File Systems as Processes

Jing Liu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Sudarsun Kannan*
University of Wisconsin-Madison, *Rutgers University**

Abstract

We introduce *file systems as processes (FSP)*, a storage architecture designed for modern ultra-fast storage devices. By building a direct-access file system as a standalone user-level process, FSP accelerates file system development velocity without compromising essential file system properties. FSP promises to deliver raw device-level performance via highly tuned inter-process communication mechanisms; FSP also ensures protection and metadata integrity by design. To study the potential advantages and disadvantages of the FSP approach, we develop *DashFS*, a prototype user-level file system. We discuss its architecture and show preliminary performance benefits.

1 Introduction

The evolution of storage devices continues apace. For example, the access latency of a 4KB block on NVM storage devices like Optane SSD now takes less than *10 microseconds* [9, 11, 14], whereas in the last generation, the same operation costs roughly *100 microseconds* on flash-based SSDs [22]. Because storage devices are now realizing the promise of the "microsecond era" [2], the need to exploit their performance potential is growing.

These dramatic changes place great strain on existing operating system architecture. Due to legacy design decisions, based primarily on the presence of ultra-slow devices (i.e., hard drives), the traditional operating system storage stack hinders an application's ability to realize potential performance gains. For example, the switch into the kernel via system call incurs microseconds [33]; while this overhead was tolerable in the era of hard drives, it is now a dominant cost [7]. With fast devices arriving, kernel involvement is considered harmful and operating system designers can no longer ignore this overhead [2].

Researchers have recognized the problem that software is becoming bottleneck and several prior works attempt to overcome this problem, including systems such as Moneta [7], Arrakis [29], Aerie [38], Strata [24], and DevFS [19]. However, most of those systems [7, 24, 29, 38] do not completely eliminate kernel overhead, instead trapping into the kernel for control plane operations. One reason for retaining kernel mediation in the control plane is that device interaction traditionally relies on kernel drivers (like in Aerie). Furthermore, ensuring storage system metadata integrity and handling data sharing are fundamentally difficult without a centralized management component [19]. As such, library based operating system architectures cannot fully remove kernel intervention. One alternative is found in DevFS [19], a direct-access file system built into the device, but it is inherently limited by hardware constraints.

In this paper, we introduce a new storage architecture, *file systems as processes (FSP)*, which builds a true direct-access file system as a user-level process. A file system process directly manages the device, enforces permissions, and ensures metadata integrity, with nearly zero kernel involvement. This architecture is enabled by the availability of user-level NVMe device drivers [20, 45]; it is now feasible to operate fast storage hardware entirely in user space, instead of relying on software virtualization [23, 29].

FSP is a reinvention of microkernel architectures [13, 15, 25], with a new purpose in mind: high-performance storage I/O. We believe this renaissance is attractive for the following reasons.

The first reason is *developer velocity*; developing user-level code is easier than kernel hacking, allowing developers to innovate rapidly. User-level system building allows developers to bring a full range of development tools to bear during system construction, and it is no surprise that most of the systems built at large-scale companies such as Google are realized at user-level [5, 8, 12].

Second, a standalone process can ensure metadata integrity, coordinate data sharing, and guarantee crash consistency, because the process serves as part of the trusted computing base. Library-based approaches, in contrast, cannot realize these important properties, because the library is part of the application and thus cannot be trusted directly.

Third, FSP eases cluster management [6]. With care in its design, a file system process could be readily upgraded without terminating user applications.

Finally, and critically, FSP can deliver high performance. Our user-level inter-process communication (IPC) mechanism is efficient, because it is designed to avoid costly kernel crossings and leverage fast inter-core IPC. As cache-to-cache transfers cost only tens of cycles on modern multi-core processors [33], inter-core IPC incurs minimal overhead, thus avoiding same-core context switches between applications and file system processes. Consequently, the renaissance of *file systems as processes* could possibly bridge the performance gap between the "free kernel trapping" age and the ultra-fast storage device era.

The rest of this paper is structured as follows. We first discuss background and related work (§2), and then describe the FSP storage architecture and its challenges (§3). Next, we discuss the implementation of our DashFS prototype, and show a preliminary evaluation (§4). Finally, we conclude (§5).

