

EvFS: User-level, Event-Driven File System for Non-Volatile Memory

Takeshi Yoshimura
IBM Research - Tokyo

Tatsuhiko Chiba
IBM Research - Tokyo

Hiroshi Horii
IBM Research - Tokyo

Abstract

The extremely low latency of non-volatile memory (NVM) raises issues of latency in file systems. In particular, user-kernel context switches caused by system calls and hardware interrupts become a non-negligible performance penalty. A solution to this problem is using direct-access file systems, but existing work focuses on optimizing their non-POSIX user interfaces. In this work, we propose *EvFS*, our new user-level POSIX file system that directly manages NVM in user applications. *EvFS* minimizes the latency by building a user-level storage stack and introducing asynchronous processing of complex file I/O with page cache and direct I/O. We report that the event-driven architecture of *EvFS* leads to a 700-ns latency for 64-byte non-blocking file writes and reduces the latency for 4-Kbyte blocking file I/O by 20 μ s compared to a kernel file system with journaling disabled.

1 Introduction

Non-volatile memory (NVM) brings a significant benefit to file systems in terms of read/write latency. For example, a non-volatile memory express (NVMe) SSD shows 7 μ s read and 18 μ s write latency [1]. Non-volatile main memory (NVMM) reaches between 20 and 85 ns for reads and between 10 and 1000 ns for writes depending on memory technology [22]. At this scale, however, user-kernel context switches caused by system calls and hardware interrupts become a non-negligible performance penalty [15]. User applications can minimize these overheads by leveraging user-level storage frameworks for NVMe (e.g., Storage Performance Development Kit (SPDK) [21] and NVMeDirect [11]) or direct memory-mapped I/O (mmap) for NVMM (e.g., [9, 10, 19, 20]), but those frameworks require changes in application code.

A solution to this problem is using direct access enabled file systems, but existing work focuses on optimizing their specialized user interfaces for efficient I/O processing. For example, BlobFS [3], which is a user-level file system in SPDK, exposes its optimized user-level storage stack to applications

through its non-POSIX interface. BlobFS experimentally provides POSIX interface with FUSE, but it incurs a cost of IPCs [17, 24]. Other user-level filesystems [15, 16, 18] also show that their specialized interfaces outperform POSIX interface. However, optimization of I/O processing with POSIX interfaces for NVM remains an important direction for accelerating existing user applications.

In this work, we propose *EvFS*, our new user-level POSIX file system that directly manages NVM in user applications. Our key insight is that an event-driven architecture of *EvFS* minimizes the latency of the file system since it enables a user application to invoke a non-blocking file I/O with ns-level latency for event submission. This design is inspired by the event-driven architecture of BlobFS for efficient polling-based I/O processing. We adopt their efficient execution model to the communication between users and page cache in *EvFS*. It enables *EvFS* to highly utilize the bandwidth of DRAM and NVM even with few user threads. The file system can also reduce the latency for a user's persistent requests like `fsync` by coalescing multiple writes with managed persistent ordering.

EvFS is provided as a dynamic link library to avoid changes in application code and binaries. The library exposes POSIX APIs to eliminate system calls. We built *EvFS* on top of the user-level storage stack in SPDK. The storage stack in SPDK manages both NVMe and NVMM within its block layer and is promising for the future enhancement of *EvFS*. *EvFS* contains page cache for both NVMe and NVMM, but users can still bypass the feature with open flags (e.g., `O_DIRECT` in Linux).

We also report our preliminary experimental results with FIO, a microbenchmark for file I/O. Non-blocking I/O with *EvFS* reached 700 ns for 64-B writes. For blocking I/O, *EvFS* reduced file I/O latency by 20 μ s compared to EXT4 with journaling disabled.

2 Background and Motivation

In this section, we first discuss prior user-level file systems that aim to offer high performance I/O to user applications.

Then, we summarize their problems, which our user-level file system will solve.

