Survive: Pointer-Based In-DRAM Incremental Checkpointing for Low-Cost Data Persistence and Rollback-Recovery

Amirhossein Mirhosseini, Aditya Agrawal, and Josep Torrellas

Abstract—This paper introduces the *Survive* DRAM architecture for effective in-memory micro-checkpointing. Survive implements low-cost incremental checkpointing, enabling fast rollback that can be used in various architectural techniques such as speculation, approximation, or low voltage operation. Survive also provides crash consistency when used as the frontend of a hybrid DRAM-NVM memory system. This is accomplished by carefully copying the incremental checkpoints generated in the DRAM frontend to the NVM backend. Simulations show that Survive only imposes an average 3.5 percent execution time overhead over an unmodified DRAM main-memory system with no checkpointing, while reducing the number of NVM writes by 89 percent over an NVM-only main-memory system.

Index Terms—Non-volatile memory, checkpointing, reliability

1 Introduction

An increasing number of performance-enhancing or energy-saving architectural techniques such as speculation [1], approximation [2], and low-voltage operation [3] can benefit from low-overhead in-memory incremental checkpointing [4], [5], [6]. With in-memory incremental checkpointing, changes to the state of the system are periodically saved in memory, and are quickly undone when execution needs to be rolled back. However, in-memory checkpointing is insufficient when the state of the system is partially lost, such as in a power failure or in a variety of system crashes—e.g., in the frequent erratic reboots experienced by intermittently-powered energy-harvesting systems [7], [8]. For these types of failures, emerging non-volatile memory (NVM) technologies, such as Intel/Micron's 3D XPoint [11], provide efficient storage support.

Apart from their persistence characteristics, NVM technologies are known to be more scalable and more energy efficient than DRAM [12], which is the standard memory technology. Therefore, they are promising alternatives to DRAM for future memory systems. However, most of the NVM technologies that are dense enough to be used for main memory suffer from expensive writes. The excessive cost of these writes usually comes from their high write latency and write energy, or from the limited write endurance of the NVM memory. For example, PCM is known to have 8X-10X higher write latency and 2X-20X higher write energy than DRAM. It also has a limited value of write endurance (10^8), which is problematic in systems that require long lifetimes [9], [13]. As a result, many recent research studies have attempted to reduce the number of NVM writes by using different techniques, such as small write granularities [9], redundant-write avoidance [14], NVM-aware write scheduling [15] and page migration [16].

An effective system-level method to minimize the number of NVM writes is to use a hybrid two-level memory system, composed of a DRAM frontend and an NVM backend [17]. Most of the writes to such a memory system are filtered out and coalesced by the DRAM frontend and, as a result, the number of writes to the NVM backend is significantly reduced (similar to the behavior of caches). Nevertheless, designing such a system requires special care in order to ensure crash consistency [18]—i.e., that the data stored in NVM can be recovered to a consistent state after a power failure. In a more formal definition, the hybrid memory system should be able to guarantee the strict persistency model [19] between the DRAM frontend and the NVM backend. This means that the relative order of the global states that become durable in the NVM should be consistent with the order of the states that become visible to the system. All reorderings and coalescings performed by the hybrid memory system have to maintain this property.

To provide effective in-memory checkpointing, this paper proposes a DRAM architecture called *Survive*. Survive provides two functionalities. First, it provides low-cost incremental checkpointing, enabling fast rollback for a large class of architectural techniques such as speculation, approximation, or low-voltage operation. Second, it provides crash consistency when used as the frontend of a hybrid DRAM-NVM memory architecture. This is accomplished by carefully and gradually copying the incremental checkpoints generated in the DRAM frontend to the NVM backend—guaranteeing that the state of the NVM is always the same as the state of the system at one of the checkpoints. Our simulations show that Survive only imposes an average 3.5 percent execution time overhead over an unmodified DRAM memory system with no checkpointing, while reducing the number of NVM writes by 89 percent over an NVM-only main-memory system.

