Relaxed Persist Ordering Using Strand Persistency

Vaibhav Gogte
University of Michigan, USA
vgogte@umich.edu

William Wang
Arm Research, UK
william.wang@arm.com

Stephan Diestelhorst
Xilinx DCG, UK[§]
stephand@xilinx.com

Peter M. Chen
University of Michigan, USA
pmchen@umich.edu

Satish Narayanasamy
University of Michigan, USA
nsatish@umich.edu

Thomas F. Wenisch
University of Michigan, USA
twenisch@umich.edu

***Abstract*—Emerging persistent memory (PM) technologies promise the performance of DRAM with the durability of Flash. Several language-level persistency models have emerged recently to aid programming recoverable data structures in PM. Unfortunately, these persistency models are built upon hardware primitives that impose stricter ordering constraints on PM operations than the persistency models require. Alternative solutions use fixed and inflexible hardware logging techniques to relax ordering constraints on PM operations, but do not readily apply to general synchronization primitives employed by language-level persistency models. Instead, we propose StrandWeaver, a hardware strand persistency model, to minimally constrain ordering on PM operations. StrandWeaver manages PM order within a *strand*, a logically independent sequence of operations within a thread. PM operations that lie on separate strands are unordered and may drain concurrently to PM. StrandWeaver implements primitives under strand persistency to allow programmers to improve concurrency and relax ordering constraints on updates as they drain to PM. Furthermore, we design mechanisms that map persistency semantics in high-level language persistency models to the primitives implemented by StrandWeaver. We demonstrate that StrandWeaver can enable greater concurrency of PM operations than existing ISA-level ordering mechanisms, improving performance by up to 1.97× (1.45× avg.).**

***Index Terms*—Persistent memories, memory persistency, strand persistency, failure atomicity, language memory models**

I. Introduction

Persistent memory (PM) technologies, such as Intel and Micron's 3D XPoint, are here [1]—cloud vendors have already started public offerings with support for Intel's Optane DC persistent memory [2, 3, 4]. PMs combine the byte-addressability of DRAM and durability of storage devices. Unlike traditional block-based storage devices, such as hard disks and SSDs, PMs can be accessed using a byte-addressable load-store interface, avoiding the expensive software layers required to access storage, and allowing for fine-grained PM manipulation.

Because PMs are durable, they retain data across failures, such as power interruptions and program crashes. Upon failure, the volatile program state in hardware caches, registers, and DRAM is lost. In contrast, PM retains its contents—a *recovery* process can inspect these contents, reconstruct required volatile state, and resume program execution [5, 6, 7, 8].

[§]Work done while at Arm Research, Cambridge, UK

Several *persistency models* have been proposed in the past to enable writing recoverable software, both in hardware [9, 10] and programming languages [11, 12, 13, 14, 15]. Like prior works [16, 17, 18], we refer to the act of completing a store operation to PM as a *persist*. Persistency models enable two key properties. First, they allow programmers to reason about the order in which persists are made [16, 18, 17, 19]. Similar to memory consistency models [20, 21, 22, 23, 24], which order visibility of shared memory writes, memory persistency models govern the order of persists to PM. Second, they enable failure atomicity for a set of persists. In case of failure, either all or none of the updates within a failure-atomic region are visible to recovery [25, 26, 27, 28].

Recent works [11, 12, 29, 13, 27, 26, 13, 30, 31] extend the memory models of high-level languages, such as C++ and Java, with persistency semantics. These language-level persistency models differ in the synchronization primitives that they employ to provide varying granularity of failure atomicity. These persistency models are still evolving and are fiercely debated in the community [26, 27, 29, 11, 12, 13, 32]. Specifically, ATLAS [11], Coupled-SFR [12, 30], and Decoupled-SFR [12, 30] employ general synchronization primitives in C++ to prescribe the ordering and failure atomicity of PM operations. Other works [29, 27, 26, 25] ensure failure atomicity at a granularity of transactions using software libraries [27, 26, 25] or high-level language extensions [29].

These language-level models rely on low-level hardware ISA primitives to order PM operations [9, 10]. For instance, Intel x86 systems employ the `CLWB` instruction to explicitly flush dirty cache lines to the point of persistence and the `SFENCE` instruction to order subsequent `CLWB`s and stores with prior `CLWB`s and stores [9]. Under Intel's persistency model, `SFENCE` enforces a bi-directional ordering constraint on subsequent persists and introduces high-latency stalls until prior `CLWB`s and stores complete; `SFENCE` imposes stricter ordering constraints than required by language-level models.

Prior research proposals relax ordering constraints by proposing relaxed persistency models [16, 33, 18] in hardware and/or building hardware logging mechanisms [34, 35, 36, 37] to ensure failure-atomic updates to PM. These works propose relaxed persistency models, such as epoch persistency [18, 28, 19], that implement *persist barriers* to divide regions of

code into *epochs*; they allow persist reordering within epochs and disallow persist reordering across epochs. Unfortunately, epoch persistency labels only consecutive persists that lie within the same epoch as concurrent. It fails to relax ordering constraints on persists that may be concurrent, but do not lie in the same epoch. In contrast, hardware logging mechanisms [34, 36, 35, 38] aim to provide efficient implementations for ensuring failure atomicity for PM updates in hardware. These works ensure failure atomicity for transactions by emitting logging code for PM updates transparent to the program. These mechanisms impose fine-grained ordering constraints (*e.g.*, between log and PM updates) on persists but propose fixed and inflexible hardware that fails to extend to a wide range of evolving language-level persistency models.

In this work, we propose StrandWeaver, which formally defines and implements the *strand persistency* model to minimally constrain ordering on persists to PM. The principles of the strand persistency model were proposed in earlier work [16], but no hardware implementation, ISA primitives, or software use cases have yet been reported. The strand persistency model defines the order in which persists may drain to PM. It decouples persist order from the write visibility order (defined by the memory consistency model)—memory operations can be made visible in shared memory without stalling for prior persists to drain to PM. To implement strand persistency, we introduce three new hardware ISA primitives to manage persist order. A *NewStrand* primitive initiates a new *strand*, a partially ordered sequence of PM operations within a logical thread—operations on separate strands are unordered and may persist concurrently. A *persist barrier* orders persists within a strand—persists separated by a persist barrier within a strand are ordered. Persist barriers do not order persists that lie on separate strands. A *JoinStrand* primitive ensures that persists issued on the previous strands complete before any subsequent persists can be issued.

StrandWeaver proposes hardware mechanisms to build the strand persistency model upon these primitives. StrandWeaver implements a *strand buffer unit* alongside the L1 cache that manages the order in which updates drain to PM. The strand buffer unit enables updates on different strands to persist concurrently to PM, while persists separated by persist barriers within a strand drain in order. Additionally, StrandWeaver implements a *persist queue* alongside the load-store queue to track ongoing strand persistency primitives. The persist queue guarantees persists separated by `JoinStrand` complete in order even when they lie on separate strands.

StrandWeaver decouples volatile and persist memory order and provides the opportunity to relax persist ordering even when the system implements a conservative consistency model (*e.g.*, TSO [39]). Unfortunately, programmers must reason about persist order at the abstraction of the ISA, making it burdensome and error-prone to program persistent data structures. To this end, we integrate the ISA primitives introduced by StrandWeaver into high-level language persistency models to enable programmer-friendly persistency semantics. We build a logging design that employs StrandWeaver's primitives to enforce only the minimal ordering constraints on persists required for correct recovery. We showcase the wide applicability of StrandWeaver primitives by integrating our logging with three prior language-level persistency models that provide failure-atomic transactions [29, 26, 27], synchronization-free regions [12, 30], and outermost critical sections [40, 11], respectively. These persistency models provide simpler primitives to program recoverable data structures in PM—programmer-transparent logging mechanisms layered on top of our StrandWeaver hardware hide low-level hardware ISA primitives and reduce the programmability burden.

In summary, we make the following contributions:

- We formally define primitives for strand persistency that enable relaxed persist order, decoupled from visibility of PM operations.
- We propose StrandWeaver, hardware mechanisms to implement the primitives defined by the strand persistency model. Specifically, we show how the strand buffer unit and persist queue can order and schedule persists concurrently.
- We build logging designs that rely on low-level hardware primitives proposed by StrandWeaver and integrate them with several language-level persistency models.
- We demonstrate how StrandWeaver relaxes persist order, resulting in a performance improvement of up to 1.97× (1.45× average) over Intel's persistency model [9] and up to 1.55× (1.20× average) over the state-of-the-art implementation [19] of the epoch persistency model.

II. Overview and Motivation

We now present an overview of memory persistency models.

A. Language-Level Persistency Models

Language-level persistency models define persistency semantics for high-level programming languages such as C++ and Java [11, 12, 15, 13, 30, 31, 41, 42]. They prescribe persist ordering constraints in PM and provide failure atomicity to ensure that either all or none of the updates are visible to recovery after failure. These persistency models rely on synchronization primitives in languages to provide persistency semantics. For instance, ATLAS [11] assures failure atomicity of outermost critical sections—code region bounded by lock and unlock operations in lock-based programs. It also ensures inter-thread persist order using synchronizing lock and unlock operations. Similarly, SFR-based persistency models [12, 30] assure failure atomicity for synchronization-free regions—regions of code delimited by low-level synchronization operations, such as acquire and release—and order persists using these synchronizing acquire and release operations. Other models [29, 26, 27, 43, 25] provide failure atomicity for transactions and order persists using external synchronization.

These persistency models build compiler frameworks or software libraries to map high-level semantics in languages to low-level hardware ISA primitives. Figure 1(a) shows example code for a failure-atomic region in ATLAS enclosed by lock and unlock operations. ATLAS instruments each store with

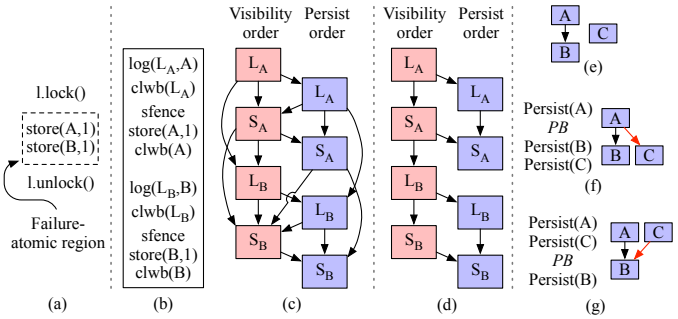

Fig. 1: (a) Example failure-atomic region in ATLAS bounded by lock and unlock operations, (b) ATLAS logging using Intel's ISA extensions for PM, (c) Visibility and persist ordering of logs and in-place updates on a TSO system, (d) Ideal persist ordering constraints sufficient for correct recovery, (e) Desired order on persists `A`, `B`, and `C`, (f) Persist barrier (PB) additionally orders persists `A` and `C`, (g) PB additionally orders persists `C` and `B`.

undo logging to assure failure atomicity. On failure, recovery inspects undo logs and rolls back partially executed failure-atomic regions. For correct recovery, logs need to persist before in-place updates are made to PM—ATLAS relies on low-level hardware ISA primitives to assure this ordering. Unfortunately, hardware imposes stricter ordering constraints than required by these persistency models for correct recovery.