2.1 High performance user-level file systems

Moneta-D [7] and Arrakis [15, 16] offer their user-level file systems with hardware-assisted virtualization. Their key idea is to delegate software complexity such as security checks to hardware. They require rich storage features such as protection mechanisms, flash-backed SRAM on the device (i.e., volatile write cache in NVMe), and the use of a flash translation layer for wear leveling. However, their requirements include auxiliary features that are often limited due to cost efficiency. For example, the volatile write cache is set to a limited size or zero compared to main memory. In that case, we need to emulate the hardware cache in software to work around the limited hardware features, which can be complicated.

Aerie [18] provides a user-level file system for NVMM. It bypasses page cache because of the high performance of NVMM, but HiNFS [14] shows the effectiveness of cache line-sized page cache to solve the issue of long write latency of NVM [6, 23]. Aerie also provides `mmap` to allow user applications to enable users’ direct accesses to NVMM as done by kernel file systems for NVMM [9, 10, 14, 19, 20]. In the case of `mmap`, those file systems can remove both context switches and a storage stack from critical I/O paths. However, it also raises challenges to guarantee crash safety. NOVA-Fortis [20] resolves the challenges and reports performance degradation caused by page faults in particular workloads.

ScaleFS [5] and Strata [12] employ user-level loggers to record per-process updates and digest them into their kernel file systems. They move file cache from the kernel to user processes. As a result, cache reads/writes do not need context switches. These in-memory logs are lazily collected at `fsync`, so users can increase total throughput by coalescing multiple writes. However, their `fsync` needs context switches because it involves their kernel file systems.

FUSE enables user-level file systems running within a user process. Applications with FUSE can easily export their mount point to an OS while serving the high customizability in file system implementation. As a result, there are many use-cases of FUSE for high-performance file systems such as distributed file systems (e.g., GlusterFS and HDFS). However, FUSE incurs a cost of IPCs [17, 24], which cause a relatively large latency for NVM.

SPDK also has BlobFS [3], which is its own simple user-level file system for RocksDB enhancement. We initially extend BlobFS, but *EvFS* departs from the original design. For example, BlobFS provides POSIX APIs with FUSE and their page cache does not allow random accessing. As a result, we need to design and implement *EvFS* from scratch although we reuse some SPDK utilities in its implementation.

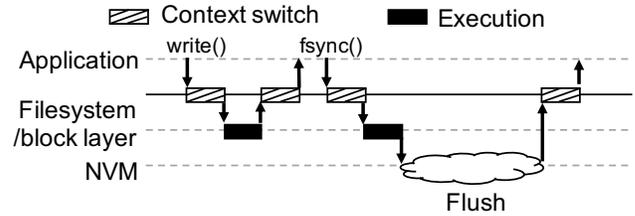


Figure 1: **Breakdown of a persistent write with a kernel file system.** We regard in-memory processing of file systems such as memory copy from a user buffer to page cache and preparing an I/O submission as “Execution.” We omit journaling from the figure.

2.2 Problem summary

In summary, prior user-level file systems aim to reduce OS involvement with modern hardware features and sophisticated storage software stacks. Prior work aimed to simplify critical I/O paths in user applications by using hardware features or `mmap`. However, the reality of NVM raises the requirement for complex but essential software features such as page cache and crash safety. The complexity is derived from the never-ending demand for crash safety and file I/O optimization. From this observation, we argue that we should explore how we hide the file system’s complexity, which potentially increases its latency, from user applications.

Figure 1 describes the breakdown of a persistent write that prior file systems aim to optimize. Kernel file systems synchronously execute the complex processing of file I/O within the kernel context of a process that requests a system call. We divide the latency of a file system from user applications into context switches and synchronous processing of file I/O.

3 *EvFS*

EvFS is a user-level POSIX file system that minimizes its latency from user applications by reducing the number of context switches and asynchronously processing all file I/O. Figure 2 shows the comparison of a traditional kernel file system and *EvFS*. We built a user-level storage stack with SPDK to reduce context switches caused by system calls and hardware interrupts. We also leveraged SPDK utilities to build an event-driven architecture for asynchronous file I/O processing.

