial design ideas, which we aim to realize in our on-going research.

First, regarding atomicity and consistency, because CompoundOps combines multiple POSIX-styled operations, a simple approach is to provide an all-or-nothing model where an entire CompoundOps is atomic. For example, one in-progress CompoundOps to an inode (i.e. a single directory or file) would stall all other operations to that same inode. Such atomicity could impact concurrency. More specifically, an application can hold an inode-level `rwlock` (i.e., a read-write semaphore), which would completely prevent any other updates to the directory. One approach that we are currently exploring is to allow individual operations to proceed in parallel, but requiring conflict resolution for the final commit. Other possible extensions include enforcing users to control CompoundOps atomicity with user-level locks.

Next, regarding crash-consistency, combining multiple POSIX operations with data pre- and post-processing introduces crash-consistency and ordering challenges. Regarding the crash-consistency, a CompoundOps could only succeed partially (e.g., a write operation could fail in *read_modify_write*). While one approach is to adopt an all-or-nothing model (as done currently in CompoundFS), reverting partially completed operations could be useful in some cases (e.g., *read* operation in a *read_modify_write*), whereas realizing partial reverts could be complicated.

Next, ordering threads performing CompoundOps (e.g., read-modify-write) and traditional POSIX operation (e.g., *write*) could be challenging, specifically in terms of performance. Should a thread (T_1) performing simple POSIX operation (e.g., *write*) ordered behind a thread (T_2) performing CompoundOps (*read-modify-write*), or could interpose $T_1's$ write with $T_2's$ *read_modify_write*? Our current implementation treats CompoundOps as one large operation with an inode-level lock. However, we are exploring ways to extend the journaling mechanism to support a flexible (partial) crash-consistency or redesign journaling for CompoundOps by lending ideas from prior work on databases to support composite transactions [15].

4.2.1 CompoundFS Scheduling Challenges

The number of I/O operations (simple operations as well as CompoundOps) would most likely exceed the number of available device-level CPUs, which demands efficient device-CPU scheduling and management. Further, introducing CompoundOps with data processing (e.g., CRC, compression) brings new challenges for avoiding starvation or workload imbalance across device-CPUs performing both simple and CompoundOps. Traditional block-level I/O schedulers (e.g., Linux *blk-queue*) are a misfit for CompoundOps operations because blk-queue does not consider computation. While our current implementation uses first-come-first-serve scheduling, we are exploring the benefits and implications of OS-level CPU fairness schedulers such as Linux CFS [1].

5 Evaluation

To understand the benefits and implications, we evaluate CompoundFS using micro-benchmarks and an application (LevelDB). Our evaluation answers the following questions:

(1) How effective are CompoundOps in improving the I/O performance by reducing the interaction and data copy overheads between applications and the storage stack?

(2) What is the performance impact of offloading compute to device-CPUs?

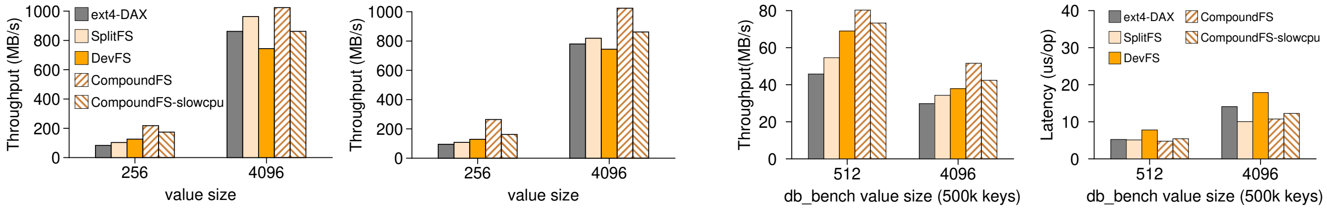
(3) How sensitive are CompoundFS benefits to device-CPU speeds?

5.1 Experimental Setup

We use a dual-socket, 64-core, 2.4GHz Intel(R) Xeon(R) Gold platform with 32GB DRAM, 512GB DC Optane persistent memory, and 1TB NVMe. To emulate StorageFS, we use persistent memory with four 128 GB NVM DIMMs, which can provide a maximum of 8GB/sec read and 3.8GB/sec write bandwidth [17]. To emulate PCIe latency, we add 900ns software delay [27] between the time a request is added to hosts inode-queue and marked ready with a doorbell for the device to process. We compare CompoundFS against kernel-level ext4-DAX [34], state-of-the-art hybrid user and kernel-level SplitFS [20], and firmware-level DevFS [21].

5.2 Microbenchmarks

To understand the preliminary benefits of CompoundFS with CompoundOps, we model a microbenchmark that performs *read-modify-write* and *write-and-checksum* operations



(a) Read Modify Write **(b)** Write and Checksum **(c)** LevelDB Random Writes **(d)** LevelDB Random Reads

Figure 3: CompoundFS Performance with Microbenchmark and LevelDB.

on a large 16GB file varying I/O size to 256-bytes and 4K-bytes. Figure 3a shows the throughput of *read-modify-write* benchmark that reads, modifies, and writes a block. We compare CompoundFS and CompoundFS-slowcpu (slowed-down device-CPUs running at 1.2GHz) against ext4-DAX, SplitFS, and DevFS. Figure 3b shows the throughput of *write-and-checksum* benchmark in the y-axis.

