Release Persistency

Mahesh Dananjaya

Master of Science by Research

Institute of Computing Systems Architecture
School of Informatics
University of Edinburgh

2019

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 the 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 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 33%-52% performance improvement over the state-of-the-art full barrier.

Acknowledgements

First and foremost, I would like to express my gratitude to my supervisor, Dr. Vijay Nagarajan, for his guidance and support throughout the year. Working with Vijay is a great experience not only in acquiring new knowledge but also learning new skills. In fact, Vijay's supervision taught me to think outside the box and approach problems in very different ways. Again, I am very thankful to Vijay for all the patience and encouragement. Second, I thank my parent and my family. Also, ever since I came to Edinburgh, I am pleased to have a wonderful set of friends around me. Special thank goes to Vasilis, Antonis, Rodrigo, Adarsh, Muyang, Arpit and all my friends from CDT for their kind support and encouragement. Finally, I thank all the directors, supervisors and administrators from CDT in pervasive parallelism for giving me an opportunity to engage in cutting-edge research and learning new knowledge.

This work was also supported in part by the EPSRC Centre for Doctoral Training in Pervasive Parallelism, funded by the UK Engineering and Physical Sciences Research Council (grant EP/L01503X/1) and the University of Edinburgh.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Mahesh Dananjaya*)

Table of Contents

Chapter 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 [1, 20]. Program recovery, however, hinges on primitives that control the order in which data becomes persistent. 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. [25] 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 only going to guarantee atomicity of 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 attempts to precisely enforce the orderings intended by the programmer.

In subsequent work, however, Gogte et al. [15] 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 non-blocking data structures [38, 19, 34, 11, 12, 5] 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 [19, 40, 6]. The emergence of NVM has sparked interest in these

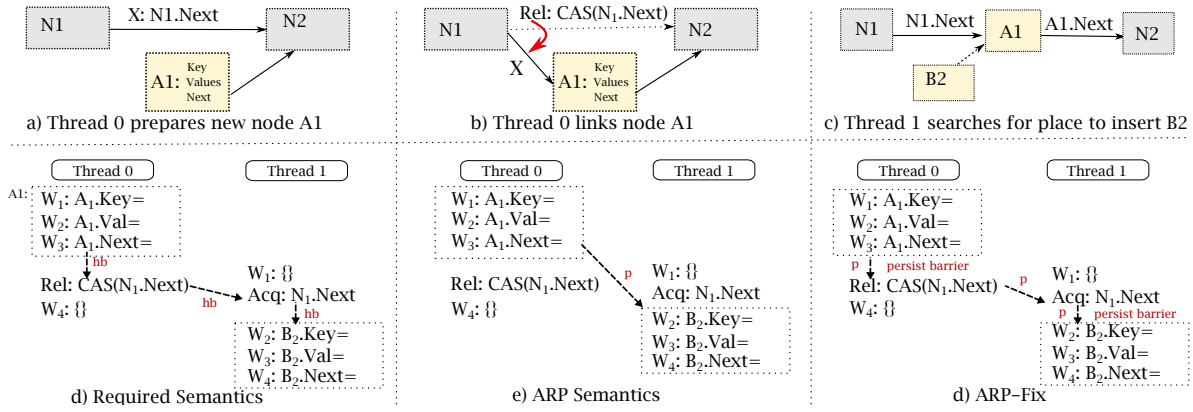

Figure 1.1: ARP's one-sided barriers are not strong enough for log-free programs.

log-free data structures (LFDs) [11] primarily because such programs enable recovery without the overhead of logging.

Therefore, while we concur with Gotge et al. [15] 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 1.1 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 1.1a). Then, it links A1 with the rest of the list via a single atomic Compare-and-Swap (CAS) instruction (Figure 1.1b). 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 1.1d). However, ARP's one-way barrier does not provide this guarantee. (As shown in Figure 1.1e, it only ensures that writes to A1 persist before writes to B2 from the acquiring thread persists). Therefore, to enable recovery, the programmer must place full persist barriers before the release and after the acquire (Figure 1.1f). 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 1.1d, the challenge is to enforce $A1 \xrightarrow{p} X \xrightarrow{p} B2$,

without enforcing either $A1 \xrightarrow{p} A2$ or $B1 \xrightarrow{p} B2$. (The relation $\xrightarrow{p}$ denotes the persist order). We observe that efficiency necessitates a buffered implementation in which persistency is decoupled from visibility [23, 26, 10]. Taking inspiration from lazy release consistency [24], 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. Specifically, when an acquire is performed, LRP detects the matching release via the coherence protocol and enforces the release-side persist ordering ($A1 \xrightarrow{p} X$) lazily.