3.1 User-level storage stack

3.1.1 Library calls

EvFS is provided as a shared library, which has the same POSIX file APIs as LIBC (e.g., `open`). By preloading the shared library before LIBC, *EvFS* can hook these APIs. We

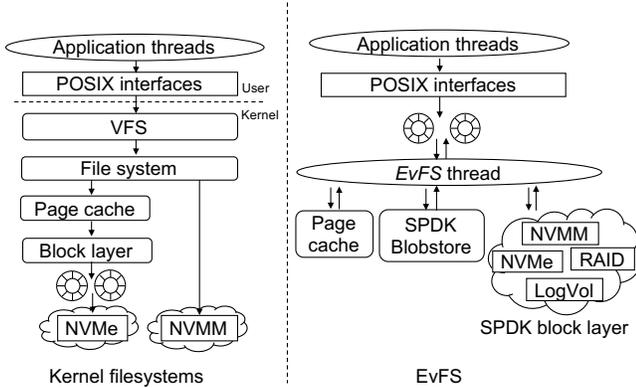


Figure 2: **Comparison of a traditional kernel file system and *EvFS*.** We use SPDK’s storage stack and utilities to implement *EvFS*. *EvFS* has poller threads for asynchronous file I/O processing and block-level I/O

eliminate system calls that cause context switches between users and the kernel by replacing these APIs into our library function calls. Those library functions operate storage read/writes via file descriptors like traditional file systems (Section 3.1.4 and 3.1.5), but invoke user-level storage stack and NVM drivers in SPDK to directly operate storage within a user process context (Section 3.1.2 and 3.1.3). This design enables us to avoid modifying existing Linux applications to enable *EvFS*.

3.1.2 User-level block layer

We architect a backend storage stack with SPDK to support various NVMe and NVMM devices in our user-level block layer. SPDK contains a user-level NVMe driver and also supports PMDK, which is an external library for NVM. We can easily apply *EvFS* to new storage by developing a new SPDK block driver. SPDK also provides a scalable thread library, event-based I/O processing, and extended storage drivers such as software RAID and logical volumes. We expect that we can easily combine and enhance modern NVM devices with this rich storage stack and *EvFS* to our future enterprise solutions.

EvFS builds a UNIX file-directory structure within an NVM namespace by using SPDK Blobstore [4]. It builds and maintains a simple logical block layout for flash storage. The superblock of Blobstore maintains metadata for each *BLOB*, which represents a chunk of block page clusters in storage. Blobstore provides interfaces for reads, writes, and resizes on a *BLOB*, and thus, we regard a *BLOB* as an inode. However, Blobstore itself has a flat structure and does not maintain any trees or directories. Thus, we emulate a UNIX-like directory by using a *BLOB* that contains the pointer to other *BLOB*s and subdirectories under the directory.

A limitation of Blobstore is that the size of a *BLOB* must be sufficient for a write. Thus, *EvFS* has to track the size of

each *BLOB* and invoke an expand request before a *BLOB* write occurs if the write exceeds the size. Each *BLOB* has customizable extended attributes that contain the name and length of a *BLOB*. Blobstore updates these metadata in memory and writes it back to storage when we explicitly invoke a synchronization API. Blobstore has the crash safety at page granularity, but *EvFS* handles the file-level consistency by using the synchronization API.

3.1.3 User-level page cache

In addition to the SPDK storage stack, we introduce user-level page cache in *EvFS* to coalesce multiple writes and reads on *BLOB*s. With page cache, user applications can achieve either DRAM-level speed at no memory pressure or NVM-level speed at high memory pressure. Note that user applications can choose direct I/O by specifying it at an open call to avoid redundant memory copies for page cache.

EvFS enables users to set the limit of memory usage in page cache. *EvFS* evicts page cache from memory to NVM if the ratio of dirty page cache reaches a configurable threshold. In the case of no memory space for page cache, *EvFS* delays a user’s request by using *event chaining* described in Section 3.2.2. *EvFS* reduces its speed to NVM-level when there is no memory space for event descriptors described in Section 3.2.1. In that case, a requesting thread sleeps until *EvFS* finishes the processing of a chained event and releases the memory for it.

