Memory Persistency: Semantics for Byte-Addressable Nonvolatile Memory Technologies

Implementing correct recoverable data structures requires constraints on the order of writes. This article introduces memory persistency, a framework for allowing programmers to express minimal ordering constraints on writes to byte-addressable nonvolatile memory (NVRAM). By enabling higher NVRAM write concurrency, relaxed persistency models can accelerate system throughput 30 times.

Steven Pelley
Peter M. Chen
Thomas F. Wenisch
University of Michigan

Emerging nonvolatile memory (NVRAM) technologies, such as phase change memory, memristor, or spin-torque transfer RAM, offer the durability of disk with the high performance and byte addressability of DRAM.[1] Future systems will connect such devices on a memory bus like those used for DRAM, providing memory performance approaching that of DRAM, yet also providing recoverability after failure. We anticipate that systems incorporating high-performance, byte-addressable NVRAMs will become widely available within a decade. Indeed, as cost and yield improve, such memories may become ubiquitous across the computing spectrum: they could become the preferred storage for small devices in the Internet of Things, and similarly could become critical to performance and recoverability in cloud systems.

Byte-addressable NVRAMs enable the construction of high-performance, in-memory, recoverable data structures (RDS). Conventionally, RDSs have been placed either on disk or Flash and accessed through a slow, serial, block-based interface. With NVRAM, these data structures can be accessed at word granularity via loads and stores orders of magnitude faster than disk and Flash accesses. This paradigm shift in access latency and interface implies that we must rethink how we construct these RDSs. For example, recent research has looked at ways to optimize file systems for NVRAMs.[2,3]

Programs written for such systems will have to manage the transfer of data between volatile and nonvolatile memories. Whereas programmers are familiar with designing recoverable systems using block-oriented file-system interfaces, NVRAM's byte addressability creates the potential for new and higher-performing recoverable data structures and calls for new programming interfaces to succinctly express ordering requirements. Future systems may incorporate two explicit memory spaces for volatile and nonvolatile data, or they may use a unified address space and precede nonvolatile memory spaces with volatile caches to enable write coalescing. In either scenario, the correctness of recoverable data structures relies on knowing the order of durable writes.

Existing memory hierarchies do not provide mechanisms to express and enforce write ordering all the way to NVRAM. Memory consistency requirements are enforced at the processor and do not govern write-backs from caches to memory. Ordering constraints that arise from memory consistency requirements are usually enforced at the processor, which is insufficient for failure tolerance with acceptable performance.

In our earlier work,[4] we introduced Memory Persistency, a framework motivated by memory consistency to provide an interface for enforcing the order in which NVRAM writes become durable. Persistency models prescribe ordering constraints on writes to NVRAM (an operation we refer to as a *persist*) that software and hardware mechanisms must enforce, similar to memory consistency models and ordering constraints on memory-access visibility across cores. Persistency models range from easy-to-program, low-performing models to complex, high-performing (that is, relaxed) models. These models let programmers precisely specify ordering constraints on persists by using model-specific annotations without having to worry about the underlying hardware implementation.

Our central observation is that the differing performance, endurance, and failure characteristics of multiprocessor caches and NVRAM lead to scenarios where it makes sense to decouple the ordering rules for volatile and nonvolatile accesses. Memory persistency introduces a second consistency model for the (slower and more costly) persistent writes. Such a design allows synchronization primitives and other complex, concurrent volatile data structures to be written with the simpler programming interface of strong consistency, while allowing greater concurrency through relaxed ordering for expensive persist operations. The potential performance gain from this relaxation is large: for a 500-ns NVRAM write latency, these concurrency gains improve performance to the throughput limit of instruction execution—as much as 30× speedup.

Memory persistency

Recovery mechanisms require particular orders of persists. Failure to enforce this order results in data corruption. Memory persistency prescribes the order of persist operations with respect to one another and to loads and stores, allowing the programmer to reason about guarantees on the ordering of persists with respect to system failures.

A persistency model enables software to label those persist-order constraints necessary for recovery correctness while allowing concurrency among other persists. As with consistency models, our objective is to strike a balance between the programmer's annotation burden and the amount of concurrency (and therefore, improved performance) the model enables.