2 Background and Related Work

To enable direct storage access for user-level applications, several file systems or I/O frameworks have been proposed. Moneta-D [7], which customizes SSDs to enable direct data access, pushes permissions checking into device registers while maintaining metadata management inside the kernel. Aerie [38] is a trusted user-level file server process. However, as Aerie primarily targets storage-class memory and the kernel is in charge of storage access, trapping overhead is not completely removed from I/O path.

Arrakis [29] relies on hardware virtualization techniques to manages data plane operations in user space, while the OS is required for handling control plane operations. As a result, all file system metadata operations (e.g., increasing the inode size) must trap into the kernel. Strata [24] aims to unify storage devices of different speeds and its foreground threads directly access the NVM device from user-level. However, Strata must trap into the kernel when files are shared, and concurrent access is managed at the OS-level with a complicated lease mechanism. Reflex [21] uses a user-level NVMe driver to provide comparable performance for remote flash, but does not provide a file abstraction and mostly works for a single application.

DevFS [19] and Willow [32] address this direct access problems at the device level. However, both of them suffer from the memory capacity and CPU constraints inside the device. Device memory is limited, making it hard for those hardware-oriented file systems to use sophisticated data structures to optimize performance.

FUSE [36] (Filesystem in Userspace) is a widely used user-space file system framework. It is significantly different from our approach because it relies upon a kernel module which leads to poor performance [31, 37]. ZUFS [16] aims to be a replacement of FUSE by leveraging "zero-copy" in I/O path for low latency, but it suffers from a similar performance penalty as FUSE due to its costly in-kernel control plane.

Our architecture is derived from classic microkernels [15]. However, microkernels have generally traded performance primarily for increased security and modularity. While researchers have reduced performance overheads [25], they were not focused upon high-performance I/O, which is a fundamental difference in our work.

Microkernel overhead mainly stems from kernel involvement in IPC [26], which includes the direct cost of kernel crossing and indirect pollution of processor structures (i.e., caches). Our approach, instead, requires minimal kernel interaction for IPC between application and file system processes, and leverages modern multi-core systems for cache efficiency. In addition, the file system process leverages the kernel's general OS functionality, such as memory management and CPU scheduling, thus directly benefiting from years of kernel innovation.

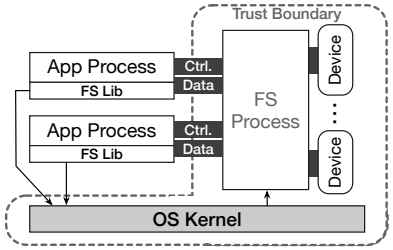


Figure 1: **File Systems as Processes Architecture**

3 Direct-Access File Systems Processes

In this section, we first present the general architecture of a user-level file system process and then discuss the challenges of realizing this new storage architecture.

3.1 Architecture

As shown in Figure 1, our file system process is a standalone user-level process, which directly operates storage devices and communicates with user applications.

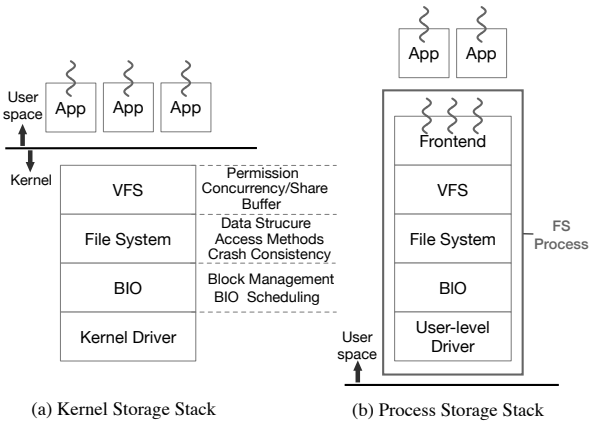
Each application has a private communication channel with the file system process; the channel is logically separated into a control plane and data plane. During the processing of file I/O requests, the kernel is not involved. The file system process is responsible for run-time file management, metadata integrity, concurrent file access support, data persistence, crash consistency, and all other features commonly found within file systems.

