# Towards High-Performance Application-Level Storage Management

Simon Peter Jialin Li Doug Woos Irene Zhang Dan R. K. Ports
Thomas Anderson Arvind Krishnamurthy Mark Zbikowski
*University of Washington*

## Abstract

We propose a radical re-architecture of the traditional operating system storage stack to move the kernel off the data path. Leveraging virtualized I/O hardware for disk and flash storage, most read and write I/O operations go directly to application code. The kernel dynamically allocates extents, manages the virtual to physical binding, and performs name translation. The benefit is to dramatically reduce the CPU overhead of storage operations while improving application flexibility.

## 1 Introduction

Modern data center application workloads are frequently characterized by many small, low-latency storage operations. Examples include both web and big data applications, where web servers, file servers, databases, persistent lock managers, and key-value stores need to interact on short timescales to execute each user request under tight time constraints.

Hardware designers have responded with both PCIe-attached flash storage and cached storage arrays to provide very low latency yet high throughput local storage architectures. For example, RAID controllers, such as the Intel RS3 family [13], contain gigabytes of battery-backed DRAM as a cache with end-to-end access latencies as low as 25 $\mu$s and bandwidth of up to 12 Gb/s. RAID cache controllers, such as the Intel RCS25ZB family [14], provide up to 1TB of NAND-flash-backed cache memory with similar access latencies and bandwidths. PCIe-attached flash storage adapters, such as the Fusion-IO ioDrive2, report hardware access latencies as low as 15 $\mu$s [10].

In this paper, we propose a corresponding revolution in the operating system stack to take maximum advantage of these opportunities. Otherwise, as we detail in Section 2, the operating system is at risk of becoming the critical performance bottleneck. Even as I/O performance improves at a staggering rate, CPU frequencies have effectively stalled. Inefficiencies in the operating system storage stack which could be safely ignored up to now, are becoming predominant.

Previous work has relied on limited-use APIs and compromised semantics to improve performance. For example, the Linux `sendfile` system call [17] allows data to be copied directly from the disk buffer into the network buffer, without traversing the application. In the Redis persistent key-value store, flushes to stable storage are delayed to amortize storage subsystem overheads [6].

We step back to ask: how should we architect the operating system for maximal storage I/O performance? We take the radical step of moving the operating system kernel completely "off the data path". (This contrasts with Linux sendfile, which moves the *application* off the data path.) Rather, the kernel establishes the binding between applications and their storage; applications then access the storage directly without kernel involvement.

A key enabler is recent hardware I/O virtualization technology, such as single-root I/O virtualization (SR-IOV) [16]. The traditional slow data path to storage is made even worse by the widespread deployment of virtual machines, since both the hypervisor and the guest operating systems must be traversed on every operation. SR-IOV and the IOMMU make it possible to bypass the hypervisor to deliver I/O events directly to the guest operating system. SR-IOV-capable PCIe storage adapter cards ("physical functions") are able to dynamically create virtual copies ("virtual functions") of themselves on the PCIe interconnect. Each virtual function resembles a PCIe storage adapter that can be directly mapped into virtual machines and programmed as if it was a regular physical device, with a normal device driver and an unchanged OS I/O stack.

We can extend this model to enable applications, and not just operating systems, to directly manage their storage. This bypasses both the host and guest operating system layers. Applications have direct user-level access to storage adapter virtual functions, allowing most reads and writes to bypass the OS kernel. The operating system kernel manages the mapping of virtual storage extents to physical disk blocks, the allocation and deallocation of extents, data protection, and name translation.

This approach combines low overhead with greatly improved flexibility for applications to customize storage management to their own needs. Speedups can be gained by eliminating generic kernel code for multiplexing, security, copying, and unnecessary features that are not used by a particular application. For example, a key-value store can implement a persistent hashtable for storing key-value pairs directly, without having to rely on a hierarchical inode space to locate corresponding disk blocks. Cache management and pre-fetching can be more efficient, as the application has a better idea of its workload than a generic OS kernel storage stack. The inflexibility of existing kernel storage stacks has been lamented many times in the past [24].

