Memory Persistency

Steven Pelley Peter M. Chen Thomas F. Wenisch
University of Michigan
{spelley,pmchen,twenisch}@umich.edu

Abstract

Emerging nonvolatile memory technologies (NVRAM) promise the performance of DRAM with the persistence of disk. However, constraining NVRAM write order, necessary to ensure recovery correctness, limits NVRAM write concurrency and degrades throughput. We require new memory interfaces to minimally describe write constraints and allow high performance and high concurrency data structures. These goals strongly resemble memory consistency. Whereas memory consistency concerns the order that memory operations are observed between numerous processors, persistent memory systems must constrain the order that writes occur with respect to failure. We introduce memory persistency*, a new approach to designing persistent memory interfaces, building on memory consistency. Similar to memory consistency, memory persistency models may be relaxed to improve performance. We describe the design space of memory persistency and desirable features that such a memory system requires. Finally, we introduce several memory persistency models and evaluate their ability to expose NVRAM write concurrency using two implementations of a persistent queue. Our results show that relaxed persistency models accelerate system throughput 30-fold by reducing NVRAM write constraints.*

1. Introduction

Emerging nonvolatile memories (NVRAM) promise high performance recoverable systems. These technologies, required as replacements for Flash and DRAM as existing technologies approach scaling limits [16], pair the high performance and byte addressability of DRAM with the durability of disk and Flash memory. Future systems will place these devices on a DRAM-like memory bus, providing systems with memory performance similar to DRAM, yet recoverability after failures.

However, ensuring proper recovery requires constraints on the ordering of NVRAM writes. Existing DRAM interconnects lack the interface to describe and enforce write ordering constraints; ordering constraints that arise from memory consistency requirements are usually enforced at the processor, which is insufficient for failure tolerance with acceptable performance. Recent work has suggested alternative interfaces to enforce NVRAM write order and guarantee proper recovery, for example, durable transactions and persist barriers [28, 10]. Although intuitive and suitable to specific applications, we wish to investigate a more general framework for reasoning about NVRAM write ordering, including mechanisms for expressing write constraints that are independent of specific concurrency control mechanisms.

Instead, we recognize that the problem of ordering NVRAM writes strongly resembles memory consistency. Memory consistency restricts the visible order of loads and stores (equivalently, allowable visible memory states) between processors or cores, allowing many operations to reorder while providing the intended behavior. Memory consistency models define an interface and set of memory order guarantees for the programmer, but separate the implementation; several distinct implementations may fulfill the same memory consistency model, allowing sophisticated optimization (e.g., speculation [2, 29, 4, 14, 25]). Relaxing the memory consistency model places an additional burden on the programmer to understand the model and insert correct annotations, but often allows greater performance.

We introduce *Memory Persistency*, a framework motivated by memory consistency to provide an interface for enforcing the order NVRAM writes become durable, an operation we refer to as a "persist." Memory persistency prescribes the order of persist operations with respect to one another and loads and stores, and allows the programmer to reason about guarantees on the ordering of persists with respect to system failures; memory persistency is an extension of consistency models for persistent memory operations. The memory persistency model relies on the underlying memory consistency model and volatile memory execution to define persist ordering constraints and the values written to persistent memory.

In this paper, we define memory persistency, describe the design space of memory persistency models, and evaluate several persistency models. Much like consistency, we identify *strict* and *relaxed* classes of persistency models. Strict persistency relies on implicit guarantees to order persists and couples persistent semantics to the underlying memory consistency model: any two stores to the persistent address space that are guaranteed to be observed in a well-defined order from the perspective of a third processor imply well-ordered NVRAM writes. Thus, the same mechanisms the consistency model provides a programmer to enforce order for stores also enforce order for the corresponding persists. Alternatively, relaxed persistency separates volatile and persistent memory execution, allowing the order of persist operations to differ from the order in which the corresponding stores become visible. Relaxed persistency facilitates concurrent persists even when sequential consistency orders shared memory access visibil-

978-1-4799-4394-4/14/$31.00

ity. Whereas separating memory consistency and persistency provides advantages to programmability and performance, it also introduces new challenges, as separate annotations define allowable reorderings for visibility and persistence of writes to the persistent address space.

Using this framework, we consider successively relaxed memory persistency models (two existing models, including [10], and a newly introduced model) and demonstrate how programmers can exploit the reorderings they allow through example implementations of a thread-safe persistent queue. We discover that conservative memory consistency (such as sequential consistency) with strict persistency must rely on thread parallelism to enable NVRAM write concurrency. On the other hand, relaxed persistency allows high instruction execution performance, NVRAM write concurrency, and simplified data structures.

Finally, we evaluate our memory persistency models and queue designs. Just as with memory consistency, a memory persistency model is defined separately from its implementation. Instead of assuming specific storage technologies and memory system implementations, we measure 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. We demonstrate that relaxed persistency models substantially improve write concurrency over sequential consistency with strict persistency; for a 500ns NVRAM write latency, these concurrency gains improve performance to the throughput limit of instruction execution—as much as 30x speedup over strict persistency.

2. Background

This section provides an overview of NVRAM technologies and memory consistency.

2.1. Nonvolatile Memories

Several NVRAM storage technologies promise performance and byte-addressability comparable to DRAM with the persistence of disk and flash memory. Examples include phase change memory (PCM), which stores data as different phases in a chalcogenide glass, and spin-transfer torque memory (STT-RAM), which stores state as electron spin. Due to their high performance and byte-addressability, we expect future NVRAM devices to connect to processors via a DRAM-like bus. Whereas integrating byte-addressable persistent storage presents interesting problems to operating systems and memory management, we assume that memory provides both volatile and persistent address spaces.

NVRAMs frequently exhibit asymmetric read and write latencies; writing to a cell requires more time than reading. Long write times are compounded by using multi-level cells (MLC), which increase storage density but require iterative writes to change the cell value. As a result, NVRAM writes require up to 1µs, depending on NVRAM cell technology, device interconnect, and the use of MLC cells [23].

NVRAM technologies differ from existing storage technologies in other ways. For example, many NVRAM technologies have limited write endurance; cells may only change value a limited number of times. While important, previous work suggests efficient hardware to mitigate write-endurance concerns [24]. We do not consider write endurance in this work.

2.2. Memory Consistency

Memory consistency models allow programmers to reason about the visible order of loads and stores among threads. The most conservative model, Sequential Consistency (SC), prescribes that all loads and stores occur as some interleaving of the program orders of each thread, often requiring delays. To avoid these delays, relaxed consistency models explicitly allow certain memory operations to reorder. Threads view operations from other threads occurring out of program order, and observed orders may differ between threads. Popular examples of relaxed consistency include Total-Store Order (TSO) and Relaxed-Memory Order (RMO) [27].

We view persists as memory events for which memory consistency must account. Whereas traditional memory consistency models define allowable orders of loads and stores, memory persistency must also determine allowable orders of persists. We leverage a rich history of memory consistency work to help define the memory persistency design space.

