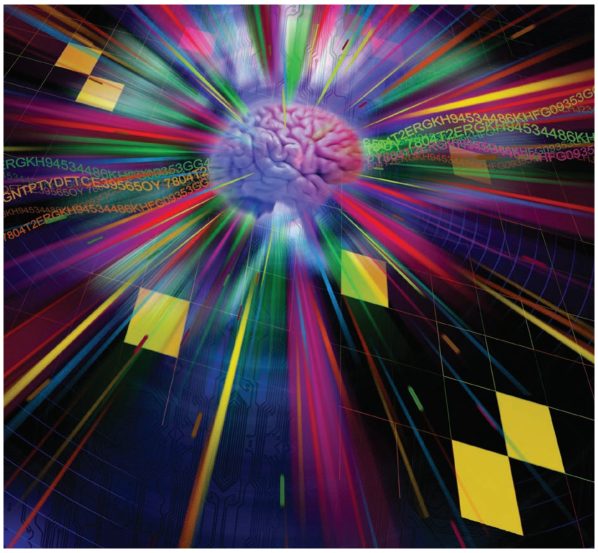
RGKH94534486KHFG09353G
GNTPTYDFTCE395650Y 7804T2E
7804T2ERGKH94534486KHFG09353G
6SUT2ERGKH94534486KHFG09353G
7804T2ERGKH94534486KH

Refactor, Reduce, Recycle: Restructuring the I/O Stack for the Future of Storage

Steven Swanson and Adrian M. Caulfield, *University of California, San Diego*

Emerging nonvolatile storage technologies promise orders-of-magnitude bandwidth increases and latency reductions, but fully realizing their potential requires minimizing storage software overhead and rethinking the roles of hardware and software in storage systems.

The 1956 introduction of IBM's first hard drive revolutionized how computer systems store data, but since then, computer systems have had to deal with storage system performance that lags far behind that of memory, processors, and networks. This reality has shaped the evolution of computing hardware, system software, and applications in fundamental and complex ways. In particular, poor storage hardware performance has made the performance of software layers that manage storage—local and remote file systems, storage hardware drivers, storage networks, databases, and virtual memory systems—relatively unimportant to overall system performance.

Emerging fast, nonvolatile memory (NVM), such as flash and phase-change memory (PCM), alter the performance landscape of storage entirely and require software designers to rethink the importance and role of software in storage systems. Storage software's latency and energy will become dominant costs in storage systems, and that will mean reengineering these systems to minimize those costs.

In existing disk-based systems, software plays an important role in improving system performance. For example, scheduling storage accesses more carefully or managing a database's buffer pool more intelligently can significantly reduce both the number and cost of I/O operations. Software can also provide useful services like replication, encryption, compression, transactional guarantees, provenance tracking, and thin provisioning.

But executing all this code costs both time and energy. For disk-based storage systems, the software costs are minuscule relative to hardware costs: for a 4-Kbyte access to a commodity disk, the stock Linux software stack accounts for just 0.3 percent of the latency and 0.4 percent of the energy.

NVM alters this balance entirely. For the same 4-Kbyte access to a prototype solid-state drive (SSD) we built in the lab, the same software accounts for 70 percent of the latency and 87.7 percent of the energy. This means that running existing storage software stacks on top of new storage technologies is a recipe for disappointment and inefficiency. The resulting system will not realize the performance and efficiency gains that the NVM's raw speed and efficiency should enable.

Making the best use of NVM requires rethinking the role and structure of software and hardware in storage systems. We have spent the past four years redesigning storage software and hardware to meet the needs of fast NVM and have identified three Rs that have proven useful in letting NVM performance shine through while minimizing disruptive changes to other system components: *refactor* storage hardware and software to reduce software bottlenecks, *reduce* software overhead where possible, and *recycle* existing components. We have developed and evaluated these principles during the construction of several prototype storage systems that target fast NVM.

SHIFTING STORAGE TECHNOLOGIES

Storage system performance, efficiency, and features stem from interactions among the storage hardware, the CPU, and, in distributed systems, the network. Solid-state NVM is revolutionizing storage hardware and altering its relationship to the other components.

Plumbing a storage system

Modern storage systems—particularly their software components—are complex and diverse. Systems can be tens of gigabytes in a smartphone to tens of petabytes in a datacenter. Myriad file systems, remote access protocols, and schemes are available to improve reliability, accessibility, and performance.

