Reducing NVM Writes with Optimized Shadow Paging*

Yuanjiang Ni[1] Jishen Zhao[2] Daniel Bittman[1] Ethan L. Miller[1,3]

[1]*Univ. of California, Santa Cruz* [2]*Univ. of California, San Diego* [3]*Pure Storage*

Abstract

Byte-addressable non-volatile memory (BNVM) technologies are closing the performance gap between traditional storage and memory. However, the integrity of persistent data structures after an unclean shutdown remains a major concern. Logging and shadow paging are commonly used to ensure consistency of BNVM systems. But both approaches can impose significant performance and energy overhead by writing extra data into BNVM. Our approach leverages the indirection of virtual memory to avoid the need for logging actual data and uses a novel cache line-level mapping mechanism to eliminate the need to write unnecessary data. Thus, our approach is able to significantly reduce the overhead of committing data to BNVM. Our preliminary evaluation results show that using OSP for transactions reduces the overhead necessary to persist data by up to 1.96× as compared to undo-log. Moreover, our approach can be used to provide fast, low-overhead persistence for hardware transactional memory, further facilitating the acceptance of BNVM into computing systems.

1 Introduction

Byte-addressable Non-Volatile Memory (BNVM) is becoming a reality, as technologies such as STT-RAM [10], PCM [16] (Phase-Change Memory) and memristor [18] show DRAM-like performance and disk-like persistence. Building applications directly upon BNVM can be simpler, since code that serializes and deserializes in-memory data structures to and from persistent storage is no longer needed [7] and more efficient, since overhead from legacy system software layers such as file systems and the block layer may be avoided [2, 3, 8, 9, 20]. However, applications need a mechanism to safely update their persistent data structures so they can recover from an inconsistent state after an unclean shutdown such as a crash.

Logging is commonly used to ensure storage consistency [3, 6, 9, 20]. However, when logging is used, old data (or new data) need to be copied to somewhere else (a logging area in BNVM) before the data can be updated in-place. However, using CPU instructions such as `clwb`, `movnti`, and `mfence` to persist the copied data greatly harms performance [24]. To combat this problem, we observed that shadow paging can **eliminate data copying by taking advantage of virtual memory indirection**. Shadow paging is different from logging in that it has no distinct log area and data area. It keeps the old data in-place, writes the new data to any other free physical page, and atomically updates the persistent virtual-to-physical mappings when a transaction completes.

One challenge of using shadow paging, however, is the amplified overhead caused by the gap between the large page size (typically 4–8 KB) and the number of bytes that are actually modified, which could be as small as few bytes. Simply reducing the page size would result in an unacceptable increase in virtual-to-physical mapping and page table walk overhead [17].

To address this issue, we propose Optimized Shadow Paging (OSP), which offers the same transactional semantics as shadow paging with **fine-grained persistence** at the level of a cache line. The key idea of OSP is to employ **compact** cache line level mappings within each valid page. OSP requires only **two bits per cache line** to construct these mappings, and only requires them for pages whose entries are currently in the TLB, since those are the only pages actively being accessed. As Figure 1 shows, when OSP is used to perform transactional updates, each virtual page is associated with *two* physical pages, P0 and P1, enabling the system to maintain two physical versions of each virtual cache line. This, in turn, allows the system to preserve the previously-consistent state of a virtual page as it presents the current state of

*This research was supported by the NSF under grant IIP-1266400 and by the industrial members of the NSF IUCRC Center for Research in Storage Systems.

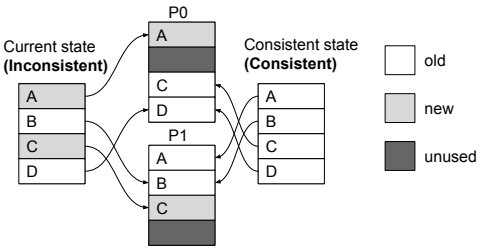


Figure 1: Two states of a virtual page.

