



# What the Future Holds for Solid-State Memory

Karin Strauss and Doug Burger, *Microsoft Research*

**The memory industry faces significant disruption due to challenges related to scaling. Future memory systems will have more heterogeneity at individual levels of the hierarchy, with management support from multiple layers across the stack.**

**T**he computer industry's phenomenal growth over the last half-century could not have been achieved without continuous improvement in solid-state memory density—a direct result of scaling chip memory to smaller cell sizes. But growing application storage and memory requirements, along with the demand for big data in an increasing number of domains, will pose enormous future challenges.

Think about memory for a moment: each chip contains billions of bits—working constantly (or nearly so)—stored in tiny memory cells that each holds a relatively small number of electrons. The semiconductor industry has done an amazing job so far of developing memory technologies, such as DRAM and flash, that provide this abstraction of stable, precise, “always correct” memory, with durability guarantees comparable to, or many times longer, than other system components. Under the covers, however, increased density requires copious mechanisms to compensate for failures. In DRAM, for example, refreshes maintain constantly leaking data values while error correction codes catch and correct bits that occasionally flip (soft errors); flash uses a translation layer to delay and hide transient and permanent (hard) errors. Up to now, innovative fixes like these have made possible rapid advances in the semiconductor industry.

Unfortunately, scaling memory to ever-higher densities while maintaining the “always correct” interface is becoming so difficult and costly that radical departures from current practices will likely have disruptive implications for software and system designers. As cells in charge-based memory get smaller, they also become less stable; even the smallest disturbances on their stored charge might be sufficient to change their state, resulting in decreased reliability. Possible solutions include new—but costly—fabrication techniques, 3D cell stacking, alternative storage principles, more heterogeneous memory hierarchies, and enhanced system support that will tolerate degraded memory and increased error rates.

Here, we describe some of the challenges to be overcome in scaling memories to higher densities, and predict even more heterogeneous memory hierarchies (including special-purpose memories) in the future that will require new levels of cooperation between hardware and software and significant changes in applications, development tools, and system software support.

## MEMORY CLASSES AND CHARACTERISTICS

Memory has two main classes: working and storage. Working memory temporarily buffers inputs, intermediary values, and results during active computation; examples include DRAM, used as main memory, and SRAM, used for on-chip caches. Both technologies use charge to hold a stable state, but DRAM is based on storing charge (electrons) in memory cells, while SRAM uses carriers temporarily stored in circuit nodes to stabilize a cross-coupled circuit. Storage memory saves data for later reference and processing or as backing storage when contents do not fit in working memory. Magnetic hard disk drives (HDDs) have traditionally been used for this purpose, but in the past 10 to 15 years, flash, a charge-based memory technology,

**Table 1. Basic characteristics of memory technologies.**

Characteristic	Description	Working memory (DRAM)	Storage memory (flash)
Capacity	How many bits of data it can store	Gbytes	Hundreds of Gbytes
Granularity	How much data is read/written in one access	Bytes	Kbytes
Read latency	How fast data can be read from memory	Nanoseconds	Microseconds
Write latency	How fast data can be written into memory	Nanoseconds	Microseconds
Retention	How long cells can store uncorrupted data	Microseconds	Years
Durability	How many writes on average before a cell fails	10 <sup>15</sup>	10 <sup>4</sup> -10 <sup>5</sup>

has emerged as a crucial component in storage memory systems. Flash was initially introduced in the mobile market—thumb drives, memory cards for digital cameras and other gadgets, and music players—but has made its way up the device chain, with flash-based solid-state drives (SSDs) now displacing HDDs in larger systems such as high-end laptops and even appearing side by side with HDDs in datacenter servers.

Table 1 provides a basic overview of these two types of memory. Essentially, working memory—such as DRAM—is byte-addressable and volatile, with lower capacities and lower access latencies; it functions to allow processors to quickly manipulate elements at a fine granularity for a relatively limited amount of time, after which they are saved in more stable storage memory (or no longer needed). Storage memory—such as flash—has higher capacity and is nonvolatile, with retention times on the order of years; low access latencies are not as critical for storage memory, and coarser access granularities are acceptable because data goes into working memory before being processed. Durability is important for both because both types of memory need to last roughly as long as the computing device that employs them. Energy concerns should be considered as well, but these are highly device- and application-specific: the energy and power requirements of a smartphone, for example, are very different from those of a datacenter server.

