Delegated Persist Ordering

Aasheesh Kolli*, Jeff Rosen†, Stephan Diestelhorst‡, Ali Saidi‡, Steven Pelley†,
Sihang Liu*, Peter M. Chen* and Thomas F. Wenisch*
* University of Michigan {akolli,liush,pmchen,twenisch}@umich.edu
† Snowflake Computing {jeff.rosen,steven.pelley}@snowflake.net
‡ ARM {stephan.diestelhorst,ali.saidi}@arm.com

***Abstract*—Systems featuring a load-store interface to persistent memory (PM) are expected soon, making in-memory persistent data structures feasible. Ensuring persistent data structure recoverability requires constraints on the order PM writes become persistent. But, current memory systems reorder writes, providing no such guarantees. To complement their upcoming 3D XPoint memory, Intel has announced new instructions to enable programmer control of data persistence. We describe the semantics implied by these instructions, an ordering model we call *synchronous ordering*.**

Synchronous ordering (SO) enforces order by stalling execution when PM write ordering is required, exposing PM write latency on the execution critical path. It incurs an average slowdown of 7.21× over volatile execution without ordering in PM-write-intensive benchmarks. SO tightly couples enforcing order and flushing writes to PM, but this tight coupling is unneeded in many recoverable software systems. Instead, we propose *delegated ordering*, wherein ordering requirements are communicated explicitly to the PM controller, fully decoupling PM write ordering from volatile execution and cache management. We demonstrate that delegated ordering can bring performance within 1.93× of volatile execution, improving over SO by 3.73×.

***Keywords* —** persistent memory, memory persistency, relaxed consistency, delegated ordering

I. Introduction

New persistent memory (PM) technologies with the potential to transform how software manages persistent data will soon be available. For example, Intel and Micron have announced their 3D XPoint memory technology for availability in 2017 [1], and competing offerings may follow [2]. Such devices are expected to provide much lower access latency than NAND Flash, enabling access to persistent data with a load-store interface like DRAM rather than the block-based I/O interface of Flash and disk. Persistent memory systems will allow programmers to maintain recoverable data structures in main memory.

Ensuring recoverability of persistent data structures requires constraints on the order writes become persistent [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15]. At the same time, it is desirable that PM accesses are cacheable, both to hide access latency and to enable write coalescing to conserve write bandwidth and lifetime for devices subject to wearout. Conventional memory systems delay, combine, and reorder writes to memory at multiple levels, and do not enforce any particular correspondence between the order stores are executed and writes to memory are performed. Some proposals enable recoverability by requiring persistence at all memory hierarchy levels [5], [16], [17], [18], ensuring writes are persistent as soon as they retire. However, such schemes either require replacing SRAM caches with PM or mechanisms to flush caches rapidly upon failure, which may be impractical if the aggregate cache capacity is large and challenging to implement for non-power faults (e.g., kernel panic).

Recent work proposes that programming systems be extended with a *memory persistency model*; an explicit specification and programming interface to constrain the order writes become persistent [4]. A memory persistency model is analogous to the memory consistency model that governs the ordering of reads and writes to shared memory, but instead constrains the order PM writes become persistent, an operation we refer to as a *persist*.

To complement upcoming memory technology offerings, Intel [3] has announced instruction set extensions to enable programmer control of data persistence. The `clwb` instruction allows programmers to initiate write back of specific addresses to PM, and the `pcommit` and `sfence` instructions enable order enforcement among these writebacks and subsequent execution. To our knowledge, no prior academic study has examined the semantics and performance implications of these extensions. We describe the persistency model implied by the semantics of these instructions; we refer to the model as *synchronous ordering*.

Synchronous ordering enforces order by stalling execution, preventing instructions ordered after a `pcommit` from retiring until prior PM writes complete. However, this approach tightly couples volatile execution and persistent writes, placing PM write latency on the execution critical path. As we will show, these stalls can result in a 7.21× slowdown in workloads with frequent PM writes.

Synchronous ordering couples two orthogonal operations: prescribing an order between PM writes and flushing the writes to persistent storage. However, coupling these operations is often unnecessary for software system recoverability, as data structure consistency depends principally upon the order writes become persistent [19], [9], [4], [13]. In many contexts, volatile execution may proceed ahead of properly ordered PM writes without compromising recoverability, hiding PM latency. When rare failures occur, some writes may be lost, but data structure consistency is maintained (e.g., journaling file systems maintained on disks [9]).

978-1-5090-3508-3/16/$31.00 © 2016 IEEE

In this paper, we explore a new implementation approach to enforcing persistency model ordering requirements for PM writes. Instead of enforcing ordering through stalls, we investigate an implementation approach, which we call *delegated ordering*, that communicates partial ordering requirements mandated by the persistency model explicitly to the PM controller. Delegated ordering decouples persistency model implementation from both volatile execution and cache management. Execution and communication via shared memory proceed while PM writes drain. Caches remain volatile and may communicate through cache coherence and evict blocks at will. Instead, our approach maintains persistent writes alongside the cache hierarchy in per-core *persist buffers*. Using annotations added to coherence transactions, the persist buffers observe and track persist order dependencies mandated by the persistency model. Together, they serialize PM writes into a partially ordered buffer at the PM controller, which may then schedule PM writes to exploit available bank concurrency.

Delegated ordering represents a fundamental departure from existing persistency implementations and common approaches for enforcing (volatile) write ordering for relaxed memory consistency. Relaxed consistency is often implemented like synchronous ordering, relying on stalling (e.g., at memory fence instructions) to prevent mis-ordering the visibility of memory operations. Since communication via cache coherence is comparatively fast, the cost of these stalls is tractable, and can be further hidden through speculation [20], [21], [22], [23], [24], [25]. In contrast, storing to PM is relatively slow, and we show that stalling operations at the processor to enforce ordering severely degrades performance. Delegated ordering instead tracks write ordering constraints explicitly. Execution does not stall unless buffering resources in the persist buffers and at the PM controller are exhausted or the programmer explicitly requests the stall (e.g., before issuing an irrecoverable action).

We evaluate delegated ordering by implementing it for buffered strict persistency [4], [13] (PM writes must reflect the order stores become globally visible, but their persistence may be delayed) and compare its performance to synchronous ordering. We evaluate both approaches in a cache hierarchy that implements ARM's relaxed memory consistency model, which allows reordering and concurrency among stores between ordering points. We implement a series of PM-write-intensive benchmarks, adding minimal fence instructions required for correctness under each model, and evaluate performance using the gem5 simulation infrastructure [26]. We compare both approaches for different PM technologies. In summary, our contributions are:

- We analyze the semantics and performance of synchronous ordering—the persistency model implied by Intel's ISA extensions for PM [3]—and demonstrate that it results in an 7.21× slowdown on average relative to volatile execution without ordering.
- We propose delegated ordering, an approach to memory persistency implementation that exposes partial ordering constraints explicitly to the PM controller.
- We evaluate delegated ordering and demonstrate that it improves performance by 3.73× on average over synchronous ordering for PM-write-intensive benchmarks, coming within 1.93× of volatile execution without order enforcement.

II. Synchronous Ordering

In today's systems, enforcing the desired order of persists to PM is complicated by the presence of programmer-transparent hardware structures (e.g., caches and memory controllers) that actively re-order writes to gain performance. Programmers must choreograph explicit cache flush operations (for instance `clflush` in x86 and `dccmvac` in ARM v7) to achieve the desired order of PM updates.

Prior to newly proposed instruction extensions, programmatically enforcing a desired PM write order was challenging and, in some systems, impossible [27]. Bhandari and co-authors [28] examine the subtle x86 coding idioms required to ensure proper ordering. These idioms rely on `clflush` to explicitly write back dirty lines, requiring hundreds of cycles to execute [28] and invalidating the cache line, incurring a compulsory miss upon the next access.

Intel's recently announced extensions [3] provide mechanisms to guarantee recovery correctness and improve upon the performance deficiencies of `clflush`. Nevertheless, we will show that these extensions still incur large stalls, motivating our pursuit of higher performance mechanisms. We first explore the semantics and then the performance implications of these extensions.

A. Semantics

Synchronous ordering (SO) is our attempt to describe the persistency model implied by the semantics of Intel's ISA extensions [3]. We briefly describe the most relevant of these new instructions:

- `clwb`: Requests writeback of modified cache line to memory; clean copy of cache line may be retained.
- `pcommit`: Ensures that stores that have been accepted to memory are persistent when the `pcommit` becomes globally visible.