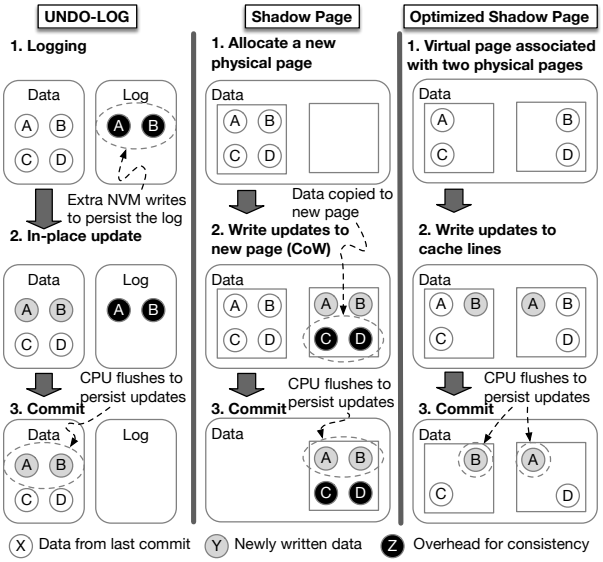


Figure 2: Comparison of three mechanisms to provide storage consistency.

this virtual page to memory accesses. To build practical durable transactions, we use lightweight journaling or super pages to provide atomicity for multi-page transactions, and we integrate page consolidation with TLB eviction for space efficiency. Our proposed design can be easily supported by hardware and only requires minimal OS changes. A hardware implementation improve performances by avoiding the extra instructions required by software [15].

Figure 2 compares two existing techniques, undo-logging and shadow paging, against OSP. It demonstrates that OSP transactions offer two advantages. First, OSP does not require data copying in the critical path, avoiding the performance overhead that results from it. Second, BNVM technologies often have limited write endurance [11], and OSP significantly reduces the number of writes to BNVM, thus mitigating this issue.

2 Design

Figure 3 shows the high-level components that are used to build durable OSP transactions: an extended TLB, a

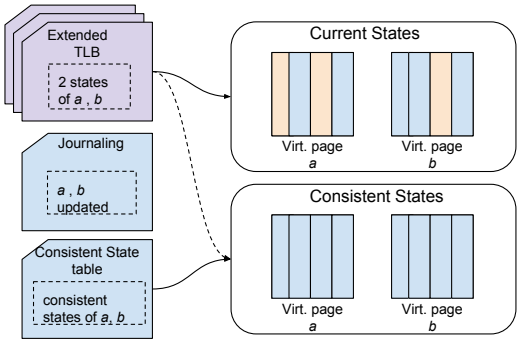


Figure 3: Overview of OSP transactions.

consistent state table, and a lightweight journal. Cache line level mappings of OSP are implemented as two sets of bitmaps, a *committed bitmap* and an *updated bitmap*. These bitmaps are combined in different ways to determine which cache lines make up the *current* state of a virtual page, and which cache lines make up the most recent *consistent* state of the page. Each TLB entry must be extended to contain both bitmaps for each page in the TLB, but the bitmaps need not be in the page table. Instead, the consistent states (*committed bitmap*) are also stored in a persistent table. These durable consistent states allow applications to recover after an unclean shutdown. Transaction commit advances the durable consistent state. When multiple virtual pages are updated in a transaction, their combined durable consistent state must be updated atomically: either all pages move to their next consistent states or none do. We propose two approaches to provide such atomicity: lightweight journalling, which only tracks whose consistent states (*e.g.*, virtual page numbers) need to be changed and how (*e.g.*, *updated bitmap*); and super page, which recursively combines multi-page transactions into a single-page transaction.