We describe memory persistency as an extension of memory consistency. Memory consistency models prescribe the order of memory operations that can be observed by processors in normal operation (that is, before a failure occurs). In the same way, memory persistency models prescribe the order of memory operations that can be observed by processors after a failure. Ordering constraints for correct recovery thus become ordering constraints on memory operations as viewed by processors that execute after a failure. Using this perspective, we can apply the reasoning tools of memory consistency to persistency—any two stores to the persistent memory address space that are ordered according to the memory persistency model imply an ordering constraint on the corresponding persists. Conversely, stores that are not ordered according to the memory persistency model allow corresponding persists to be reordered or performed in parallel.

Program execution typically defines memory order as a set of memory operations and a partial order between them. We distinguish the memory order seen by processors that communicate without an intervening failure (volatile memory order, or VMO, which is constrained by memory consistency) from the memory order seen by processors that communicate across a failure (persistent memory order, or PMO, which is constrained by memory persistency).

We formalize persistency models in terms of order relations over memory events.[5] We consider two kinds of events, loads and stores (to persistent or volatile address spaces), which we collectively call memory accesses. We assume persists are performed atomically (with respect to failures) at 8-byte granularity. We use the following notation to refer to events:

- L_a^i: A load from thread *i* to address *a*.
- S_a^i: A store from thread *i* to address *a*.
- M_a^i: A load or store by thread *i* to address *a*.

We reason about two ordering relations over memory events. VMO is an ordering relation over all memory events (loads, stores, and their values), as prescribed by the consistency model. PMO comprises the same events, but events are instead ordered by the constraints imposed by the persistency model. We denote these ordering relations as

- $A \leq_v B$: A occurs no later than B in VMO.
- $A \leq_p B$: A occurs no later than B in PMO.

VMO governs the visibility of shared memory values among processors. (Our VMO is simply "memory order" in similar formalisms of consistency models.) PMO governs the visibility of shared memory values among processors across a failure—an ordering relation between stores in PMO implies the system must guarantee the same order for the corresponding persist actions, that is, $A \leq_p B \rightarrow$ B may not persist before A.

Persistency models

Persistency models provide a programming interface to prescribe required ordering relations in PMO and how they relate to VMO. We explore the performance and expressiveness of a spectrum of persistency models. Much like consistency, we identify strict and relaxed classes of persistency.

Strict persistency

Under strict persistency, the consistency model governs both volatile and persistent memory orders—that is, VMO and PMO are identical. So, for any two stores ordered by the consistency model, the corresponding persists are also ordered. Under strict persistency,

$$M_a^i \leq_v M_b^j \leftrightarrow M_a^i \leq_p M_b^j.$$

Strict persistency unifies the problem of reasoning about allowable VMO and allowable PMO (equivalently, allowable persistent states at recovery). However, directly implementing strict persistency implies frequent stalls—consistency ordering constraints stall execution until NVRAM writes complete. A programmer seeking to maximize persist performance must either rely on relaxed consistency (with the concomitant challenges of correct program labelling) or aggressively employ thread concurrency to eliminate persist-ordering constraints (introducing complexity and synchronization overheads).

Relaxed persistency

Relaxed persistency loosens persist-ordering constraints relative to the memory consistency model—that is, the visible order of stores across failures can deviate from the visible order of stores without an intervening failure (VMO and PMO contain different constraints). Decoupling persistency and consistency allows recoverable data structures with high persist concurrency even under familiar, strict consistency models, such as sequential consistency. Layering relaxed persistency on a stricter consistency model lets programmers write synchronization code with the more intuitive interface of strict consistency while still expressing high concurrency for the much slower persist operations.

Relaxed persistency models provide programmers with an additional memory event, the *persist barrier* (different from memory consistency barriers), to prescribe only those ordering constraints in PMO required to ensure recovery correctness. By removing ordering constraints unnecessary for correct recovery, relaxed persistency models increase persist concurrency relative to strict persistency, leading to a shorter critical path of persist operations.

To date, we have formally defined two relaxed persistency models—*epoch persistency,* which is inspired by recent work on the byte-addressable persistent file system,[2] and *strand persistency,* an even more relaxed model that can express an arbitrary graph of persist-ordering constraints. Both models share a foundational axiom, strong persist atomicity, which we believe to be a key property to facilitate intuitive programming while enabling highly concurrent recoverable data structures.

