Lazy Release Persistency

Mahesh Dananjaya, Vasilis Gavrielatos, Arpit Joshi*, Vijay Nagarajan
University of Edinburgh, *Intel
Firstname.Lastname@ed.ac.uk,Arpit.Joshi@intel.com*

Abstract

Fast non-volatile memory (NVM) has sparked interest in *log-free data structures* (LFDs) that enable crash recovery without the overhead of logging. However, recovery hinges on primitives that provide guarantees on what remains in NVM upon a crash. While ordering and atomicity are two well-understood primitives, we focus on ordering and its efficacy in enabling recovery of LFDs. We identify that one-sided persist barriers of *acquire-release persistency* (ARP)—the state-of-the-art ordering primitive and its microarchitectural implementation—are not strong enough to enable recovery of an LFD. Therefore, correct recovery necessitates the inclusion of the more expensive full barriers.

In this paper, we propose strengthening the one-sided barrier semantics of ARP. The resulting persistency model, *release persistency* (RP), guarantees that NVM will hold a consistent-cut of the execution upon a crash, thereby satisfying the criterion for correct recovery of an LFD. We then propose *lazy release persistency* (LRP), a microarchitectural mechanism for efficiently enforcing RP's one-sided barriers. Our evaluation on 5 commonly used LFDs suggests that LRP provides a 14%–44% performance improvement over the state-of-the-art full barrier.

CCS Concepts • **Computer systems organization** → **Parallel architectures**; • **Hardware** → **Emerging technologies**.

Keywords Memory Consistency Models, Persistent Memory, Release Consistency, Log-Free Data Structures

ACM Reference Format:
Mahesh Dananjaya, Vasilis Gavrielatos, Arpit Joshi, Vijay Nagarajan. 2020. Lazy Release Persistency. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20), March 16–20, 2020, Lausanne, Switzerland*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3373376.3378481

ASPLOS '20, March 16–20, 2020, Lausanne, Switzerland

ACM ISBN 978-1-4503-7102-5/20/03…$15.00
https://doi.org/10.1145/3373376.3378481

1 Introduction

The advent of fast non-volatile memory (NVM) has enabled the possibility of recovering from a system crash while incurring minimal overhead during program's normal operation [7–9, 18, 20–22, 31, 36, 40, 43]. Program recovery, however, hinges on primitives that control the order in which data becomes persistent [1, 21, 24–26, 34, 35, 38]. What primitive(s) offer a programmable interface while allowing for an efficient implementation at the hardware level?

The question is the subject of an ongoing debate. Should languages support *failure-atomicity* for a group of writes, or should languages forego atomicity and support *only ordering* between individual word-granular writes?

Kolli et al. [24] make a case for *only ordering*, arguing that it is more general and performance-friendly when compared to failure-atomicity which requires logging. They reason that because future processors are likely going to guarantee atomicity of only individual persists, a library that provides failure-atomicity can be used when necessary. They then propose *acquire-release persistency* (ARP), a language-level persistency model that extends C++11 by treating its release/acquire annotations as one-sided persist barriers. They also propose a hardware mechanism for enforcing these one-sided barriers efficiently. Thus, the key to ARP's performance is its one-sided barriers that attempt to precisely enforce the orderings intended by the programmer.

In subsequent work, however, Gogte et al. [14] make a case for failure-atomicity, arguing that the absence of failure-atomicity in ARP (and indeed any ordering primitive) makes reasoning about recovery extremely cumbersome.

Not all programs require failure-atomicity, however. In fact, an important class of nonblocking data structures [4, 10, 11, 19, 33, 37] is designed specifically to avoid atomic regions. Recovery from a crash comes for free, aka *null recovery*, as long as writes persist in the order in which they become visible [5, 6, 19, 39, 41]. The emergence of NVM has sparked interest in these *log-free data structures* (LFDs) [10] primarily because such programs enable recovery without the overhead of logging. Therefore, while we concur with Gogte et al. [14] that failure-atomicity simplifies recovery in the general case, we argue that languages must *also* offer efficient ordering primitives for supporting LFDs.

However, we identify that ARP's one-sided barriers are not strong enough to enable recovery in LFDs. Consider Figure 1 that depicts an execution history of a concurrent log-free linked list. Thread T0 first prepares node A1 for insertion by writing to its fields (Figure 1a). Then, it links

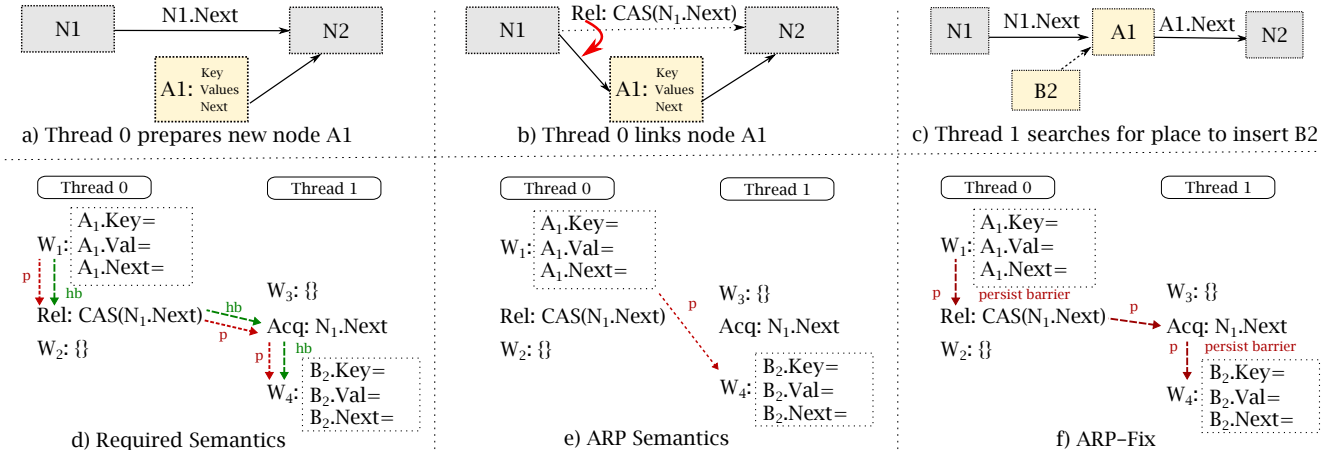

Figure 1: (a,b) Thread 0 inserts A1 in a log-free linked list. (c) Then, Thread 1 attempts to insert B2. (d) shows the required persistency semantics for an insert. (e) shows the semantics provided by ARP and (f) shows how ARP can match the required semantics. W:{} represents potential writes.

A1 with the rest of the list via a single atomic Compare-and-Swap (CAS) instruction (Figure 1b). Note that from a consistency standpoint the CAS must have release semantics to ensure that the writes to A1 become visible before the link is updated. To enable recovery, persistency must mirror visibility: the writes to A must persist before the CAS persists (Figure 1d). However, ARP's one-way barrier does not provide this guarantee. (As shown in Figure 1e, it only ensures that writes to A1 persist before writes to B2 from the acquiring thread persist). Therefore, to enable recovery, the programmer must place full persist barriers before the release and after the acquire (Figure 1f). Alas, the full barrier requirement annuls ARP's performance benefits which stem from its one-sided barriers.

In this paper, we propose strengthening the one-sided barrier semantics of ARP to enable recovery of LFDs. The resulting persistency model, dubbed *Release Persistency* (RP), ensures that any two writes that are ordered by the consistency model also persist in that order. Thus, the persistency model guarantees that the NVM will hold a consistent-cut of the execution upon a crash, thereby satisfying the criterion for correct recovery of an LFD [19].

We then propose an efficient microarchitectural mechanism for enforcing the one-sided barrier semantics of RP. Going back to Figure 1d, the challenge is to enforce $W_1 \xrightarrow{p} Rel \xrightarrow{p} W_4$, without enforcing either $W_1 \xrightarrow{p} W2$ or $W_3 \xrightarrow{p} W_4$. (The relation $\xrightarrow{p}$ denotes the persist order). We observe that efficiency necessitates a buffered implementation in which persistency lags behind visibility [9, 21, 25]. Taking inspiration from *lazy release consistency* [23], a protocol from the DSM literature that enforces RC lazily, we propose *lazy release persistency* (LRP) for enforcing RP's one-sided barrier semantics lazily. In a nutshell, on an acquire LRP detects the matching release via the coherence protocol and persists W_1 and *Rel* from the release side (in that order) before performing the acquire, thereby enforcing $W_1 \xrightarrow{p} Rel \xrightarrow{p} W_4$.