2.1 Cache line Level Mappings

The two per-page bitmaps, with one entry for each cache line in the page, allow OSP to track modifications to individual cache lines and to determine which cache line is "current" and which represents state from the last transaction. Each cache line's state is represented by a single *committed bit* and a single *updated bit*, which is analogous to the dirty bit in a regular cache. The committed bit for a cache line "points" to the page containing the cache line as it appeared at the most recent commit. The updated bit is set to one whenever the cache line is written, and is reset as part of the commit process. Since each bit refers to the cache line at the same offset in both P0 and P1, it is sufficient for the TLB entry to simply contain the physical page number for each of P0 and P1; the offset is

determined as usual from the address.

As a transaction is being processed, reads are directed to the page determined by XORing the committed bit and the updated bit. If the cache line is clean, reads go to the most recently committed version. If the cache line has been modified in the transaction, the updated bit is set and the read goes to the "other" cache line. Writes always go to the page determined by $committed\_bit \oplus 1$, and the updated bit is set as part of the write operation.

Because these operations are very simple and require few gate delays, adding support for OSP is unlikely to significantly affect the critical path for TLB translation. TLB translations and cache accesses are done in parallel in commonly-used virtually-indexed physically-tagged caches, which could further hide the increase, if any, in TLB translation latency. One potential issue, however, arises with virtually-addressed caches, since this approach does cache access *before* TLB access completes. Moreover, virtually-addressed caches have no support for including two versions of a single cache line, beyond the use of a process-specific identifier. While our approach works well with physically-addressed caches, we are currently exploring options to support virtually addressed caches.

Under OSP, TLB entries must be made wider to store information about the current states of virtual pages. A TLB entry now includes two PPNs, a *committed bitmap*, and a *updated bitmap*. This enables the memory management unit to quickly access the values as part of TLB access, and to determine *which* of the physical pages associated with a single virtual address should be used for the memory access. Although OSP might double the size of the TLB entries (assuming 64-bit bitmaps), we think such overhead is acceptable. Alternatively, we can use a separate hardware unit to store the OSP-related information, so normal TLB entries would be immune to such storage overhead.

2.2 Page Consolidation

It's a waste of precious memory space to associate these virtual pages that are not actively being updated with two physical pages. Thus, when a virtual page is evicted from the last TLB (not being actively accessed), we consolidate valid data to one physical page and free the other one. The consolidation overhead is minimized by merging the page that has fewer valid cache lines into the other one. Note that we currently couple page consolidation with TLB eviction for simplicity, but page consolidation could be done lazily. Finally, we update the virtual-to-physical mapping table so that the virtual page refers to the physical page with all the valid data. As an optimization, page consolidation could be moved out of the critical path in a TLB fault, since it need not be performed *immediately* on a TLB fault. We plan to investigate hardware/OS support for asynchronous page consolidation in the future.

2.3 OSP Semantics

Regular Memory Accesses. Because we envision future systems will have a hybrid of DRAM and BNVM, we distinguish two types of stores. Transactional stores (*e.g.*, stores to persistent memory) require a crash consistency guarantee while regular ones (*e.g.*, stores to volatile memory) don't. Regular memory accesses begin with address translation. As discussed in Section 2.1, the TLB function is changed in order to always present current states to a running thread. Given a virtual address (assume hit), depending on the corresponding bits (XOR them) for the accessed cache line in *committed bitmap* and *updated bitmap*, either P0 or P1 is returned to the processor. The remainder of the path for memory accesses is unaffected.

Transactional Stores. Transactional stores involve three steps. First, we must copy the cache line from its old location to its new location if this is the first time that the cache line has been updated in the transaction. This step can be completed in hardware by simply updating the cache tag to reflect the "current" page of the cache line after we read this cache line from its old location. Second, we update the TLB to indicate that the cache line gets updated. This can be done unconditionally, since it's easier to always set the updated bit in the TLB than it is to first determine whether it has already been set. After this is done, a transactional write proceeds as a regular write access, using the new value of the updated bit (set) to determine which page to write.

