Designing a True Direct-Access File System with DevFS

Sudarsun Kannan*, Andrea Arpaci-Dusseau+, Remzi Arpaci-Dusseau+,
Yuangang Wang**, Jun Xu**, Gopinath Palani**
Rutgers University*, Univ. of Wisconsin-Madison+, 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 are 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. New, faster flash memory technologies such as NVMe and storage class memory (SCM) such as Intel's 3D XPoint promise to revolutionize how we access and store persistent data. State-of-the-art flash memory technologies have reduced storage-access latency to tens of microseconds compared to milliseconds in the hard-drive era [1, 2]. To fully realize the potential, a careful reconsideration of storage stack is critical. 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, even simply trapping into OS and returning from a system call substantially amplifies I/O access latencies preventing applications from exploiting storage hardware benefits.

To reduce OS-level overheads and provide direct storage access for applications, prior work such as Arrakis [2], Moneta-D [3], and others 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, 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, Google's Snappy compression, shows 22% higher throughput. The full version of this abstract appeared at USENIX FAST 2018 [4].

2. Motivation

We believe that a holy grail of modern storage research is to explore software-hardware designs that can provide applications direct access to storage without OS intervention and but without compromising fundamental storage properties, such as integrity, crash-consistency, security, and sharing. We will briefly outline the challenges and attempts by prior research to reduce the OS cost for storage access.

Storage software overheads and Prior work. Traditional OS-level file systems placed inside the OS (a trusted computing base) act as a central entity for preserving fundamental storage properties such as data and metadata integrity (correctness), crash-consistency (correctness after a failure), security (verifying every I/O operation), and sharing of file system across one or more applications. However, with the advent of ultra-fast storage devices such as NVMe and NVM, the focus has been to eliminate software layers and reduce the latency of direct-access to storage. Even simply trapping into the OS file system consumes 1-4μs which are unacceptable for modern storage with 6-10μs hardware latency. Inspired by prior work on user-level networking and OS bypass, prior approaches have attempted to move file system to untrusted userspace bypassing both control plane (e.g., security check) and data plane (e.g., read and write), but compromise one or more critical properties; this is mainly because, in file systems, the control plane and data plane are closely tied together (security check for every I/O). For example, Arrakis [2] provides direct access and must trap into OS whenever applications are sharing data. Moneta-D [3] virtualizes I/O interface (creating a per-application I/O channel) instead of storage and

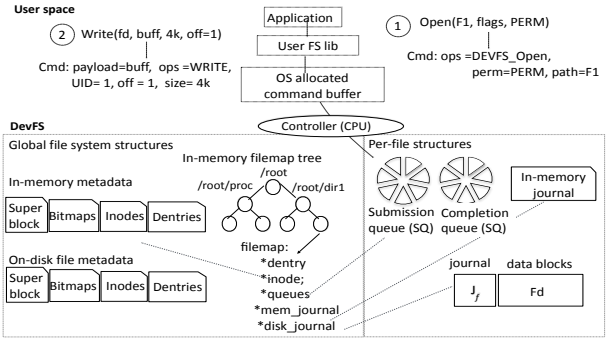

Figure 1: **DevFS high-level design.** The file system data structure is partitioned into global and per-file structures. The per-file structures are created during file setup. DevFS metadata structures are similar to other kernel-level file system.

offloads the permission check to hardware using a user-space library; however, a malicious or buggy library could avoid this step. Prior microkernel-like approaches could solve these problems, but do not provide direct-access because an application must context switch into a third party server.

3. Case for DevFS

In the pursuit of providing direct storage access and address the limitations of prior work, we design DevFS, a true direct-access file system, that embeds 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. To overcome the prototyping challenges, such as lack of commercially available programmable storage, DevFS is built as a device driver with dedicated wimpy (throttled) processors and independently managed memory. To avoid application to OS trapping and context switches, the DevFS prototype provides a userspace library built on the top of user-level NVMe driver converting application's POSIX calls to a set of extended NVMe commands and executing them using the device-level file system. Figure 1 shows the high-level design of DevFS. We next discuss key design principles.