1.1 Contributions

- We have argued thus far that languages must offer efficient ordering primitives, in addition to failure-atomicity, for supporting the important use case of LFDs.
- We observe that ARP's one-sided barriers—their semantics as well as implementation—are not strong enough to enable recovery in LFDs, which necessitates the inclusion of the relatively inefficient full barriers (§3).
- We propose strengthening ARP's one-sided barrier semantics. We argue that the resulting model, RP, enables correct recovery of LFDs upon a crash (§4).
- We propose LRP, a microarchitectural mechanism for efficiently enforcing RP's one-sided barriers (§5) .
- Our experiments on 5 commonly used LFDs suggests that LRP provides a 14%-44% (average 33%) performance improvement over the state-of-the-art full barrier, while enforcing RP (§6).

2 Background

In this section, we discuss persistency models with a focus on variants of (buffered) epoch persistency (§2.2). We then discuss log-free data structures (LFDs) and describe the actions required for their crash-recovery (§2.3). But before diving into persistency, we first discuss consistency since the two are closely intertwined.

Without loss of generality, the paper assumes a simple variant of Release Consistency with a total order on all memory events, similar to what is supported by the ARMv8 and RISC-V ISAs [2, 42]. To focus on our ideas, and not get bogged down by memory model intricacies, we assume that the language-level model is identical to the ISA-level model.

2.1 Release Consistency

A consistency model specifies how memory operations are globally ordered. This global memory order specifies what value a read must return: a read returns the value of the most recent write before it in the global memory order. Release Consistency (RC) [13] allows for reads and writes to be tagged as acquires and releases respectively. These acquires and releases have implicit one-sided barrier semantics. Specifically, memory operations that precede a release in program order appear before the release in the global order, while memory operations that follow an acquire in program order appear after the acquire in the global order. Furthermore, most consistency models support read-modify-writes (RMWs) which are essential for achieving synchronization [3, 17].

Below, we provide a simplistic RC memory model in which fences are omitted. We use the following notation for memory events:

- $\mathbf{M^i_x}$: a memory operation (of any type) to address x from (hardware) thread i. The operation can be further specified as a read: R^i_x, a write W^i_x or with an identifier (e.g., $M1^i_x$)
- $\mathbf{Rel^i_x}$: a release (release write or release-RMW) to address x from thread i.
- $\mathbf{Acq^i_x}$: an acquire (acquire read or acquire-RMW) to address x from thread i.

We use the following notation for ordering memory events:

- $\mathbf{M^i_x} \xrightarrow{\text{po}} \mathbf{M^i_y}$: M^i_x precedes M^i_y in program order.
- $\mathbf{M^i_x} \xrightarrow{\text{hb}} \mathbf{M^j_y}$: M^i_x precedes M^j_y in the global history of memory events, which we refer to as happens-before order ($\xrightarrow{\text{hb}}$).
- $\mathbf{Rel^i_x} \xrightarrow{\text{sw}} \mathbf{Acq^j_x}$: Rel^i_x *synchronizes with* Acq^j_x, i.e., Acq^j_x reads the value from Rel^i_x and $i \neq j$.

We formalize Release Consistency using the following rules:
> **Release one-sided barrier semantics.** A memory access that precedes a release in program order appears before the release in happens-before: $M^i_x \xrightarrow{\text{po}} Rel^i_y \Rightarrow M^i_x \xrightarrow{\text{hb}} Rel^i_y$.
> **Acquire one-sided barrier semantics.** A memory access that follows an acquire in program order appears after the acquire in happens-before: $Acq^i_y \xrightarrow{\text{po}} M^i_x \Rightarrow Acq^i_y \xrightarrow{\text{hb}} M^i_x$.
> **Program order address dependency.** Two memory accesses to the same address ordered in program order preserve their ordering in happens-before: $M1^i_x \xrightarrow{\text{po}} M2^i_x \Rightarrow M1^i_x \xrightarrow{\text{hb}} M2^i_x$.
> **Release synchronizes with acquire.** A release that synchronizes with an acquire appears before the acquire in happens-before: $Rel^i_y \xrightarrow{\text{sw}} Acq^j_y \Rightarrow Rel^i_y \xrightarrow{\text{hb}} Acq^j_y$.
> **RMW-atomicity axiom.** An RMW appears atomically (consecutively) in happens-before: $R^i_x \xrightarrow{RMW} W^i_x \Rightarrow R^i_x \xrightarrow{hb} W^i_x$ and there can be no memory operation from any thread M^j_y such that $R^i_x \xrightarrow{\text{hb}} M^j_y \xrightarrow{\text{hb}} W^i_x$.
> **Read value axiom.** A read to an address always reads the latest write to that address before the read in happens-before: if $W^j_x \xrightarrow{\text{hb}} R^i_x$ (and there is no other intervening write W^k_x such that $W^j_x \xrightarrow{\text{hb}} W^k_x \xrightarrow{\text{hb}} R^i_x$), the read R^i_x returns the value written by the write W^j_x.

2.2 Persistency Models

In a manner analogous to consistency models, Pelly et al. [34] introduce the notion of memory persistency, which specifies a global order in which writes can persist (i.e., *persist order*).
Persist notation. We use the following notation to denote that write W^i_x appears after write W^j_y in persist order: $W^i_x \xrightarrow{\text{p}} W^j_y$. To put it succinctly, W^j_y can persist only after W^i_x has persisted.
Buffered Epoch Persistency (BEP). BEP allows the programmer to place *persist barriers*, demarcating the program into *epochs* [9]. BEP then uses the epochs to enforce that for any two writes: W^i_x, W^i_y, if $W^i_x \xrightarrow{\text{po}} W^i_y$ and the writes belong to different epochs: e_k, e_l, then $W^i_x \xrightarrow{\text{p}} W^i_y$. BEP also involves an inter-thread component. When there is an inter-thread shared-memory dependency between two threads, the writes from the source epoch would have to be persisted before the writes from the target epoch. We note that BEP is a performance-oriented variant of the stricter epoch persistency (EP). BEP improves upon EP by decoupling persistency and visibility through buffering of writes.

2.2.1 Persist Barrier Implementations

BEP can be enforced via persist barriers. We classify prior work on persist barriers into two classes based on how writes are buffered: 1) *cache-based* implementations that use the hardware caches to buffer writes and 2) *persist-buffer-based* implementations that enqueue all writes in a global FIFO, called *persist buffer*.
Cache-based. Cache-based implementations [9, 21] buffer writes in the hardware caches and enforce epoch orderings by keeping track of each cache-line's epoch. Persisting a cache-line with epoch e_k triggers the persist of all currently buffered writes from older epochs. A persist is triggered upon a *conflict*; there are two types of conflicts.
> **Intra-thread conflicts.** There are two sources of intra-thread conflicts: 1) evicting a cache-line due to a demand access and 2) attempting a write with epoch e_k on a cache-line with an older epoch-id.
> **Inter-thread conflicts.** These are caused by inter-thread shared-memory dependencies. Such dependencies mandate

the enforcement of persist ordering between epochs of two different threads. Inter-thread conflicts can be resolved by blocking the target thread until the source epoch persists [9]. It is possible to enforce these in a lazy manner [21] although that comes with the complexity cost of avoiding deadlocks if there are cyclic dependencies.

Note that while conflicts often trigger multiple persists on the critical path of execution, state-of-the-art cache-based implementations mitigate this overhead through *proactive flushing* [21], a technique that starts flushing an epoch as soon as its execution completes.

Persist-buffer-based. Persist-buffer-based implementations [25, 30] buffer and order writes in per-thread queues that are added alongside the cache hierarchy. These queues drain into buffer(s) that are adjacent the NVM controllers. Together these buffers enforce the required intra- and inter-thread persist orderings. Delegated Persist Ordering [25] comes with a single buffer at the NVM controller and hence may enforce a global order amongst potentially independent epochs from two difference threads. HOPS [30] mitigates this with per-thread buffers alongside the NVM controllers.

LRP approach. The persist-buffer based approach arguably simplifies the design, avoiding the complexity of tracking conflicts inside the caches. On the other hand, the cache-based approach reuses the cache hierarchy for enforcing ordering. Both of these approaches are designed for enforcing full persist barriers; neither approach provides an obvious pathway for enforcing one-sided barriers. In this paper, we focus on the cache-based approach for enforcing the one-sided barriers of RP.

2.3 Log-free data structures (LFDs)

To ensure correctness, operations that modify a data structure must be atomic. Often atomicity is achieved through atomic code regions protected by locks. However, an important class of *nonblocking* data structures is designed to explicitly avoid locks. These data structures carefully bake their atomicity into a single instruction (typically a Compare And Swap, i.e., CAS). Collapsing the atomic region into a single instruction eliminates the need for locks [4, 19]. For instance, in the example of Figure 1, thread T0 first creates a node privately and then it atomically links the node with the linked list through a single CAS instruction.

