Designing a True Direct-Access File System with DevFS

Sudarsun Kannan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau,
University of Wisconsin—Madison; Yuangang Wang, Jun Xu, and
Gopinath Palani, *Huawei Technologies*

Designing a True Direct-Access File System with DevFS

Sudarsun Kannan, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau
University of Wisconsin–Madison

Yuangang Wang, Jun Xu, Gopinath Palani
Huawei Technologies

Abstract

We present DevFS, a direct-access file system embedded completely within a storage device. DevFS provides direct, concurrent access without compromising file system integrity, crash consistency, and security. A novel reverse-caching mechanism enables the usage of host memory for inactive objects, thus reducing memory load upon the device. Evaluation of an emulated DevFS prototype shows more than 2x higher I/O throughput with direct access and up to a 5x reduction in device RAM utilization.

1 Introduction

The world of storage, after decades of focus on hard-drive technologies, is finally opening up towards a new era of fast solid-state storage devices. Flash-based SSDs have become standard technology, forming a new performance tier in the modern datacenter [7, 32]. New, faster flash memory technologies such as NVMe [20] and storage class memory (SCM) such as Intel's 3D X-point [1] promise to revolutionize how we access and store persistent data [10, 13, 50, 53]. State-of-the-art flash memory technologies have reduced storage-access latency to tens of microseconds compared to milliseconds in the hard-drive era [34, 52, 58].

To fully realize the potential of these storage devices, a careful reconsideration of the software storage stack is required. The traditional storage stack requires applications to trap into the OS and interact with multiple software layers such as the in-memory buffer cache, file system, and device drivers. While spending millions of cycles is not a significant problem for slow storage devices such as hard drives [3, 13, 58], for modern ultra-fast storage, software interactions substantially amplify access latencies, thus preventing applications from exploiting hardware benefits [3, 9, 34, 50]. Even the simple act of trapping into and returning from the OS is too costly for modern storage hardware [14, 49, 58].

To reduce OS-level overheads and provide direct storage access for applications, prior work such as Arrakis [34], Moneta-D [8], Strata [26], and others [20, 49, 50] split the file system into user-level and kernel-level components. The user-level component handles all data-plane operations (thus bypassing the OS), and the trusted kernel is used only for control-plane operations such as permission checking. However, prior approaches fail to deliver several important file-system properties. First, using untrusted user-level libraries to maintain file system metadata shared across multiple applications can seriously compromise file-system integrity and crash consistency. Second, unlike user-level networking [51], in file systems, data-plane operations (e.g., read or write to a file) are closely intertwined with control-plane operations (e.g., block allocation); bypassing the OS during data-plane operations can compromise the security guarantees of a file system. Third, most of these approaches require OS support when sharing data across applications even for data-plane operations.

To address these limitations, and realize a true user-level direct-access file system, we propose **DevFS**, a device-level file system inside the storage hardware. The DevFS design uses the compute capability and device-level RAM to provide applications with a high-performance direct-access file system that does not compromise integrity, concurrency, crash consistency, or security. With DevFS, applications use a traditional POSIX interface without trapping into the OS for control-plane and data-plane operations. In addition to providing direct storage access, a file system inside the storage hardware provides direct visibility to hardware features such as device-level capacitance and support for processing data from multiple I/O queues. With capacitance, DevFS can safely commit data even after a system crash and also reduce file system overhead for supporting crash consistency. With knowledge of multiple I/O queues, DevFS can increase file system concurrency by providing each file with its own I/O queue and journal.

A file system inside device hardware also introduces new challenges. First, even modern SSDs have limited RAM capacity due to cost ($/GB) and power constraints. In DevFS, we address this dilemma by introducing *reverse caching,* an approach that aggressively moves inactive file system data structures off the device to the host memory. Second, a file system inside a device is a separate runtime and lacks visibility to OS state (such as process credentials) required for secured file access. To overcome this limitation, we extend the OS to coordinate with DevFS: the OS performs down-calls and shares process-level credentials without impacting direct storage access for applications.

To the best of our knowledge, DevFS is the first design to explore the benefits and implications of a file system inside the device to provide direct user-level access to applications. Due to a lack of real hardware, we implement and emulate DevFS at the device-driver level. Evaluation of benchmarks on the emulated DevFS prototype with direct storage access shows more than 2x higher write and 1.6x higher read throughput as compared to a kernel-level file system. DevFS memory-reduction techniques reduce file system memory usage by up to 5x. Evaluation of a real-world application, Snappy compression [11], shows 22% higher throughput.

In Section 2, we first categorize file systems, and then discuss the limitations of state-of-the-art user-level file systems. In Section 3, we make a case for a device-level file system. In Section 4, we detail the DevFS design and implementation, followed by experimental evaluations in Section 5. In Section 6, we describe the related literature, and finally present our conclusions in Section 7.

2 Motivation

Advancements in storage hardware performance have motivated the need to bypass the OS stack and provide applications with direct access to storage. We first discuss hardware and software trends, followed by a brief history of user-level file systems and their limitations.

2.1 H/W and S/W for User-Level Access

Prior work has explored user-level access for PCI-based solid state drives (SSD) and nonvolatile memory technologies.

Solid-state drives. Solid-state drives (SSD) have become the de facto storage device for consumer electronics as well as enterprise computing. As SSDs have evolved, their bandwidth has significantly increased along with a reduction in access latencies [6, 57]. To address system-to-device interface bottlenecks, modern SSDs have switched to a PCIe-based interface that can support up to 8-16 GB/s maximum throughput and 20-50 μs access latencies. Further, these modern devices use a

large pool of I/O queues to which software can concurrently submit requests for higher parallelism.

With advancements in SSD hardware performance, bottlenecks have shifted to software. To reduce software overheads on the data path and exploit device-level parallelism, new standards such as NVMe [54] have been adopted. NVMe allows software to bypass device driver software and directly program device registers with simple commands for reading and writing the device [18].

Storage class memory technologies. Storage class memory (SCM), such as Intel's 3D Xpoint [1] and HP's memristors, are an emerging class of nonvolatile memory (NVM) that provide byte-addressable persistence and provide access via the memory controller. SCMs have properties that resemble DRAM more than a block device. SCMs can provide 2-4x higher capacity than DRAMs, with variable read (100-200ns) and write latency (400-800ns) latency. SCM bandwidth ranges from 8 GB/s to 20 GB/s, which is significantly faster than state-of-the-art SSDs. Importantly, read (loads) and writes (stores) to SCMs happen via the processor cache which plays a vital role in application performance.

Several software solutions that include kernel-level file systems [10, 13, 55, 56], user-level file systems [49], and object storage [17, 50] libraries have been proposed for SCM. Kernel-level file systems retain the POSIX-based block interface and focus on thinning the OS stack by replacing page cache and block layers with simple byte-level load and store operations. Alternative approaches completely bypass the kernel by using an object-based or POSIX-compatible interface over memory-mapped files [49].

2.2 File System Architectures

We broadly categorize file systems into three types: (1) kernel-level file systems, (2) hybrid user-level file systems, and (3) true user-level direct-access file systems. Figure 1 shows these categories, and how control-plane and data-plane operations are managed by each. The figure additionally shows a hybrid user-level file system with a trusted server and a Fuse-based file system.

Kernel-level traditional file systems. Kernel-level file systems act as a central entity managing data and metadata operations as well as control-plane operations [13, 28, 56]. As shown in Figure 1, kernel-level file systems preserve the integrity of metadata and provide crash consistency. Applications using kernel-level file systems trap into the OS for both data-plane and control-plane plane operations.