Strong persist atomicity

Consistency models often guarantee that stores are serialized, a property referred to as store atomicity. Persistency models could guarantee persist atomicity, wherein stores are similarly serialized across failures. However, some persistency models might allow the order of serialized stores within one failure interval to differ from the order of serialized stores across failure intervals, producing astonishing results (for example, recovery to states unreachable under fault-free execution). Instead, we argue that persistency models should provide a stronger guarantee for conflicting accesses (accesses to the same address at least one of which is a store) to preclude such nonintuitive behavior. Strong persist atomicity requires that conflicting accesses ordered in VMO are also ordered in PMO. To provide this guarantee, implementations must ensure that the order of racing stores (to one address) established by cache coherence is maintained when ordering persists. This guarantee is a powerful mechanism—we can use it to construct precise ordering dependencies across threads. We formalize strong persist atomicity in Equation 1:

$$\begin{aligned} S_a^i \leq_v M_a^j &\rightarrow S_a^i \leq_p M_a^j \\ M_a^i \leq_v S_a^j &\rightarrow M_a^i \leq_p S_a^j. \end{aligned} \qquad (1)$$

Epoch persistency

Under epoch persistency, execution in each thread is separated into epochs by persist barriers, which ensure that no persist after the barrier occurs before any persist before the barrier. We denote persist barriers issued by thread i as PB_i. Persists within each epoch (not separated by a barrier) are concurrent and may reorder or occur in parallel. Epoch persistency addresses the most common unnecessary persist dependencies (that is, not required for correct recovery) that arise due to the program-order constraint of sequential consistency. Programs frequently persist to a large, contiguous region of memory (as in a *memcpy*) in multiple persist operations without needing those persists to be ordered.

In addition to strong persist atomicity, the epoch persistency model prescribes that any two memory accesses on the same thread separated by a persist barrier are ordered in PMO, as in Equation 2:

$$M_a^i \leq_v \mathrm{PB}_i \leq_v M_b^i \rightarrow M_a^i \leq_p M_b^i. \qquad (2)$$

Each thread's execution is separated into persist epochs via persist barrier instructions. Epoch persistency uses persist barriers to order persists on the same thread (Equation 2) and a combination of strong persist atomicity (Equation 1) and persist barriers (Equation 2) to order persists across threads.

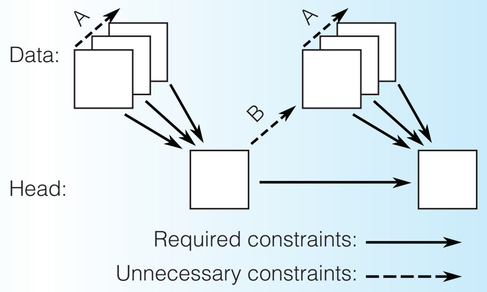
Strand persistency

Although epoch persistency relaxes persist dependencies relative to strict persistency, only persists within an epoch can be labelled as concurrent. Strand persistency lets programmers further remove unnecessary orderings, achieving the minimal set of persist-ordering constraints needed for correct recovery.

Strand persistency divides program execution into *strands,* the logically independent segments of execution that happen to execute in the same thread. The strand abstraction lets programmers express that nonconflicting execution segments (for example, transactions or tasks) are unordered with respect to persistency without incurring the overheads of placing each segment on different threads. Strands are separated by the strand barrier (SB) memory event. The strand barrier clears all prior PMO constraints from prior instructions, effectively making each strand behave as if it were a separate thread (with respect to persistency) but at a lower overhead than starting a new thread. Memory accesses within a strand are ordered using persist barriers (Equation 2), and memory accesses across strands continue to be ordered in PMO using strong persist atomicity (Equation 1). Strand barriers from thread i are denoted SB_i. The strand persistency model is defined as such: any two memory accesses on the same thread separated by a persist barrier without an intervening strand barrier are ordered in PMO:

$$\begin{aligned} &(M_a^i \leq_v \mathrm{PB}_i \leq_v M_b^i) \\ &\wedge (\forall \mathrm{SB}_i : (\mathrm{SB}_i \leq_v M_a^i) \vee (M_b^i \leq_v \mathrm{SB}_i)) \\ &\quad \rightarrow M_a^i \leq_p M_b^i. \end{aligned}$$

```
function Q.Insert(entry)
1. lock (volatile mutex)
2. PERSIST_BARRIER
3. NEW_STRAND
4. Q.Data[Head + 0] = entry[0]
5. Q.Data[Head + 1] = entry[1]
  ...
6. PERSIST_BARRIER
7. Q.Head += entry.length
8. PERSIST_BARRIER
9. unlock
```
(a)



(b)

Figure 1. Persist dependencies when inserting into a queue. (a) Pseudocode for inserting an entry in a queue. (b) Visual representation of the persist dependencies arising in the insert operation. Persist operations to the data in the queue and head pointer are indicated by boxes. Persist-ordering constraints necessary for proper recovery are shown as solid arrows; unnecessary constraints incurred under strict persistency appear as dashed arrows. Epoch persistency allows the data copy to persist concurrently (lines 4 and 5), removing the dependences labeled "A" in the diagram. Under strand persistency, the `NEW_STRAND` operation (line 3) removes the unnecessary dependence between the head update (line 7) from one insert operation and the data copy during the next (lines 4 and 5), removing the dependence labeled "B" in the diagram.