## NEW CHALLENGES

Both DRAM and flash memory are charge-based: charges (electrons) are forced into their cells to store data, and voltages are used to sense the value stored in the cell. Electrons can be moved with low energy, which is both good and bad: this mobility enables cells to be read and written quickly and with little energy expenditure, but it makes long retention a challenge because electrons can easily escape cells. Flash has been designed to prevent this by adding a thick oxide insulator layer to the cells that holds charge longer, but this requires shooting electrons through the oxide to program the cells, creating a durability problem because the process leads to oxide degradation, eventually affecting cells' charge-holding capacity.

Achieving increasingly higher densities requires shrinking cells to smaller sizes. As cells shrink, they hold proportionately smaller charge. Consequently, the charge of a single electron represents a relatively higher proportion of the total charge stored by the cell, and a small number of electrons escaping or entering a cell results in a corruption of the cell's stored value. Manufacturing variation—that is, cell-to-cell fabrication differences—aggravates this problem in relatively smaller cells. These differences affect, for example, cell geometry and composition, so certain cells might be able to hold many fewer charges or experience oxide breakdown much earlier than others.

Historically, the memory industry has innovated to address these problems by focusing on the memory technologies themselves—specifically, by using different materials, manufacturing processes, or cell geometries. Increasingly, some desirable characteristic is sacrificed to obtain higher densities. A notable example is, rather than storing a single bit per cell (single-level cells, or SLCs), storing multiple bits in a single flash cell (creating multilevel cells, or MLCs) by dividing the analog range of voltages sensed to read and write the cell into more than two regions. This division increases the memory's usable levels and thus density, but causes faster wear and thus lower durability. Another example is refreshing DRAM: left untouched, DRAM cannot maintain the values it stores, so cells have to be periodically read and rewritten to make sure the appropriate charge levels are maintained, but these continuous refreshes reduce performance and consume additional energy.

Without an obvious solution for efficiently retaining electrons in smaller DRAM cells, the overhead of handling memory errors could become prohibitively high. For example, increasing refresh rates might consume so many memory cycles that too few cycles remain for actual reads and writes. Increased error-correction capability, another common solution used in the server DRAM memory market, requires greater design complexity and additional storage for more complex codes, which could offset any density gains and result in higher cost. Error correction in working memory also raises issues of latency:

**Table 2. Alternative memory technologies, their storage principles, and specific storage mechanisms.**

Technology	Principle	Storage mechanism
Hybrid memory cube (HMC)	Electronic (charge-based)	Multiple layers of DRAM technology chip-stacked on top of a high-performance logic layer; trades total memory capacity for better performance
Vertical flash	Electronic (charge-based)	Charges are trapped in a floating-gate transistor; cells are vertically integrated (3D), significantly increasing density and allowing the use of larger cells
Phase-change memory (PCM)	Atomic (resistive)	Cell material can be crystalized or put into an amorphous state by controlled heating and cooling; material has different resistances when crystalline or amorphous
Memristors	Atomic (resistive)	Memristors use a thin film of materials such as titanium dioxide; applying high currents moves oxygen vacancies around the film, changing its resistance
Conductive-bridging RAM (CB-RAM)	Atomic (resistive)	Metal ions in a cell migrate when a current is applied and form a conductive path within a nonconductive material; this changes the resistance of the cell
Spin-transfer torque memory (STT-RAM)	Magnetic (resistive)	Cell includes a permanent and a floating ferromagnetic material; the polarity of the latter can be changed by a polarized electrical current, and their relative alignment can be determined by running a current through them and observing their resistance
Racetrack memory	Magnetic (resistive)	Multiple ferromagnetic domains share a smaller set of read/write ports; to be read or written, these domains have to be shifted into the port region

the performance requirements of working memory are much more demanding than those of storage memory; reliability solutions that work for storage memory, such as the addition of a translation layer, are not suitable for working memory due to their long latencies.