Because the application and file system are in different address spaces, the operating system is only involved at initialization of file-system access for a given client process. During initialization, the kernel provides a trusted mechanism for the file system process to recognize the application, as well as obtain and record the user's credential information. The file system process then sets up a secure and high-performance channel to the client process which is used in all subsequent communication.

User applications access the file system via a library which we calls *FS Lib*. *FS Lib* is in charge of providing a `POSIX`-like API, translating the API invocation into a request submission via the dedicated communication channel, and subsequently unpacking and returning a response to the calling application.

3.2 Challenges

We now discuss challenges in realizing a file system as a user-level process. The traditional storage stack has been developed for decades, and many ideas can be incorporated from previous systems [17, 30, 35, 41]. Our focus is upon the new challenges that our approach creates.



(a) Kernel Storage Stack (b) Process Storage Stack

Figure 2: **Anatomy of the Storage Stack**

Figure 2 (a) shows the traditional kernel storage stack and (b) shows our approach. The traditional approach places the entire file system within the kernel, including critical security components involved with permissions checking, and essential integrity maintenance aspects such as crash consistency [1]. In our approach, all such functionality is migrated into the file system process, which directly manages the underlying high-speed device.

Among all the challenges for building a high-performance file system as a process, we discuss five major concerns: efficient communication, concurrency, security, interrupt handling, and hardware integration.

3.2.1 Efficient Communication Channel

Our motivation of moving the storage stack into user-space is to avoid high kernel-trapping overheads. High performance inter-process communication (IPC), situated between the file system process and all application processes, serves as the foundation of this argument. Conceptually, IPC overhead must be lower than traditional storage costs, which includes trap handling, kernel crossing, and data movement across kernel and user-space.

One potential avenue of exploration is to utilize the multi-core nature of modern hardware. For example, cache-to-cache communication may only cost tens of cycles on modern multicore processors [33]. By locating a file system process on one core and a client process on another, we can avoid an expensive same-core context switch during communication, instead relying on fast cache-to-cache transfers and improving application cache locality while reducing latency. Furthermore, by utilizing shared memory segments between clients and the file system process, data movement can be highly efficient, in some cases even avoiding a copy [28, 39].

3.2.2 Frontend Threading Model

One of the major difference between the kernel storage stack and our FSP approach is that the user-created thread is no longer the vessel which performs file system operations; instead, the file system process creates threads which do work on behalf of client processes. In the diagram, this server concurrency is managed by the *Frontend* component, which is responsible for creating threads to service requests, and then managing said concurrency efficiently.

The complexity of threading in a file system process is similar to that of building server-based applications [34, 42]. However, we believe the problem is not well-examined in this storage-oriented domain, and thus the following challenges arise.

First, in contrast to the in-kernel model, the file system process must create and manage a number of extra threads, which may consequently increase scheduling contention and incur a performance penalty. Thus, great care must be given to the design of the concurrency architecture.

Second, although the device interface is asynchronous, the threads executing file system functions can become blocked due to other reasons such as locks. Thus, careful structuring of threads and the way in which they handle blocking is warranted.

Third, the way in which the file system process obtains requests from client processes is important to consider. Simple polling-based approaches could be fast but may waste cycles and consume energy; slower interrupt-driven techniques have the opposite concerns. Trade-offs based on two-phased techniques may be most promising [3]; we discuss this topic further below, as an analogous issue exists when interacting with the device itself.

3.2.3 Connecting Processes to I/O Operations

Moving down in the storage stack, the VFS layer mainly handles the runtime states and dynamic behavior of the storage stack, such as permission checking, file sharing coordination, and buffer management. However, I/O-related information is maintained as part of the process's OS state, and thus readily available to in-kernel software; in contrast, such state is hidden from a file system process. If a process's I/O state changes in a way that is meaningful to the file system process, the process must be informed by the OS in a robust manner.

Permission checking is a run-time behavior which compares the user process's credentials with permission information stored by the file system. In the process-based approach, our assumption is that the file system process is trusted as part of Trusted Computing Base (TCB) [40]. In this case, we implicitly trust the code of the file system process; thus, the major difficulty is the information gap of obtaining the proper process credentials.