B. ISA-Level Persistency Mechanisms

Modern hardware systems implement hardware structures to reorder, coalesce, elide, or buffer updates, which complicate ordering persists to PM [16, 44, 28]. For instance, write-back caches lazily drain dirty cache lines to memory—ordering visibility of stores to the write-back caches does not imply that the PM writes are ordered.

Similar to memory consistency models, which reason about visibility of memory operations, memory persistency models specify the order in which updates persist to PM. Pelley *et al.* [16] propose strict and relaxed persistency models to specify persist ordering. Strict persistency couples store visibility to the order in which they persist to PM—persist order follows the visibility order of PM operations. Unfortunately, strict persistency has a high performance overhead as it restricts persist concurrency especially under conservative consistency models, such as TSO, which strictly order visibility of stores.

Relaxed persistency models decouple persist order from the order in which memory operation become visible. Epoch persistency introduces *persist barriers* that divide program execution into *epochs*. Persists within epochs can be issued concurrently, while persists separated by a persist barrier are ordered to the PM. Several implementations [18, 9, 19, 28, 45], including Intel's x86 ISA, build epoch persistency models.

Intel's persistency model. Intel x86 systems employ `CLWB` (or `CLFLUSHOPT` in older systems) instructions [9] to explicitly flush dirty cache lines to an asynchronous data refresh (ADR)-supported PM controller [46]. In case of power failure, an ADR-supported PM controller flushes pending operations to PM. `SFENCE` acts as a persist barrier that orders any subsequent `CLWB`s with the preceding `CLWB`s. Additionally, `SFENCE` also orders visibility of subsequent stores after the preceding `CLWB`s complete to ensure stores do not drain from write-back caches to PM before prior `CLWB`s finish.

Limitations. Persist concurrency is limited by the size of epochs [18]. Although `CLWB`s within an epoch can flush data concurrently to PM, `SFENCE` enforces stricter ordering constraints on persists, which are not required for ensuring correct recovery. Any subsequent stores and `CLWB`s that are independent and can be issued concurrently to the PM are serialized by `SFENCE`. `SFENCE` causes long-latency stalls as it delays visibility of subsequent stores until prior `CLWB`s flush data to the PM controller.

Example. Figure 1(b) shows example undo logging code for the ATLAS's persistency model to ensure failure atomicity of updates to PM locations `A` and `B` on an Intel x86 system. Undo logging requires pairwise ordering of logs and a subsequent store—logs must persist before corresponding updates for correct recovery. Note that logs L_A and L_B (and similarly, updates to locations `A` and `B`) can persist to PM concurrently, as shown in the ideal persist ordering constraints in Figure 1(d). Unfortunately, `SFENCE` orders log creation and flush to L_A with log creation and flush to L_B. Under Intel's TSO consistency model [39], visibility of stores is ordered (visibility of L_A and S_A is ordered in Figure 1(c)). `SFENCE` additionally restricts visibility of subsequent stores until prior `CLWB`s complete—S_A is not issued until L_A persists. Thus, Intel's persistency model imposes stricter constraints on visibility and persist order that are not required for recovery by language-level models.

C. Limited Persist Concurrency

Epoch size limits persist concurrency, as language-level persistency implementations instrument each store within a failure-atomic region with a log operation followed by an `SFENCE`. Alternatively, log operations may be grouped within an epoch before PM updates are issued. Unfortunately, it is challenging to group persists, whether via static compiler mechanisms or in expertly handcrafted software libraries.

Compiler optimizations. The presence of ambiguous memory dependencies make it challenging for compilers to perform static analysis at compile time [47, 48, 49] to coalesce logging operations within failure-atomic regions. Even with ideal compiler mechanisms to group persists, epoch persistency fails to specify precise ordering constraints. Figure 1(e) shows the desired order on persists `A`, `B`, and `C`—persist `C` can be issued concurrent to persists `A` and `B`. The persist barrier precludes persist `C` from being concurrent when it is issued in either of the epochs as shown in Figure 1(f,g).

Handcrafted PM applications. Prior works [19, 50] characterize expertly-crafted PM applications that use Intel's PMDK [29] libraries to ensure failure-atomic transactions. WHISPER applications [19] build failure-atomic transactions that may contain up to a few hundred epochs. Wang *et al.* [50] studies open-source microbenchmarks based on PMDK libraries [29] that may also require >10 barriers per failure-atomic transaction to ensure correct logging. Unfortunately, these handcrafted libraries require barriers to order logs with

the PM updates for correct recovery and may not always coalesce logging operations and PM updates within separate epochs [19, 50]. As such, we require hardware primitives that may specify precise ordering constraints on PM operations.

III. Strand Persistency Model

We propose employing strand persistency model [16] to minimally constrain persists. Pelley *et al.* [16] proposes strand persistency model in principle, but does not specify the ISA primitives, hardware implementation and software use-cases. Strand persistency divides thread execution into *strands*. Strands constitute sets of PM operations that lie on the same logical thread. Ordering primitives enforce persist ordering within strands, but persists are not individually ordered across strands. We use the term "strand" to evoke the idea that a strand is a part of a logical thread, but has independent persist ordering. Strand persistency decouples the visibility and persist order of PM operations. The consistency model continues to order visibility of PM operations—PM operations on separate strands follow visibility order enforced by system's consistency model.

Strand primitives. Strand persistency employs three primitives to prescribe persist ordering: a *persist barrier* to enforce persist ordering among operations on a strand, *NewStrand* to initiate a new strand, and a *JoinStrand* to merge prior strands initiated on the logical thread. PM accesses on a thread separated by a persist barrier are ordered. Conversely, `NewStrand` removes ordering constraints on subsequent PM operations. `NewStrand` initiates a new strand—a strand behaves as a separate logical thread in a persist order. Persists on different strands can be issued concurrently to PM. Note that persist barriers, within a strand, continue to order persists on that strand. The hardware must guarantee that recovery software never observes a mis-ordering of two PM writes on the same strand that are separated by a persist barrier. Finally, `JoinStrand` merges strands that were initiated on the logical thread. It ensures the persists issued on the prior strands complete before any subsequent persists are issued.

In this work, we propose StrandWeaver to define ISA extensions and build the strand persistency model in hardware. Further, we provide techniques to map the persistency semantics offered by high-level languages to strand persistency.

A. Definitions

The strand persistency model specifies the order in which updates persist to PM. We formally define the persist order enforced under strand persistency using notation similar to prior works [17, 25].

- M_x^i: A load or store operation to PM location x on thread i
- S_x^i: A store operation to PM location x on thread i
- PB^i: A persist barrier issued by thread i
- NS^i: A `NewStrand` issued by thread i
- JS^i: A `JoinStrand` issued by thread i

Volatile memory order (VMO) defines the ordering relation on memory operations prescribed by the system's consistency model. Similarly, persist memory order (PMO) defines the ordering relation that describes the ordering of memory operations by the system's persistency model.

- $M_x^i \leq_v M_y^i$: M_x^i is ordered before M_y^i in VMO
- $M_x^i \leq_p M_y^i$: M_x^i is ordered before M_y^i in PMO

We now define the ordering constraints that are expressed by the primitives under strand persistency.

Intra-strand ordering. `NewStrand` operation initiates a new strand; subsequent memory operations will not have any ordering constraints in PMO to operations preceding the `NewStrand`. A persist barrier orders PM operations within a strand. Thus, two memory operations that are not separated by a `NewStrand` are ordered in PMO by a persist barrier.

$$\left(M_x^i \leq_v PB^i \leq_v M_y^i\right) \wedge \left(\nexists NS^i : M_x^i \leq_v NS^i \leq_v M_y^i\right) \rightarrow M_x^i \leq_p M_y^i \tag{1}$$

Additionally, `JoinStrand` introduces ordering constraints on PM operations to different memory locations that lie on separate strands. Note that a persist barrier does not order persists on different strands. `JoinStrand` orders persists initiated on prior strands with the persists on subsequent strands.

$$M_x^i \leq_v JS^i \leq_v M_y^i \rightarrow M_x^i \leq_p M_y^i \tag{2}$$

Thus, memory operations separated by `JoinStrand` are ordered in PMO.

Strong persist atomicity. Persists to the same or overlapping memory locations follow the order in which memory operations are visible (as governed by the consistency model of the system)—this property is called *strong persist atomicity* [16]. Similar to consistency models that ensure *store atomicity* by serializing memory operations to the same memory location through coherence mechanisms, strong persist atomicity serializes persists to the same memory location. We preserve strong persist atomicity to ensure that recovery does not observe side-effects due to reorderings that would not occur under fault-free execution of the program.

$$S_x^i \leq_v S_x^j \rightarrow S_x^i \leq_p S_x^j \tag{3}$$

Conflicting persists on different strands or logical threads that are ordered in VMO are also ordered in PMO.

Transitivity. Finally, persist order is transitive and irreflexive:

$$\left(M_x^i \leq_p M_y^j\right) \wedge \left(M_y^j \leq_p M_z^k\right) \rightarrow M_x^i \leq_p M_z^k \tag{4}$$

Persists on racing strands or threads (two or more strands or threads that consist of racing memory accesses) can occur in any order, unless ordered by Equations 2-4.

B. Persist Ordering

Figure 2 illustrates persist ordering under different scenarios due to strand persistency primitives.

