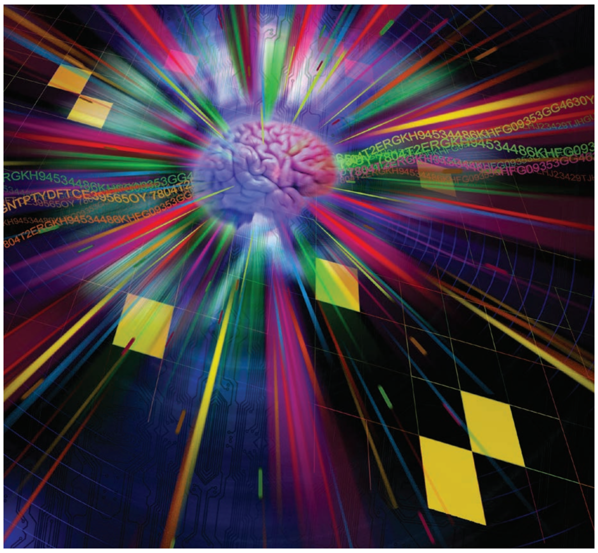

How Persistent Memory Will Change Software Systems

Anirudh Badam, *Microsoft Research*

Persistent memory promises to be byte addressable, fast, and nonvolatile as well as provide higher capacity and more efficient power consumption. System applications designers stand to gain much from these features, but some work is necessary to fully exploit them.

Memory is commonly perceived to be byte addressable, fast, and volatile, and storage to be block-oriented, slow, and nonvolatile. However, new nonvolatile memory technologies will soon mark a significant departure from these assumptions. Persistent memory, in particular, promises to deliver the byte addressability and speed of dynamic RAM (DRAM) and the nonvolatility of disk and flash memory, as well as better capacity and energy efficiency.

In the past decade, NAND flash solid-state drives (SSDs) have revolutionized storage subsystems. With two orders of magnitude less latency compared to magnetic disks, SSDs have changed how applications use storage. Many persistent memory prototypes likewise promise to transform memory subsystems—enabling nonvolatility, and scalability with 10 times more capacity relative to DRAM while maintaining DRAM's latency—at least for reads.

File-system, database, and OS researchers have stepped up efforts to embrace this impending technology. Byte addressability is highly desirable for storage technologies like file systems and databases. However, exploiting persistent memory's nonvolatility will require an evolution in virtual memory management. Researchers are already exploring memory and storage systems that use persistent memory, but open problems remain that system design research must tackle early on. Experiments at Microsoft Research show that flash-based systems suffer many of same disadvantages as persistent-memory-based prototypes, so similar lessons might apply to coping with the newer technology.

As persistent memory technology improves, it will be feasible and practical to replace DRAM, flash, and disk technologies in part or in whole.[1] Meanwhile, augmenting these technologies with persistent memory can realize practical benefits with little or no modification to applications.

WHAT PERSISTENT MEMORY OFFERS

Some of persistent memory's more compelling benefits include increased physical memory, byte-granularity persistence, and data durability with low latency.

Larger physical memory

Augmenting DRAM with persistent memory can help application designers overcome well-known DRAM shortcomings, including high cost and low capacity. As Figure 1 shows, the cost of DRAM scales nonlinearly when the amount of DRAM increases in a single system. The largest available direct dual inline memory module (DIMM) today is 32 Gbytes, and that size costs 10 times more per gigabyte than an 8-Gbyte DIMM. Also, logic boards have a limited number of DIMM slots, which restricts the amount of DRAM that a single board can hold.