Commit. When the current state of a virtual page reaches a consistent state, the *commit* operation provided by OSP is used to bring the consistent state of the virtual page update-to-date by XORing the *updated bitmap* into the *committed bitmap* and clearing the *updated bitmap*.

Abort. If a transaction must be aborted, this can easily be done by simply clearing the *updated bitmap*, restoring the page's state to the most recent (successful) commit.

2.4 Making OSP Transactions Durable

OSP allows us to always keep a consistent state of a virtual page while serving transactional updates, but we must still ensure that transactions become durable. While our techniques work well with a single page, we must also allow multiple pages to be updated in a failure-atomic way.

Make Consistent States Recoverable. The OS maintains a *consistent state table* of tuples of the form $\langle V, P0, P1, CB \rangle$, which associates a virtual page V with

two physical pages (*P0* and *P1*) as well as a committed bitmap *CB*, in BNVM. The OS only updates the *consistent state table* on TLB misses (we assume software-managed TLB is supported). A new tuple is added if there is no tuple with the same *V* already in the set, and a tuple is removed when it gets evicted from the last processor TLB.

Make Transaction Commit Durable. Durable transactions guarantee data persistence after commit requests are acknowledged. The first step towards this goal is to find the cache lines that have been updated by the transaction and force them to be written back to NVM. We are able to track the updated cache lines via the updated bitmap in the extended TLB. Then, for new consistent states to be retrievable even after power cycling, we **atomically** update the corresponding *committed bitmaps* stored in the *consistent state table*. Finally, we apply the *commit* operation to all the updated virtual pages to bring their consistent state up-to-date.

Lightweight Journaling for Multi-page Transactions. For transactions that only update a single virtual page, we are able to update a *committed bitmap* stored in the *consistent state table* atomically (8 bytes atomic in-place update). However, some transactions may update multiple virtual pages, so we must use journaling to atomically update multiple committed bitmaps in the *consistent state table*. OSP journaling is **lightweight** since we only need to record the *updated bitmap* as well as a small amount of other information (*e.g.*, virtual page number which can be used to uniquely identify a tuple in the *consistent state table*).

Super Pages for Multi-page Transactions. As an alternative to journaling, we can transform multi-page transactions into single-page transactions using super pages. We reserve a super page for each address space to store the consistent state table. Super pages are themselves updated with OSP semantics. When a transaction is committed, we persist the new updates, update the consistent state table stored in the super page, and perform a single-page transaction commit for the super page. Under this mechanism, the OS need only maintain a persistent super page table which keeps consistent states of super pages. For the rest of this paper, we assume lightweight journaling is used for multi-page atomicity.

2.5 Performance Implications

OSP can provide durable transactions with almost no cost for workloads with decent locality. The performance overhead of OSP mainly comes from journaling and page consolidation. Our journaling only requires one small record per updated virtual page. In comparison, logging might require one record with actual data per update. Page consolidation overhead is not a per-transaction overhead. We only need to pay for it for space efficiency when a virtual page is not being actively updated. Journaling overhead depends on the spatial locality of the workloads while page consolidation overhead depends on the temporal locality of the workloads. OSP can benefit from workloads with either of these localities. Moreover, unlike redo-log, OSP has almost no overhead of address remapping, since it doesn't need to keep the updates in a separate log area.

3 Evaluation

We evaluated OSP by comparing it, under simulation, with an undo-logging approach. Our experiments show that OSP is much more efficient than undo-logging, and allow us to measure the components of overhead in OSP.