Intra-strand persist concurrency. Figure 2(a) shows example code that employs `NewStrand` to issue persists concurrently on different strands, and a persist barrier to order persists

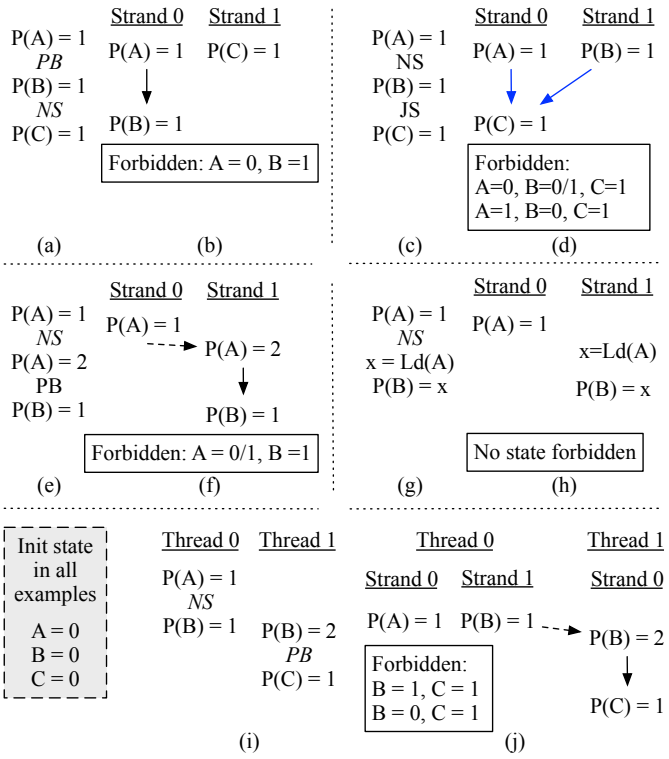

Fig. 2: Persist order due to StrandWeaver's primitives. Figure uses following notations for strand primitives: PB: persist barrier, NS: `NewStrand` and JS: `JoinStrand`. In each case, we also show the forbidden PM state. Black solid arrow, blue solid arrow, and black dotted arrow show order due to persist barrier, `JoinStrand`, and SPA, respectively. (a,b) Intra-strand ordering due to persist barrier, (c,d) Inter-strand ordering due to `JoinStrand`, (e,f) Persist order due to SPA, (g,h) Loads to the same PM location do not order persists, (i,j) Inter-thread persist ordering due to SPA.

within a strand. Persist barrier `PB` orders persist `A` before persist `B` (Equation 1) on strand `0` as shown in Figure 2(b). The `NewStrand` operation clears ordering constraints on following persists due to the previous persist barrier `PB` (Equation 1) and initiates a new strand `1`. Persist barrier `PB` does not order persists that lie on different strands. Persist `C` lies on strand `1`, and can be issued to PM concurrent to persists `A` and `B`.

Inter-strand persist ordering. Figure 2(c) shows example code that employs `JoinStrand` to order persists. `JoinStrand` merges strands `0` and `1` to ensure that persists `A` and `B` are ordered in PMO before persist `C` (as per Equation 2) as shown in Figure 2(c,d). Figure 2(d) shows forbidden PM state that requires persist `C` to reorder before persists `A` and `B` and so, is disallowed under strand persistency.

Inter-strand strong persist atomicity. Strong persist atomicity (SPA) governs the order of persists on different strands or threads to the same or overlapping memory locations (as per Equation 3). SPA orders persists as per their visibility enforced due to program order or cache coherence. Figure 2(e) shows an example of conflicting persists that occur on separate strands within a thread. Persist `A` on strand `0` is ordered before persist `A` on strand `1` as corresponding stores to the memory location `A` follow their program order [51]. Note that, persist `B` on strand `1` is ordered after persist `A` on strand `0` due to transitivity (as per Equation 4)—this relationship guarantees that recovery never observes the PM state shown in Figure 2(f).

Note that, a conflicting load to PM on another strand does not establish persist order in PMO (as per Equations 1 and 3). As shown in Figure 2(g), although load `A` is program-ordered after persist to `A`, persist `B` on strand `1` can be issued concurrently to PM. Although visibility of the memory operations is ordered, persists can be issued concurrently on the two strands—PM state (`A`=0, `B`=1) is not forbidden. Persist order due to SPA on separate strands can be established by having write-semantics for both memory operations to the same location (*e.g.* read-modify-write instead of loads). Alternatively, persist order across strands can be achieved using `JoinStrand` as shown in Figure 2(c,d).

Inter-thread strong persist atomicity. Similar to inter-strand order, SPA orders persists that occur on different logical threads. Figure 2(i) shows an example execution on two threads. On thread `0`, persists `A` and `B` lie on different strands and are concurrent, as shown in Figure 2(j). If a store to memory location `B` on thread `0` is ordered before that on thread `1`, order that is established through cache coherence, they are ordered in PMO. SPA orders persist `B` (and following persist `C` due to intervening persist barrier) on thread `1` after persist `B` on thread `0`. Conversely, if store `B` on thread `1` is ordered before store `B` on thread `0` in VMO (case not shown in Figure 2(j)), the forbidden state changes to (B=0, C=1).

Establishing inter-thread persist order. Persists on different strands or threads may occur in any order, unless ordered by Equations 2-4. Synchronization operations establish *happens-before* ordering relation between threads [52, 53], ordering visibility of memory operations, but do not enforce persist order. Persists can potentially reorder across the synchronizing lock and unlock operations. This reordering can be suppressed by placing a `JoinStrand` operation before unlock and after synchronizing lock operations. Synchronizing lock and unlock operations establish a happens-before ordering relation between threads, and `JoinStrand` operations prevent any persists from reordering across synchronizing operations. Note that locks may be persistent or volatile. If locks reside in PM, persists resulting from lock and unlock operations are ordered in PM due to SPA. Thus, recovery may observe correct lock state and reset it after failure [54].

IV. Hardware Implementation

We now describe hardware mechanisms that guarantee these persist orderings.

Microarchitecure. We implement StrandWeaver's persist barrier, `NewStrand`, and `JoinStrand` primitives as ISA extensions. A persist occurs due to a voluntary data flush from volatile caches to PM using a `CLWB` operation, or a writeback resulting from cacheline replacement. We use `CLWB`, which is *issued* to write-back caches by the CPU, to flush dirty cache lines to the PM controller. Note that `CLWB` is a non-invalidating operation—it retains a clean copy of data in caches. A `CLWB` *completes* when the CPU receives an acknowledgement of its receipt from the PM controller.

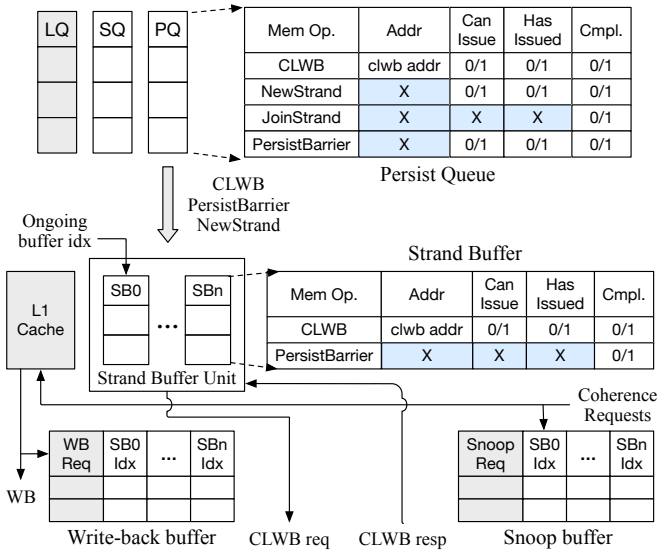

Fig. 3: StrandWeaver architecture. Persist queue and strand buffer unit implement persist ordering due to primitives in strand persistency model.

Architecture overview. Figure 3 shows the high-level architecture of StrandWeaver. The persist queue and strand buffer unit jointly enforce persist ordering. The persist queue, implemented alongside the load-store queue (LSQ), ensures that `CLWB`s and stores separated by a persist barrier within a strand are *issued* to the L1 cache in order, and `CLWB`s separated by `JoinStrand` *complete* in order. The strand buffer unit is primarily responsible for leveraging inter-strand persist concurrency to schedule `CLWB`s to PM. It resides adjacent to the L1 cache and comprises an array of strand buffers that may issue `CLWB`s from different strands concurrently. Each strand buffer manages persist order within a strand and guarantees that persists separated by persist barriers within that strand complete in order. The strand buffer unit also coordinates with the L1 cache to ensure that persists due to cache writebacks are ordered as per PMO. It also tracks cache coherence messages to ensure that inter-thread persist dependencies are preserved.

Persist queue architecture. Figure 3 shows the persist queue architecture and operations appended to it by the CPU pipeline. The persist queue manages entries that record ongoing `CLWB`s, persist barriers, `NewStrand`, and `JoinStrand` operations. Its architecture resembles that of a store queue—it supports associative lookup by address to identify dependencies between ongoing stores and `CLWB`s. Figure 3 also shows `Addr`, `CanIssue`, `HasIssued`, and `Completed` fields per entry in the persist queue. The `Addr` field records the memory address for an incoming `CLWB` operation that needs to be flushed from caches to the PM. The `CanIssue` field is set when an operation's persist dependencies resolve and the operation is ready to be issued to the strand buffer unit. `CLWB`s, persist barriers, and `NewStrand` are issued to the strand buffer unit when `CanIssue` is set; `HasIssued` is set as they are issued. The `Completed` field is set when the persist queue receives a completion acknowledgement for the operation. An operation can retire from the queue when `Completed` is set.

Persist queue operation. The persist queue tracks persist barriers to monitor intra-strand persist dependencies. On insertion, a persist barrier imposes a dependency so that `CLWB`s and stores are ordered within its strand. It orders issue of prior stores before subsequent `CLWB`s, and prior `CLWB`s before subsequent stores. These constraints ensure that stores do not violate persist order by updating the cache and draining to PM via a cache writeback before preceding `CLWB`s. The persist queue also coordinates with the store queue to ensure that younger `CLWB`s are issued to the strand buffer unit only after elder store operations to the same memory location. On `CLWB` insertion, the persist queue performs a lookup in the store queue to identify elder stores to the same location. This lookup is similar to that performed by the load queue for load-to-store forwarding [55, 56].

`CLWB`s, persist barriers, and `NewStrand` operations in the persist queue are issued to the strand buffer unit in order. Note that, unlike Intel's persistency model, which stalls stores separated by `SFENCE` until prior `CLWB`s *complete* (as described in Section II-B), persist barriers stall subsequent stores only until prior `CLWB`s have *issued*. `JoinStrand` ensures that `CLWB`s and stores issued on prior strands complete before any subsequent `CLWB`s and stores can be issued. Unlike a persist barrier, `JoinStrand` stalls issue of subsequent `CLWB`s and stores until prior `CLWB`s and stores *complete*. On `JoinStrand` insertion, the persist queue coordinates with the store queue to ensure that subsequent stores are not issued until prior `CLWB`s complete. As `JoinStrand` is not issued to the strand buffer unit, its `CanIssue` and `HasIssued` fields are not used.

`CLWB`s, persist barriers, and `NewStrand` operations complete when the persist queue receives a completion acknowledgement from the strand buffer unit. `JoinStrand` completes when prior `CLWB`s, persist barrier, and `NewStrand` are complete and removed from the persist queue, and prior stores are complete and removed from the store queue.

Strand buffer unit architecture. The strand buffer unit coordinates with the L1 cache to guarantee `CLWB`s and cache writebacks drain to PM and complete in the order specified by PMO. It maintains an array of strand buffers—each strand buffer manages persist ordering within one strand. `CLWB`s that lie in different strand buffers can be issued concurrently to PM. Strand buffers manage ongoing `CLWB`s and persist barriers and record their state in fields similar to the persist queue. The `CanIssue` and `HasIssued` fields mark when a `CLWB` is ready to issue and has issued to PM, respectively. The strand buffer retires entries in order when operations complete.

