Providing Safe, User Space Access to Fast, Solid State Disks

Adrian M. Caulfield Todor I. Mollov Louis Alex Eisner
Arup De Joel Coburn Steven Swanson

Computer Science and Engineering Department
University of California, San Diego
{acaulfie,leisner,arde,jdcoburn,swanson}@cs.ucsd.edu

Abstract

Emerging fast, non-volatile memories (e.g., phase change memories, spin-torque MRAMs, and the memristor) reduce storage access latencies by an order of magnitude compared to state-of-the-art flash-based SSDs. This improved performance means that software overheads that had little impact on the performance of flash-based systems can present serious bottlenecks in systems that incorporate these new technologies. We describe a novel storage hardware and software architecture that nearly eliminates two sources of this overhead: Entering the kernel and performing file system permission checks. The new architecture provides a private, virtualized interface for each process and moves file system protection checks into hardware. As a result, applications can access file data without operating system intervention, eliminating OS and file system costs entirely for most accesses. We describe the support the system provides for fast permission checks in hardware, our approach to notifying applications when requests complete, and the small, easily portable changes required in the file system to support the new access model. Existing applications require no modification to use the new interface. We evaluate the performance of the system using a suite of microbenchmarks and database workloads and show that the new interface improves latency and bandwidth for 4 KB writes by 60% and 7.2×, respectively, OLTP database transaction throughput by up to 2.0×, and Berkeley-DB throughput by up to 5.7×. A streamlined asynchronous file IO interface built to fully utilize the new interface enables an additional 5.5× increase in throughput with 1 thread and 2.8× increase in efficiency for 512 B transfers.

Categories and Subject Descriptors D.4.2 [*Operating Systems*]: Storage Management—Secondary storage

General Terms Design, Measurement, Performance

Keywords File Systems, IO Performance, Non-Volatile Memory, Storage Systems, Virtualization

1. Introduction

Emerging fast, non-volatile technologies such as phase change, spin-torque transfer, and memristor memories make it possible to build storage devices that are orders of magnitude faster than even the fastest flash-based solid-state disks (SSDs). These technologies will rewrite the rules governing how storage hardware and software interact to determine overall storage system performance. In particular, software overheads that used to contribute marginally to latency (because storage hardware was slow) will potentially squander the performance that these new memories can provide.

ASPLOS'12, March 3–7, 2012, London, England, UK.

Recent work describing Moneta [5], a fast, next-generation storage architecture, showed that optimizing the existing IO stack and tuning the hardware/software interface can reduce software overheads by up to 62% and increase sustained bandwidth for small accesses by up to 19×. However, even with these reduced overheads, IO processing places large demands on the system's compute resources – sustaining peak performance on Moneta for 4 KB requests requires the dedicated attention of 9 Nehalem thread contexts. Entering the kernel, performing file system checks, and returning to user space account for 30% (8 μs) of the latency of a 4 KB request. Together those costs reduce sustained throughput by 85%. However, simply removing those layers is not possible because they provide essential management and protection mechanisms.

This paper describes extensions to Moneta that remove these costs by transparently bypassing the operating and file systems while preserving their management and protection functions. The extensions provide each process with a private interface, or *channel*, to Moneta. Unlike other systems that virtualize an entire device (e.g., a graphics card or network card), Moneta's channels are virtual *interfaces* to a single device. Each process uses its channel to access Moneta directly, without interacting with the operating system for most accesses. Hardware permission verification replaces permission checks in the operating system, preserving all of the protection guarantees the operating system normally provides.

To utilize channel-based IO, unmodified applications link with an untrusted user space library that intercepts IO system calls and performs the operations directly. The library works with a trusted driver and a slightly modified file system to extract file protection information and transfer it to Moneta. The library presents a standard POSIX interface.

We refer to the new system as *Moneta Direct* (*Moneta-D*) throughout the remainder of the paper. Moneta-D's unique features eliminate file system overheads and restructure the operating system storage stack to efficiently support direct, user space access to fast non-volatile storage arrays:

- Moneta-D removes trusted code and the associated overheads from most accesses. Only requests that affect meta-data need to enter the operating system.
- Moneta-D provides a fast hardware mechanism for enforcing the operating and file system's data protection and allocation policy.
- Moneta-D trades off between CPU overhead and performance by using different interrupt forwarding strategies depending on access size and application requirements.
- Moneta-D provides an asynchronous software interface, allowing applications to leverage its inherently asynchronous hardware interface to increase performance and CPU efficiency.

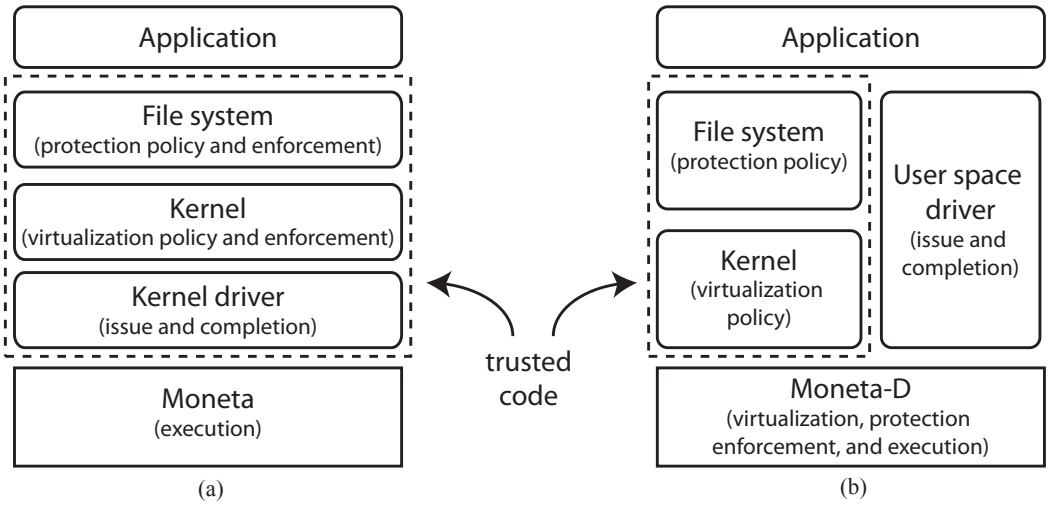


Figure 1. Changes to the high-level system architecture The operating and file systems protect storage devices like Moneta (a) by requiring a system call on every access. In Moneta-D (b), the OS and file system are only responsible for setting protection and sharing policy. Moneta-D enforces that policy in hardware, eliminating the need for most system calls and file system accesses. A virtualized hardware interface allows for an untrusted driver.

We evaluate options for key design decisions in Moneta-D and measure its performance under a collection of database workloads and direct file access benchmarks. Our results show that Moneta-D improves performance for simple database operations by between 2.6× and 5.7×. For full SQL server workloads, performance improves by between 1.1× and 2.0×. For file access benchmarks, our results show that Moneta-D can reduce the latency for a 512 byte read operation by 64% (to 4.1 μs) relative to the original Moneta design. The reduction in latency leads to a 14.8× increase in bandwidth for 512 byte requests in the presence of a file system, allowing the storage array to sustain up to 1.8 million 512 byte IO operations per second. With a single thread, Moneta-D's asynchronous IO interface improves performance by 5.5× for 4 KB accesses compared to synchronous IO. Asynchronous IO also improves efficiency by up to 2.8×, reducing CPU utilization and saving power.

The remainder of this paper is organized as follows. Section 2 describes the baseline Moneta storage array and provides an overview of the changes required to virtualize it. Section 3 places our work in the context of other research efforts. Section 4 describes Moneta-D's architectural support for virtualization. Section 5 evaluates the system. Finally, Section 6 concludes.

2. System overview

Moneta-D's goal is to remove operating system and file system overheads from accesses while maintaining the strong protection guarantees that these software layers provide. The resulting system should be scalable in that many applications should be able to access Moneta-D concurrently without adversely affecting performance. Furthermore, it should not be necessary to modify the applications to take advantage of Moneta-D.

Figures 1(a) and (b) summarize the changes Moneta-D makes to the hardware and software components of the original Moneta system. In Moneta (Figure 1(a)), all interactions with the hardware occur via the operating system and file system. Together, they set the policy for sharing the device and protecting the data it contains. They also enforce that policy by performing checks on each access. Performing those checks requires the file system to be trusted code. The driver is trusted since it accesses a shared, non-virtualized hardware resource.

Figure 1(b) shows the revised organization that Moneta-D uses. The kernel and the file system remain responsible for making policy decisions that control access to data, but the Moneta-D hardware enforces that policy. The hardware exposes a set of virtual *channels* that each provide a single process with access to storage. The kernel manages these channels, assigns them to processes, and maintains protection information associated with each channel. Since the hardware enforces protection and each process has a private channel, there is no need for a privileged driver. Instead, applications access their channels via an untrusted driver library, avoiding system call overheads.