2 Background on DRAM Operations

As depicted in Fig. 1, DRAM cells are composed of a storage capacitor and an access transistor. They are arranged in sub-arrays, which group together to form memory banks. Each sub-array consists of multiple rows of DRAM cells connected to an array of sense amplifiers. Each row of DRAM cells shares the Wordline. Each column of DRAM cells shares the Bitline that connects those cells to the corresponding sense amplifier. The steps involved in a DRAM access are as follows: in the initial precharged state, the Bitlines are charged to 1/2VDD and the sense amplifiers are disabled. The access transistors of a row are triggered by the ACTIVATE command, and the voltage of the Bitlines are affected by the small charge stored in the capacitor cells of that row, changing to the value of 1/2VDD+d or 1/2VDD-d. Sense amplifiers will then be activated to amplify the deviations until the Bitlines reach VDD or 0. After the amplifications, the data is moved from the row to the sense amplifiers (row buffer), and the capacitors are restored to their original values. Finally, a specific column of the amplified row is multiplexed out of the row buffer using READ/WRITE commands, and the sub-array can be taken back to the precharged state by issuing a PRECHARGE command.

3 Survive Architecture

In this section, we describe the Survive architecture. First, we explore the in-DRAM checkpointing approach by which Survive generates and keeps the incremental checkpoints efficiently in memory (DRAM frontend). Then, we introduce a pointer-based scheme that Survive uses to manage the ordering by which the incremental checkpoints are copied to the NVM backend. Finally, we investigate different approaches that can be used in order to guarantee crash consistency if a power failure occurs in the middle of copying the data into the NVM.

- *A. Mirhosseini is with the University of Michigan, Ann Arbor. E-mail: miramir@umich.edu.*
- *A. Agrawal is with NVIDIA Corp, Santa Clara, CA 95050. E-mail: adityaa@nvidia.com.*
- *J. Torrellas is with the University of Illinois at Urbana-Champaign, Champaign, IL 61801. E-mail: torrella@illinois.edu.*

Manuscript received 29 Aug. 2016; revised 26 Oct. 2016; accepted 20 Nov. 2016. Date of publication 28 Dec. 2016; date of current version 5 Jan. 2018.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/LCA.2016.2646340

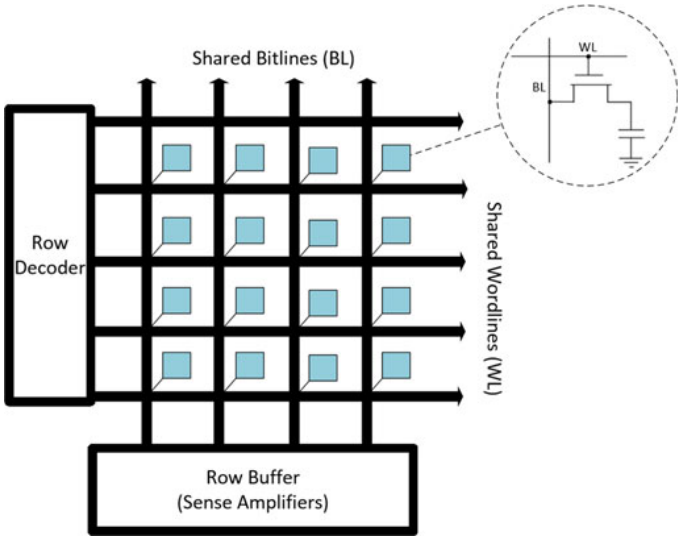

Fig. 1. Organization of a DRAM sub-array.

3.1 In-DRAM Incremental Checkpointing

Survive generates and stores incremental checkpoints in the memory. Therefore, we define a period, called checkpointing epoch, which is the checkpointing granularity of our system. The state of the system can be rolled back only to the starting point of a checkpointing epoch, and the state of the NVM backend will also be the view of the system at the starting point of a checkpointing epoch. We set our epoch length to be 10 ms, which is consistent with previous designs [4], [5], [18].

In order to make efficient in-memory copies, we use the RowClone-FPM (Fast Parallel Mode) technique introduced in [21]. RowClone-FPM performs a rapid and efficient copy operation between two rows within the same DRAM sub-array by issuing back to back ACTIVATE commands to the source and destination rows without any intervening PRECHARGE command. To enable RowClone-FPM, we set our copying granularity to be as large as a DRAM row (i.e., 8 KB in our design).

As depicted in Fig. 2, we partition each DRAM sub-array into two regions: the main region and the checkpointing region. The main region is used for the regular allocation of the physical memory pages of the system. The checkpointing region is used for storing the incremental checkpoints. The rows in the checkpointing partition are reserved for checkpointing copies and are not visible beyond the memory controller.