Executing a `clwb` instruction, by itself, does not ensure data persistence because the PM controller is permitted to have volatile write queues that may delay a PM write even after the `clwb` operation is complete. The semantics of `pcommit` are subtle; it is a request to the PM controller to flush its write queues. However, `pcommit` execution is not implicitly ordered with respect to preceding or following stores or `clwb` operations. Hence, neither `pcommit` nor `clwb` alone assure persistence.

A store operation to cacheable ("write back") memory is assured to be "accepted to memory" when a `clwb` operation ordered after the store becomes globally visible ([3] p. 10-8). However, since `pcommit` is not ordered with respect to the completion of `clwb` operations, an intervening `sfence` is needed to ensure the `clwb` is globally visible. Similarly,

a fence operation is required after the pcommit to order its global visibility with respect to subsequent stores.

With these two instructions, stores on one thread to addresses A and B can be guaranteed to to be updated in PM in the order $A < B$, using the following pseudo-code:

```
st A;clwb A;sfence;pcommit;sfence;st B;
```

We refer to the code sequence sfence; pcommit; sfence as a *sync barrier*. The first sfence orders the pcommit with earlier stores and clwbs, while the second orders later stores with the pcommit.

B. Performance

SO's overhead can be broken into two components:

- The overhead due to clwb instructions. clwb writes modified data back from the cache hierarchy to the memory controller. Each clwb must snoop all caches (including private caches of peer-cores) for a cache block in dirty state and write it back to the PM. Effectively, each clwb instruction incurs a worst-case on-chip access latency.
- The overhead due to pcommit instructions. A pcommit does not complete until *all* writes that have been accepted to memory are persistent (regardless of which processor issued them).

Modern out-of-order (OoO) cores can often hide some access latencies, however the sfence instructions (required for correct ordering) preceding and following a pcommit will expose the latency of the clwb and pcommit operations. A PM write may take several times as long as a DRAM write [29] and drastically reduces the effectiveness of OoO mechanisms.

We study the performance of SO over three different PM designs:

- DRAM: a battery-backed DRAM.
- PCM: a Phase Change Memory (PCM) technology.
- PWQ: a persistent write queue at the PM controller that ensures data becomes durable when it arrives at the PM controller (e.g., because a supercapacitor guarantees enqueued writes will drain despite failure). We assume a PCM main memory.

PWQ provides freedom to the PM controller to arbitrarily schedule writes for performance without compromising persistency guarantees. Studying these three configurations highlights the technology independence of the observations made in this paper. We contrast performance under two different persistency models:

Volatile: Under this model, benchmarks are run without support for persistence and are vulnerable to corruption in the event of a failure. We use this as a baseline to measure the costs of persistence.

Synchronous Ordering (SO): Under this model, the necessary clwb, sfence, and pcommit operations are inserted ensuring data persistence to PM. Note that in case of PWQ, ensuring the data reaches the PM controller in the desired order is sufficient to guarantee correct recovery. Hence, SO requires only clwb and sfence operations, without any pcommits. The latency of the cache flush operations is exposed on the execution critical path (at the sfence), but the PM write itself is not. Increased execution times for SO (over a baseline volatile execution) for our PM-centric benchmarks (please see Section V for methodology details) are shown in Table I:

PM design	Geo. mean	Range
PWQ	1.54×	1.12× to 2.1×
DRAM	2.97×	1.16× to 7.96×
PCM	7.21×	1.33× to 35.15×

TABLE I: Slowdowns due to SO over volatile execution.

C. Discussion

SO suffers from four main drawbacks that hamper its performance and usability. First, it couples the operation that prescribes the order between PM writes with the operation that flushes the writes to persistent storage. In many contexts, stalling execution until the flush is complete is not needed to assure data consistency and recoverability. Rather, such stalling is needed only before irrecoverable side effects, such as sending a network packet. We believe that one of the major deficiencies of the proposed model is that no mechanisms are provided to ensure ordering of writes to PM without requiring completion.

Second, a programmer must explicitly enumerate the addresses for which persistence is needed via flush operations. Although flexible, this interface greatly complicates software development. For example, one cannot easily construct a software library that provides transactional persistence semantics for a user-supplied data structure while hiding the details of the persistency model (the user must supply a list of addresses requiring clwb operations). In contrast, fence operations in relaxed consistency [30] and relaxed persistency [4] do not require addresses to be enumerated, facilitating synchronization primitives in libraries.

Third, the pcommit operation does not complete until *all* operations accepted to memory are persistent, even those issued by other threads. In contrast, memory fences typically stall only until preceding operations from or observed by the same thread are globally visible. A pcommit may be significantly delayed by PM writes of an unrelated application, where the relative PM write interleaving is immaterial.

Finally, the clwb instruction relinquishes write permission to a cache block, which will unnecessarily incur a coherence transaction to obtain write permission if the block is written again. Coherence state is orthogonal to persistency.

Interestingly, ARM has also recently announced a new instruction for persistence support as part of ARM V8.2 [31]. The new instruction dccvap (data cache clean virtual address to point of persistence), is similar to clwb in that, it forces a writeback of a cache block. However, unlike clwb, dccvap requires the writeback to become persistent (reach PM), rather

than just being accepted at the PM controller, obviating the need for a separate `pcommit`-like instruction. The `dccvap` instruction implies a different synchronous persistency model than SO (which is based on Intel's `clwb`, and `pcommit` instructions). However, a complete ARM V8.2 specification is not yet available.

We can address synchronous ordering's deficiencies with delegated ordering. Before we describe delegated ordering (Section IV), we next discuss the persistency model and semantics provided by our approach.

III. Memory Persistency

Delegated ordering is an implementation strategy for persistency models that allow volatile execution to proceed ahead of store persistence—a store may retire and become globally visible in shared memory before it is persistent. Synchronous ordering allows visibility and persistence to deviate only until a `pcommit` is executed. Several persistency models proposed in the literature similarly allow such buffering, including BPFS [7], buffered strict persistency [4], [13], (buffered) epoch persistency [4], [13], and strand persistency [4]. Prior models have been proposed and evaluated in the context of sequential consistency. However, in this work, we target the ARM v7 architecture [32], which provides a relaxed consistency model.

In this section, we detail the semantics of buffered strict persistency when applied to ARMv7 consistency, yielding a model that we call *relaxed consistency buffered strict persistency*, or RCBSP. Because ARMv7 already allows store reordering between memory fences, RCBSP enables concurrency among persist operations similar to what is allowed by BPFS and under epoch persistency in sequentially consistent systems, without the need to introduce new fence instructions for persists. We first provide brief background and then precisely specify RCBSP using nomenclature from Pelley [4] and notation derived from Kolli [33].

A. Background

Formally, we express an ordering relation over memory events *loads*, *stores*, and *fences* (collectively *accesses*). The term *persist* refers to the act of durably writing a store to persistent memory. We assume persists are performed atomically (with respect to failures) at naturally aligned 8-byte granularity. By "thread", we refer to execution contexts—cores or hardware threads. We use the following notation:

- L_a^i: A load from thread i to address a
- S_a^i: A store from thread i to address a
- F^i: A fence (full strength *dmb*) from thread i.
- M_a^i: A load/store/fence by thread i (to address a)

Further, we use the following notation for dependencies between memory events:

- $M_a^i \xrightarrow{d} M_b^i$: An addr/data/control dependence from M_a^i to M_b^i, two accesses on the same thread.
- $S_a^i \xrightarrow{rf} L_a^j$: A load "reads from" [34] a prior store.
- $L_a^i \xrightarrow{fr} S_a^j$: A store "from reads" [34] a prior load.

We reason about three ordering relations over memory events, *local memory order*, *volatile memory order* and *persist memory order*.

Local memory order (LMO^i) is an ordering relation over all memory events (loads and stores), observed by thread i, prescribed by the memory consistency model [30]. In relaxed consistency models, especially non-multi-copy-atomic models like ARMv7 [34], [35], different threads may legally disagree on the order in which stores become visible. It is important to note that, *no thread disagrees* with at least a subset of ordering relations, for example, coherence order and orderings enforced by fence cumulativity [35], [34], [36], [37]. Volatile memory order (VMO) is an ordering relation over all memory events as observed by a hypothetical thread that atomically reads all contents of persistent memory at the moment of failure (defined as "recovery observer" in [4]). Note that VMO agrees with all other threads w.r.t. coherence order and fence cumulativity. Persist memory order (PMO) is an ordering relation over all memory events but may have different ordering constraints than any LMO^i or VMO. PMO is governed by the "memory persistency model" [4].