Figure 1 illustrates the hardware and software components that make up a typical storage system. Within a single computer, the operating system's I/O stack provides applications with access to persistent storage. The file system provides mechanisms for naming and organizing files and manages the storage spread across one or more block devices. A block device might be a local hard drive, a local redundant array of independent disks (RAID), or a remote disk available through a storage area network (SAN) protocol like Fibre Channel or Internet Small Computer System Interface (iSCSI).

The operating system's generic block device driver works in conjunction with device-specific device drivers (for example, for a high-performance RAID card or hardware-accelerated SAN card) to issue read and write requests to a physical storage device, such as a disk, a RAID, or a remote disk.

One strength of storage-system software stacks is their modular design. Kernel modules, for example, can implement useful features like encryption, compression, and replication by creating virtual block devices that interpose on I/O requests and pass them on to another (also potentially virtual) block device.

But this approach also has weaknesses. First, it relies on the operating system for all aspects of storage access and management. The kernel not only sets policy, such as allocating disk space to files and maintaining permissions, but also enforces that policy by checking permissions on each access, for example, and it manages low-level hardware access. This setting and enforcing combination can lead to inefficiencies. The second weakness of this approach, at least for NVM, is its generality. Because the storage system and its layers are meant to be generic, it is difficult to implement optimizations that are based on the

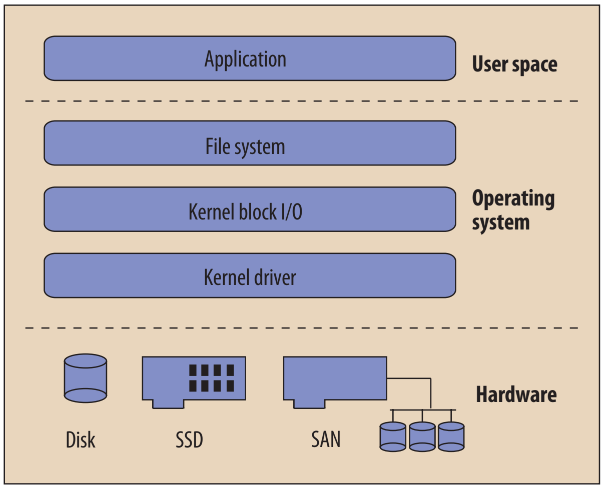

Figure 1. I/O system components. A typical I/O stack combines file systems, a generic kernel block I/O layer, device-specific drivers, and one or more block-based storage devices. SSD: solid-state drive; SAN: storage area network.

underlying hardware. As hardware diversity increases, this difficulty will translate to more and more missed opportunities.

Storage technologies

Compared to the riot of diversity in storage software, the landscape for storage hardware has been largely homogeneous, since hard drives have been the only option. Flash-based SSDs bring a big dose of heterogeneity, and emerging NVM promises even more.

Spinning-disk drives. Spinning-disk hard drives have been the primary storage media for most of computing history, and they will continue to be the cheapest option (excluding tape) on a cost-per-bit basis. Although hard drive performance does vary, those variations are small. For example, the fastest enterprise hard drives offer only about 30 percent more bandwidth than commodity drives (≈200 MBps versus 150 MBps).

The largest shortcoming of hard drives is their long latency. Average hard drive latencies are from 2 ms to over 11 ms. This sluggishness is especially painful for applications such as databases, which require acknowledgment that data is safe on the disk before moving on.

Hard drives are also power hungry. Under heavy load, high-performance hard drives consume about 16 W, or 45 mJ, per random access I/O operation.

Flash-based SSDs. NAND flash is the first type of NVM to see wide commercial adoption. SSDs based on flash memory have become ubiquitous as the storage medium for mobile devices. They are also supplanting spinning disks in storage applications in which performance is more important than cost per bit.

Software Considered Harmful

Nonvolatile memory (NVM) shifts the balance between hardware and software costs in storage systems, and thereby redefines software's role. In disk-based systems, the energy that the storage stack consumes running on a power-hungry CPU pales in comparison to a disk's energy requirements. As a result, it is possible to improve performance and save energy by adding software to a disk-based system.

But host-side software is slow and energy-hungry compared to NVM, and the more software the host executes to manage I/O requests, the slower those requests will be. This means that using existing storage stacks to manage NVM-based storage is a recipe for disappointment and inefficiency, and it will be difficult if not impossible to improve performance and efficiency by adding software to the system. Conversely, reducing interactions with software components and refactoring them to reduce their costs is an effective way to improve performance and efficiency.