3. Memory Persistency Goals

Correct recovery of durable systems requires persists to observe some set of happens-before relations, for example, that persists occur before an externally observable action, such as a system call. However, we expect NVRAM persists to be much slower than stores to the volatile memory system. Provided the minimal set of happens-before relations is observed, the gap between volatile execution and NVRAM write performance can be shrunk by optimizations that increase concurrency. We are interested in defining persistency models that create opportunities for two specific optimizations: *persist buffering*, and *persist coalescing*.

Persist Buffering. Buffering durable writes and allowing thread execution to proceed ahead of persistent state greatly accelerates performance [10]. Such buffering overlaps NVRAM write latency with useful execution. To be able to reason about buffering, we draw a distinction between a “store," the cache coherence actions required to make a write (including an NVRAM write) visible to other processors, and a “persist," the action of writing durably to NVRAM. Buffering permits persists to occur asynchronously; persists occur after their corresponding stores, but persists continue to execute in their properly constrained order.

Ideally, persist latency is fully hidden and the system executes at native instruction execution speed. With finite buffering, performance is ultimately limited by the slower of the

average rate that persists are generated (determined by volatile execution rate) and the rate persists complete. At best, the longest chain (critical path) of persist ordering constraints determines how quickly persists occur (at worst, constraints within the memory system limit persist rate, such as bank conflicts or bandwidth limitations). In defining persistency models, our goal is to admit as much persist concurrency as possible by creating a memory interface that avoids unnecessary constraints. Persistency model implementations might buffer persists in existing store queues and caches or via new mechanisms, such as buffers within NVRAM devices.

Persist Coalescing. We expect NVRAM devices will guarantee atomic persists of some size (e.g., eight-bytes [10]), a feature we call *persist granularity*. Multiple persists within a memory block of persist granularity may coalesce (be performed in a single persist operation) provided no happens-before constraints are violated. Persist coalescing creates an opportunity to avoid frequent persists to the same address, and allows caching/buffering mechanisms to act as bandwidth filters [15]. Coalescing also reduces the total number of NVRAM writes, which may be important for NVRAM devices that are subject to wear. Larger persist granularity facilitates greater coalescing. Similarly, persistency models that avoid unnecessary ordering constraints enable additional coalescing.

We do not consider specific implementations for persist coalescing. Coalescing may occur in software or as an extension to existing cache systems. Regardless of which mechanism coalesces persists, it must ensure that no happens-before constraints are violated.

Whereas similar performance gains may be achieved in software by the application programmer explicitly storing values in volatile memory and precisely controlling when they persist, we believe that automatic persist coalescing is an important feature. Automatic coalescing relieves the programmer of the burden to manually orchestrate coalescing and specify when persists occur via copies. Additionally, automatic coalescing provides backwards compatibility by allowing new devices to increase persist granularity and improve coalescing frequency.

4. Memory Persistency

Recovery mechanisms define specific required orders on persists. Failure to enforce this order results in data corruption. 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 programmer annotation burden and the amount of concurrency (and therefore improved performance) the model enables.

We introduce memory persistency as an extension to memory consistency to additionally order persists and facilitate reasoning about persist order with respect to failures; memory persistency is a new consistency model for persists. Conceptually, we reason about failure as a *recovery observer* that atomically reads all of persistent memory at the moment of failure. Ordering constraints for correct recovery thus become ordering constraints on memory and persist operations as viewed from the recovery observer. With this abstraction, we can apply the reasoning tools of memory consistency to persistency—any two stores to the persistent memory address space that are ordered with respect to the recovery observer imply an ordering constraint on the corresponding persists. Conversely, stores that are not ordered with respect to the observer allow corresponding persists to be reordered or performed concurrently. The notion of the recovery observer implies that even a uniprocessor system requires memory persistency as the single processor must still interact with the observer (i.e., uniprocessor optimizations for cacheable volatile memory may be incorrect for persistent memory).

In defining a new consistency model for persists, we additionally define a new memory order. Program execution typically defines memory order as a set of memory operations and a partial order between them. We distinguish the memory order of communicating processors (volatile memory order—constrained by memory consistency) from the memory order that defines persist order (persistent memory order—constrained by memory persistency). Any store operations ordered according to persistent memory order imply ordered persists. Both orderings comprise memory events to the volatile and persistent address spaces—loads and stores to the volatile address space may still order stores to the persistent address space in persistent memory order (and thus order persists). Persistent memory order contains the same *events* as volatile memory order (e.g., load and store events and their values), but the two may contain different sets of *ordering constraints*. The constraints of persistent memory order need not be a subset of the constraints from volatile memory order; complex persistency models constrain the order of persists whose underlying stores are concurrent, as we show later.

Much like consistency models, there may be a variety of implementations for a particular memory persistency model. Like the literature on consistency, we separate model semantics from implementation; our focus in this work is on exploring the semantics. Whereas we do discuss some implementation considerations, we omit details and leave system design and optimization to future work. We divide persistency models into *strict* and *relaxed* classes, and consider each with respect to the underlying consistency model.

4.1. Strict Persistency

Strict persistency couples memory persistency to the memory consistency model, using the existing consistency model to specify persist ordering. Under strict persistency, the recovery observer participates in the memory consistency model precisely as if it were an additional processor; persistent memory order is identical to volatile memory order. Hence, any store ordering that can be inferred by observing (volatile) mem-

ory order implies a persist ordering constraint. Persist order must match the (possibly partial) order in which stores are performed in a particular execution.

Conservative consistency models, such as SC, do not allow stores from each thread to reorder from the perspective of other threads; all stores, and therefore persists, occur in each thread's program order. However, such models can still facilitate persist concurrency by relying on thread concurrency (stores from different threads are often concurrent). On the other hand, relaxed consistency models, such as RMO, allow stores to reorder. Using such models, it is possible for many persists from the same thread to occur in parallel. However, the programmer is now responsible for inserting the correct memory barriers to enforce the intended behavior, as is currently the case for shared-memory workloads.

Strict persistency unifies the problem of reasoning about allowable volatile memory order and allowable persist order (equivalently, allowable persistent states at recovery). However, directly implementing strict persistency implies frequent stalls—consistency ordering constraints (e.g., at every memory operation under SC and at memory barriers under RMO) stall execution until NVRAM writes complete. A programmer seeking to maximize persist performance must rely either on relaxed consistency (with the concomitant challenges of correct program labelling), or must aggressively employ thread concurrency to eliminate persist ordering constraints (introducing complexity and synchronization overheads). As we will show, decoupling persistency and consistency ordering allows recoverable data structures with high persist concurrency even under SC.