Principle 1: Disentangle file system data structures to embrace hardware-level parallelism. Modern storage hardware controllers contain up to four CPUs, and this amount is expected to increase. Also, storage hardware such as NVMe also supports up to 64K hardware-level I/0 queues to increase I/O parallelism driven by a PCIe-based interface that can support up to 8-16 GB/s maximum throughput. To utilize the hardware-level concurrency of multiple CPU controllers and I/O queues from which a device can process I/O requests, DevFS assigns each fundamental data unit (i.e., a file) to an independent hardware resource. DevFS partitions file system into global and per-file structures (see Figure 1) and assigns per-file structures such as I/O queue and in-memory journal to a file to enable concurrent I/O across different files.

Principle 2: Guarantee file system integrity. To maintain integrity, file system metadata is always updated by the trusted DevFS. Unlike user-level file systems, there is no concern about the legitimacy of metadata content beyond file-system bugs. When a user-level library issues an I/O command, the command is added to a per-file 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. The trusted device-level file system also supports file sharing across processes without trapping into the kernel [4], and serializes shared updates using a per-file filemap lock and time stamp counters (TSC). Applications requiring strict data ordering for shared files could implement custom user-level synchronization at application-level. As shown in Figure 2, DevFS on random access benchmarks shows more than 2x higher throughput compared to the state-of-the-art kernel-level filesystem called NOVA (more results in [4]).

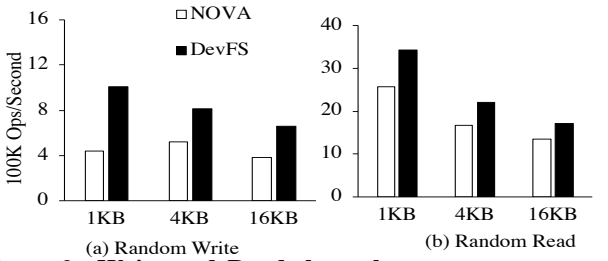

Figure 2: **Write and Read throughput.** Results for filebench random write and read micro-benchmark. X-axis varies the write size, and the file size is kept constant to 32 GB. Results show single thread performance compared to kernel-level NOVA file system.

Principle 3: Simplify crash consistency with storage hardware capacitance. Modern storage hardware internally contain power-loss-protection capacitors to flush device memory state to storage. Hence, unlike traditional OS-level file systems that rely on expensive journaling or log-structured (i.e., copy-on-write) mechanisms for crash consistency, DevFS exploits the power-loss-protection capacitors in the hardware to safely update data and metadata in-place without compromising crash consistency.

Principle 4: Reverse caching to reduce the device memory footprint. 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 active files are kept in device memory, caching inactive data structures to host memory, which we call as reverse caching. The inactive structures include files that were closed but their in-memory inodes cannot be deleted. We observe up to 5x reduction in device memory usage, and mainly from reverse caching inactive inodes.

Principle 5: Enable minimal OS-level state sharing for permission management. DevFS is a separate runtime and cannot access OS-level data structures, DevFS maintains a credential table in device-level DRAM for each host CPU; only the OS (a trusted layer) can share and update the table with credential information of a process currently scheduled to a CPU. Before processing a request, DevFS performs a simple table lookup to compare the credentials of a process currently running on the hosh CPU with the corresponding inode's permissions. As a result of state sharing between the OS and DevFS, our design can perform permission checking but without trapping into the kernel.

References

[1] J. Yang, D. B. Minturn, and F. Hady, "When Poll is Better Than Interrupt," FAST'12.

[2] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe, "Arrakis: The Operating System is the Control Plane," OSDI'14.

[3] A. M. Caulfield, T. I. Mollov, L. A. Eisner, A. De, J. Coburn, and S. Swanson, "Providing Safe, User Space Access to Fast, Solid State Disks," *SIGARCH Comput. Archit. News*, vol. 40, Mar. 2012.

[4] S. Kannan, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, Y. Wang, J. Xu, and G. Palani, "Designing a true direct-access file system with devfs," in *FAST 2018*.