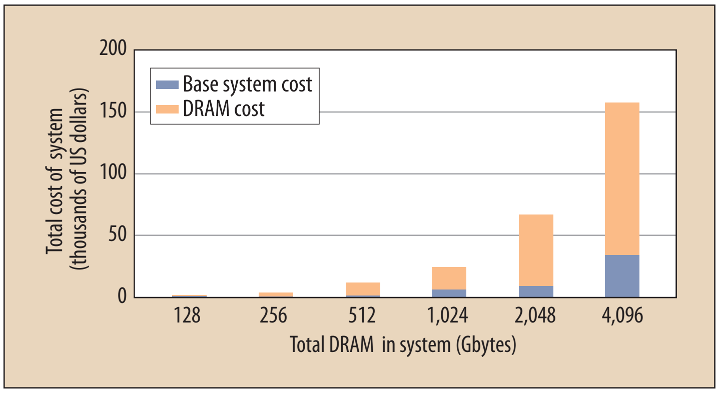

Figure 1. System cost as a function of total dynamic RAM. DRAM cost increases nonlinearly with the amount of system DRAM. Augmenting DRAM with persistent memory can help mitigate this trend.

It is currently impossible to find a single-system image server with more than 4 Tbytes of DRAM. To increase the memory for applications, some systems scale out over a cluster,[2] but for I/O-intensive applications that need more memory, the initial and operational costs of the additional CPUs in a scaled-out system represent unavoidable overhead. Systems such as battery-backed DRAM[3] and RamCloud[2] face these DRAM scalability challenges.

Thanks in part to Moore's law, these costs are decreasing as DRAM density increases, but data generation and consumption are vastly outpacing any improvement.[4] This trend exacerbates DRAM's already problematic scaling limitations. Persistent memory prototypes promise to increase memory capacity by an order of magnitude, thereby helping applications (distributed or not) scale their performance.

Because of their scalability, some persistent-memory-based prototypes can accommodate more physical memory per server.[5] Because larger memory would let applications store more data, those now bottlenecked by memory capacity might see gains right away with little or no modification. Analytics and caching-like workloads, which load data into memory temporarily and thus have no durability requirements, can benefit immediately from this larger capacity. Persistent memory's low read latency would enable read-heavy workloads to run at speeds comparable to those possible with DRAM.

Byte-addressable persistence

In persistent-memory-based devices, hardware-level data access is at a much finer granularity than a block, which simplifies the job of runtime systems that help applications' in-memory data become persistent. A simple data flush from the CPU caches to persistent memory ensures that the application's in-memory data persists. Application runtimes need not implement complex locking and callback mechanisms, traditionally used to mask the latency of synchronized writes over block-oriented nonvolatile media. However, ensuring consistency will require more hardware and software support.

Low-latency durability

The ability to make data durable at a low latency would help transactional applications reduce data/work loss during crashes. Reducing the latency of a durable write would also reduce the number of outstanding transactions waiting purely for the write to complete. Such an improvement would not only lower individual transaction latency, but also the loss of work done because fewer outstanding transactions would be waiting just for the writes to become durable. Additionally, the faster transactions commit, the faster they can renounce their locked resources, which would improve the performance of applications with a high concurrency.

With low-latency durability, OSs and applications will be able to provide instant power-up and shutdown features. If the system caches the state needed to execute a program in persistent memory before the reboot, the latency of spinning up a program after a reboot could be as low as when executing a context switch. The advent of persistent memory would also blur the gap between sleep and hibernation, which could reap greater energy savings for mobile systems.

ADAPTING STORAGE TECHNOLOGIES

To cache frequently used content, storage technologies like file systems and databases must rely on DRAM. To ensure data durability, these systems must implement write-through mechanisms to bypass the cache so that the data can persist on traditional nonvolatile media, such as disk or flash drives. With persistent memory's byte addressability, these systems will be able to readily repurpose their DRAM-based caches to work with the new technology, and nonvolatility would eliminate the need for a write-through to ensure data persistence. A file system or a database might also use persistent memory as a standalone data location without any backup nonvolatile media.

Although these systems can benefit from nonvolatility, they must avoid data persistence in an inconsistent state. In addition, to ensure persistent memory's rapid adoption, applications that use these systems must continue to work with no or minimal modifications.