Another challenge for working memory, especially in the server market, is providing sufficient memory bandwidth to processing cores. Multicore CPUs have a greater number of processing units per chip. Unfortunately, memory technology has not kept up with the increasing memory bandwidth needs created by these additional cores, and the resulting sharing of memory resources among cores results in inefficiencies.

## IMPACT OF ALTERNATIVE TECHNOLOGIES

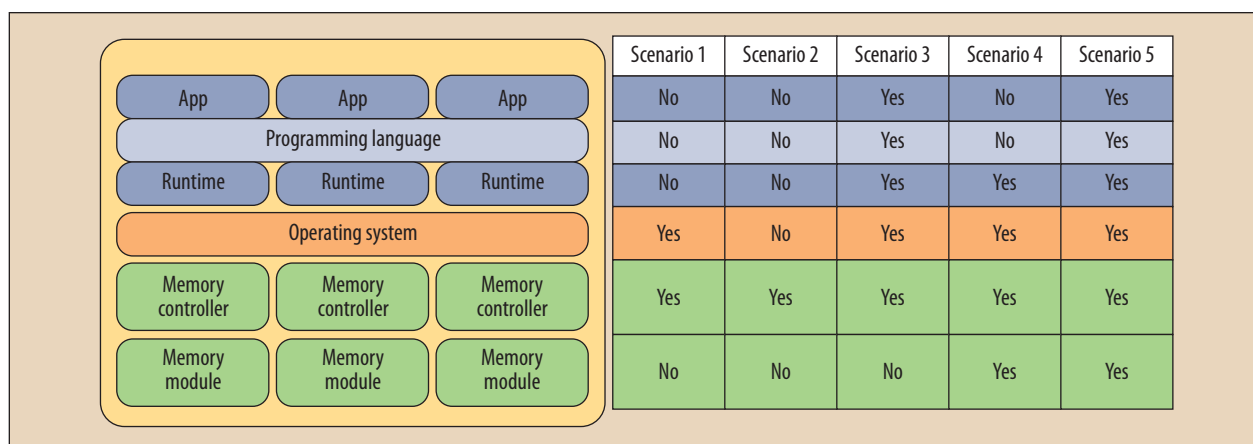
To address such challenges, the memory industry and its academic counterparts are developing new memory alternatives. Table 2 shows several proposed candidates to replace today's working and storage memory technologies, along with the principle each uses for storing information and its specific storage mechanism.

The path currently pursued by charge-based manufacturers for both DRAM (hybrid memory cube; HMC) and flash (vertical flash) is to stack memory vertically, thereby gaining density per area, improving cost per bit, and improving performance. Atomic (resistive) memory technologies are based on changing the physical configuration of atoms or vacancies within a cell, thus affecting the cell's resistance. Magnetic (resistive) memory models are based on changing the relative orientation of a floating-orientation ferromagnetic material with respect to a fixed-orientation material, which again affects cell resistance. We leave out other storage approaches, such as photonic storage, as we are not aware of any current, practical large-scale

implementations for them. Yoon-Jong Song and his colleagues offer detailed comparisons of some of these technologies (PCM, STT-RAM, and CB-RAM) elsewhere.<sup>1</sup> Additional information on competing technologies is available from other publications.<sup>2-6</sup>

Vertical flash is the likely technology to be used as storage memory because its manufacture most resembles the flash memory used today. Its higher density, which results from using a third dimension, will allow manufacturers initially to use larger cells, temporarily mitigating the reliability problems described earlier. Density scaling can continue for some time by increasing the number of cells vertically, but manufacturing costs will ultimately limit the number of layers. Once this limit is reached, manufacturers will have to resume reducing cell sizes, incurring the associated challenges. After that, we speculate that atomic memories will follow, assuming that their large-scale fabrication costs can be reduced to levels comparable to those of flash. Two articles in the August 2013 issue of this magazine cover the impact of these storage-memory improvement technologies on software systems.<sup>7,8</sup>

Solutions for working memory are more difficult to predict because of its stricter performance requirements. In the short term, we believe that HMC arrays are likely to be adopted in server markets due to their increased performance. However, HMC's lower capacity might mean that it will not replace current DRAM chips and memory modules and will instead be used side by side with them as working memory. Longer term, each candidate technology poses challenges yet to be surmounted. Magnetic memory has desirable characteristics, but high densities have proven difficult to achieve and are still not comparable to DRAM densities. Atomic memory technologies seem to have