Hybrid user-level file systems. To allow applications to access storage hardware directly without trapping into the kernel, a class of research file systems [8, 26, 34] split the file system across user and kernel space. In this paper, we refer to these as hybrid user-level file systems. As
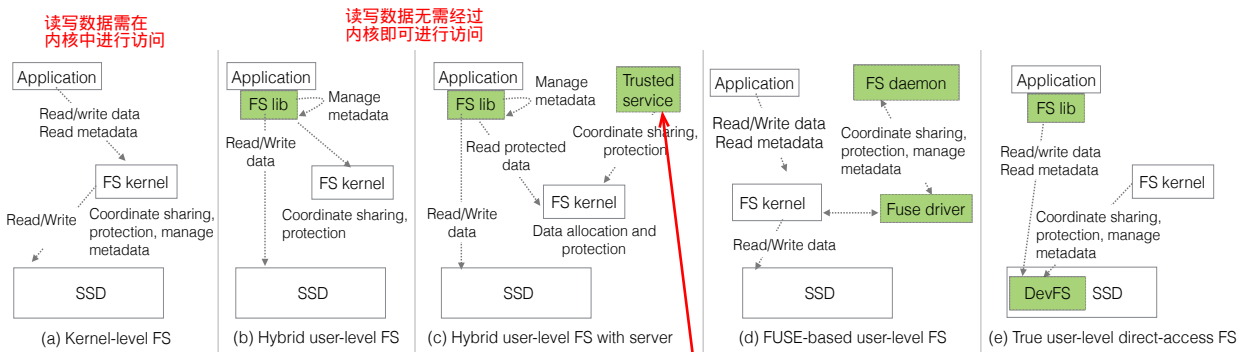
Figure 1: **File system categories.** *(a) shows a kernel-level file system, which manages control-plane and data-plane operations. (b) shows a hybrid user-level file system in which a user library manages data-plane operations. (c) shows a hybrid user-level file system with a trusted server. The server partially manages control-plane operations. (d) shows a Fuse-based file system. (e) shows a true user-level direct-access file system inside the device. The device file system fully manages the control-plane and data-plane.*

shown in Figure 1.b, the user-level file system manages all data and metadata updates without trapping into the kernel for common-case read and writes. The kernel file system is used only for control-plane operations such as permission checks, security, and data sharing across applications.

More specifically, Arrakis [34] is a hybrid user-level file system that proposes a generic user-level I/O framework for both network and storage. Arrakis aims to realize the ideas of U-Net [51] for modern hardware by virtualizing storage for each application and managing all data-plane operations at user-level, but does trap into the OS for control-plane operations. Strata [26], a hybrid user-level file system designed to combine ultra-fast NVM with high-capacity SSD and hard disk, uses NVM as a memory-mapped user-space log and writes application's data-plane operations to a log. A background thread uses a kernel-level file system to digest the logs to SSD or hard disk. For sharing files across processes, Strata traps into the kernel-level file system, which coordinates concurrent data and metadata updates.

Moneta-D [8] is a hybrid user-level file system that customizes SSDs to provide direct-access for data-plane operations. Moneta-D virtualizes an I/O interface (I/O channel) instead of storage to provide isolation and concurrency. Metadata operations are split between user-level and kernel-level. Operations such as file creation and size extension happen inside the kernel. Moneta-D enforces permission checks for data-plane operations with a user-level driver that reads a file's permission and stores them in hardware registers; during an I/O operation, the driver compares the hardware register values with process credentials.

Finally, TxDev [37] proposes a transactional flash system in which each process encapsulates its updates into a transaction request, and the flash device serializes and atomically commits the transactions. While TxDev can reduce the overheads of transactions in either user-level or kernel-level file systems, the resulting system has the

same structural advantages and disadvantages of other hybrid user-level file systems.

Hybrid file systems with trusted server. Another class of hybrid file systems, such as Aerie [49], aims to reduce kernel overheads for control-plane operations by using a trusted user-level third-party server similar to a microkernel design [30] (see Figure 1.c). The trusted server runs in a separate address space and facilitates control-plane operations such as permission checking and data sharing; the server also interacts with the OS for other privileged operations.

Fuse-based user-level file systems. Another class of user-level file systems widely known as Fuse [38, 47], are mainly used for customizing and extending the in-kernel file system. As shown in Figure 1.d, in Fuse, the file system is split across a kernel driver and a user-level daemon. All I/O operations trap into the kernel, and the kernel driver simply queues I/O requests for the custom user-level file system daemon to process requests and return control to the application via the driver; as a result, Fuse file systems add an extra kernel trap for all I/O operations. Because we focus on direct-access storage solutions, we do not study Fuse in rest of this paper.

True direct-access user-level file system. In this paper, we propose DevFS, a true user-level direct-access file system as shown in Figure 1.e. DevFS pushes the file system into the device, thus allowing user-level libraries and applications to access storage without trapping into the OS for both control-plane and data-plane operations.

2.3 Challenges

Current state-of-the-art hybrid user-level file systems fail to satisfy three important properties – integrity, crash consistency, and permission enforcement – without trading away direct storage access. We discuss the challenges in satisfying these properties while providing direct access next.

2.3.1 File System Integrity

Maintaining file system integrity is critical for correct behavior of a file system. In traditional file systems, only the trusted kernel manages and updates both in-memory and persistent metadata. However, satisfying file system integrity is hard with a hybrid user-level file system for the following reasons.

Single process. In hybrid user-level file systems such as Arrakis and Moneta-D, each application uses an instance of a file system library that manages both data and metadata. Consider an example of appending a block to a file: the library must allocate a free block, update the bitmap, and update the inode inside a transaction. A buggy or malicious application sharing the address space with the file system library can easily bypass or violate the transaction and incorrectly update the metadata; as a result, the integrity of the file system is compromised. An alternative approach is to use a trusted user-level server as in Aerie [49]. However, because applications and the user-level server run in different address spaces, applications must context-switch even for data-plane operations, thus reducing the benefits of direct storage access [58]. The metadata integrity problem cannot be solved by using TxDev (a transactional flash) in a hybrid user-level file system because TxDev cannot verify the contents of transactions composed by an untrusted user-level library.

Concurrent access and sharing. Maintaining integrity with hybrid user-level file systems is more challenging when applications concurrently access the file system or share data. Updates to in-memory and on-disk metadata must be serialized and ordered across all library instances with some form of shared user-level locking and transactions across libraries. However, a malicious or buggy application can easily bypass the lock or the transaction to update metadata or data, which can lead to an incorrect file system state [25]. Prior systems such as Arrakis [34] and Strata [26] sidestep this problem by trapping into the OS for concurrent file-system access (common-case) and concurrent file access (rare). In contrast, approaches such as Aerie suffer from the context-switch problem.

2.3.2 Crash Consistency

A system can crash or lose power before all in-memory metadata is persisted to storage, resulting in arbitrary file system state such as a persisted inode without its pointed-to data [4, 35, 36]. To provide crash consistency, kernel file systems carefully orchestrate the order of metadata and data updates. For example, in an update transaction, data blocks are first flushed to a journal, followed by the metadata blocks, and finally, a transaction commit record is written to the journal; at some point, the log updates are checkpointed to the original data and metadata locations to free space in the log.

For user-level file systems, every application's un-

trusted library instance must provide crash consistency, which is challenging for the following reasons. First, if even a single library or application violates the ordering protocol, the file system cannot recover to a consistent state after a crash. Second, with concurrent file system access, transactions across libraries must be ordered; as discussed earlier, serializing updates with user-level locking is ineffective and can easily violate crash consistency guarantees. While a trusted third-party server can enforce ordering, applications suffer from context switches and thus do not achieve the goal of direct access.