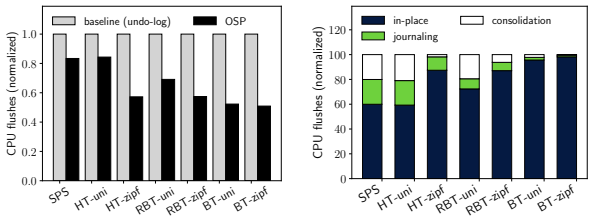
3.1 Experimental Setup

Simulator. We are implementing our design in `McSimA+` [1], a Pin-based cycle-accurate simulator. We configure the simulator to model a system with out-of-order processors, 64-entry L1 DTLB, etc. Our prototype has not been fully implemented yet, so we can not provide overall performance measurement at this point. However, we are able to collect some performance-critical information such as CPU flushes. Note that we use CPU flushes to represent cache flushes (*e.g.*, data) as well as write-combining buffer flushes (*e.g.*, uncacheable log). We do not use the bytes written; although the CPU might issue word writes, the caching system typically writes data back to memory at cache line granularity.

Workloads. We ran our system under seven transactional workloads. These workloads cover data structures that are commonly used in storage systems and cover different access patterns. *SPS* workloads randomly swap elements in a large array. *HT-uni*, *RBT-uni*, and *BT-uni* insert/delete nodes in a hash table, red-black tree, and B-tree (respectively) in a uniformly random fashion. *HT-zipf*, *RBT-zipf*, and *BT-zipf* insert/delete nodes in a hash table, red-black tree, and B-tree (respectively) following a Zipf distribution [5]. The elements, keys or values used in these workloads are all 8-byte integers. The footprints of these workloads range from 1 GB to 6 GB.

3.2 Results

CPU flushes. OSP can improve performance via saving extra expensive persisting operations. As shown in Figure 4a, compared to baseline (a hardware undo-log), OSP is able to reduce the number of CPU flushes by 1.6× on average. Workloads *SPS* and *HT-uni* have no spatial and temporal locality, and thus represent the least ideal work-



(a) CPU flushes (b) Breakdown

Figure 4: Preliminary results. OSP significantly reduces flushes relative to undo logging.

miss ratio	uniform (%)	Zipf (%)
SPS	3.6	3.6
HT	5.66	0.09
RBT	12.93	0.75
BT	1.44	0.15

Table 1: TLB miss ratio

loads for using OSP. However, OSP is still able to save 17% of CPU flushes.

Analysis. We break down CPU flushes into three sources, as shown in Figure 4b, for OSP transactions. We make two observations. First, better temporal locality leads to lower page consolidation overhead. As shown in Table 1, workloads *HT-zipf* and *RBT-zipf* have noticeably lower TLB miss ratios, which means less page consolidations caused by evictions, than workloads *HT-uni* and *RBT-uni*. As a result, we observe fewer CPU flushes caused by page consolidations in workloads *HT-zipf* and *RBT-zif* than in their counterparts. Second, better spatial locality leads to lower journaling overhead. Figure 4b shows that journaling only contributes a minor portion of the total CPU flushes in workload *BT-uni* because BTree is optimized for the better spatial locality. Better locality allows updates within a transaction to touch as few virtual pages as possible.

To sum up, OSP incurs comparable CPU flushes (1.2× fewer in our experiments) than logging when workloads has poor locality. However, OSP eliminates nearly *all* of the storage consistency cost for workloads with good locality. For example, in workload *BT-zipf*, 98% of CPU flushes come from persisting new updates (in-place updates) that are necessary for data persistence.

4 Related Work

BNVM-aware Data Structures and Systems. BNVM-aware data structures [19, 23] only focus on reducing storage consistency costs for particular data structures. In contrast, OSP transactions support atomic and durable updates of *any* data structure. Lightweight Persistent Memory [3, 8, 9, 20] and BNVM-aware file systems [4,6,21,22] provide programming support for Non-Volatile Memory, but they use either undo/undo logging or shadow page to build durable transactions. OSP can help make these approaches more efficient.

