Theme Article

Nonblocking DRAM Refresh

Kate Nguyen
Virginia Tech

Kehan Lyu
Duke University

Xianze Meng
University of California at San Diego

Vilas Sridharan
AMD, Inc.

Xun Jian
Virginia Tech

***Abstract*—Since its invention half a century ago, dynamic random access memory (DRAM) has required dynamic refresh operations that block read accesses to refreshing data; this fundamental behavior gave DRAM its name. In contrast, DRAM's close relative—static random access memory (SRAM)—can statically re-enforce charge in the background without blocking read accesses at the cost of more expensive circuit structure. Nonblocking DRAM Refresh blurs this fundamental distinction between DRAM and SRAM at the system level to enable the best of both worlds—allowing read accesses to refreshing data in DRAM while preserving DRAM's low-cost circuit structure.**

Dynamic random access memory (DRAM) is arguably the most dominant type of memory technology today. Due to emerging trends such as 3-D die-stacked DRAM in processor packages (e.g., HBMs) and embedded DRAM on processor dies (e.g., to implement large LLCs), DRAM plays critical role in computing. However, since its inception, a half a century ago, DRAM has had an inherent physical characteristic that increases latency compared to its close relative—static random access memory (SRAM). While DRAM and SRAM are both volatile, DRAM requires dynamic/active refresh operations that stall accesses to refreshing data; in comparison, SRAM statically re-enforces electric charge in the background without stalling accesses but requires a more expensive circuit structure (e.g., 6T1C versus 1T1C for DRAM).

Refresh-induced stalls in DRAM reduce system performance. This has been a universal feature for systems using any DRAM memory. The performance overhead also grows as DRAM density increases; denser DRAM devices have more cells to refresh and, therefore, require either longer or more frequent refresh operations. Figure 1 shows DRAM refresh latency versus density for DDRx DRAM.

Digital Object Identifier 10.1109/MM.2019.2907486
Date of publication 26 March 2019; date of current version 8 May 2019.

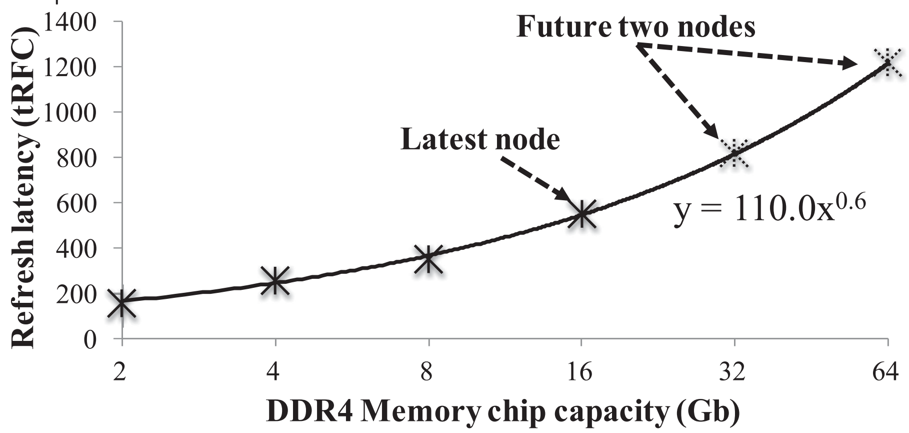

Figure 1. Historical (solid markers) and projected (dashed markers) refresh latency in nanoseconds for DDR4 DRAM chips.[2]

Nonblocking DRAM refresh eliminates this fundamental difference between DRAM and SRAM at the system level by refreshing the DRAM in the background without stalling read requests to refreshing memory blocks. In a conventional DRAM refresh scheme, a single refresh operation refreshes all of the data in a memory block simultaneously. Under Nonblocking DRAM Refresh, on the other hand, a single refresh operation refreshes only a portion of the data in a memory block. A system implementing nonblocking refresh equips each memory block with enough redundant data (e.g., parity) to reconstruct the blocks refreshing data when it is accessed, allowing the memory controller to issue reads to refreshing blocks. The memory controller issues refresh operations more frequently to keep the overall refresh time in memory constant.