2.3.3 Permission Enforcement

Enforcing permission checks for both the control-plane and data-plane is critical for file system security. In a kernel-level file system, when a process requests an I/O operation, the file system uses OS-level process credentials and compares it with the corresponding file (inode) permission. Hybrid user-level file systems [34] use the trusted OS for permission checks only for control-plane operation, and bypass the checks for common-case data-plane operations. Avoiding permission checks for data-plane operations violates security guarantees, specifically when multiple applications share a file system.

3 The Case For DevFS

In the pursuit of providing direct storage access to user-level applications, prior hybrid approaches fail to satisfy one or more fundamental properties of a file system. To address the limitations, and design a true direct-access file system, we propose DevFS, a design that moves the file system inside the device. Applications can directly access DevFS using a standard POSIX interface. DevFS satisfies file system integrity, concurrency, crash consistency, and security guarantees of a kernel-level file system. DevFS also supports a traditional file-system hierarchy such as files and directories, and their related functionality instead of primitive read and write operations. DevFS maintains file system integrity and crash consistency because it is trusted code that acts as a central entity. With minimal support and coordination with the OS, DevFS also enforces permission checks for common-case data-plane operations without requiring applications to trap into the kernel.

3.1 DevFS Advantages And Limitations

Moving a file system inside the device provides numerous benefits but also introduces new limitations.

Benefits. An OS-level file system generally views storage as a black box and lacks direct control over many hardware components, such as device memory, I/O queues, power-loss-protection capacitors, and the file

translation layer (FTL). This lack of control results in a number of limitations.

First, even though storage controllers often contain multiple CPUs that can concurrently process requests from multiple I/O queues [20], a host-based user-level or kernel-level file system cannot control how the device CPUs are utilized, the order in which they process request from queues, or the mapping of queues to elements such as files. However, a device-level file system can redesign file system data structures to exploit hardware-level concurrency for higher performance.

Second, some current devices contain capacitors that can hold power until the device CPUs safely flush all device-memory state to persistent storage in case of an untimely crash [22, 44]. Since software file systems cannot directly use these capacitors, they always use high-overhead journaling or copy-on-write crash consistency techniques. In contrast, a device-level file system can ensure key data structures are flushed if a failure occurs.

Finally, in SSDs and storage class memory technologies, the FTL [21] performs block allocation, logical-to-physical block translation, garbage collection, and wear-leveling, but a software file system must duplicate many of these tasks since it lacks visibility of the FTL. We believe that DevFS provides an opportunity to integrate file system and FTL functionality, but we do not yet explore this idea, leaving it to future work.

Limitations. Moving the file system into the storage device introduces both hardware and software limitations. First, device-level RAM is limited by cost and power consumption; currently device RAM is used mainly by the FTL [16] and thus the amount is proportional to the size of the logical-to-physical block mapping table (e.g., a 512 GB SSD requires a 2 GB RAM). A device-level file system will substantially increase memory footprint and therefore must strive to reduce its memory usage. Second, the number of CPUs inside a storage device can be limited and slower compared to host CPUs. While the lower CPU count impacts I/O parallelism and throughput, the slower CPUs reduce instructions per cycle (IPC) and thus increase I/O latency. Finally, implementing OS utilities and features, such as deduplication, incremental backup, and virus scans, can be challenging. We discuss these limitations and possible solutions in more detail in § 4.7.

Regarding software limitations, a device-level file system runs in a separate environment from the OS and hence cannot rely on the OS for certain functionality or information. In particular, a device-level file system must manage its own memory and must provide a mechanism to access process credentials from the OS.

3.2 Design Principles

To exploit the benefits and address the limitations of a device-level file system, we formulate the following DevFS design principles.

Principle 1: Disentangle file system data structures to embrace hardware-level parallelism. To utilize the hardware-level concurrency of multiple CPU controllers and thousands of I/O queues from which a device can process I/O requests, DevFS maps each fundamental data unit (i.e., a file) to an independent hardware resource. Each file has its own I/O queue and in-memory journal which enables concurrent I/O across different files.

Principle 2: Guarantee file system integrity without compromising direct user-level access. To maintain integrity, the DevFS inside the device acts as a trusted central entity and updates file system metadata. To further maintain integrity when multiple process share data, DevFS shares per-file structures across applications and serializes updates to these structures.

Principle 3: Simplify crash consistency with storage hardware capacitance. Traditional OS-level file systems rely on expensive journaling or log-structured (i.e., copy-on-write) mechanisms to provide crash consistency. While journaling suffers from "double write" [35] costs, log-structured file systems suffer from high garbage-collection overheads [5]. In contrast, DevFS exploits the power-loss-protection capacitors in the hardware to safely update data and metadata in-place without compromising crash consistency. DevFS thus avoids these update-related overheads.

Principle 4: Reduce the device memory footprint of the file system. Unlike a kernel-level file system, DevFS cannot use copious amounts of RAM for its data and metadata structures. To reduce memory usage, in DevFS, only in-memory data structures (inodes, dentries, per-file structures) of <mark>active files</mark> are kept in device memory, spilling inactive data structures to host memory.

Principle 5: Enable minimal OS-level state sharing with DevFS. DevFS is a separate runtime and implements its own memory management. Concerning state sharing, because DevFS does not have information about processes, we extend the OS to share process credentials with DevFS. The credentials are used by DevFS for permission checks across control-plane and data-plane operations without forcing applications to trap into the kernel.

4 Design

DevFS provides direct user-level access to storage without trapping into the OS for most of its control-plane and data-plane operations. DevFS does not compromise basic file system abstractions (such as files, directories) and properties such as integrity, concurrency, consistency, and security guarantees. We discuss how DevFS realizes
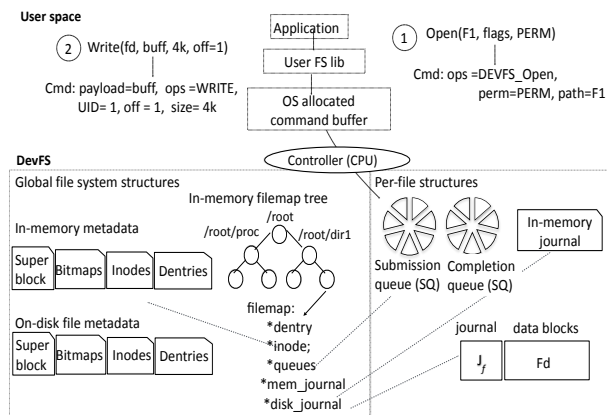
Figure 2: **DevFS high-level design.** *The file system data structure is partitioned into global and per-file structures. The per-file structures are created during file setup. DevFS metadata structures are similar to other kernel-level file system.*

the design principles discussed earlier.

4.1 Disentangling File System Structures

To exploit hardware-level concurrency, DevFS provides each file with a separate I/O queue and journal. DevFS is compatible with traditional POSIX I/O interface.

Global and per-file structures. In DevFS, the file system data structures are divided into global and per-file structures as shown in Figure 2. The global data structures manage state of an entire file system, including metadata such as the superblock, inodes, and bitmaps. The per-file structures enable concurrency: given that modern controllers contain up to four CPUs [41], and this amount is expected to increase [19], DevFS attempts to utilize multiple CPUs. In contrast to prior approaches such as Moneta-D that provide each application with its own I/O channel, DevFS provides a per-file I/O queue and journal. DevFS also maintains an in-memory filemap structure for each file. The filemap structure is created during file creation (or during file open if it is not available in device memory) and is maintained in a redblack tree as shown in the figure. Processes sharing a file also share a filemap structure which serializes access across the per-file I/O queue and the journal.