Reducing Storage Consistency Cost. Like OSP, both Kamino-TX [14] and LSNVMM [8] attempt to reduce the persistence overhead by eliminating the need to log actual data. Kamino-TX maintains another backup, and LSNVMM uses log-structured updates. However, both of them have inefficiencies. LSNVMM introduces significant address remapping overhead by adding another indirection in userspace, and Kamino-TX still needs to apply updates to backup asynchronously. DudeTM [13] enjoys the benefits of a redo-log (fewer CPU flushes/barriers) while avoiding the drawbacks (address remapping overhead). However, DudeTM still must log the actual data and apply updates to persistent storage afterward. In comparison, OSP requires no costly software mapping, need not write the actual data twice, and doesn't need to apply the updates afterward.

Virtual Memory Systems. Page overlay [17] aims to provide fine-grained memory management. However, page overlay semantics don't allow us to build durable transactions efficiently, since page consolidations are required for every transaction commit. Instead, with OSP semantics, we only need to do page consolidations upon TLB evictions. EXCITE-VM [12] leverages the indirection of virtual memory to build more efficient snapshot isolation transactions, but it doesn't address the durability issue for transactions on non-volatile memory.

5 Conclusions and Future Work

OSP extends shadow paging with cache line level mappings, allowing durable transactions to be implemented without the need to log actual data for each update and without incurring unnecessary persisting overhead. Our preliminary results show that OSP can reduce overall CPU flushes, including CPU flushes used to persist new updates, by 1.2–1.96× as compared to undo-log. We also show that OSP is able to provide a strong storage consistency guarantee with almost no cost for workloads with decent locality.

In our future work, we will investigate integrating our OSP transaction with Hardware Transactional Memory (HTM) to provide general, ACID transactions. OSP is ideally suited for HTM because it allows speculative updates to BNVM with nearly free rollback. Our future work will also address issues such as correct and efficient TLB shootdown and managing transactions that don't fit in the TLB. We also plan to investigate optimizations such as asynchronous page consolidation.

References

[1] AHN, J. H., LI, S., SEONGIL, O., AND JOUPPI, N. P. McSimA+: A manycore simulator with application-level+ simulation and detailed microarchitecture modeling. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS '13)* (2013), IEEE, pp. 74–85.

[2] CAULFIELD, A. M., DE, A., COBURN, J., MOLLOV, T., GUPTA, R., AND SWANSON, S. Moneta: A high-performance storage array architecture for next-generation, non-volatile memories. In *Proceedings of The 43rd Annual IEEE/ACM International Symposium on Microarchitecture* (2010).

[3] COBURN, J., CAULFIELD, A. M., AKEL, A., GRUPP, L. M., GUPTA, R. K., JHALA, R., AND SWANSON, S. NV-Heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2011)* (Mar. 2011).

[4] CONDIT, J., NIGHTINGALE, E. B., FROST, C., IPEK, E., LEE, B., BURGER, D., AND COETZEE, D. Better I/O through byte-addressable, persistent memory. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP '09)* (Big Sky, MT, Oct. 2009), pp. 133–146.

[5] COOPER, B. F., SILBERSTEIN, A., TAM, E., RAMAKRISHNAN, R., AND SEARS, R. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SOCC '10)* (June 2010), pp. 143–154.

[6] DULLOOR, S. R., KUMAR, S., KESHAVAMURTHY, A., LANTZ, P., REDDY, D., SANKARAN, R., AND JACKSON, J. System software for persistent memory. In *Proceedings of EuroSys 2014* (Apr. 2014).

[7] GUERRA, J., MÁRMOL, L., CAMPELLO, D., CRESPO, C., RANGASWAMI, R., AND WEI, J. Software persistent memory. In *Proceedings of the 2012 USENIX Annual Technical Conference* (2012).

[8] HU, Q., REN, J., BADAM, A., AND MOSCIBROD, T. Log-structured non-volatile main memory. In *Proceedings of the 2017 Usenix Annual Technical Conference* (June 2017).

[9] INTEL CORPORATION. Persistent memory programming. `http://http://pmem.io/`, 2015.

[10] KAWAHARA, T., ITO, K., TAKEMURA, R., AND OHNO, H. Spin-transfer torque RAM technology: Review and prospect. *Microelectronics Reliability 52*, 4 (2012), 613–627.