**Figure 1.** Five scenarios based on a hypothetical system, and system layers (left) affected in each scenario (right). In scenario 1, the operating system is responsible for allocating three types of memory, each with its own advantages and disadvantages. Scenario 2 shows that protecting nonvolatile data from cold-boot attacks might require modifications to the memory controller. Scenario 3 shows that exposing nonvolatility via loads and stores to the regular application address space could require modifications to multiple system layers. Scenario 4 shows how exposing permanent failures can affect multiple system layers. Finally, Scenario 5 shows layers affected by exposing performance versus reliability tradeoffs to programmers.

good density scaling and reasonable read latencies, but because their storage principle is based on changing the memory cell's atomic configuration, the time and energy required to program these cells might be prohibitively high. Furthermore, programming strains cell material, causing degradation and resulting in premature wear that gets cells stuck at high or low resistance and could compromise device durability. Certain mechanisms developed for DRAM, such as certain error-correcting codes, may aggravate this problem, reducing atomic memory durability even further.

## A MORE COMPLEX FUTURE

Despite their likely ability to scale to higher densities, none of the multiple contenders for working memory match DRAM on every relevant dimension, with the following implications:

- Memory component heterogeneity might create further system heterogeneity.
- Future memory technologies might sacrifice one dimension to satisfy others, which could expose certain properties beyond the memory module interface, requiring further cooperation across system layers.
- Multiple points of operation for the same memory, either fixed at design time or tunable at runtime, might be required.

These developments could in turn have repercussions at multiple layers of the system stack: memory interfaces, hardware memory controllers, the operating system, and runtimes, as well as for languages and how applications are written. As engineers and designers, we must decide where most profitably to expose information and tunable knobs

to the rest of the system. Greater exposure generates more opportunity but also introduces more complexity and disruption into the computing and memory ecosystem.

To illustrate these implications and their resulting tradeoffs, we describe a hypothetical system that uses three memory technologies side by side as components of total working memory: standard DRAM, HMC (a higher-performance, lower-capacity memory), and MLC phase-change memory (PCM; a higher-capacity, lower-performance memory that wears out as it is written). The three occupy the same flat physical address space; consequently, this organization requires more support across the system stack than traditional working memory does. Figure 1 shows such a hierarchy (bottom left: three types of memory modules: DRAM, HMC, and PCM), along with the pertinent system layers (middle and top left: memory controllers, operating system, runtimes, programming language, and applications), and identifies which of these layers must offer additional support for each of five scenarios, the requirements of which we will describe more fully later.

## Memory heterogeneity furthering system heterogeneity

Increased heterogeneity is the most straightforward implication of multiple emerging memory technologies. As a variety of memory technologies become available—and lacking a single technology that performs optimally across all relevant dimensions—designers might choose to equip systems with multiple memory types, which combine to form a more effective working memory. These would differ from the strictly linear hierarchies we are used to, which employ only one type of working memory within a system.

In this world, software might have to be tailored in order to take advantage of different memory characteristics

**Table 3. Some existing proposals for hardware-only wearable memory management.**

Proposed solution	Cell type	Mechanism
Differential writes	SLC/MLC	Writes only those cells that have changed values
Flip-n-write	SLC	Flips groups of bits if flipped group requires fewer bit flips to write than original group
Dynamically replicated memory	SLC	Once a block in a page wears out, pairs pages to make two bad pages into one good page
Start-gap wear leveling	SLC/MLC	Rotates blocks within pages to spread wear more uniformly and prevent software attacks that intentionally direct excessive wear to certain regions of memory
Error-correcting pointers	SLC	Uses error-correction metadata storage in block to point to worn bits and replace them
Stuck-at-fault error recovery	SLC	Divides block into areas that contain only one worn-out bit and flips all bits in the area if needed to match the value the cell is stuck at
Fine-grained remapping with error-correcting code and embedded-pointers	SLC	Points to a replacement block from a worn-out block
Pay-as-you-go	SLC	Uses error-correction metadata storage to replace first failed bit, but then uses a common pool of replacement entries for any further worn-out bits
Coset coding	SLC	Provides multiple encodings for a data word, then uses a convenient encoding to minimize wear and tolerate errors
Zombie memory	SLC/MLC	Uses memory regions with uncorrectable worn-out bits to extend error-correction capabilities of blocks nearing end of life