Our intent is that this new architecture be the default mechanism for file access rather than a specialized interface for high-performance applications. To make it feasible for all applications running on a system to use the interface, Moneta-D supports a large number of virtual channels. This decision has forced us to minimize the cost of virtualization.

Below, we describe the channel interface, the user space driver library, and discuss the interactions between our system and the file system.

2.1 Channels

A channel is a virtual interface to the storage array. Each channel provides all the facilities necessary for a process to access data in the array and for the kernel to restrict access to only files that the process has successfully opened.

A channel has two interfaces, a privileged interface for configuration by the kernel and an unprivileged interface for application access. The privileged interface comprises a set of control registers that let the kernel manage the channel and install permission information. The unprivileged interface has three components: 1) a set of user registers that the user space library uses to access array data, 2) a set of tags that distinguish between outstanding requests on the channel, and 3) a DMA buffer.

Below, we describe how a process and the kernel use their respective interfaces to initialize a channel, access data, and manage permission information. Section 4 describes these mechanisms in more detail.

Channel initialization The user space driver library initializes a channel by opening the storage device's file in `/dev/` and `mmap()`ing several pages into its address space. Mapping these pages allocates a channel for the process and grants it access to the hardware and shared-memory software interfaces. The first mapped page contains the user registers that the process will use to communicate with the hardware. The next pages provide communication with the kernel via shared memory. The remaining pages make up the channel's DMA buffer. Initially, the channel does not have permission to access any of the data in Moneta-D.

Managing permissions To gain access to data in Moneta-D, the user space library issues a system call that takes a file descriptor and an offset in the file. The system call returns a description of the file system extent (i.e., the range of physical bytes in Moneta-D) containing that offset. The process uses this information to populate a user space table that maps file offsets onto physical extents. If the process does not have access to that data, the system call returns an error.

The system call also installs the permission record for the extent in the process's channel. Moneta-D's permission record storage is finite, so installing one permission record may require the kernel to evict another. This also means that the process may issue a request for data that it should be able to access and have the request fail. In this case, the process re-issues the system call to re-install the permission record and retries the request.

Issuing and completing commands Once the process has installed a permission record, it can start making requests. To initiate a command, the process writes a 64-bit command word to the channel's memory-mapped command register. The command encodes the operation to perform (read or write), the portion of the DMA buffer to use, the physical location in Moneta-D to access, and a tag to differentiate between requests. After issuing the command, the thread waits for the command to complete.

When Moneta-D receives the command word, it checks the hardware permission table to determine whether the channel has permission to access the location. If it does, it performs the command and signals its completion. In Section 4 we describe several schemes for notifying the thread when a command completes.

2.2 The user space driver

The user space library for accessing Moneta-D performs the low-level driver functions including tag management, extent lookup, and command retry. The library transparently replaces the standard library calls for accessing files using LD_PRELOAD. Dynamically linked applications do not require any modification or recompilation. When the program `open()`s a file on Moneta-D, the library allocates a channel if necessary and then handles all future accesses to that file. The library forwards operations on all other files to the normal `libc` functions.

The POSIX compatibility layer implements POSIX calls (e.g. `read()`, `write()`, and `seek()`) calling Moneta-D specific read and write functions. The layer also tracks file descriptor manipulation functions (e.g. `dup()`, `dup2()`, and `close()`) to track per-file descriptor state (e.g. the file pointer's position) and file descriptor aliasing relationships.

Other, Non-POSIX interfaces are also possible. Moneta-D's hardware interface is inherently asynchronous, so a high-performance asynchronous IO library is a natural fit. In addition, since the channel's DMA buffers reside in the process's address space, an optimized application could avoid copying data to and from the DMA buffer and operate on the data in place instead. We explore both these options in Section 4.

2.3 The file system

Moneta-D changes the way applications interact with the file system to increase performance. These changes require minor modifications in the file system to support moving protection checking into hardware. They also introduce some challenges to maintaining existing functionality and consistency in the file system.

The only change required to the file system is the addition of a function to extract extent information. We implemented this change in XFS [10] and found it to be straightforward, even though XFS is a very sophisticated file system. The single 30-line function accesses and transfers file extent meta-data into Moneta-D's data structures. We expect that adding support to other file systems would also be relatively easy.

All meta-data updates and accesses use the conventional operating system interface. This requirement creates problems when the driver uses the kernel to increase a file's size and then accesses the resulting data directly. When it extends a file (or fills a hole in a sparse file), XFS writes zeros into the kernel's file buffer cache. Although the operating system writes out these dirty blocks after a short period of time, the user space library accesses the newly-allocated blocks as soon as the system call returns and that access will not "see" the contents of the buffer cache. If the application updates the new pages in Moneta-D before the operating system flushes the cached copy, a race will occur and the flushed pages will overwrite the application's changes. To avoid this problem, we flush all blocks associated with a file whenever we fill a hole or extend a file. After the first access to the file, this is usually a fast operation because the buffer cache will be empty.

Guaranteeing consistency while accessing files concurrently through Moneta-D and via the operating system remains a challenge, because of potential inconsistencies caused by the kernel's buffer cache. One solution is to detect files that applications have opened a file using both interfaces and force the application using the user space interface to switch to the system-call based interface. The library could do this without that application's knowledge. Alternatively, disabling the buffer cache for files residing on Moneta-D would also resolve the problem. Our system does not yet implement either option.

Moneta-D's virtual interface also supports arbitrarily sized and aligned reads and writes, eliminating the need to use read-modify-write operations to implement unaligned accesses.

3. Related Work

Our extensions to Moneta touch on questions of virtualization, fast protection and translation, and light-weight user space IO. Below we describe related work in each of these areas and how the system we describe differs from and extends the current state of the art.

3.1 Virtualization

Moneta-D differs from other efforts in virtualizing high-speed IO devices in that it provides virtual *interfaces* to the device rather than logically separate virtual devices. In our system, there is a single logical SSD and a single file system to manage it. However, many client applications can access the hardware directly. Creating multiple, independent virtual disks or multiple, independent virtual network interfaces for multiple virtual machines is a simpler problem because the virtual machine monitor can statically partition the device's resources across the virtual machines.

Previous work in high-speed networking [4, 21, 31] explores the idea of virtualizing network interfaces and allowing direct access to the interface from user space. DART [21] implements network interface virtualization while also supporting offloading of some packet processing onto the network card for additional performance enhancements. Our work implements similar hardware

interface virtualization for storage accesses while enabling file systems protection checking in hardware.

Many projects have developed techniques to make whole-device virtualization more efficient [18, 19, 25, 29, 33], particularly for graphics cards and high-performance message-passing interconnects such as Infiniband. Virtualization techniques for GPUs are most similar to our work. They provide several "rendering contexts" that correspond to an application window or virtual machine [6]. A user space library (e.g., OpenGL) requests a context from the kernel, and the kernel provides it a set of buffers and control registers it can use to transfer data to and from the card without OS involvement. Some Infiniband cards [9] also provide per-application (or per-virtual machine) channels and split the interface into trusted and untrusted components. The work in [16] has explored how to expose these channels directly to applications running inside virtual machines. However, neither of these applications requires the hardware to maintain fine-grain permission data as Moneta-D does.

The concurrent, direct network access (CDNA) model [33] is also similar, but applies to virtual machines. In this model, the network card provides multiple independent sets of queues for network traffic, and the VMM allows each virtual machine to access one of them directly. On an interrupt, the OS checks a register to determine which queues need servicing and forwards a virtual interrupt to the correct VMs.

Recent revisions of the PCIe standard include IO virtualization (IOV) [22] to support virtual machine monitors. PCIe IOV allows a single PCIe device to appear as several, independent virtual devices. The work in [35] describes a software-only approach to virtualizing devices that do not support virtualization, assuming the devices satisfies certain constraints. In both cases the support is generic, so it cannot provide the per-channel protection checks that Moneta-D requires. Some researchers have also found the PCIe approach to be inflexible in the types of virtualized devices it can support [13].

The work in [26] and [24] present new IO architectures with virtualization as the driving concern. In [26] researchers propose a unified interface to several of the techniques described above as well as extensions to improve flexibility. [24] proposes a general approach to self-virtualizing IO devices that offloads many aspects of virtualization to a processor core embedded in the IO device.

3.2 User space IO

Efficiently initiating and completing IO requests from user space has received some attention in the high-speed networking and message passing communities. In almost all cases, the VMs issue requests via stores to PIO registers, and the VMM is responsible for delivering virtual interrupts to the VMs. We describe two alternative approaches below.