We denote these ordering relations as:

- $A \leq_{li} B$: A occurs no later than B in LMO^i
- $A \leq_{v} B$: A occurs no later than B in VMO
- $A \leq_{p} B$: A occurs no later than B in PMO

An ordering relation between stores in PMO implies the corresponding persist actions are ordered; that is,

$A \leq_p B \rightarrow B$ may not persist before A.

Based on the relationship between VMO and PMO, Pelley classifies persistency models into two types: *strict* and *relaxed*. Under strict persistency, the PMO is the same as VMO, that is, a programmer uses the memory consistency model to govern both store visibility and the order in which stores persist. Under relaxed persistency, PMO and VMO may differ, that is, a programmer needs to reason separately about store visibility and the order in which stores persist.

The motivation for relaxed persistency arises because of the use of conservative consistency models such as sequential consistency (SC) and total store order (TSO). These consistency models require a strict order (in VMO) for all stores and allow little re-ordering or coalescing. Pelley shows that following the same strict order for persists (each of which could take 100s of nano-seconds [29]), hinders performance, much like synchronous ordering. Relaxed persistency models allow programmers to impose a different set of constraints for the PMO than the VMO, thereby allowing more re-order-ing and coalescing in the PMO. Pelley shows that the additional parallelism afforded to persists by relaxed persistency models significantly improves performance.

Even though relaxed persistency models improve performance by exposing additional parallelism, they increase the burden on the programmer by forcing her to reason about two different memory models. ARMv7 consistency already enables parallelism among memory accesses and requires reasoning about proper ordering of shared memory accesses (including non-multi-copy-atomic stores). In this context, we

consider the alternate choice of using strict persistency. This choice of relaxed consistency and strict persistency exposes persist parallelism but does not saddle the programmer with an additional memory model. Instead, reasoning about recovery is akin to reasoning about an additional thread.

B. Buffering

Pelley further suggests that buffering persists in hardware will expose more opportunities to re-order and coalesce, thus improving performance. Buffering implies (under strict persistency) that some stores, which have already been executed by a processor and are visible to other processors, might not yet have been persisted to PM. However, the hardware (memory hierarchy) guarantees that, eventually, all these stores will persist *in the order dictated by the VMO*. This ordering guarantee is the key distinction between buffered strict persistency and the behavior of writeback caches under synchronous ordering, in which PM writes drain from the cache hierarchy in arbitrary order in the absence of `clwb` and `pcommit` instructions.

The coupling must be enforced by an additional mechanism that records dependencies between stores (both inter- and intra-thread) and honors the recorded dependencies while persisting the stores. BPFS [7] and epoch barriers [13] achieve such ordering by constraining L1 cache write backs. We propose new ordering mechanisms, decoupled from the caches, in Section IV.

Buffering improves performance by ensuring that the memory hierarchy does not have to persist an executed store immediately, but can perform the persist eventually, as long as the correct order is maintained [4], [13]. Volatile execution and cache coherence may proceed while the persist operation is drained lazily to PM.

C. RCBSP

We describe the semantics of buffered strict persistency under ARMv7 relaxed consistency. Memory events on the same thread are locally ordered by:

- Executing a *FENCE* instruction between them in program order. Formally:

$$M_a^i; F^i; M_b^i \rightarrow M_a^i \leq_{li} F^i \leq_{li} M_b^i \tag{1}$$

- Using an address/data/control dependence between a memory access and a subsequent memory access in program order. Formally:

$$M_a^i \xrightarrow{d} M_b^i \rightarrow M_a^i \leq_{li} M_b^i \tag{2}$$

Further, a thread may "observe" memory events on an another thread using "reads from" and "from reads" dependencies [34]. Formally:

$$S_a^i \xrightarrow{rf} L_a^j \rightarrow S_a^i \leq_{lj} L_a^j \tag{3}$$

$$L_a^i \xrightarrow{fr} S_a^j \rightarrow L_a^i \leq_{lj} S_a^j \tag{4}$$

Memory events are globally ordered across threads using coherence and fence cumulativity [35], [36], [34], [37].

Core-0		Core-1	
S_X^0:	$St\ X = x$		
F^0:	$FENCE$		
S_Y^0:	$St\ Y = y$		
		L_Y^1:	$r1 = Ld\ Y$
		S_Z^1:	$St\ Z = r1$

Fig. 1: Fence cumulativity example.

Coherence: Two stores to the same address are globally ordered, that is, all threads agree on the order of stores (from any thread) to the same address.

$$\forall (S_a^i, S_a^j), (S_a^i \leq_v S_a^j) \vee (S_a^j \leq_v S_a^i) \tag{5}$$

Fence Cumulativity: Loosely, a FENCE (F_i) instruction provides ordering in VMO between the set of all memory accesses (from any thread) ordered before the FENCE (Group G_A) and the set of all memory accesses (from any thread) ordered after the FENCE (Group G_B). The set of memory accesses belonging to G_A can be constructed using the following algorithm [38], [35]:

(1) $\forall M_a^i \mid M_a^i \leq_{li} F^i, G_A = G_A \cup M_a^i$
(2) Repeat:
(3) $\quad \forall (M_a^i \in G_A, M_b^j) | M_b^j \leq_v M_a^i, G_A = G_A \cup M_b^j$
(4) $\quad \forall (M_a^i \in G_A, M_b^j) | M_b^j \leq_{li} M_a^i, G_A = G_A \cup M_b^j$

Line (1) indicates all memory accesses thread-locally ordered before the FENCE belong to Group G_A. The next steps recursively add to G_A additional accesses transitively observed before the FENCE. Line (3) adds all accesses ordered by VMO before any in G_A. Line (4) for each access in G_A, adds accesses ordered before it w.r.t its thread's LMO in G_A. The algorithm stops when no new accesses can be added to G_A.

Group G_B is similarly constructed from accesses after the FENCE. Once G_A and G_B are constructed, fence cumulativity offers the following guarantee:

$$\forall (M_a^i \in G_A, M_b^j \in G_B), M_a^i \leq_v M_b^j \tag{6}$$

The example in Figure 1 (a variant of the ISA2 litmus test from [34]) highlights fence cumulativity. A FENCE (F^0) instruction is executed on Core-0. So, S_X^0, preceding F^0, is placed in G_A. Note that S_X^0 is the only member of G_A. S_Y^0 is placed in G_B. We assume that L_Y^1 "reads from" S_Y^0, and hence gets added to G_B. The data dependency between L_Y^1 and S_Z^1, requires that S_Z^1 gets added to G_B. So, from Eq 6, we have that $S_X^0 \leq_v S_Z^1$, implying that all threads can only observe S_Z^1 after S_X^0. Interestingly, S_Y^0 and S_Z^i are not ordered in VMO as they both belong to the G_B.

Under strict persistency, PMO = VMO. Formally:

$$M_a^i \leq_v M_b^j \leftrightarrow M_a^i \leq_p M_b^j \tag{7}$$

Specifically under RCBSP, Eq 7, allows two behaviors:
1) Two stores to the same persistent address on different threads will persist in coherence order.
2) Two stores to persistent addresses, one belonging to G_A,

and the other belonging to G_B of a $FENCE$ (on any thread), will persist in order (G_A before G_B)

D. Discussion

When programming for persistence, to guarantee two stores persist in order, the programmer must ensure that a hypothetical thread would observe the stores in the desired order. This requirement even holds for single-threaded applications, where programmers rarely concern themselves with memory consistency models. Formally defining a consistency model is a complex task [35], [36], [34], [37] and is ill-suited to a page-limited conference paper. The intent behind the definitions above is not to fully and precisely specify ARMv7, but rather to highlight the ways in which a programmer can use the memory consistency model to order persists. We have manually verified that our RCBSP definitions enforce required persist order for each of the litmus tests presented in [34]. (More specifically, we confirmed that RCBSP precludes recovery from observing outcomes forbidden by any litmus test). Nevertheless, automatic formal verification (e.g., via a proof assistant), is beyond the scope of this paper.

IV. Delegated Ordering

We now describe delegated ordering, our implementation strategy for RCBSP persistency.