across the hierarchy (Scenario 1 in Figure 1): time-critical applications with reasonably small memory footprints, for example, would use HMC for maximum performance; applications with much larger memory footprints, but that do not write to memory very frequently, would use PCM, which provides higher capacity and benefits from the absence of writes by avoiding the wear that would otherwise result. Write-intensive applications that do not fit in HMC might use regular DRAM. In a scenario like this, the operating system must be aware of the memory's heterogeneity so that it can allocate the best memory for each application and the memory controllers must coordinate to steer memory requests accordingly; no other system layer needs to be involved.

Scenario 2 in Figure 1 considers the nonvolatility dimension. While working memory has traditionally been volatile, PCM in our hypothetical system adds nonvolatility, which at this level starts to blur the line between working memory and storage memory and has implications across the stack—and particularly on system security: part of working memory is now nonvolatile, and thus much more susceptible to cold-boot attacks.<sup>9</sup> Countering these attacks might require additional system support, such as requiring that memory controllers encrypt data before storing it in PCM.

Another challenge for working memory nonvolatility is how to expose it to software. The most straightforward way is to expose PCM as a fast mass-storage volume, which requires only operating system support. Although relatively simple to implement, this solution misses some potential benefits, as the software stack is not tuned to fast storage and could introduce high access overhead.<sup>10</sup>

An alternative is to access working data as part of the virtual address space of applications and persist these data in place, without any need for serialization into a file (Scenario 3 in Figure 1). At first glance, this possibility seems ideal because data is “automatically” persisted, but, to be exposed to programmers, mapping nonvolatility directly into the virtual address space requires language support and carries several correctness risks. First, common software bugs such as dangling pointers and other memory-management errors can have persistent effects—restarting an application or a system might not make an abnormal state disappear. Second, if both volatile and nonvolatile memories are available and in use by the same application, additional pointer safety bugs arise. Imagine a pointer in nonvolatile memory that points to a datum in volatile memory; when the application is restarted, the volatile datum is gone and the nonvolatile pointer still points to it. Finally, recovering from unexpected system failures could be more complicated: even if applications are correctly written, events such as power failures can corrupt data. Some recent work in the area of nonvolatile heaps addresses these issues,<sup>11,12</sup> but we view this as a difficult, long-term problem.

### Exposing feature sacrifices beyond the memory module interface

Sacrificing some features to benefit other features is not new to the memory industry. SSDs built from SLC NAND flash guarantee longer endurances than higher-capacity, lower-endurance SSD successors built from MLC NAND



flash. Here, the sacrifice affects only the product specification because failures and their resulting fragmentation of memory are hidden by the flash translation layer (FTL). However, disruptive aspects of new memory technologies can be difficult to hide completely from software, such as wear-out in a PCM-based working memory.

### **Wearable memories and opportunities for cooperation**

We call memories subject to wear *wearable memories*. The issue of wear in atomic (as opposed to charge-based) working memories attracts our attention because it directly impacts the correct operation of a device, rather than just its performance or initial specification. If working memory wears out during the memory's lifetime and certain regions become unusable, this breaks the existing, convenient abstraction of a contiguous physical address space.

One could argue that the same problem afflicts flash, but wear only affects its product specification—the number of write cycles the flash module will last—not its correct operation. The reason shorter longevity affects only the product specification for flash is that flash is used as storage memory, and, as such, higher-access latencies are tolerable. The higher acceptable latencies make possible the inclusion of the FTL in the memory hierarchy, which provides an illusion of contiguous physical address space. The FTL detects and corrects errors with complex error-handling mechanisms and performs wear leveling with an additional level of indirection in addressing. This indirection also remaps accesses to healthy memory when certain portions of the flash memory become unusable due to wear. However, the additional latencies inserted by the FTL are unacceptable for working memory.