Prior work [3, 4, 31] proposed supporting user space IO and initiating DMA transfers from user space without kernel intervention. SHRIMP [3] proposes user space DMA through simple load and store operations, but requires changes to the CPU and DMA engine to detect and initiate transfers. Our work requires no changes to the CPU or chipset.

The work in [27, 28] proposes architectural support for issuing multi-word PIO commands atomically. In effect, it implements a simple form of bounded transactional memory. Moneta-D would be more flexible if our processors provided such support. The same work also suggests adding a TLB to the PCIe controller to allow the process to specify DMA targets using virtual addresses. The PCIe IOV extensions mentioned above provide similar functions. The combination of multi-word atomic PIO writes and the DMA TLB would eliminate the need for a dedicated DMA buffer and make it possible to provide a zero-copy interface on top of Moneta-D that was POSIX-compatible.

The same work also proposes hardware support for delivering interrupts to user space. The device would populate a user space buffer with the results of the IO operation, and then transmit data to the CPU describing which process should receive the interrupt. The OS would then asynchronously execute a user-specified handler. Moneta-D would benefit from this type of support as well, and one of the request completion techniques we examine is similar in spirit. More recently, researchers have proposed dedicating an entire core to polling IO device status and delivering notifications to virtual machines through memory [15]. The driver for recent PCIe-attached flash-based SSDs from Virident dedicates one processor core solely to interrupt handling.

Several papers have argued against user space IO [17]. They posit that efficient kernel-level implementations can be as fast as user-level ones and that the kernel should be the global system resource controller. However, our work and prior work [36] have found that user-level IO can provide significant benefit without significantly increasing complexity for application developers. Our work maintains the kernel as the global policy controller — only policy enforcement takes place in hardware.

3.3 Protection and translation

Moneta-D removes file system latency by copying permission information into hardware and caching the physical layout of data in user space. Some approaches to distributed, networked storage use similar ideas. The latest version of the network file system (NFS) incorporates the pNFS [20] extension that keeps the main NFS server from becoming a bottleneck in cluster-based NFS installations. Under pNFS, an NFS server manages storage spread across multiple storage nodes. When a client requests access to a file, it receives a map that describes the layout of the data on the storage nodes. Further requests go directly to the storage nodes. NASD [7] is similar in that a central server delegates access rights to clients. However, it uses intelligent drives rather than separate storage servers to provide access to data. NASD uses cryptographic capabilities to grant clients access to specific data.

Modern processors provide hardware support for translation and protection (the TLB) and for servicing TLB misses (the page table walker) in order to reduce both translation and miss costs. Supporting multiple file systems, many channels, and large files requires Moneta-D to take a different approach. Our SSD provides hardware support for protection only. Translation must occur on a per-file basis (since "addresses" are file offsets), and hardware translation would require Moneta-D to track per-file state rather than per-channel state.

Rather than addressing physical blocks in a storage device, object-based storage systems [1, 8, 32], store objects addressed by name. They provide a layer of abstraction mapping between object names and physical storage in the device. Moneta-D performs a similar mapping in its user space library, mapping between file descriptor and offset. Shifting these translations into the hardware has several drawbacks for a system like Moneta-D. First, the file system would require significant alterations, breaking the generic support that Moneta-D currently enables. Second, performing the translations directly in hardware could limit Moneta-D's performance if the lookups take more than a few hundred nanoseconds. Finally, dedicated DRAM in Moneta-D for storing lookup information might be better located in the host system where it could be repurposed for other uses when not needed for translations.

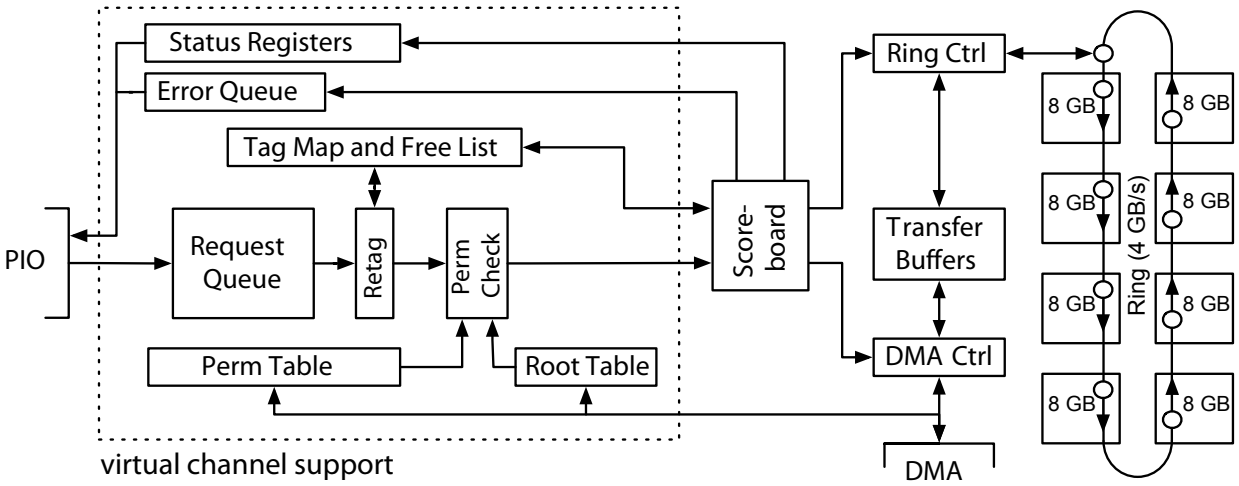


Figure 2. Controller architecture Components inside the dashed box provide support for virtualization and are the focus of this work. The other components (at right) execute storage access commands.

	Name	R/W Kernel	R/W User	R/W HW	Description
Kernel global registers	CHANNELSTATUS	R	-	W	Read and clear channel status and error bits
	ERRORQUEUE	R	-	W	Read and pop one error from the error queue
User per-channel registers	COMMAND	W	W	R	Issue a command to the device.
	TAGSTATUSREGISTER	R	R	W	Read and clear tag completion bits and error flag.
Per-channel mapped memory	TAGSTATUSTABLE	W	R/W	W	Tracks completion status of outstanding requests.
	COMPLETIONCOUNT	W	R	-	Count of completed requests on each channel.
	DMABUFFER	-	R/W	R/W	Pinned DMA buffer for data transfers.

Table 1. Moneta-D channel interfaces There are three interfaces that control Moneta-D: The kernel registers that the kernel uses to configure channels, the per-channel user registers that applications use, and the per-channel mapped memory region shared between the kernel and the application to maintain channel state.

4. Moneta-D Implementation

This section describes the changes to the Moneta [5] hardware and software that comprise Moneta-D. The baseline Moneta system implements a highly optimized SSD architecture targeting advanced non-volatile memories. The Moneta-D modifications enable the hardware and software to work together to virtualize the control registers and tags, efficiently manage permission information, and deliver IO completions to user space. We discuss each of these in turn. Section 5 evaluates their impact on performance.

4.1 The baseline Moneta hardware

The right side of Figure 2 (outside the dotted box) shows the architecture of the baseline array. It spreads 64 GB of storage across eight memory controllers connected via a high-bandwidth ring. An 8-lane PCIe 1.1 interface provides a 2 GB/s full-duplex connection (4 GB/s total) to the host system. The baseline design supports 64 concurrent, outstanding requests with unique tags identifying each. The prototype runs at 250 MHz on a BEE3 FPGA prototyping system [2].

The baseline Moneta array emulates advanced non-volatile memories using DRAM and modified memory controllers that insert delays to model longer read and write times. We model phase change memory (PCM) in this work and use the latencies from [11] — 48 ns and 150 ns for array reads and writes, respectively. The array uses start-gap wear leveling [23] to distribute wear across the PCM and maximize lifetime.

The baseline Moneta design includes extensive hardware and software optimizations to reduce software latency (e.g. bypassing the Linux IO scheduler and removing unnecessary context switches) and maximize concurrency (e.g., by removing all locks in the driver). These changes reduce latency by 62% compared to the standard Linux IO stack, but system call and file system overheads still account for 65% of the remaining software overheads.

The baseline design implements one channel that the operating system alone may access. It provides a set of configuration and command registers and targets a single DMA buffer in the kernel's address space.

Figure 3 shows the latency breakdown for 512 B reads and writes on Moneta-D. The hardware, DMA, and copy overheads are common across the baseline and the extensions we describe in this work. These, combined with the file system, system call, and interrupt processing overheads bring the total request latency in the baseline to 15.36 and 16.78 μs for reads and writes, respectively.

4.2 Virtual channels

Supporting virtual channels on Moneta-D requires replicating the control registers, tags, and DMA buffers mentioned above, while maintaining file coherency across multiple processes. This section describes the hardware and software implementation of virtual channels on Moneta-D. The dashed box in Figure 2 contains the components that implement virtual channels.