The primary cost of implementing nonblocking refresh is equipping each memory block with redundant data to compute the refreshing/unreadable data in refreshing memory blocks. However, many existing memory systems already contain substantial redundant data to protect against hardware failures. We observe that these redundant data are budgeted to protect against worst-case hardware failure scenarios. Therefore, the redundant data are underutilized in the common case when hardware faults are absent. A memory block that has underutilized redundant data has *memory error resilience slack*. Memory architectures containing redundant data for hardware failure protection can leverage their memory error resilience slack to implement nonblocking refresh with little additional overhead.

Nonblocking refresh can be applied across all DRAM-based memories (e.g., off-package DRAM, in-package DRAM, and embedded DRAM); however, the microarchitectural details (e.g., how to refresh only a portion of data in a memory block at a time) vary depending on the particular DRAM technology. This paper explores the microarchitectural details for making DRAM refresh nonblocking for server memory systems built from DDRx DRAM. We end the paper by also briefly describing how to enable Nonblocking DRAM Refresh for other classes of DRAM memories.

IMPACT OF DRAM REFRESH ON SYSTEM PERFORMANCE

Each DRAM cell stores one bit of data as electric charge. However, the charge leaks over time; a DRAM cell loses its data if it loses its charge. To refresh many DRAM cells in memory sufficiently quickly, many DRAM cells are refreshed together; this causes large regions in memory to be unreadable during each refresh operation. In DDRx DRAM, the entire DRAM chip is unreadable when the chip performs refresh. As the DRAM density increases with newer generations of DRAM, refresh latency also grows because each refresh operation has to refresh more cells. In contrast, other memory latencies have remained steady or decreased across generations. Figure 2 shows the historical DRAM latency scaling; it shows that other DRAM latencies, such as bus cycle time and minimum read latency, have improved while refresh latency has increased. If these trends continue, refresh latency may become a determining factor in overall memory system performance.

The inability to read data from refreshing chips stalls program execution. To mitigate the resultant performance overhead, many recent works have proposed skipping half or more of the refresh operations to improve performance.[3–6] For example, RAIDR[4] profiles the retention time of DRAM cells and skips refresh operations to memory cells with

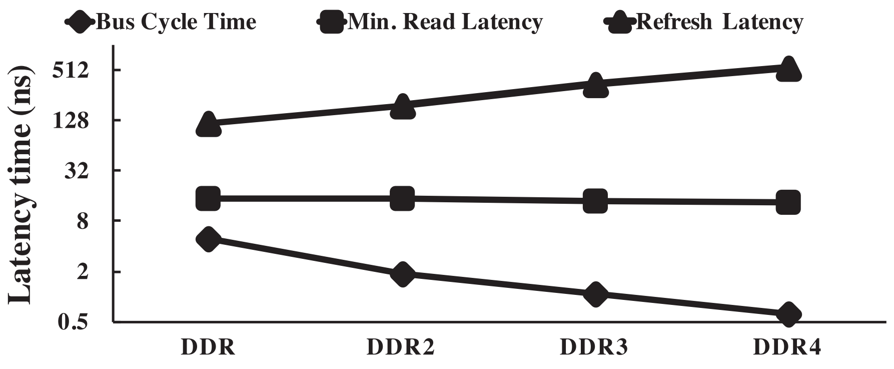

Figure 2. Historical scaling of memory latencies.[1,2]

long retention time. However, skipping refresh reduces the average amount of charge stored in DRAM cells and, therefore, can increase DRAM vulnerability to read disturb errors.[7]

MAKING DRAM REFRESH NONBLOCKING FOR SERVER MEMORY SYSTEMS

Background on Server Memory Systems

Modern server memory architectures operate groups of DRAM chips in lockstep. A group of DRAM chips that operate in lockstep is called a *rank*. For example, when the processor issues a read request for a 64B memory block to a rank, every chip in the rank replies with a part (e.g., 4B or 8B) of the memory block in lockstep. Similarly, when the processor issues a refresh command to a rank, all chips in the rank refresh in lockstep.

In server memory systems, each memory block may contain redundant data for hardware failure protection. These redundant data can correct up to complete memory chips failure(s) in a rank. The ratio of redundant data to program data in each block ranges from 12.5% to 40.6%.

Nonblocking DRAM Refresh Operations

Each nonblocking DRAM refresh operation refreshes a limited amount of data in a block at a time; this makes it possible for the server processor to use each memory block's redundant data to compute the block's small amount of data that are currently unreadable due to refresh (see detail later). In comparison, each conventional refresh operation refreshes entire blocks at the same time because it refreshes all chips in a rank in lockstep.