Most data structures of DevFS are similar to a kernellevel file system. Hence, we reuse and extend in-memory and on-disk data structures from the state-of-the-art persistent memory file system (PMFS) [13]. We use PMFS because it provides direct-access to storage bypassing the file system page cache. Specifically, the DevFS superblock contains global information of a file system, each inode contains per-file metadata and a reference to per-file memory and disk journal, and finally, directory entries (dentries) are maintained in a radix-tree indexed by hash values of file path names.

File system interface. Unlike prior approaches that

expose the storage device as a block device for direct access [34], DevFS supports the POSIX I/O interface and abstractions such as files, directories, etc. Similar to modern NVMe-based devices with direct-access capability, DevFS uses command-based programming. To support POSIX compatibility for applications, a user-level library intercepts I/O calls from applications and converts the I/O calls to DevFS commands. On receipt, the DevFS controller (device CPU) uses the request's file descriptor to move the request to a per-file I/O queue for processing and writing to storage.

4.2 Providing File System Integrity

To maintain integrity, file system metadata is always updated by the trusted DevFS. In contrast to hybrid userlevel file systems that allow untrusted user-level libraries to update metadata [34], in DevFS, there is no concern about the legitimacy of metadata content (beyond that caused by bugs in the file system).

When a command is added to a per-file I/O queue, DevFS creates a corresponding metadata log record (e.g., for a file append command, the bitmap and inode block), and adds the log record to a per-file in-memory journal using a transaction. When DevFS commits updates from an in-memory I/O queue to storage, it first writes the data followed by the metadata. Updates to global data structures (such as bitmaps) are serialized using locks.

DevFS supports file sharing across processes without trapping into the kernel. Because each file has separate in-memory structures (i.e., an I/O queue and journal), one approach would be to use separate per-file structures for each instance of an open file and synchronize updates across structures; however, synchronization costs and device-memory usage would increase linearly with the number of processes sharing a file. Hence, DevFS shares in-memory structures across processes and serializes updates using a per-file filemap lock; to order updates, DevFS tags each command with a time-stamp counter (TSC). Applications requiring strict data ordering for shared files could implement custom user-level synchronization at application-level.

4.3 Simplifying Crash Consistency

DevFS avoids logging to persistent storage by using device capacitors that can hold power until the device controller can safely flush data and metadata to storage. Traditional kernel-level file systems use either journaling or a copy-on-write techniques, such as log-structured updates, to provide crash consistency; the benefits and implications of these designs are well documented [5, 40]. Journaling commits data and metadata updates to a persistent log before committing to the original data and metadata location; as a result, journaling suffers from the "double write" problem [40, 56]. The log-structured design avoids double writes by treating an entire file system
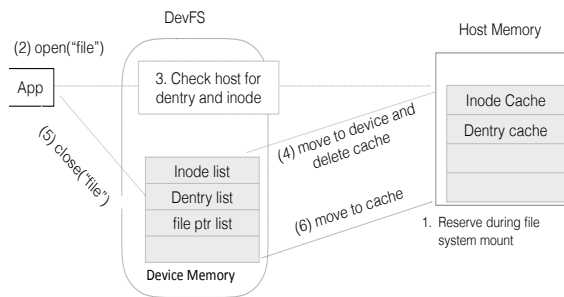
Figure 3: **DevFS reverse caching design.** *DevFS keeps only active and essential file system structures in device memory, and reverse caches others to host memory.*
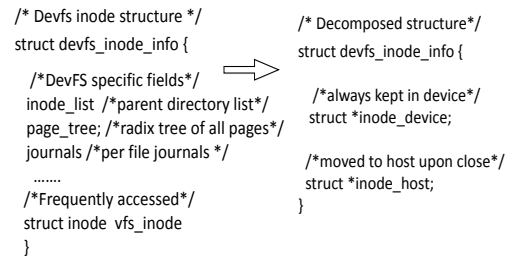

Figure 4: **Decomposing large structures.** *Large static in-memory inode is decomposed to a dynamically allocatable device and host structure. The host structure is reverse cached.*

as a log, appending data and metadata blocks; however, a log-structured design suffers from high garbage collection costs [40]. DevFS uses device-level capacitance to avoid both the double-write and garbage-collection problems.

Modern enterprise SSDs provide power-loss-protection capacitors inside device hardware that can hold power until controllers can safely flush contents of device-level DRAM [22, 44]. In existing systems, the device DRAM primarily contains the FTL's logical-to-physical block translation table, block error correction (ECC) flags, and in-flight data yet to be flushed to storage. Since DevFS runs inside the device, it uses device-level DRAM for all file system data structures.

Although the goal of hardware capacitance is to safely flush device in-memory contents to storage, flushing larger amounts of memory would require a more expensive capacitor; in addition, not all DevFS state needs to be made persistent. To minimize the memory state that must be flushed, DevFS leverages its per-file in-memory journals, as shown in Figure 2. As described previously, after an I/O command is added to a device queue, DevFS writes the command's metadata to a per-file in-memory journal. If a power failure or crash occurs, the device capacitors can hold power for controllers to safely commit in-memory I/O queues and journals to storage, thus avoiding journal writes to storage.

4.4 Minimizing Memory Footprint

We next discuss how DevFS manages device memory followed by three memory reduction techniques. The techniques include on-demand allocation, reverse caching, and a method to decompose inode structures.

DevFS uses its own memory allocator. Unlike the complex-but-generic Linux slab allocator [15], the DevFS allocator is simple and customized to manage only DevFS data structures. In addition to device memory, DevFS reserves and manages a DMA-able region in the host for reverse caching.

In DevFS, there are four types of data structures that dominate memory usage: in-memory inodes, dentries, file pointers, and the DevFS-specific per-file filemap

structure. Examining the data structures in detail, we see that each inode, dentry, file pointer, and filemap consume 840 bytes, 192 bytes, 256 bytes, and 156 bytes respectively. Since inodes are responsible for the most memory usage, we examine them further. We find that 593 bytes (70.5%) of the inode structure are used by generic fields that are frequently updated during file operations; referred to as the VFS inode in other file systems, this includes the inode number, a pointer to its data blocks, permissions, access times, locks, and a reference to the corresponding dentry. The remaining 247 bytes (29.5%) of the inode are used by DevFS-specific fields, which include a reference to in-memory and on-disk journals, the dentry, the per-file structure, other list pointers, and per-file I/O queues. To reduce the device memory usage, we propose the following techniques.

On-demand memory allocation. In a naive DevFS design, the in-memory structures associated with a file, such as the I/O queue, in-memory journal, and filemap, are each allocated when a file is opened or created and not released until a file is deleted; however, these structures are not used until an I/O is performed. For workloads that access a large number of files, device memory consumption can be significant. To reduce memory consumption, DevFS uses on-demand allocation that delays allocation of in-memory structures until a read or write request is initiated; these structures are also aggressively released from device memory when a file is closed. Additionally, DevFS dynamically allocates the per-file I/O queue and memory journal and adjusts their sizes based on the availability of free memory.

Reverse caching metadata structures. In traditional OS-level file systems, the memory used by in-memory metadata structures such as inodes and dentries is a small fraction of the overall system memory; therefore, these structures are cached in memory even after the corresponding file is closed in order to avoid reloading the metadata from disk when the file is re-accessed. However, caching metadata in DevFS can significantly increase memory consumption. To reduce device memory usage, DevFS moves certain metadata structures such as in-memory inodes and dentries to host memory after a

file is closed. We call this **reverse caching** because meta-data is moved off the device to the host memory.