Control registers and data structures The interface for a channel comprises several memory-mapped hardware control registers and a shared memory segment. Together, these allow the kernel and the user space library to configure the channel, perform operations on it, and receive notifications when they complete.

Table 1 describes the most important control registers and the shared memory segment. The kernel's global registers allow the kernel to manage Moneta-D's functions that apply to multiple channels, such as error reporting and channel status. The user per-channel registers allow the process to access the hardware. Finally, the per-channel mapped memory shared by the kernel and user contains the channel's DMA buffer and data structures used to notify threads when operations complete. We discuss the role of these components in detail below.

In the non-virtualized system, a single set of control registers exposes Moneta's interface to the kernel. In the virtualized system, Moneta-D exposes 1024 channels, each with a private set of control registers located at a unique physical address. Reading or writing to any of these registers will send a PIO request to Moneta-D. Moneta-D uses the address bits to determine which channel the command targets. To give a process access to a particular channel, the kernel maps the registers for the channel into the process's address space. The unique mapping of physical addresses to channels allows Moneta-D to reliably know which process issued a particular request and prevents processes from accessing channels other than their own.

Request tags The baseline design supports 64 concurrent, outstanding requests. To maximize performance and concurrency, each channel needs its own set of tags. One option is to support 65,536 tags (64 tags for each of the 1024 channels) in hardware and statically partition them across the channels. In a custom ASIC implementation this might be possible, but in our FPGAs maintaining a request scoreboard of that size is not feasible at our 250 MHz clock frequency.

Instead, we provide each channel with 64 virtual tags and dynamically map them onto a set of 64 physical tags. The virtual tag number comprises the channel ID and the tag number encoded in the command word. The "retag" unit shown in Figure 2 assigns physical tags to requests by drawing physical tags from a hardware free tag list. If a physical tag is not available, the retag unit stalls until a request completes and releases its physical tag.

DMA buffer Each channel has a private 1 MB DMA buffer pinned in system DRAM that Moneta-D uses as the source and destination for writes and reads. The target DMA address for a request depends on its tag with each tag corresponding to one 16 KB slice of the channel's DMA buffer. If the process issues a command on tag k, the DMA transfer will start at the kth slice. The access that uses the tag can be larger than 16 KB, but it is the software's responsibility to not issue requests that overlap in the buffer. Moneta-D supports arbitrarily large DMA buffers, but since they must be contiguous in physical memory allocating bigger DMA buffers is challenging. Better kernel memory management would eliminate this problem.

Asynchronous interface Moneta-D's user space library provides asynchronous versions of its `pread()` and `pwrite()` calls (i.e., read/write to a given offset in a file). The asynchronous software interface allows applications to take advantage of the inherently asynchronous hardware by overlapping storage accesses with computation. For example, double buffering allows a single thread to load a block of data at the same time as it processes a different block. The asynchronous calls return immediately after issuing the request to the hardware and return an asynchronous IO state structure that identifies and tracks the request. The application can continue executing and only check for completion when it needs the data from a read or to know that a write has completed.

4.3 Translation and protection

The Moneta-D hardware, the user space library, and the operating system all work together to translate file-level accesses into hardware requests and to enforce permissions on those accesses. Translations between file offsets and physical storage locations occur in the user space library while the hardware performs permission checks. Below, we describe the role of both components and how they interact with the operating system and file system.

Hardware permission checks Moneta-D checks permissions on each request it receives after it translates virtual tags into physical tags ("Perm Check" in Figure 2). Since the check is on the critical path for every access, the checks can potentially limit Moneta-D's throughput. To maintain the baseline's throughput of 1.8 M IOPS, permissions checks must take no more than 500 ns.

Moneta-D must also cache a large amount of permission information in order to minimize the number of "misses" that will occur when the table overflows and the system must evict some entries. These *hard permission misses* require intervention from both the user space driver and the operating system to remedy, a process that can take 10s of microseconds (Figure 3).

To minimize the number of permission entries it must store for a given set of files, Moneta-D keeps extent-based permission information for each channel and merges entries for adjacent extents. All the channels share a single permission table with 16K entries. To avoid the need to scan the array linearly and to allow channels to dynamically share the table, Moneta-D arranges the extent information for each channel as a balanced red-black binary tree, with each node referring to a range of physical blocks and the permission bits for that range. The "Root Table" (Figure 2) holds the location of the root of each channel's tree. Using balanced trees keeps search times fast despite the potentially large size of the permission tree: The worst-case tree traversal time is 180 ns, and in practice the average latency is just 96 ns. With a linear scan, the worst-case time would exceed 65 μs.

To reduce hardware complexity, the operating system maintains the binary trees, and the hardware only performs look ups. The OS keeps a copy of the trees in system DRAM. When it needs to update Moneta-D's permission table, it performs the updates on its copy and records the changes it made in a buffer. Moneta-D then reads the buffer via DMA, and applies the updates to the tree while temporarily suspending protection checking.

User space translation When the user space library receives a read or write request for a file on Moneta-D, it is responsible for translating the access address into a physical location in Moneta-D and issuing requests to the hardware.

The library maintains a translation map for each file descriptor it has open. The map has one entry per file extent. To perform a translation, the library looks up the target file location in the map. If the request spans multiple extents, the library will generate multiple IO requests.

The library populates the map on-demand. If a look up fails to find an extent for a file offset, we say that a *soft permissions miss* has occurred. To service a soft miss, the library requests information for the extent containing the requested data from the operating system. The request returns the mapping information and propagates the extent's protection and physical location information to hardware.

Once translation is complete, the library issues the request to Moneta-D and waits for it complete. If the request succeeds, the operation is complete. Permission record eviction or an illegal request may cause the request to fail. In the case of an eviction,

Component		Latency R/W (μs)	
		1 extent	1K extents
Hardware + DMA		1.26 / 2.18	
Copy		0.17 / 0.13	
SW Extent lookup		0.12	0.23
HW Permission check		0.06	0.13
Soft miss handling		7.28	29.9
Hard miss handling		14.7	38.1
Permission update		3.23	3.26
File System	Baseline	4.21/4.64	
	Moneta-D	0.21/0.29	
System call	Baseline	3.90/3.96	
	Moneta-D	0.00/0.00	
Completion	Baseline (interrupts)	5.82 / 5.87	
	OS forwarding	2.71 / 2.36	
	DMA	2.32 / 2.68	
	issue-sleep	14.65 / 14.29	

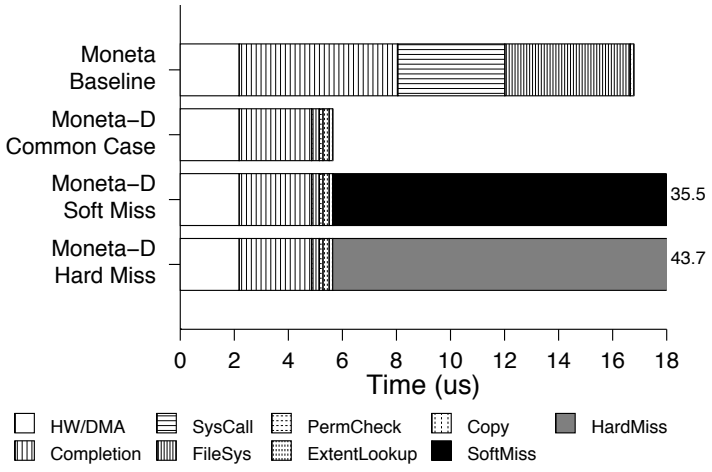


Figure 3. Component latencies Software latencies required to manage permissions, tags, and user data all contribute to operation latency. DMA and copying values are for 512 byte accesses. Cells with a single value have the same latency for both read and write accesses. The graph at right shows the latency breakdown graphically, for 1K extents and using DMA completion.

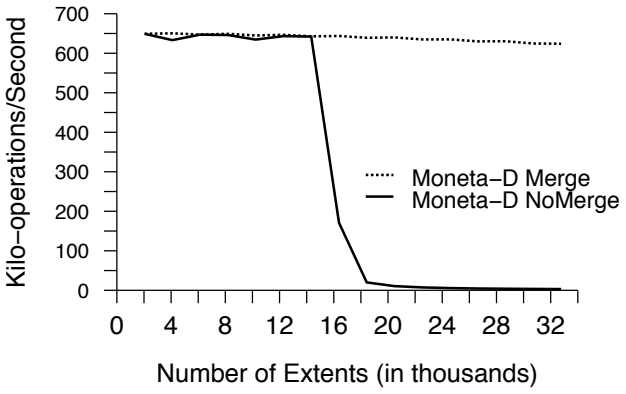


Figure 4. Extent merging to alleviate permission table contention Merging permission entries improves performance because it allows Moneta-D to take advantage of logically discontinuous file extents that the file system allocates in physically adjacent storage.

the permission entry is missing in hardware so the library reloads the permission record and tries again.