1.0.1 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 (§)
- We propose strengthening ARP's one-sided barrier semantics. We argue that the resulting model RP enables correct recovery of LFDs upon a crash (§).
- We propose LRP, a microarchitectural mechanism for efficiently enforcing RP's one-sided barriers (§) .
- Our experiments on 5 commonly used LFDs suggests that LRP provides a x-y performance improvement over the state-of-the-art full barrier, while enforcing RP (§).

Chapter 2

Background

In this section, we discuss persistency models with a focus on variants of epoch persistency (§2.0.2). We then discuss log-free data structures and describe the actions required for null-recovery of such structures (§2.0.3). 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, similar to what is supported by the ARMv8 and RISC-V ISAs [3, 42].

2.0.1 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) [14] allows for writes to be tagged as releases and reads as acquires, which have with implicit one-side barrier semantics. Specifically, memory operations before a release appear in the global order before the release, and memory operations after an acquire appear in the order after the acquire. Furthermore, most consistency models support RMWs which are essential for achieving synchronization [4, 18].

Below, we provide a simplistic RC memory model. In our model RMWs have both release and acquire semantics, fences are omitted for simplicity and memory operations to the same address preserve their program order in the global memory order. We use the following notation for memory events:

- $\mathbf{M^i_x}$: a memory operation (of any type) to address x from 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:

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

We formalize Release Consistency using the following rules:

> **Release one-way barrier semantics.** A memory access that precedes a release in program order appears before the release in happens-before: $M_x^i \xrightarrow{\text{po}} Rel_y^i \Rightarrow M_x^i \xrightarrow{\text{hb}} Rel_y^i$.

> **Acquire one-way barrier semantics.** A memory access that follows an acquire in program order appears after the acquire in happens-before: $Acq_y^i \xrightarrow{\text{po}} M_x^i \Rightarrow Acq_y^i \xrightarrow{\text{hb}} M_x^i$.

> **Release to Acquire.** An acquire that follows a release in program order appears before the release in happens-before: $Rel_y^i \xrightarrow{\text{po}} Acq_x^i \Rightarrow Rel_y^i \xrightarrow{\text{hb}} Acq_x^i$.

> **Program order.** Two memory accesses to the same address ordered in program order preserve their ordering in happens-before: $M1_x^i \xrightarrow{\text{po}} M2_x^i \Rightarrow M1_x^i \xrightarrow{\text{hb}} M2_x^i$.

> **Release synchronizes with acquire.** An acquire that synchronizes with a release appears after the release in happens-before: $Rel_y^i \xrightarrow{\text{sw}} Acq_y^j \Rightarrow Rel_y^i \xrightarrow{\text{hb}} Acq_y^j$.

> **RMW-atomicity axiom.** An RMW appears to executes atomically, i.e., for an RMW composed of a read R_x^i and a write W_x^i, there can be no write W_x^j such that $R_x^i \xrightarrow{\text{hb}} W_x^j \xrightarrow{\text{hb}} W_x^i$.

> **Load value axiom.** A read to an address always reads the latest write to that address before the read in happens-before: if $W_x^j \xrightarrow{\text{hb}} R_x^i$ (and there is no other intervening write W_x^k such that $W_x^j \xrightarrow{\text{hb}} W_x^k \xrightarrow{\text{hb}} R_x^i$), the read R_x^i returns the value written by the write W_x^j.

2.0.2 Persistency Models

In a manner analogous to consistency models, Pelly et al. [36] 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_x^i appears after write W_y^j in persist order: $W_x^i \xrightarrow{\text{p}} W_y^j$. To put it succintly, W_y^j can persist only after W_x^i has persisted.