Nonblocking data structures strive to avoid blocking; they avoid situations in which a thread is in a blocked state, unable to make progress on its own as it is waiting for other threads. As a happy consequence, these data structures also eliminate the need for explicit failure atomicity of a group of writes: since the atomicity is now incorporated into a single instruction, there is no longer need to persist multiple instructions atomically. Thus, nonblocking data structures are *log-free*, as they do not require the logging mechanisms that are typically associated with failure atomicity [10].

To ensure recovery on a crash, intuitively the NVM must be kept in a consistent state. What represents a consistent state for an LFD? Izraelevitz and Scott [19] show that LFDs can be recovered without any effort (i.e., *null recovery*), if what remains on the NVM after a crash is a consistent cut of the program's execution.

In this work, we enable null recovery for LFDs through an RC-based language-level persistency model called release persistency (RP). RP, being a language-level model, has to be compiled down to an ISA-level persistency model. Without loss of generality, we assume the ISA-level model is *identical* to the language-level model. (It is worth noting that our ISA model is similar in spirit to a recent proposal for persistency extensions to ARMv8 [41]). We then propose LRP, an efficient microarchitectural mechanism for implementing (the ISA-level) RP.

3 Limitations of ARP

Gotge et al. [14] argue that a persistency model based on *only* ordering (such as ARP) is unsatisfactory because, the lack of atomicity guarantees makes recovery cumbersome. While agreeing with Gotge et al. for general programs, we argue for ordering's utility for LFDs. Recall that LFDs can be recovered without any effort after a crash (i.e., null recovery), as long as the NVM reflects a consistent cut of the program's execution.

Design goal. In order to maximize performance, while allowing for null recovery, we set the following design goal: the persistency model must *precisely* mirror the consistency model, which in our case is RC. The key requirement to match RC semantics is treating releases and acquires as *one-sided persist barriers*, in the same manner as RC treats them as one-sided barriers.

Alas, ARP [24], the only RC-based persistency model with one-sided barriers falls short of that goal. In the rest of this section, we first describe ARP's semantics (§3.1) and implementation (§3.2), focusing on how ARP fails to achieve our design goal. We then motivate the need for a new persistency model that can rectify ARP's shortcomings (§3.3).

3.1 ARP semantics

ARP [24] is a language-level persistency model with explicit release and acquire annotations. Notably, these semantics comprise the *ARP-rule*, which is defined below.

ARP-rule. When a release synchronizes with an acquire, all writes that precede the release must persist before writes that follow the acquire:

$$W_y^i \xrightarrow{\text{po}} Rel_x^i \xrightarrow{\text{sw}} Acq_x^j \xrightarrow{\text{po}} W_z^j \Rightarrow W_y^i \xrightarrow{\text{p}} W_z^j$$

3.1.1 ARP semantics shortcomings

We note that the ARP-rule is not strong enough to mirror the happens-before order of RC. Thus, it is unable to provide

null recovery as it does not preserve a consistent cut in the NVM. For instance, RC mandates that if a release is visible, all preceding writes must be visible too, i.e., $W_y^i \xrightarrow{\text{po}} Rel_x^i \Rightarrow W_y^i \xrightarrow{\text{hb}} Rel_x^i$. However, ARP allows for a release to persist before all preceding writes have persisted; i.e., $W_y^i \xrightarrow{\text{po}} Rel_x^i \not\Rightarrow W_y^i \xrightarrow{\text{p}} Rel_x^i$.

Going back to the example of Figure 1b, in the event of a crash, it may be the case that the release of thread T0, which links a new node into the linked list, has persisted, but the preceding writes that created the node have not persisted. This would leave the linked list in an inconsistent, and hence unrecoverable state.

3.2 ARP implementation

ARP is implemented on top of a persist-buffer based implementation, originally designed to enforce the RCBSP [25] model. RCBSP is a persistency model that is based on the ARMv7 consistency model and hence involves full persist barriers. Thus, in the original RCBSP implementation, on executing a persist barrier, the buffer's epoch is incremented so that subsequent writes coming after the barrier are ordered after those before the barrier.

For enforcing ARP, however, releases and acquires need not be treated as full persist barriers: writes that precede a release need not be ordered before writes that follow the release; and writes that follow an acquire need not be ordered after writes that precede the acquire. The ARP implementation enhances RCBSP hardware to leverage this observation.

Thus, on a release, no barrier is placed; rather a flag is raised denoting that the next acquire must place a persist barrier. On an acquire, a persist barrier is placed only if the flag is found raised. These additions enforce the ARP-rule as follows: if a release Rel_x^i is inserted into the buffer before an acquire Acq_y^j, then any write that precedes Rel_x^i must belong to an older epoch than any write that follows Acq_y^j, thus ensuring that writes that precede a release persist before writes that follow an acquire.

3.2.1 ARP implementation shortcomings

First, we note that the ARP implementation precisely enforces the ARP-rule *without* mirroring RC. For instance, a write that precedes a release is likely to belong to the same epoch as the release, and could thus persist *after* the release.

Second, even though the ARP authors identify that maximizing performance hinges on enforcing one-sided persist barriers, their implementation still uses full persist barriers. (This is because their implementation builds on top of RCBSP hardware). The lack of one-sided barriers in the implementation makes it impossible to parallel the RC semantics: on the one hand, when the barrier is elided (i.e., on a release) the implementation fails to match the RC semantics, while on the other hand, when the barrier is placed (i.e., on an acquire) the implementation provides more orderings than RC requires.

3.3 Why not simply fix ARP?

It is possible to honor RC semantics on top of ARP (and thus enable null recovery of LFDs) as long as a persist barrier is placed before every release. Going back to the linked list example, a persist barrier placed before the release (Figure 1d) ensures that the new node would persist before the link (i.e., the CAS) persists.

Recall, however, that the design goal is to not only enable null recovery of LFDs, but also to maximize performance through one-sided persist barriers. Placing a full persist barrier before every release falls short of achieving this goal. Aggravating the problem, the persist-buffer-based implementation of ARP pertains solely to full persist barriers, making it impossible to provide the desired one-sided persist semantics.

Therefore, it is clear that there is a need for a new persistency model and its efficient implementation, built from the grounds up to provide efficient null recovery for LFDs.

4 Release Persistency

In this section, we introduce Release Persistency (RP), a persistency model that reconciles the performance of one-sided persist barriers with the stronger semantics that is required for the recovery of LFDs. First, we formally specify RP (§4.1) and then we discuss the performance implications of our specification (§4.2).

4.1 Formal Specification

RP must ensure that the persist order reflects the RC happens-before order, which we formally specified in §2.1, for enabling crash recovery. Note that because the persist order defines the order in which writes persist, only those RC rules that pertain to writes must translate into the RP formalism. Therefore, RP can be succinctly formalized as follows. Any two writes in the RC happens-before order must also persist in that order: $W1_x^i \xrightarrow{hb} W2_y^j \Rightarrow W1_x^i \xrightarrow{p} W2_y^j$
From the happens-before rules of §2.1, we can expand and specify RP via the following rules:
> **Release one-sided barrier semantics.** A write that precedes a release in program order appears before the release in persist order: $W_x^i \xrightarrow{\text{po}} Rel_y^i \Rightarrow W_x^i \xrightarrow{\text{p}} Rel_y^i$.
> **Acquire one-sided barrier semantics.** A write that follows an acquire in program order appears after the acquire in persist order: $Acq_y^i \xrightarrow{\text{po}} W_x^i \Rightarrow Acq_y^i \xrightarrow{\text{p}} W_x^i$.
> **Release synchronizes with acquire.** A release that synchronizes with an acquire appears before the acquire in persist order: $Rel_y^i \xrightarrow{\text{sw}} Acq_y^j \Rightarrow Rel_y^i \xrightarrow{\text{p}} Acq_y^j$.
> **Program order address dependency.** Two writes to the

same address ordered in program order preserve their ordering in persist order: $W1_x^i \xrightarrow{po} W2_x^i \Rightarrow W1_x^i \xrightarrow{p} W2_x^i$.

> **RMW-atomicity axiom.** Read and write of an RMW appear consecutively in persist order: $R_x^i \xrightarrow{RMW} W_x^i \Rightarrow R_x^i \xrightarrow{p} W_x^i$ and there is no write W_y^j (from any thread) such that $R_x^i \xrightarrow{p} W_y^j \xrightarrow{p} W_x^i$.