To limit the amount of data that is refreshing in a memory block at a time, the memory controller issues a refresh command to only a few DRAM chips in a rank at a time. Limiting the amount of data that is refreshing in a memory block at a time enables the memory controller to use the memory blocks redundant data to reconstruct the small amount of currently inaccessible data in the block. Because Nonblocking Refresh operations do not stall read requests, the memory controller can compensate for refreshing only some of the data in a block at a time by issuing refresh commands more frequently. In comparison, conventional refresh operations, which block read requests, can only refresh each rank infrequently to avoid excessively stalling read requests. A server system, for instance, can issue a refresh operation to the next subgroup of chips in a rank back-to-back in round robin fashion as soon as the previous subgroup finishes refresh

Reading From Refreshing Memory Blocks

Modern server systems equip each memory block with sufficient redundant data to correct a complete memory chip failure. The redundant data are underutilized in the common case when hardware faults are absent. Figure 3 quantifies the expected fraction of nonfaulty memory pages over time based on a large-scale field study,[8] assuming eight ranks per channel and 18 chips per rank. On average across seven years of operations, 97% of memory pages are not affected by any fault. While only a small fraction of memory

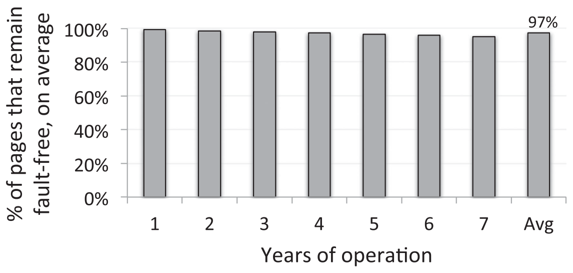

Figure 3. Expected fraction of memory pages that have not yet been affected by any hardware fault as a function of time.

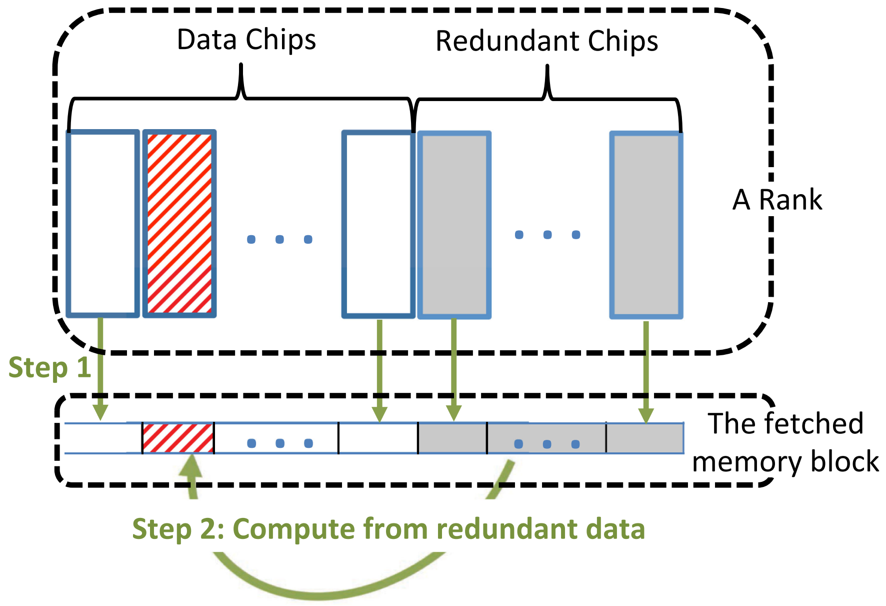

Figure 4. Reading a block from a rank performing Nonblocking DRAM Refresh. Red stripes represent data currently unreadable due to refresh.

experience faults, server systems protect all memory blocks with uniform redundant data because memory faults are stochastic events, whose time and location are difficult to predict.