Buffered Epoch Persistency (BEP). BEP allows the programmer to place *persist barriers*, demarcating the program in *epochs* [10, 23]. BEP then uses the epochs to enforce that for any two writes W_x^i, W_y^i, if $W_x^i \xrightarrow{\text{po}} W_y^i$ and the writes belong to different epoch e_k, e_l, then $W_x^i \xrightarrow{\text{p}} W_y^i$. 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.

2.0.2.1 BEP Implementations

We classify prior work on BEP 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*. Below we describe the two classes.

Cache-based. Cache-based implementations (e.g., [23]) buffer writes in the hardware caches and enforce the 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 from a *conflict*; there are two types of conflicts.

$\succ$ **Intra-thread conflicts.** There are two types of intra-thread conflicts: 1) simple cache-line evictions caused by a demand accesses and 2) visibility conflicts caused by attempting a write with epoch e_k on a cache-line with an older epoch-id.

$\succ$ **Inter-thread conflicts.** Caused by a coherence request, e.g., a remote read that is forcing the cache-line to downgrade its coherence state.

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* [23], a technique that starts flushing an epoch as soon as its execution completes.

Persist-buffer-based. This technique buffers writes in a persist-buffer: a FIFO queue at the PM memory controller, that orders all writes [26]. Writes enter the persist buffer in program order and before becoming visible to other threads, i.e., a write cannot be read by a remote thread, unless it has entered the persist buffer. Notably, in this approach, each thread does not have its own epoch; rather the persist buffer maintains one epoch for all threads. Executing a persist buffer in any thread will increment the persist buffer's epoch, affecting all subsequent writes from all threads which will be assigned to the new epoch.

Persist buffers greatly simplify the design, avoiding the complexity of tracking conflicts inside the caches. Note however, that the simplicity comes at the expense of 1) scalability, as there needs to be a single PM memory controller and 2) performance, as persist barriers enforce a persist order among unrelated writes.

RP approach. We choose to implement RP using the cache-based approach (§ 4), because despite its complexity, it is the most efficient and scalable of the two techniques.

2.0.3 Log-free programs

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 data structures, called *nonblocking*, are designed to explicitly avoid locks. To achieve that, nonblocking 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. For instance, in the example of Figure 1.1, Thread T0 first creates a node privately and then it atomically links the node with the linked list through a single CAS instruction.

The intention of this design pattern is to avoid blocking, i.e., avoid states in which a thread is unable to make a progress without the cooperation of one or more peers [38]. In doing so, nonblocking data structures also eliminate the need for failure atomicity: since the atomicity is now incorporated into a single instruction, there is no longer need to persist multiple instructions atomically. Therefore, nonblocking data structures are *log-free*, as they do not require the logging mechanisms that are typically associated with failure atomicity.

To recover a program's progress, the PM must be kept in a consistent state. For a log-free program, where failure atomicity is not required, a consistent state can be achieved by simply ensuring that the PM writes a *consistent cut* of the program's execution, i.e., the persistency order need only mirror the happens-before relations mandated by consistency [19, 5, 11]. Indeed, Izraelevitz and Scott [19] prove that log-free programs can be recovered without any effort (i.e., null recovery), if what remains on the PM after a crash is a consistent cut of the program's execution. In this work, we aspire to provide null recovery for log-free programs, through a novel RC-based persistency model that efficiently mirrors the RC happens-before via one-way persist barriers.

2.0.4 Full Persist Barriers

Full barriers have a substantial overhead on the cache hierarchy due to their strong semantics. So, far, we have discussed the ordering of updates at the persist level. Remember, full barriers do not allow reordering of persists (i.e. write updates to memory) across the barrier. As a result, when an update is being persisted, write updates that belong to all prior epochs must be durably written to memory before the corresponding update has been persisted, i.e. epoch conflict or persist conflicts. Therefore, when memory updates (i.e. persists) are buffered in the cache hierarchy, full barriers implicitly create boundaries between cache-line writebacks and separate them into different epochs. As a result, eviction of a modified cache-lines creates

an epoch conflict and flush all conflicting cache-lines, that possess all the prior epochs, in the critical path of execution. Unlike an individual persists, flushing cache-lines in the critical path creates a considerable persistency overhead. Persist conflicts in buffered epoch persistency (BEP) can be categorized into two main types: *intra-thread and inter-thread conflicts* [23]. We observe that these conflicts are implementation artifacts of full barriers, if occurred, force a set of cache-lines to be flushed before persisting the corresponding cache-line that contains the conflicting persist.