File systems

File systems have traditionally used disk and flash technology to ensure application data persistence. Adequately supplementing file systems can exploit persistent memory's byte addressability as well as ensure consistency. BPFS,[6] a file system for byte-addressable persistent memory, exploits byte addressability by using ordered and atomic writes on word-size regions. The system can then use these writes to avoid the copy-on-write and write-ahead-logging of file-system blocks—both of which are unnecessary when changing only a small block region. Thus, when used with persistent memory, BPFS gets a significant performance boost over other file systems. BPFS also requires no modification: applications continue to open, read, write, seek, and close files across the system.

File system designers could also build persistent caches for data that reside primarily on disk or flash chips. Because these caches can retain data across reboots, the cache need not write through to disk or flash to provide durability and consistency. Rather, when updating the journal, the system could permanently move all write-ahead logging and journaling of such file-system data to persistent memory, thus avoiding high latency.

Traditional file systems are not designed to operate efficiently over low-latency devices.[7] Although flash and disk latencies are large enough to mask software costs, persistent memory's access latency is small enough to make traditional storage interfaces counterproductive.

Moneta[7] is a recently developed storage system that provides user-space applications with direct access to persistent-memory-based virtualized disks. In addition, Moneta separates security policy from enforcement. The OS maintains the policies, while Moneta enforces them at the hardware level, thereby significantly reducing the latency of file system operations. Additionally, Moneta's developers have shown that using synchronous I/O operations will improve performance because latency is low enough that the benefit from asynchronous I/O does not outweigh the overhead from a context switch. Other recent research suggests that polling for an I/O completion might be more beneficial than waiting for an interrupt from a storage device based on persistent memory.[8]

Although systems like Moneta substantially reduce the performance overhead from enforcing the file system's security policies, open problems remain. One is the need for some mechanism to provide safe, consistent, and efficient updates to file-system metadata—particularly for files shared across multiple applications. Without efficient software and hardware support for metadata updates, file systems will not be able to provide low-latency operations to persistent memory.

Databases

Databases help applications preserve consistency while enabling rich data-processing capabilities. To maximize persistent memory's benefits, designers can take a page from previous work to optimize in-memory (DRAM-only) databases like Hekaton[9] by exploiting DRAM's byte addressability. Persistent-memory-only databases can exploit byte addressability in a similar manner. A finer granularity will potentially enable the system to process more queries in parallel, thus boosting database query performance—particularly for online transactional processing (OLTP) workloads. Additionally, databases using only persistent memory (without a backup disk) could improve performance by reducing reliance on synchronous logging to disk and flash media to provide durability guarantees. In that respect, databases could be both in-memory (high performance) and disk-like (durability); however, to provide high availability, they must still rely on replication.

Similar to file systems, disk-based databases can use persistent memory as a cache to support larger workloads. To optimize performance, traditional databases use DRAM caches for read-only transactions. Buffer caches based on persistent memory could support write transactions (with

Similar to file systems, disk-based databases can use persistent memory as a cache to support larger workloads.

write-back support to disk) while providing atomicity, consistency, isolation, and durability (ACID) guarantees. Other database caching techniques could also benefit. Systems like Memcache (http://memcached.org), for example, can build persistent query-output caches to ensure that the cache is hot across reboots and crashes.

Whether as a stand-alone database or as a buffer or query-output cache, persistent memory's ability to atomically modify data in place for small updates, as opposed to using copy-on-write, will enable persistent memory-based databases to perform better than their disk or flash counterparts, particularly for OLTP workloads. For databases that must ensure consistency in case of power loss, this atomicity capability can also reduce the load on the logging process or eliminate it altogether. Providing these benefits requires no modification of the database's Structured Query Language (SQL) interface or the data's relational model.