Measurements of software and hardware costs in contemporary storage systems illustrate software's shifting role. In the off-the-shelf Linux storage stack, a single 512-byte I/O operation requires about 19 µs of processor time on a 2.27-GHz Nehalem processor. A single active Nahalem core consumes around 28 W, or 532 µJ, per I/O operation.

These software costs are roughly constant regardless of the underlying storage technology, but the relative cost of software changes completely. Figure A illustrates this shift. For disks, software accounts for just 0.27 percent of I/O operational latency, but for the ioDrive (a high-end flash-based solid-state drive) and Moneta (our prototype SSD for next-generation memory), it accounts for 22 percent and 70 percent, respectively. The shift is almost as dramatic for energy: 0.42 percent of the disk's I/O operational energy goes to software versus 73 percent and 95 percent for ioDrive and Moneta.

The shifting ratio of software to hardware costs has profound effects on how designers should approach crafting a storage system. As an example, consider the decision to add the logical volume manager (LVM) to a Linux storage stack to make expanding system capacity easier. Table A shows the comparison between a disk-based system, ioDrive, and the NVM-based Moneta SSD. Adding this layer increases software latency by 2 µs and energy consumption by 56 µJ per I/O operation. In the disk-based system, these increases are negligible, but they are much higher for the ioDrive, and highest of all for Moneta.

To minimize the harm that operating system software causes, system designers need to reengineer storage systems to minimize software's role. In some cases, this will require extensions or modifications to storage hardware, but often it means applying well-known design principles to refactor existing systems.

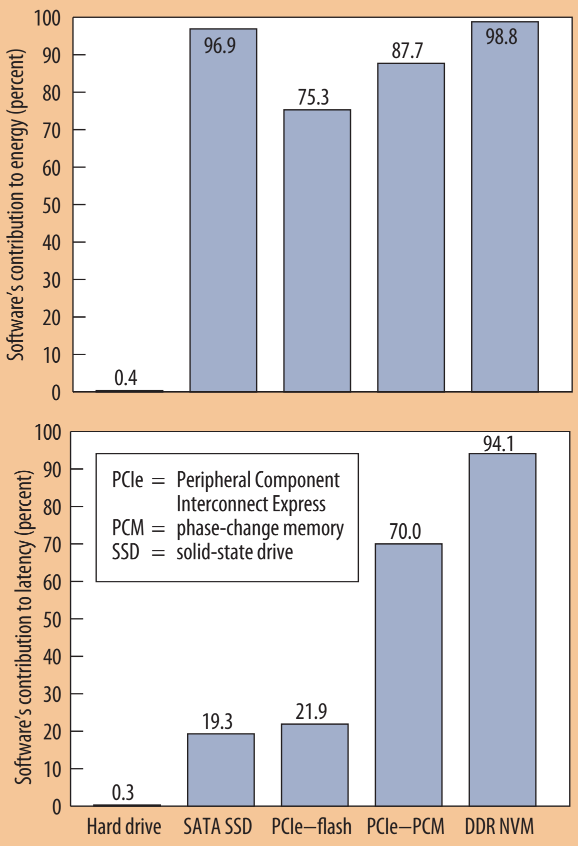

Figure A. Software's impact on latency and energy. Without significantly reengineering the storage software stack, software latency and energy per I/O operation will quickly dominate I/O costs.

Table A. Impact of adding a logical volume manager (LVM) to three storage systems.

Storage system	Software latency increase (percent)	Energy consumption increase (percent)
Disk-based	0.03	0.04
ioDrive (flash-based SSD)	4.30	15.50
Moneta (NVM-based SSD)	10.70	18.70

SSDs excel at performing random, unpredictable accesses very quickly and efficiently. A state-of-the-art SSD, like FusionIO's ioDrive2 Duo, offers a read (write) latency of 47 µs (15 µs). SSDs can handle many requests in parallel, sustaining hundreds of thousands of random I/O operations per second (IOPS) at just 75 µJ per IOP—a 600× difference relative to disk. A high-end SSD can sustain between 1.5 and 3 GBps.

Next-generation memory. Several emerging technologies promise to improve on flash's performance. Both PCM and spin-torque transfer magnetic RAM (STT-MRAM) are available commercially. Micron and Samsung have produced gigabit PCM devices, and Everspin is sampling DDR3-based STT-MRAM DIMMs.