A note on RP acquires. Because an acquire being a read cannot persist, the $Acq_y^i \xrightarrow{p} W_x^i$ ordering may appear bizarre at first. The intention here is to allow for two or more rules linked by an acquire to apply transitively.

For example, when a release synchronizes with an acquire, the released value must persist before any of the writes following the acquire persist. This is captured by applying the "release synchronizes with acquire" rule and the "acquire one-sided barrier semantics" rule transitively.

In a similar vein, when a release synchronizes with an RMW marked acquire: (1) the released value must first persist; (2) then the value written by the RMW must persist (follows from the RMW atomicity axiom that mandates that read and write must appear consecutively in persist order); and finally (3) writes following the RMW must persist.

4.2 RP reduces conflicts

In Section 2.2.1, we discussed the conflicts that can occur in a cache-based implementation of a BEP model. Conflicts can adversely impact performance because they can trigger the persist of entire epochs in the critical path. RP, with its one-sided barriers, substantially reduces the number of conflicts that need to be handled, as will see next with an example.

Consider Figure 2a which shows two writes, W_A and W_B, on different epochs e_0 and e_1 respectively, segregated by a full persist barrier that immediately precedes the release $rel : W_{1,0}$. Even though RC allows for W_B to become visible before W_A, the full persist barrier mandates that W_B cannot persist before W_A has persisted. Therefore, if the cache-line written by W_B were to be evicted, it would in turn trigger the persist of W_A in the critical path. Making matters worse, if writes W_A and W_B are to the same cache-line, the mere act of performing the write W_B triggers the persist of W_A. (In order to ensure that writes persist in epoch-order, we must ensure that a dirty cache-line contains writes of the same epoch.)

These conflicts are eliminated when placing a one-sided barrier instead, as in Figure 2b. The one-sided barrier captures the exact semantics of RC, allowing W_B to be persisted out-of-order. In the special case where W_A and W_B are to the same cache-line, the one-sided barrier enables the coalescing of writes to the same dirty cache-line. Similarly, in Figure 2c, one-sided persist barriers capture the exact semantics of an acquire: the persist of W_B does not trigger the persist of W_A.

In summary, with one-sided barriers, persisting a write does not automatically trigger the persists of writes from previous epochs. This relaxation substantially reduces the number of both inter- and intra- thread conflicts including: conflicts due to writing to a cache-line with an older epoch, conflicts triggered by evicting a cache-line and conflicts due to forwarding a cache-line upon receiving a coherence request. Furthermore, eliminating these conflicts could allow for significant coalescing of writes to the same cache-line, thus potentially reducing the absolute number of persists. We hypothesize that all of this has a significant impact on performance. Our evaluation (§6) vindicates this hypothesis.

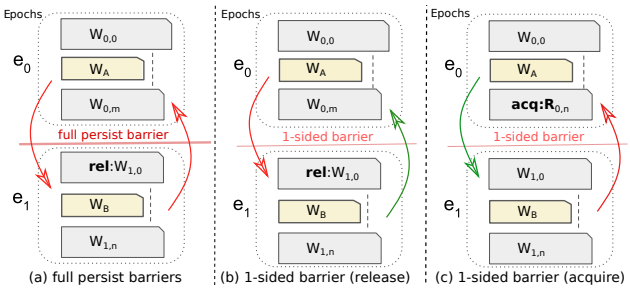

Figure 2: RP's one sided persist barrier allows for W_B to be persisted before W_A and this reordering could significantly reduce intra- and inter-thread conflicts.

5 Lazy Release Persistency

In this section, we present lazy release persistency (LRP), our microarchitectural mechanism for enforcing RP.

Big Picture. As discussed in §2.2.1, a buffered implementation where visibility does not wait for persistency is critical for minimizing persistency-related overheads. On the one hand, this calls for a mechanism that *maximizes* buffering. On the other hand, too much buffering can pose problems.

Recall that RP mandates the enforcement of inter-thread persist orderings when a thread synchronizes with another. Allowing for buffering to extend across threads incurs complexity in the form of coordination across memory controllers for enforcing inter-thread persist dependencies. It also involves complex deadlock-avoidance mechanisms to eliminate potential cyclic inter-thread dependencies [21][1].

Therefore, we make the design choice of buffering persists within a thread until an inter-thread dependency is detected, at which point we persist the buffered writes. We will see later that our experimental evaluation (§6.4) vindicates this choice.

How to realize these design requirements? Our insight here is that the requirements matches that of lazy release consistency (LRC) [23], a protocol first proposed in the context of DSMs for enforcing RC lazily.

Specifically, LRC buffers writes to shared data locally, past a release operation, until the time of the corresponding

[1] Although DRF programs do not pose a deadlock risk, the hardware must be able to handle racy programs as well

acquire. At this point, the releasing processor makes the buffered writes globally visible, thereby enforcing RC.

Taking inspiration from LRC, we propose LRP, a protocol for enforcing RP and its one-sided barriers. In LRP, writes to the L1 are simply buffered and do not trigger persists. When an inter-thread synchronization is detected, i.e., when a release synchronizes with an acquire, all of the cache-lines written before the release (and the release) are persisted before the acquire is performed, thereby enforcing RP.

Next, we provide a high-level overview of LRP and how it satisfies the RP semantics (§5.1). We then dive into the specifics of our implementation (§5.2).

5.1 LRP Overview

In this section we provide a high-level overview by outlining the invariants satisfied by LRP. We informally argue for correctness by reasoning that the invariants are sufficient to enforce RP. In the next section, we discuss the detailed microarchitecture, explaining how LRP enforces the invariants.

LRP uses a buffering-based approach where persistency trails visibility. Therefore, writes to the L1 do not trigger persists. Instead, whenever a dirty cache-line is written back from the L1 (owing to a cache-line eviction or a downgrade), the cache-line is persisted by the LLC/directory controller. LRP ensures that these persists enforce RP via ensuring four key invariants:

- **Invariant-1 (I1):** When the L1 controller receives an eviction request for a cache-line written by a release, it blocks the request until all of the cache-lines written by writes prior to the release have been persisted.
- **Invariant-2 (I2):** When the L1 controller receives a downgrade request for a cache-line written by a release from the directory, the request is blocked until: (a) all of the cache-lines written by writes prior to the release have been persisted; and (b) the release has been persisted.
- **Invariant-3 (I3):** When an RMW, marked acquire, is successful (i.e., if its write is successful), the acquire blocks the pipeline until the write of the RMW persists.
- **Invariant-4 (I4):** When the directory controller receives a write-back from the L1, the directory persists the cache-line, blocking requests for the cache-line until it persists.

We now argue that the four invariants are sufficient to enforce RP's persistency rules.

> **Release one-sided barrier semantics.** Invariant-1 ensures that before a release is allowed to persist, all previous writes have been persisted.

> **Release synchronizes with acquire.** Suppose a release from thread T1 synchronizes with an acquire from thread T2 and issues a read request for the cache-line. There are three cases.

- Case-1: The acquired cache-line is in M(odified) state in T1's L1. In this case, T2's acquire would cause a coherence request (downgrade) to be sent to T1. Invariant-2 ensures that the acquire will block until the release and its preceding writes have been persisted, thereby ensuring this rule.
- Case-2: The acquired cache-line is in the LLC. Invariant-1 ensures that all writes before the release would have persisted. Invariant-4 ensures that the release itself would have persisted.
- Case-3: The acquired cache-line is in NVM. This implies that the release has already persisted. Invariant-1 ensures that all writes before the release also would have persisted.

> **Acquire one-sided barrier semantics.** This comes naturally out of the consistency model. Any store following the acquire cannot perform (and hence cannot persist) before the acquire performs.

> **Program order address dependency.** This again comes naturally. Since writes coalesce, it is impossible for two writes to the same variable to be persisted out-of-order.

> **RMW-atomicity axiom.** The only interesting case is when the RMW is marked an acquire. Invariant-3 ensures that the write of the RMW persists before following writes, thereby ensuring this rule.

An example. Consider the required semantics of Figure 1d: T0's W1 must persist before T0's Rel persists, and T0's Rel must persist before T1's W4. RP fulfills the requirements as follows. Let us assume that when T1's Acq performs, the cache-line is held in T0's L1. Therefore, the Acq will trigger a downgrade request for the block and hence from Invariant-2, T1's Acq will complete only after triggering the persist of T0's Rel and its previous writes (W1). Finally, T1's W4 cannot be issued to the memory system until T1's Acq completes, thereby ensuring $W1 \xrightarrow{p} Rel \xrightarrow{p} W4$.