We introduce one important optimization to strict persistency, *buffered strict persistency*, which can improve performance while still guaranteeing strict ordering of persists and visible side effects. Buffered strict persistency allows instruction execution to proceed ahead of persistent state, thus allowing overlap of volatile execution and serial draining of queued persist operations. In terms of the recovery observer, buffered strict persistency allows the observer to lag arbitrarily far behind other processors in observing memory order. Therefore, the persistent state of the system corresponds to some prior point in the observable memory order. As side effects may otherwise become visible prior to earlier persists, we introduce a *persist sync* operation to synchronize instruction execution and persistent state (i.e., require the recovery observer to "catch up" to present state). The persist sync allows the programmer to order persists and non-persistent, yet visible, side effects. While we recognize the need for persist sync, we do not consider its design, nor evaluate it in this work.

4.2. Relaxed Persistency

Strict persistency provides mechanisms to reason about persist behavior using pre-existing memory consistency models. However, memory consistency models are often inappropriate for persist performance. Conservative consistency models such as SC and TSO serialize the visible order of stores; although high performance implementations of these models exist for volatile instruction execution [26, 17], the high latency of NVRAM persists suggests that more relaxed persistency models may be desirable.

We decouple memory consistency and persistency models via *relaxed persistency.* Relaxed persistency loosens persist ordering constraints relative to the memory consistency model—that is, the visible order of persists (from the perspective of the recovery observer) is allowed to deviate from the visible order of stores (volatile and persistent memory orders contain different constraints). Relaxing persistency requires separate memory consistency and persistency barriers. Memory consistency barriers enforce the visibility of memory operation ordering with respect to other processors, while memory persistency barriers constrain the visible order of persists from the perspective of only the recovery observer.

Relaxing persistency allows systems with conservative consistency, such as SC, to improve persist concurrency without requiring additional threads or complicating thread communication. In the rest of this paper we introduce and explore relaxed persistency models under SC.

Simultaneously relaxing persistency and consistency allows the visibility of loads and stores to reorder among processors, and further allows the order of persists to differ from the order of stores. An interesting property of such systems is that memory consistency and persistency barriers are decoupled—store visibility and persist order are enforced separately—implying that persists may reorder across store barriers and store visibility may reorder across persist barriers. Separating store and persist order complicates reasoning about persists to the same address, as we show next.

4.3. Relaxed Persistency and Persist Atomicity

Memory models with separate consistency and persistency barriers allow stores and persists to occur in different orders, including stores and persists to a single address. While unintuitive, this decoupling is desirable when synchronization (such as a lock) guarantees that races cannot occur; treating persist barriers additionally as store barriers would unnecessarily delay instruction execution. We extend the concept of store atomicity to define and enforce intended behavior.

Consistency models often guarantee *store atomicity*—stores to each address are serialized—a property provided by cache coherence [1]. Similarly, persistency models may guarantee *persist atomicity*—persists to each address are serialized, implying that recovery determines a unique value for each address (our recovery observer implies persist atomicity, but failure models involving several recovery observers would not). Memory systems with both store and persist atomicity may still allow the order of stores and persists to deviate—the serialized order of stores differs from the serialized order of persists. We define *strong persist atomicity* to describe pro-

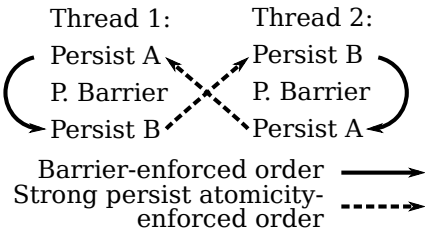

Figure 1: Cache Coherence Ordered Persists. Thread 1's store visibility reorders while still attempting to enforce persist order. The resulting persist order cycle is resolved by violating strong persist atomicity or by preventing stores from reordering across persist barriers.

gram executions that observe store and persist atomicity, and where the order of stores to each address agrees with the order of persists. Strong persist atomicity allows programmers to order persists by relying on store atomicity to order stores to the same address. Just as relaxed consistency may not always provide store atomicity (e.g., release consistency), relaxed persistency may not guarantee persist atomicity. Such persistency models must define new notions of "properly labelled programs" that guarantee strong persist atomicity.

Despite being an intuitive property, strong persist atomicity removes many memory reordering opportunities. The example in Figure 1 demonstrates that it is not possible to simultaneously (1) allow store visibility to reorder across persist barriers, (2) enforce persist barriers, and (3) guarantee strong persist atomicity. The example considers two distinct objects in the persistent address space, A and B. Two threads persist to these objects in different program orders. Thread 1's execution reorders the visibility of stores, while Thread 2 executes its stores in program order. Note that persist barriers signify that we still intend for thread 1's persist to B to occur after its persist to A, but the values produced by these operations may become visible to other processors out of program order.

We annotate Figure 1 with the persist order constraints (happens-before relationships) due to persist barriers and strong persist atomicity. As shown, these constraints form a cycle. If A and B cannot persist atomically, the defined order cannot be enforced. The cycle can be resolved by either coupling persist and store barriers—every persist barrier also prevents store visibility from reordering—or by relaxing strong persist atomicity, providing additional barriers to enforce strong persist atomicity only where needed.

5. Persistency Models

Section 4 outlined potential classes of persistency models. We now introduce several specific persistency models to be evaluated later in Section 8. All models assume SC as the underlying memory consistency model, and successively relax persistency to introduce specific optimizations. For each model we discuss its motivation, give a definition, describe necessary program annotations, and offer possible implementations.

5.1. Strict Persistency

Motivation. Our first persistency model is Strict Persistency, as discussed in Section 4. Strict persistency simplifies reasoning about persist ordering by coupling persist dependences to the memory consistency model. No additional persist barriers are required, easing the burden on the programmer. While strict persistency provides an intuitive first model, under SC it imposes persist ordering constraints that unnecessarily limit persist concurrency for many data structures, and requires programmers to resort to multi-threading to obtain concurrency.

Definition. Under strict persistency, persist order observes all happens-before relations implied by volatile memory order (i.e., volatile and persistent memory orders are identical). Thus, all persists are ordered with respect to the program order of the issuing thread. Note that, like store operations, persists from different threads that are unordered by happens-before (i.e., the recovery observer cannot distinguish which is first) are concurrent.

Implementation. A straight-forward implementation of strict persistency stalls issue of subsequent memory accesses until a store and its corresponding persist both complete. Conventional speculation mechanisms may allow loads to speculatively reorder with respect to persistent stores [13]. Buffered strict persistency can be implemented by serializing persists to a single, totally ordered queue in front of persistent memory (e.g., in a bus-based multiprocessor, persists can be queued after they are serialized by the bus). Delays occur when buffers fill or when persist sync instructions drain the queue.

Advanced implementations might use one persist queue per thread/core or extensions to the existing cache system. Mechanisms are required to ensure that persists on each thread occur in program order and that persists obey strong persist atomicity. Load-before-store races must correctly order persists between threads; enforcing such orders remains a challenge even for existing memory consistency (hence the popularity of TSO). Strict persistency under SC may also be implemented using in-hardware NVRAM logs or copy-on-write and indirection to give the appearance of SC while persists occur concurrently. An intriguing possibility couples existing hardware transactional memories (HTM) and persistent transactions (e.g., [30, 28]), partitioning program execution into transactions that enforce atomicity, isolation, and durability—resembling persistent BulkSC [5].