[11] LEE, B. C., IPEK, E., MUTLU, O., AND BURGER, D. Architecting phase change memory as a scalable DRAM alternative. In *Proceedings of the 36th Int'l Symposium on Computer Architecture* (New York, NY, USA, 2009), ISCA '09, ACM, pp. 2–13.

[12] LITZ, H., BRAUN, B., AND CHERITON, D. EXCITE-VM: Extending the virtual memory system to support snapshot isolation transactions. In *2016 International Conference on Parallel Architecture and Compilation Techniques (PACT)* (2016), IEEE, pp. 401–412.

[13] LIU, M., ZHANG, M., CHEN, K., QIAN, X., WU, Y., ZHENG, W., AND REN, J. DudeTM: Building durable transactions with decoupling for persistent memory. In *Proceedings of the 2017 International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (New York, NY, USA, 2017), ASPLOS '17, ACM, pp. 329–343.

[14] MEMARIPOUR, A., BADAM, A., PHANISHAYEE, A., ZHOU, Y., ALAGAPPAN, R., STRAUSS, K., AND SWANSON, S. Atomic in-place updates for non-volatile main memories with kamino-tx. In *Proceedings of EuroSys 2017* (New York, NY, USA, 2017), ACM, pp. 499–512.

[15] OGLEARI, M., MILLER, E. L., AND ZHAO, J. Steal but no force: Efficient hardware undo+redo logging for persistent memory systems. In *Proceedings of the 24th Int'l Symposium on High-Performance Computer Architecture (HPCA-24)* (2018), IEEE.

[16] RAOUX, S., BURR, G. W., BREITWISCH, M. J., RETTNER, C. T., CHEN, Y.-C., SHELBY, R. M., SALINGA, M., KREBS, D., CHEN, S.-H., LUNG, H.-L., AND LAM, C. H. Phase-change random access memory: A scalable technology. *IBM Journal of Research and Development 52*, 4/5 (July 2008), 465–480.

[17] SESHADRI, V., PEKHIMENKO, G., RUWASE, O., MUTLU, O., GIBBONS, P. B., KOZUCH, M. A., MOWRY, T. C., AND CHILIMBI, T. Page overlays: An enhanced virtual memory framework to enable fine-grained memory management. In *Proceedings of the 42th Int'l Symposium on Computer Architecture* (2015), IEEE, pp. 79–91.

[18] STRUKOV, D. B., SNIDER, G. S., STEWART, D. R., AND WILLIAMS, R. S. The missing memristor found. *Nature 453* (May 2008), 80–83.

[19] VENKATARAMAN, S., TOLIA, N., RANGANATHAN, P., AND CAMPBELL, R. H. Consistent and durable data structures for non-volatile byte-addressable memory. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies (FAST)* (Feb. 2011).

[20] VOLOS, H., JAAN TACK, A., AND SWIFT, M. M. Mnemosyne: Lightweight persistent memory. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2011)* (Mar. 2011).

[21] WU, X., QIU, S., AND REDDY, A. L. N. SCMFS: A file system for storage class memory and its extensions. *ACM Transactions on Storage 9*, 3 (Aug. 2013).

[22] XU, J., AND SWANSON, S. NOVA: a log-structured file system for hybrid volatile/non-volatile main memories. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST)* (Feb. 2016).

[23] YANG, J., WEI, Q., CHEN, C., WANG, C., YONG, K. L., AND HE, B. NV-Tree: Reducing consistency cost for NVM-based single level systems. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST)* (Feb. 2015), pp. 167–181.

[24] ZHAO, J., LI, S., YOON, D. H., XIE, Y., AND JOUPPI, N. P. Kiln: Closing the performance gap between systems with and without persistence support. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture* (New York, NY, USA, 2013), MICRO-46, ACM, pp. 421–432.