Table 3 shows multiple solutions proposed for detecting and correcting errors, as well as for performing wear leveling on wearable working memory, including a few we ourselves have proposed (along with our collaborators Engin Ipek, Jeremy Condit, Edmund Nightingale, Thomas Moscibroda, Stuart Schechter, Gabriel Loh, Rodolfo Azevedo, John Davis, Parikshit Gopalan, Mark Manasse, and Sergey Yekhanin).<sup>13-22</sup> The goal of these is to provide fast and durable working memory with low hardware complexity. However, these proposals focus on solving the reliability problem mostly in hardware, hiding as much as possible from software layers. In particular, none of the solutions performs the kind of aggressive remapping of failed blocks that FTLs do because the overhead would be prohibitive for working memory. Because arbitrary remapping may be too expensive, an alternative approach exposes working memory wear-out failures to software.

An example of cross-layer cooperation to manage wear-out comes from work we did with our colleagues Tiejun Gao, Steven Blackburn, Kathryn McKinley, and James Larus.<sup>23</sup>

Our work uses managed runtimes, such as those running C# or Java, to tolerate failures in wearable working memories (Scenario 4 in Figure 1). Because managed runtimes offer an abstracted view of memory, they are capable of allocating or moving objects to arbitrary locations. This capability allows managed runtimes to dodge “holes” in memory created by block failures due to wear. Another advantage is the granularity at which managed runtimes can handle memory. Objects can be quite small, which is a good match for the granularity at which unusable memory chunks are discarded, typically the size of a single cache block (64 bytes). This solution relies on the assumption that the system still has a region of memory that does not wear out (such as current-generation DRAM), so that the operating system and

---

**If working memory wears out during the memory's lifetime and certain regions become unusable, this breaks the existing, convenient abstraction of a contiguous physical address space.**

---

applications that do not tolerate memory failures can run seamlessly. Programs that can tolerate failures must indicate they are able to use memory that can fail during execution or that has already partially failed. The motivation to use such memory is that it will likely be cheaper and more plentiful.

This cooperation between hardware and software works as follows. The hardware can detect wear-out failures when it attempts to write back cached data into the wearable memory. If the intended value cannot be written, a failure is detected. The hardware then buffers this value, and the operating system is invoked via an interrupt. Note that wear-out failures result in unwritable bits, instead of the random flips common to DRAM. Once a wear-out failure is detected, the corresponding memory location will no longer be used, so reads will not expose new wear-out failures. The operating system is responsible for recovering the unwritten data from the hardware buffer. The operating system (already in charge of managing physical memory) must now maintain a failure map that specifies which memory blocks have failed. For memory allocations from a managed runtime, the operating system relays the failure map covering the memory being allocated. The managed runtime's memory manager then ensures that no objects are allocated on failed blocks.

Tolerating failures does not come without cost for managed runtimes. The location of failures is arbitrary, especially if wear leveling is in use, so failures will cause fragmentation. Contiguous regions of memory that could

once store an object might now be fragmented by failures, making it impossible to store that same object. Lower-level support can mitigate this problem; we devised hardware that can defragment itself within groups of pages, greatly decreasing fragmentation issues. Overall, for the benchmarks we used (DaCapo), the memory manager we modified (Immix) only incurs performance losses after failures occur. Even for high levels of failures (50 percent memory-capacity degradation), our experiments showed performance losses of only 12 percent on average.

### Static and dynamic selection of memory operating points

At design time, engineers can make multiple tradeoffs to obtain a memory part with the desired characteristics. For example, using different materials in PCM fabrication will affect read and write performance by modifying how easily the material can change its physical configuration. Once engineering teams have determined the material to be used and the memory has been fabricated, it cannot be changed, fixing read and write performance.