5.2. Epoch Persistency

Motivation. Strict persistency under SC introduces many persist dependences unnecessary for correct recovery. The most common unnecessary persist dependence occurs due to the program-order constraint of SC. Programs frequently persist to a large, contiguous regions of memory that logically represent single objects, but which cannot persist atomically (due to their size). Under strict persistency, persists serialize.

Previous work proposed for the Byte-Addressable File Sys-

tem defined a new model to allow concurrent persists from each thread while constraining persist order when necessary (we refer to this model as BPFS) [10]. Doing so requires annotation by the programmer in the form of persist barriers to divide execution into persist epochs. Epoch persistency is similar to BPFS (we introduce subtle differences, described below). Epoch persistency additionally allows persists to addresses protected by a lock to reorder with respect to the lock operations (e.g., avoid delaying the lock release while the persist completes).

Definition. Epoch persistency separates volatile and persistent memory orders; persistent memory order contains a subset of constraints from volatile memory order. Any pair of persists ordered by persistent memory order may not be observed out of that order with respect to the recovery observer.

Volatile memory order satisfies SC. Each thread's execution is additionally separated into *persist epochs* by persist barrier instructions. Epoch persistency inherits persistent memory order constraints from volatile memory order: (1) any two memory accesses on the same thread and separated by a persist barrier are ordered, (2) any two memory accesses that conflict (are to the same or overlapping addresses and at least one is a store/persist) assume the order from volatile memory order (strong persist atomicity is guaranteed), and (3) eight-byte persists are atomic with respect to failure.

Persist barriers enforce that no persist after the barrier may occur before any persist before the barrier. Persists within each epoch (not separated by a barrier) are concurrent and may reorder or occur in parallel. Additional complexity arises in reasoning about persist ordering across threads. We define a *persist-epoch race* as persist epochs from two or more threads that include memory accesses that race (to volatile or persistent memory), including synchronization races, and at least two epochs include persist operations. Persists between racing epochs may not be ordered, even though the underlying stores are ordered by SC—epochs are not serializable. However, strong persist atomicity (rule 2) continues to order persists. Consequently, two persists to the same address are always ordered even if they occur in racing epochs.

Discussion. Epoch persistency provides an intuitive mechanism to guarantee proper recovery as it is impossible at recovery to observe a persist from after a barrier while failing to observe a persist from before the same barrier. However, many persists (those within the same epoch) are free to occur in parallel, improving persist concurrency.

As noted in our definition, reasoning about persist order across threads can be challenging. Synchronization operations within persist epochs impose ordering across the store and load operations (due to SC memory ordering), but do not order corresponding persist operations. Hence, persist operations correctly synchronized under SC by volatile locks may nevertheless result in astonishing persist ordering. A simple (yet conservative) way to avoid persist-epoch races is to place persist barriers before and after all lock acquires and releases, and to only place locks in the volatile address space. The persist behavior of strict persistency can be achieved by preceding and following all persists with a persist barrier.

Persist-epoch races may be intentionally introduced to increase persist concurrency; we discuss such an optimization in Section 6. Enforcing persist order between threads with volatile locks requires that the persists be synchronized outside of the epochs in which the persists occur. However, synchronization through persistent memory is possible. Since persists to the same address must observe strong persist atomicity, even if they occur in epochs that race, the outcome of persist synchronization is well defined. Hence, atomic read-modify-write operations to persistent memory addresses provide the expected behavior.

Epoch persistency is inspired by BPFS. However, we introduce several subtle differences that we believe make epoch persistency a more intuitive model. Our definition considers all memory accesses when determining persist ordering among threads, whereas BPFS orders persists only when conflicts occur to the persistent address space (i.e., persistent memory order contains only accesses to the persistent address space). Whereas the BPFS file system implementation avoids persist-epoch races, it is not clear that the burden falls to the programmer to avoid such accesses or what persist behavior results when such races occur. Furthermore, BPFS detects conflicts to the persistent address space by recording the last thread and epoch to persist to each cache line; the next thread to access that line will detect the conflict. Such an implementation, however, cannot detect conflicts where the first access is a load and the second a store. As a result, BPFS detects conflicts to persistent memory according to TSO rather than SC ordering [27].

Implementation. BPFS outlines cache extensions that provide a persistency model similar to epoch persistency. Modifications must be made to detect load-before-store conflicts (and thus enforce SC rather than TSO ordering) and to track conflicts to volatile memory addresses as well as persistent memory addresses. Instead of delaying execution to enforce persist ordering among threads, optimized implementations avoid stalling execution by buffering persists while recording and subsequently enforcing dependences among them, allowing persists to occur asynchronously despite access conflicts.

5.3. Strand Persistency

Motivation. Epoch persistency relaxes persist dependences within and across threads. However, only consecutive persists within a thread may be labelled as concurrent. Likewise, persists from different threads are only concurrent if their epochs race or if they are not synchronized. Many persists within and across threads may still correctly be made concurrent even if they do not fit these patterns. We introduce *strand persistency*, a new model to minimally constrain persist dependences.

Definition. A strand is an interval of memory execution from a single thread. Strands are separated by *strand barrier*

instructions; each strand barrier begins a new strand. The strand barrier clears all previously observed persist dependences from the executing thread—each strand appears in persist order as a separate thread. Persist barriers continue to order persists: memory accesses on the same *strand* separated by a persist barrier assume the order observed from volatile memory order. Strand persistency guarantees strong persist atomicity and eight-byte atomic persists. Accesses from different strands, even from the same thread, are concurrent unless ordered by strong persist atomicity.

Discussion. There are no implicit persist ordering constraints across strands for persists to different addresses on the same thread of execution. Ordering constraints arise only for persists to the same address as implied by strong persist atomicity. Hence, persists on a new strand may occur as early as possible and overlap with all preceding persists. Strand persistency allows programmers to indicate that logical tasks on the same thread are independent from the perspective of persistency. To enforce necessary ordering, a persist strand begins by reading persisted memory locations after which new persists must be ordered. These reads introduce ordering dependences through strong persist atomicity, which can then be enforced with a subsequent persist barrier. Strand persistency removes all unintended ordering constraints, maximizing persist concurrency. Additionally, this programming interface allows ordering constraints to be specified at the granularity of individual addresses; the minimal set of persist dependences is achieved by placing each persist in its own strand, loading all addresses the persist must depend on, inserting a persist barrier, and then executing the persist.