ADAPTING MEMORY TECHNOLOGIES

As flash technology matures, more applications, including file-system and database caches, will begin to use it directly as memory (through virtual memory), as opposed to accessing it as storage (block interface). Flash is similar to disk technology because of its block-addressed nature, and it is similar to DRAM because of a lack of seek latency. Relative to disk technology, flash offers a significantly higher number of I/O operations per second at a much lower latency, which makes adapting memory technologies to use flash a good approach for memory-intensive

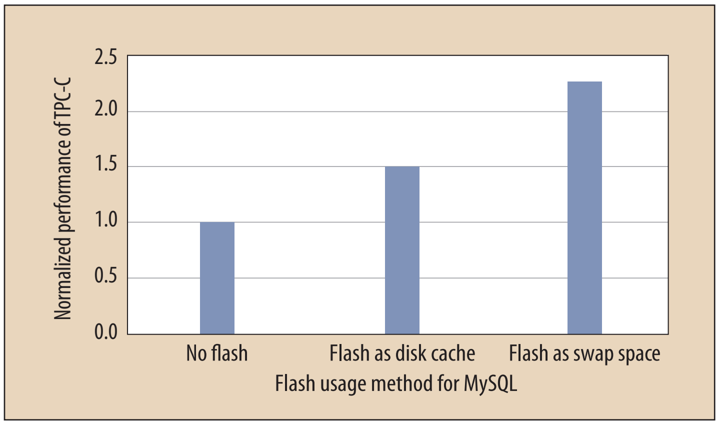

Figure 2. Comparison of flash adoption strategies with MySQL. Using flash as a swap space—the DRAM augmentation, or slow-memory approach—outperforms using flash as a disk cache—the fast-disk approach. TPC-C: Transaction Processing Performance Council's transaction-processing benchmark, version 5.11.

systems. Additionally, flash's nonvolatility can provide durability for applications' in-memory data. Finally, systems that adapt to using flash to augment DRAM can readily adopt PRAM when it arrives.

Slow-memory vs. fast-disk adoption

In studying flash memory adoption at Microsoft Research, my colleagues and I demonstrated the performance advantage of a *slow-memory* approach—DRAM augmentation—over a *fast-disk* strategy—using flash as part of the storage subsystem.

Figure 2 shows how using flash as a high-speed swap provides better performance than using it as a transparent disk cache. In one setting, we configured the SSD as a transparent block cache for the disk where the database resides. In another setting, we configured a flash-based SSD as memory extension using SSDAlloc,[10] enabling MySQL, an open source database, to use flash as a buffer cache.

For the same amount of flash, the buffer cache provides higher performance than the disk cache. This is not unexpected: data structures designed for DRAM have a high concurrency and finer-grained locking, making them more appropriate for new memory technologies than the ones designed for disks, which have a large seek latency and data locking at the block level.

Applications with separate memory and storage components that must leverage existing code will reap more performance benefits with the slow-memory approach than the fast-disk one—particularly for persistent memory packaged to be directly addressable by the CPU as opposed to packaged as an SDD. The slow-memory approach avoids the software overhead that the fast-disk approach would incur.

Although the slow-memory approach provides performance benefits, existing applications cannot use it to exploit persistent memory's low-latency durability. Because existing applications treat memory as volatile, data persistence is possible only through software interfaces to nonvolatile media. For applications that must leverage persistent memory's low-latency durability, the fast-disk approach is more suitable but the software requires optimization.

The slow-memory approach can exploit software latency improvements in systems like Moneta by using persistent-memory-based disks as a swap. Systems that move virtual memory pages between DRAM- and persistent-memory-based disks[10,11] on demand would also encounter software overhead when using the file system to serve page faults. However, such systems cannot exploit persistent memory's nonvolatility without additional software support.

Virtual memory management

Memory management is critical to persistent memory's adoption because it enables applications to take advantage of persistence with minimal code modification. If physical memory managers can distinguish between pages from DRAM and pages from persistent memory, they could give applications the flexibility to choose between them. Many approaches are possible in deciding which of these memory types a particular virtual memory page resides on.

Application-driven persistence. At one extreme, applications provide all the necessary information to implement such a separation. Two recent examples are Mnemosyne[12] and NV-Heaps[13]—virtual memory management systems that help applications differentiate among DRAM, persistent memory, and flash when allocating new pages for their heap or stack. Similar to battery-backed DRAM-based systems,[3] both Mnemosyne and NV-Heaps help applications store some state in a consistent and durable fashion.