Other memory tradeoffs can change dynamically during memory operation. Along with our colleagues Adrian Sampson, Jacob Nelson, and Luis Ceze, we observed that it is possible to increase memory density if the application can tolerate small errors.<sup>24</sup> Certain applications such as sensor data processing, machine learning, and image processing have error-tolerant data structures; these applications can produce acceptable output even if some bits of error-tolerant data structures are incorrect (Scenario 5 in Figure 1). Some of these applications can have large capacity needs so there is an incentive to increase density at the cost of errors. To exploit larger but error-prone memories, we need cross-system support. Memory modules provide a “knob” that allows portions of memory to be tuned for higher density at higher error rates. Memory controllers expose this knob to the operating system, which must control this knob and how these portions of memory are allocated. Languages should allow application programmers to identify data that can be stored in these dense but error-prone portions of memory, and runtimes must be aware of and manage these different types of memory.

### System support for future working memory hierarchy

As noted earlier, multiple layers of the system stack, both hardware and software, are likely to be affected by future disruptions in semiconductor memory technologies. Designers must decide where to expose information such as performance, nonvolatility, and failures, and which knobs—for example, dynamic tradeoffs between density, reliability, performance, and power—to provide to software. Exposing optimizations only to the

lower-level layers is less disruptive to the system stack but requires that optimizations be widely applicable; those targeting specific cases will remain unexploited. Exposing more about memories to other system layers introduces complexity but could better leverage specific behaviors observed at these layers.

**M**any alternatives are possible for memory hierarchy designs and for the support provided to each, and in choosing among these alternatives, designers will have to find the right tradeoffs between functionality and complexity. We believe that the best designs will emerge from tight cooperation between multiple layers across the system and that the best tradeoffs will be made when these layers are designed in tandem.

Multiple experimental memory technologies are under development, creating tremendous uncertainty about how memory hierarchies will evolve. DRAM and flash face difficulties, but which technologies will succeed them and how these technologies might be organized remain unclear. We believe that systems will migrate toward a hierarchy of specialized memories—much like counterpart processing technologies migrated toward specialized processors. Rethinking the abstractions and increasing the cooperation among multiple software and hardware layers in the stack will be imperative to further memory density scaling. 

### References

1. Y.-J. Song et al., “What Lies Ahead for Resistance-Based Memory Technologies?,” *Computer*, Aug. 2013, pp. 30-36.
2. Y. Li and K.N. Quader, “NAND Flash Memory: Challenges and Opportunities,” *Computer*, Aug. 2013, pp. 23-29.
3. “International Technology Roadmap for Semiconductors 2011 Edition, Executive Summary,” Int’l Tech. Roadmap for Semiconductors, 2011; <http://www.itrs.net/Links/2011itrs/2011Chapters/2011ExecSum.pdf>.
4. S. Williams, “How We Found the Missing Memristor,” *IEEE Spectrum*, Dec. 2008, pp. 28-35.
5. S.S.P. Parkin, M. Hayashi, and L. Thomas, “Magnetic Domain-Wall Racetrack Memory,” *Science*, vol. 320, no. 5873, 2008, pp. 190-194.
6. “About Hybrid Memory Cube,” Hybrid Memory Cube Consortium, 2013; <http://hybridmemorycube.org/technology.html>.
7. A. Badam, “How Persistent Memory Will Change Software Systems,” *Computer*, Aug. 2013, pp. 45-51.
8. S. Swanson and A.M. Caulfield, “Refactor, Reduce, Recycle: Restructuring the I/O Stack for the Future of Storage,” *Computer*, Aug. 2013, pp. 52-59.
9. K. Bailey et al., “Operating System Implications of Fast, Cheap, Non-Volatile Memory,” *Proc. 13th Workshop Hot Topics in Operating Systems (HotOS 11)*, Usenix, 2011, [https://www.usenix.org/legacy/events/hotos11/tech/final\\_files/Bailey.pdf](https://www.usenix.org/legacy/events/hotos11/tech/final_files/Bailey.pdf).
10. A.M. Caulfield et al., “Providing Safe, User Space Access to Fast, Solid State Disks,” *Proc. 17th Int’l Conf. Architecture*

Support for Programming Languages and Operating Systems (ASPLOS 12), ACM, 2012, pp. 387–399.