5.2 LRP: Microarchitecture

We have established that LRP enforces the RP rules by upholding four invariants (I1-I4). I1 and I2 pose a significant microarchitectural challenge: on evicting/downgrading a released cache-line, all prior writes must be tracked and persisted. Conversely, I3 is trivially implemented by altering the processor pipeline to wait for an ack from the NVM controller on an RMW-acquire. I4 requires a minor alteration in the directory controller which we discuss more elaborately in §5.2.3.

Therefore, this section focuses on I1 and I2, presenting a mechanism that, upon evicting/downgrading a release, can scan the L1 cache and persist all prior writes. We note that the mechanism does not extend beyond the L1 cache and thus can be simply implemented by enhancing the L1 controller, the L1 cache and the processor core. We begin by discussing the required hardware extensions.

5.2.1 Hardware extensions

Figure 3 illustrates the LRP hardware extensions, which are described one by one below.

Per thread metadata. Each (hardware) thread maintains an *epoch-id* counter which gets incremented on every release. In addition, the number of pending persists are denoted by a *pending-persists* counter. Upon issuing a persist for *any* write, the pending-persists counter gets incremented; upon receiving an acknowledgment from the NVM controller for *any* of the issued persists, the pending-persists counter gets decremented. The pending-persists counter allows a persisting release to ensure that all previous writes have persisted.

Per L1-cache-line metadata. Each cache-line maintains: (1) a *min-epoch*, that holds the epoch of the earliest write to the cache-line and (2) a *release-bit*, that denotes whether the cache-line holds a value written by a release.

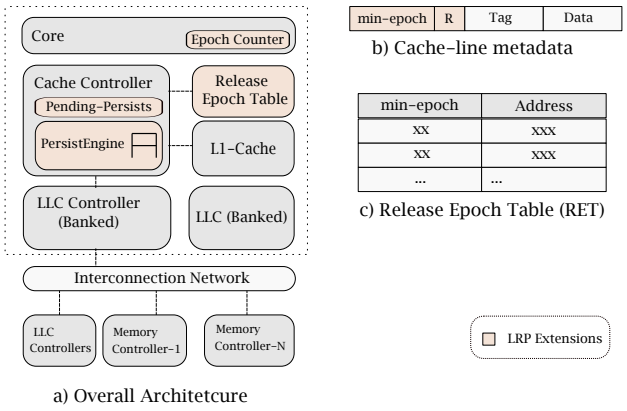

Figure 3: Hardware extensions involved in LRP

Release Epoch Table (RET). A small content-addressable table, called *Release Epoch Table* (RET), holds the *release-epoch* of cache-lines that holds a value written by a release. We note that it is possible to maintain a release-epoch for every L1 cache-line. However, we expect that at any given moment only a handful of cache-lines will hold values written by a release. This is because in most programs variables that are released account for a small percentage of the program's working dataset. Through our experiments, we have found that a 32-entry RET for each L1 cache adequately over-provisions for the needs of most programs. On executing a release, a RET entry is allocated, storing the release-epoch and the cache-line's address. When a release persists, its respective entry in the RET is squashed. To avoid filling the RET when the capacity reaches a watermark, the persist of the oldest release in the RET is triggered.

Persist engine. The *persist engine* is an FSM that takes as an input a release-epoch e_{rel} and scans the L1 cache, examining all cache-lines and persisting every cache-line with a smaller min-epoch than e_{rel}.

Hardware Overhead. We assume a 32KB L1 cache with 40-bit tags. LRP adds an 8-bit min-epoch and a release-bit to each cache-line, amounting to 576 bytes for the entire L1. Note that when the epoch-id overflows, all not-yet-persisted cache-lines of L1 are persisted and the epochs are restarted. In addition, each of the 32 RET entries stores the physical address of a cache-line (i.e., 40 bits, same as the L1 tag) plus a release-epoch (8-bit), amounting to 192 bytes for the entire RET. In total, LRP requires less than 1KB per hardware thread.

5.2.2 A mechanism to enforce the release barrier

Having described the hardware extensions in each L1 controller, we now discuss how these extensions are leveraged to ensure that persisting a released cache-line triggers the persists of all writes of previous epochs. We then use this mechanism to enforce invariants I1 and I2. But first, we establish some necessary terminology.

Terminology. If a cache-line holds a not-yet persisted write, the cache-line must reside in L1 in modified (M) coherence state. If a cache-line is in M state and has its release-bit set, it implies that the cache-line holds a value written by a release; we refer to such cache-lines as *released*. If the cache-line is in M state but its release-bit is not set, it implies that the cache-line holds a value written by regular writes only; we refer to such cache-lines as *only-written*. If a cache-line is neither released nor only-written, we refer to it as *clean*.

On a write. On performing a regular write, if the cache-line is clean, the thread's epoch-id is stored in the cache-line's min-epoch. If the cache-line is not clean, the write need not overwrite the cache-line's min-epoch, as the cache-line already has a valid min-epoch that is smaller than the current thread's epoch.

On a release. On a release, the thread's epoch-id is incremented; the new epoch will be the release-epoch, ensuring that all writes that precede the release are in an earlier epoch. There are two distinct cases for the state of the cache-line that the release intends to write. (1) Clean: the release assigns its epoch to the cache-line's min-epoch, it sets the cache-lines release-bit and it allocates a new entry in RET, where it also stores its release-epoch. (2) Not clean: the cache-line is first persisted and then treated as clean (i.e., case (1)). Note that case (2) implies that the release cannot be coalesced in the same cache-line with any previous write.

On a read/acquire. No additional action is necessary on a read or on an acquire.

On an RMW-acquire. An RMW marked acquire blocks the pipeline until its write is persisted. Beyond this, additional action is not necessary.

On downgrading a cache-line. Attempting to downgrade a cache-line from M state triggers its persist. If the cache-line is only-written, then the persist happens off the critical path. But, if the cache-line is released, then the downgrade cannot complete before the cache-line has persisted.

On evicting a written/released cache-line. Evicting a cache-line that is written but not released has the same effect as downgrading it. If the cache-line is released, then the eviction triggers its persist, but need not wait for the persist to complete (i.e., the persist is off the critical path).

Note that there is a subtle distinction between evicting and downgrading a released cache-line: while both actions cannot complete unless all previous writes/releases persist, downgrading also requires that the released cache-line itself persists; there is no such requirement for evicting. To simplify the rest of the discussion, we refer to both downgrading and eviction as the act of persisting a released cache-line. To enforce the eviction invariant (i.e., I1) we do not wait for an ack from the NVM controller for the released cache-line, while to enforce the downgrade Invariant (i.e., I2), we wait for the ack.

On persisting a released cache-line. First, the RET is accessed to read out the release-epoch e_{rel} of the cache-line. The e_{rel} along with the address of the cache-line are propagated to the persist engine, which begins scanning the L1 cache, discovering all only-written/released cache-lines with min-epoch smaller than e_{rel}. The persist engine must issue a persist for all discovered cache lines, but there is a catch: amongst the discovered cache-lines, there may exist a released cache-line CL_r with epoch e_k and a written cache-line CL_w with epoch e_{k-1}, for which the release one-sided barrier semantics mandate that CL_w must persist before CL_r.

Figure 4 illustrates this case. When attempting to persist the Release ($F2$), the persist engine tracks down all only-written/released cache-lines of previous epochs. One of the tracked cache-lines will be the released CL_c which holds the Release ($F1$) and the only-written CL_d which holds the Write (X). The one-sided persist barrier semantics of the release mandate that the only-written CL_d must persist before the released CL_c.

Persist engine algorithm. The persist engine achieves this ordering by persisting first all the only-written cache-lines and then persisting the released cache-lines in their epoch order. Specifically, the persist engine operates as follows: as the persist engine scans the L1 cache it keeps discovering cache-lines that must be persisted; on discovering an only-written cache-line, it immediately schedules its persist, incrementing the pending-persists counter. Otherwise, on discovering a released cache-line, it simply buffers it in a local queue inside the persist engine. In either case, the engine immediately resumes scanning the L1 cache.

After the scanning completes, the engine starts polling on the pending-persist counter, waiting for it to become zero. Recall, that the pending-persist counter gets decremented every time an ack from the memory controller reaches the L1, denoting that a pending persist has completed. When the pending-persist counter reaches zero, the persist engine infers that all scheduled persists have completed, and thus it can start scheduling the persists of the released cache-lines.