Strand buffer unit operation. The strand buffer unit receives `CLWB`, persist barriers, and `NewStrand` operations from the persist queue. In the strand buffer unit, the ongoing buffer index points to the strand buffer to which an incoming `CLWB` or persist barrier is appended. This index is updated when the strand buffer unit receives a `NewStrand` operation indicating the beginning of a new strand. Subsequent `CLWB`s and persist barriers are then assigned to the next strand buffer. StrandWeaver assigns strand buffers upon `NewStrand` operations in a round-robin fashion. The strand buffer unit

acknowledges completion of NewStrand operations to the persist queue when it updates the current buffer index.

Each strand buffer manages intra-strand persist order arising from persist barriers. It orders completion of prior CLWBs before any subsequent CLWBs can be issued to PM. On insertion in a strand buffer, a persist barrier creates a dependency that orders any subsequent CLWBs appended to the buffer. A persist barrier completes when CLWBs ahead of it complete and retire from the strand buffer. On completion of a persist barrier, the strand buffer resolves dependencies for subsequent CLWBs and marks them ready to issue by setting CanIssue. When CLWBs are inserted, the strand buffer performs a lookup to identify any persist dependencies from incomplete persist barriers. If there are none, the strand buffer immediately sets CanIssue.

When its dependencies resolve (when CanIssue field is set), the strand buffer issues a CLWB—it performs an L1 cache lookup to determine if the cache line is dirty. If so, it flushes the dirty cache block to PM and retains a clean copy in the cache. Upon a miss, it issues the CLWB to lower-level caches. When the CLWB is performed, HasIssued is set.

The strand buffer receives an acknowledgement when a CLWB completes its flush operation. It marks the corresponding entry Completed and retires completed entries in order.

Managing cache writebacks. PM writes can also happen due to cache line writebacks from write-back caches. The persist queue does not stall visibility of stores following persist barriers until prior CLWBs complete—it only ensures that prior CLWBs are issued to the strand buffer unit before any subsequent stores are issued. Thus, stores might inadvertently drain from the cache before ongoing CLWBs in the strand buffer unit complete. StrandWeaver extends the write-back buffer, which manages in-progress writebacks from the L1 cache, with a field per strand buffer (as shown in Figure 3) that records the tail index of the buffer when the L1 cache initiates a writeback. The write-back buffer drains writebacks only after the strand buffers drain operations beyond these recorded indexes. This constraint guarantees that older CLWBs complete before subsequent writebacks are issued, and thus prevents any persist order violation. Note that, since CLWBs never stall in strand buffers to wait for writebacks, there is no possibility of circular dependency and deadlock in StrandWeaver.

Enabling inter-thread persist order. As explained earlier in Section III, strong persist atomicity establishes order on persists to the same memory location across different threads—persists follow the order in which stores become visible. As cache coherence determines the order in which stores become visible, we track incoming coherence requests to the L1 cache to establish persist order. If a cache line is dirty in the L1 cache, other cores might steal ownership and persist the cache line before ongoing CLWBs in the strand buffer complete (violating the required order shown in Figure 2(i,j)). Similar to the write-back buffer, we provision per-strand-buffer fields in the snoop buffers that track and respond to ongoing coherence requests. On an incoming read-exclusive coherence request, if the corresponding cache line is dirty, we record the tail index of the strand buffer in the snoop buffer. The read-exclusive request stalls until the strand buffers drain to the recorded index. This stall ensures that CLWBs that are in progress when the coherence request was received complete before the read-exclusive reply is sent. Again, there is no possibility of circular dependency/deadlock.

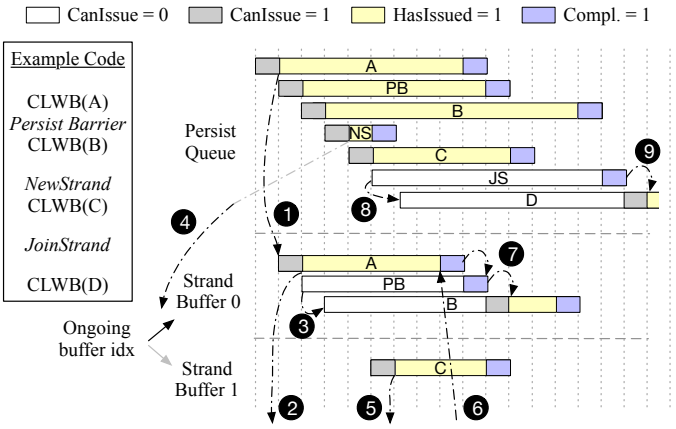

Fig. 4: Running example. Figure uses following notations. PB: persist barrier, NS: NewStrand, JS: JoinStrand.

PM controller. We do not modify the PM controller; we assume it supports ADR [46, 57] and so lies in the persistent domain. When the PM controller receives a CLWB, it returns an acknowledgement to the strand buffer unit.

A. Example

Figure 4 shows an example code with the desired order on persists prescribed by PMO. We show a step-by-step illustration of operations as they are executed by StrandWeaver. ❶ CLWB(A) is appended to an entry in the persist queue, and is issued to the strand buffer unit, as it encounters no earlier persist dependencies. Since the current buffer index is 0, CLWB(A) is added to strand buffer 0. ❷ CLWB(A) is issued and performs an L1 access to flush the dirty cache line. ❸ A persist barrier and CLWB(B) are appended to strand buffer 0; CLWB(B) stalls and waits for the preceding persist barrier (and CLWB(A)) to complete. ❹ NewStrand from the persist queue updates the ongoing buffer index in the strand buffer unit to 1. Consequently, subsequent CLWB(C) is appended to strand buffer 1. ❺ As CLWB(C) incurs no prior dependencies in its strand buffer 1 due to persist barriers, it issues to PM concurrent to CLWB(A). ❻ The strand buffer unit receives a completion for CLWB(A); the operation is complete. ❼ As CLWB(A) and the persist barrier complete, the ordering dependency of CLWB(B) is resolved, and it issues. ❽ JoinStrand stalls issue of CLWB(D) until prior CLWBs complete. ❾ When the persist queue receives a completion acknowledgement for CLWB(A), CLWB(B), and CLWB(C), JoinStrand completes and CLWB(D) is issued to the strand buffer unit.

V. Designing Language-level Persistency Models

The strand persistency model decouples persist order from the visibility order of memory operations—it provides opportunity to relax persist ordering even in the presence of conservative consistency models (*e.g.* TSO [39]). Unfortunately,

```
FA-Begin()          FA-End()          Store(A,val)
log_begin()         JoinStrand        log_store(L_A,A)
JoinStrand          log_end()         CLWB(L_A)
                                      Persist Barrier
                                      store(A,val)
                                      NewStrand
```

Fig. 5: Logging using strand primitives. Figure shows instrumentation for failure-atomic region begin and end, and PM store operation.

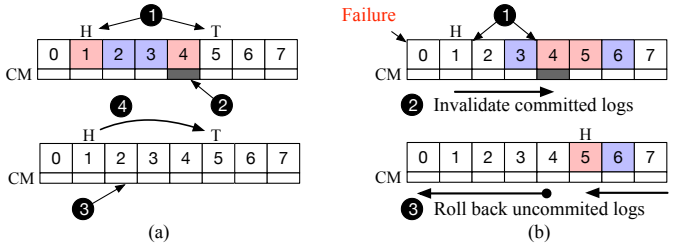

Fig. 6: Logging example. Entries for store operations are shown in blue and entries for synchronization operations are shown in red. `CM` refers to the commit marker in the log entry. (a) Running example of log entry allocation and commit. (b) Running example of recovery process on failure.

programmers must reason about memory ordering at the ISA abstraction, making it error-prone and burdensome to write recoverable PM programs. Recent efforts [11, 12, 29, 13, 27, 26, 13, 30, 31] extend persistency semantics and provide ISA-agnostic programming frameworks in high-level languages, such as C++ and Java. These proposals use existing synchronization primitives in high-level languages to also prescribe order on persists and enforce failure atomicity for groups of persists. Failure atomicity reduces the state space visible to recovery and greatly simplifies persistent programming by ensuring either all or none of the updates within a region are visible in case of failure.

Some models [29, 27, 26] enable failure-atomic transactions for transaction-based programs and rely on external synchronization [54, 26] to provide transaction isolation. ATLAS [11] and SFR-based [12, 30, 15] persistency models look beyond transaction-based programs to provide failure atomicity using languages' low-level synchronization primitives. ATLAS employs undo logging to provide failure atomicity for outermost critical sections—code region bounded by lock and unlock synchronization operations. In contrast, SFR-based persistency models [12, 30, 15] enable failure-atomic synchronization-free regions—code regions bounded by low-level synchronization primitives, such as acquire and release. The models enable undo logging as a part of language semantics—compiler implementations emit logging for persistent stores in the program, transparent to the programmer. We propose logging based on strand persistency primitives. We integrate our logging mechanisms into compiler passes to implement language-level persistency model semantics.

Logging implementation. Undo logging ensures failure atomicity by recording the old value of data before it is updated in a failure-atomic region. Undo logs are committed when updates persist in PM. On failure, a recovery process uses uncommitted undo logs to roll back partial PM updates. For correct recovery, undo logs need to persist before in-place updates (as shown earlier in Figure 1(b)). A pairwise persist ordering between an undo log and corresponding in-place update ensures correct recovery. Within a failure-atomic region, undo logs for different updates need not be ordered—logging operations can persist concurrently (as shown under the ideal ordering constraints in Figure 1(d)). Similarly, in-place updates may persist concurrently, too, provided they do not overlap.

Figure 5 shows our logging mechanism, which employs StrandWeaver's ISA primitives to enable failure-atomic updates. We persist undo logs and in-place updates using `CLWB` and order these persists using a persist barrier. The persist barrier ensures that the log is created and flushed to PM before the update. Each logging operation and update is performed on a separate strand; we issue `NewStrand` after each log-update sequence, enabling persist concurrency across the independent updates. We ensure that all persists within a failure-atomic region complete before exiting the region by enclosing it within `JoinStrand` operations—these ensure that persists on different strands do not "leak" out of the failure-atomic region. The precise implementation of `log_begin()` and `log_end()` vary based on the semantics prescribed by various language-level persistency models.

Integrating with language persistency models. Under ATLAS, we initiate and terminate failure-atomic regions at the lock and unlock operations of outermost critical sections. `log_begin()` creates a log entry for the lock operation. The log entry captures happens-after ordering relations on the lock due to prior unlock operations on the same lock, similar to the mechanism employed by ATLAS [11]. `log_end()` for an unlock operation updates metadata (similar to [11]) to record happens-before ordering information required by the subsequent lock operation on that lock. `log_store()` creates an undo log entry that records the address and prior value of an update. Under SFR-based persistency, we emit `log_begin()` and `log_end()` at the acquire and release synchronization operations bracketing each SFR. As in prior work [12], `log_begin()` and `log_end()` log happens-before ordering relations in their log entries to ensure correct recovery. Under failure-atomic transactions [29, 26], `log_end()` flushes all PM mutations in the transaction and ensures that they persist before committing the logs.