A key feature of our approach is that the user naming and protection model is unchanged. Today's file systems inappropriately bind naming and implementation together. A disk partition formatted for NTFS uses that file format for every file in that subtree, whether or not that is the most appropriate. The result is to force file system designers into the most general-purpose designs imaginable.

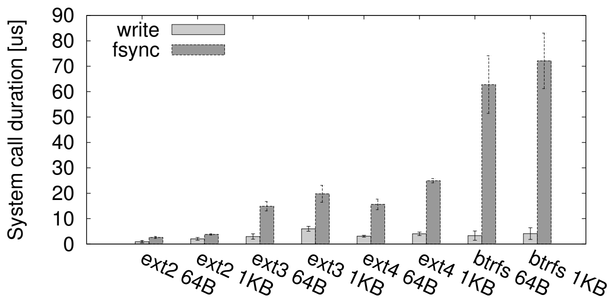

Figure 1: Overhead in $\mu$s of various Linux filesystem implementations, when conducting small, persistent writes.

Our approach instead is to associate each application with the directories and files it manages, as if they were dynamically mounted network file systems. This way, an application is free to organize its files, and its directories, however it chooses. Other applications can still read those files by indirecting through the kernel, which hands the directory or read request to the appropriate application.

To present the case for a user-level storage architecture, we make the following contributions: We introduce an ideal set of hardware features for the efficient realization of user-level storage architectures (Section 3). Building upon these features, we present the initial design of an OS storage architecture that enables the construction of user-level storage stacks (Section 4). We then present a number of application use cases for such user-level storage stacks (Section 5).

This storage proposal forms part of our larger Arrakis project that sets out to build a high-performance server operating system by bypassing the OS kernel for I/O operations in the common case [20]. Our experience shows that we can expect other I/O stacks, such as that of the network, to achieve comparable latency and throughput benefits [21].

## 2 Inefficiencies in Storage Stacks

To illustrate the overhead of today's OS storage stacks when conducting small writes with persistence requirements, we conduct an experiment, where we execute small write operations immediately followed by an fsync[1] system call in a tight loop of 10,000 iterations and measure each operation's execution latency. For this experiment, the file system is stored on a RAM disk, so the measured latencies represent purely CPU overhead. The experiment was conducted on a 6-core 2.2GHz Sandy Bridge system with an Intel RS3DC040 RAID storage controller, running Linux 3.8.

Figure 1 shows the kernel storage stack overheads for differently sized writes under various filesystems. A write followed by fsync of 1KB via the second extended (ext2) filesystem has a total latency of 6 $\mu$s. More advanced filesystems incorporate functionality like journaling to avoid metadata corruption on system crashes, but this functionality comes at a cost: the same write via the ext3 and ext4 filesystems has a latency of up to 29 $\mu$s. For the modern btrfs, average latency is as high as 76 $\mu$s.

[1]We also tried fdatasync, with negligible difference in latency.

Historically, these latency figures might not raise concern, being dwarfed by the 5-10 ms latency of each disk seek. However, modern storage systems can offer lower latency: consider both the near-ubiquity of caching RAID controllers and fast persistent storage devices based on emerging non-volatile memory technologies. In this environment, OS storage stack overhead becomes a major factor. Compared to the available latencies of down to 15 $\mu$s offered by the storage hardware, OS storage stack overheads are high, between 40% and 2x for the extended filesystems, depending on journal use, and up to 5x for btrfs.

These overheads are in part due to general purpose kernel code accumulating bloat in order to accomodate a wide variety of application needs. Despite this generality, applications often layer their own storage techniques on top of the features already offered by filesystems. The resulting duplication of functionality leads to even higher latency. For example, the Redis key-value store can ensure consistency by writing a journal of operations to durable storage via the filesystem. In many cases, the filesystem conducts its own journaling, too.