A. Design goals

Delegated ordering is based on four key design goals:

Enforce persist ordering: Under RCBSP, the persist order must match the store order given by the consistency model (as is necessary for strict persistency). Under ARMv7, intra-thread ordering arises from $FENCE$s, which divide the instructions within a thread into epochs. Stores to PM from the same epoch may persist concurrently (assuming they are not ordered by the fence cumulativity property, Eq 6, of a remote FENCE), however, stores from successive epochs must be totally ordered. Inter-thread persist ordering arises from coherence order (Eq 5) and fence cumulativity (Eq 6). When accesses conflict, corresponding persists (and their cumulative dependents) must occur in cache coherence order. Our implementation must observe, record, and enforce these persist dependencies.

Decouple data persistence from volatile execution: Under SO, ensuring the desired persist order frequently stalls execution. RCBSP decouples persist order enforcement from thread execution, by buffering persists in hardware.

Express lane for persists: Under SO, persists reach PM via successive writebacks from subsequent cache levels. Such an architecture optimizes for read performance at the expense of write latency—an appropriate trade-off for volatile memory. However, in PM-write-intensive applications, persist latency plays a major role in determining recoverable system performance. Some epoch persistency implementations buffer unpersisted stores in the cache hierarchy [7], [13]. However, buffering unpersisted stores in cache implies that a later store to the same address may not become globally visible until the prior store has persisted, leading to stalls. Moreover, performance-sensitive cache replacement policies may have to be modified to account for the desired persist order. We provide a separate persist path with dedicated storage that reduces persist latency and decouples persist order from cache eviction and store visibility.

Expose ordering constraints explicitly: Central to our strategy is the principle of exposing ordering constraints explicitly to the PM controller, a significant departure from SO and conventional memory consistency implementations, which stall to enforce order. Our intuition is that the PM controller is the proper (indeed, only) system component that manages the precise timing of PM reads and writes, has knowledge of which physical addresses are assigned to which banks, and has visibility into the conditions under which PM accesses can be concurrent.

Prior research has proposed sophisticated schedulers at DRAM controller to jointly optimize access latency, concurrency, and fairness [39], [40], [41]. Unlike DRAM controllers, the PM controller is additionally expected to honor persist ordering constraints. Initial research on PM-aware scheduling is under way [8], [42]. Our goal is to communicate persist ordering constraints to the PM controller as precisely and minimally as possible, providing it maximal scheduling flexibility, and yet without placing burdens on the cache hierarchy.

Delegated ordering succeeds in decoupling persistency enforcement *entirely* from cache management. Nevertheless, the goal of communicating *minimal* ordering constraints is aspirational; to reduce hardware complexity, our design serializes per-core persists into a single, partially ordered write queue at the PM controller, which is insufficient to represent a fully general dependence graph among persists. A single write queue cannot represent a dependence graph where two accesses must be ordered by an epoch boundary, but a third access is unordered with respect to both—we must place the third access in either the first epoch or the second, introducing an unnecessary constraint. Nevertheless, our design provides prodigious performance advantage; we believe the remaining gap to the performance of unordered volatile execution does not warrant additional hardware complexity to communicate minimal constraints.

B. System Architecture

Figure 2 shows our system architecture to implement delegated ordering for RCBSP. Responsibility for ensuring proper persist ordering is divided between persist buffers, located alongside each L1 D-cache, and the PM controller, which ultimately issues persist operations. The persist buffers each track persist requests and fences from their associated core to discover intra-thread persist dependencies and monitor cache coherence traffic to discover inter-thread persist dependencies. The buffers coordinate to then serialize their per-core persist operations into a single, partially ordered write queue at the PM controller through the path marked “Persist requests” in Figure 2. (We show a dedicated persist path for clarity;

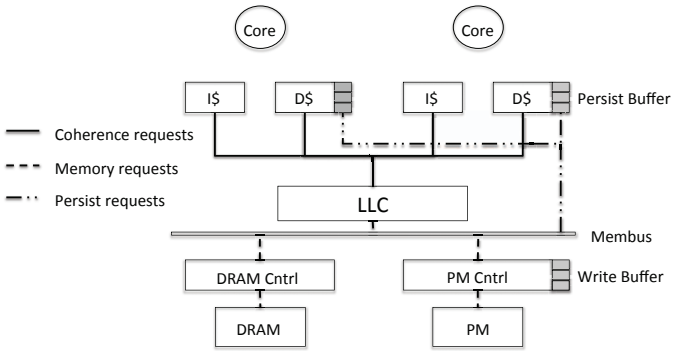

Fig. 2: Our system architecture implementing delegated ordering for RCBSP, with a persist buffer at the L1 D-cache for every core and write queue at the PM controller.

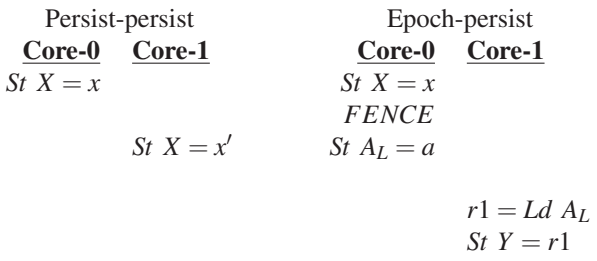

Fig. 3: Dependency examples.

persist traffic may be multiplexed on existing interconnects, much like uncacheable memory requests or non-temporal store operations [28]). The persist buffers drain persists into totally ordered epochs. The PM controller may freely schedule within an epoch, but not across epoch boundaries.

We describe delegated ordering assuming a snooping protocol for cache coherence and to drain persists. As this design is already quite complex, we leave generalization to non-snooping protocols to future work.

The persist buffer bears structural similarity to a write queue in a write-through cache (but buffers data at cache block rather than word granularity). It supports associative lookup by block address to facilitate coalescing and interaction with the coherence protocol. A persist buffer is quite small; as we will show, eight entries at most four of which may contain *FENCE* operations is sufficient.

A new persist request is appended to the persist buffer every time a store to a persistent address or a FENCE completes at an L1 D-cache. Upon completion of the store, the entire cache block is copied into the persist buffer entry (and later drained to the PM as a persist request). The *FENCE* entries divide the persist requests from the corresponding thread into epochs.

Persist buffer entries drain to the PM write queue when both intra- and inter-thread persist dependencies (governed by the PMO) have been resolved. When a *FENCE* drains, it creates an epoch separator in the PM write queue, across which persists may not be reordered. Epochs from different persist buffers that are unordered with respect to one another join a single epoch at the PM controller's write queue.

Persist buffers decouple volatile execution from persist operations, unlike synchronous ordering. Further, they also absolve caches of the responsibility to persist data to the PM. Caches may continue to hold and transfer ownership of data that is buffered in persist buffers. However, when a cache block in persistent memory is evicted from the LLC, it is silently dropped (persist buffers ensure updates are not lost and service subsequent reads).

C. Enforcing Dependencies

Persist buffers collaborate to jointly drain persists to the PM write queue, constructing unified epochs that are consistent with the persistency model ordering constraints at each core. We first describe at a high level how we ensure a correct drain order with reference to the example code sequences in Figure 3. We defer details to Section IV-E.

Intra-thread dependencies are enforced by draining persists from a persist buffer in order. It is important to note that even though persists are drained in order, they may still coalesce/reorder at the PM write queue, as long as no intervening *FENCE*s have been drained (by any thread). Additionally, adopting this simple drain policy allows us to obey fence cumulativity dependencies without having to employ complex dependency tracking mechanisms to accurately enforce dependencies. Overall, our in-order drain policy trades off some reordering/coalescing opportunities for a simpler design. Inter-thread dependencies are communicated among the persist buffers by leveraging existing coherence traffic. Dependencies can arise between individual persists to the same address, due to conflicting accesses, or between epochs, due to *FENCE* operations becoming transitively ordered by conflicting accesses.

Figure 3 (left) illustrates a dependence between two persists. Persist-persist dependencies arise when two stores to the same persistent address are executed at two different cores. RCBSP mandates that the two stores persist in coherence order (via Eqs 5 and 7). At a high level, the dependency is discovered as part of the cache coherence transaction that transfers ownership of the cache block from Core-0 to Core-1. Core-0 will include in its write response an annotation with the ID of its persist, indicating that Core-1's persist must be ordered after it. This annotation will prevent the persist from draining from Core-1's persist buffer. When Core-0 drains its persist, Core-1 will observe the drain and resolve the dependency (clear the annotation), allowing its persist to then drain.