Log structure. We initialize and manage a per-thread circular log buffer in PM as an array of 64-byte cache-line-aligned log entries. Additional log entries are allocated dynamically if the log space is exhausted.

Our log entry structure is similar to prior work [12, 11]:

- `Type`: Entry type [`Store`, `Acquire`, `Release`] in ATLAS or SFR, [`Store`, `TX_BEGIN`, `TX_END`] for transactions
- `Addr`: Address of the update
- `Value`: Old value of an update in a store log entry, or the metadata for happens-before relations for a sync. operation
- `Size`: Size of the access

Core	8-cores, 2GHz OoO 6-wide Dispatch, 8-wide Commit 224-entry ROB 72/64-entry Load/Store Queue
I-Cache	32kB, 2-way, 64B 1ns cycle hit latency, 2 MSHRs
D-Cache	32kB, 2-way, 64B 2ns hit latency, 6 MSHRs
L2-Cache	28MB, 16-way, 64B 16ns hit latency, 16 MSHRs
DRAM, PM controller	64/32-entry write/read queue, 1kB row buffer
PM	Modeled as per [58], 346ns read latency, 96ns write latency to controller 500ns write latency to PM

TABLE I: Simulator Specifications.

- `Valid`: Valid bit for the entry
- `Commit marker`: Commit intent marker for log commit

Our logging implementation maintains *head* and *tail* pointers to record the bounds of potentially valid log entries in the log buffer. Figure 6(a) (step ❶) shows the head and tail pointers and the valid log entries that belong to synchronization operations (marked red) and store operations (marked blue). The tail pointer indicates the location to which the next log entry will be appended—we advance the tail pointer upon creation of each log entry. We maintain the tail pointer in volatile memory so that log entries created on different strands are not ordered by updates to the tail pointer (as a consequence of strong persist atomicity, see Equation 3).

The head pointer marks the beginning of potentially uncommitted log entries. In Figure 6(a), suppose log entry `4` marks the end of a failure-atomic region. Before commit begins, we set the commit marker of the log entry that terminates the failure-atomic region as shown in step ❷ in Figure 6(a)—this marks that the log commit has initiated. We mark undo-log entries corresponding to a failure-atomic region invalid (step ❸ in Figure 6(a)), and update and flush the head pointer to commit those log entries (step ❹ in Figure 6(a)).

On failure, the tail pointer in volatile memory is lost, and the persistent head pointer is used to initiate recovery. First, the recovery process identifies the log entries that were committed, but not invalidated prior to failure; this scenario occurs if failure happens during an ongoing commit operation. Figure 6(b) shows an example with the commit marker for log entry `4` set, log entries `1,2` invalidated, and log entries `3,4` yet to be invalidated (step ❶). The recovery process invalidates the log entries from the head pointer to the log entry `4` with the commit marker set, and advances the head pointer, as shown in Figure 6(b) (step ❷). Then, the recovery process scans the log buffer starting from the head pointer and rolls back values recorded in valid log entries in reverse order of their creation, as shown in Figure 6(b) (step ❸).

VI. Evaluation

We next describe our evaluation of StrandWeaver.

A. Methodology

We implement StrandWeaver in the gem5 simulator [59], configured as per Table I. We model a PM device as per the recent characterization studies of Intel's Optane memory [58], as shown in Table I. We configure our design with 16-entry persist queue and four 4-entry strand buffers. StrandWeaver requires a total of 144B of additional storage each in the persist queue and strand buffer unit per core. It also extends the write-back buffer and snoop buffer, 8 bits per entry each, to record a 2-bit tail index for four strand buffers. We consider other StrandWeaver configurations in Section VI-C.

Benchmarks	Description	CKC
Queue	Insert/delete to queue [16, 18]	0.78
Hashmap	Read/update to hashmap [26, 17]	4.83
Array Swap	Swap of array elements [26, 17]	4.45
RB-Tree	Insert/delete to RB-Tree [26, 18]	3.46
TPCC	New Order trans. from TPCC [61, 17]	1.58
N-Store (rd-heavy)	90% read/10% write KV workload [60]	4.41
N-Store (balanced)	50% read/50% write KV workload [60]	8.06
N-Store (wr-heavy)	10% read/90% write KV workload [60]	10.05

TABLE II: Benchmarks. `CLWB`s per 1000 cycles (CKC) measures benchmark write intensity in the non-atomic design.

Benchmarks. Table II describes the microbenchmarks and benchmarks we study, and reports `CLWB`s issued per thousand CPU cycles (CKC) as a measure of their write-intensity. Queue performs insert and delete operations to a persistent queue. Hashmap performs updates to a persistent hash, array-swap swaps two elements in an array, RB-tree performs inserts and deletes to a persistent red-black tree, and TPCC performs new order transactions, which model an order processing system. Additionally, we study N-Store [60], a persistent key-value store benchmark, using workloads with different read-write ratios, as listed in Table II. We use the YCSB engine with N-Store to generate load, and modify its undo-log engine to integrate our logging mechanisms. The microbenchmarks and benchmark each run eight threads and perform 50K operations on persistent data structures. As shown in Table II, N-Store under a write-heavy workload is the most write-intensive benchmark and queue and TPCC are the least write-intensive microbenchmarks in our evaluation.

Language-level persistency models. As explained in Section V, we design language-level implementations that map persistency semantics in high-level languages to StrandWeaver's ISA primitives. We implement failure-atomic transactions (TXNs), outermost critical sections (ATLAS), and SFRs to evaluate StrandWeaver for each of the benchmarks.

We compare following designs in our evaluation:

Intel x86. This design implements language-level persistency models using Intel's existing ISA primitives, which divide program regions into epochs using `SFENCE`, and allow persist reordering only within the epochs, as discussed earlier in Section II-B. Under Intel's persistency model, `SFENCE` orders durability of persists—it ensures that the earlier `CLWB`s drain to the PM before any subsequent `CLWB`s are issued. In this design, logs and in-place updates are ordered by `SFENCE`.

HOPS. This design implements HOPS [19], a state-of-the-art hardware mechanism that builds a delegated epoch persistency model. HOPS implements an `ofence` barrier that divides program execution into epochs and ensures that they are ordered to PM. Unlike Intel x86 `SFENCE` operation, a lightweight

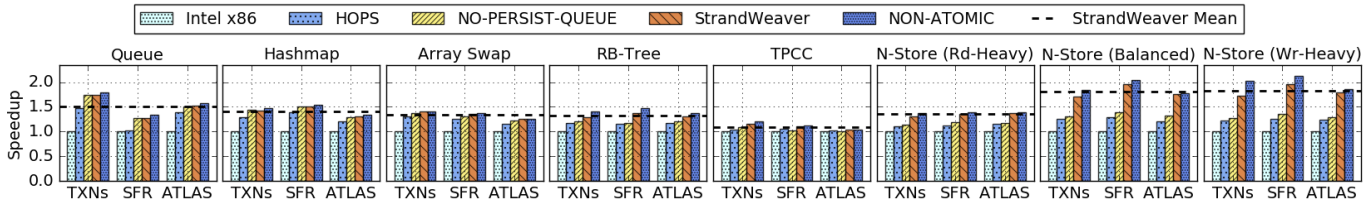

Fig. 7: Speedup of StrandWeaver, HOPS, and Non-atomic design normalized to the implementation based on Intel's persistency model.

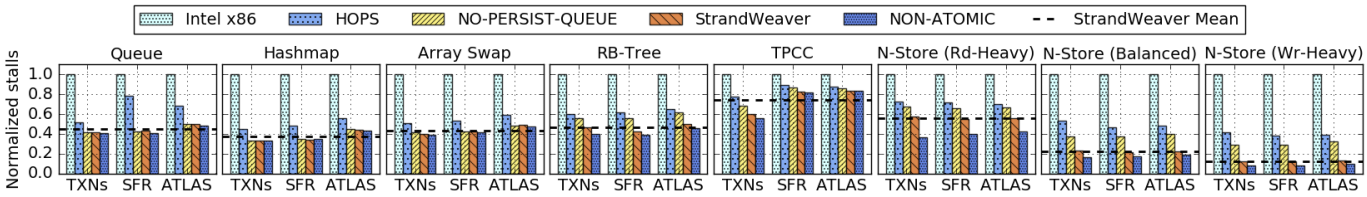

Fig. 8: CPU stalls as hardware enforces persist order. Stalls due to barriers create back pressure in CPU pipeline, and blocks program execution.

`ofence` barrier does not enforce durability of earlier persists. It delegates the persist ordering due to `ofence` to the persist buffers (similar to the strand buffers in StrandWeaver). The `dfence` barrier ensures durability of prior epochs — it persists updates issued in earlier epochs to the PM by flushing the persist buffers to the point of persistence. HOPS employs `ofence` to order the logs before the corresponding updates, and `dfence` to flush the updates to PM before committing the undo logs at the end of each failure-atomic region.

NO-PERSIST-QUEUE. This is StrandWeaver's intermediate hardware design that implements the strand persistency model, but without the addition of a persist queue. Incoming `CLWB`s, persist barriers, `NewStrand` and `JoinStrand` are inserted in the existing store queue. The store queue manages the order in which `CLWB`s, `NewStrand`, and persist barriers issue to the strand buffer unit. We use this design to study the concurrency enabled by the strand buffer unit.

StrandWeaver. This design implements our proposal, as detailed in Sections IV-V.

NON-ATOMIC. In this design, we do not order log persists with in-place updates—we remove the `SFENCE` between the log entry creation and in-place update. Absent any ordering constraints, this design shows the best-case performance that StrandWeaver can obtain due to relaxed persist ordering. Note that, since logs are not ordered before in-place updates, this design does not assure correct failure recovery.

B. Performance Comparison

Figure 7 shows the performance comparison for our microbenchmarks and benchmarks, implemented under the three language-level persistency models across the hardware designs. Figure 8 shows CPU pipeline stalls as hardware enforces persist ordering constraints—frequent stalls due to barriers fill hardware queues and block program execution.

StrandWeaver outperforms Intel x86. StrandWeaver outperforms the baseline Intel x86 design in all the benchmarks we study, as it relaxes persist order relative to Intel's existing ordering primitives. Intel x86 orders log operations and in-place updates using `SFENCE`, enforcing drastically stricter ordering constraints than required for correct recovery. As explained in Section II-B, `SFENCE` divides program execution into epochs and `CLWB`s are allowed to reorder/coalesce only within the epochs. Unfortunately, the persist concurrency within epochs is limited by their small size [18]. In contrast, StrandWeaver enforces only pairwise ordering between the undo log and in-place update. As a result, StrandWeaver outperforms the Intel baseline by 1.45× on average.