Finally, traditional operating systems conflate the concepts of file and block naming and their layout. For example, virtual file system (VFS) APIs are almost always hierarchical in today's OSes. Filesystem implementations and applications are forced to follow this API. However, hierarchical naming might not always fit the storage requirements. For example, key-value stores typically locate data via a hashtable, instead of traversing a hierarchical namespace. Similarly, email servers locate mail by header or body text, not by mailbox name or identifier. Supporting a hierarchical namespace comes at additional cost, both in terms of additional levels of indirection [24] and lock contention on common directories [4].

## 3 Hardware Model

An efficient implementation of a user-level storage stack requires hardware support. We are aiming our design at an idealized storage hardware architecture that captures the functionality required to implement in hardware the data plane operations of a traditional kernel-level storage stack. Much of this functionality is available in existing hardware controllers with virtualization support. However, certain aspects of our model are not yet available. Thus, our proposal aims to provide guidance for future storage hardware architecture designs.

In our model, storage controllers can be virtualized. Each controller exposes multiple *virtual storage interface controllers* (VSICs) to the operating system. Each VSIC provides independent storage command interfaces which are multiplexed by the hardware. For example, each VSIC has at least one command queue (e.g., of SCSI or ATA command format) that can be managed directly within user-space without kernel mediation, including separate interrupts for each queue to signal command completion. In addition, it would be beneficial to be able to specify a bandwidth allocation for each VSIC to limit an application's I/O bandwidth, but this is not required. The IOMMU performs address translation and protection of DMA commands executed on behalf of the user-space application.

We assume that storage devices will continue to be accessible via a request-based DMA protocol, although some proposals for NVRAM technology have called for it to be directly incorporated into the memory interconnect. We anticipate that DMA-based protocols will remain available, given the need to provide compatibility with existing devices, as well as to provide an asynchronous interface to devices with higher latency than DRAM.

In addition to independent virtual command interfaces, storage controllers need to provide an interface via their physical function to create and delete *virtual storage areas* (VSAs), map them to extents of physical drives, and associate them with VSICs. It might be sufficient if each VSIC supports only one VSA, but in some cases support for multiple different VSAs can have benefits. For example, parallel writes to different storage devices, such as multiple flash chips, may improve performance. Multiple VSAs can also be useful to applications for organizational purposes. Allowing application-level control over parallel writes can improve performance further due to the application's intricate knowledge of the workload.

Today's storage controllers have most of the technology needed to provide the interface we describe. For example, RAID adapters have a translation layer that is able to provide virtual disks out of physical disk stripes and SSDs use a flash translation layer for wear-leveling. Furthermore, storage host-bus adapters (HBAs) have supported SR-IOV technology for virtualization [18, 19], which allows them to expose multiple VSICs. Only the required protection mechanism is missing. We anticipate VSAs to be allocated in large chunks (tens of megabytes to gigabytes) and thus hardware translation layers can be coarse-grained.

## 4 OS Model

To allow applications unmediated access to VSICs, we divide the OS storage stack into a control and a data plane. Under normal operation, applications interact directly with the hardware using the data plane interface. The control plane interface, which is mediated by the kernel, is used to set up VSICs and VSAs.

### 4.1 Control Plane Interface

The control plane interface is used to create and delete VSICs, associate them with applications, create, modify, and delete virtual storage areas (VSAs), establish a mapping between VSAs and physical drive extents, and to setup bandwidth limits for VSICs.

Creation and deletion of VSICs can simply be accomplished by corresponding API calls from the applications in question. The newly created VSIC is associated with the requesting. The creation of multiple VSICs per application can be useful if the application makes use of multiple protection domains.

As long as physical storage space is available, the interface should allow creating new VSAs and extending existing ones. Deleting an existing VSA is always possible and shrinking it as long as its size is larger than zero. Shrink and delete operations permanently delete the data contained in the VSA. This can be done asynchronously by the control plane upon deletion/shrink. Extending and shrinking might be done in increments of a fixed size, whichever is most efficient given the hardware, and the (de-)allocation of physical storage blocks is also carried out by the control plane. As we envision VSAs to be modified at an infrequent pace and by large amounts, the overhead of these operations does not significantly affect application performance.