Mnemosyne enables applications to declare static variables with values that persist across restarts. It also lets the application allocate memory from the heap, which is backed by persistent memory. Applications can define lightweight transactions to atomically modify several of these persistent variables at a time on the stack or the heap.

NV-Heaps supports the application in separating volatile and nonvolatile data, which ensures application consistency. For example, applications can make certain that no pointers exist from persistent memory to DRAM space, which ensures the persisted state's consistency after a reboot. With NV-Heaps, applications can name heaps to make programming with persistent media easier. However, naming conventions for persistent heaps and stacks must improve to enable the richer data indexing, sharing, and privacy and security features that file systems provide.

Both Mnemosyne and NV-Heaps expect the application to provide additional information to exploit persistent memory's benefits, but these systems use careful interface design to minimize such modifications, which allows for quicker adoption of this new technology.

Whole-system persistence. At the other extreme, applications are expected to work as they do now with some other mechanism ensuring data persistence in their heap and stack. In the proposed whole-system persistence[14] approach, the system provides a residual energy window to flush the contents of CPU registers and caches to persistent memory when a power failure occurs. Researchers have implemented this solution at the OS level with no special hardware support. Such a feature would provide suspend and resume functionality and require no modifications to the application to exploit the persistence. However, all system memory must be persistent—DRAM cannot be used.

Hybrid approaches. Many settings fall between these two extremes. In a hybrid approach, the default might be similar to whole-system persistence, but the application can provide hints to improve efficiency, which allows volatile data to be stored in DRAM. For example, it might proactively declare some parts of the data to be volatile to help reduce the amount of state that needs to persist in a power loss. Other hybrid approaches might involve clever ways to periodically sync data from DRAM to persistent memory to limit the amount of dirty state that resides in volatile media (DRAM and processor caches) at any time.

MEMORY-MAPPED FILES

Memory-mapped files reside at the border of memory and storage. Persistent-memory-based page caches can help improve the performance of applications that rely on memory-mapped files. Persistent caches can maintain the cache to always be hot even after a reboot. The recently proposed FlashTier[15] builds such a cache using flash, offering techniques that are easy to mimic for memory-mapped files using persistent memory.

Applications that already use memory-mapped files can exploit these new benefits without modification, but they could also obtain other benefits by providing hints relevant to persistent memory. For example, a memory-mapped file might be further empowered by providing the application with BPFS-like ordered writes and atomic word-size writes, which would help transactional applications maintain a consistent state at a low overhead.

Application designers could place some blocks from memory-mapped files in DRAM and map the rest in persistent memory—a strategy that exploits DRAM's lower latency. The OS then has the additional responsibility of choosing a physical memory technology for each page mapped to a file, although the application could provide hints about what properties it expects from a particular mapping. In this way, the OS and application can cooperate to appropriately migrate mappings between DRAM and persistent memory to meet an application's changing needs from a memory-mapped file.

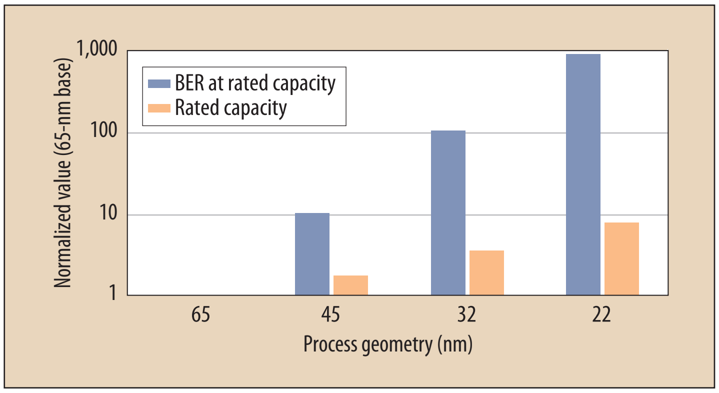

Figure 3. Bit error rate (BER) as a function of capacity. Capacity increases are not sufficient to offset higher flash BERs. This trend argues strongly for aggressively seeking solutions to increase persistent memory's lifespan.