11. 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. 105–117.
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. 91–103.
13. S. Schechter et al., “Use ECP, not ECC, for Hard Failures in Resistive Memories,” *Proc. 37th Annual Int’l Symp. Computer Architecture (ISCA 10)*, ACM, 2010, pp. 141–152.
14. D.H. Yoon et al., “FREE-p: Protecting Non-Volatile Memory against both Hard and Soft Errors,” *Proc. 2011 IEEE 17th Int’l Symp. High Performance Computer Architecture (HPCA 11)*, IEEE, 2011, pp. 466–477.
15. M.K. Qureshi, “Pay-As-You-Go: Low-Overhead Hard-Error Correction for Phase Change Memories,” *Proc. 44th Ann. IEEE/ACM Int’l Symp. Microarchitecture (MICRO 11)*, ACM, 2011, pp. 318–328.
16. A.N. Jacobvitz, A.R. Calderbank and D.J. Sorin, “Coset Coding to Improve the Lifetime of Memory,” *Proc. 19th Int’l Symp. High Performance Computer Architecture (HPCA 13)*, IEEE, 2013, pp. 222–233.
17. R. Azevedo et al., “Zombie Memory: Extending Memory Lifetime by Reviving Dead Blocks,” *Proc. 40th Ann. Int’l Symp. Computer Architecture (ISCA 13)*, ACM, 2013, pp. 464–474.
18. E. Ipek et al., “Dynamically Replicated Memory: Building Resilient Systems from Unreliable Nanoscale Memories,” *Proc. 15th Int’l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS 10)*, ACM, 2010, pp. 3–14.
19. S. Cho and H. Lee, “Flip-N-Write: A Simple Deterministic Technique to Improve PRAM Write Performance, Energy and Endurance,” *Proc. 42nd Ann. IEEE/ACM Int’l Symp. Microarchitecture (MICRO 09)*, ACM, 2009, pp. 347–357.
20. N.H. Seong et al., “SAFER: Stuck-at-Fault Error Recovery for Memories,” *Proc. 2010 43rd Ann. IEEE/ACM Int’l Symp. Microarchitecture (MICRO 10)*, ACM, 2010, pp. 115–124.
21. W. Zhang and T. Li, “Characterizing and Mitigating the Impact of Process Variations on Phase Change based Memory Systems,” *Proc. 42nd Ann. IEEE/ACM Int’l Symp. Microarchitecture (MICRO 09)*, ACM, 2009, pp. 2–13.
22. M.K. Qureshi et al., “Enhancing Lifetime and Security of PCM-Based Main Memory with Start-Gap Wear Leveling,” *Proc. 42nd Ann. IEEE/ACM Int’l Symp. Microarchitecture (MICRO 09)*, ACM, 2009, pp. 14–23.
23. T. Gao et al., “Using Managed Runtime Systems to Tolerate Holes in Wearable Memories,” *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI 13)*, ACM, 2013, pp. 297–308.
24. A. Sampson et al., “Approximate Storage in Solid-State Memories,” *Proc. 46th Ann. IEEE/ACM Int’l Symp. Microarchitecture (MICRO 13)*, ACM, 2013, pp. 25–36.

**Karin Strauss** is a researcher at Microsoft Research and an affiliate faculty member at the University of Washington. Her research interests include computer architecture, mobile and cloud systems, and living systems in-cell computation. Strauss received a PhD in computer science from the University of Illinois at Urbana-Champaign. She is an IEEE and ACM Senior Member. Contact her at [ks Strauss@microsoft.com](mailto:ks Strauss@microsoft.com).

**Doug Burger** is a director in Microsoft Research’s Extreme Computing Group. His research interests include computer architecture, large-scale machine learning, new computing paradigms, and advanced natural user interfaces. Burger is an IEEE and ACM Fellow. Contact him at [dburger@microsoft.com](mailto:dburger@microsoft.com).



Selected CS articles and columns are available for free at <http://ComputingNow.computer.org>.

## Showcase Your Multimedia Content on Computing Now!

IEEE Computer Graphics and Applications seeks computer graphics-related multimedia content (videos, animations, simulations, podcasts, and so on) to feature on its Computing Now page, [www.computer.org/portal/web/computingnow/cga](http://www.computer.org/portal/web/computingnow/cga).

If you’re interested, contact us at [cga@computer.org](mailto:cga@computer.org). All content will be reviewed for relevance and quality.

**IEEE Computer Graphics and Applications**