3.1.4 Private mount point

EvFS creates a private mount point for the user application when the library loads. Users can specify the mounted path and a used NVM namespace through environmental variables. Thus, multiple applications can share data on NVM by specifying the same variable. However, *EvFS* currently does not allow concurrent accesses to the same NVM namespace from different user processes. NVM namespace that is mounted becomes inaccessible and invisible to the OS kernel and other concurrent processes. To enable concurrent accesses, we are planning to develop an exposed interface as done in prior work [16]. We use a hardware-based NVM namespace to partition storage that our NVMe supports. Theoretically, we can apply logical volumes in SPDK or isolated virtual volumes [13] to *EvFS* with separated partitions even if NVM does not support namespaces.

3.1.5 POSIX interface

Figure 3 lists all the APIs and open flags *EvFS* currently provides. *EvFS* associates a file descriptor (FD) to an inode, i.e., a *BLOB* at an open call. We keep the integer for the FD to be consistent for files under file systems other than *EvFS* by opening a pseudo file (e.g., /dev/null in Linux) and reusing it. By changing open flags such as O_SYNC, users

Type	APIs
File	open, read, write, pread, pwrite, lseek, close, __xstat, __lxstat, __fxstat, posix_fadvise, fsync, unlink, unlinkat, stat, access, truncate, ftruncate, creat
Directory	opendir, readdir, closedir, mkdir
Flags	O_SYNC, O_DIRECT
Thread	pthread_create, __libc_start_main

Figure 3: **Supported APIs and flags in *EvFS* under Ubuntu 18.04 LTS.** This figure excludes 64-bit variations such as `open64`. `__libc_start_main` calls the `main()` function and handles the return from it in an application. Note that exact names of APIs depend on the version of an OS and LIBC, CPU type, and other environments.

can control the blocking level and consistency for file I/O. *EvFS* enables a non-blocking write if a user does not specify `O_SYNC`. `O_DIRECT` enables direct I/O on a file so that users can choose to bypass page cache. *EvFS* will support `mmap` as well as the APIs shown in Figure 3 in the future.

We also hook APIs for thread creation (`pthread_create` and `__libc_start_main`) to associate I/O channels to individual application threads. We attach thread-local memory to application threads during the thread creation so that we can avoid frequent memory allocation to prepare events for file reads and writes. Memory allocation increases page table updates, which cause not only user-kernel context switches but also potential scalability issues [8].

3.2 Asynchronous file I/O processing

3.2.1 Executions with event descriptors

In *EvFS*, dedicated threads poll event queues and execute submitted events. Each event consists of an event descriptor that contains a callback address and arguments such as a BLOB, target offset, length, and user buffer. We follow the SPDK’s event-driven execution model for our file I/O processing.

Suppose that a user thread called a `read`. In that case, `read` in *EvFS* allocates an event descriptor with *EvFS*’s internal `read` callback, the arguments for `read`, and a semaphore for I/O completion notification. Then, the poller thread expands the event descriptor and executes *EvFS*’s internal `read` with the argument. The internal `read` also allocates an event descriptor with Blobstore’s internal `read` callback, a callback for the completion handler for Blobstore’s `read`, and the arguments for the `read`. Finally, the poller thread extracts it and executes Blobstore’s `read`.

These recursive event submissions finally invoke SPDK’s user-level NVMe device driver if the file system mounts an NVMe. The NVMe driver writes memory-mapped I/O to trigger data transfer from the NVMe with direct memory

access (DMA). Unfortunately, we cannot directly set a user buffer that an application specifies at the first `read`’s argument to the DMA target, since it is often allocated from non-page aligned heap memory such as `malloc`, which is not DMA-enabled memory. Instead, *EvFS* allocates page-aligned, DMA-enabled memory for page cache (or temporary memory for direct I/O) and uses it for the physical data transfer from/to NVMe.

After the DMA request, a poller thread periodically checks NVMe’s doorbell device register to catch physical I/O completion. If the thread detects the I/O completion, it synchronously calls the upper-level completion handlers. For example, the completion handler for Blobstore’s `read` is called, and it should post the semaphore to notify the completion to the user thread. This polling-based I/O completion handling eliminates hardware interrupts and reduces the latency of *EvFS*.