Permission management overheads Permission management and checking add some overhead to accesses to Moneta-D, but they require less time than the conventional system call and file system overheads that provide the same functions in conventional systems. Figure 3 shows the latencies for each component of an operation in the Moneta-D hardware and software. To measure them, we use a microbenchmark that performs 512 B random reads and writes to a channel with one permission record and another with 1000 records present. The microbenchmark selectively enables and disables different system components to measure their contribution to latency.

In the common case, accesses to Moneta-D incur software overhead in the user space library for the file offset-to-extent lookup. This requires between 0.12 and 0.23 μs, depending on the number of extents. The hardware permission check time is much faster – between 60 ns and 130 ns.

The miss costs are significantly higher: Handling a soft miss requires between 4.1 μs and 26.8 μs to retrieve extent information from the file system and 3.2 μs to update the permission tree in hardware. In total, a soft miss increases latency for a 512 B access by between 7.3 μs and 30 μs, depending on the number of extents in use. The hard miss adds another 7.7 μs of latency on average, because the user space library does not detect it until the initial request fails and reports an error.

In the best case, only one soft miss should occur per file extent. Whether hard misses are a problem depends on the number of processes actively using Moneta-D and the number of extents they are accessing. Since fragmented files will place more pressure on the permission table, the file system's approach to preventing fragmentation is important.

XFS uses aggressive optimizations to minimize the number of extents per file, but fragmentation is still a problem. We measured fragmentation on a 767 GB XFS file system that holds a heavily-used Postgres database [14] and found that, on average, each file contained 21 extents, and ninety-seven percent of files contained a single extent. However, several files on the file system contain 1000s of extents, and one database table contained 23,396.

We have implemented two strategies to deal with fragmentation. The first is to allocate space in sparse files in 1 MB chunks. When the library detects a write to an unallocated section of a file, it allocates space by writing up to 1 MB of zeroed data to that location before performing the user's request. This helps reduce fragmentation for workloads that perform small writes in sparse files. The second is to merge contiguous extents in the hardware permission table even if the file contents they contain are logically discontiguous in the file. This helps in the surprising number of cases in which XFS allocates discontinuous portions of a file in adjacent physical locations.

Figure 4 shows the benefits of merging permission entries. It shows aggregate throughput for a single process performing random 4 KB accesses to between 2048 and 32,768 extents. The two lines depict the workload running on Moneta-D with (labeled "Moneta-D Merge") and without ("Moneta-D NoMerge") combining permission table entries. Moneta-D Merge combines entries if they belong to the same channel, represent data from the same file,

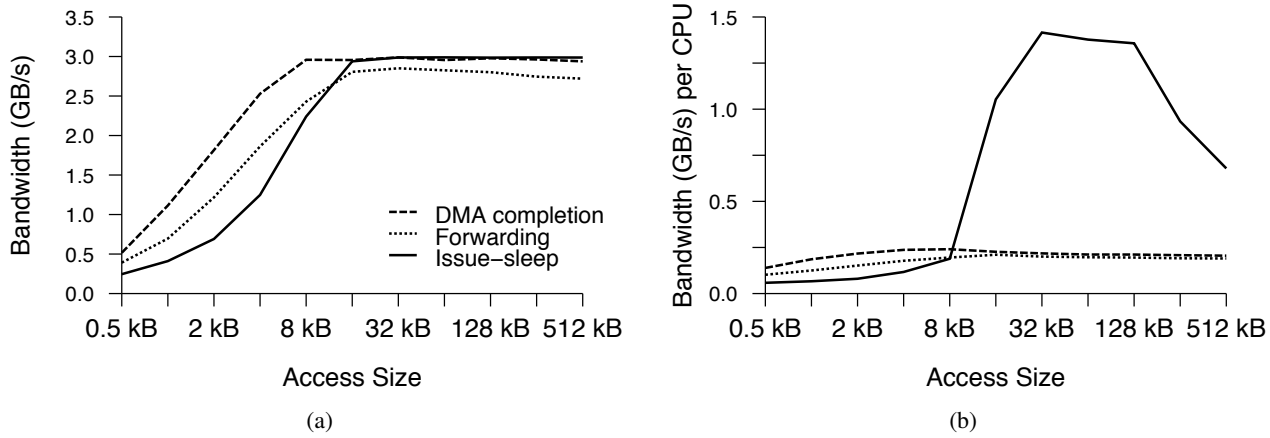


Figure 5. Completion strategies and bandwidth The graphs compare performance and CPU behavior for our three completion strategies for 32 threads performing a 50/50 mix of random reads and writes. Graph (a) measures maximum sustained bandwidth and (b) measures efficiency as the ratio of bandwidth to CPU utilization. For small accesses, DMA completion is the best alternative by both measures. For larger accesses, however, issue-sleep enjoys a large advantage in efficiency.

have the same permission bits set, and cover physically adjacent blocks. Moneta-D NoMerge does not merge extents.

Throughput remains high for Moneta-D NoMerge when there are enough permission table entries to hold all the extent information. Once all 16K permission table entries are in use, throughput drops precipitously as the hard miss rate rises. For Moneta-D Merge, performance remains high even when the number of extents exceeds the permission table size by 2×, because many extents merge into a smaller number of entries.

Avoiding hard misses requires having a sufficient number of permission table entries available for the process accessing Moneta-D directly. There are (at least) three ways to achieve this. The first is to increase the permission table size. In a custom ASIC implementation this would not be difficult, although it would increase cost. The second is to detect over-subscription of the permission table and force some processes to use the conventional system call interface by evicting all their permission table entries, refusing to install new ones, and returning an error code informing the process of the change in policy. Finally, enhancements can be made to more aggressively avoid fragmention in the file system block allocator.

4.4 Completing requests and reporting errors

Modern hardware provides no mechanism for delivering an interrupt directly to a process, so virtualizing this aspect of the interface efficiently is difficult. Moneta-D supports three mechanisms for notifying a process when a command completes that trade-off CPU efficiency and performance. Moneta-D also provides a scalable mechanism for reporting errors (e.g., permission check failures).

Forwarding interrupts The first notification scheme uses a traditional kernel interrupt handler to notify channels of request status through a shared memory page. Moneta-D's kernel driver receives the interrupt and reads the CHANNELSTATUS register to determine which channels have completed requests. The kernel increments the COMPLETIONCOUNT variable for each of those channels.

After issuing a request, the user space library spins on both COMPLETIONCOUNT and the TAGSTATUSTABLE entry for the request. Once the kernel increments COMPLETIONCOUNT one thread in the user space library sees the change and reads the per-channel TAGSTATUSREGISTER from Moneta-D. This read also clears the register, ensuring that only one reader will see each bit that is set. The thread updates the TAGSTATUSTABLE entries for all of the completed tags, signalling any other threads that are waiting for requests on the channel.

DMA completion The second command completion mechanism bypasses the operating system entirely. Rather than raise an interrupt, Moneta-D uses DMA to write the request's result code (i.e., success or an error) directly to the tag's entry in the channel's TAGSTATUSTABLE. In this case, the thread spins only on the TAGSTATUSTABLE.

Issue-sleep The above techniques require the issuing thread to spin. For large requests this is unwise, since the reduction in latency that spinning provides is small compared to overall request latency, and the spinning thread occupies a CPU, preventing it from doing useful work.

To avoid spinning, issue-sleep issues a request to hardware and then asks the OS to put it to sleep until the command completes. When an interrupt arrives, the OS restarts the thread and returns the result code for the operation. This approach incurs the system call overhead but avoids the file system overhead, since permission checks still occur in hardware. The system call also occurs *in parallel* with the access.

It is possible to combine issue-sleep on the same channel with DMA completions, since the latter does not require interrupts. This allows the user library to trade-off between completion speed and CPU utilization. A bit in the command word that initiates a requests tells Moneta-D which completion technique to use. We explore this trade-off below.

Reporting errors Moving permission checks into hardware and virtualizing Moneta's interface complicates the process of reporting errors. Moneta-D uses different mechanisms to report errors depending on which completion technique the request is using.

For interrupt forwarding and issue-sleep, the hardware enqueues the type of error along with its virtual tag number and channel ID in a hardware error queue. It then sets the error bit in the CHANNELSTATUS register and raises an interrupt.

The kernel detects the error when it reads the CHANNELSTATUS register. If the error bit is set, it extracts the error details from the queue by reading repeatedly from the ERRORQUEUE register. Each read dequeues an entry from the error queue. For interrupt forwarding the kernel copies the error codes into the TAGSTATUSREGISTERs for the affected channels. For issue-sleep completion it returns the error when it wakes up the sleeping thread. The kernel reads from ERRORQUEUE until it returns zero.