Implementation. Strand persistency builds on the hardware requirements to track persist dependencies in epoch persistency, but separates dependence tracking for different strands. In addition to tracking the last thread to access each address, the strand within the thread must also be tracked. Unordered persists on different strands traverse separate queues (e.g., on separate virtual channels) throughout the memory system. Strand persistency gives enormous implementation latitude and designing efficient hardware to track and observe only the minimal ordering requirements remains an open research challenge. In this work, we focus on demonstrating the potential performance that the model allows.

5.4. Summary

Relaxed persistency offers new tools to enforce recovery correctness while minimizing delays due to persists. We consider three persistency models to successively relax persist ordering constraints and improve persist concurrency. Our models are built on top of SC, but may be easily modified to interact with relaxed consistency models. The next section uses these persistency models to outline the software design of a persistent queue.

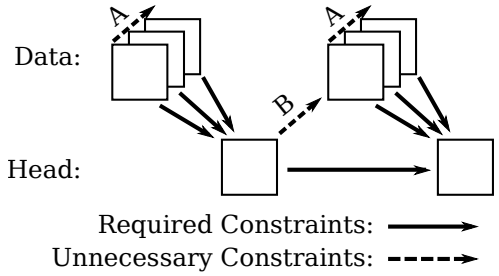

Figure 2: Queue Persist Dependences. Persist ordering dependences for *Copy While Locked* and *Two-Lock Concurrent*. Constraints necessary for proper recovery shown as solid arrows; unnecessary constraints incurred by strict persistency under SC appear as dashed arrows and are labelled as A (removed with epoch persistency) and B (further removed by strand persistency).

6. Persistent Queue

To understand and evaluate persistency models we introduce a motivating microbenchmark: a thread-safe persistent queue. Several workloads require high-performance persistent queues, such as write ahead logs (WAL) 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. Both our designs are concurrent (thread-safe) but allow varying degrees of persist concurrency. Additionally, our designs are fashioned as circular buffers, containing a data segment in addition to head and tail pointers. Psuedo-code for the designs is shown in Algorithm 1. We outline their execution, recovery, and the minimal necessary persist dependences.

The first design, *Copy While Locked* (CWL), serializes insert operations with a lock, first persisting each entry's length and data to the data segment, then persisting the new head pointer. As a result, persists from subsequent insert operations, even if they occur on separate threads, are ordered by lock accesses. If the system fails before the persist to the head pointer in line 9, the entry is ignored and the insert has failed.

We improve persist concurrency in the second design, *Two-Lock Concurrent* (2LC), by using two different locks to reserve data segment space and persist to the head pointer, respectively. Neither lock is held while entry data persists to the data segment, allowing concurrent persists from different threads. Additionally, a volatile *insert list* is maintained to detect when insert operations complete out of order and prevent holes in the queue. *Two-Lock Concurrent* employs the same recovery as *Copy While Locked*—an entry is not valid and recoverable until the head pointer encompasses the associated portion of the data segment.

Both queue designs use the persistency model to prevent persists to the head pointer from occurring before persists to

Algorithm 1 Psuedo-code for queue insert operations. We include the annotations required by our relaxed persistency models, discussed in Section 5. *PersistBarrier* **applies to epoch persistency and strand persistency,** *NewStrand* **applies only to strand persistency.**

Require: *head* is a persistent pointer, *data* a persistent array.

```
1:  sl ← SIZEOF(length)
2:  function INSERTCWL(length, entry)
3:      PERSISTBARRIER
4:      LOCK(queueLock)
5:      PERSISTBARRIER                    ▷ removing allows race
6:      NEWSTRAND
7:      COPY(data[head], (length, entry), length + sl)
8:      PERSISTBARRIER
9:      head ← head + length + sl
10:                                       ▷ strong persist atomicity serializes
11:     PERSISTBARRIER                    ▷ removing allows race
12:     UNLOCK(queueLock)
13:     PERSISTBARRIER
14: end function
15:
16: function INSERT2LC(length, entry)
17:     LOCK(reserveLock)
18:     start ← headV; headV ← headV + length + sl
19:     node ← insertList.APPEND(headV)
20:     UNLOCK(reserveLock)
21:     NEWSTRAND
22:     COPY(data[start], (length, entry), length + sl)
23:     LOCK(updateLock)
24:     (oldest, newHead) ← insertList.REMOVE(node)
25:         ▷ double-checked lock may acquire reserveLock
26:     if oldest then
27:         PERSISTBARRIER
28:         head ← newHead
29:                                       ▷ strong persist atomicity serializes
30:     end if
31:     UNLOCK(updateLock)
32: end function
```

the data segment. Algorithm 1 includes barriers for the persistency models described in Section 5. Additionally, persist dependences (and unnecessary constraints introduced by strict persistency models) are shown in Figure 2. 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 occur concurrently without compromising recovery correctness. While not necessary for correct recovery, these persist dependences are difficult to describe minimally; ordering mechanisms often introduce unnecessary persist constraints (dashed lines in the Figure).

Strict persistency under SC serializes persists to the data segment ("A" in the Figure) and serializes persists between all insert operations ("B" in the Figure). Epoch persistency allows concurrent persists to the data segment (removes "A") but still serializes persists between insert operations. Intentionally allowing persist-epoch races, instead synchronizing inserts by relying on strong persist atomicity to serialize persists to the head pointer, allows concurrent persists from inserts on different threads ("B" occurs only between inserts on the same thread). Finally, strand persistency removes all unnecessary dependences ("B" is removed entirely).

Our queues are similar to the design proposed by Fang [11]. *Two-Lock Concurrent* provides an equivalent mechanism to allow threads to concurrently copy entries into the data segment. As in *Copy While Locked*, Fang's queue contains persists ordered by a critical section, and consequently achieves similar persist throughput under our models.

7. Methodology

We evaluate our persistent queue designs and persistency models to measure the opportunity for relaxed persistency models to improve persist performance. To this end we measure instruction execution rate on a real server and persist concurrency via memory traces. This section outlines our methodology for these experiments.

All experiments run the queue benchmarks optimized for volatile performance. Memory padding is inserted to objects and queue inserts to provide 64-byte alignment to prevent false sharing (conflicting cache accesses to disjoint memory locations). Critical sections are implemented using MCS locks [20], a high-throughput queue based lock. Experiments insert 100-byte queue entries. Instruction execution rate is measured as inserts per second while inserting 100,000,000 entries between all threads using an Intel Xeon E5645 processor (2.4GHz). The remainder of this section describes how we measure persist concurrency for the queue benchmarks.

Persist Ordering Constraint Critical Path. Instead of proposing specific hardware designs and using architectural simulation, we instead measure, via memory traces, persist ordering constraint critical path. Our evaluation assumes a memory system with infinite bandwidth and memory banks (so bank conflicts never occur), but with finite persist latency. Thus, persist throughput is limited by the longest chain (critical path) of persist ordering constraints observed by execution under different memory persistency models. Whereas real memory systems must necessarily delay elsewhere due to limited bandwidth, bank conflicts, and other memory-related delays incurred in the processor, measuring persist ordering constraint critical path offers a best case, implementation-independent measure of persist concurrency.