Note that we achieve speedup over this design even though the memory controller lies in the persistent domain and hides the write latency of the PM device. Under the Intel x86 persistency model, `SFENCE`s stall issue for subsequent updates until prior `CLWB`s complete. The additional constraints due to `SFENCE` fill up the store queue, creating back pressure and stalling the CPU pipeline. StrandWeaver encounters 62.4% fewer pipeline stalls, resulting in a performance gain of 1.45× on average over Intel x86. Table II shows that N-Store, under a write-heavy workload, is the most write-intensive benchmark that we evaluate. As a result, StrandWeaver achieves the highest speedup of 1.82× on average in N-Store.

StrandWeaver outperforms HOPS. StrandWeaver achieves a speedup of 1.20× on average compared to HOPS, a state-of-the-art implementation of the epoch persistency model. HOPS implements a lightweight `ofence` barrier that orders undo logs before the corresponding updates within a failure-atomic region. Unfortunately, the `ofence` barrier also orders the subsequent logs and PM updates, which may be issued concurrently to PM. On the contrary, StrandWeaver performs each logging operation and update on a separate strand. Thus, StrandWeaver achieves greater persist concurrency and a speedup of up to 1.55× over HOPS.

Persist concurrency due to strand buffers. The strand buffers issue `CLWB`s that lie on different strands concurrently. As shown in Figure 7, StrandWeaver's intermediate design, without the persist queue, achieves 1.29× speedup over Intel x86 design on average, with 52.3% fewer pipeline stalls. Adding the persist queue prevents head-of-the-line blocking

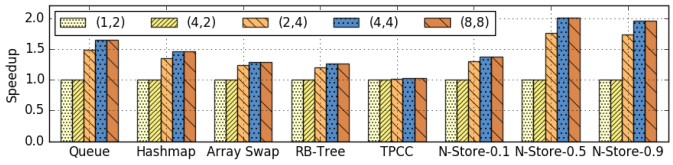

Fig. 9: Sensitivity study with different StrandWeaver configurations denoted as (Number of strand buffers, Number of entries per strand buffer).

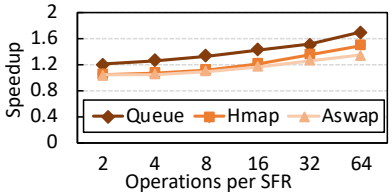

Fig. 10: Speedup with varying number of operations per SFR

due to long-latency CLWBs in the store queue—stores on different strands may enter the store queue and issue concurrent to CLWBs. StrandWeaver attains an additional speedup of 1.13× over the variant without the persist queue.

Performance comparable to non-atomic design. Figure 7 shows performance for the non-atomic design that removes the pairwise ordering constraint between the updates and their logs. We include this design to study the limit on performance that StrandWeaver might achieve—this design does not ensure correct recovery as updates can persist before their logs. StrandWeaver incurs 3.1% slowdown in microbenchmarks and 5.7% slowdown in N-Store relative to this upper bound due to additional persist ordering within each strand.

Low write-intensity benchmarks. StrandWeaver achieves its lowest speedup of 5.32% on average in TPCC for the three persistency model implementations. TPCC acquires multiple locks per new order transaction to ensure isolation. As such, there is high lock acquisition overhead per failure-atomic region. As per Table II, Queue has the lowest write intensity, but achieves a speedup of 1.64× on average. Queue has the least concurrency among the benchmarks we study, as all its threads contend on a single lock to serialize push and pop operations to the queue. CLWBs fall on the critical execution path and additional ordering constraints incur execution delay.

Sensitivity to language-level persistency model. StrandWeaver's implementation that ensures failure-atomic transactions flushes in-place updates and commits logs at the end of the failure-atomic region. In contrast, the SFR implementation issues batched commits by logging happens-before relations in logs at the end of each SFR and continuing execution without stalling for log commits. ATLAS issues batched log commits too, but employs heavier-weight mechanisms to record happens-before order between the lock and unlock operation, as compared to SFR [12]. Thus, StrandWeaver achieves the highest speedup of 1.50× for SFR, followed by 1.45× speedup for failure-atomic transactions, and 1.40× speedup for ATLAS.

C. Sensitivity Study

Next, we evaluate StrandWeaver for different configurations.

Configuring strand buffer unit. Figure 9 shows the evaluation of StrandWeaver with varying number of strand buffers and entries per strand buffer. Due to space limitations, we show only the results for the SFR implementation—the performance trend for the other implementations is similar. With fewer than four entries per buffer, the strand buffer unit fails to leverage available persist concurrency on different strands, even when we configure the unit with four buffers. As we increase the number of buffer entries to four, even with two strand buffers, StrandWeaver's performance improves by 1.36×, as persists on different strands can drain concurrently. Finally, StrandWeaver's performance improves by a further 7.7% with four strand buffers and four buffer entries each. As we see no further improvement with additional state (*e.g.* eight strand buffers with eight buffer entries, in Figure 9), we configure the strand buffer unit with four buffers, each with four entries.

Size of failure-atomic regions. The number of operations per failure-atomic region determines the available persist concurrency in StrandWeaver. In Figure 10(a), we vary number of operations performed per failure-atomic SFR in our microbenchmarks. The number of concurrent strands reduces with the number of operations per region. StrandWeaver achieves 1.10× speedup (avg.) over Intel x86 baseline with two operations per SFR; the speedup increases with the number of operations per region.

VII. Related Work

PM adoption has been widely studied in hardware design [28, 16, 18, 17, 35, 19, 62, 38, 34, 63], file systems [28, 64, 65, 66, 5, 67, 68, 69], runtime systems [70, 71, 72, 73, 74, 60, 25, 43, 11, 75, 76, 15, 77, 78, 79, 80, 32, 81, 78], persistent data structures [82, 83, 84, 85], and distributed systems [86, 87, 88, 89].

Persistency models. Prior works propose memory persistency models [10, 9, 16, 28] to define the order in which updates persist to PM. Several works [18, 19, 17, 45] propose hardware mechanisms to build strict and relaxed persistency models defined by Pelley *et al.* [16]. DPO [17] builds a buffered strict persistency model for hardware that provides a relaxed consistency model. Unfortunately, DPO does not provide mechanisms to enforce data durability (*e.g.* at the end of a failure-atomic region), and is limited to hardware that builds snoop-based coherence mechanisms with a single PM controller. HOPS [19], BPFS [28], Shin *et al.* [45], and Joshi *et al.* [18] implement epoch persistency models in hardware by extending the cache hierarchy and PM controller to track persist dependencies. As we demonstrate in Section VI-B, StrandWeaver outperforms HOPS [19], an epoch persistency model, by 1.20× on average.

Hardware logging. Several prior works propose mechanisms to perform write-ahead logging in hardware to enable efficient failure-atomic updates to PM. Doshi *et al.* [35] builds redo logging in hardware by implementing a victim cache to avoid out-of-order cache evictions. We primarily explore undo logging because a wide class of language implementations use it to ensure failure-atomic regions [12, 13, 11]. Other

logging mechanisms, such as redo logging, may also benefit from the relaxed semantics under strand persistency. Redo logging involves recording new updates in logs, followed by performing the in-place updates. Under strand persistency, each failure-atomic transaction may be performed on a separate strand. Within each strand, transactions can create redo logs, issue a persist barrier and then perform in-place updates. A group commit operation can merge strands and commit prior transactions. We leave this analysis to future work.

ATOM [18], FIRM [90], Ogleari *et al.* [36], and DHTM [37] build undo- or hybrid undo-redo logging mechanisms in hardware. These hardware mechanisms primarily provide failure atomicity for transactions, but fail to extend to other synchronization primitives or other logging implementations used by high-level language persistency models [11, 12, 13]. Proteus [38] introduces specialized log-load and log-flush instructions to perform software-assisted logging in hardware—it allows concurrency only for logging persists. StrandWeaver's primitives can be employed for a variety of language implementations and logging mechanisms.

Software-based mechanisms. NV-Heaps [26] and Mnemosyne [27] provide library interfaces to allow programmers to build persistent memory data structures. Similarly, SoftWrAP [91], REWIND [71], Pisces [92], Wang *et al.* [93], Kamino-Tx [75], and DUDETM [43] provide software libraries that enable failure-atomic transactions. NVthreads [94] extends ATLAS [11] by updating copy of data in critical section and merging it to live copy at the end of a critical section. Similarly, JUSTDO logging [95] extends ATLAS to guarantee program recovery to the end of ongoing outermost critical section. Kolli *et al.* [25] provides efficient implementation of failure-atomic software transactions and proposes a deferred log commit mechanism to delay commits until transactions conflict. Janus [96] proposes software interface to efficiently implement storage features such as encryption and deduplication. Alpaca [97] and Coati [98] provide a task-based programming model for intermittent systems, where updates in a task are failure atomic. StrandWeaver's primitives may be employed to enable failure-atomic transactions.

Others. Scoped fence [99] restricts memory order due to consistency model to a programmer-annotated scope. Similarly, StrandWeaver restricts persist order due to a persist barrier to a strand. Idetic [100] and Hibernus [101] detect imminent power failures and flush volatile state to PM. Survive [102], Kannan *et al.* [103], and ThyNVM [104] propose coarse-grained checkpointing for PM systems. Others [105, 106, 107, 108] leverage PM's density to use it as a scalable DRAM alternative. These proposals are orthogonal to StrandWeaver.

VIII. Conclusion

In this work, we proposed StrandWeaver, a hardware strand persistency model to minimally constrain orderings on PM operations. We formally defined primitives under strand persistency to specify intra-strand, inter-strand, and inter-thread persist ordering constraints. We constructed hardware mechanisms to implement strand persistency model that expose ISA primitives to relax persist order. Furthermore, we implemented logging mechanisms that map persistency semantics in high-level languages to the low-level ISA primitives using our logging mechanism. Finally, we demonstrated that StrandWeaver can achieve 1.45× speedup on average as it can enable greater persist concurrency than existing ISA-level mechanisms.

Acknowledgements

We would like to thank the anonymous reviewers for their valuable feedback. We would also like to thank Akshitha Sriraman for her insightful suggestions, and Shriya Sethuraman for proof-reading our draft. This work was supported by ARM and the National Science Foundation under the award NSF-CCF-1525372. William Wang and Stephan Diestelhorst received funding from the European Union's Horizon 2020 research and innovation programme under project Sage 2, grant agreement 800999.

References

[1] "INTEL OPTANE DC PERSISTENT MEMORY," https://www.intel.com/content/www/us/en/products/memory-storage/optane-dc-persistent-memory.html.

[2] "Understand and deploy persistent memory," https://docs.microsoft.com/en-us/windows-server/storage/storage-spaces/deploy-pmem.

[3] "Available first on Google Cloud: Intel Optane DC Persistent Memory," https://tinyurl.com/gcp-release.