For DMA completion, the hardware writes the result code for the operation directly into the TAGSTATUSTABLE when the operation completes.

Completion technique performance The four completion method lines of Figure 3 measure the latency of each complation strategy in addition to the interrupt processing overhead for the baseline Moneta design. Interrupt forwarding and DMA completion all have similar latencies – between 2.5 and 2.7 μs. Issue-sleep takes over 14 μs, but for large requests, where issue-sleep is most useful, latency is less important.

Figure 5 compares the performance of the three completion techniques. The data are for 32 threads performing random accesses of the size given on the horizontal axis. Half of the accesses are reads and half are writes. Figure 5(a) measures aggregate throughput and shows that DMA completion outperforms the other schemes by between 21% and 171%, for accesses up to 8 KB. Issue-sleep performs poorly for small accesses, but for larger accesses its performance is similar to interrupt forwarding. We use DMA completion throughout the rest of the paper.

Figure 5(b) measures efficiency in terms of GB/s of bandwidth per CPU. The two spinning-based techniques fare poorly for large requests. Issue-sleep does much better and can deliver up to 7× more bandwidth per CPU. The drop in issue-sleep performance for requests over 128 KB is an artifact of contention for tags in our microbenchmark: Threads spin while waiting for a tag to become available and yield the processor between each check. Since our microbenchmark does not do any useful work, the kernel immediately reschedules the same thread. In a real application, another thread would likely run instead, reducing the impact of the spinning thread.

The data show that DMA completion is a good choice for all request sizes, but that issue-sleep has a slight advantage for CPU-intensive workloads. Issue-sleep is the best choice for accesses of 8 KB or larger because it is more CPU-efficient.

5. Results

Moneta-D's virtualized interface reduces both file and operating system overhead, but it also introduces new sources of potential latency as described in Section 4. This section quantifies the overall impact of these changes on Moneta-D's performance using an IO microbenchmark and several database applications. We also evaluate the benefits of using asynchronous IO on Moneta-D rather than the conventional synchronous operations.

5.1 Operation latency

Figure 6 shows how end-to-end single thread access latency varies over a range of write request sizes from 512 B to 32 KB on the baseline Moneta design and Moneta-D. Read latencies are similar. The graph shows data for accesses running with 1000 permission table entries installed. We collect these data using XDD [34], a flexible IO benchmarking tool. Moneta-D extends the baseline Moneta's performance by a wide margin, while Moneta outperforms state-of-the-art flash-based SSDs by up to 8.7×, with a harmonic mean speedup of 2.1× on a range of file system, paging, and database workloads [5].

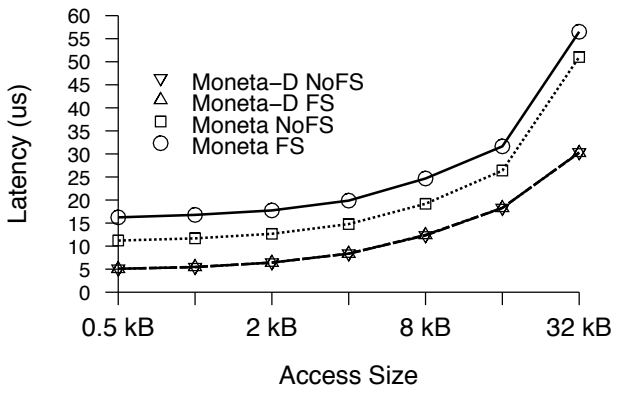


Figure 6. Write access latency Moneta-D's user space interface eliminates most of the file system and operating system overheads to reduce file system access latency by between 42% and 69%.

Figure 6 shows that Moneta-D eliminates most file system and operating system overheads from requests of all sizes, since the lines for Moenta-D FS and NoFS lay on top of each other. Figure 3 provides details on where the latency savings come from for small requests. Assuming the access hits in the permission table, Moneta-D all of this, reducing latency by 60%. Reducing software overheads for small (512 B) requests is especially beneficial because as request size decreases, hardware latency decreases and software latency remains constant.

5.2 Raw bandwidth

Since removing the operating and file systems from common case accesses reduces software overhead per IO operation, it also increases throughput, especially for small accesses. Figure 7 compares the bandwidth for Moneta-D and baseline Moneta with and without the file system. For writes, the impact of virtualization is large: Adding a file system reduces performance for the original Moneta system by up to 13×, but adding a file system to Moneta-D has almost no effect. Moneta-D eliminates the gap for reads as well, although the impact of the file system on the baseline is smaller (at most 34%).

Reducing software overheads also increases the number of IO operations the system can complete per second, because the system must do less work for each operation. For small write accesses with a file system, throughput improves by 26×, and Moneta-D sustains 1.8 M 512 B IO operations per second.

5.3 Application level performance

Table 2 describes the workloads we use to evaluate the application level performance of Moneta-D compared to the baseline Moneta design. The first two workloads are simple database applications that perform random single-value updates to a large key-value store in Berkeley-DB, backed either by a B+tree or a hash table. The six MySQL and PGSQL workloads consist of full OLTP database servers that aggressively optimize storage accesses and have strong consistency requirements. They are running the OLTP portion of the SysBench benchmark suite [30] which includes a variety of OLTP operations including read-only lookups, single-value increments, and complex transactions with multiple lookups and updates.

Table 3 shows the performance results for baseline Moneta and the Moneta-D systems for all of our workloads. We also include performance numbers from a FusionIO 80 GB Flash SSD for comparison. Moneta-D speeds up the Berkeley-DB applications by between 2.6× and 5.7× in terms of operations/second, compared to

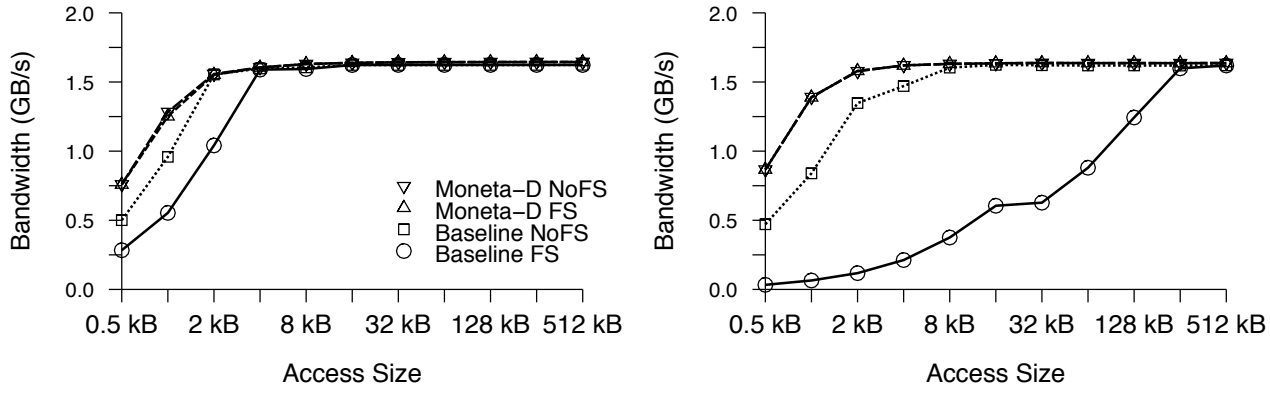


Figure 7. File system overhead Each pair of lines compares bandwidth with and without the file system for reads (a) and writes (b) for the baseline system and Moneta-D. The data show that giving the applications direct access to the hardware nearly eliminates the performance penalty of using a file system and the cost of entering the operating system.

Name	Data footprint	Description
Berkeley-DB Btree	45 GB	Transactional updates to a B+tree key/value store
Berkeley-DB Hash	41 GB	Transactional updates to a hash table key/value store
MySQL-Simple	46 GB	Single value random select queries on MySQL database
MySQL-Update	46 GB	Single value random update queries on MySQL database
MySQL-Complex	46 GB	Mix of read/write queries in transactions on MySQL database
PGSQL-Simple	55 GB	Single value random select queries on Postgres database
PGSQL-Update	55 GB	Single value random update queries on Postgres database
PGSQL-Complex	55 GB	Mix of read/write queries in transactions on Postgres database

Table 2. Benchmarks and applications We use eight database benchmarks and workloads to evaluate Moneta-D.