We measure persist critical path under the following assumptions. Every persist to the persistent address space occurs in place (there is no hardware support for logging or indirection to allow concurrent persists). We track persist dependences at

variable granularity (e.g., eight-byte words or 64-byte cache lines). Coarse-grained persist tracking is susceptible to false sharing—persists to nearby but disjoint addresses may be ordered unnecessarily. We similarly vary persist granularity. Every persist attempts to coalesce with the last persist to that address. A persist successfully coalesces if the persist fits within an atomically persistable memory block and coalescing with the previous persist to the same block does not violate any persist order constraints. A persist that cannot coalesce is ordered after the previous persist to the same address.

Memory Trace Generation. We use PIN to instrument the queue benchmarks and generate memory access traces [19]. Tracing multi-threaded applications requires additional work to ensure analysis-atomicity—application instructions and corresponding instrumentation instructions must occur atomically, otherwise the traced memory order will not accurately reflect execution's memory order. We provide analysis-atomicity by creating a bank of locks and assigning every memory address (or block of addresses) to a lock. Each instruction holds all locks corresponding to its memory operands while being traced. In addition to tracing memory accesses, we instrument the queue benchmarks with persist barriers and persistent malloc/free to distinguish volatile and persistent address spaces. As only one instruction from any thread can access each address at once, and instructions on each thread occur in program order, our trace observes SC. Our tracing framework is available online [22].

Performance Validation. It is important that tracing not unduly influence thread interleaving, which would affect persist concurrency. We measure the distance of insert operations between successive inserts from the same thread (i.e., how many insert operations occurred since the last time this thread inserted). We observe that the distribution of insert distance is the same when running each queue natively and with instrumentation enabled, suggesting that thread interleaving is not significantly affected.

Persist Timing Simulation. Persist times are tracked per address (both persistent and volatile) as well as per thread according to the persistency model. For example, under strict persistency each persist occurs after or coalesces with the most recent persists observed through (1) each load operand, (2) the last store to the address being overwritten, and (3) any persists observed by previous instructions on the same thread. Persists' ability to coalesce is similarly propagated through memory and thread state to determine when coalescing will violate persist ordering constraints. The persistency models differ as to the events that propagate persist ordering constraints through memory and threads.

We use this methodology to establish a need for relaxed persistency models, as well as measure their opportunity to accelerate recoverable systems.

8. Evaluation

We use the previously described methodology to demonstrate that persist ordering constraints present a performance bottleneck under strict persistency. Relaxed persistency improves persist concurrency, removing stalls caused by persists. We also show that relaxed persistency models are resilient to large persist latency, allowing maximum throughput as persist latency increases. Finally, we consider the effects of persist granularity and dependence tracking granularity.

8.1. Relaxed Persistency Performance

NVRAM persists are only a concern if they slow down execution relative to non-recoverable systems. If few enough persists occur, or those persists are sufficiently concurrent, performance remains bounded by the rate that instructions execute with few delays caused by persists. To determine system performance, we assume that only one of the instruction execution rate and persist rate is the bottleneck: either the system executes at its instruction execution rate (measured on current hardware), or throughput is limited solely by persist rate (while observing persist dependencies and retaining recovery correctness).

Table 1 shows the achievable throughput for our queue microbenchmarks and persistency models for both one and eight threads assuming 500ns persists. Rates are normalized to instruction execution—normalized rates above one (bold) admit sufficient persist concurrency to achieve the instruction execution rate while normalized rates below one are limited by persists. Instruction execution rates vary between log version and number of threads (not shown).

Strict persistency, our most conservative model, falls well below instruction execution rate, suggesting that memory systems with such restrictive models will be persist-bound. *Copy While Locked* with one thread suffers nearly a 30× slowdown; over-constraining persist order greatly limits workload throughput.

Relaxing persistency improves throughput for persist-bound configurations. We consider epoch persistency both with and without persist-epoch races. "Epoch" (similar to BPFS) prevents persist-epoch races by surrounding lock accesses with persist barriers; persists are always ordered across critical sections. "Racing Epochs" removes this constraint, allowing persist-epoch races and enforcing persist order via strong persist atomicity. There is no distinction between the two when using a single thread (races cannot occur within one thread) and for *Two-Lock Concurrent* (concurrent persists are already provided by the software design).

Epoch persistency improves persist concurrency by allowing entire queue entries to persist concurrently and removes a number of unnecessary persist constraints via intentional persist-epoch races. Both queue designs see a substantial increase in throughput, with the eight-thread *Two-Lock Concurrent* achieving instruction execution rate. Other configurations

Threads	Copy While Locked				Two-Lock Concurrent			
	Strict	Epoch	Racing Epochs	Strand	Strict	Epoch	Racing Epochs	Strand
1	0.034	0.17	0.17	**12**	0.080	0.56	0.56	**29**
8	0.058	0.40	**3.2**	**21**	0.43	**3.4**	**3.4**	**22**

Table 1: Relaxed Persistency Performance. Persist-bound insert rate normalized to instruction execution rate assuming 500ns persist latency. Throughput is limited by the lower of persist and instruction rates—at greater than 1 (bold) instruction rate limits throughput; at lower than 1 execution is limited by the rate of persists. While strict persistency limits throughput, relaxed models achieve instruction execution rate.

remain persist-bound; their throughput suffers relative to a nonrecoverable system. Epoch persistency is insufficient to maximize system performance without relying on multithreading to provide concurrent persists. Nevertheless, *Copy While Locked* with one thread is now only 5.9× slower than the instruction execution rate. On the other hand, racing epochs improve persist concurrency for 8-thread configurations, allowing persist throughput to surpass instruction execution rate.

While execution for all queue designs with many threads is already compute bound and does not benefit from further relaxing persistency, the single thread configurations require additional persist concurrency to improve performance. Strand persistency allows concurrent persists from the same thread while still ensuring correct recovery. This model enables incredibly high persist concurrency such that all log versions are compute-bound even for a single thread. Sufficiently relaxed persistency models allow data structures and systems that recover from failure while retaining the throughput of existing main-memory data structures.

Persist Latency. The previous results argue for relaxed persistency models under large persist latency. However, for fast enough NVRAM technologies, additional persist concurrency is unnecessary to achieve instruction execution rate. Figure 3 shows the achievable execution rate (limited either by persist rate or instruction execution rate) for *Copy While Locked* with one thread. The x-axis shows persist latency on a logarithmic scale, ranging from 10ns to 100µs.

At low persist latency, all persistency models achieve instruction execution rate (horizontal line formed at the top). However, as persist latency increases each model eventually becomes persist-bound and throughput quickly degrades. Strict persistency becomes persist-bound at only 17ns. Epoch persistency improves persist concurrency—instruction execution rate and persist rate break even at 119ns. While this is a great improvement, we expect most NVRAM technologies to exhibit higher persist latency. Finally, strand persistency offers sufficient persist concurrency to become persist-bound only above 6us.