Figure 3 shows the reverse caching mechanism. A DMA-able host-memory cache is created when DevFS is initialized. The size of the host cache can be configured when mounting DevFS depending on the availability of free host memory; the cache is further partitioned into inode and dentry regions. After moving an inode or dentry to host memory, all its corresponding references (e.g., the inode list of a directory) are updated to point to the host-memory cache. When a file is re-opened, the cached metadata is moved back to device memory. Directories are reverse-cached only after all files in a directory are also reverse-cached. Note that the host cache contains only inodes and dentries of inactive (closed) files, since deleted files are also released from the host cache. Furthermore, in case of an update to in-memory structures that are reverse-cached, the structures are moved to device memory and cached structures in the host memory are deleted. As a result, reverse caching does not introduce any consistency issues. Although using host memory instead of persistent storage as a cache avoids serializing and deserializing data structures, the overhead of data movement between device and host memory depends on interface bandwidth. The data movement overhead could be further reduced by using incremental (delta-based) copying techniques.

Decomposing file system structures. One problem with reverse caching for a complicated and large structure such as an inode is that some fields are accessed even after a file is closed. For example, a file's inode in the directory list is traversed for search operations or other updates to a directory. Moving these structures back and forth from host memory can incur high overheads. To avoid this movement, we decompose the inode structure into a device-inode and host-inode structure as shown in the Figure 4. The device-inode contains fields that are accessed even after a file is closed, and therefore only the host-inode structure is moved to host memory. Each host inode is approximately 593 bytes of the overall 840 bytes. Therefore, this decomposition along with reverse caching significantly reduces inode memory use.

4.5 State Sharing for Permission Check

DevFS provides security for control-plane and data-plane operations without trapping into the kernel by extending the OS to share application credentials.

In a kernel-level file system, before an I/O operation, the file system uses the credentials of a process from the OS-level process structure and compares them with permission information stored in an inode of a file or directory. However, DevFS is a separate runtime and cannot access OS-level data structures directly. To overcome this limitation, as shown in Figure 5, DevFS maintains
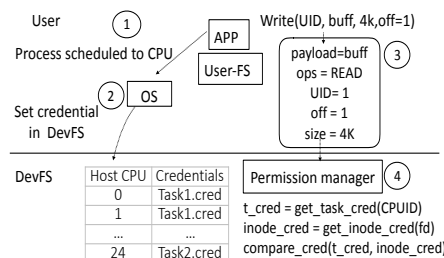

Figure 5: **DevFS permission check design.** *The OS is responsible for updating DevFS credential table with process credentials after a context-switch.*

a credential table in device memory that can be accessed and updated only by the OS, which updates the table with credential information of a new process scheduled on a host CPU. When an I/O request is sent from the host, the request is tagged with an ID number of the initiating CPU. We assume that CPU ID tagged with a request is unforgeable by an untrusted process; DevFS can be easily extended to support other types of unforgeable IDs. Before processing a request, DevFS performs a simple table lookup to compare credentials of a process running on the initiating CPU with the corresponding inode's permissions. Invalid requests are returned with a permission error in the request's completion flag.

We note that one intricate scenario can occur when a process is context-switched from its host CPU before DevFS can process the request. We address this scenario using the following steps: first, whenever a new process is scheduled to use a host CPU, the OS scheduler updates the credential table in DevFS with credentials of currently running process; second, a request is admitted to the device I/O queue only after a permission check. These steps allow DevFS to safely execute requests in the I/O queue even after a process is context-switched. Our future work will examine the overheads of OS down-calls to update the device-level credential table when processes are frequently context-switched across host CPUs.

4.6 Implementation and Emulation

We implement the DevFS prototype to understand the benefits and implications of a file system inside a storage device. Due to the current lack of programmable storage hardware, we implement DevFS as a driver in the Linux 4 kernel and reserve DRAM at boot time to emulate DevFS storage. We now describe our implementation of the DevFS user-level library and device-level file system.

User-level library and interface. DevFS utilizes command-based I/O, similar to modern storage hardware such as NVMe [54, 57]. The user library has three primary responsibilities: to create a command buffer in host memory, to convert the applications POSIX interface into DevFS commands and add them to the com-

mand buffer, and to ring a doorbell for DevFS to process the request. When an application is initialized, the user-level library creates a command buffer by making an ioctl call to the OS, which allocates a DMA-able memory buffer, registers the allocated buffer, and returns the virtual address of the buffer to the user-level library. Currently, DevFS does not support sharing command buffers across processes, and the buffer size is restricted by the Linux kernel allocation (*kmalloc()*) upper limit of 4 MB; these restrictions can be addressed by memory-mapping a larger region of shared memory in the kernel. The user-library adds I/O commands to the buffer and rings a doorbell (emulated with an ioctl) with the address of the command buffer from which DevFS can read I/O requests, perform permission checks, and add them to a device-level I/O queue for processing. For simplicity, our current library implementation only supports synchronous I/O operations: each command has an I/O completion flag that will be set by DevFS, and the user-library must wait until an I/O request completes. The user-library is implemented in about 2K lines of code.

DevFS file system.   Because DevFS is a hardware-centric solution, DevFS uses straightforward data structures and techniques that do not substantially increase memory or CPU usage. We extend PMFS with DevFS components and structures described earlier. Regarding DevFS block management, each block in DevFS is a memory page; pages for both metadata and data are allocated from memory reserved for DevFS storage. The per-file memory journal and I/O queue size are set to a default of 4 KB but are each configurable during file system mount. The maximum number of concurrently opened files or directories is limited by the number of I/O queues and journals that can be created in DevFS memory. Finally, DevFS does not yet support memory-mapped I/O. DevFS is implemented in about 9K lines of code.

4.7   Discussion

Moving the file system inside a hardware device avoids OS interaction and allows applications to attain higher performance. However, a device-level file system also introduces CPU limitations and adds complexity in deploying new file system features.

CPU limitations.   The scalability and performance of DevFS is dependant on the device-level CPU core count and their frequency. These device CPU limitations can impact (a) applications (or a system with many applications) that use several threads for frequent and non-dependant I/O operations, (b) multi-threaded applications that are I/O read-intensive or metadata lookup-intensive, and finally, (c) CPU-intensive file system features such as deduplication or compression. One possible approach to address the CPU limitation is to iden-

tify file-system operations and components that are CPU-intensive and move them to the user-level library in a manner that does not impact integrity, crash consistency, and security. However, realizing this approach would require extending DevFS to support a broader set of commands from the library in addition to application-level POSIX commands. Furthermore, we believe that DevFS's direct-access benefits could motivate hardware designers to increase CPU core count inside the storage device [19], thus alleviating the problem.

Feature support.   Moving the file system into storage complicates the addition of new file system features, such as snapshots, incremental backup, deduplication, or fixing bugs; additionally, limited CPU and memory resources also add to the complexity. One approach to solving this problem is by implementing features that can be run in the background in software (OS or library), exposing the storage device as a raw block device, and using host CPU and memory. Another alternative is to support "reverse computation" by offloading file system state and computation to the host. Our future work will explore the feasibility of these approaches by extending DevFS to support snapshots, deduplication, and software RAID. Regarding bug fixes, changes to DevFS would require a firmware upgrade, which is supported by most hardware vendors today [45]. Additionally, with increasing focus on programmability of I/O hardware (e.g., NICs [8, 29]) as dictated by new standards (e.g., NVMe), support for embedding software into storage should become less challenging.

5   Evaluation

Our evaluation of DevFS aims to answer the following important questions.