2.0.4.1 Intra-thread conflicts

If an epoch conflict is triggered by the same thread, i.e. due to self-evictions or visibility conflicts, it is an intra-thread conflict. Based on our insights, we organize intra-thread conflicts into three main types: *same-address visibility*, *same-block visibility* and *conflicting cache-line eviction*.

same-address visibility: Writing to the same address from different epochs demarcated by a full pbarrier creates a persist conflict in BEP and forces the particular cache-line to be flushed in the critical path before the consecutive write to the same address is visible. Following full pbarrier semantics, second write starts an epoch conflict and not only flush the particular cache-line, but also all conflicting cache-lines that belongs to prior epochs.

Same-block visibility: Full barriers block the coalescing of updates from different epochs inside a cache-line. For example, writing to different addresses (e.g. A and B) in the same cache-line from different epochs creates a conflict and forces to flush the cache-line before the second write (i.e. B) is visible. Full pbarrier semantics mandate that the first write need to be persisted before the second write. As a result, the second write invokes an epoch conflict and the second write has to wait until the corresponding cache-line and all cache-lines belong to previous conflicting epochs are durably written back to the memory through cache hierarchy. However, remember that full barriers allow updates within an epoch to have coalesced inside the same cache-line.

Conflicting evictions: Full barriers do not allow reordering persists across epoch boundaries, meaning that no update can be persisted until all prior epochs have been persisted. Therefore, if used to order two data writes in different cache-lines, evicting the second line creates an epoch conflict and enforce flushing all previous epochs in the critical path. Therefore, an evicted cache-line starts flushing all cache-lines retaining prior epochs in the critical path of program execution. This severely damages the performance by blocking natural evictions and locality of caching.

2.0.4.2 Inter-thread conflicts

Self-evictions are not the only type of cache-line write-backs. When serving a coherence request from another thread, cache downgrades or invalidates a modified cache-line and write-back data to memory. As a result, it triggers an epoch conflict and flushes all cache-lines belong to prior epochs. These conflicts are known as inter-thread conflict. Unlike, intra-thread conflicts, inter-thread conflicts occur out of the critical path of execution of conflicting sides. However, inter-thread conflicts happen in the critical path of read/write operation of the remote thread.

Chapter 3

Limitations of ARP

Gotge et al. [15] argue that BEP is unsatisfactory because the lack of atomicity guarantees makes recovery cumbersome. In this work, we revisit this statement, arguing for BEP's value for log-free data structures, which can be recovered without any effort after a crash (i.e., null recovery), as long as PM 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 BEP model must mirror the RC semantics, without exceeding them. The key requirement to match RC semantics is treating releases and acquires as *one-way persist barriers*, in the same manner as RC treats them as one-way barriers.

Alas, ARP [25], the only RC-based BEP model falls short of that goal.In the rest of this section, we first describe ARP's semantics (§3.0.1) and implementation (§3.0.2) focusing on how ARP fails to achieve our design goal and then we motivate the need for a new BEP model that can rectify ARP's shortcomings (§3.0.3).

3.0.1 ARP semantics

ARP [25] is a language-level BEP persistency model with a C++11 twist: explicit release and acquire annotations of C++11, have persist semantics. These semantics comprise the *ARP-rule*, which we define below.

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

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

3.0.1.1 ARP semantics shortcomings

The ARP-rule is the only addition of ARP over BEP. We note that the ARP-rule does not mirror the happens-before relations of RC and thus it is unable to provide null recovery as it does not preserve a consistent cut in the PM. 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., $W_y^i \xrightarrow{\text{po}} Rel_x^i \Rightarrow W_y^i \xrightarrow{\text{p}} Rel_x^i$.

Therefore, in the example of Figure 1.1b, 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 thus unrecoverable state.

3.0.2 ARP implementation

ARP is implemented on top of RCBSP [26], a persist-buffer-based BEP model. ARP modifies the ISA, enhancing the release and acquire instructions with persist semantics that enforce the ARP-rule. Note that for the ARP-rule, releases and acquires need not be treated as persist barriers: writes that precede a release need not be ordered with writes that follow the release and writes that follow an acquire need not be ordered with writes that precede the acquire. ARP enhances RCBSP to leverage this observation.