In all cases, throughput quickly decreases once execution is persist-bound as persist latency continues to increase. Persists limit the most conservative persistency models even at DRAM-like write latencies. However, relaxed persistency models are resilient to large persist latencies and achieve instruction execution rate.

8.2. Atomic Persist and Tracking Granularity

The previous experiments consider performance for queue designs and persistency models assuming that persist ordering constraints propagate through memory at eight-byte granularity (i.e., a race to addresses in the same eight-byte, aligned memory block introduces a persist ordering constraint according to the persistency model). Additionally, we assume that persists occur atomically to eight-byte, aligned memory blocks. Both of these may vary in real implementations; we measure their effect on persist ordering constraint critical path for *Copy While Locked* using a single thread.

Atomic Persist Granularity. Atomic persist granularity is an important factor for persist concurrency and performance. As in [10], we assume NVRAM persists atomically to at least eight-byte (pointer-sized) blocks of memory (a persist to an eight-byte, aligned memory block will always have occurred or not after failure; there is no possibility of a partial persist). However, increasing persist granularity creates opportunities for additional persist coalescing. Nearby or adjacent persists may occur atomically and coalesce so long as no persist dependences are violated. If the originally enforced ordering between two persist operations appeared on the persist dependence critical path, coalescing due to increased atomic persist granularity may decrease the critical path and reduce the likelihood of delay due to persists.

Figure 4 displays average persist ordering critical path per insert for *Copy While Locked* for both strict persistency and epoch persistency as atomic persist size increases from eight to 256 bytes. At eight-byte persists, there is a large separation between strict persistency and epoch persistency. As atomic persist size increases, the persist critical path of strict persistency steadily decreases while the critical path of epoch persistency remains unchanged. At 256-byte atomic persists (right of the Figure) strict persistency matches epoch persistency. For our queue microbenchmarks, larger atomic persists provide the same improvement to persist critical path as relaxed persistency, but offer no improvement to relaxed models. Increasing atomic persist granularity offers an alternative to relaxed persistency models.

Persist False Sharing. Just as in existing memory systems, persists suffer from false sharing, degrading performance. False sharing traditionally occurs under contention to the same cache line even though threads access disjoint addresses in that cache line. Similarly, *persistent false sharing* occurs when

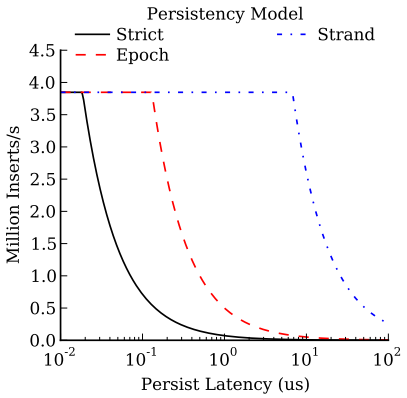

Figure 3: Persist Latency. *Copy While Locked*, 1 thread. All models initially compute-bound (line at top). As persist latency increases each model becomes persist-bound. Relaxed models are resilient to large persist latency.

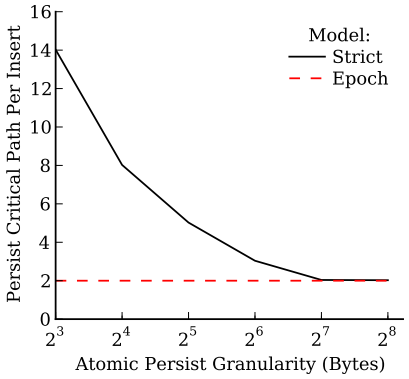

Figure 4: Atomic Persist Size. 1 Thread. Large atomic persists allow coalescing, increasing persist concurrency. While effective for strict persistency, large atomic persists do not improve persist concurrency for relaxed models.

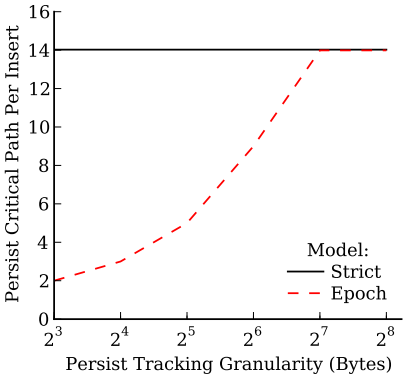

Figure 5: Persistent False Sharing. 1 Thread. False sharing negligibly affects strict persistency (persists already serialized); relaxed models reintroduce constraints.

a persist ordering constraint is unnecessarily introduced due to the coarseness at which conflicts are observed. Persistent false sharing occurs in conflicts to both persistent and volatile memory, as conflicts to both address spaces establish persist ordering constraints.

Figure 5 shows average persist critical path per insert for *Copy While Locked* for both strict persistency and epoch persistency as the granularity at which persist ordering constraints propagate increases. For fine-grained tracking, epoch persistency provides a far lower persist critical path than strict persistency. As tracking granularity increases, strict persistency performance remains the same while epoch persistency decreases (critical path increases). At 256-byte tracking granularity, strict persistency and epoch persistency provide comparable persist critical paths; many of the persist constraints removed by relaxing consistency are reintroduced through false sharing.

9. Related Work

Durable storage has long been used to recover data after failures. All systems that use durable storage must specify and honor dependencies between the operations that update that storage. For example, file systems must constrain the order of disk operations to metadata to preserve a consistent file system image [12, 7], and databases must obey the order of durable storage updates specified in write-ahead logging [21].

Specifying and honoring these dependencies becomes harder when the interface to durable storage are loads and stores to a persistent address space. Store instructions to an address space are more frequent and fine grained than update operations when using a block-based interface to durable storage (such as a file system). In addition, CPU caches interpose on store instructions, which leads to the interaction of persistency and cache consistency discussed in this paper.

Recent developments in nonvolatile memory technologies have spurred research on how to use these new technologies. Some research projects keep the traditional block-based interface to durable storage and devise ways to accelerate this interface [3]. Other projects provide a memory-based interface to durable storage [9]. Our paper follows the path of providing a memory-based interface to durable storage, because we feel that the high speed and byte addressability of new nonvolatile memories provides a natural fit with native memory instructions.

Combining a memory interface to durable storage with multiprocessors adds concurrency control issues to those of durability. Transactions are a common and powerful paradigm for handling both concurrency control and durability, so many authors have proposed layering transactions on top of nonvolatile memory [18, 9, 28, 8]. Similarly, a recent paper proposes to couple concurrency control with recovery management by committing execution to durable storage at the granularity of the outermost critical section [6]. Additionally, researchers recently proposed Kiln, work concurrent to ours that outlines a memory and cache system that provide persistent transactions [30]. Kiln importantly introduces the ability to separately enforce multithreaded and persist synchronization (transactions are atomically persistent, but provide no guarantee of isolation between threads), a feature central to relaxed persistency.