According to projections from the International Technology Roadmap for Semiconductors,[1] a semiconductor manufacturing trade group, PCM will eventually offer read performance on par with dynamic RAM and write performance within a small factor of that possible with DRAM. STT-MRAM and the memristor promise similar or better performance. Experiments in our lab suggest that SSDs built from memory will perform I/O operations in 1.2 to 4 μs, consume no more than 20 to 40 μJ per I/O operation, and sustain at least 1 million IOPS.

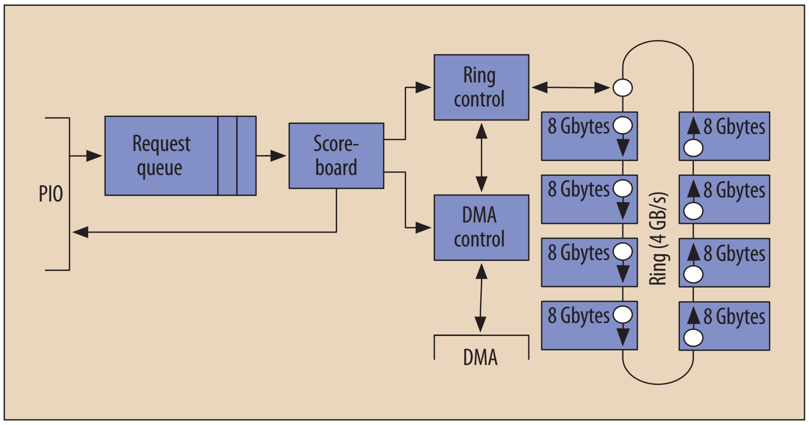

Figure 2. Moneta's internal architecture. Like other high-end solid-state drives (SSDs), Moneta attaches to a host server through Peripheral Component Interconnect Express and provides fast access to eight banks of nonvolatile memory (NVM) arranged around a high-performance ring-based interconnect.

THREE Rs IN BRIEF

Storage software's changing role requires a change in the software stack's architecture. Over the past four years, we have built a sequence of prototype NVM storage systems that have enabled us to wrestle with these changes firsthand. We have found that the combination of refactoring, reducing, and recycling can maximize performance with minimal disruption to the rest of the system.

Refactor

Redistributing storage system functionality across the application, operating system, and hardware can improve software efficiency. Existing software stacks rely on the operating system for all aspects of I/O processing. The operating system and file system are responsible for both setting storage policy and enforcing that policy. In our experience, the largest gains have come from separating these two, minimizing the amount of enforcement necessary, and then moving enforcement into hardware. As the "Software Considered Harmful" sidebar explains, relying on hardware, rather than software, for policy enforcement and implementation can be particularly profitable.

Reduce

Executing common case operations should require as little software interaction as possible. Designers should codesign hardware and software to get software out of the way so systems can fully exploit the raw performance of NVM. Using disk-centric software to manage NVM, for example, often reveals latent performance bottlenecks and disk-centric optimizations that restrain performance. Bottlenecks can include highly contended locks and extra layers of indirection or queuing. Troublesome disk-centric optimizations include complex I/O schedulers and, surprisingly, the operating system's disk-caching mechanisms.

Recycle

NVM demands big changes in how storage works, but it is often possible to leverage existing technologies and software components, which reduces design effort and thus eases the path of adoption.

Although reducing and refactoring software interactions can significantly alter the architecture of the storage system, we have found that careful design can allow for extensive reuse of software components. This saves time and makes it easier to integrate NVM into existing systems.

CASE STUDY: MONETA

One of our prototypes is the Moneta SSD,[2-4] which targets emerging, fast NVM and is the first system we built to understand how NVM performance should influence software architecture. As Figure 2 shows, I/O requests arrive from the host machine through programmed I/O (PIO) requests over Peripheral Component Interconnect Express (PCIe) and enter a request queue. The scoreboard tracks the status of in-flight operations and signals the host when they complete. The scoreboard also coordinates the operation of the DMA and ring controllers.

We implemented Moneta on the BEE3 field-programmable gate array (FPGA) prototyping platform. The BEE3 provides four FPGAs and can host either 64 Gbytes of DDR2 DRAM or 8 Gbytes of PCM. Moneta has been running in our lab since 2009, and in 2011, the PCM-equipped version became the world's first publicly demonstrated PCM SSD.[5]

For the case studies we describe, Moneta used DRAM (and a custom memory controller) to emulate the performance characteristics of future PCM technology. In this configuration, Moneta can perform a random 4-Kbyte read or write in 8 μs, and we estimate that a commercial version of the device would consume between 88 and 240 μJ per I/O operation.