[4] M. Andrei, C. Lemke, G. Radestock, R. Schulze, C. Thiel, R. Blanco, A. Meghlan, M. Sharique, S. Seifert, S. Vishnoi, and et al., "Sap hana adoption of non-volatile memory," *Proc. VLDB Endow.*, vol. 10, no. 12, p. 1754–1765, Aug. 2017.

[5] "Instruction prefetching using branch prediction information," in *Proceedings of the 1997 International Conference on Computer Design (ICCD '97)*, ser. ICCD '97. USA: IEEE Computer Society, 1997, p. 593.

[6] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1992.

[7] D. E. Lowell and P. M. Chen, "Free transactions with rio vista," *SIGOPS Oper. Syst. Rev.*, vol. 31, no. 5, p. 92–101, Oct. 1997.

[8] 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 Trans. Database Syst.*, vol. 17, no. 1, pp. 94–162, Mar. 1992.

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

[10] ARM, "Armv8-a architecture evolution," 2016, https://tinyurl.com/arm-nvm.

[11] D. R. Chakrabarti, H.-J. Boehm, and K. Bhandari, "Atlas: Leveraging locks for non-volatile memory consistency," in *Proc. OOPSLA*. New York, NY, USA: ACM, 2014, pp. 433–452.

[12] V. Gogte, S. Diestelhorst, W. Wang, S. Narayanasamy, P. M. Chen, and T. F. Wenisch, "Persistency for synchronization-free regions," in *Proc. PLDI*. New York, NY, USA: ACM, 2018, pp. 46–61.

[13] A. Kolli, V. Gogte, A. Saidi, S. Diestelhorst, P. M. Chen, S. Narayanasamy, and T. F. Wenisch, "Language-level persistency," in *Proc. ISCA*. New York, NY, USA: ACM, 2017, pp. 481–493.

[14] A. Kolli, V. Gogte, A. Saidi, S. Diestelhorst, P. M. Chen, S. Narayanasamy, and T. F. Wenisch, "TARP: Translating acquire-release persistency," in *NVMW*, San Diego, CA, 2017. [Online]. Available: http://nvmw.eng.ucsd.edu/2017/assets/abstracts/1

[15] V. Gogte, S. Diestelhorst, W. Wang, S. Narayanasamy, P. M. Chen, and T. F. Wenisch, "Failure-atomic synchronization-free regions," in *NVMW*, San Diego, CA, 2018. [Online]. Available: http://nvmw.ucsd.edu/nvmw18-program/unzip/current/nvmw2018-final42.pdf

[16] S. Pelley, P. M. Chen, and T. F. Wenisch, "Memory persistency," *SIGARCH Comput. Archit. News*, vol. 42, no. 3, p. 265–276, Jun. 2014.

[17] A. Kolli, J. Rosen, S. Diestelhorst, A. Saidi, S. Pelley, S. Liu, P. M. Chen, and T. F. Wenisch, "Delegated persist ordering," in *Proc. MICRO*. IEEE Press, 2016.

[18] A. Joshi, V. Nagarajan, M. Cintra, and S. Viglas, "Efficient persist barriers for multicores," in *Proc. MICRO*. New York, NY, USA: ACM, 2015, pp. 660–671.

[19] S. Nalli, S. Haria, M. D. Hill, M. M. Swift, H. Volos, and K. Keeton, "An analysis of persistent memory use with whisper," in *Proc. ASPLOS*. New York, NY, USA: ACM, 2017, pp. 135–148.

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

[21] H.-J. Boehm and S. V. Adve, "Foundations of the c++ concurrency memory model," in *Proc. PLDI*. New York, NY, USA: ACM, 2008, pp. 68–78.

[22] K. Gharachorloo, A. Gupta, and J. Hennessy, "Two techniques to enhance the performance of memory consistency models," in *Proc. ICPC*, 1991.

[23] L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," *IEEE Trans. Comput.*, vol. 28, no. 9, p. 690–691, Sep. 1979.

[24] D. Lustig, C. Trippel, M. Pellauer, and M. Martonosi, "Armor: Defending against memory consistency model mismatches in heterogeneous architectures," in *Proc. ISCA*. New York, NY, USA: Association for Computing Machinery, 2015, p. 388–400.

[25] A. Kolli, S. Pelley, A. Saidi, P. M. Chen, and T. F. Wenisch, "High-performance transactions for persistent memories," in *Proc. ASPLOS*. New York, NY, USA: ACM, 2016, pp. 399–411.

[26] 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 *Proc. ASPLOS*. New York, NY, USA: ACM, 2011, pp. 105–118.

[27] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: Lightweight persistent memory," in *Proc. ASPLOS*. New York, NY, USA: ACM, 2011, pp. 91–104.

[28] 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 *Proc. SOSP*. New York, NY, USA: ACM, 2009, pp. 133–146.

[29] "pmem.io: Persistent memory programming," https://pmem.io/pmdk/.

[30] A. Kolli, V. Gogte, A. Saidi, S. Diestelhorst, W. Wang, P. M. Chen, S. Narayanasamy, and T. F. Wenisch, "Language support for memory persistency," *IEEE Micro*, vol. 39, no. 3, p. 94–102, May 2019.

[31] M. Wu, Z. Zhao, H. Li, H. Li, H. Chen, B. Zang, and H. Guan, "Espresso: Brewing java for more non-volatility with non-volatile memory," in *Proc. ASPLOS*. New York, NY, USA: ACM, 2018, pp. 70–83.

[32] J. Izraelevitz, H. Mendes, and M. L. Scott, "Linearizability of persistent memory objects under a full-system-crash failure model," in *Proc. DISC*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 313–327.

[33] A. Kolli, S. Pelley, A. Saidi, P. M. Chen, and T. F. Wenisch, "Persistency programming 101," in *NVMW*, San Diego, CA, 2015. [Online]. Available: http://nvmw.ucsd.edu/2015/assets/abstracts/33

[34] A. Joshi, V. Nagarajan, S. Viglas, and M. Cintra, "Atom: Atomic durability in non-volatile memory through hardware logging," in *Proc. HPCA*, Feb 2017, pp. 361–372.

[35] K. Doshi, E. Giles, and P. Varman, "Atomic persistence for scm with a non-intrusive backend controller," in *Proc. HPCA*, March 2016, pp. 77–89.

[36] M. A. Ogleari, E. L. Miller, and J. Zhao, "Steal but no force: Efficient hardware undo+redo logging for persistent memory systems," in *Proc. HPCA*, Feb 2018, pp. 336–349.

[37] A. Joshi, V. Nagarajan, M. Cintra, and S. Viglas, "Dhtm: Durable hardware transactional memory," in *Proc. ISCA*. IEEE Press, 2018, p. 452–465.

[38] S. Shin, S. K. Tirukkovalluri, J. Tuck, and Y. Solihin, "Proteus: A flexible and fast software supported hardware logging approach for nvm," in *Proc. MICRO*. New York, NY, USA: ACM, 2017, pp. 178–190.

[39] S. Owens, S. Sarkar, and P. Sewell, "A better x86 memory model: X86-tso," in *Proc. TPHOLs*. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 391–407.

[40] H.-J. Boehm and D. R. Chakrabarti, "Persistence programming models for non-volatile memory," in *Proc. ISMM*. New York, NY, USA: ACM, 2016, pp. 55–67.

[41] T. Shull, J. Huang, and J. Torrellas, "Autopersist: An easy-to-use java nvm framework based on reachability," in *Proc. PLDI*. New York, NY, USA: ACM, 2019, pp. 316–332.

[42] "Use Persistent Memory with Go," https://blogs.vmware.com/opensource/2019/04/03/persistent-memory-with-go/.

[43] M. Liu, M. Zhang, K. Chen, X. Qian, Y. Wu, W. Zheng, and J. Ren, "Dudetm: Building durable transactions with decoupling for persistent memory," in *Proc. ASPLOS*. New York, NY, USA: ACM, 2017, pp. 329–343.

[44] 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.

[45] S. Shin, J. Tuck, and Y. Solihin, "Hiding the long latency of persist barriers using speculative execution," in *Proc. ISCA*. New York, NY, USA: ACM, 2017, pp. 175–186.

[46] Intel, "Deprecating the pcommit instruction," 2016, https://software.intel.com/en-us/blogs/2016/09/12/deprecate-pcommit-instruction.

[47] D. M. Gallagher, W. Y. Chen, S. A. Mahlke, J. C. Gyllenhaal, and W.-m. W. Hwu, "Dynamic memory disambiguation using the memory conflict buffer," in *Proc. ASPLOS*. New York, NY, USA: ACM, 1994, pp. 183–193.

[48] R. Ghiya, D. Lavery, and D. Sehr, "On the importance of points-to analysis and other memory disambiguation methods for c programs," in *Proc. PLDI*. New York, NY, USA: ACM, 2001, pp. 47–58.

[49] S. Sethumadhavan, R. Desikan, D. Burger, C. R. Moore, and S. W. Keckler, "Scalable hardware memory disambiguation for high ilp processors," in *Proc. MICRO*. Washington, DC, USA: IEEE Computer Society, 2003, pp. 399–.

[50] W. Wang and S. Diestelhorst, "Quantify the performance overheads of pmdk," in *Proc. MEMSYS*. New York, NY, USA: ACM, 2018, pp. 50–52.

[51] A. Arvind and J.-W. Maessen, "Memory model = instruction reordering + store atomicity," in *Proc. ISCA*. USA: IEEE Computer Society, 2006, p. 29–40.

[52] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, p. 558–565, Jul. 1978.

[53] M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber, "Mathematizing C++ concurrency," in *Proc. POPL*. New York, NY, USA: ACM, 2011, pp. 55–66.

[54] "C++ bindings for libpmemobj - synchronization primitives," http://pmem.io/2016/05/31/cpp-08.html.

[55] C. Blundell, M. M. Martin, and T. F. Wenisch, "Invisifence: performance-transparent memory ordering in conventional multiprocessors," in *ACM SIGARCH Computer Architecture News*, vol. 37, no. 3. ACM, 2009, pp. 233–244.

[56] T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos, "Mechanisms for store-wait-free multiprocessors," *SIGARCH Comput. Archit. News*, vol. 35, no. 2, p. 266–277, Jun. 2007.

[57] Intel, "Persistent memory programming," 2015, http://pmem.io/.

[58] J. Izraelevitz, J. Yang, L. Zhang, J. Kim, X. Liu, A. Memaripour, Y. J. Soh, Z. Wang, Y. Xu, S. R. Dulloor, J. Zhao, and S. Swanson, "Basic performance measurements of the intel optane DC persistent memory module," *CoRR*, vol. abs/1903.05714, 2019. [Online]. Available: http://arxiv.org/abs/1903.05714

[59] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, and et al., "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, p. 1–7, Aug. 2011.