To address this problem, we believe the following actions must be taken. First, the file system process needs the kernel to help establish a secure communication channel. As such, the kernel needs to have mechanisms for detecting and denying an initialization request from unqualified processes and for passing valid process

credentials to file system process securely. Second, when a process exits, the kernel is required to trigger the file system process's cleanup procedure (e.g., by releasing file handles and tearing down communication channels). Third, there are corner cases that are related to access capabilities, such as when `fork()` is called. Specifically, for a new process created by `fork()`, the kernel needs to prevent the child from accessing its parent's connection to the file system process; if the child wishes to access the file system process, it will have to establish a new connection.

Support for memory-mapped files typically requires integration with kernel page-fault handling. However, this approach is problematic for a file system process because it may not be aware of page faults that occur. One way to support user-level `mmap()` could be to implement a dedicated kernel module for the file system process that handles page faults by allocating pages and adding them to the page table, thus ensuring the same level of performance as kernel-level file systems.

3.2.4 Handling of Interrupt-free I/O Requests

The block layer, located beneath the file system in the traditional storage stack, is another layer that will differ greatly with the process-based approach. Making use of user-level storage device drivers means removing kernel's interrupt handling mechanism entirely. As a result, the file system process must poll for the completion of the I/O request instead of relying on the OS to invoke the interrupt handler upon request completion. Although polling can improve performance [20, 43], exactly when and where to poll for completion is an open question, because there is a trade-off between CPU utilization and I/O latency with respect to different workloads [27].

The problem becomes more interesting when we consider the threading model of both device polling and the *Frontend* together. The file system process can have dedicated threads for polling, which makes the management of polling simple. Alternatively, the file system process can integrate the I/O completion event into an event-driven framework in *Frontend*. The former choice, despite its simplicity, may result in performance losses in terms of latency, a phenomenon observed in mTCP [18].

The interrupt-free I/O mechanism also introduces a new challenge to buffer management in the block layer. After allocating buffers and before issuing a device I/O request, we must pin the target buffer, which is not straightforward in user-space. As for block I/O scheduling, accomplishing the same scheduling effect requires not only that we re-implement a block I/O request scheduler, but that we also re-design the scheduler to handle both issuing and polling phases of every single request.

3.2.5 Indirections Designed for Direct Access

Although the kernel storage stack is a robust and solid starting point, implementing an identical stack within the file system process is not our ultimate goal. Instead, we foresee changes to these layers and components inside each layer for several reasons.

First, modern storage devices provide new advanced features which could make certain aspects of the file system process much simpler. For example, the power-loss protection capacitors inside flash devices provide new opportunities for crash consistency [19]. Second, this multi-layer software architecture can be harmful to performance, due to the excessive costs of modularity. For example, a recent study [46] has found that linux multiqueue [4] (inside the BIO layer), developed to improve flash performance, fails to exploit device performance and incurs long latencies. Third, researchers have pointed out the defects of current storage layers. For example, the block-I/O scheduler cannot reorder I/O requests due to consistency semantics [44], thus limiting performance.

Given that the file system process must be created from scratch, perhaps there exists a new opportunity to reconsider file system architecture. By considering these concerns from the start, it is possible that a new, more efficient, and coherent whole can be realized.

4 DashFS Prototype

We implement a prototype of file system process, DashFS, to further examine the issues in realizing the FSP approach. We first describe its design principles, then briefly sketch its implementation, and finally show results from an early evaluation.

4.1 DashFS Design Principles

From the challenges listed in the last section, we have derived the following design principles:

Trust boundary: DashFS is regarded as a trusted extension of the kernel. User applications are not trusted and must be isolated from other users and the DashFS address space.

Minimal kernel interaction: Kernel involvement is heavyweight and should be avoided. The number of traps into the kernel should not scale with I/O requests.

Concurrency awareness: To provide a fully functional file system, DashFS should be designed with sharing in mind and scale gracefully under load.

4.2 Implementation

We implement DashFS by extending the user-level NVMe driver provided by SPDK [45]. Our current DashFS prototype supports regular file system operations without considering crash consistency and uses simple data structures.