Figure 3 (right) illustrates a code sequence creating a dependence between an epoch and a persist. An epoch-persist dependency arises when an epoch and a persist on different threads are ordered due to intervening conflicting accesses (here via accesses to A_L) and fence cumulativity. Due to the *FENCE* instruction on Core-0, we have $S^0_X \in G_A$ and $S^0_{A_L} \in G_B$. Since $L^1_{A_L}$ reads from $S^0_{A_L}$, we have $L^1_{A_L} \in G_B$. Further, since S^1_Y is data dependent on $L^1_{A_L}$ (via register r1), we have $S^1_Y \in G_B$. Since S^0_X and S^1_Y are in G_A and G_B respectively, via fence cumulativity (Eq 6)) and strict persistency(Eq 7) we have $S^0_X \leq_p S^1_Y$.

In this scenario, our design ensures that the persist buffer

entry corresponding to store Y on Core-1 is drained only after the persist buffer entry corresponding to the $FENCE$ instruction on Core-0, which in turn ensures that persists to X and Y are drained to PM in order.

At a high level, the ordering between the $FENCE$ operation and the store (S_Y) is again discovered as a consequence of the coherence transaction on the conflicting address A_L. When Core-0 receives a Read-Exclusive request for A_L, it discovers there is a preceding, undrained $FENCE$. Its reply includes an annotation indicating an ordering requirement against its $FENCE$. When Core-1 receives this annotation, it records the persist ordering dependence and will enforce it against the *next* persist/$FENCE$ it encounters, which in this case is the persist buffer entry of the store to Y.

A particular challenge of this mechanism is that ordering relationships between epochs ($FENCE$ operations) and persists can arise due to conflicting accesses to *volatile* as well as persistent addresses. In ARMv7, causal ordering between two $FENCE$ operations or $FENCE$-persist operations is established by *any* conflicting access pair. Therefore, the persist buffers must detect and honor ordering constraints established through volatile memory accesses. Indeed, in the example, we label the conflicting address A_L as it represents a lock or other synchronization variable, which likely resides in volatile memory.

To detect all conflicting accesses that follow a $FENCE$, the persist buffer must keep a record of all addresses read or written by its core until either the $FENCE$ drains or the processor executes a subsequent $FENCE$. Incoming coherence requests must be checked against the read- and write-sets to detect conflicts and discover dependencies. This requirement is similar to the read- and write-set tracking required to implement transactional memory [43]. As in many transactional memory designs, these sets may be maintained approximately, because false positives (identifying a conflict when there is none) introduce unnecessary persist ordering edges, but do not compromise correctness. However, given that the lifetime of a $FENCE$ in the persist buffer is much smaller than an entire transaction, a simple design would suffice. We enumerate the steps of this exchange in detail in Section IV-E.

Note that persist-epoch and epoch-epoch dependencies may also arise, and are enforced by the hardware structures described in Section IV-D. It is also important to note that by tracking dependencies at an individual persist or $FENCE$ granularity, our design does not suffer from the epoch dependency deadlocks identified in [13]. We omit examples in the interest of space.

D. Hardware Structures

Next, we describe the hardware structures required for delegated ordering. At each core, we provision a persist buffer, a pair of bloom filters for tracking read and write sets, and a register for tracking accumulated ordering dependences that must be applied to a yet-to-be-executed $FENCE$ or persist. Persist requests and $FENCE$ operations drain from persist buffers into the write queue at the PM controller. Figure 4 illustrates the hardware and fields in these structures.

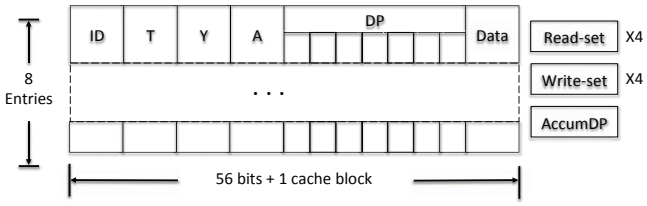

Fig. 4: HW structures at each core.

Persist Buffer. The persist buffer is the key structure that buffers pending persist requests while the core continues executing. Each persist buffer entry contains either a persist operation or a $FENCE$. We briefly describe each field:

- **T** - The "Type"; persist request or fence.
- **A** - The cache block "Address" of a persist request; supports associative search by address. For $FENCE$s, this field associates the entry with a read/write-set.
- **D** - The "Data" cache block to be persisted.
- **ID** - An "ID" that uniquely identifies each in-flight persist or $FENCE$, comprising the core id and entry index. These IDs are used to track and resolve dependencies across persist buffers. We denote IDs as "{Core index}:{Entry index}".
- **Y** - The "Youngest" bit, marks the youngest persist request to a particular address across all persist buffers. This bit is set when a persist request is appended to the buffer and reset upon an invalidation of the cache block, indicating a subsequent store by another core. When set, this bit indicates this persist buffer must service coherence requests for the address.
- **DP** - An array of inter-thread dependencies for this entry. The number of fields in each "DP" entry is one less than the number of persist buffers, tracking at most one dependency from each other core. An entry can be drained to PM only when all its dependencies have been resolved (drained to PM write queue). The dependencies are tracked via IDs; When an ID drains on the persist bus, matching "DP" fields are cleared.

Read/Write Sets & AccumDP. We provision pairs of bloom filters to track addresses accessed by the core after a $FENCE$, as described in Section IV-C. Each persist buffer also requires an additional dependence ("AccumDP") register that is not associated with any persist buffer entry. "AccumDP" tracks dependences that are discovered via cache coherence and must be applied as order constraints against the next persist/$FENCE$ issued by the core. When a persist or $FENCE$ is appended to the persist buffer, its "DP" field is initialized from "AccumDP" and "AccumDP" is cleared.

PM Write Queue. The PM Write Queue, like buffers in a conventional memory controller, holds writes until they are issued to the PM storage array. When a $FENCE$ operation

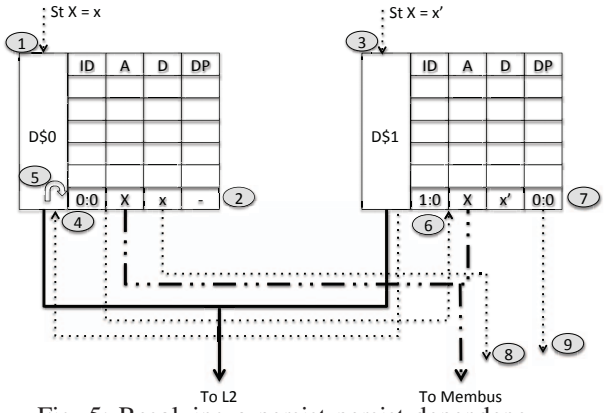

Fig. 5: Resolving a persist-persist dependency.

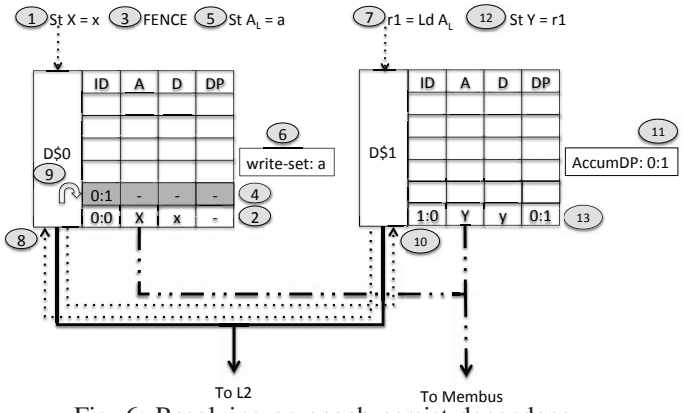

Fig. 6: Resolving an epoch-persist dependency.

is drained from a persist buffer, it creates an epoch boundary across which persists may not be reordered.

Overheads. The storage overhead for each persist buffer entry is 72B. Considering the short duration a *FENCE* spends in the persist buffer (due to the aggressive draining employed at the persist buffer), we use 32B bloom filters. An AccumDP register of 64B stores dependencies from all other persist buffers. In all, each persist buffer requires 8 persist buffer entries, 8 bloom filters for read/write sets (a maximum of 4 active *FENCE* entries are allowed in a persist buffer) and one AccumDP register, placing the storage overhead at 896B/core. It is important to note that if a persist buffer becomes full (either due to exhaustion of entries or *FENCE* slots), the corresponding L1-D$ is blocked until an entry drains from the persist buffer. The results presented in Section V account for all such blockages.

E. Detailed Examples.

We next walk through detailed examples that illustrate how persist buffers track inter-thread dependencies, with the aid of Figures 5 and 6

Persist-persist dependency. Figure 5 depicts the evolution of the persist buffer state for a persist-persist dependency (see Figure 3).