CompoundOps Performance. First, for the *read-modify-write* workload, ext4-DAX suffers from two system calls and the related data copy between the user library and the kernel. This impacts throughput considerably, especially for small I/O sizes. Next, SplitFS performs better than ext4-DAX by converting read and write operations to load and store instructions on staged memory-mapped files [20]. However, with SplitFS, we see a huge surge in the kernel activity with an increase in workload size. We suspect three primary causes: (1) increase in the user-level staged mmap files increase OS activity (for example, searching for free blocks); (2) pre-paging with MAP_POPULATE; (3) high filesystem internal data copy. Next, DevFS avoids system call costs to bypass the OS. However, the data copy problem between host and device (read and write) remains the same. Note that the host CPU waits for the device to complete the I/O. In contrast, CompoundFS avoids two system calls and converts two data copy operations into one, resulting in significant performance gains. Finally, despite reducing the device-CPU frequency to half of the host-CPU frequency, CompoundFS-slowcpu achieves up to 109% gains over ext4-DAX, 68% over SplitFS, and 38% over DevFS.

CompoundOps with data processing. We next model the payload (data) write followed by CRC write found in modern workloads, such as persistent key-value stores [3]. Note that data and CRC are written separately for correctness and durability, as discussed earlier. All approaches other than CompoundFS lack the capability to exploit storage hardware to process data (generating checksum), resulting in two system calls and data copy overheads in addition to performing checksum in the host. CompoundFS combines append and checksum to one operation, offloads the operation for StorageFS to process (and freeing up host CPUs), and overcoming the data copy cost. Consequently, CompoundFS improves performance over ext4-DAX, SplitFS, and DevFS by up to 178%, 144%, and 105%, respectively. Finally, CompoundFS-slowcpu reduces gains to 71%, 50%, and 25% over ext4-DAX, SplitFS, and DevFS, respectively.

5.3 Real-World Application

We next study the performance implications of CompoundFS using LevelDB [3], a persistent key-value store. Figure 3c and Figure 3d show the write and read random throughput along the y-axis. We compare CompoundFS against ext4-DAX, SplitFS, and DevFS. For CompoundFS, we replace LevelDB's dual payload and checksum write operations with one *checksum_and_write* operation. First, for the random write workload, as observed in the microbenchmarks, even with our preliminary design, CompoundFS is able to achieve up to 75% performance improvement. For the blocking read operations, the data block and checksum are read at once. While both DevFS and CompoundFS avoid the system call cost and show performance improvement over DAX, the benefits of CompoundFS over DevFS are minimal. Our future work will explore more applications and pursue other opportunities for optimizations such as designing SplitFS-like hybrid user- and firmware-level design.

6 Conclusion

In this paper, we propose CompoundFS, a firmware-level filesystem that provides direct access to storage without compromising file system guarantees or POSIX support. To reduce data copy overheads between applications and the storage stack (e.g., filesystem), CompoundFS proposes CompoundOps, that combines one or more POSIX operations into a compound operation. CompoundOps also exploits device-level storage compute capability. Our ongoing work is currently focusing on crash-consistency and scheduling challenges. Results from our preliminary design show more than 75% performance gains for real applications.

Acknowledgements

We thank our Shepherd, Jeanna Matthews, and anonymous reviewers for their insightful feedback. We also thank Rutgers Panic Lab for giving us access to systems with DC Optane persistent memory. This work is supported by NSF CNS 1850297 award. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of NSF.

References

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

sched-design-CFS.html.

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

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

[4] Revolutionary Memory Technology. https://www.intel.com/content/www/us/en/architecture-and-technology/intel-optane-technology.html.

[5] Ringing in a new asynchronous I/O API. https://lwn.net/Articles/776703/.

[6] Anurag Acharya, Mustafa Uysal, and Joel Saltz. Active Disks: Programming Model, Algorithms and Evaluation. *SIGPLAN Not.*, 33(11):81–91, October 1998.

[7] 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*, pages 353–369, 2019.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] B. Gu, A. S. Yoon, D. Bae, I. Jo, J. Lee, J. Yoon, J. Kang, M. Kwon, C. Yoon, S. Cho, J. Jeong, and D. Chang. Biscuit: A Framework for Near-Data Processing of Big Data Workloads. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 153–165, June 2016.

[15] Abdelkader Hameurlain and Roland R. Wagner, editors. *Transactions on Large-Scale Data- and Knowledge-Centered Systems XXXVII*, volume 10940 of *Lecture Notes in Computer Science*. Springer, 2018.

[16] Zsolt István, David Sidler, and Gustavo Alonso. Caribou: Intelligent Distributed Storage. *Proc. VLDB Endow.*, 10(11):1202–1213, August 2017.

[17] 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.

[18] 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.

[19] Sang-Woo Jun, Ming Liu, Sungjin Lee, Jamey Hicks, John Ankcorn, Myron King, Shuotao Xu, and Arvind. BlueDBM: Distributed Flash Storage for Big Data Analytics. *ACM Trans. Comput. Syst.*, 34(3):7:1–7:31, June 2016.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Hariharan Gopalakrishnan, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Wisckey: Separating keys from values in ssd-conscious storage. *ACM Transactions on Storage (TOS)*, 13(1):1–28, 2017.

[27] 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.

[28] 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.

[29] Luis Cavazos Quero, Young-Sik Lee, and Jin-Soo Kim. Self-sorting SSD: Producing sorted data inside active SSDs. In *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–7. IEEE, 2015.

[30] 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.

[31] 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.

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

[33] 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.

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

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

[36] 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.

[37] 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.