The memory controller keeps a list of the rows that have been checkpointed during each epoch. When a row is written to for the first time during an epoch, the memory controller adds its number to the end of the list and performs a Copy-on-Write (CoW) operation (i.e., copies its old value) to the next available row in the checkpointing region. CoW operations can be efficiently performed using the RowClone-FPM method. For example, if the order in which the rows of a particular sub-array have been written to in

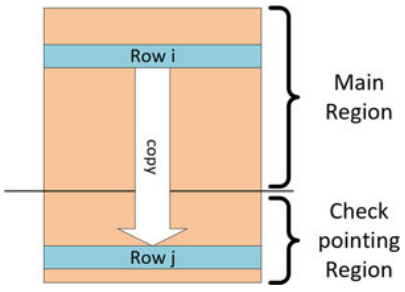

Fig. 2. Main region and checkpointing region in a DRAM sub-array.

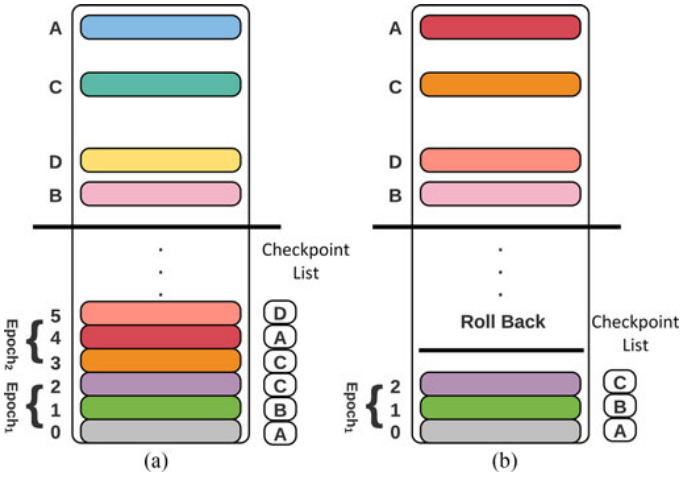

Fig. 3. Placement of data a) before and b) after rollback in the sub-array.

two consecutive epochs is AABCBAB and CADACD (namely, steps 1-7 and 8-13), a CoW operation is performed only the first time each of the rows has been written to during each of the two epochs (i.e., steps 1, 3, 4 for the first epoch, and steps 8, 9, 10 for the second one). As a result, rows that have not been written during an epoch (i.e., row D in the first epoch and row B in the second one) will not be copied. The placement of data in this sub-array after the example steps is shown in Fig. 3a.

If a rollback command is generated by the processor, the memory controller uses its checkpointing list to copy back the old values to their original locations in the main region, and then deletes the row numbers from the end of the list one by one. Since the values of the modified rows are only available for the starting point of an epoch, the system can only be rolled back to one of these points. Fig. 3b shows the placement of the data in the previous example after a rollback to the start of the second epoch.

3.2 Pointer-Based Scheme for Updating the NVM Backend

To be able to update the NVM backend with incremental checkpoints, we need to have a list of the modified rows during an epoch and their values at the end of that epoch (as opposed to their values at the beginning of the epoch). To attain this goal, we could perform a CoW at every single write operation to a row (rather than only at the first write in each epoch). However, this would have huge performance and storage overheads. Another solution would be to keep a list of the modified rows during an epoch, and perform all of the copy operations to NVM at the end of the epoch, by stalling the program execution. However, this method would have non-negligible performance overheads. Moreover, if the incremental checkpoints are generated at the end of the epochs (instead of the starting points), they cannot be used for rollback anymore.

In Survive, we solve this problem by proposing a pointer-based scheme that keeps track of the last version of the rows at the end of each epoch. To this end, we augment the scheme explained Section 3.1 by extending each DRAM row with some meta-data bits as in [22]. These meta-data bits are only visible and accessible to the memory controller, and can only be accessed with special commands. They are cheaply accessed with the data in the same row. We use these meta-data bits to store pointers in some rows in order to keep track of the different versions of the data to update the NVM. As illustrated in Fig. 4, the pointer in each row of the main region points to the *last checkpoint* of that row. On the other hand, the pointer in each row of the checkpointing region points to the *next version* of that checkpointed row.