(1) D$0 receives a store request to a persistent address X. For simplicity, assume a cache hit at D$0. **(2)** A new value x is written to the cache for address X, and a persist request for X is appended to the persist buffer at D$0 with ID "0:0". Its address is set to X, the cache block data (x) is copied into the buffer, and the Y (Youngest) bit is set. Assume that there were no earlier dependencies for the store to X, so DP is cleared. **(3)** D$1 receives a store request to address X. **(4)** D$1 sends a read-exclusive request to D$0. **(5)** D$0 receives the read-exclusive request and snoops both the cache and the persist buffer. In the persist buffer, it finds a match with the Y bit set. It copies the value x into the response, invalidates the line in the cache, and clears the Y bit in the persist buffer. The coherence reply includes an annotation with ID "0:0" as a dependence. **(6)** D$1 receives the response with the latest data for X and the dependence annotation. **(7)** D$1 completes the store, creates a new persist request in its persist buffer, marking ID "0:0" as a dependency to its persist "1:0". **(8)** Persist buffer at D$0 entry "0:0" has no dependencies and is thus eligible to drain. It now does so, broadcasting its drain request to all persist buffers and the PM controller. **(9)** Persist buffer at D$1 observes that persist "0:0" has drained, resolves the dependency for persist "1:0" and subsequently drains it.

Epoch-Persist dependency. Figure 6 depicts the evolution of the persist buffer state an epoch-persist dependency.

(1) D$0 receives a store request to a persistent address X. Assume that it hits at D$0. **(2)** A new persist request is created for X with ID "0:0". Assume no dependencies. **(3)** D$0 receives a *FENCE* request. **(4)** A new entry is created for the *FENCE* with ID "0:1". (Gray entries indicate a *FENCE*). **(5)** D$0 receives a store request to a volatile address A_L. Assume it hits at D$0. **(6)** The volatile address A_L is recorded in the write-set associated with the *FENCE*. **(7)** D$1 receives a load request for address A_L. **(8)** D$1 sends a read request for address A_L to D$0. **(9)** D$0 snoops its cache and persist buffer, locating its cached copy of A_L. Since it has a pending *FENCE*, it compares address A_L to the write-set of the *FENCE* and discovers a match, indicating a persist order dependence. The coherence response is annotated to indicate a *FENCE* with ID "0:1" as a dependence. **(10)** D$1 receives the response with the latest data for A_L and the persist dependency annotation. **(11)** D$1 updates its "AccumDP" register to store the dependence on "0:1". This dependence will be applied to the next persist/*FENCE* instruction executed at D$1. **(12)** D$1 receives a store request to a persistent address Y. Assume it results in a cache hit at D$1. **(13)** A new persist request is created with ID "1:0". The dependence on "0:1" from the "AccumDP" register is recorded and the register is cleared.

The persist at D$1 with ID "1:0" will not be permitted to drain until D$0 broadcasts the drain of *FENCE* "0:1", ensuring that the persists to X and Y fall into successive epochs at the PM controller.

We note that our hardware might be substantially simplified

under a programming model where conflicting accesses must be explicitly annotated as synchronization accesses, such as the DRF0 model [30]. In such models, only synchronization accesses may create ordering relationships between epochs in properly labeled programs. Unfortunately, ARMv7 does not mandate that racing accesses be annotated, requiring the additional complexity of the read- and write-set tracking.

F. Coalescing Persists

One of the aims of our design is to enable persist operations to coalesce, where allowed by the persistency model, to improve performance and reduce the total number of PM writes. Coalescing may occur at two points. First, an incoming persist may coalesce with the most recent persist in the persist buffer if: (1) they are to the same cache block, (2) "accumDP" is empty and (3) the "Youngest" bit is still set. The implications of fence cumulativity require these restrictions. Sophisticated schemes may enable more coalescing, but would require complex tracking to ensure all persist dependencies are properly enforced. Second, persists may coalesce in the PM write queue, even if issued by different cores, provided they do not cross an epoch boundary.

In our design, we drain persist operations eagerly at both the persist buffer and PM write queue, as soon as ordering constraints allow. However, in the absence of a *FENCE*, it may be advantageous to delay persist operations in an attempt to coalesce more persists. The PM-write-intensive benchmarks we study do not afford additional coalescing opportunity, so we leave such optimizations to future work.

V. Evaluation

Core	8-cores, 2GHz OoO 6-wide Dispatch, 8-wide Commit 40-entry ROB 16/16-entry Load/Store Queue
I-Cache	32kB, 4-way, 64B 1ns cycle hit latency, 2 MSHRs
D-Cache	64kB, 4-way, 64B 2ns hit latency, 6 MSHRs
L2-Cache	8MB, 16-way, 64B 16ns hit latency, 16 MSHRs
Memory controller (DRAM, PM)	64/32-entry write/read queue
DRAM	DDR3, 800MHz
PCM	533MhZ, timing from [44]

TABLE II: Simulator Configuration.

We compare delegated ordering against synchronous ordering for three different memory designs: DRAM, PCM, and PWQ (described in Section II-B. We model DDR3 DRAM operating at 800MHz and PCM using timing parameters derived from [44] operating at 533MHz. We use PCM memory assumptions for PWQ. The PWQ design isolates the effect of fences and ordering instructions from PM access latency, while DRAM and PCM provide plausible performance projections. We model an 8-core system with ARM A15 cores in gem5 [26] using the configuration details in Table II. RCBSP uses an 8-entry persist buffer at each core, allowing at most four in-flight *FENCE*s.

It is important to note that gem5 implements a conservative multi-copy atomic version of ARMv7 consistency. Whereas ARMv7 allows non-store-atomic systems, there is reason to believe that, in practice, multi-copy atomicity *may* be provided (for example, Tegra 3 forbids some litmus test outcomes normally observed for non-store-atomic systems [34]). Nevertheless, we have presented an RCBSP design that is also correct for non-store-atomic systems.

Benchmark	Description	CKC
Conc. queue	Insert/Delete entries in a queue	1.2
Array Swaps	Random swaps of array elements	7.1
TATP	Update location transaction in TATP [45]	4.5
RB Tree	Insert/Delete entries in a Red-Black tree	0.1
TPCC	New Order transaction in TPCC [46]	0.8

TABLE III: Benchmarks. CKC = `clwb`s per 1000 cycles

Benchmarks: We study a suite of five PM-centric multi-threaded benchmarks, described in Table III. Our Concurrent Queue is similar to that of Pelley [4]. The Array Swaps and RB Tree are similar to those in NV-Heaps [10]. Our TATP [45] and TPCC [46] benchmarks execute the "update location" and "new order" transactions, respectively the benchmark's most write-intensive transactions. We select these benchmarks specifically because they stress PM write performance; larger applications may amortize slowdown of PM-write-intensive phases over periods of volatile execution. As a heuristic for the "write-intensive"ness of the benchmarks, we report the number of `clwb`s issued per 1000 cycles per core (CKC) in Table III. Array Swaps is our most write-intensive benchmark while RB Tree is the least, so we expect them to show the most and least sensitivity to persistency models, respectively.

A. Performance Comparison

Figure 7 contrasts the performance of RCBSP with SO, for three different memory designs: PWQ, DRAM, and PCM. Execution times in the figure are normalized to volatile execution with the corresponding memory design. The main takeaways from the figure are:

RCBSP outperforms SO: RCBSP consistently outperforms SO in nearly all cases. On average, RCBSP reduces the cost of persistence from 1.54× to 1.21×, 2.97× to 1.18×, and 7.21× to 1.93× for PWQ, DRAM, and PCM, respectively. It is particularly noteworthy that RCBSP outperforms SO even with PWQ for all workloads; even though PWQ hides the entire write latency of the memory device, the `clwb` latency exposed by SO still incurs noticeable delays that are hidden by RCBSP.

SO sensitivity to memory design: The performance of SO gets progressively worse for PWQ, DRAM, and PCM for each workload. This behavior is to be expected, as PWQ, DRAM, and PCM expose increasing memory access latencies on the critical path.