Recall that RCBSP orders all writes in the persist buffer (described in §2.0.2.1). On executing a persist barrier, the buffer's epoch is incremented such that subsequent writes belong to a later epoch, than writes that precede the barrier. ARP enhances this implementation as follows: 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 in the queue 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.

3.0.2.1 ARP implementation shortcomings

Firstly, we note that the implementation abides by the ARP semantics, enforcing only the ARP-rule, without mirroring the RC orderings. For instance, a write that precedes a release is likely

to belong to the same epoch as the release, and can thus persist after the release.

Secondly, even though the ARP authors identify that maximizing performance hinders on providing one-way persist barriers, their implementation still uses full persist barriers (i.e., not one-way). The lack of one-way 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) APR fails to match the RC semantics, while on the other hand, when the barrier is placed (i.e., on an acquire) ARP provides more orderings than RC, as RC treats acquires as one-way barriers.

Finally, ARP inherits the inefficiencies that follow persist-buffer-based implementations (described in §2.0.2.1).

3.0.3 Why not simply fix ARP?

It is possible for ARP to honour the RC semantics, and thus enable the null recovery of log-free data structures, as long as a persist barrier is placed before every release. For instance, in the linked list example of Figure 1.1d, a persist barrier is placed before the release, guaranteeing that the new node is persisted, before it can be linked to the structure.

Recall, however that the design goal is not only to enable null recovery of log-free data structures, but also to maximize performance through one-way persist barrier semantics. By placing a persist barrier before every release, ARP regresses into a generic BEP model with full persist barriers. Aggravating the problem, the persist-buffer-based implementation of ARP pertains solely to full persist barriers, making it impossible to provide the desired one-way persist semantics.

Therefore, it is clear that there is a need for a new BEP model built from the grounds up to provide efficient null recovery for log-free programs, by mirroring RC semantics through the use of one-way persist barriers.

Chapter 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 log-free programs null recovery. Firstly, we formally specify RP (§4.0.1) and then we discuss the implementation implications of our specification, with a focus on the advantages of one-way persist barriers affect (§4.0.2).

4.0.1 Formal Specification

RP must ensure that the persist order reflects the RC happens-before order, which we formally specified in Section 2.0.1. Note that, because the persist order defines the order in which writes persist, only the RC rules that pertain to the order of writes in the happens-before, need to transpose into the RP formalism. Therefore, the load value axiom and the RMW-atomicity axiom, which reason about ordering of writes and reads, do not translate into our RP model.
We formalize RP using 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.** An acquire that synchronizes with a release appears after the release in persist order: $Rel_y^i \xrightarrow{\text{sw}} Acq_y^j \Rightarrow Rel_y^i \xrightarrow{\text{p}} Acq_y^j$.

≻ **Release to Acquire.** An acquire that follows a release in program order appears after the release in persist order: $Rel_y^i \xrightarrow{\text{po}} Acq_x^i \Rightarrow Rel_y^i \xrightarrow{\text{p}} Acq_x^i$.

≻ **Program order.** Two writes to the same address ordered in program order preserve their ordering in persist order: $W1_x^i \xrightarrow{\text{po}} W2_x^i \Rightarrow W1_x^i \xrightarrow{\text{p}} W2_x^i$.

A note on RP acquires. Because an acquire-read cannot persist, the notation $Acq_y^i \xrightarrow{\text{p}} W_x^i$ may appear bizarre. Note, however, that the $\xrightarrow{\text{p}}$ notation simply defines a persist order; that order can contain an acquire-read and treat it as a dummy operation, as long as the acquire-read has meaningful persist semantics that order the persists around it.

4.0.2 Specification implications

In Section 2.0.2.1, we discussed the conflicts that can occur in a cache-based implementation of a buffered epoch persistency model. Conflicts can adversely impact performance, because they can trigger the persist of entire epochs in the critical path of one instruction's execution. However, not all types of conflicts are necessary to to capture the intention of RC; a number of these conflicts are merely an artefact of full persist barriers, which inescapably overshoot the necessary guarantees. One-way persist barriers capture the exact intention of RC and as a result substantially reduce the number of conflicts that need to be handled, pruning all unnecessary constraints.