A memory block that has underutilized redundant data has *memory error resilience slack*. Nonblocking DRAM Refresh uses memory error resilience slack to compute the unreadable data residing in refreshing chips without compromising reliability. When the processor fetches a block from a rank currently performing Nonblocking DRAM Refresh, the part of its data that reside in refreshing chips will be missing in the fetched block. The processor can use the fetched block's redundant data to compute the missing data. Figure 4 illustrates how to read from a rank that is currently performing Nonblocking Refresh. Computing the missing data is fast because the processor knows which memory chip(s) are refreshing and, therefore, which part of the block's data is missing. This is unlike regular error correction, where the processor needs to first determine where is the missing data before it can compute the correct value at that location; the majority of the latency incurred by error correction is due to locating the error.[9]

A system implementing Nonblocking Refresh must maintain the same level of error detection and correction as a system with conventional refresh. In a server memory system, each memory block contains some redundant data for error detection and some redundant data for error correction. To preserve error detection strength, a system with Nonblocking Refresh uses the same amount of redundant data to detect hardware errors for each read request as baseline systems. Nonblocking Refresh only uses the remaining redundant data in the memory block to compute missing data due to refresh. To preserve error correction strength, the memory controller computes data missing due to refresh only when the memory controller does not detect any hardware errors. When the memory controller does detect hardware error(s) for a read request, the processor waits for the rank to finish its in-flight refresh and then fetches the previously inaccessible data directly from DRAM. As all the data from the block are now available, the memory controller performs error correction exactly the same way as in a conventional system and, therefore, preserves the same error correction strength.

Write Requests During Nonblocking Refresh

Write requests to refreshing blocks still need to wait for refresh to complete; this is challenging for Nonblocking Refresh as it must refresh each memory block more frequently. Because write requests are not on the critical path of program execution, write latency has low impact on performance; as such, the processor may simply buffer write requests until the corresponding rank is not refreshing by adding a small (e.g., 32 KB) write-buffer cache to the memory controller.

Because all ranks in the same channel share the same memory bus, the processor can only write to one rank at a time in a channel. Therefore, total write bandwidth in a channel is divided across all the ranks in the channel as illustrated in Figure 5(a). Second, all ranks in a channel receive fairly even number of writes because logically adjacent memory pages are often interleaved across ranks to maximize rank-level parallelism.

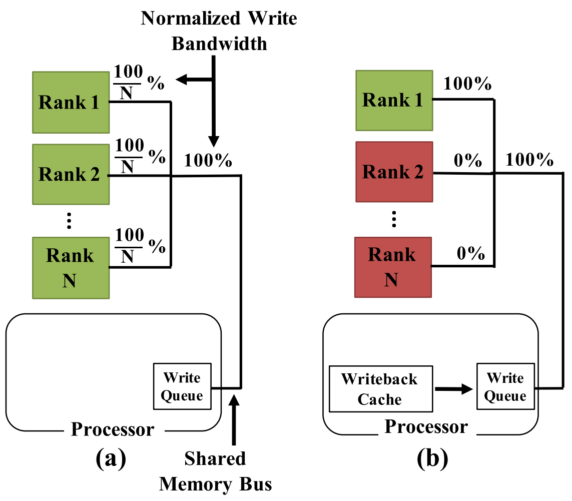

Figure 5. Write distribution in (a) conventional and (b) proposed memory systems. Green ranks are not refreshing and, therefore, writable; red ranks are refreshing and, therefore, not writable.

As such, the processor can reorder write requests to concentrate each channel's write bandwidth to one rank at a time as shown in Figure 5(b). This maintains the same channel-level write bandwidth while allowing the remaining ranks in the channel to perform Nonblocking Refresh.

MAKING DRAM REFRESH NONBLOCKING FOR OTHER TYPES OF MEMORY SYSTEMS

Nonblocking DRAM Refresh can apply to all DRAM-based memory systems. For example, desktop/laptop memory systems use the same rank architecture as server memory systems, except that they do not contain redundant data chips. As such, the same implementation of Nonblocking DRAM Refresh described for server memory systems is applicable to desktop/laptop memory systems after adding a redundant chip per rank to these memory systems.

Nonblocking DRAM Refresh is also applicable to memory systems that access only one DRAM chip per memory request, such as high bandwidth memory (HBM), graphics DDR (e.g., GDDRx), and smartphone memory (e.g., LPDDRx), because the internal organization within each DRAM die mirrors a memory channel's organization. For example, an HBM die contains multiple banks that share a common data bus,[10] just like DDRx contains multiple ranks in a channel sharing a common data bus. There are also multiple subarrays per bank just like there are multiple chips per rank.[10] In addition, each memory block is spread across multiple subarrays in one bank of a DRAM die, just like how a memory block is spread across a rank.[10] As such, HBM devices can implement Nonblocking DRAM Refresh by refreshing a portion of the subarrays in a bank at a time and adding redundant subarrays to each bank to compute the unreadable data in refreshing subarrays.