NEW CHALLENGES AHEAD

Some of persistent memory's benefits must come at the cost of serious work to address open problems in data reliability and in security and privacy.

Data reliability

Many promising prototypes that exhibit persistent memory's characteristics have a considerably shorter lifetime than DRAM:[5,16] in other words, once the system has written a persistent memory region some number of times, that region can no longer hold data reliably. Although flash chip capacity has been steadily increasing because of process geometry shrinkage, bit error rates are increasing even faster.[17] Consequently, the amount of data that the system can write to flash before an error occurs (normalized per dollar) is falling. If the flash trend in Figure 3 is any indicator, persistent memory's short lifespan problem will only worsen. Therefore, aggressive steps must be taken to tackle this problem early on.

Many proposed hardware solutions address this problem[18,19] by either masking all the hardware errors from software or exposing the errors and letting the software handle them in a way that makes the most sense for the OS or application.

Careful design to make hardware and software cooperate might increase a solution's overall effectiveness. An example is dynamically replicated memory (DRM),[20] which relies on cooperation between hardware and software to mask persistent memory failures. Current approaches retire persistent memory pages even when the page has only a single failed bit. DRAM views two pages that fail in different regions as a single logical page, relying on changes to the memory controller, and translation look-aside buffer (TLB) management inside the OS to be aware of such mappings.

Although DRM opens the door for handling persistent memory failures at a higher level, managing these failures at the application level will entail much additional research, since the application's reliability requirements from memory might not be as strict as what the hardware imposes.

Security and privacy

Persistent memory creates new security and privacy challenges, particularly for mobile systems. If execution state is stored in persistent memory, then the application and OS must be able to erase sensitive data or encrypt it when that data is no longer needed for computation. Cold-boot attacks can easily steal sensitive information from data in persistent memory when the machine is lost or stolen. To thwart such attacks, security and privacy policies could potentially dictate if certain data can reside unencrypted in persistent memory.

Security and privacy policies could also determine how and when the OS erases and encrypts such data when the application has finished using it. Application designers can implement these policies by allowing applications to differentiate between DRAM and persistent memory when storing sensitive information. If the application stores sensitive data in persistent memory during memory allocation, it could specify the way to maintain that data.

A simpler backward-compatible solution that provides the current privacy level would, on every shutdown, simply erase from persistent memory the contents of every application that does not specify a privacy or durability policy. However, such applications would be exploiting persistent memory purely for its energy-efficiency and high-capacity benefits, not for its nonvolatility. Such a model provides developers with incentives to differentiate between DRAM and persistent memory when absolutely necessary for security and privacy.

So far, the research community has only scratched the surface of possible persistent memory use within existing storage and memory systems. A complete overhaul of all systems might be required in the distant future, but in the meantime, each system could devise a backward-compatible or less intrusive solution to embrace persistent memory. Such solutions will involve major changes to hardware and the OS, but only minimal changes to application-level interfaces. This incremental process will enable quicker and wider adoption of promising new memory technology. C