Eliminated conflicts. Specifically, one-way persist barriers allow for a write to persist before writes of previous epochs. For example, assume $W1_x^i \xrightarrow{\text{po}} Rel_y^i \xrightarrow{\text{po}} W2_z^i$; even though $W2_z^i$ belongs to a later epoch than $W1_x^i$, the persist of $W2_z^i$ does not trigger the persist of $W1_x^i$. This relaxation substantially reduces the number of both inter- and intra- thread conflicts. Notably, if the two writes are to the same cache-line, this rule enables the coalescing of the two writes, eliminating the *visibility conflict*. We identify coalescing of writes as a vital optimization as it reduces the absolute number of persists.

Performance Implication. The ability to recover a program incurs an overhead in the program's execution, as its writes need to persist in the order mandated by the persistency model. The performance degrades because in order to honour the persist order, the processor often needs to stall. Note that, in eliminating the above types of conflicts, RP reduces the number of times the processor must stall. As a result, we hypothesize that the specification of RP can have a profound impact in performance. We prove this hypothesis in our evaluation section (§7).

Chapter 5

Micro-architecture

In this section, we present a cache-based microarchitectural implementation of RP. Firstly, we provide the big picture of how the implementation matches the RP semantics (§5.0.1) and then we dive into the specifics of the implementations (§5.0.2).

5.0.1 Big picture

Firstly, we provide an overview of the metadata kept in RP; based on that metadata, we then provide an operational model of RP, specifying how each of the RP rules is enforced.

Metadata. Each thread has its own epoch-id, which get incremented upon executing a release or an acquire. Furthermore, each cache-line maintains two epoch-ids the *min-epoch* and the *max-epoch*, accounting for the fact that writes of different epochs may coalesce in the cache-line. Upon executing a write, if the cache-line has no valid epoch, both the min-epoch and the max-epoch are assigned, otherwise only the max epoch is assigned. Finally, each cache-line has metadata that denotes whether it stores the value of an acquire or a release.

RP enforces the RP rules as follows:

≻ **Release one-sided barrier semantics.** Before persisting a cache-line that stores a release, all cache-lines with a lower epoch are tracked and persisted.

≻ **Acquire one-sided barrier semantics.** RP only takes action for RMWs with acquire semantics: before a write persists, RP first persists all cache-lines that store the values of RMWs with lower epochs.

≻ **Release synchronizes with acquire.** In order to perform a release, its cache-line must be obtained in modified (M) coherence state. After the release is performed and upon downgrading the M state of the cache-line, RP persists the release, before the downgrade completes. This ensures that if a released value is visible then the release has already persisted.

≻ **Release to Acquire.** Before executing an acquire, all prior releases are tracked and persisted.
≻ **Program order.** RP honours the program order of writes to the same cache-line in the persist order by coalescing the writes, thus making it impossible for two writes to the same variable to be inverted.

RP operation example. Consider the required semantics of Figure 1.1d: T0's W1-W3 must persist before T0's Rel persists and T0's Rel must persist before T1's writes W2-W4. RP fulfills the requirements as follows. T1's Acq will complete only after triggering the persist of T0's Rel, by sending a coherence request for its cache-line. Before T0's Rel can persist, it must first ensure that all writes of previous epochs have persisted (i.e., W1-W3). Finally, T1's W2-W4 cannot begin before T1's Acq has returned, and thus are guaranteed to begin executing after T0 has persisted both its Rel and its preceding writes (W1-W3).

Necessity of two epochs. Each cache-line maintains two epochs enabling the coalescing of writes in the same cache-line. The min-epoch denotes the epoch of the oldest not-yet-persisted write to the cache-line and the max-epoch the epoch of the newest write. Maintaining both epochs is necessary for the following reason: if the persist of cache-line CL_i must trigger the persist of previous writes, then the max-epoch of CL_i must be compared against the min-epochs of the rest of the cache-lines. For instance, when persisting a release, its max-epoch is compared versus the min-epoch of all not-yet-persisted cache-lines.

5.0.2 Hardware Extensions

Figure 5.1(a) shows the overall architecture.