- What is the performance benefit of providing applications with direct-access to a hardware-level file system?
- Does DevFS enable different processes to simultaneously access both the same file system and the same files?
- What is the performance benefit of leveraging device capacitance to reduce the double write overhead of a traditional journal?
- How effective are DevFS's memory reduction mechanisms and how much do they impact performance?
- What is the impact of running DevFS on a slower CPU inside the device compared to the host?

We begin by describing our evaluation methodology and then we evaluate DevFS on micro-benchmarks and real-world applications.
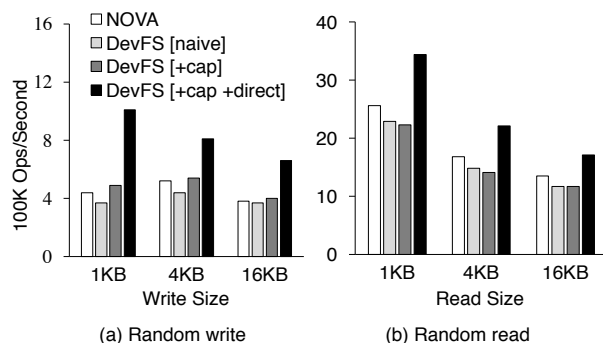
Figure 6: **Write and Read throughput.** *The graph shows results for Filebench random write and read micro-benchmark. X-axis varies the write size, and the file size is kept constant to 32 GB. Results show single thread performance. For DevFS, the per-file I/O queue and in-memory journal is set to 4 KB.*

5.1 Methodology

For our experiments, we use a 40-core Intel Xeon 2.67 GHz dual socket system with 128 GB memory. DevFS reserves 60 GB of memory to emulate storage with maximum bandwidth and minimum latency. DevFS is run on 4 of the cores to emulate a storage device with 4 CPU controllers and with 2 GB of device memory, matching state-of-the-art NVMe SSDs [41, 42].

5.2 Performance

Single process performance. We begin by evaluating the benefits of direct storage access for a very simple workload of a single process accessing a single file with the Filebench workload generator [48]. We study three versions of DevFS: a naive version of DevFS with traditional journaling, DevFS with hardware capacitance support (+cap), and DevFS with capacitance support and without kernel traps (+cap +direct). We emulate DevFS without kernel traps by replicating the benchmark inside a kernel module. For comparison, we use NOVA [56], a state-of-the-art kernel-level file system for storage class memory technologies. Although NOVA does not provide direct access, it does use memory directly for storage and uses a log-structured design.

Figure 6.a shows the throughput of random writes as a function of I/O size. As expected, NOVA performs better than naive DevFS with traditional journaling. Because NOVA uses a log-structured design and writes data and metadata to storage only once, it outperforms DevFS-naive with traditional journaling since DevFS-naive writes to an in-memory journal, a per-file storage log, and the final checkpointed region. For larger I/O sizes (16 KB), the data write starts dominating the cost, thus reducing the impact of journaling on the performance.

However, DevFS with capacitance support, DevFS+cap, exploits the power-loss-protection capability and only writes metadata to the in-memory journal; both the metadata and the data can be directly
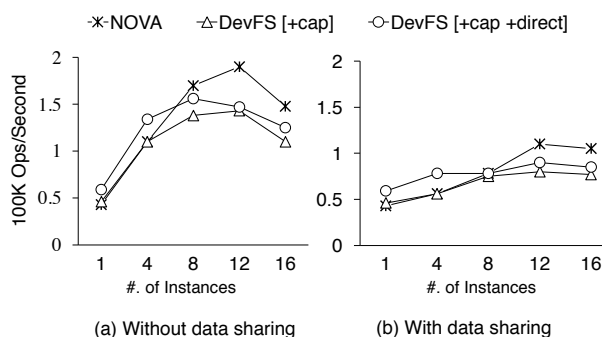

Figure 7: **Concurrent access throughput.** *(a) shows throughput without data sharing. (b) shows throughput with data sharing. The x-axis shows the number of concurrent instances. Each instance opens ten files, appends 256 MB to each file using 4 KB writes, and then closes the files. DevFS uses up to 4 device CPUs.*

committed to storage in-place without a storage log. For 1-KB writes, DevFS+cap achieves up to 27% higher throughput than the naive DevFS approach and 12% higher than NOVA. DevFS+cap outperforms NOVA because NOVA must issue additional instructions to flush its buffers, ordering writes to memory with a barrier after each write operation. Finally, by avoiding kernel traps, DevFS+cap+direct provides true direct-access to storage and improves performance by more than 2x and 1.8x for 1-KB and 4-KB writes respectively.

Figure 6.b shows random read throughput. NOVA provides higher throughput than both the DevFS-naive and DevFS+cap approaches because our prototype manages all 4 KB blocks of a file in a B-tree and traverses the tree for every read operation; in contrast, NOVA simply maps a file's contents and converts block offsets to physical addresses with bit-shift operations, which is much faster. Even with our current implementation, DevFS+direct outperforms all other approaches since it avoids expensive kernel traps. We believe that incorporating NOVA's block mapping technique into DevFS would further improve read performance.

Concurrent access performance. One of the advantages of DevFS over existing hybrid user-level file systems is that DevFS enables multiple competing processes to share the same file system and the same open files. To demonstrate this functionality, we begin with a workload in which processes share the same file system, but not the same files: each process opens ten files, appends 256 MB to each file using 4-KB writes, and then closes the files. In Figure 7.a, the number of processes is varied along the x-axis, where each process writes to a separate directory.

For a single process, DevFS+direct provides up to a 39% improvement over both NOVA and DevFS+cap by avoiding kernel traps. Since each file is allocated its own I/O queues and in-memory journal, the performance of DevFS scales well up to 4 instances; since we are emulating 4 storage CPUs, beyond four instances, the device

CPUs are shared across multiple instances and performance does not scale well. In contrast, NOVA is able to use all 40 host CPUs and scales better.

To demonstrate that multiple processes can simultaneously access the same files, we modify the above workload so that each instance accesses the same ten files; the results are shown in Figure 7.b. As desired for file system integrity, when multiple instances share and concurrently update the same file, DevFS serializes metadata updates and updates to the per-file I/O queue and in-memory journal. Again, scaling of DevFS is severely limited beyond 4 instances given the contention for the 4 device CPUs. In other experiments, not shown due to space limitations, we observe that increasing the number of device CPUs directly benefits DevFS scalability.

Summary. By providing direct-access to storage without trapping into the kernel, DevFS can improve write throughput by 1.5x to 2.3x and read throughput by 1.2x to 1.3x. DevFS also benefits from exploiting device capacitance to reduce journaling cost. Finally, unlike hybrid user-level file systems, DevFS supports concurrent file-system access and data sharing across processes; lower I/O throughput beyond four concurrent instances is mainly due to a limited number of device-level CPUs.

5.3 Impact of Reverse Caching

A key goal of DevFS is to reduce memory usage of the file system. We first evaluate the effectiveness of DevFS memory optimizations to reduce memory usage and then investigate the impact on performance.

5.3.1 Memory Reduction

To understand the effectiveness of DevFS memory-reduction techniques, we begin with DevFS+cap and analyze three memory reduction techniques: DevFS+cap+demand allocates each in-memory filemap on-demand and releases them after a file is closed; DevFS+cap+demand+dentry reverse caches the corresponding dentry after a file is closed; DevFS+cap+demand+dentry+inode also decomposes a file's inode into inode-device and inode-host structures and reverse caches the inode-host structure. Because we focus on memory reduction, we do not consider DevFS+direct in this experiment.