POSIX APIs allow both blocking and non-blocking file I/O. Non-blocking file I/O enables user applications to return to their thread execution immediately after an event submission. In this case, we cannot eliminate additional memory copies before the submission since the copy elimination would result in an inconsistent write if the user conducted an in-place update on the buffer. For blocking I/O, user threads need to wait for the I/O completion. In this case, we can add the address of the user buffer to the event descriptor to directly copy data from a DMA-enabled memory.

We need to carefully avoid memory allocation for event submissions since it increases system calls and page table updates. *EvFS* employs a simple memory pool for reducing the number of system calls for memory allocation.

3.2.2 Event Chaining

In cases where *EvFS* needs to handle complex event dependencies, we postpone the event until the dependent one finishes by chaining the event descriptor to the dependent object. The dependent object invokes the postponed event after it finishes. For example, a BLOB write must be called after a `resize` if the write occurs on a larger offset than the on-disk BLOB length. In this case, we chain the event to one for the dependent `resize`. *EvFS* also supports page cache, so we need to handle cases where a write happens on a page that is being written back. In that case, we postpone the write event by chaining it to one for the dependent page in page cache. We also have to release non-dirty pages if the memory pressure or a `fsync` call occurs. Users expect that all the API calls will finish before `fsync` returns, so *EvFS* chains a “barrier” event to a queue that manages on-going events and synchronously start cache eviction after all the on-going events complete.

4 Preliminary Evaluation

In this section, we show the result of our preliminary evaluation of *EvFS*. Specifically, we compare the performance

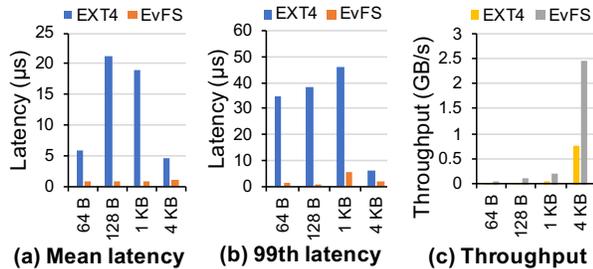


Figure 4: **Latency of non-blocking writes.** The mean and 99th-percentile latencies of non-blocking writes with different request sizes on *EvFS* and *EXT4* are shown in (a) and (b), respectively. The throughputs for each workload are shown in (c).

of *EvFS* and *EXT4*. Our benchmark is *FIO*, which executes file I/O with POSIX APIs. Reported results are the average of ten *FIO* runs. For each run, a single *FIO* thread executes 40-GB random reads or writes (no mixed reads and writes). Our evaluation uses Ubuntu 18.04 LTS on an IBM Power System AC922 machine with 160 logical CPU cores (POWER9, 3.8GHz), 1-TB DRAM, and 6.4-TB NVMe SSD [2]. We disabled readahead at the block layer and journaling for the *EXT4* to simplify our analysis. We do not show the result with memory pressure. *EvFS* shows higher latency and lower bandwidth under memory pressure as well as other file systems. Note that even if the memory is sufficient, *EvFS* evicts cache to reach the configured ratio of dirty pages as well as other file systems. We set the ratio to 20% in our experiments.

4.1 Does *EvFS* minimize latency?

Figure 4 (a) and (b) shows the mean and 99th-percentile latency of non-blocking writes on *EXT4* and *EvFS*, respectively. *EvFS* shows a lower latency than *EXT4* for all the cases. In this workload, the dominant factor of write latency for *EXT4* is the time for memory copies from the user buffer to kernel page cache and user-kernel context switches. *EvFS* also copies the user buffer to a per-thread event buffer, but we eliminated the context switches and busy-waits for racy writes on a page by event chaining. As a result, non-blocking writes with *EvFS* reached 700 ns for 64-B writes.

An advantage of the ns-level latency is that the storage bandwidth can be highly utilized with a single thread. Fig. 4 (c) shows a large throughput gain at 4-KB writes. We expect that *EvFS* can mitigate the issue of NVMM’s limited bandwidth by utilizing DRAM.