RCBSP sensitivity to memory design: The performance overheads of RCBSP are similar for PWQ and DRAM, and, as expected PCM is a distant third. Similar performance with

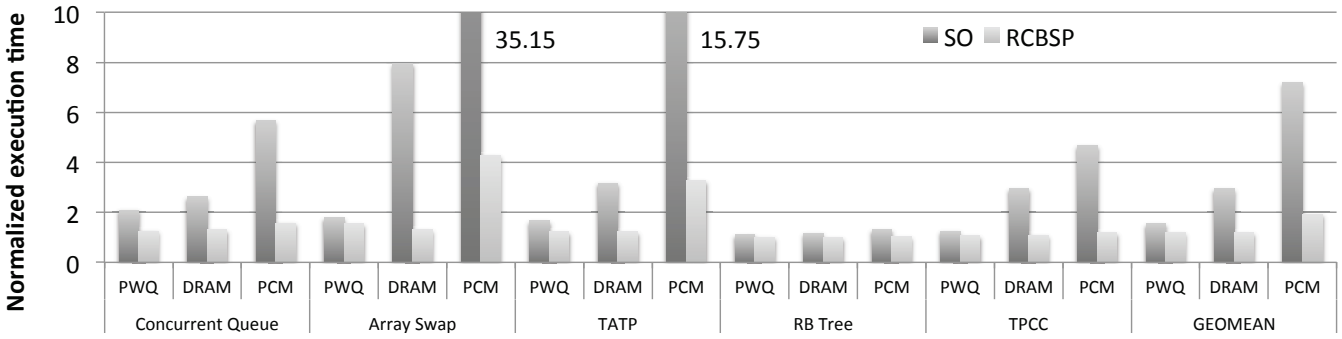

Fig. 7: Normalized execution time for PWQ, DRAM, and PCM.

PWQ and DRAM is due to two competing factors: (1) The main memory technology in PWQ is PCM, which is slower than DRAM—advantage DRAM. (2) PWQ has fewer ordering constraints on persists draining from the memory controller write queue—advantage PWQ. Moreover, we employ a 64-entry write queue at the memory controller, which hides most of the memory access latency for DRAM.

Least affected: For both SO and RCBSP, as expected, RB Tree incurs the least cost of persistence.

Most affected: Under SO, Array Swaps incurs the highest cost of persistence for DRAM and PCM, while Concurrent Queue is the most affected for PWQ. Under RCBSP, Array Swaps is the most affected for PWQ and PCM, Concurrent Queue suffers the highest slowdowns for DRAM. Array swaps is our most write-intensive benchmarks and its high costs of persistence are to be expected. It is interesting to observe that Concurrent Queue is most-affected in certain scenarios, because it exhibits the least thread concurrency among our benchmarks. Threads frequently contend for the enqueue and dequeue locks, causing `clwb` latencies to be exposed on the critical path of lock handoffs for SO under PWQ, which is then reflected in the overall execution time. For RCBSP, the high thread contention results in severely constrained drain order of persists at the PM controller, which increases overall execution time.

Persist buffer configuration: We studied the performance of RCBSP with different persist buffer configurations, varying the size of the buffer and the number of supported active *FENCE*s. We found that an 8-entry buffer supporting four simultaneously active *FENCE*s provided the best trade-off between hardware overhead and performance.

PM wear out: PMs like PCM suffer from wear out due to writes. RCBSP increases the overall writes to PM by 30%, averaged over all the benchmarks. This increase is to be expected as RCBSP is a hardware mechanism that tries to aggressively move persists from persist buffers to the PM write queue, while SO has the advantage of programmer inserted `clwb` instructions as triggers for writebacks, allowing better coalescing. Nevertheless, effective wear-leveling schemes have been proposed [47], [48], [49] to mitigate wear out and those solutions are orthogonal to persistency models and may be deployed with RCBSP.

Overall, RCBSP outperforms SO by 1.28× (PWQ), 2.58× (DRAM), and 3.73× (PCM) on average and by up to 8.23×.

VI. Related work

We briefly discuss related hardware designs that seek to facilitate the adoption of PM in future systems. We broadly classify works into five categories based on the write-ordering guarantees they provide.

No ordering: Apart from durability, cost, scalability, and energy efficiency may make PMs an attractive alternative to DRAM. Some hardware designs focus on PM only as a scalable replacement for DRAM [29], [49] and don't seek to use PM's non-volatility. Deploying PM as a volatile memory alternative requires addressing media-specific issues, such as wear-leveling [47], [48], [49], slow writes [50], [51], [52], and resistance drift [53]. These techniques are essential and orthogonal to our use PM.

Persistent caches: By making the caches themselves persistent, some proposals ensure that stores become durable as they execute, obviating the need for a persistency model. Cache persistence can be achieved by building cache arrays from non-volatile devices [16], [5], by ensuring that a battery backup is available to flush the contents of caches to PM upon power failure [17], [18], or by not caching PM accesses [16]. However, integrating NV devices in high performance logic poses manufacturing challenges, present NV access latencies (e.g., for STT-RAM) are more suitable for the LLC than all cache levels [5], and it is not clear if efficient backup mechanisms are available for systems with large caches. Our approach assumes volatile caches.

Synchronous ordering: SO (see Section II) is our attempt to formalize the persistency model implied by Intel's recent ISA extensions [3]. Without these extensions, it may be impossible to ensure proper PM write order in some x86 systems [27]. Mnemosyne [6] and REWIND [54] use SO to provide transaction systems optimized for PM. Atlas [11], uses it to provide durability semantics for lock-based code. SCMFS [55] uses SO to provide a PM-optimized file system. SO provides few opportunities to overlap program execution and persist operations and Bhandari et al. [28] show that

write-through caching sometimes provides better performance. We propose delegated ordering to increase overlap between program execution and persist operations.

Epoch barriers: As proposed in BPFS [7], epoch barriers divide program execution into epochs in which stores may persist concurrently. Stores from different epochs must persist in order. BPFS [7] implements epoch barriers by tagging all cache blocks with the current epochID (incremented after every epoch barrier instruction) on every store, and modifying the cache replacement policy to write epochs back to PM in order in a lazy fashion. This approach allows for more overlap of program execution and persist operations (no need to stall at epoch barriers) than SO. However, BPFS is tightly coupled with cache management, restricting cache replacements and suffers from some other drawbacks of SO, such as discarding write permissions as epochs drain from the cache. Pelley et al. [4] propose a subtle variation of epoch barriers, and show the potential performance improvement due to a better handling of inter-thread persist dependencies. Joshi et al. [13] define efficient persist barriers to implement buffered epoch persistency. However, Joshi does not study persistency models with a detailed PM controller, which is a central theme of our work. Delegated ordering fully decouples cache management from the path persistent writes take to memory and requires no changes to the cache replacement policy.

Other: Kiln [5] and LOC [56] provide a storage transaction interface (providing Atomicity, Consistency and Durability) to PM, wherein the programmer must ensure isolation. Kiln [5] employs non-volatile LLCs and leverages the inherent versioning of data in the caches and main memory to gain performance. LOC [56] reduces intra- and inter-transaction dependencies using a combination of custom hardware logging mechanisms and multi-versioning caches. Pelley [4] explores several persistency models, which range from conservative (strict persistency) to very relaxed (strand persistency) and shows the potential performance advantages of exposing additional persist concurrency to the PM controller. However, Pelley does not propose hardware implementations for the persistency models. FIRM [8] and NVM-Duet [42] optimize memory scheduling algorithms to manage resource allocation at the memory controller to optimize for performance and application fairness while respecting the constraints on the order of persists to PM.

VII. Conclusions and future work

Future systems will provide a load-store interface to PM, allowing persistent data structures in memory. Recoverability of these data structures relies on the programmer's ability to order PM updates, leading industry and academia to propose programming interfaces to prescribe ordering.

We show that synchronous ordering (based on Intel's recent ISA extensions for PM) incurs 7.21× slowdown on average over volatile execution for write-intensive benchmarks. SO conflates enforcing order and flushing writes to PM, incurring frequent stalls and poor performance. We show that forward progress can be effectively decoupled from PM write ordering by delegating ordering requirements explicitly to the PM. Our approach outperforms SO by 3.73× on average.

While our RCBSP implementation provides substantial performance improvement over SO, our design, as presented, is limited to systems which employ snoop-based coherence and have a single PM controller. We are looking into extending our design to work with systems with directory coherence and multiple PM controllers.

VIII. Acknowledgements

We would like to thank the anonymous reviewers for their valuable feedback. This work was supported by the National Science Foundation under the award NSF-CCF-1525372.

References

[1] Intel and Micron, "Intel and micron produce breakthrough memory technology," 2015, http://newsroom.intel.com/community/intel_newsroom/blog/2015/07/28/intel-and-micron-produce-breakthrough-memory-technology.