The *FS Lib* library implements an interface that is similar to POSIX, including `open()`, `read()`, `write()`, `stat()` and `close()`. The *FS Lib* also offers `init()` for a user application to begin file system usage, which creates a private communication channel for each user application. The communication channel is based upon

shared memory, organized as several request buffers, data buffers, and a lockless request queue.

In the current DashFS implementation of *Frontend*, we maintain a thread pool with a configurable number of threads. Each thread handles the whole lifetime of a request. We also proactively pull requests from the request ring to avoid kernel trapping.

During initialization (`init()`), an application sends a message via UNIX domain socket to DashFS. The kernel (while processing the socket) copies the application's credential information to the DashFS address space from an in-kernel process structure, allocates a shared memory segment for the private communication channel, and returns a key for the channel to the user application. In the block I/O layer (BIO) of our prototype, each thread in *Frontend* synchronously polls for completion of a request.

4.3 Preliminary Evaluation

Considering our preference for simplicity in DashFS, we do not yet expect DashFS to outperform state-of-the-art file systems. However, we believe it is valuable to understand the basic performance implications of DashFS and start examining the challenges. In this section, we first present results of simple micro-benchmarks; we then demonstrate the effectiveness of our communication channel, which can serve as a solid basis for future development of DashFS.

All experiments are conducted on a system with Intel(R) Core(TM) i7-8700K CPU, 32G RAM and Intel Optane SSD 905P (960GB). The maximal throughput of the device is reported as 575K IOPS [10].

4.3.1 Micro-benchmark

We use a single-thread micro-benchmark to measure the basic latency of file system operations. The benchmark creates 1,000 files, writes a 4KB block to each file followed by issuing a `fsync`, and then closes the file. The page cache is cleared for each iteration. When using ext4, the application must trap into the OS at least four times for each file; hence average latency is $74.5us$ for each operation, With DashFS, the average latency decreases to $41.7us$.

For deeper insight on the overheads of ext4's kernel software stack, we measure the latency for single 4KB write requests with `O_DIRECT` enabled for a pre-allocated file and observe the average latency as $11.52us$. In contrast, using the direct-access NVMe driver reduces access latency to just $6.68us$, a 43% reduction. While it is too early to claim that DashFS outperforms kernel-level file systems, the results show the potential gains with direct-access that one could achieve after further improvements.

4.3.2 Evaluation of Communication Channel

We now investigate the raw costs of IPC. Our experiments show that sub-microsecond latency is achievable on modern hardware.

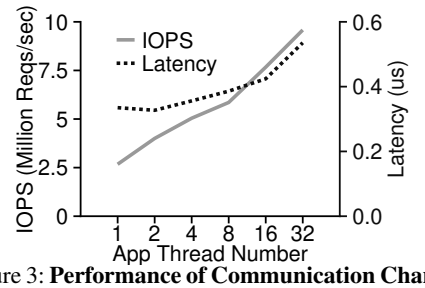


Figure 3: **Performance of Communication Channel**

Our communication channel between DashFS and each user application uses a shared lockless ring buffer. In our experiment (Figure 3), we configure the ring buffer to 64 entries and let user processes issue 4KB sequential file system write requests. We use memory as DashFS's backend to isolate the overhead of this communication channel. Each user thread continuously appends to different files with DashFS having two serving threads. The overall latency of each request includes adding the request to the request ring, dequeuing the request from the ring, copying the data from the corresponding data buffer region associated with the operation, and finally the user application checking the return value of the I/O request. This experiment demonstrates that with an increasing number of user threads, the communication channel scales well in terms of both latency and throughput.

Based on our preliminary results, the communication channel between the file system process and application threads can be regarded as efficient enough to support the performance goals of our file system process. First, for a reasonable level of user concurrency, our communication channel is able to achieve sub-microsecond latency. Second, as modern NVMe devices currently can deliver roughly one million IOPS, the communication channel is unlikely to be a throughput bottleneck [10].

5 Conclusion

Accessing fast storage without kernel involvement has been a central concern for many years. In this paper, we have argued that *file systems as processes* may be one avenue for progress in this important direction. Our study on the challenges of realizing such a file system reveals some promise, but many future hurdles must be overcome to fully reap the benefits that this new (and yet, old) idea promises.