RESULTS

Using the Gem5 architectural simulator and Ramulator DRAM simulator, we simulated various NASA Parallel, Parsec, and SPEC2006 benchmarks. We simulated 16-core processor with four memory channels and four ranks per channel.

Figure 6 shows the average performance benefit when applying Nonblocking DRAM Refresh to Intel/AMD and IBM Power8 server memory systems. For 16–Gb DRAM chips, DRAM Refresh provides 13% and 17% average performance improvement for Intel/AMD and IBM server systems, respectively. Nonblocking DRAM Refresh may become even more beneficial for future DRAM chips with longer refresh latency as DRAM density increases; for our

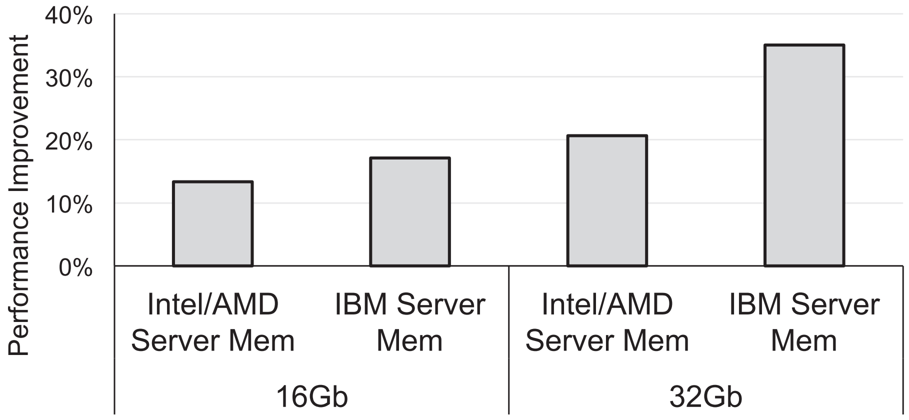

Figure 6. Average performance benefits of Nonblocking DRAM Refresh.

extrapolated 32Gb DRAM refresh latency, the average performance improvement increases to 21% and 35% for Intel/AMD and IBM server systems, respectively. The performance benefits for Intel/AMD server systems are lower because their amount of redundant data is lower; as such, to allow a refreshing memory block to be still calculable from its redundant data, each Nonblocking DRAM Refresh operation can only refresh one chip per rank at time, which provides low DRAM refresh throughput. As such, some conventional full-rank refresh operations that block read requests are still occasionally needed to achieve the same DRAM refresh throughput as conventional systems, resulting in reduced performance benefits.

CONCLUSION

Since its invention half a century ago, DRAM has required dynamic refresh operations that block read accesses to refreshing data; this fundamental behavior gave DRAM its name. Nonblocking DRAM Refresh improves upon this fundamental DRAM behavior by allowing read accesses to refreshing memory blocks. It operates at the microarchitecture level; it refreshes only a limited amount of data in a memory block at a time and uses redundant data in the block to compute the block's refreshing/unreadable data to complete read requests. The primary cost of Nonblocking DRAM Refresh is requiring some redundant data in each memory block.

In systems containing redundant data for hardware failure protection, Nonblocking DRAM Refresh can eliminate this primary cost by safely leveraging memory error resilience slack to calculate the unreadable data in refreshing memory blocks. In this paper, we demonstrate in detail such a memory-overhead-free Nonblocking DRAM Refresh design for server memory systems.

DRAM density scaling has been slowing down. An important reason is that increasing DRAM density often requires reducing DRAM cell size, which in term reduces DRAM retention time; reduced DRAM retention time requires more frequent DRAM refresh operations, which incur higher performance overhead. Also exacerbating the performance overhead is the accompanying fact that increasing DRAM density means more cells need to be refreshed. By effectively tackling DRAM refresh overheads, Nonblocking DRAM Refresh can help sustain density scaling for all DRAM-based memories, as the high-level idea of Nonblocking DRAM Refresh is applicable to all DRAM-based memories that are accessed at the memory block granularity. These include DDRx DRAMs used in servers and personal computers, GDDRx DRAMs used in GPUs, mobile DRAMs use in smartphones, in-package 3-D die-stacked DRAMs used in HPC systems, and embedded DRAMs used in LLCs within processors.