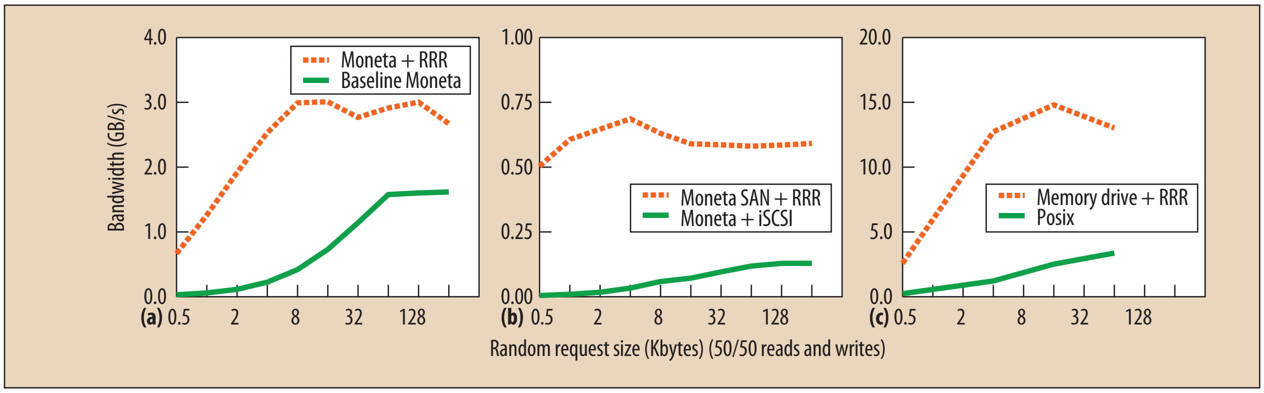

Figure 3. Bandwidth improvements from the three Rs for (a) Moneta, (b) distributed Moneta, and (c) memory drive systems. Minimizing software overhead leads to huge improvements in bandwidth for all three systems across a wide range of access sizes. SAN: storage access network; iSCSI: Internet Small Computer System Interface.

Reduce

The first set of software optimizations we implemented for Moneta[3] streamlined the Linux software stack by addressing two bottlenecks that lay dormant in disk-based systems but crippled Moneta's performance. We also implemented many optimizations to reduce latency and increase parallelism. Overall, we reduced software latency by 43 percent and realized an 18-fold increase in bandwidth for small accesses.

I/O requests. The Linux I/O scheduling framework forces the thread issuing the I/O request to place the request in a queue. Later, the single scheduling thread removes the request, schedules it, and eventually issues it. The framework is flexible and supports pluggable I/O schedulers, but the context switch it requires adds 2 μs of software latency. The Moneta driver eliminates the queues and allows each thread to issue its own I/O requests to hardware, avoiding context switches and increasing parallelism.

Interrupts. Hard drive controllers use interrupts to notify the kernel when operations complete. When a thread issues an I/O request, it goes to sleep and waits for the kernel to reschedule it in response to the interrupt. The process of sleeping and waking from sleep adds 6 μs of software latency. Instead of interrupts, threads spin while waiting for Moneta to complete requests smaller than 4 Kbytes, since doing so is faster (and more efficient) than sleeping and awakening.

Refactor and recycle

Even with a streamlined software stack, software accounts for 57 percent of the latency and 87.7 percent of the energy required to perform an I/O request to Moneta. Further improvements required moving beyond software optimization to software restructuring.

The file system was a clear culprit, since XFS adds 5 μs to each request. Optimizing or rewriting the file system were unattractive options for two reasons. First, we would need to repeat the process for every file system, making it more difficult to widely deploy NVM. Second, the file system was only part of the problem. The costs of entering and exiting the operating system were also significant (accounting for 18 percent of the latency for small accesses), so Amdahl's law would limit the gains an optimized file system could provide.

The approach we took addressed both of these problems. We refactored the storage stack to change the role of the operating system and file system in managing, accessing, and protecting storage. At the same time, we recycled existing file systems to ensure that Moneta can leverage the existing tools and features that these file systems provide.

Refactor. We broke the software overhead for processing an I/O request into two parts: the cost of implementing file system protection and the cost of translating a file and file offset into a physical storage location.

The file system plays two roles in implementing permissions. It is responsible for setting access policy for the storage device by modifying permission bits, allocating extents on the storage device, and assembling extents into logically contiguous files. These policy decisions ultimately determine which processes should have access to which extents. The file system is also responsible for enforcing permissions. Every read and write operation enters the kernel, where the file system mediates access, ensuring that applications access only the files they have successfully opened.