Every time that a row is copied to the checkpointing region, three pointers must be updated: the pointer at the original row of the main region that is being modified (since its last checkpoint is

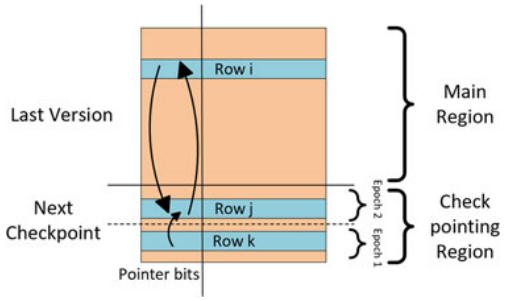

Fig. 4. Pointer-based scheme.

changing), the pointer at the last (previous) checkpoint of the row (since its next version is changing), and the pointer at the new checkpoint row that is currently being created by the CoW operation. However, two of these three pointers (the ones in the original row and in the new checkpoint) are updated in parallel with the updates of the row data and, hence, have very small overheads. Consequently, each CoW operation now is, in terms of latency, almost equivalent to the CoW of Section 3.1 followed by another write (to update a pointer in another row). Nonetheless, the aggregate cost is still very low due to the high row locality of the writes (CoWs are performed for each modified row only once in an epoch). Fig. 5a shows Fig. 3a augmented with the row pointers. Now, suppose that a new value is written to row B before the end of the second epoch. In that case, a CoW has to be performed from this row to row 6. The result is shown in Fig. 5b. The value of the pointer in row B is changed to be 6 (its last checkpoint), and the value of the pointer in row 1 is also changed to be 6 (the next version of row B). Finally, row 6 must point to row B (the last version of itself). If B is updated in the next checkpointing epoch, the pointer chain is enlarged in a similar manner.

These pointers are used to update the NVM when the DRAM frontend is not busy with read and write commands issued by the processor. As explained above, the memory controller needs a list of the modified rows during each epoch and their values at the end of that epoch to be able to update the NVM correctly. Using the pointer-based design described, this can be easily done by: (1) getting the ID of the modified rows from the checkpointing list and (2) getting their values using the pointers to get to their next version. Specifically, the controller traverses the checkpointing list in order, while getting the values of the rows one by one by going to where the corresponding pointers point to. This provides the values of the modified rows at the end of each epoch. In the example of Fig. 5a, when the memory controller attempts to move the rows modified in the first epoch to the NVM, it starts from the first entry of the checkpointing list, which is A. It gets the value using the pointer to row 4, which is the value of row A at the end of the first checkpoint. The controller then moves to the second entry, which is B. It gets the value using the pointer, which points directly to row B, since this row has not been updated during the second epoch or after that. Once all the rows checkpointed in an epoch have been moved to the NVM, the entries can be removed from the checkpointing list. Clearly, moving the rows checkpointed in an epoch to the NVM can only start when that epoch has finished.

3.3 Ensuring Epoch Atomicity

As explained before, the NVM should always contain the view of the DRAM at the starting point of a checkpointing epoch. On that account, none of the rows copied to the NVM should become durable (i.e., visible after the power failure) unless all of the rows modified during an epoch have been copied to the NVM. This property, which we call *Epoch Atomicity*, is crucial for guaranteeing crash consistency. In general, any of the checkpoint separation mechanisms (buffering, logging and renaming [4]) that have been widely

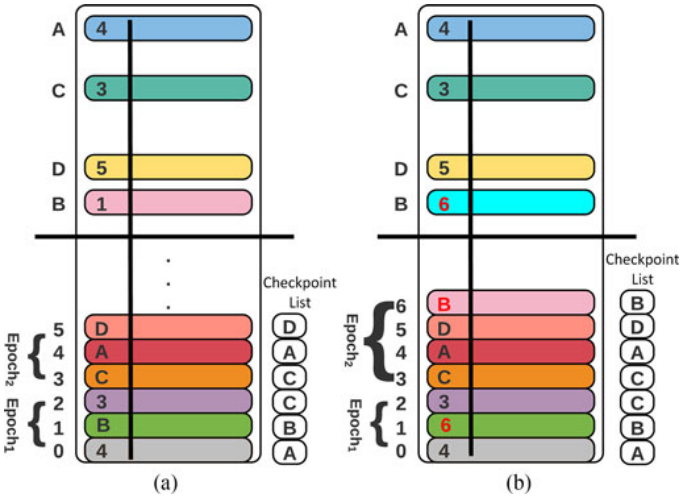

Fig. 5. Pointers a) before and b) after a write to row B.

used in various rollback-recovery systems can be deployed to guarantee this property.