Figure 8 shows the amount of memory consumed for the four versions of DevFS on Filebench's file-create workload that opens a file, writes 16 KB, and then closes the file for 1 million files. In the baseline (DevFS+cap), three data structures dominate memory usage: the DevFS inode (840 bytes), the dentry (192 bytes), and the filemap (156 bytes). While file pointers, per-file I/O queues, and in-memory journals are released after a file is closed, the three other structures are not freed until the file is deleted.

The first memory optimization, DevFS+cap+demand, dynamically allocates the filemap when a read or write
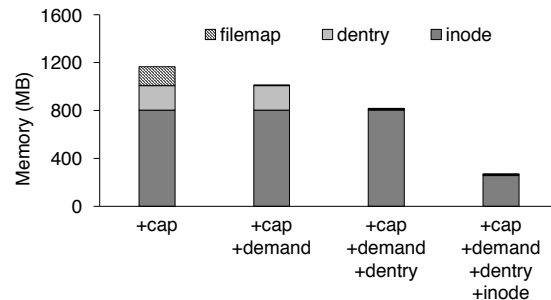

Figure 8: **DevFS memory reduction.** +*cap represents a baseline without memory reduction. Other bars show incremental memory reduction technique impact.*
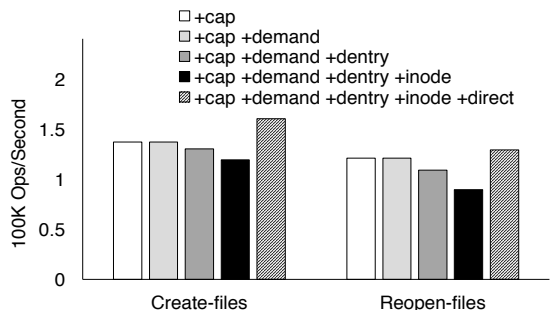

Figure 9: **Throughput impact of memory reduction.** *Reopen-files benchmark reopens closed files; as a result, structures cached in host memory are moved back to device.*

is performed and releases the filemap after closing the file; this reduces memory consumption by 156 MB (13.4%). Reverse caching of dentries, shown by DevFS+cap+demand+dentry, reduces device memory usage by 193 MB (16.6%) by moving them to the host memory; the small dentry memory usage visible in the graph represents directory dentries which are not moved to the host memory in order to provide fast directory lookup. Finally, decomposing the large inode structure into two smaller structures, inode-device (262 bytes) and inode-host (578 bytes), and reverse caching the inode-host structure reduces memory usage significantly. The three mechanisms cumulatively reduce device memory usage by up to 78% (5x) compared to the baseline. In our current implementation, we consider only these three data structures, but reverse caching could easily be extended to other file system data structures.

5.3.2 Performance Impact

The memory reduction techniques used by DevFS do have an impact on performance. To evaluate their impact on throughput, in addition to the file-create benchmark used above, we also evaluate a file-reopen workload that re-opens each of the files in the file-create benchmark immediately after it is closed. We also show the throughput for direct-access (DevFS+cap+demand+dentry+inode+direct) that avoids expensive kernel traps.

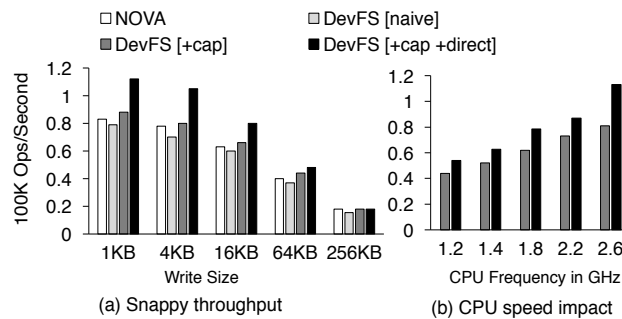For both benchmarks, DevFS with no memory opti-

Figure 10: **Snappy compression throughput and CPU speed impact.** *Application uses 4 CPUs. Memory reduction techniques are enabled for DevFS (+cap) and DevFS (+cap +direct). For DevFS (+cap +direct), Snappy is run as a kernel module. The CPU speed is varied by scaling the frequency.*

mizations and DevFS with on-demand allocation have similar throughput because the only difference is exactly when the filemap is allocated. However, the reverse caching techniques do impact throughput. For the file-create benchmark, reverse caching only the dentry (DevFS+cap+demand+dentry) reduces throughput by 5%, while also reverse caching the inode (DevFS+cap+demand+dentry+inode) by 13%. Performance degradation occurs because reverse caching involves significant work: allocating memory in the host DRAM, copying structures to host memory, updating the parent list with the new memory address, and later releasing the device memory. The performance of reverse caching inodes is worse than that of dentries, due to their relative sizes (578 bytes vs 196 bytes). While the direct-access approach has similar overheads, by avoiding kernel traps for file open, write, and close, and it provides higher performance compared to all other approaches.

With the file-reopen benchmark, reverse caching moves the corresponding inodes and dentries back to device memory, causing a throughput drop of 26%. Our results for the file-reopen benchmark can be considered worst-case behavior since most real-world applications spend more time performing I/O before closing a file. Our current mechanism performs aggressive reverse caching, but could easily be extended to slightly delay reverse caching based on the availability of free memory in the device.

Summary. DevFS memory-reduction techniques can reduce device DRAM usage by up to 5x. Although worst-case benchmarks do suffer some performance impact with these techniques, we believe memory reduction is essential for device-level file systems and that DevFS will obtain both memory reduction and high performance for realistic workloads.

5.4 Snappy File Compression

To understand the performance impact on a real-world application, we use Snappy [11] compression. Snappy is widely used as a data compression engine for several applications including MapReduce, RocksDB, MongoDB, and Google Chrome. Snappy reads a file, performs compression, and writes the output to a file; for durability, we add an `fsync()` after writing the output. Snappy optimizes throughput and is both CPU- and I/O-intensive; for small files, the I/O time dominates the computation time. Snappy can be used at both user-level and kernel-level [23] which helps us to understand the impact of direct access. For the workload, we use four application threads, 16 GB of image files from OpenImage repository [24], and vary the size of files from 1 KB to 256 KB.

Comparing the performance of NOVA, DevFS-naive, DevFS+cap, and DevFS+direct, we see the same trends for the Snappy workload as we did for the previous micro-benchmarks. As shown in Figure 10.a, NOVA performs better than DevFS-naive due to DevFS-naive's journaling cost, while DevFS+cap removes this overhead. Because DevFS+direct avoids trapping into the kernel when reading and writing across all application threads, it provides up to 22% higher throughput than DevFS-cap for 4-KB files; as the file size increases, the benefit of DevFS+direct is reduced since compression costs dominate runtime.

Device CPU Impact. One of the challenges of DevFS is that it is restricted to the CPUs on the storage device, and these device CPUs may be slower than those on the host. To quantify this performance impact, we run the Snappy workload as we vary the speed of the "device" CPUs, keeping the "host" CPUs at their original speed of 2.6 GHz [27]; the threads performing compression always run on the fast "host" CPUs. Figure 10.b shows the performance impact for 4-KB file compression for two versions of DevFS; we choose 4-KB files since it stresses DevFS performance more than with larger files (which instead stress CPU performance). As expected, DevFS-direct consistently performs better than DevFS-cap. More importantly, we do see that reducing device CPU frequency does have a significant impact on performance (e.g., reducing device CPU frequency from 2.6 GHz to 1.4 GHz reduces throughput by 66%). However, comparing across graphs, we see that even with a 1.8 GHz device CPU, the performance of DevFS-direct is similar to that of NOVA running on all high-speed host CPUs. For workloads that are more CPU intensive, the impact of slower device CPUs on DevFS performance is smaller (not shown due to space constraints).