Although the file system's allocation and protection policies are complex and file system specific, the policy's summary—the boundary and permission bits for each extent—is compact and generic. Moreover, the file system makes policy decisions only when it must extend a file or modify metadata. Common-case read and write operations require only enforcement.

On the basis of this analysis, we refactored the I/O stack to remove trusted software (the operating system and file system) from the code path for common-case read and write operations. The refactored system comprises four components:

- *User-space driver for Moneta*. The LibMoneta library provides direct access to the Moneta hardware and, for most operations, does not need to call into the operating system. The library provides a Posix-compatible interface, so applications require no modifications.
- *Virtualized hardware interface*. Each process uses a private logical interface to access Moneta, allowing multiple applications to use Moneta simultaneously.
- *Hardware permission checks*. The Moneta hardware provides a protection table for each application that specifies which portion of the storage device the process can access. Accesses outside those regions fail.
- *Permissions manager in the kernel*. To grant an application access to the storage device, LibMoneta communicates with the kernel's permissions manager, which manages the application's protection table. The cost of communicating with the permissions manager is small, since it is amortized over all future accesses to file data. The permissions manager is generic and works with many file systems.

As a result of these changes, the software latency for common-case reads and writes does not include entering the kernel or executing any file system code, and the additional code executed in LibMoneta is small by comparison. Overall, these changes reduce software latency by 11.13 μs (69 percent), with a similar reduction in software energy per I/O operation. Permission checks add just 0.18 μs to access latency.

As Figure 3a shows, refactoring the Moneta software stack reduces software overhead by 62 percent and greatly increases bandwidth as well. Relative to the initial implementation, sustained bandwidth for small, random reads and writes is 22 times greater, and large transfer performance nearly doubles.

Recycle. Implementing the refactored software stack was complex but required no modifications to the file system. Like most file systems, XFS provides an interface that allows the kernel to extract file-layout information in a generic format. This interface is all that the Moneta kernel and user-space drivers need to function properly.

Distributed storage

Because scale-out computing is on the rise, we also wanted to understand how to leverage NVM in large, distributed storage systems. Modern enterprise and cloud storage systems span thousands of host processors and bring with them new and costly software latency and energy overhead.

SAN-based storage systems allow storage systems to scale, but they suffer from two sources of overhead that can obscure the underlying storage performance. The first is the device-independent software stack layers, such as the file system, and the second is the block transport, the combination of software and interconnection hardware that provides access to remote storage.

To illustrate the impact of these sources, consider the iSCSI and Fibre Channel SAN block transport layers. iSCSI uses a conventional network stack to implement block transport functions. Including the SAN file system, network stack, and so on, the iSCSI layer adds 284 μs to a 4-KByte read access. Fibre Channel uses specialized storage adapter cards, network gear, and software to reduce software overhead. It adds 86 μs to the same read. Left unchecked, this overhead would squander any performance gains that Moneta provides.

Changes to Moneta ensure that common-case reads and writes do not require entering the kernel or executing any file system code.

We adapted Moneta for use in distributed storage by integrating a network interface directly into the storage hardware, and applying the three Rs.[6] With the integrated network interface, we could refactor and reduce the software portion of remote accesses by adding hardware support for forwarding remote accesses to other Moneta devices on the network. Because distributed Moneta uses the same interface as before, we were able to recycle LibMoneta and the rest of the optimized I/O interface. As a result, Moneta behaves just like a large, shared-disk block device. Existing shared-disk file systems work seamlessly with LibMoneta to provide consistency across the nodes while maintaining the low overhead software stack.

As Figure 3b shows, distributed Moneta performance is 107 times higher than the iSCSI configuration performance for small requests and 4.5 times higher for large requests.

Refactor and reduce. We designed this version of Moneta to move all the software overhead of remote storage accesses into hardware. Implementing the block transport directly in the SSD minimizes software overhead for remote accesses at the client, and the server SSD can service the request with no software interaction.

Recycle. Distributed Moneta transforms Moneta from a locally attached storage device into a gateway to a large, shared-disk block device distributed throughout the network. This model is a great match for shared-disk file systems like general parallel file system (GPFS). Further-more, those file systems provide the same interface to extract extent and permission information that Moneta needs to enforce protections, so we were able to recycle LibMoneta as well.

CASE STUDY: MEMORY DRIVES

NVM is also destined for main memory—memory attached to the processor's memory bus—and the resulting storage system will be much faster than Moneta. As a result, software is even more harmful than in PCIe-attached storage.