As related techniques, we also conducted similar experiments with Linux’s *LIBAIO* and *POSIXAIO*. However, they do not show better latency for I/O submissions than simple non-blocking I/O with *EXT4* because *LIBAIO* needs to call Linux APIs and *POSIXAIO* also needs the overhead of thread creations. Thus, we expect that *EvFS* can optimize

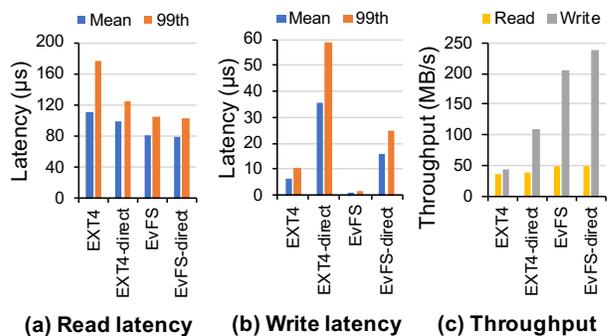


Figure 5: **Performance of direct and buffered I/O.** The mean and 99th-percentile latencies for 4-KB reads and writes on *EXT4*, *EXT4* with direct I/O (*EXT4-direct*), *EvFS*, and *EvFS* with direct I/O (*EvFS-direct*) are shown in (a) and (b), respectively. We regard a buffered I/O for a write as a pair of write and *fsync*. The throughputs for each workload are shown in (c).

asynchronous I/O.

4.2 Does *EvFS* optimize direct I/O? Does direct I/O reduce latency?

Figure 5 shows the mean and 99th-percentile latencies for direct and buffered I/O. We regard a buffered I/O for a write as a pair of write and *fsync* in this case. For both I/O types, *EvFS* enables applications to avoid user-kernel context switches for their critical I/O paths. *EvFS* reduced mean latency for direct reads and writes by 20 μs from *EXT4*. As a result, the bandwidth of *EvFS* reached 2.2x higher for writes and 1.3x higher for reads than *EXT4*. Direct I/O in *EvFS* reduced the read latency by two μs from buffered reads. This result indicates that *EvFS* is promising for future NVM with much lower latency for reads and writes. Latency sensitive applications can be optimized with *EvFS* and its direct I/O, although they should maintain a self-managed cache.

5 Conclusion

In this work, we introduced *EvFS*, our new user-level, event-driven, POSIX file system for NVM. The event-driven architecture of *EvFS* enables low latency and high throughput I/O with NVM for user applications. *EvFS* potentially mitigates the limited bandwidth of NVMM by utilizing DRAM. However, at the time of writing, *EvFS* is not a production-ready file system because it neither provides all the POSIX APIs or crash-safe properties. We believe that the event-driven architecture enables us to mitigate the latency of future *EvFS* and NVM due to their increased complexity and requirements through its 700-ns latency of non-blocking I/O and 20-μs improvements in blocking I/O.

6 Discussion

We are looking to receive feedbacks around `mmap` with user-level file systems like *EvFS*. As a file system for NVM, *EvFS* should enable user applications to use `mmap`. We are planning to hook LIBC `mmap` to provide this functionality as well as other POSIX APIs. However, we can provide two different ways for `mmap`: allowing or disallowing direct mapping of NVMM area to user applications.

The former answer is to allow direct mapping as well as other file systems, but it exposes the NVM complexity to user applications. Applications can easily degrade performance or causes inconsistency if they incorrectly specify memory barriers. Also, a slow write latency may affect the applications' performance.

Disallowing the direct mapping means *EvFS* enables user applications to map page cache instead of raw NVM area. The biggest advantage is that user applications can use the mapped area with the same manners as other storage. Applications can read and write the mapped area at DRAM speeds. Even if the size of NVM is limited, we can provide a larger area for `mmap` by using DRAM. However, this idea may raise another challenge for how we replace pages on DRAM and NVMM. Without smart replacement policies, application performance will easily degrade due to frequent page faults.