If we use buffering, all of the rows modified during an epoch will be accumulated in a buffer, and then moved to the NVM when all of them have been successfully copied to the buffer. The buffer should itself be non-volatile, in order to ensure crash consistency if a power failure happens while moving the modified rows from the buffer to the NVM backend. If, instead, we use renaming, a mapping table should be available in the NVM that specifies the current version of each row through an extra level of indirection. The entries of this table are changed when all of the rows modified during an epoch have been successfully moved to the NVM. Finally, logging implies copying the old value of the rows in NVM to another NVM location, called a log, before copying the rows from DRAM to NVM—to be able to recover on a power failure in the middle of moving the rows from DRAM to NVM.

Although the focus of this paper is primarily DRAM architecture, and the details of the epoch atomicity mechanism are beyond its scope, the mechanism that we suggest and have used in our design is buffering. There are three reasons for this. First, buffering is simpler, as it does not involve mapping tables or logs. The buffer can be a small non-volatile storage that is able to contain as many rows as the number of modified rows during an epoch. Unlike the NVM backend that has to be made of dense technologies such as PCM, the buffer can be built with technologies such as STT-RAM [10] that have larger endurance values and lower write latencies. Second, logging and renaming require writing a row in either multiple places (for the same epoch) or different places (across epochs) in the NVM. Consequently, they both expect individual rows to be copied *in their entirety* (rather than partially), even if some parts of the row have not been modified. With buffering, an individual row is always copied to the same place across epochs in the NVM. Hence, if the entries of the checkpointing list are extended with additional indicator bits that specify which portions (e.g., which cache lines) of the row have been modified, only those portions are copied to the NVM. As a result, the number of writes to the NVM backend is minimized. Finally, our buffer can leverage the Asynchronous DRAM Refresh (ADR) technology of the future Intel-compatible platforms, which guarantees the persistency of the write queues in the main memory subsystem[20]. Therefore, no additional modification will be needed in order to implement the persistent buffer.

4 Evaluation

We have used the ESESC [23] full-system simulation environment and the SPEC CPU2006 benchmark suite to evaluate Survive. Our

TABLE 1
System Parameters

Processor	2.5 GHz, OoO, 3-wide issue
L1 I/D	32 KB, 8-way, 64 B block
L2 Cache	256 KB, 8-way, 64 B block
L3 Cache	2 MB, 16-way, 64 B block
Memory	DDR3-1066

system configuration and parameters are listed in Table 1. In order to guarantee crash consistency, at the end of each epoch, the whole CPU state (i.e., registers, flags, etc) and all of the dirty cache lines are flushed into the memory (without being invalidated). This causes a small execution stall at the end of each epoch. Also, we have used per-line indicator bits to determine which cache lines of a row have been modified. Therefore, only the lines of the row that have been modified during an epoch are copied into the NVM backend.

We have measured the execution slowdown compared to a checkpoint-free execution, and the number of NVM writes that can be reduced compared to an NVM-only main-memory system. Fig. 6 shows performance penalty with respect to a checkpoint-free execution. As we can see in this figure, the execution time with Survive is, on average, 3.5 percent longer than a regular execution with no checkpointing. As a result, Survive has a negligible performance penalty—while providing rollback capability and crash consistency support with an NVM backend. Note that the calculated performance overhead is composed of three components: creating the in-memory incremental checkpoints, reading the data from DRAM to be written into NVM, and the stalls at the end of each epoch for flushing the volatile state into the NVM. The write latency of the NVM does not have much impact because the NVM writes are performed in parallel with the program execution. As explained before, the number of NVM writes is significantly smaller than the number of DRAM writes and hence, provided that the NVM backend uses typical technologies such as PCM, NVM writes do not impose any visible performance overheads to the system. In fact, Fig. 7 shows that Survive is able to reduce the average number of writes to the NVM backend by 89 percent relative to an NVM-only main memory. This shows that the DRAM frontend is a very effective filter of writes with minimum overhead. Survive can also be implemented in collaboration with other techniques that attempt to reduce the number of NVM writes from the backend side. Such techniques could help applications such Sjeng where, as seen in Figs. 6 and 7, Survive has more overhead. Sjeng has overhead because it includes a lot of pointer chasing and, therefore, exhibits poor row locality in its write accesses.