Workload	Raw Performance			Speedup of Moneta-D vs.	
	FusionIO	Moneta	Moneta-D	FusionIO	Moneta
Berkeley-DB Btree	4066 ops/s	8202 ops/s	21652 ops/s	5.3 ×	2.6 ×
Berkeley-DB Hash	6349 ops/s	10988 ops/s	62124 ops/s	9.8 ×	5.7 ×
MySQL-Simple	13155 ops/s	13840 ops/s	15498 ops/s	1.2 ×	1.1 ×
MySQL-Update	1521 ops/s	1810 ops/s	2613 ops/s	1.7 ×	1.4 ×
MySQL-Complex	390 ops/s	586 ops/s	866 ops/s	2.2 ×	1.5 ×
PGSQL-Simple	23697 ops/s	49854 ops/s	63308 ops/s	2.7 ×	1.3 ×
PGSQL-Update	2132 ops/s	2523 ops/s	5073 ops/s	2.4 ×	2.0 ×
PGSQL-Complex	569 ops/s	1190 ops/s	1809 ops/s	3.2 ×	1.5 ×
			Harmonic mean	2.4 ×	1.7 ×

Table 3. Workload performance Moneta-D provides significant speedups compared to baseline Moneta and FusionIO across a range of workloads. The Berkeley-DB workloads benefit more directly from the increased IO throughput, while the full SQL databases see large gains for write intensive queries. All of the data use the best performing thread count (between 1 and 16) for each workload. FusionIO performance is provided for reference.

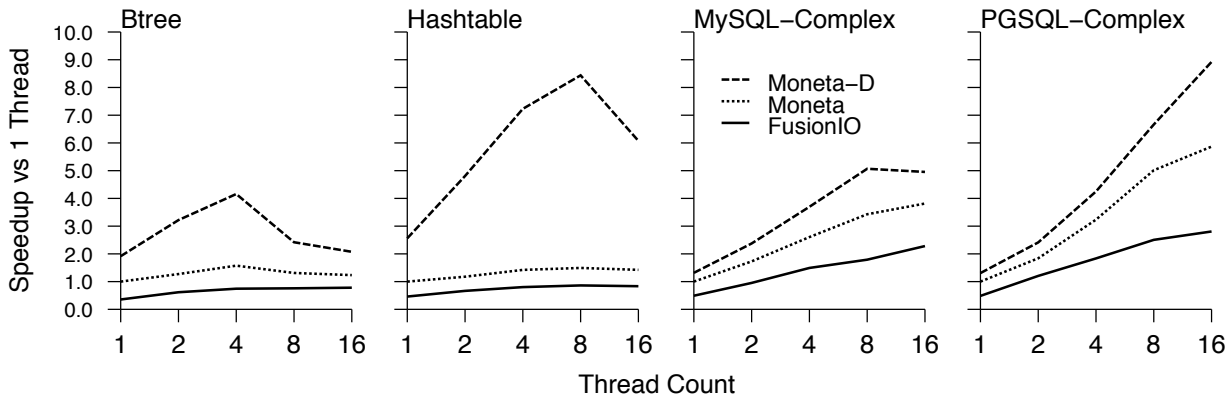
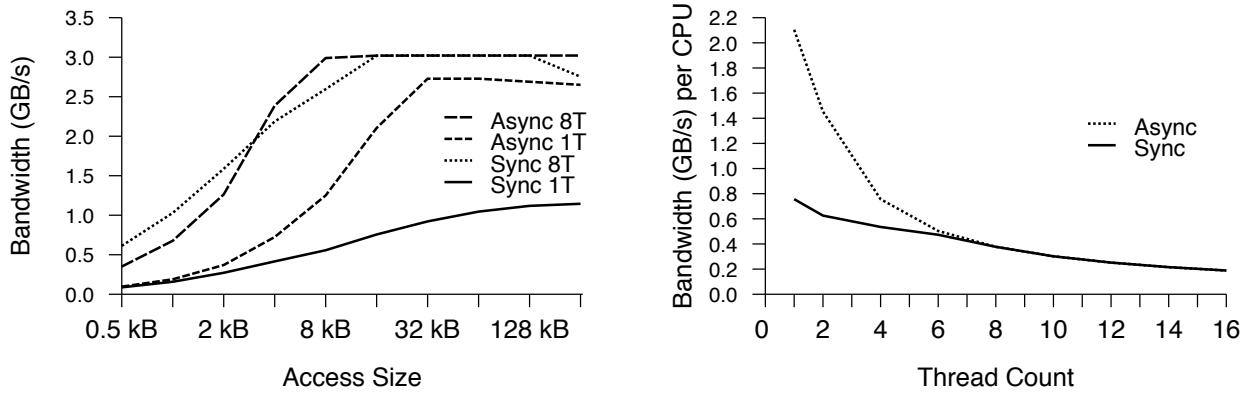


Figure 8. Workload scalability For most workloads, Moneta-D allows application performance to scale better with additional threads, especially for workloads with little inter-thread contention, like Hashtable. Changing Hashtable from 1 to 8 threads on Moneta-D increases performance by 2.2 ×more than the same change on either FusionIO or baseline Moneta.



(a) (b)

Figure 9. Asynchronous performance (a) Moneta-D's asynchronous interface improves single threaded performance by up to 3.5× by eliminating time spent waiting for completions. The data in (b) measure efficiency in terms of bandwidth per CPU with synchronous and asynchronous 16 KB accesses for varying numbers of threads.

baseline Moneta and by between 5.3× and 9.8× compared to FusionIO. We attribute the difference in performance between these two workloads to higher data structure contention in the B+tree database implementation.

Figure 8 shows application performance speedup for varying thread counts from 1 to 16 for the Berkeley-DB and complex MySQL and Postgres databases. Other MySQL and Postgres results are similar. The results are normalized to 1-thread baseline Moneta performance. Baseline Moneta out-performs FusionIO for all thread counts across all of workloads, while Moneta-D provides additional speedup. Increasing thread counts on Moneta-D provides significantly more speedup than on Moneta or FusionIO for the Berkeley-DB workloads, and improves scaling on PGSQL-Complex.

The larger database applications, MySQL and Postgres, see performance improvements from 1.1× to 2.0× under Moneta-D, compared to baseline Moneta. The data show that for these workloads, write-intensive operations benefit most from Moneta-D, with transaction throughput increases of between 1.4× to 2.0×. Read-only queries also see benefits but the gains are smaller — only 1.1× to 1.3×. This is consistent with Moneta-D's smaller improvements for read request throughput.

We found that Postgres produces access patterns that interact poorly with Moneta-D, and that application level optimizations enable better performance. Postgres includes many small extensions to the files that contain its database tables. With Moneta-D these file extensions each result in a soft miss. Since Postgres extends the file on almost all write accesses, these soft misses eliminate Moneta-D's performance gains. Pre-allocating zeroed out data files before starting the database server enables Postgres to take full advantage of Moneta-D. Although Moneta-D requires no application level changes to function, this result suggests that, large performance improvements could result from additional optimizations at the application level, such as allocating large blocks in the file system rather than many small file extensions.

5.4 Asynchronous IO

Providing an asynchronous IO interface to Moneta-D allows applications to take advantage of its inherently asynchronous hardware interface. Figure 9 compares the performance of Moneta-D with

and without asynchronous IO. Figure 9(a) shows sustained bandwidth for the synchronous and asynchronous interfaces with 1 and 8 threads. Asynchronous operations increase throughput by between 1.1× and 3.0× on access sizes of 512 B to 256 KB when using 1 thread. With 8 threads, asynchronous operations boost performance for requests of 4 KB or larger. Small request performance suffers from software overheads resulting from maintaining asynchronous request data structures and increased contention during tag allocation.

Figure 9(b) shows the efficiency gains from using asynchronous requests on 16 KB accesses for varying numbers of threads. The data show that for one thread, asynchronous requests are 2.8× more efficient than synchronous requests with respect to the amount of bandwidth per CPU. As the number of threads increases, the asynchronous accesses slowly lose their efficiency advantage compared to synchronous accesses. As the number of threads increases, the per-thread performance decreases because of increased contention for hardware bandwidth and tags.

To understand the application-level impact of an asynchronous interface, we modified the ADPCM codec from Mediabench [12] to use Moneta-D's asynchronous IO interface and then used it to decode a 100 MB file. Using the asynchronous IO interface results in an 1.4× speedup over the basic Moneta-D interface. By using three buffers, ADPCM can process one block while reading in another and writing out a third. ADPCM's performance demonstrates how overlapping data accesses with data processing enables significant gains. In this case, Moneta-D transformed an IO bound workload into a CPU bound one, shifting from 41% CPU utilization for one thread on the baseline Moneta system to 99% CPU utilization with the asynchronous interface.