A key property of strand persistency is that it lets programmers express an arbitrary directed graph of dependencies among persist events. That is, with sufficient program annotation, any desired ordering constraints can be expressed in PMO (for example, the minimal order constraints required for correct recovery), irrespective of the ordering constraints in VMO (such as stronger ordering constraints used to correctly synchronize threads).

Applying persistency: A persistent queue

To understand and evaluate persistency models, we studied a motivating microbenchmark: a thread-safe persistent queue. Several workloads require high-performance persistent queues, such as write-ahead logs in databases and journaled file systems. Fundamentally, a persistent queue inserts and removes entries while maintaining their order. The queue must recover after failure, preserving proper entry values and order.

The goal in designing a persistent queue is to improve the persist concurrency of insert operations both through improved thread concurrency and relaxed persistency. We study implementations similar to those proposed by Fang et al.,[6] which are concurrent (thread-safe) but allow varying degrees of persist concurrency.

Figure 1a shows pseudocode for an insert operation, including the annotations required for epoch and strand persistency. Figure 1b shows a diagram of the persistent writes required for two consecutive inserts into the queue (boxes) and persist-ordering constraints among those steps that arise under strict persistency. Recovery requires that persists to the head pointer are ordered after persists to the data segment from the same insert operation, and that persists to the head pointer occur in insert-order to prevent holes in the queue (persists to the head pointer may coalesce so long as no ordering constraint is violated). All other persists within the same insert operation and between operations may occur concurrently without compromising recovery correctness.

Under strict persistency, various ordering constraints arise that are unnecessary for correct recovery (the dashed lines in Figure 1). Strict persistency under sequential consistency serializes persists to the data segment ("A" in the diagram) and serializes persists between all insert operations ("B" in the diagram). Epoch persistency allows the individual writes in the copy operation (lines 4 and 5) to persist concurrently, removing the dependences labeled "A" in the diagram. However, the head update from one insert

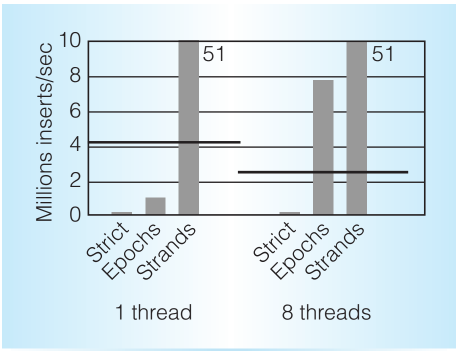


Figure 2. Persistency model performance. We evaluate performance of inserts into a persistent queue under varying persistency models. Bars indicate the performance constraint imposed by the persist-ordering critical path with a 500-ns persist latency. Lines indicate the instruction execution rate. When the bars fall short of the lines, persist constraints limit performance. Relaxed persistency models reduce ordering constraints, improving performance up to the limit of the instruction execution rate—30× over strict persistency.

operation remains serialized with the data copy of the next, as indicated by the dependence labeled "B." Under strand persistency, the NEW_STRAND operation (line 3) removes the unnecessary dependence between the head update (line 7) from one insert operation and the data copy during the next (lines 4 and 5), removing the dependence labeled "B" in the diagram.

Evaluation and impact

We measured the opportunity for relaxed persistency models to improve persist performance for our thread-safe persistent queue microbenchmark. Instead of assuming specific storage technologies and memory system implementations, we measured NVRAM write performance as the critical path of persist-ordering constraints, assuming that NVRAM writes form the primary system bottleneck and that practical memory systems effectively use available concurrency. Our results present a best-case persist performance for a given device persist latency; real devices are expected to introduce additional delays (for example, for buffering and routing within the device). Figure 2 demonstrates that relaxed persistency models substantially improve write concurrency over strict persistency. For a 500-ns NVRAM write latency, these concurrency gains improve performance to the throughput limit of instruction execution—as much as 30× speedup over strict persistency under sequential consistency.