In terms of storage overhead, the meta-data bits that are added to each row of the sub-array for storing the pointers impose about

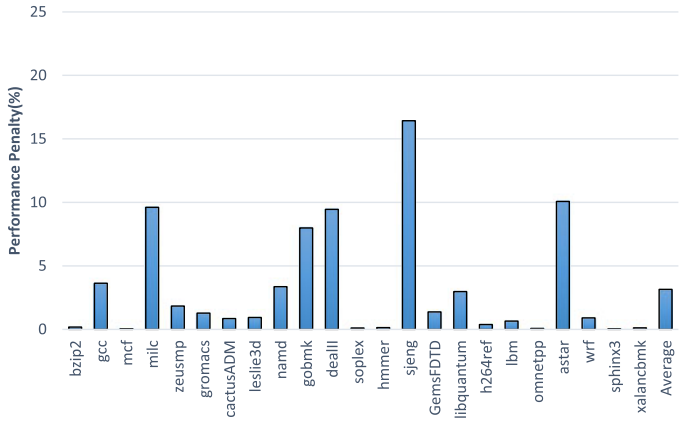

Fig. 6. Performance penalty for SPEC CPU2006.

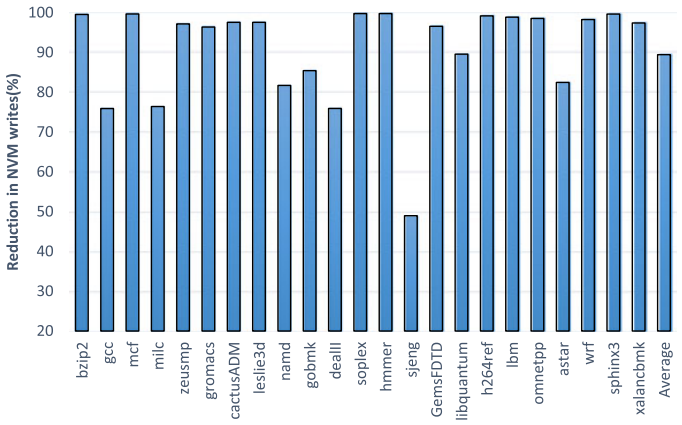

Fig. 7. Reduction in NVM writes for SPEC CPU2006.

0.26 percent overhead to the whole DRAM array, which is negligible. We do not consider the storage needed for storing the checkpointed rows as overhead. Storing checkpoints for four epochs in the checkpointing region requires on average 21 percent more storage. This number could be reduced by storing checkpoints for fewer epochs, or by shortening the epoch lengths. However, we would be reducing the length and flexibility of the rollbacks.

5 Conclusion

This paper proposed the *Survive* DRAM architecture for effective in-memory checkpointing. Survive provides low-cost incremental checkpointing, enabling fast rollback for a large class of architectural techniques such as speculation, approximation, and low-voltage operation. In addition, it provides crash consistency when used as the frontend of a hybrid DRAM-NVM memory architecture. This is accomplished by carefully and gradually copying the incremental checkpoints generated by the DRAM frontend to the NVM backend—guaranteeing that the state of the NVM is always the same as the state of the system at one of the checkpoints. Our simulations showed that Survive only imposes an average 3.5 percent execution time overhead over an unmodified DRAM memory system with no checkpointing, while reducing the number of NVM writes by 89 percent over an NVM-only main-memory system.

References

[1] J. F. Martinez and J. Torrellas, "Speculative synchronization: Applying thread-level speculation to explicitly parallel applications," in *Proc. 10th Int. Conf. Archit. Support Program. Languages Operating Syst.*, 2002, pp. 18–29.

[2] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Architecture support for disciplined approximate programming," in *Proc. 17th Int. Conf. Archit. Support Program. Languages Operating Syst.*, 2012, pp. 301–312.

[3] D. Ernst, et al., "Razor: A low-power pipeline based on circuit-level timing speculation," in *Proc. 36th Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2003, Art. no. 7.

[4] M. Prvulovic, Z. Zhang, and J. Torrellas, "ReVive: Cost-effective architectural support for rollback recovery in shared-memory multiprocessors," in *Proc. 29th Annu. Int. Symp. Comput. Archit.*, 2002, pp. 111–122.