While transactions and critical sections are powerful mechanisms for concurrency control, many programs use other mechanisms besides these, such as conditional waits. Because of this diversity of concurrency control, we believe it is useful to treat the issues of consistency and persistency separately. Just as much work has been done to create a framework of memory consistency models [1], we seek to begin a framework on memory persistency models.

The techniques most closely related to those proposed in this paper are the primitives for describing persist dependencies in the Byte-Addressable Persistent File System (BPFS) [10]. We assume similar mechanisms in this paper. We view BPFS as a single point in the memory persistency design space. While similar to our persist epochs design, there are subtle, yet important differences, described in Section 5.2. We investigate the more-general design space of memory persistency and its interactions with memory consistency, including consideration of persist-epoch races.

10. Conclusion

Future NVRAM technologies offer the performance of DRAM with the durability of disk. However, existing memory interfaces are incapable of leveraging this performance while simultaneously enforcing proper data recovery. In this paper we introduced *memory persistency*, an extension of memory consistency that allows programmers to describe persist order constraints. We outline the design space of possible memory persistency models. In addition, we detail three persistency models and their use in implementing a persistent queue. Using memory tracing and simulation we demonstrate that strict persistency models suffer a 30× slowdown relative to instruction execution rate, and that relaxed persistency effectively regains this performance.

Acknowledgements

This work was partially supported by NSF CCF-0845157 and CNS-0905149 and by grants from ARM and Oracle.

References

[1] S. V. Adve and K. Gharachorloo, "Shared memory consistency models: a tutorial," *IEEE Computer*, vol. 29, no. 12, December 1996.

[2] C. Blundell, M. M. Martin, and T. F. Wenisch, "Invisifence: Performance-transparent memory ordering in conventional multiprocessors," in *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA)*, 2009.

[3] A. M. Caulfield, A. De, J. Coburn, T. I. Mollov, R. K. Gupta, and S. Swanson, "Moneta: A High-Performance Storage Array Architecture for Next-Generation, Non-volatile Memories ," in *Proceedings of the 2010 International Symposium on Microarchitecture (MICRO)*, December 2010.

[4] L. Ceze, J. Tuck, P. Montesinos, and J. Torrellas, "Bulksc: Bulk enforcement of sequential consistency," in *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA)*, 2007.

[5] L. Ceze, J. Tuck, P. Montesinos, and J. Torrellas, "Bulksc: Bulk enforcement of sequential consistency," in *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA)*, 2007.

[6] D. R. Chakrabarti and H.-J. Boehm, "Durability Semantics for Lock-based Multithreaded Programs," in *Proceedings of the 2013 USENIX Workshop on Hot Topics in Parallelism (HotPar)*, June 2013.

[7] V. Chidambaram, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Optimistic crash consistency," in *Proceedings of the 2013 Symposium on Operating System Principles (SOSP)*, November 2013.

[8] J. Coburn, T. Bunker, M. Shwarz, R. K. Gupta, and S. Swanson, "From ARIES to MARS:Transaction Support for Next-Generation Solid-State Drives ," in *Proceedings of the 2009 Symposium on Operating System Principles*, October 2009.

[9] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson, "NV-Heaps: Making Persistent Objects Fast and Safe with Next-Generation, Non-Volatile Memories," in *Proceedings of the 2011 International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, March 2011.

[10] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, D. Burger, B. Lee, and D. Coetzee, "Better I/O Through Byte-Addressable, Persistent Memory," in *Proceedings of the 2009 Symposium on Operating System Principles*, October 2009.

[11] R. Fang, H.-I. Hsiao, B. He, C. Mohan, and Y. Wang, "High performance database logging using storage class memory," in *Proc. of the 27th International Conf. on Data Engineering (ICDE)*, 2011.

[12] G. R. Ganger, M. K. McKusick, C. A. N. Soules, and Y. N. Patt, "Soft Updates: A Solution to the Metadata Update Problem in File Systems," *ACM Transactions on Computer Systems*, vol. 18, no. 2, May 2000.

[13] K. Gharachorloo, A. Gupta, and J. Hennessy, "Two techniques to enhance the performance of memory consistency models," in *In Proceedings of the 1991 International Conference on Parallel Processing (ICPP)*, 1991.

[14] C. Gniady, B. Falsafi, and T. N. Vijaykumar, "Is sc + ilp = rc?" in *Proceedings of the 26th Annual International Symposium on Computer Architecture (ISCA)*, 1999.

[15] J. R. Goodman, "Using cache memory to reduce processor-memory traffic," in *Proceedings of the 10th Annual International Symposium on Computer Architecture (ISCA)*, 1983.

[16] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting phase change memory as a scalable dram alternative," in *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA)*, 2009.

[17] C. Lin, V. Nagarajan, R. Gupta, and B. Rajaram, "Efficient sequential consistency via conflict ordering," *SIGARCH Comput. Archit. News*, vol. 40, no. 1, Mar. 2012.

[18] D. E. Lowell and P. M. Chen, "Free Transactions with Rio Vista," in *Proceedings of the 1997 Symposium on Operating Systems Principles (SOSP)*, October 1997.

[19] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation (PLDI)*, 2005.

[20] J. M. Mellor-Crummey and M. L. Scott, "Algorithms for scalable synchronization on shared-memory multiprocessors," *ACM Trans. Comput. Syst.*, vol. 9, no. 1, Feb. 1991.

[21] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz, "ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging," *ACM Transactions on Database Systems*, vol. 17, no. 1, March 1992.

[22] S. Pelley, "Atomic memory tracing," https://github.com/stevenpelley/atomic-memory-trace.

[23] M. K. Qureshi, M. M. Franceschini, A. Jagmohan, and L. A. Lastras, "Preset: Improving performance of phase change memories by exploiting asymmetry in write times," in *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA)*, 2012.

[24] M. K. Qureshi, J. Karidis, M. Franceschini, V. Srinivasan, L. Lastras, and B. Abali, "Enhancing lifetime and security of pcm-based main memory with start-gap wear leveling," in *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2009.

[25] P. Ranganathan, V. S. Pai, and S. V. Adve, "Using speculative retirement and larger instruction windows to narrow the performance gap between memory consistency models," in *Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, 1997.

[26] A. Singh, S. Narayanasamy, D. Marino, T. Millstein, and M. Musuvathi, "End-to-end sequential consistency," in *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA)*, 2012.

[27] *The SPARC Architecture Manual*, Version 9 ed., SPARC International, Inc., 1994.

[28] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: Lightweight Persistent Memory," in *Proceedings of the 2011 International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, March 2011.

[29] T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos, "Mechanisms for store-wait-free multiprocessors," in *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA)*, 2007.

[30] J. Zhao, S. Li, D. H. Yoon, Y. Xie, and N. P. Jouppi, "Kiln: closing the performance gap between systems with and without persistence support," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2013.