By effectively tackling DRAM refresh overheads, Nonblocking DRAM Refresh can help sustain density scaling for all DRAM-based memories, as the high-level idea of Nonblocking DRAM Refresh is applicable to all DRAM-based memories that are accessed at the memory block granularity.

REFERENCES

1. M. T. Inc., "Speed vs. latency: Why CAS latency isn't an accurate measure of memory performance," 2015. [Online]. Available: https://pics.crucial.com/wcsstore/CrucialSAS/pdf/en-us-c3-whitepaper-speed-vs-latency-letter.pdf
2. JEDEC, "JEDEC STANDARD. DDR4 SDRM. JESD74-4B," June, 2017. https://www.jedec.org/standards-documents/docs/jesd79-4a
3. A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman, "EnerJ: Approximate data types for safe and general low-power computation," *SIGPLAN Notices*, vol. 39, pp. 164–174, Jun. 2011.
4. J. Liu, B. Jaiyen, R. Veras, and O. Mutlu, "RAIDR: Retention-aware intelligent dram refresh," in *Proc. 39th Annu. Int. Symp. Comput. Architect.*, Jun. 2012, pp. 1–12.
5. I. Bhati, Z. Chishti, S. L. Lu, and B. Jacob, "Flexible auto-refresh: Enabling scalable and energy-efficient dram refresh reductions," in *Proc. ACM/IEEE 42nd Annu. Int. Symp. Comput. Archit.*, Jun. 2015, pp. 235–246.
6. M. Patel, J. S. Kim, and O. Mutlu, "The reach profiler (reaper): Enabling the mitigation of dram retention failures via profiling at aggressive conditions," in *Proc. 44th Annu. Int. Symp. Comput. Archit.*, 2017, pp. 255–268.

7. Y. Kim *et al.*, “Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors,” in *Proc. ACM/IEEE 41st Int. Symp. Comput. Archit.*, Jun. 2014, pp. 361–372.
8. V. Sridharan, J. Stearley, N. DeBardeleben, S. Blanchard, and S. Gurumurthi, “Feng Shui of supercomputer memory: Positional effects in DRAM and SRAM faults,” in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, pp. 22:1–22:11, 2013.
9. A. Kumar and S. Sawitzki, “High-throughput and low-power architectures for reed solomon decoder,” in *Proc. Conf. Rec. 39th Asilomar Conf. Signals, Syst. Comput.*, Oct. 2005, pp. 990–994.
10. B. Giridhar *et al.*, “Exploring DRAM organizations for energy-efficient and resilient exascale memories,” in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, Nov. 2013, pp. 1–12.

Kate Nguyen has an MS from the Computer Science Department, Virginia Tech. Her research interests include computer security. Contact her at katevy@vt.edu.

Kehan Lyu is currently a graduate student at Duke University. He has an undergraduate degree from the Computer Science Department, Virginia Tech. Contact him at kehan@vt.edu.

Xianze Meng is currently a graduate student at the University of California at San Diego. He has an undergraduate degree from the Computer Science Department, Virginia Tech. Contact him at xianze@vt.edu.

Vilas Sridharan is currently an AMD Fellow in the Reliability, Availability, and Serviceability Architecture Group at AMD, Inc., where he is the Lead RAS Architect for all of AMD’s products. His research focuses on the modeling of hardware faults and architectural and micro-architectural approaches to reliability and fault tolerance in high-performance microprocessors. He has a BSE in computer engineering from Princeton University and an MSE and a PhD from the Department of Electrical and Computer Engineering, Northeastern University. From 2000 to 2004, he worked in the SPARC Server Division, Sun Microsystems. He is a Senior Member of the IEEE. Contact him at vilas.sridharan@amd.com.

Xun Jian is an assistant professor in the Computer Science Department, Virginia Tech. His research interests include high-performance, energy-efficient, reliable, and secure processor and memory architectures. He has a PhD from the University of Illinois, Champaign. Contact him at xunj@vt.edu.