[5] R. Agarwal, P. Garg, and J. Torrellas, "Rebound: Scalable checkpointing for coherent shared memory," in *Proc. 38th Annu. Int. Symp. Comput. Archit.*, 2011, pp. 153–164.

[6] S. Gao, B. He, and J. Xu, "Real-time in-memory checkpointing for future hybrid memory systems," in *Proc. 29th ACM Int. Conf. Supercomputing*, 2015, pp. 263–272.

[7] A. Colin, G. Harvey, B. Lucia, and A. P. Sample, "An energy-interference-free hardware-software debugger for intermittent energy-harvesting systems," in *Proc. 21st Int. Conf. Archit. Support Program. Languages Operating Syst.*, 2016, pp. 577–589.

[8] K. Ma, et al., "Architecture exploration for ambient energy harvesting nonvolatile processors," in *Proc. IEEE 21st Int. Symp. High Performance Comput. Archit.*, 2015, pp. 526–537.

[9] B. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting phase change memory as a scalable DRAM alternative," in *Proc. 36th Annu. Int. Symp. Comput. Archit.*, 2009, pp. 2–12.

[10] E. Kultursay, M. Kandemir, A. Sivasubramaniam, and O. Mutlu, "Evaluating STT-RAM as an energy-efficient main memory alternative," in *Proc. IEEE Int. Symp. Performance Anal. Syst. Softw.*, 2013, pp. 256–267.

[11] Intel corporation, Jul. 2015. [Online]. Available: https://newsroom.intel.com/news-releases/intel-and-micron-produce-breakthrough-memory-technology/
[12] O. Mutlu, "Memory scaling: A systems architecture perspective," in *Proc. Memory Workshop*, 2013.
[13] X. Dong, N. P. Jouppi, and Y. Xie, "PCRAMsim: System-level performance, energy, and area modeling for phase-change RAM," in *Proc. Int. Conf. Comput.-Aided Des.*, 2009, pp. 269–275.
[14] P. Zhou, B. Zhao, J. Yang, and Y. Zhang, "A durable and energy efficient main memory using phase change memory technology," in *Proc. 36th Annu. Int. Symp. Comput. Archit.*, 2009, pp. 14–23.
[15] J. Hu, Q. Zhuge, C. J. Xue, W.-C. Tseng, S. Gu, and E. H.-M. Sha, "Scheduling to optimize cache utilization for non-volatile main memories," *IEEE Trans. Comput.*, vol. 63, no. 8, pp. 2039–2051, Aug. 2014.
[16] N. Agarwal and F. W. Thomas, "Thermostat: Keeping your DRAM hot and NVRAM cool," in *Proc. Int. Conf. Archit. Support Program. Languages Operating Syst.*, 2017.
[17] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, "Scalable high performance main memory system using phase-change memory technology," in *Proc. 36th Annu. Int. Symp. Comput. Archit.*, 2009, pp. 24–33.
[18] J. Ren, J. Zhao, S. Khan, J. Choi, Y. Wu, and O. Mutlu, "ThyNVM: Enabling software-transparent crash consistency in persistent memory systems," in *Proc. 48th Int. Symp. Microarchitecture*, 2015, pp. 672–685.
[19] S. Pelley, P. M. Chen, and T. F. Wenisch, "Memory persistency," in *Proc. 41st Annu. Int. Symp. Comput. Archit.*, 2014, pp. 265–276.
[20] Intel corporation, Oct. 2016. [Online]. Available: https://software.intel.com/en-us/blogs/2016/09/12/deprecate-pcommit-instruction
[21] V. Seshadri, et al., "RowClone: Fast and energy-efficient in-DRAM bulk data copy and initialization," in *Proc. 46th Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2013, pp. 185–197.
[22] A. Agrawal, M. O'Connor, E. Bolotin, N. Chatterjee, J. Emer, and S. Keckler, "CLARA: Circular linked-list auto and self refresh architecture," in *Proc. 2nd Int. Symp. Memory Syst.*, 2016, pp. 338–349.
[23] E. K. Ardestani and J. Renau, "ESESC: A fast multicore simulator using time-based sampling," in *Proc. IEEE 19th Int. Symp. High Performance Comput. Archit.*, 2013, pp. 448–459.