Our work lays a foundation for reasoning about the programming interface for a system that includes both volatile and nonvolatile memories. We build on the rich literature and powerful framework of memory consistency research, allowing the familiar models and reasoning tools of consistency to be applied to persistent writes and failures. The memory consistency literature has had a clear and lasting impact on the development of computer architectures to mitigate the high cost of communicating shared memory writes. We hope that our work helps establish a similar literature to mitigate the even higher costs of writes to NVRAM.

Past works have proposed NVRAM interfaces for particular applications (such as file systems[2]); we provide a taxonomy for analyzing these instances of persistency models. Much of the existing work has focused on a transactional interface.[7–9] Our work reasons about NVRAM accesses at a lower abstraction level; transactional systems can be built on the ordering guarantees provided by a persistency model. Whereas transactions might ultimately prove to be a familiar and desirable interface for NVRAM, there is evidence (for example, our queue implementations) that higher performance may be possible when designing data structures for a relaxed persistency model.

In this article, we introduced persistency with the specific goal of designing an interface for byte-addressable NVRAM. However, the notion of layering two different consistency models for two performance classes of memory is generally applicable. For example, our persistency models might be applied in a system with two coherence domains, where performing coherence operations across domains is much more expensive than maintaining coherence locally. Persistency provides a framework for expressing two layers of ordering guarantees. There is a rich relationship between our work and prior work on distributed shared memories that remains to be explored. Such systems grow more relevant as "durability" comes to mean "replication in a distributed cloud" (for example, in schemes like RAMCloud[10]). We believe many connections between memory persistency and such past and future research topics remain to be discovered. MICRO

Acknowledgments

This work was partially supported by NSF CCF-0845157 and CNS-0905149 and by grants from ARM and Oracle. We thank Aasheesh Kolli for assistance in formalizing our descriptions of persistency models and Ali Saidi for feedback on this work.

References

1. B.C. Lee, et al., "Architecting Phase Change Memory as a Scalable DRAM Alternative," *Proc. 36th Ann. Int'l Symp. Computer Architecture*, 2009, pp. 2–13.
2. J. Condit et al., "Better I/O through Byte-Addressable, Persistent Memory," *Proc. ACM SIGOPS 22nd Symp. Operating System Principles*, 2009, pp. 133–146.
3. S.R. Dulloor et al., "System Software for Persistent Memory," *Proc. 9th European Conf. Computer Systems*, 2014, article 15.
4. S. Pelley, P.M. Chen, and T.F. Wenisch, "Memory Persistency," *Proc. 41st Ann. Int'l Symp. Computer Architecture*, 2014, pp. 265–276.
5. A. Kolli et al., "Persistency Programming 101," Non-Volatile Memory Workshop, 2015; http://nvmw.ucsd.edu/2015/assets/abstracts/33.
6. R. Fang et al., "High Performance Database Logging using Storage Class Memory," *Proc. IEEE 27th Int'l Conf. Data Eng.* (ICDE), 2011, pp. 1221–1231.
7. D.R. Chakrabarti and H.-J. Boehm, "Durability Semantics for Lock-based Multithreaded Programs," *Proc. USENIX Workshop Hot Topics in Parallelism*, 2013; www.usenix.org/conference/hotpar13/workshop-program/presentation/chakrabarti.
8. 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), 2011, pp. 91–104.
9. J. Zhao et al., "Kiln: Closing the Performance Gap between Systems with and without Persistence Support," *Proc. 46th Ann. Int'l Symp. Microarchitecture* (MICRO), 2013, pp. 421–432.
10. J. Ousterhout et al., "The Case for RAMCloud," *Comm. ACM*, vol. 54, no. 7, 2011, pp. 121–130.

Steven Pelley is a software developer at Snowflake Computing. His research interests include new database designs and memory interfaces for NVRAM. Pelley has a PhD in computer science and engineering from the University of Michigan, where he performed the work for this article. Contact him at spelley@umich.edu.

Peter M. Chen is an Arthur F. Thurnau Professor in the Department of Electrical Engineering and Computer Science at the University of Michigan. His research interests include operating systems, computer security, and fault-tolerant computing. Chen has a PhD in computer science from the University of California at Berkeley. He is a fellow of IEEE and the ACM. Contact him at pmchen@umich.edu.

Thomas F. Wenisch is an associate professor in the Department of Electrical Engineering and Computer Science at the University of Michigan. His research interests include multiprocessor and multicore systems, multicore programmability, smartphone architecture, datacenter architecture, and performance evaluation methodology. Wenisch has a PhD in electrical and computer engineering from Carnegie Mellon University. He is a member of IEEE and the ACM. Contact him at twenisch@umich.edu.