References

1. K. Bailey et al., "Operating System Implications of Fast, Cheap, Non-Volatile Memory," *Proc. 13th Usenix Workshop Hot Topics in Operating Systems* (HotOS 11), Usenix, 2011; www.usenix.org/legacy/event/hotos/tech/final_files/Bailey.pdf.
2. J. Ousterhout et al., "The Case for RAMClouds: Scalable High-Performance Storage Entirely in DRAM," *Operating Systems Rev.*, vol. 43, no. 4, 2010, pp. 92-105.
3. D.E. Lowell and P.M. Chen, "Free Transactions with Rio-Vista," *Proc. 16th Symp. Operating Systems Principles* (SOSP 97), ACM, 1997, pp. 92-101.
4. "Cisco Visual Networking Index: Global Mobile Data Traffic Forecast Update, 2012-2017: Executive Summary," Cisco; www.cisco.com/en/US/solutions/collateral/ns341/ns525/ns537/ns705/ns827/white_paper_c11-520862.html.
5. C.W. Smullen et al., "Relaxing Non-Volatility for Fast and Energy-Efficient STT-RAM Caches," *Proc. 17th IEEE Int'l Symp. High-Performance Computer Architecture* (HPCA 11), IEEE, 2011, pp. 50-61.
6. J. Condit et al., "Better I/O through Byte-Addressable, Persistent Memory," *Proc. 22nd Symp. Operating Systems Principles* (SOSP 09), ACM, 2009, pp. 133-146.
7. 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.
8. J. Yang, D.B. Minturn, and F. Hady, "When Poll Is Better than Interrupt," *Proc. 10th Usenix Symp. File and Storage Technologies* (FAST 12), Usenix, 2012; www.usenix.org/legacy/events/fast12/tech/full_papers/Yang.pdf.
9. C. Diaconu et al., "Hekaton: SQL Server's Memory-Optimized OLTP Engine," *Proc. SIGMOD Int'l Conf. Management of Data* (SIGMOD 13), ACM, 2013, pp. 1243-1254.
10. A. Badam and V.S. Pai, "SSDAlloc: Hybrid SSD/RAM Memory Management Made Easy," *Proc. 8th Usenix Symp. Networked Systems Design and Implementation* (NSDI 11), Usenix, 2011; www.usenix.org/legacy/events/nsdi11/tech/full_papers/Badam.pdf?CFID=343857637&CFTOKEN=34381778.
11. M. Saxena and M.M. Swift, "FlashVM: Virtual Memory Management on Flash," *Proc. Usenix Ann. Tech. Conf.* (ATC 10), Usenix, 2010; www.usenix.org/legacy/event/atc10/tech/full_papers/Saxena.pdf.
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.
13. J. Coburn et al., "NV-Heaps: Making Persistent Objects Fast and Safe with Next-Generation, Non-Volatile Memories," *Proc. 16th Int'l Conf. Architectural Support for Programming Languages and Operating Systems* (ASPLOS 11), ACM, 2011, pp. 105-118.
14. D. Narayanan and O. Hodson, "Whole-System Persistence," *Proc. 17th Int'l Conf. Architectural Support for Programming Languages and Operating Systems* (ASPLOS 12), ACM, 2012, pp. 401-410.
15. M. Saxena, M.M. Swift, and Y. Zhang, "FlashTier: A Lightweight, Consistent and Durable Storage Cache," *Proc. 7th European Conf. Computer Systems* (EuroSys 12), ACM, 2012, pp. 267-280.

16. B.C. Lee et al., "Architecting Phase Change Memory as a Scalable DRAM Alternative," *Proc. 36th Int'l Symp. Computer Architecture* (ISCA 09), ACM, 2009, pp. 2-13.
17. L.M. Grupp, J.D. Davis, and S. Swanson, "The Bleak Future of NAND Flash Memory," *Proc. 10th Usenix Symp. File and Storage Technologies* (FAST 12), Usenix, 2012; http://cseweb.ucsd.edu/users/swanson/papers/FAST2012BleakFlash.pdf.
18. S. Schechter et al., "Use ECP, Not ECC, for Hard Failures in Resistive Memories," *Proc. 37th ACM/IEEE Int'l Symp. Computer Architecture* (ISCA 10), IEEE, 2010, pp. 141-152.
19. M.K. Qureshi et al., "Enhancing Lifetime and Security of Phase Change Memories via Start-Gap Wear Leveling," *Proc. 42nd IEEE/ACM Symp. Microarchitecture* (MICRO 42), IEEE, 2009, pp. 14-23.
20. E. Ipek et al., "Dynamically Replicated Memory: Building Reliable Systems from Nanoscale Resistive Memories," *Proc. 15th Int'l Conf. Architectural Support for Programming Languages and Operating Systems* (ASPLOS 10), ACM, pp. 3-14.

***Anirudh Badam** is a researcher at Microsoft Research. His research interests include problems in distributed systems and OSs pertaining to server and middle-box scalability. Badam received a PhD in computer science from Princeton University. He is a member of ACM, IEEE, and Usenix. Contact him at anirudh.badam@microsoft.com.*