VSAs are protected by a security capability reference that is granted upon VSA creation and that needs to be held by the application wishing access to an existing VSA. The capability is the sole reference to an existing VSA and is stored in a capability storage area in the control plane. The control plane is responsible for remembering an association between a VSA and an application and to grant the corresponding capability back to that application. A traditional kernel-mediated filesystem can be used to store application binaries, capabilities, and their association, and they can be named like regular files. Capabilities may be passed to other applications, so they may access the corresponding VSA. Concurrent access is handled at application-level.

For each VSA, the control plane maintains a mapping of virtual storage blocks to physical ones and programs the hardware accordingly. As long as the storage hardware allows mappings to be made at a block granularity, a regular physical memory allocation algorithm can be used, without having to worry about fragmentation. Maintaining head locality for hard disks is outside of the scope of our work.

### 4.2 Data Plane Interface

At the lowest level, the data plane API is the hardware interface of the VSIC: command queues and interrupts. To make programming of the storage hardware more convenient, we also provide an abstract API in an application library. On top of this, the familiar POSIX calls can be provided for backwards compatibility.

The abstract API shall provide a set of commands to asynchronously read and write at any offset and of arbitrary size (subject to block granularity if more efficient) in a VSA via a specified command queue in the associated VSIC. To do so, the caller provides an array of virtual memory ranges (address and size) in RAM to be read/written, the VSA identifier, queue number, and matching array of ranges (offset and size) within the VSA. The API implementation makes sure to enqueue the corresponding commands to the VSIC, coalescing and reordering commands only if this makes sense to the underlying media. I/O completion events are reported via a callback registered by another API call.

Interaction with the storage hardware is designed for maximizing performance from the perspective of both latency and throughput. This design results in a storage stack that decouples hardware from software as much as possible using command queues as a buffer, maximizing throughput and achieving low latency by delivering interrupts from hardware directly to user programs via hardware virtualization.

## 5 Use Cases

A number of applications can benefit from a user-level storage architecture. We survey three of them in this section and, for each, describe effective new storage techniques that become possible using the control and data plane APIs proposed in the previous section.

### 5.1 File Server

File servers, such as the NFS daemon, are typically run within the kernel for performance reasons. Running file servers at user-level, however, has security and flexibility benefits. We describe how a flexible, secure, high-performance NFS service can be built using our storage architecture, by taking advantage of control over file layout, application-controlled caching, and fast remote procedure calls (RPCs).

Many NFS workloads are dominated by file metadata operations. We can store metadata in its external data representation (XDR) and (taking advantage of our application-level access to the network) pre-compile remote procedure call responses for the underlying network transport (e.g., Ethernet). This allows the server to fulfill metadata requests with a single read and send operation. Finally, the low write latencies provided by direct VSIC access allow us to persist the server-side reply cache after each operation for improved consistency in the face of network and server errors (cf. §2.10.6.5 in [23]).

Protection for each user connection can be achieved by storing each user's data in a separate VSA and forking the server for protection of each active VSA. Having full control over the underlying file system and its buffer cache, we can deploy a specialized file layout such as WAFL [12] to provide performance and recoverability within each VSA. We can also easily support client cache control hints, e.g. via the `IO_ADVISE` operation proposed in NFSv4.2 [11]. In addition, a user-level NFS server can keep per-client file access records and prefetch based on this information. Implementing these techniques is difficult at best with a layered architecture, where the file system and buffer cache are managed by the OS kernel.

### 5.2 Mail Relay

Mail relays typically organize and store mail in individual files in a single directory. However, due to this conflation of naming and storage, concurrent file metadata operations are protected by a directory lock in many standard file systems, which stifles scalability in multicore systems [4]. In addition, blocks of larger files are typically referenced indirectly, meaning that storing all mail in a single file would require additional read operations.