The vision for using nonvolatile main memory as storage is straightforward: all or part of the main memory attached to the processor is a nonvolatile technology. The main memory could be PCM, the memristor, or STT-RAM, for example. The operating system treats this nonvolatile region as a storage device analogous to a RAM disk, which we call a *memory drive*.[7]

Running a conventional software I/O stack on a memory drive would intensify all the software-related inefficiencies that Moneta encountered. It also presents new opportunities for reducing software interactions. The most notable of these is copying data between the storage device and main memory.

Reduce

Memory drives can avoid copying by using virtual memory techniques to map a file's contents directly into the process's virtual address space. Memory-mapped files are nothing new, but the effect here is quite different. With conventional memory-mapped files, the virtual memory system pages (copies) the contents of the file back and forth between disk and main memory, so stores to the mapped file do not immediately become persistent. With a memory drive, the file itself (rather than a copy) resides in the application's address space, so there is no need to copy data.

Refactor

The virtual memory system also provides the means to refactor file systems for memory drives using techniques similar to Moneta's approach. Memory drives can leverage the existing virtual memory system to enforce file system protection policies. During the mapping operation, the file system transfers file system permissions into memory page permissions, and the translation look-aside buffer (TLB) enforces file system protection policy.

Recycle

Using the TLB to enforce file system permissions works for almost all file systems, so we were able to recycle them along with their associated tools and utilities. We also provided a Posix-like interface to the memory drive so existing applications can run without modification.

Results

Reducing, refactoring, and recycling for memory drives yielded performance gains even larger than what was possible for Moneta. Bypassing the operating system and file system for common-case accesses reduced latency by between 87 and 96 percent. As Figure 3c shows, bandwidth for random 4-Kbyte requests was 1.9 to 14.8 times greater.

OPEN QUESTIONS

The reductions in software overhead that Moneta and memory drives demand are just the beginning of what is necessary to fully leverage fast NVM. Open challenges remain on at least two fronts: handling metadata and application-software costs.

Handling metadata

Both Moneta and memory drives focus on minimizing the software overhead in accessing file content. Updates to file system metadata, such as creating, enlarging, renaming, and deleting files, still go through the kernel and incur all the associated latency and energy costs. For metadata-intensive applications, the cost of metadata updates can dominate file access in its impact on performance.

Addressing metadata costs is challenging because the system cannot trust applications to modify file system data structures correctly. A possible approach is to accumulate metadata updates in the user space and apply them at once;[8] another is to provide applications with private file systems with explicit share or export capabilities.

Application-level software costs

The operating system and file system are not the only sources of software overhead in storage systems. Databases include sophisticated mechanisms for optimizing accesses to storage, many of which are ineffective or counterproductive for NVM-based storage systems. Consequently, the techniques we recommend to improve time and energy efficiency are likely to apply to databases as well.

Our group has taken the first step to test this idea by refactoring the ShoreMT database storage manager to better use Moneta. The first step was to reexamine the Algorithms for Recovery and Isolation Exploiting Semantics (ARIES)[9] write-ahead logging algorithm that many databases use to support scalable transactions. ARIES works hand in glove with disks and includes several disk-centric designs that unnecessarily reduce performance on Moneta. Adding simple hardware support for logging to Moneta allowed us to refactor ARIES into Modified ARIES Redesigned for SSDs (MARS),[10] a more efficient logging scheme that can replace ARIES in transactional databases and is tailored to NVM characteristics. In our preliminary evaluations, MARS reduced software overhead by 65 percent for small atomic write logging operations.

THE FOURTH R: REVOLUTION

The three Rs constitute an effective approach to integrating NVM into existing storage and memory hierarchies because they help expose NVM performance without requiring significant, fundamental changes to dealing with files and persistent storage. But although this evolutionary change is important, it is not sufficient.

NVM has the potential to reshape how programmers and applications use and reason about both persistent and transient state. For disk-based systems, the performance gap between main memory and storage was so great that the interface used to access persistent storage made little difference.

Now, however, the interface does indeed make a difference. For Moneta, providing a Posix-like interface to storage increases latency by at least half for small accesses; for memory drives, the opportunity cost is even greater: treating memory drive content as files misses the idea that a memory drive is a type of memory and programmers can use it as such—employing the full breadth and depth of rich, pointer-based, strongly typed data structures, rather than relying on the file-based interface to an untyped byte array.