Summary. DevFS-direct provides considerable performance improvement even for applications that are both CPU and I/O-intensive. We observe that although slower device CPUs do impact performance of DevFS, DevFS can still outperform other approaches.

6 Related Work

Significant prior work has focused on providing direct-access to storage, moving computation to storage, or programmability of SSDs.

Direct-access storage. Several hybrid user-level file system implementations, such as Intel's SPDK [18], Light NVM [6], and Micron's User Space NVME [33] provide direct-access to storage by exposing them as a raw block device and exporting a userspace device driver for block access. Light NVM goes one step further to enable I/O-intensive applications to implement their own FTL. However, these approaches do not support traditional file-system abstractions and instead expose storage as a raw block device; they do not support fundamental properties of a file system such as integrity, concurrency, crash consistency, or security.

Computation inside storage. Providing compute capability inside storage for performing batch tasks have been explored for past four decades. Systems such as CASSM [46] and RARES [31] have proposed adding several processors to a disk for performing computation inside storage. ActiveStorage [2, 39] uses one CPU inside a hard disk for performing database scans and search operations, whereas Smart-SSD [12] is designed for query processing inside SSDs. Architectures such as BlueDBM [19] have shown the benefits of scaling compute and DRAM inside flash memory for running "big data" applications. DevFS also uses device-level RAM and compute capability; however, DevFS uses these resources for running a high-performance file system that applications can use.

Programmability. Willow [43] develops a system to improve SSD programmability. Willow's I/O component is offloaded to an SSD to bypass the OS and perform direct read and write operations. However, without a centralized file system, Willow also has the same general structural advantages and disadvantages of hybrid user-level file systems.

7 Conclusion

In this paper, we address the limitations of prior hybrid user-level file systems by presenting DevFS, an approach that pushes file system functionality down into device hardware. DevFS is a trusted file system inside the device that preserves metadata integrity and concurrency by exploiting hardware-level parallelism, leverages hardware power-loss control to provide low-overhead crash consistency, and coordinates with the OS to satisfy security guarantees. We address the hardware limitations of low device RAM capacity by proposing three memory reduction techniques (on-demand allocation, reverse caching, and decomposing data structures) to reduce file system memory usage by 5x(at the cost of a small performance reduction). Performance evaluation of our DevFS prototype shows more than 2x improvement in I/O throughput with direct-access to storage. We believe our DevFS prototype is a first step towards building a true direct-access file system. Several engineering challenges, such as realizing DevFS in real hardware, supporting RAID, and integrating DevFS with the FTL, remain as future work.

Acknowledgements

References

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

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

[3] Ameen Akel, Adrian M. Caulfield, Todor I. Mollov, Rajesh K. Gupta, and Steven Swanson. Onyx: A Protoype Phase Change Memory Storage Array. In *Proceedings of the 3rd USENIX conference on Hot topics in storage and file systems*, HotStorage'11, Portland, OR, 2011.

[4] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 0.91 edition, May 2015.

[5] Valerie Aurora. Log Structured File System Issues. `https://lwn.net/Articles/353411/`.

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

[7] Eric Brewer. FAST Keynote: Disks and their Cloudy Future, 2015. `https://www.usenix.org/sites/default/files/conference/protected-files/fast16_slides_brewer.pdf`.

[8] 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.

[9] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. NV-Heaps: Making Persistent Objects Fast and Safe with Next-generation, Non-volatile Memories. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, Newport Beach, California, USA, 2011.

[10] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. Better I/O Through Byte-addressable, Persistent Memory. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, Big Sky, Montana, USA, 2009.

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

[12] 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.

[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] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. Exokernel: An Operating System Architecture for Application-level Resource Management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, Copper Mountain, Colorado, USA, 1995.

[15] Mel Gorman. Understanding the Linux Virtual Memory Manager. `http://bit.ly/1n1xIhg`.

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

[17] Intel. NVM Library. `https://github.com/pmem/nvml`.

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

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

[21] 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.

[22] Kingston. Kingston power loss control. `https://www.kingston.com/us/ssd/enterprise/technical_brief/tantalum_capacitors`.

[23] Andi Kleen. Snappy Kernel Port. `https://github.com/andikleen/snappy-c`.

[24] Ivan Krasin, Tom Duerig, Neil Alldrin, Vittorio Ferrari, Sami Abu-El-Haija, Alina Kuznetsova, Hassan Rom, Jasper Uijlings, Stefan Popov, Andreas Veit, Serge Belongie, Victor Gomes, Abhinav Gupta, Chen Sun, Gal Chechik, David Cai, Zheyun Feng, Dhyanesh Narayanan, and Kevin Murphy. OpenImages: A public dataset for large-scale multi-label and multi-class image classification. *Dataset available from https://github.com/openimages*, 2017.

[25] Harendra Kumar, Yuvraj Patel, Ram Kesavan, and Sumith Makam. High Performance Metadata Integrity Protection in the WAFL Copy-on-Write File System. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 197–212, Santa Clara, CA, 2017. USENIX Association.

[26] 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.

[27] Etienne Le Sueur and Gernot Heiser. Dynamic Voltage and Frequency Scaling: The Laws of Diminishing Returns. In *Proceedings of the 2010 International Conference on Power Aware Computing and Systems*, HotPower'10, Vancouver, BC, Canada, 2010.

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

[29] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. KV-Direct: High-Performance In-Memory Key-Value Store with Programmable NIC. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, Shanghai, China, 2017.

[30] J. Liedtke. On Micro-kernel Construction. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, Copper Mountain, Colorado, USA, 1995.

[31] Chyuan Shiun Lin, Diane C. P. Smith, and John Miles Smith. The Design of a Rotating Associative Memory for Relational Database Applications. *ACM Trans. Database Syst.*, 1(1):53–65, March 1976.

[32] Justin Meza, Qiang Wu, Sanjev Kumar, and Onur Mutlu. A Large-Scale Study of Flash Memory Failures in the Field. *SIGMETRICS Perform. Eval. Rev.*, 43(1):177–190, June 2015.

[33] Micron. Micron User Space NVMe. `https://github.com/MicronSSD/unvme/`.

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

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

[36] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-consistent Applications. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, Broomfield, CO, 2014.

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

[38] Aditya Rajgarhia and Ashish Gehani. Performance and Extension of User Space File Systems. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, pages 206–213, New York, NY, USA, 2010. ACM.

[39] 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.

[40] Mendel Rosenblum and John K. Ousterhout. The Design and Implementation of a Log-structured File System. *ACM Trans. Comput. Syst.*, 10(1), February 1992.

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

[42] 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`.

[43] Sudharsan Seshadri, Mark Gahagan, Sundaram Bhaskaran, Trevor Bunker, Arup De, Yanqin Jin, Yang Liu, and Steven Swanson. Willow: A User-programmable SSD. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, Broomfield, CO, 2014.

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

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

[46] 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.

[47] Bharath Kumar Reddy Vangoor, Vasily Tarasov, and Erez Zadok. To FUSE or Not to FUSE: Performance of User-space File Systems. In *Proceedings of the 15th Usenix Conference on File and Storage Technologies*, FAST'17, Santa clara, CA, USA, 2017.

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

[49] 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.

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

[51] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: A User-level Network Interface for Parallel and Distributed Computing. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, Copper Mountain, Colorado, USA, 1995.

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

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

[54] NVM Express Workgroup. NVMExpress Specification. `://www.nvmexpress.org/resources/specifications/`.

[55] Xiaojian Wu and A. L. Narasimha Reddy. SCMFS: A File System for Storage Class Memory. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, Seattle, Washington, 2011.

[56] 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.

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

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