Using our architecture, we can develop a scalable, persistent mail queue. Leveraging techniques from scalable memory allocators [22], we can pre-allocate queue entries of different sizes (e.g., 1KB, 4KB, 16KB) in a contiguous VSA region for small mail (e.g., without attachments), which can be filled directly upon mail arrival. Each worker thread can be assigned its own set of queue entries without requiring synchronization. Large mail can be stored in extents and referenced by offset and size from queue entries.

### 5.3 Persistent Key-Value Store

Using our storage architecture, we can develop a truly persistent key-value store that logs each operation to persistent storage with low latency, while keeping an in-memory cache. Upon a write operation, a corresponding log entry can refer directly to the payload from the network. The remaining write latency can be masked by concurrently updating the memory-resident cache and preparing the acknowledgement, while waiting for the asynchronous I/O completion event upon which the acknowledgement is sent out on the network. Furthermore, log entries can be formatted in a way that matches the block size and alignment of the storage medium and multi-block entries can be written out in order, to keep consistency.

In contrast, current server operating systems do not support asynchronous writes to stable storage. For example, the Linux kernel is free to cache write buffers in RAM until an fsync system call is invoked. fsync blocks the caller and all further calls regarding the same file descriptor until all cached buffers are flushed to disk. Masking this time requires complex multi-threaded application designs. Other kernel-mediated asynchronous I/O approaches still require a round-trip through the kernel to deliver I/O events, including a copy between user and kernel buffers for I/O data [8].

Finally, the kernel amortizes blocks when flushing and might write half-records when they do not match the storage medium's block size, or might write records out of order. This complicates consistency if the system crashes while a write is in progress.

## 6 Related Work

There has long been a tension between applications that desire customization for performance and file systems that want to cater to the widest range range of applications. This tension has led to the development of operating system interfaces like `fadvise` that give the application greater control over the file system, or `sendfile` that speed up common interactions between the application and the operating system. As storage technologies become faster relative to CPU speeds, this tension will only increase. For example, recent work like Bankshot [3] proposes bypassing the kernel to eliminate cache hit overheads to NVRAM.

We are not the first to observe that many applications would benefit from a customized storage stack. Library operating systems, like Exokernel [9], Nemesis [1] and SPIN [2], supported user-level I/O to provide flexibility and better performance for applications. Unlike these systems, our approach utilizes virtualization hardware to safely give applications direct access to the raw storage interface. Closest to our vision is a proposal to set up a virtual I/O path per file, rather than per application as in our proposal, so its benefits are restricted to large files [25].

Similarly, recent work has focused on reducing the overheads imposed by traditional filesystems and block device drivers to persistent memory (PM). DFS [15] and PMFS [7] are filesystems designed for these devices. DFS relies on the flash storage layer for functionality traditionally implemented in the OS, such as block allocation. PMFS exploits the byte-addressability of PM, avoiding the block layer. Both DFS and PMFS are implemented as kernel-level filesystems, exposing POSIX interfaces. They focus on optimizing filesystem and device driver design for specific technologies, while we investigate how to allow applications fast, customized device access.

Moneta-D [5] is a hardware and software platform for fast, user-level I/O that bypasses the kernel by introducing a per-process storage interface called a virtual channel. Permissions are cached in hardware and checked on every access. Moneta-D differs from our proposal in its treatment of file system metadata, which Moneta-D centrally manages within the kernel by using a traditional file system. In contrast, our proposal decouples naming from storage management and thus allows applications to fully bypass the kernel through per-process management of all aspects of the storage system, including metadata and access control.

Aerie [26] proposes an architecture in which multiple processes communicate with a trusted user-space filesystem service to modify file metadata and perform lock operations, while directly accessing the hardware for reads and data-only writes. Our proposal provides more flexibility than Aerie, since storage solutions can integrate metadata tightly with applications rather than provide it in a trusted shared service, allowing for further optimizations, such as early allocation of metadata that can be integrated with application-level data structures, without having to RPC to a shared service.