Acknowledgements

We thank Bill Bolosky (our shepherd), the anonymous reviewers and the members of ADSL for their valuable input. This material was supported by funding from NSF grants CNS-1421033, CNS-1763810 and CNS-1838733, and DOE grant DE-SC0014935. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and may not reflect the views of NSF, DOE, or any other institutions.

References

[1] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 1.00 edition, August 2018.

[2] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. Attack of the Killer Microseconds. *Communications of the ACM*, 60(4):48–54, 2017.

[3] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI '14)*, Broomfield, CO, October 2014.

[4] Matias Bjørling, Jens Axboe, David Nellans, and Philippe Bonnet. Linux Block IO: Introducing Multi-queue SSD Access on Multi-core Systems. In *Proceedings of the 6th ACM International on Systems and Storage Conference (SYSTOR '13)*, Haifa, Israel, June 2013.

[5] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, et al. TAO: Facebook's Distributed Data Store for the Social Graph. In *Proceedings of the USENIX Annual Technical Conference (USENIX '13)*, San Jose, CA, June 2013.

[6] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. Borg, Omega, and Kubernetes. *ACM Queue*, 14(1):10, 2016.

[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 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVII)*, London, England, UK, March 2012.

[8] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A Distributed Storage System for Structured Data. *ACM Transactions on Computer Systems*, 26(2):4, 2008.

[9] Wooseong Cheong, Chanho Yoon, Seonghoon Woo, Kyuwook Han, Daehyun Kim, Chulseung Lee, Youra Choi, Shine Kim, Dongku Kang, Geunyeong Yu, et al. A Flash Memory Controller for 15μs Ultra-Low-Latency SSD Using High-Speed 3D NAND Flash with 3μs Read Time. In *2018 IEEE International Solid-State Circuits Conference-(ISSCC)*. IEEE, 2018.

[10] Intel Cooperation. Intel Optane SSD 905P Series Specification. https://ark.intel.com/content/www/us/en/ark/products/series/129835/intel-optane-ssd-905p-series.html.

[11] Annie Foong and Frank Hady. Storage As Fast As Rest of the System. In *2016 IEEE 8th International Memory Workshop (IMW)*, pages 1–4. IEEE, 2016.

[12] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, Bolton Landing, New York, October 2003.

[13] David B. Golub, Daniel P. Julin, Richard F. Rashid, Richard P. Draves, Randall W. Dean, Alessandro Forin, Joseph Barrera, Hideyuki Tokuda, Gerald Malan, and David Bohman. Microkernel Operating System Architecture and Mach. In *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, 1992.

[14] Frank T. Hady, Annie Foong, Bryan Veal, and Dan Williams. Platform Storage Performance With 3D XPoint Technology. *Proceedings of the IEEE*, 105(9):1822–1833, 2017.

[15] Per Brinch Hansen. The Nucleus of a Multiprogramming System. *Communications of the ACM*, 13(4):238–241, 1970.

[16] Boaz Harrosh. Zero Copy User-Mode FileSystem. https://lwn.net/Articles/756625/.

[17] William Jannen, Jun Yuan, Yang Zhan, Amogh Akshintala, John Esmet, Yizheng Jiao, Ankur Mittal, Prashant Pandey, Phaneendra Reddy, Leif Walsh, et al. BetrFS: A Right-Optimized Write-Optimized File System. In *Proceedings of the 13th USENIX Symposium on File and Storage Technologies (FAST '15)*, Santa Clara, CA, February 2015.

[18] EunYoung Jeong, Shinae Wood, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems. In *Proceedings of the 10th Symposium on Networked Systems Design and Implementation (NSDI '14)*, Seattle, WA, April 2014.

[19] Sudarsun Kannan, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Yuangang Wang, Jun Xu, and Gopinath Palani. Designing a True Direct-Access File System with DevFS. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST '18)*, Oakland, CA, February 2018.

[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, June 2016.

[21] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. ReFlex: Remote Flash $\approx$ Local Flash. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)*, Xi'an, China, April 2017.

[22] Sungjoon Koh, Changrim Lee, Miryeong Kwon, and Myoungsoo Jung. Exploring System Challenges of Ultra-Low Latency Solid State Drives. In *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage '18)*, Boston, MA, July 2013.