References

- [1] Intel® optane™ memory series (32gb, m.2 80mm pcie 3.0, 20nm, 3d xpoint) product specifications. <https://ark.intel.com/content/www/us/en/ark/products/series/99743/intel-optane-memory-series.html>.
- [2] Pcie3 x8 nvme 6.4 tb ssd nvme flash adapter (fc ec5e and ec5f; ccin 58fe). <https://www.ibm.com/support/knowledgecenter/8335-GTH/p9hcd/fcec5e.htm>.
- [3] Spdk: Blobstore filesystem. <https://spdk.io/doc/blobfs.html>.
- [4] Spdk: Blobstore programmer's guide. <https://spdk.io/doc/blob.html>.
- [5] Srivatsa S. Bhat, Rasha Eqbal, Austin T. Clements, M. Frans Kaashoek, and Nikolai Zeldovich. Scaling a file system to many cores using an operation log. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*, pages 69–86, 2017.
- [6] Jalil Boukhobza, Stéphane Rubini, Renhai Chen, and Zili Shao. Emerging nvm: A survey on architectural integration and research challenges. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 23(2):14:1–14:32, November 2017.
- [7] 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. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVII)*, pages 387–400, 2012.
- [8] Austin T. Clements, M. Frans Kaashoek, and Nikolai Zeldovich. Radixvm: Scalable address spaces for multi-threaded applications. In *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys '13)*, pages 211–224, 2013.
- [9] 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)*, pages 133–146, 2009.
- [10] 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, 2014.
- [11] 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 *Proceedings of the 8th USENIX Conference on Hot Topics in Storage and File Systems (HotStorage '16)*, pages 41–45, 2016.
- [12] 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)*, pages 460–477, 2017.
- [13] Mihir Nanavati, Jake Wires, and Andrew Warfield. Decibel: Isolation and sharing in disaggregated rack-scale storage. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation (NSDI '17)*, pages 17–33, 2017.
- [14] Jiaxin Ou, Jiwu Shu, and Youyou Lu. A high performance file system for non-volatile main memory. In *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys '16)*, pages 12:1–12:16, 2016.
- [15] Simon Peter, Jialin Li, Doug Woos, Irene Zhang, Dan R. K. Ports, Thomas Anderson, Arvind Krishnamurthy, and Mark Zbikowski. Towards high-performance application-level storage management. In *Proceedings of the 6th USENIX Conference on Hot Topics in Storage and File Systems (HotStorage '14)*, 2014.

- [16] 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)*, pages 1–16, 2014.
- [17] 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)*, pages 59–72, 2017.
- [18] 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)*, pages 14:1–14:14, 2014.
- [19] 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)*, pages 323–338, 2016.
- [20] Jian Xu, Lu Zhang, Amirsaman Memaripour, Akshatha Gangadharaiah, Amit Borase, Tamires Brito Da Silva, Steven Swanson, and Andy Rudoff. Nova-fortis: A fault-tolerant non-volatile main memory file system. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*, pages 478–496, 2017.
- [21] Z. Yang, J. R. Harris, B. Walker, D. Verkamp, C. Liu, C. Chang, G. Cao, J. Stern, V. Verma, and L. E. Paul. Spdk: A development kit to build high performance storage applications. In *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom '17)*, pages 154–161, 2017.
- [22] X. Zhang, D. Feng, Y. Hua, and J. Chen. Optimizing file systems with a write-efficient journaling scheme on non-volatile memory. *IEEE Transactions on Computers*, 68(3):402–413, March 2019.
- [23] X. Zhang, D. Feng, Y. Hua, and J. Chen. Optimizing file systems with a write-efficient journaling scheme on non-volatile memory. *IEEE Transactions on Computers*, 68(3):402–413, March 2019.
- [24] Yue Zhu, Teng Wang, Kathryn Mohror, Adam Moody, Kento Sato, Muhib Khan, and Weikuan Yu. Direct-fuse: Removing the middleman for high-performance fuse file system support. In *Proceedings of the 8th International Workshop on Runtime and Operating Systems for Supercomputers (ROSS '18)*, pages 6:1–6:8, 2018.