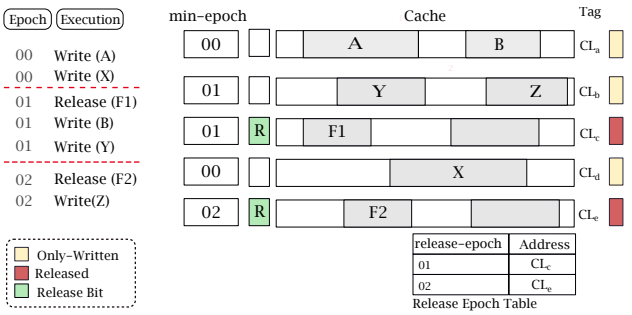

Figure 4: The state of the cache after an example execution.

For instance, in the example of Figure 4, the persist engine first persists the written cache-lines Cl_a, CL_b, and CL_d; then, and only after these cache-lines have persisted, the persist engine will first persist CL_c that holds Release (F1) and then CL_e that holds Release (F2).

Enforcing invariants I1 and I2. The persist engine algorithm enforces both I1 and I2 by persisting a release, with one simple distinction: for I1 (i.e., release eviction) the persist engine does not wait for the released cache-line to be acked by the NVM memory controller, whereas for I2 (i.e., release downgrade) it waits for the released cache-line to be acked.

Persist engine correctness. The persist engine essentially reorders the persist of writes with the persist of *prior* releases, while ensuring that releases persist in their epoch order. This reordering does not violate RP, which only mandates that writes be ordered before a *subsequent* release.

5.2.3 Coherence controller

LRP involves modest (local) changes to the L1 coherence controller. Specifically, a downgrade request (e.g. a Fwd-GetS request) for a released cache-line in M state could block until previous writes in the L1 (if any) persist. It is important to note that this does not pose a deadlock risk since the persist actions are guaranteed to complete without themselves being blocked.

Thus far, we have discussed a stalling implementation where the cache controller blocks on a Fwd-GetS. Stalling is not fundamental to our technique. It is possible to avoid this stalling by moving to a transient state upon a Fwd-GetS, which logically moves the state to S, waiting on an acknowledgement from the persist engine to move back to a stable state. However, we have not experimented with this non-stalling variant yet.

LRP also involves a minor change to the directory controller. Upon an L1 eviction of a released cache-line, a PutM request is sent to the directory. Normally, the directory would immediately transition to S state. However, the directory now enters a transient state that would block coherence requests to (only) that block until it receives a persist acknowledgement. Note that this does not stall the directory controller and hence is not expected to affect its performance.

6 Experimental Evaluation

Thus far, we have established that RP must be enforced for enabling recovery of LFDs. We conducted experiments seeking to answer two main questions. First and foremost, how much does our one-sided barrier mechanism (LRP) improve on the state-of-the-art full barrier when enforcing RP? Second, how much performance overhead does enforcing RP incur over a volatile execution that provides no persistency guarantees? Before we go to the results, we first discuss our workloads and methodology.

6.1 Workloads

LFDs are essentially nonblocking data structures with persist barriers inserted for ensuring crash recovery. We obtained 4 of our workloads from the SynchroBench suite [15], which is a collection of nonblocking data structures. Specifically, we used the linkedlist [16], hashtable [28], binary search tree (balanced tree) [32] and skip-list [44] workloads. We also implemented the lock-free queue from Michael and Scott [29]. All workloads are data-race-free in that synchronization operations are properly labelled using releases and acquires. For each workload, we use a harness that creates 1–32 workers and issues inserts and deletes at 1:1 ratio. Since we only use insert and delete operations, the *update-rate* of the benchmark suite is 100%. The data structure size refers to the initial number of nodes in the data structure before statistics are collected: we vary the size from 8K entries–1M entries, and the default value is 64K entries.

6.2 Comparison Points

We compare LRP against alternative methods for enforcing RP using full barriers. We also compare against volatile execution.

LRP. This represents our approach for enforcing RP. Releases and acquires are automatically treated as one-sided barriers and perform the actions described in §5.

SB. This represents an RP enforcement approach using a strict full barrier (SB). SB blocks until all the cache lines modified by the writes before the barrier have been persisted. SB also has an inter-thread component; when a shared memory dependency is detected via the coherence protocol, the target thread blocks until the writes in the ongoing epoch of the source thread have persisted. Therefore, in order to enforce RP: (1) an SB has to be inserted before each release to ensure that all writes before the release persist before the release; (2) an SB also has to be inserted after the release to ensure that inter-thread persist ordering is captured. (I.e., to ensure that when a release synchronizes with an acquire, the acquire correctly blocks until the release persists.)

BB. This represents an RP enforcement approach using the the state-of-the-art full barrier [21]. As discussed in §2, the barrier enforces the persist orderings (both intra-thread and inter-thread) similarly to SB, but employs an efficient buffered implementation that minimizes blocking. Hence, we refer to it as buffered barrier (BB). In order to enforce RP: (1) a BB has to be inserted before each release to ensure that all writes before the release persist before the release; (2) a BB has to be inserted after the release and before the acquire for capturing the inter-thread persist ordering (I.e., to ensure that when a release synchronizes with an acquire, all writes following the acquire persist after the release persists).

NOP. Finally, we also compare against volatile execution which does not enforce any persistency model (NOP).

Processor	64-Core (out-of-order) 2.5 GHz
ISA	Intel x86-64
L1 I+D -Cache (pvt.) line-width	32KB, 2 cycles, 8-way 64B
L2 (NUCA, shared)	1MB x64 tiles, 16-way 30 cycles
On-chip Network	2D-Mesh 32 bit flits and links
Coherence	Directory-based, MESI
NVM (PCM)	cached mode: 120 cycles uncached mode: 350 cycles
RET (private)	32 Entries

Table 1: Simulator Configuration

6.3 Simulator

Our hardware implementation[2] is built on top of the pin-based [27] PRiME [12] simulator, with 64 out-of-order-cores processor (single thread per core), a logically shared LLC and multiple memory controllers. Table 1 shows the details of the simulated processor and memory system.

We model NVM latencies based on the performance measurements observed on Intel Optane persistent memory [20]. Specifically, there are two modes which determine the NVM latency. In the *cached mode*, an NVM writeback persists as soon as it is written to a battery-backed NVM-side DRAM cache. In the uncached mode, an NVM writeback persists only after it is actually written to the NVM. We assume the faster cached mode for our experiments unless specified otherwise.

PRiME only supports x86-64 ISA and hence enforces the TSO (Total-Store-Order) consistency model. As such the simulator lacks releases and acquires in its ISA. Therefore, we implemented a simple extension to the ISA for taking in release/acquire annotations. We make use of Pin's capability to instrument the binary and generate these special stores and loads with release/acquire annotations corresponding to releases and acquires in the program.

It is worth noting that we did not alter the simulator's consistency enforcement mechanism to take advantage of the release/acquire annotations. (This is sound because TSO

[2] https://github.com/dananjayamahesh/lrp

Figure 5: Execution time normalized to No-Persistency (lower the better).

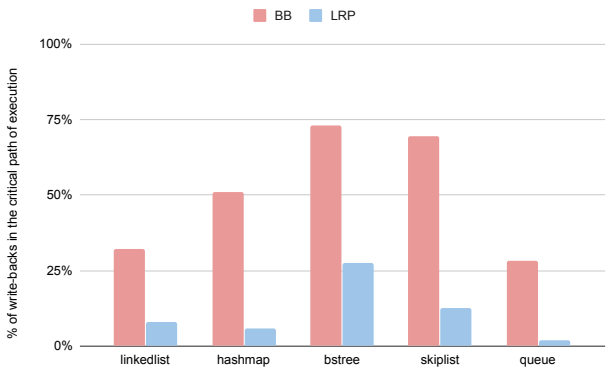

Figure 6: Percentage of write backs in the critical path (lower the better).

stores and loads already have release and acquire semantics respectively.) However, we take advantage of these annotations to implement our LRP mechanisms in order to enforce RP.

6.4 Results

LRP outperforms BB and SB. Figure 5 shows the execution times of LRP, BB and SB normalized to NOP with 32 worker threads and 64K elements. We first observe that BB outperforms SB, showing a 24%-68% (average 52%) improvement over BB. This is primarily because BB, which is a buffered implementation, avoids stalls in the critical path. This vindicates our design decision of striving for a buffered implementation for enforcing RP. How does LRP stack up against BB? Our key result is that LRP significantly outperforms BB, showing a 14%-44% (average 33%) improvement over BB.

LRP is 2%-8% within NOP. Figure 5 also reveals that LRP is only 2%-8% (average 6%) within volatile execution, which suggests that the persistency-related overheads incurred by LRP is nominal for these workloads.