[23] Patrick Kutch. PCI-SIG SR-IOV Primer: An introduction to SR-IOV technology. *Intel application note*, pages 321211–002, 2011.

[24] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. Strata:

A Cross Media File System. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP '17)*, Shanghai, China, October 2017.

[25] Jochen Liedtke. On-μkernel Construction. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, Copper Mountain Resort, CO, December 1995.

[26] Zeyu Mi, Dingji Li, Zihan Yang, Xinran Wang, and Haibo Chen. SkyBridge: Fast and Secure Inter-Process Communication for Microkernels. In *Proceedings of the EuroSys Conference (EuroSys '19)*, Dresden, Germany, March 2019.

[27] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads. In *Proceedings of the 16th Symposium on Networked Systems Design and Implementation (NSDI '15)*, Boston, MA, February 2019.

[28] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. IO-Lite: A Unified I/O Buffering and Caching System. *ACM Transactions on Computer Systems*, 18(1):37–66, 2000.

[29] 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. *ACM Transactions on Computer Systems*, 33(4):11:1–11:30, 2016.

[30] 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. *ACM Transactions on Storage*, 13(3):19:1–19:29, 2017.

[31] 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)*, March 2010.

[32] 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 Symposium on Operating Systems Design and Implementation (OSDI '14)*, Broomfield, CO, October 2014.

[33] Livio Soares and Michael Stumm. FlexSC: Flexible System Call Scheduling with Exception-Less System Calls. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI '10)*, Vancouver, Canada, December 2010.

[34] Akshitha Sriraman and Thomas F. Wenisch. μTune: Auto-Tuned Threading for OLDI Microservices. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation (OSDI '18)*, Carlsbad, CA, October 2018.

[35] Christopher A. Stein, John H. Howard, and Margo I. Seltzer. Unifying File System Protection. In *Proceedings of the USENIX Annual Technical Conference (USENIX '01)*, Boston, MA, June 2001.

[36] Miklos Szeredi. Filesystem in Userspace. https://github.com/libfuse/libfuse.git.

[37] 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, February 2017.

[38] 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 EuroSys Conference (EuroSys '14)*, Amsterdam, The Netherlands, April 2014.

[39] Thorsten Von Eicken, Anindya Basu, Vineet Buch, and Werner Vogels. U-Net: A User-level Network Interface for Parallel and Distributed Computing. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, Copper Mountain Resort, CO, December 1995.

[40] Carsten Weinhold and Hermann Härtig. VPFS: Building a Virtual Private File System with a Small Trusted Computing Base. In *Proceedings of the EuroSys Conference (EuroSys '08)*, Glasgow, Scotland UK, March 2008.

[41] Zev Weiss, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. DenseFS: a Cache-Compact Filesystem. In *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage '18)*, Boston, MA, July 2013.

[42] Matt Welsh, David Culler, and Eric Brewer. SEDA: An Architecture for Well-conditioned, Scalable Internet Services. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, Banff, Canada, October 2001.

[43] Jisoo Yang, Dave B. Minturn, and Frank Hady. When Poll Is Better than Interrupt. In *Proceedings of the 10th USENIX Symposium on File and Storage Technologies (FAST '12)*, San Jose, CA, February 2012.

[44] Suli Yang, Tyler Harter, Nishant Agrawal, Salini Selvaraj Kowsalya, Anand Krishnamurthy, Samer Al-Kiswany, Rini T. Kaushik, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Split-level I/O Scheduling. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP '15)*, Monterey, California, October 2015.

[45] Ziye Yang, James R. Harris, Benjamin Walker, Daniel Verkamp, Changpeng Liu, Cunyin Chang, Gang Cao, Jonathan Stern, Vishal Verma, and Luse E. Paul. SPDK: A Development Kit to Build High Performance Storage Applications. In *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 154–161. IEEE, 2017.

[46] Jie Zhang, Miryeong Kwon, Donghyun Gouk, Sungjoon Koh, Changlim Lee, Mohammad Alian, et al. FlashShare: Punching Through Server Storage Stack from Kernel to Firmware for Ultra-Low Latency SSDs. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation (OSDI '18)*, Carlsbad, CA, October 2018.