Work on replacing these interfaces has begun,[11,12] and performance gains are already impressive. However, moving beyond these early results requires completely rethinking how applications use storage. Systems such as Moneta and memory drives give a glimpse of NVM's potential and pave the way for redefining notions about computer storage.

Fast NVM is on the rise, and it offers, at long last, the promise of storage systems that can keep up with processors, networks, and main memory. However, the software that manages and provides access to storage needs significant reengineering to match the performance of these new storage media. Our experience with Moneta and memory drives provides a blueprint for how to reshape the role storage software plays in mediating storage access and how it interacts with storage hardware. By reducing software interactions where possible, refactoring management functions between hardware and software, and recycling existing software when possible, software designers can provide a smooth path to broader adoption of fast NVM and lay the groundwork for more sweeping changes in how applications interact with persistent data. C

References

1. "ITRS 2011 Tables: Emerging Research Devices," Int'l Tech. Roadmap for Semiconductors, 2011; www.itrs.net/Links/2011ITRS/2011Tables/ERD_2011Tables.xlsx.
2. A.M. Caulfield et al., "Understanding the Impact of Emerging Nonvolatile Memories on High-Performance, I/O-Intensive Computing," *Proc. ACM/IEEE Int'l Conf. High-Performance Computing, Networking, Storage and Analysis* (SC 10), IEEE, 2010, pp. 1-11.
3. A.M. Caulfield et al., "Moneta: A High-Performance Storage Array Architecture for Next-Generation, Nonvolatile Memories," *Proc 43rd IEEE/ACM Int'l Symp. Microarchitecture* (MICRO 43), IEEE, 2010, pp. 385-395.
4. A.M. Caulfield et al., "Providing Safe, User Space Access to Fast, Solid-State Disks," *Proc. 17th Int'l Conf. Architectural Support for Programming Languages and Operating Systems* (ASPLOS 12), ACM, 2012, pp. 387-400.
5. A. Akel et al., "Onyx: A Prototype Phase-Change Memory Storage Array," *Proc. 3rd Usenix Conf. Hot Topics in Storage and File Systems* (HotStorage 11), Usenix, 2011, pp. 1-15.
6. A.M. Caulfield and S. Swanson, "QuickSAN: A Storage Area Network for Fast, Distributed, Solid State Disks," to appear in *Proc. 40th Int'l Symp. Computer Architecture* (ISCA 13), ACM, 2013.
7. L.A. Eisner, T. Mollov, and S. Swanson, "Quill: Exploiting Fast Nonvolatile Memory by Transparently Bypassing the File System," tech. report CS2013-0991, Dept. Computer Science & Eng., Univ. of Calif., San Diego, 2013.
8. H. Volos et al., "Storage-Class Memory Needs Flexible Interfaces," to appear in *Proc. 4th Asian-Pacific Workshop on Systems* (APSys 13), ACM, 2013.
9. C. Mohan et al., "ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging," *IEEE Trans. Database Systems*, vol. 17, no. 1, 1992, pp. 94-162.
10. J. Coburn et al., "From ARIES to MARS: Transaction Support for Next-Generation Solid-State Drives," to appear in *Proc. 24th Int'l Symp. Operating Systems Principles* (SOSP 13), ACM, 2013.
11. J. Coburn et al., "NV-Heaps: Making Persistent Objects Fast and Safe with Next-Generation, Nonvolatile Memories," *Proc. 16th Int'l Conf. Architectural Support for Programming Languages and Operating Systems* (ASPLOS 11), ACM, 2011, pp. 105-118.
12. H. Volos, A.J. Tack, and M.M. Swift, "Mnemosyne: Lightweight Persistent Memory," *Proc. 16th Int'l Conf. Architectural Support for Programming Languages and Operating Systems* (ASPLOS 11), ACM, 2011, pp. 91-104.

Steven Swanson *is an associate professor in the Department of Computer Science and Engineering at the University of California, San Diego, where he also directs the university's Non-Volatile Systems Laboratory. His research interests include the systems, architecture, security, and reliability issues surrounding nonvolatile, solid-state memory. Swanson received a PhD in computer science and engineering from the University of Washington. He is a member of ACM. Contact him at swanson@cs.ucsd.edu.*

Adrian M. Caulfield *is a graduate student in the Non-Volatile Systems Laboratory at the University of California, San Diego. His research focuses on the application of emerging NVM technologies in high-performance storage systems. Caulfield received a PhD in computer science and engineering from the University of California, San Diego. He is a member of ACM. Contact him at acaulfie@cs.ucsd.edu.*