Why LRP outperforms BB? Recall that the expected advantage of LRP over BB is that it significantly minimizes intra-thread persistency overheads being a one-sided barrier. On the other hand, BB is expected to incur lesser inter-thread persistency overhead; this is because, whereas LRP blocks on an acquire to enforce the inter-thread persistency orderings, BB enforces those lazily well. To understand why LRP outperforms BB, we conducted experiments to study the effect of intra- vs inter-thread persistency overheads of LRP and BB.

In Figure 6, we classify write backs into two categories: those that are in the critical path of the execution (of the processor doing the write back) and those that are not. For BB, a significant 51% of the write backs are in the critical path, whereas for LRP only 10% of the write backs are in the critical path. Since almost all of the write back are due to

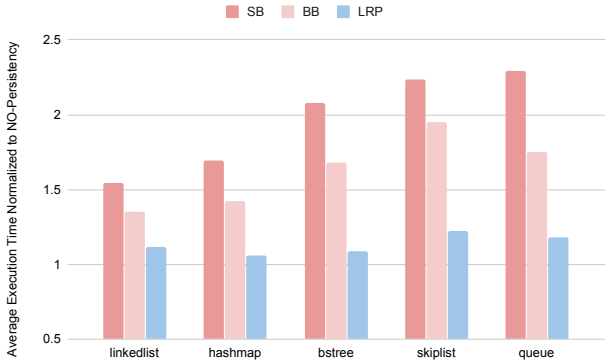

Figure 7: Execution time normalized to No-Persistency in the Uncached mode (lower the better).

persistency orderings, this suggests that LRP significantly minimizes intra-thread persistency overheads in comparison with BB.

Figure 8 compares the normalized execution time overheads of LRP vs BB as the number of worker threads are varied from 1–32. Greater the number of threads, greater the probability of inter-thread conflicts and hence potentially higher inter-thread persist ordering overhead for LRP. However, as seen in Figure 8, this effect is nominal: for LRP the persistency overhead remains relatively flat with increasing threads. For BB there is a marginal decrease in performance overhead as the number of threads is increased.

The above two experiments suggests that the effect of intra-thread persistency overhead far outweighs the effect of inter-thread persistency overhead. Therefore, this vindicates the design choice of LRP in seeking to optimize the intra-thread overheads vs inter-thread overheads.

Individual workload analysis. Whereas LRP consistently outperforms BB, as we can see from Figure 5, the gap between LRP and BB varies. One trend we observe is that, for read-intensive workloads the gap between LRP and BB is smaller

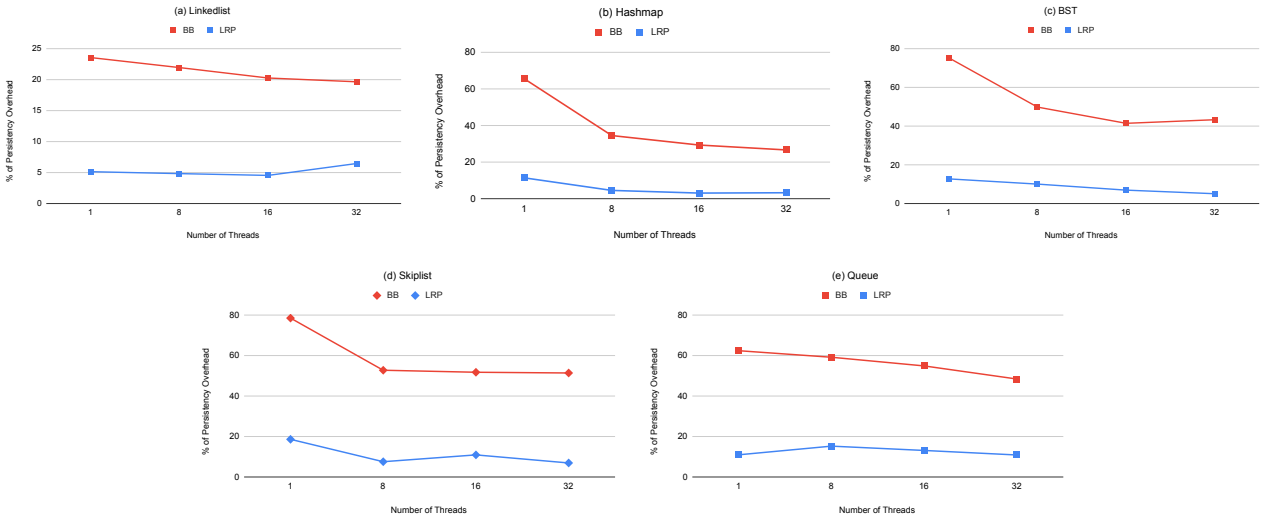

Figure 8: Percentage overhead over and above No-Persistency, varying the number of threads from 1 through 32 (lower the better).

than for write-intensive ones. As discussed earlier, BB suffers from intra-thread conflicts and these are more pronounced for write-intensive workloads. Thus, we can observe that linkedlist, a read-intensive workload owing to read-heavy link traversals, shows a lesser 14% gain over BB compared to BST, a write-intensive workload which shows a relatively higher 41% gain.

Cached vs Uncached mode. Recall that up until now we assumed the cached mode where a write back is said to persist as soon as it reaches the NVM-side DRAM cache. In this experiment, we consider the uncached mode by disabling the NVM-side DRAM cache, thereby exposing the slower NVM to applications. Figure 7 presents the normalized execution time overhead over NOP on the uncached mode. As we can see, and comparing with the results on the cache mode shown in Figure 7, LRP is more robust to this change when compared with BB or SB. LRP continues to incur a nominal 3%-19% (average 10%) overhead compared to NOP. BB (and SB) are affected more by this change because they have more writebacks in the critical path when compared to LRP. Thus, LRP shows a significant 53% improvement over BB in this configuration.

Sensitivity to data structure size. In order to measure the sensitivity of LRP to data structure size, we varied the size from 8K–1M nodes. However, we did not observe a significant change in the results (and hence we do not show these results). Changing the number of elements in the data structure largely affects inter-thread conflicts compared to intra-thread. Our observation is that even though the number of inter-thread conflicts changes with data structure size, it does not affect the execution time overheads significantly because, as established earlier, the effect of intra-thread conflicts are more significant.

7 Conclusion

We have argued that languages must support ordering primitives that are strong enough to enable recovery of log-free data structures (LFDs) without compromising on efficiency. Specifically, a release (and the writes before it) must persist before the writes that follow the corresponding acquire persist. We formalize this ordering requirement via a persistency model called release persistency (RP). The challenge is to realize RP via one-sided barriers, while also retaining a buffered implementation where visibility does not wait for persistency.

We addressed this challenge by taking inspiration from lazy release consistency, a protocol from the DSM literature that enforces release consistency lazily. Our proposed mechanism dubbed lazy release persistency (LRP) buffers writes in the cache until an acquire is detected, at which point the buffered writes are persisted. Experiments on 5 commonly used LFDs suggest that LRP efficiently enforces RP, significantly improving upon the state of the art.

Acknowledgments

We thank Aasheesh Kolli, Antonios Katsarakis, Adarsh Patil and the anonymous reviewers for their valuable feedback. This work was supported by EPSRC grant EP/L01503X/1 to the University of Edinburgh.

References

[1] Mohammad Alshboul, James Tuck, and Yan Solihin. 2018. Lazy Persistency: A High-Performing and Write-Efficient Software Persistency Technique *(ISCA '18)*. IEEE Press, 439–451.

[2] ARM Limited 2018. *ARM Architecture Reference Manual ARMv8, for ARMv8-A architecture profile.* ARM Limited.

[3] Hagit Attiya, Rachid Guerraoui, Danny Hendler, Petr Kuznetsov,

Maged M. Michael, and Martin Vechev. 2011. Laws of Order: Expensive Synchronization in Concurrent Algorithms Cannot Be Eliminated *(POPL '11)*. ACM, New York, NY, USA, 487–498.

[4] Guy E. Blelloch, Phillip B. Gibbons, Yan Gu, Charles McGuffey, and Julian Shun. 2018. The Parallel Persistent Memory Model *(SPAA '18)*. ACM, New York, NY, USA, 247–258.

[5] Hans-J. Boehm. 2012. Can Seqlocks Get Along with Programming Language Memory Models? *(MSPC '12)*. ACM, New York, NY, USA, 12–20.

[6] Hans-J. Boehm and Dhruva R. Chakrabarti. 2016. Persistence Programming Models for Non-volatile Memory *(ISMM 2016)*. ACM, New York, NY, USA, 55–67.

[7] Dhruva R. Chakrabarti, Hans-J. Boehm, and Kumud Bhandari. 2014. Atlas: Leveraging Locks for Non-volatile Memory Consistency *(OOPSLA '14)*. ACM, New York, NY, USA, 433–452.