## 7 Conclusion

We argue that next generation high-throughput, low-latency data center storage architectures will require a new approach to operating system kernel design. Instead of mediating all storage I/O through the kernel, we propose to separate file system naming from file system implementation. Applications dynamically allocate virtual storage extents; a runtime library in each application implements directory and file system operations for that application, with direct access to the storage hardware.

## References

[1] P. R. Barham. *Devices in a Multi-Service Operating System*. PhD thesis, Churchill College, University of Cambridge, July 1996.

[2] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, safety and performance in the SPIN operating system. In *SOSP*, 1995.

[3] M. S. Bhaskaran, J. Xu, and S. Swanson. Bankshot: Caching slow storage in fast non-volatile memory. In *INFLOW*, 2013.

[4] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich. An analysis of linux scalability to many cores. In *OSDI*, 2010.

[5] 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. ASPLOS, 2012.

[6] Citrusbyte. *Redis Persistence*, Mar. 2014. `http://redis.io/topics/persistence`.

[7] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson. System software for persistent memory. In *EuroSys*, 2014.

[8] K. Elmeleegy, A. Chanda, A. L. Cox, and W. Zwaenepoel. Lazy asynchronous I/O for event-driven servers. In *USENIX ATC*, 2004.

[9] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. Exokernel: An operating system architecture for application-level resource management. In *SOSP*, 1995.

[10] Fusion-IO. *ioDrive2 and ioDrive2 Duo Multi Level Cell*, 2014. Product Datasheet.

[11] T. Haynes. NFS version 4 minor version 2. RFC proposal version 21, NetApp, Feb. 2014.

[12] D. Hitz, J. Lau, and M. Malcolm. File system design for an NFS file server appliance. In *USENIX WTEC*, 1994.

[13] Intel Corporation. *Intel RAID Controllers RS3DC080 and RS3DC040*, Aug. 2013. Product Brief.

[14] Intel Corporation. *Intel RAID SSD Cache Controller RCS25ZB040*, Dec. 2013. Product Brief.

[15] W. K. Josephson, L. A. Bongo, K. Li, and D. Flynn. DFS: A file system for virtualized flash storage. *Trans. Storage*, 6(3):14:1–14:25, Sept. 2010.

[16] P. Kutch. PCI-SIG SR-IOV primer: An introduction to SR-IOV technology. *Intel application note*, 321211–002, Jan. 2011.

[17] Linux Programmer's Manual. *sendfile(2)*, Sept. 2011.

[18] LSI Corporation. *LSISAS2308 PCI Express to 8-Port 6Gb/s SAS/SATA Controller*, Feb. 2010. Product Brief.

[19] LSI Corporation. *LSISAS3008 PCI Express to 8-Port 12Gb/s SAS/SATA Controller*, Feb. 2014. Product Brief.

[20] S. Peter and T. Anderson. Arrakis: A case for the end of the empire. In *HotOS*, 2013.

[21] S. Peter, J. Li, I. Zhang, D. R. K. Ports, A. Krishnamurthy, T. Anderson, and T. Roscoe. Arrakis: The operating system is the control plane. Technical Report UW-CSE-13-10-01, University of Washington, Oct. 2013.

[22] S. Schneider, C. D. Antonopoulos, and D. S. Nikolopoulos. Scalable locality-conscious multithreaded memory allocation. In *ISMM*, 2006.

[23] S. Shepler, M. Eisler, and D. Noveck. Network file system (NFS) version 4 minor version 1 protocol. RFC 5661, NetApp, 2010.

[24] M. Stonebraker. Operating system support for database management. *Communications of the ACM*, 24(7):412–418, July 1981.

[25] A. Trivedi, P. Stuedi, B. Metzler, R. Pletka, B. G. Fitch, and T. R. Gross. Unified high-performance I/O: One stack to rule them all. In *HotOS*, 2013.

[26] H. Volos, S. Nalli, S. Panneerselvam, V. Varadarajan, P. Saxena, and M. M. Swift. Aerie: Flexible file-system interfaces to storage-class memory. In *EuroSys*, 2014.