[60] J. Arulraj, A. Pavlo, and S. R. Dulloor, "Let's talk about storage & recovery methods for non-volatile memory database systems," in *Proc. SIGMOD*. New York, NY, USA: ACM, 2015, pp. 707–722.

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

[62] 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 *Proc. MICRO*. New York, NY, USA: ACM, 2013, pp. 421–432.

[63] D. Gope, A. Basu, S. Puthoor, and M. Meswani, "A case for scoped persist barriers in gpus," in *Proc. GPGPU*. New York, NY, USA: ACM, 2018, pp. 2–12.

[64] X. Wu and A. L. N. Reddy, "Scmfs: A file system for storage class memory," in *Prof. SC*, ser. SC '11. New York, NY, USA: Association for Computing Machinery, 2011.

[65] H. Volos, S. Nalli, S. Panneerselvam, V. Varadarajan, P. Saxena, and M. M. Swift, "Aerie: Flexible file-system interfaces to storage-class memory," in *Proc. EuroSys*. New York, NY, USA: ACM, 2014, pp. 14:1–14:14.

[66] J. Xu and S. Swanson, "Nova: A log-structured file system for hybrid volatile/non-volatile main memories," in *Proc. FAST*. USA: USENIX Association, 2016, p. 323–338.

[67] J. Xu, L. Zhang, A. Memaripour, A. Gangadharaiah, A. Borase, T. B. Da Silva, S. Swanson, and A. Rudoff, "Nova-fortis: A fault-tolerant non-volatile main memory file system," in *Proc. SOSP*. New York, NY, USA: ACM, 2017, pp. 478–496.

[68] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson, "System software for persistent memory," in *Proc. EuroSys*. New York, NY, USA: ACM, 2014, pp. 15:1–15:15.

[69] J. Xu, J. Kim, A. Memaripour, and S. Swanson, "Finding and fixing performance pathologies in persistent memory software stacks," in *Proc. ASPLOS*. New York, NY, USA: ACM, 2019, pp. 427–439.

[70] T. Wang and R. Johnson, "Scalable logging through emerging non-volatile memory," *Proc. VLDB Endow.*, vol. 7, no. 10, p. 865–876, Jun. 2014.

[71] A. Chatzistergiou, M. Cintra, and S. D. Viglas, "REWIND: recovery write-ahead system for in-memory non-volatile data-structures," *Proc. VLDB Endow.*, vol. 8, no. 5, p. 497–508, Jan. 2015.

[72] I. Oukid, D. Booss, W. Lehner, P. Bumbulis, and T. Willhalm, "Sofort: A hybrid scm-dram storage engine for fast data recovery," in *Proc. DaMoN*, 2014.

[73] H. Kimura, "Foedus: Oltp engine for a thousand cores and nvram," in *Proc. SIGMOD*. New York, NY, USA: Association for Computing Machinery, 2015, p. 691–706.

[74] J. Arulraj, M. Perron, and A. Pavlo, "Write-behind logging," *Proc. VLDB Endow.*, vol. 10, no. 4, p. 337–348, Nov. 2016.

[75] A. Memaripour, A. Badam, A. Phanishayee, Y. Zhou, R. Alagappan, K. Strauss, and S. Swanson, "Atomic in-place updates for non-volatile main memories with kamino-tx," in *Proc. EuroSys*. New York, NY, USA: ACM, 2017, pp. 499–512.

[76] J. Guerra, L. Marmol, D. Campello, C. Crespo, R. Rangaswami, and J. Wei, "Software persistent memory," in *Proc. ATC*. Boston, MA: USENIX, 2012, pp. 319–331.

[77] T. M. Nguyen and D. Wentzlaff, "Picl: A software-transparent, persistent cache log for nonvolatile main memory," in *Proc. MICRO*. IEEE Press, 2018, p. 507–519.

[78] V. Gogte, W. Wang, S. Diestelhorst, A. Kolli, P. M. Chen, S. Narayanasamy, and T. F. Wenisch, "Software wear management for persistent memories," in *Proc. FAST*. Boston, MA: USENIX Association, Feb. 2019, pp. 45–63.

[79] I. Narayanan, A. Ganesan, A. Badam, S. Govindan, B. Sharma, and A. Sivasubramaniam, "Getting more performance with polymorphism from emerging memory technologies," in *Proc. SYSTOR*. New York, NY, USA: ACM, 2019, pp. 8–20.

[80] J. Huang, K. Schwan, and M. K. Qureshi, "Nvram-aware logging in transaction systems," *Proc. VLDB Endow.*, vol. 8, no. 4, p. 389–400, Dec. 2014.

[81] H.-J. Boehm and D. R. Chakrabarti, "Persistence programming models for non-volatile memory," in *Proc. ISMM*. New York, NY, USA: ACM, 2016, pp. 55–67.

[82] S. Venkataraman, N. Tolia, P. Ranganathan, and R. H. Campbell, "Consistent and durable data structures for non-volatile byte-addressable memory," in *Proc. FAST*. Berkeley, CA, USA: USENIX Association, 2011, pp. 5–5.

[83] 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.

[84] D. Hwang, W.-H. Kim, Y. Won, and B. Nam, "Endurable transient inconsistency in byte-addressable persistent b+-tree," in *Proc. FAST*. Oakland, CA: USENIX Association, 2018, pp. 187–200.

[85] H. Chauhan, I. Calciu, V. Chidambaram, E. Schkufza, O. Mutlu, and P. Subrahmanyam, "NVMOVE: Helping programmers move to byte-based persistence," in *INFLOW*. Savannah, GA: USENIX Association, Nov. 2016.

[86] D. Kim, A. Memaripour, A. Badam, Y. Zhu, H. H. Liu, J. Padhye, S. Raindel, S. Swanson, V. Sekar, and S. Seshan, "Hyperloop: Group-based nic-offloading to accelerate replicated transactions in multi-tenant storage systems," in *Proc. SIGCOMM*. New York, NY, USA: ACM, 2018, pp. 297–312.

[87] Y. Zhang, J. Yang, A. Memaripour, and S. Swanson, "Mojim: A reliable and highly-available non-volatile memory system," in *Proc. ASPLOS*. New York, NY, USA: ACM, 2015, pp. 3–18.

[88] Y. Zhou, R. Alagappan, A. Memaripour, A. Badam, and D. Wentzlaff, "Hnvm: Hybrid nvm enabled datacenter design and optimization," *Microsoft, Microsoft Research, Tech. Rep. MSR-TR-2017-8*, 2017.

[89] Y. Lu, J. Shu, Y. Chen, and T. Li, "Octopus: an rdma-enabled distributed persistent memory file system," in *Proc. ATC*. Santa Clara, CA: USENIX Association, 2017, pp. 773–785.

[90] C. Xu, D. Niu, N. Muralimanohar, R. Balasubramonian, T. Zhang, S. Yu, and Y. Xie, "Overcoming the challenges of crossbar resistive memory architectures," in *Proc. HPCA*, Feb 2015, pp. 476–488.

[91] E. R. Giles, K. Doshi, and P. Varman, "Softwrap: A lightweight framework for transactional support of storage class memory," in *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*, May 2015, pp. 1–14.

[92] J. Gu, Q. Yu, X. Wang, Z. Wang, B. Zang, H. Guan, and H. Chen, "Pisces: A scalable and efficient persistent transactional memory," in *Proc. ATC*. Renton, WA: USENIX Association, Jul. 2019, pp. 913–928.

[93] T. Wang, S. Sambasivam, Y. Solihin, and J. Tuck, "Hardware supported persistent object address translation," in *Proc. MICRO*. New York, NY, USA: ACM, 2017, pp. 800–812.

[94] T. C.-H. Hsu, H. Brügner, I. Roy, K. Keeton, and P. Eugster, "Nvthreads: Practical persistence for multi-threaded applications," in *Proc. EuroSys*. New York, NY, USA: Association for Computing Machinery, 2017, p. 468–482.

[95] J. Izraelevitz, T. Kelly, and A. Kolli, "Failure-atomic persistent memory updates via justdo logging," in *Proc. ASPLOS*. New York, NY, USA: ACM, 2016, pp. 427–442.

[96] S. Liu, K. Seemakhupt, G. Pekhimenko, A. Kolli, and S. Khan, "Janus: Optimizing memory and storage support for non-volatile memory systems," in *Proc. ISCA*. New York, NY, USA: ACM, 2019, pp. 143–156.

[97] K. Maeng, A. Colin, and B. Lucia, "Alpaca: Intermittent execution without checkpoints," *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, Oct. 2017.

[98] E. Ruppel and B. Lucia, "Transactional concurrency control for intermittent, energy-harvesting computing systems," in *Proc. PLDI*. New York, NY, USA: ACM, 2019, pp. 1085–1100.

[99] C. Lin, V. Nagarajan, and R. Gupta, "Fence scoping," in *Proc. SC*, ser. SC '14. IEEE Press, 2014, p. 105–116.

[100] A. Mirhoseini, E. M. Songhori, and F. Koushanfar, "Idetic: A high-level synthesis approach for enabling long computations on transiently-powered asics," in *Proc. PerCom*, March 2013, pp. 216–224.

[101] D. Balsamo, A. S. Weddell, A. Das, A. R. Arreola, D. Brunelli, B. M. Al-Hashimi, G. V. Merrett, and L. Benini, "Hibernus++: A self-calibrating and adaptive system for transiently-powered embedded devices," *Trans. CAD*, vol. 35, no. 12, p. 1968–1980, Nov. 2016.

[102] A. Mirhosseini, A. Agrawal, and J. Torrellas, "Survive: Pointer-based in-dram incremental checkpointing for low-cost data persistence and rollback-recovery," *IEEE Computer Architecture Letters*, vol. 16, no. 2, pp. 153–157, July 2017.

[103] S. Kannan, A. Gavrilovska, K. Schwan, and D. Milojicic, "Optimizing checkpoints using nvm as virtual memory," in *Proc. IPDPS*. USA: IEEE Computer Society, 2013, p. 29–40.

[104] J. Ren, J. Zhao, S. Khan, J. Choi, Y. Wu, and O. Mutlu, "Thynvm: Enabling software-transparent crash consistency in persistent memory systems," in *Proc. MICRO*. New York, NY, USA: ACM, 2015, pp. 672–685.

[105] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting phase change memory as a scalable dram alternative," in *Proc. ISCA*. New York, NY, USA: ACM, 2009, pp. 2–13.

[106] S. Kannan, A. Gavrilovska, and K. Schwan, "pvm: Persistent virtual memory for efficient capacity scaling and object storage," in *Proc. EuroSys*. New York, NY, USA: ACM, 2016, pp. 13:1–13:16.

[107] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, "Scalable high performance main memory system using phase-change memory technology," in *Proc. ISCA*. New York, NY, USA: ACM, 2009, pp. 24–33.

[108] 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 *Proc. MICRO*. New York, NY, USA: ACM, 2009, pp. 14–23.