[2] C. World, "Hp and sandisk partner to bring storage-class memory to market," 2015, http://www.computerworld.com/article/2990809/data-storage-solutions/hp-sandisk-partner-to-bring-storage-class-memory-to-market.html.

[3] Intel, "Intel architecture instruction set extensions programming reference (319433-022)," 2014, https://software.intel.com/sites/default/files/managed/0d/53/319433-022.pdf.

[4] S. Pelley, P. M. Chen, and T. F. Wenisch, "Memory persistency," in *Proceedings of the 41st International Symposium on Computer Architecture*, 2014.

[5] 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 46th International Symposium on Microarchitecure*, 2013.

[6] H. Volos, A. J. Tack, and M. M. S. E, "Mnemosyne: Leightweight persistent memory," in *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2011.

[7] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee, "Better i/o through byte-addressable, persistent memory," in *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, 2009.

[8] J. Zhao, O. Mutlu, and Y. Xie, "Firm: Fair and high-performance memory control for peristent memory systems," in *Proceedings of 47th International Symposium on Microarchitecure*, 2014.

[9] V. Chidambaram, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Optimistic crash consistency," in *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, 2013.

[10] 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 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2011.

[11] D. R. Chakrabarti, H.-J. Boehm, and K. Bhandari, "Atlas: leveraging locks for non-volatile memory consistency," in *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2014.

[12] H.-J. Boehm and D. R. Chakrabarti, "Persistence programming models for non-volatile memory," Hewlett-Packard, Tech. Rep. HPL-2015-59, 2015.

[13] A. Joshi, V. Nagarajan, M. Cintra, and S. Viglas, "Efficient persist barriers for multicores," in *Proceedings of the international symposium on Microarchitecture*, 2015.

[14] A. Kolli, S. Pelley, A. Saidi, P. M. Chen, and T. F. Wenisch, "High-performance transactions for persistent memories," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, 2016.

[15] J. Izraelevitz, T. Kelly, and A. Kolli, "Failure-atomic persistent memory updates via justdo logging," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, 2016.

[16] T. Wang and R. Johnson, "Scalable logging through emerging non-volatile memory," *Proceedings of the VLDB Endowment*, vol. 7, no. 10, pp. 865–876, June 2014.

[17] D. Narayanan and O. Hodson, "Whole-system persistence," in *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2012.

[18] F. Nawab, D. Chakrabarti, T. Kelly, and C. B. M. III, "Procrastination beats prevention: Timely sufficient persistence for efficient crash resilience," Hewlett-Packard, Tech. Rep. HPL-2014-70, December 2014.

[19] 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.

[20] 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*, 2009.

[21] 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*, 2007.

[22] 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*, 2007.

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

[24] 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*, 1997.

[25] 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*, 1991.

[26] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011.

[27] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson, "System software for persistent memory," in *Proceedings of the 9th European Conference on Computer Systems*, 2014.

[28] K. Bhandari, D. R. Chakrabarti, and H.-J. Boehm, "Implications of cpu caching on byte-addressable non-volatile memory programming," Hewlett-Packard, Tech. Rep. HPL-2012-236, December 2012.

[29] 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*, 2009.

[30] S. V. Adve and K. Gharachorloo, "Shared memory consistency models: A tutorial," *IEEE Computer*, vol. 29, no. 12, pp. 66–76, December 1996.

[31] ARM, "Armv8-a architecture evolution," 2016, https://community.arm.com/groups/processors/blog/2016/01/05/armv8-a-architecture-evolution.

[32] ——, *ARM Architecture Reference Manual*. ARM, 2007.

[33] A. Kolli, S. Pelley, A. Saidi, P. M. Chen, and T. F. Wenisch, "Persistency programming 101," 2015, http://nvmw.ucsd.edu/2015/assets/abstracts/33.

[34] M. Luc, S. Inria, Sarkar, and P. Sewell, "A tutorial introduction to the arm and power relaxed memory models," 2012.

[35] D. Lustig, C. Trippel, M. Pellauer, and M. Martonosi, "Armor: Defending against memory consistency model mismatches in heterogeneous architectures," in *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, 2015.

[36] S. Sarkar, P. Sewell, J. Alglave, L. Maranget, and D. Williams, "Understanding power multiprocessors," in *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2011.

[37] J. Alglave, L. Maranget, and M. Tautschnig, "Herding cats: Modelling, simulation, testing, and data-mining for weak memory," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2014.

[38] ARM, "Barrier litmus tests and cookbook," 2009, http://infocenter.arm.com/help/topic/com.arm.doc.genc007826/Barrier_Litmus_Tests_and_Cookbook_A08.pdf.

[39] R. Ausavarungnirun, K. K.-W. Chang, L. Subramanian, G. H. Loh, and O. Mutlu, "Staged memory scheduling: schieving high performance and scalability in heterogeneous systems," in *In Proceedings of the International Symposium on Computer Architecture*, 2012.

[40] Y. Kim, D. Han, O. MUtlu, and M. Harchol-Balter, "Atlas: A scalable and high-performance scheduling algorithm for multiple memory controllers," in *In Proceedings of the International Symposium on High Performance Computer Architecture*, 2010.

[41] Y. Kim, M. Papamichael, O. Mutlu, and M. Harchol-Balter, "Thread cluster memory scheduling: Exploiting differences in memory access behavior," in *In Proceedings of the International Symposium on Microarchitecture*, 2010.

[42] R.-S. Liu, D.-Y. Shen, C.-L. Yang, S.-C. Yu, and C.-Y. M. Wang, "Nvm duet: unified working memory and persistent store architecture," in *Proceedings of the international conference on Architectural Support for Programming Languages an Operating Systems*, 2014.

[43] T. Harris, J. Larus, and R. Rajwar, *Transactional memory*. Morgan & Claypool Publishers, 2010.

[44] C. Xu, D. Niu, N. Muralimanohar, R. Balasubramonian, T. Zhang, S. Yu, and Y. Xie, "Overcoming the challenges of crossbar resistive memory architectures," in *In Proceedings of the International Symposium on High Performance Computer Architecture*, 2015.

[45] S. Neuvonen, A. Wolski, M. Manner, and V. Raatikka, "Telecom application transaction processing benchmark," 2011, http://tatpbenchmark.sourceforge.net/.

[46] T. P. P. C. (TPC), "Tpc benchmark b," 2010, http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-c_v5-11.pdf.

[47] M. K. Qureshi, M. M. Franchescini, V. Srinivasan, L. A. Lastras, B. Abali, and J. Karidis, "Enhancing lifetime and security of pcm-based main memory with start-gap wear leveling," in *Proceedings of the International Symposium on Microarchitecture*, 2009.

[48] M. K. Qureshi, A. Seznec, L. A. Lastras, and M. M. Franchescini, "Practical and secure pcm systems by online detection of malicious write streams," in *Proceedings of the 17th International Symposium on High Performance Computer Architecture*, 2011.

[49] P. Zhou, B. Zhao, J. Yang, and Y. Zhang, "A durable and energy efficient main memory using phase change memory technology," in *Proceedings of the 36th International Symposium on Computer Architecture*, 2009.

[50] J. Yue and Y. Zhu, "Accelerating write by exploiting pcm asymmetries," in *Proceedings of the International Symposium on High Performance Computer Architecture*, 2013.

[51] S. Cho and H. Lee, "Flip-n-write: a simple deterministic technique to improve pram write performance, energy and endurance," in *Proceedings of the International Symposium on Microarchitecture*, 2009.

[52] A. Hay, K. Strauss, T. Sherwood, G. H. Loh, and D. Burger, "Preventing pcm banks from seizing too much power," in *Proceedings of the International Symposium on Microarchitecture*, 2011.

[53] M. Awasthi, M. Shevgoor, K. Sudan, B. Rajendran, and R. Balasubramonian, "Efficient scrub mechanisms for error-prone emerging memories," in *Proceedings of the International Symposium on High Performance Computer Architecture*, 2012.

[54] A. Chatzistergiou, M. Cintra, and S. D. Vaglis, "Rewind: Recovery write-ahead system for in-memory non-volatile data structures," *Proceedings of the VLDB Endowment*, vol. 8, no. 5, 2015.

[55] X. Wu and A. L. N. Reddy, "Scmfs: a file system for storage class memory," in *In Proceedings of the International Conference for High Performance Computing*, 2011.

[56] Y. Lu, J. Shu, L. Sun, and O. Mutlu, "Loose-ordering consistency for persistent memory," in *Proceedings of the 32nd IEEE International Conference on Computer Design*, 2014.