6. Conclusion

As emerging non-volatile memory technologies shrink storage hardware latencies, hardware interfaces and system software must adapt or risk squandering the performance these memories offer. Moneta-D avoides this danger by moving file system permission checks into hardware and using an untrusted, user space driver to issue requests. These changes reduce latency for 4 KB write requests through the file system by up to 58% and increase throughput for the same requests by 7.6×. Reads are 60% faster. These increases in raw performance translate into large application level gains. Throughput for an OLTP database workload increased 2.0× and our Berkeley-DB based workloads sped up by 5.7×. Asynchronous IO support provides 5.5× better 4 KB access throughput with 1 thread, and 2.8× better efficiency for 512-B operations, resulting in a 1.7× throughput improvement for a streaming application. Overall, out results demonstrate the importance of eliminating software overheads in IO-intensive applications that will use these emerging memories and point to several opportunities to improve performance further by modifying the applications themselves.

Acknowledgements

The authors would also like to thank Geoff Voelker for his feedback and Jack Sampson for his comments. We would also like to thank the reviewers for their feedback and suggestions and the Xilinx University Program for their support. This work was supported by NSF awards CCF-1018672 and OCI-0910847.

References

[1] The lustre project. http://www.lustre.org/.

[2] http://www.beecube.com/platform.html.

[3] M. A. Blumrich, C. Dubnicki, E. W. Felten, and K. Li. Protected, user-level DMA for the SHRIMP network interface. In *Proceedings of the 2nd IEEE Symposium on High-Performance Computer Architecture*, HPCA '96, pages 154–, Washington, DC, USA, 1996. IEEE Computer Society.

[4] G. Buzzard, D. Jacobson, M. Mackey, S. Marovich, and J. Wilkes. An implementation of the hamlyn sender-managed interface architecture. In *Proceedings of the second USENIX symposium on Operating systems design and implementation*, OSDI '96, pages 245–259, New York, NY, USA, 1996. ACM.

[5] A. M. Caulfield, A. De, J. Coburn, T. I. Mollov, R. Gupta, and S. Swanson. Moneta: A high-performance storage array architecture for next-generation, non-volatile memories. In *In the Proceedings of The 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, 2010.

[6] M. Dowty and J. Sugerman. Gpu virtualization on vmware's hosted i/o architecture. In *Proceedings of the First conference on I/O virtualization*, WIOV'08, pages 7–7, Berkeley, CA, USA, 2008. USENIX Association.

[7] G. A. Gibson, D. F. Nagle, K. Amiri, J. Butler, F. W. Chang, H. Gobioff, C. Hardin, E. Riedel, D. Rochberg, and J. Zelenka. A cost-effective, high-bandwidth storage architecture. In *Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, ASPLOS-VIII, pages 92–103, New York, NY, USA, 1998. ACM.

[8] G. A. Gibson, D. F. Nagle, K. Amiri, J. Butler, F. W. Chang, H. Gobioff, C. Hardin, E. Riedel, D. Rochberg, and J. Zelenka. A cost-effective, high-bandwidth storage architecture. *SIGOPS Oper. Syst. Rev.*, 32:92–103, October 1998.

[9] D. Goldenberg. Infiniband device virtualization in xen. http://www.mellanox.com/pdf/presentations/xs0106_infiniband.pdf.

[10] S. G. International. XFS: A high-performance journaling filesystem. http://oss.sgi.com/projects/xfs.

[11] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger. Architecting phase change memory as a scalable dram alternative. In *ISCA '09: Proceedings of the 36th annual international symposium on Computer architecture*, pages 2–13, New York, NY, USA, 2009. ACM.

[12] C. Lee, M. Potkonjak, and W. H. Mangione-smith. Mediabench: A tool for evaluating and synthesizing multimedia and communications systems.

[13] J. LeVasseur, R. Panayappan, E. Skoglund, C. du Toit, L. Lynch, A. Ward, D. Rao, R. Neugebauer, and D. McAuley. Standardized but flexible i/o for self-virtualizing devices. In *Proceedings of the First conference on I/O virtualization*, WIOV'08, pages 9–9, Berkeley, CA, USA, 2008. USENIX Association.

[14] K. Levchenko, A. Pitsillidis, N. Chachra, B. Enright, M. Félegyházi, C. Grier, T. Halvorson, C. Kanich, C. Kreibich, H. Liu, D. McCoy, N. Weaver, V. Paxson, G. M. Voelker, and S. Savage. Click Trajectories: End-to-End Analysis of the Spam Value Chain. In *Proceedings of the IEEE Symposium and Security and Privacy*, Oakland, CA, May 2011.

[15] J. Liu and B. Abali. Virtualization polling engine (vpe): using dedicated cpu cores to accelerate i/o virtualization. In *Proceedings of the 23rd international conference on Supercomputing*, ICS '09, pages 225–234, New York, NY, USA, 2009. ACM.

[16] J. Liu, W. Huang, B. Abali, and D. K. Panda. High performance vmm-bypass i/o in virtual machines. In *Proceedings of the annual conference on USENIX '06 Annual Technical Conference*, pages 3–3, Berkeley, CA, USA, 2006. USENIX Association.

[17] K. Magoutis, M. Seltzer, and E. Gabber. The case against user-level networking. In *Proceedings of the Workshop on Novel Uses of System-Area Networks*, SAN-3, 2004.

[18] D. McAuley and R. Neugebauer. A case for virtual channel processors. In *Proceedings of the ACM SIGCOMM workshop on Network-I/O convergence: experience, lessons, implications*, NICELI '03, pages 237–242, New York, NY, USA, 2003. ACM.

[19] A. Menon, A. L. Cox, and W. Zwaenepoel. Optimizing network virtualization in Xen. In *Proceedings of the annual conference on USENIX '06 Annual Technical Conference*, pages 2–2, Berkeley, CA, USA, 2006. USENIX Association.

[20] http://www.pnfs.com/.

[21] R. Osborne, Q. Zheng, J. Howard, R. Casley, D. Hahn, and T. Nakabayashi. Dart – a low overhead atm network interface chip.

[22] Pci-sig - i/o virtualization. http://www.pcisig.com/specifications/iov/.

[23] M. K. Qureshi, J. Karidis, M. Franceschini, V. Srinivasan, L. Lastras, and B. Abali. Enhancing lifetime and security of pcm-based main memory with start-gap wear leveling. In *MICRO 42: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 14–23, New York, NY, USA, 2009. ACM.

[24] H. Raj and K. Schwan. High performance and scalable i/o virtualization via self-virtualized devices. In *Proceedings of the 16th interna-*

tional symposium on High performance distributed computing, HPDC '07, pages 179–188, New York, NY, USA, 2007. ACM.

[25] M. Rosenblum and T. Garfinkel. Virtual machine monitors: Current technology and future trends. *Computer*, 38:39–47, May 2005.

[26] J. Satran, L. Shalev, M. Ben-Yehuda, and Z. Machulsky. Scalable i/o - a well-architected way to do scalable, secure and virtualized i/o. In *Proceedings of the First conference on I/O virtualization*, WIOV'08, pages 3–3, Berkeley, CA, USA, 2008. USENIX Association.

[27] L. Schaelicke and A. Davis. Improving i/o performance with a conditional store buffer. In *Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture*, MICRO 31, pages 160–169, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press.

[28] L. Schaelicke and A. L. Davis. Design trade-offs for user-level i/o architectures. *IEEE Trans. Comput.*, 55:962–973, August 2006.

[29] J. Sugerman, P. Venkitachalam, and B.-H. Lim. Virtualizing i/o devices on vmware workstation's hosted virtual machine monitor. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, pages 1–14, Berkeley, CA, USA, 2001. USENIX Association.

[30] http://sysbench.sourceforge.net/index.html.

[31] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-net: a user-level network interface for parallel and distributed computing. *SIGOPS Oper. Syst. Rev.*, 29:40–53, December 1995.

[32] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn. Ceph: a scalable, high-performance distributed file system. In *Proceedings of the 7th symposium on Operating systems design and implementation*, OSDI '06, pages 307–320, Berkeley, CA, USA, 2006. USENIX Association.

[33] P. Willmann, J. Shafer, D. Carr, A. Menon, S. Rixner, A. L. Cox, and W. Zwaenepoel. Concurrent direct network access for virtual machine monitors. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 306–317, Washington, DC, USA, 2007. IEEE Computer Society.

[34] Xdd version 6.5. http://www.ioperformance.com/.

[35] L. Xia, J. Lange, P. Dinda, and C. Bae. Investigating virtual passthrough i/o on commodity devices. *SIGOPS Oper. Syst. Rev.*, 43:83–94, July 2009.

[36] Y. Zhou, A. Bilas, S. Jagannathan, C. Dubnicki, J. F. Philbin, and K. Li. Experiences with VI communication for database storage. In *Proceedings of the 29th annual international symposium on Computer architecture*, ISCA '02, pages 257–268, Washington, DC, USA, 2002. IEEE Computer Society.