[8] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. 2011. NV-Heaps: Making Persistent Objects Fast and Safe with Next-generation, Non-volatile Memories *(ASPLOS XVI)*. ACM, New York, NY, USA, 105–118.

[9] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. 2009. Better I/O Through Byte-addressable, Persistent Memory *(SOSP '09)*. ACM, New York, NY, USA, 133–146.

[10] Tudor David, Aleksandar Dragojević, Rachid Guerraoui, and Igor Zablotchi. 2018. Log-Free Concurrent Data Structures. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 373–386. https://www.usenix.org/conference/atc18/presentation/david

[11] Michal Friedman, Maurice Herlihy, Virendra Marathe, and Erez Petrank. 2018. A Persistent Lock-free Queue for Non-volatile Memory *(PPoPP '18)*. ACM, New York, NY, USA, 28–40.

[12] Yaosheng Fu and David Wentzlaff. 2014. PriME: A parallel and distributed simulator for thousand-core chips. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 116–125.

[13] Kourosh Gharachorloo, Anoop Gupta, and John L. Hennessy. 1991. Two Techniques to Enhance the Performance of Memory Consistency Models. In *ICPP (1)*. 355–364.

[14] Vaibhav Gogte, Stephan Diestelhorst, William Wang, Satish Narayanasamy, Peter M. Chen, and Thomas F. Wenisch. 2018. Persistency for Synchronization-free Regions *(PLDI 2018)*. ACM, New York, NY, USA, 46–61.

[15] Vincent Gramoli. 2015. More Than You Ever Wanted to Know About Synchronization: Synchrobench, Measuring the Impact of the Synchronization on Concurrent Algorithms *(PPoPP 2015)*. ACM, New York, NY, USA, 1–10.

[16] Timothy L. Harris. 2001. A Pragmatic Implementation of Non-blocking Linked-Lists. In *Proceedings of the 15th International Conference on Distributed Computing (DISC '01)*. Springer-Verlag, London, UK, UK, 300–314. http://dl.acm.org/citation.cfm?id=645958.676105

[17] Maurice Herlihy and Nir Shavit. 2008. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[18] Intel. 2015. Intel and Micron Produce Breakthrough Memory Technology. https://newsroom.intel.com/news-releases/intel-and-micron-produce-breakthrough-memory-technology/.

[19] Joseph Izraelevitz, Hammurabi Mendes, and Michael L. Scott. 2016. Linearizability of Persistent Memory Objects Under a Full-System-Crash Failure Model. In *Distributed Computing - 30th International Symposium, DISC 2016, Paris, France, September 27-29, 2016. Proceedings*. 313–327.

[20] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. 2019. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module. *CoRR* abs/1903.05714 (2019). arXiv:1903.05714 http://arxiv.org/abs/1903.05714

[21] Arpit Joshi, Vijay Nagarajan, Marcelo Cintra, and Stratis Viglas. 2015. Efficient Persist Barriers for Multicores *(MICRO-48)*. ACM, New York, NY, USA, 660–671.

[22] Arpit Joshi, Vijay Nagarajan, Marcelo Cintra, and Stratis Viglas. 2018. DHTM: Durable Hardware Transactional Memory *(ISCA '18)*. IEEE Press, 452–465.

[23] Pete Keleher, Alan L. Cox, and Willy Zwaenepoel. 1992. Lazy Release Consistency for Software Distributed Shared Memory. *SIGARCH Comput. Archit. News* 20, 2 (April 1992), 13–21.

[24] Aasheesh Kolli, Vaibhav Gogte, Ali Saidi, Stephan Diestelhorst, Peter M. Chen, Satish Narayanasamy, and Thomas F. Wenisch. 2017. Language-level Persistency *(ISCA '17)*. ACM, New York, NY, USA, 481–493.

[25] Aasheesh Kolli, Jeff Rosen, Stephan Diestelhorst, Ali Saidi, Steven Pelley, Sihang Liu, Peter M. Chen, and Thomas F. Wenisch. 2016. Delegated Persist Ordering *(MICRO-49)*. IEEE Press, Piscataway, NJ, USA, Article 58, 13 pages. http://dl.acm.org/citation.cfm?id=3195638.3195709

[26] Youyou Lu, Jiwu Shu, Long Sun, and Onur Mutlu. 2014. Loose-Ordering Consistency for persistent memory. In *2014 IEEE 32nd International Conference on Computer Design (ICCD)*. 216–223.

[27] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation *(PLDI '05)*. ACM, New York, NY, USA, 190–200.

[28] Maged M. Michael. 2002. High Performance Dynamic Lock-free Hash Tables and List-based Sets *(SPAA '02)*. ACM, New York, NY, USA, 73–82.

[29] Maged M. Michael and Michael L. Scott. 1996. Simple, Fast, and Practical Non-blocking and Blocking Concurrent Queue Algorithms *(PODC '96)*. ACM, New York, NY, USA, 267–275.

[30] Sanketh Nalli, Swapnil Haria, Mark D. Hill, Michael M. Swift, Haris Volos, and Kimberly Keeton. 2017. An Analysis of Persistent Memory Use with WHISPER *(ASPLOS '17)*. ACM, New York, NY, USA, 135–148.

[31] Dushyanth Narayanan and Orion Hodson. 2012. Whole-system Persistence *(ASPLOS XVII)*. ACM, New York, NY, USA, 401–410.

[32] Aravind Natarajan and Neeraj Mittal. 2014. Fast Concurrent Lock-free Binary Search Trees *(PPoPP '14)*. ACM, New York, NY, USA, 317–328.

[33] Faisal Nawab, Joseph Izraelevitz, Terence Kelly, Charles B. Morrey III, Dhruva R. Chakrabarti, and Michael L. Scott. 2017. Dalí: A Periodically Persistent Hash Map. In *31st International Symposium on Distributed Computing (DISC 2017) (Leibniz International Proceedings in Informatics (LIPIcs))*, Andréa W. Richa (Ed.), Vol. 91. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 37:1–37:16.

[34] Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. 2014. Memory Persistency *(ISCA '14)*. IEEE Press, Piscataway, NJ, USA, 265–276. http://dl.acm.org/citation.cfm?id=2665671.2665712

[35] Azalea Raad and Viktor Vafeiadis. 2018. Persistence Semantics for Weak Memory: Integrating Epoch Persistency with the TSO Memory Model. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 137 (Oct. 2018), 27 pages.

[36] Jinglei Ren, Jishen Zhao, Samira Khan, Jongmoo Choi, Yongwei Wu, and Onur Mutlu. 2015. ThyNVM: Enabling Software-Transparent Crash Consistency in Persistent Memory Systems *(MICRO-48)*. Association for Computing Machinery, New York, NY, USA, 672–685.

[37] Michael L. Scott. 2013. *Shared-Memory Synchronization*. Morgan & Claypool Publishers.

[38] Seunghee Shin, James Tuck, and Yan Solihin. 2017. Hiding the Long Latency of Persist Barriers Using Speculative Execution *(ISCA '17)*. Association for Computing Machinery, New York, NY, USA, 175–186.

[39] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and

Roy H. Campbell. 2011. Consistent and Durable Data Structures for Non-volatile Byte-addressable Memory *(FAST'11)*. USENIX Association, Berkeley, CA, USA, 5–5. http://dl.acm.org/citation.cfm?id=1960475.1960480

[40] Haris Volos, Andres Jaan Tack, and Michael M. Swift. 2011. Mnemosyne: Lightweight Persistent Memory *(ASPLOS XVI)*. ACM, New York, NY, USA, 91–104.

[41] William Wang and Stephan Diestelhorst. 2019. Persistent Atomics for Implementing Durable Lock-Free Data Structures for Non-Volatile Memory (Brief Announcement) *(SPAA '19)*. Association for Computing Machinery, New York, NY, USA, 309–311.

[42] Andrew Waterman, Yunsup Lee, David A. Patterson, Krste Asanovic, Volume I User level Isa, Andrew Waterman, Yunsup Lee, and David Patterson. 2014. The RISC-V Instruction Set Manual.

[43] Song Wu, Fang Zhou, Xiang Gao, Hai Jin, and Jinglei Ren. 2019. Dual-Page Checkpointing: An Architectural Approach to Efficient Data Persistence for In-Memory Applications. *ACM Trans. Archit. Code Optim.* 15, 4, Article Article 57 (Jan. 2019), 27 pages.

[44] Deli Zhang and Damian Dechev. 2016. An Efficient Lock-Free Logarithmic Search Data Structure Based on Multi-dimensional List. *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)* (2016), 281–292.