5.0.3 Coherence Protocol

We use the directory-based MESI protocol and modify it to send an acknowledgement (**Ack**) to L1 cache for every writeback update to memory through last-level cache (LLC). MESI protocol has 4 stable states, Modified (M), Exclusive (E), Shared (S) and Invalid (I), and a number of transient states. In our implementation, we do not change states or semantics of MESI protocol and only appropriately flush cache-line and receive an *Ack* from the memory controller (MC).

Persist conflicts occur only when a dirty, modified cache-line in the L1-cache gets flushed into to the persistent memory (PM), i.e. M→I and M→S coherence state transitions due to evictions. Therefore, when a modified cache-line is evicted from the L1-cache, it must be safely written back to the LLC and PM with all dependencies (i.e. conflicting cache-lines) before serving any other coherence request. Once a cache-line has been safely persisted in PM,

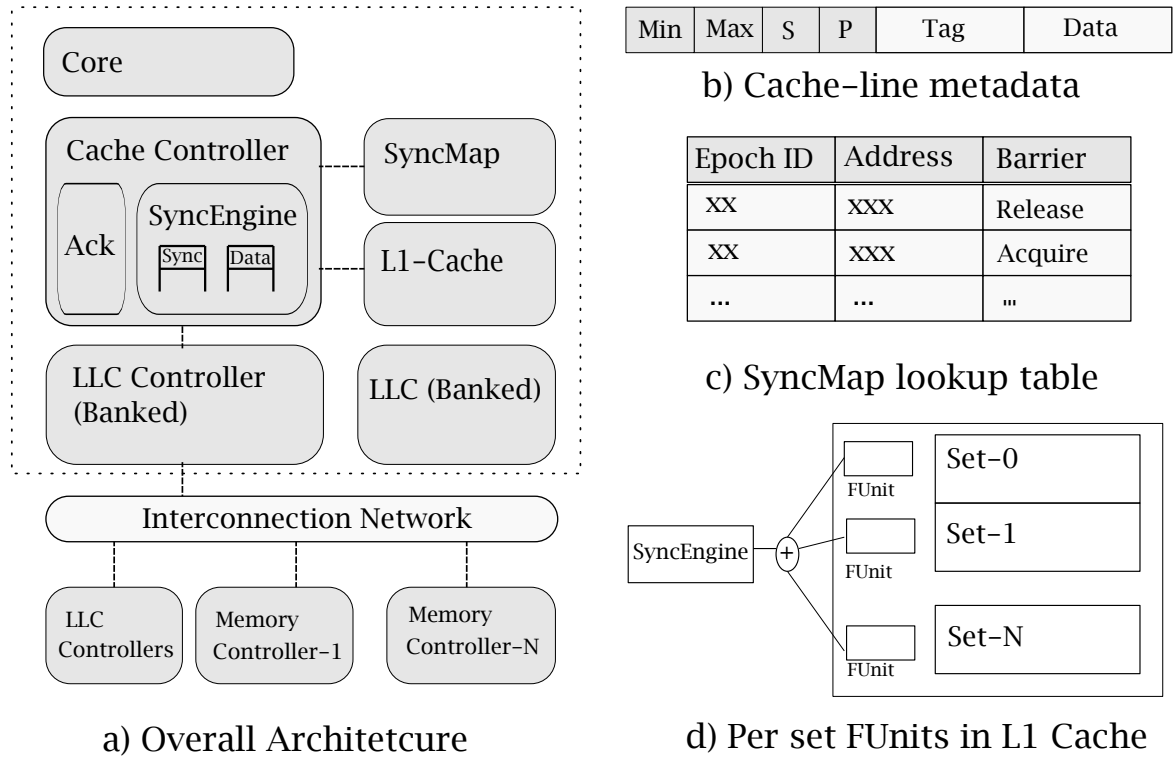

Figure 5.1: Overall Architecture

MC sends an Ack directly to the corresponding L1-cache. Therefore, in order make a cache-line persistent, RP needs to either invalidate (M→I) or self-downgrade (M→S) the line. Similar to clwb in hardware, BEP use a special action to persist a cacheline without downgrading (M→M). Our protocol also support this coherence action. (but hardly use)

5.0.3.1 L1 Cache Modification

Each cache-line in the L1-cache is augmented with a few metadata fields together with two special bits. As shown in the Figure 5.1(b), *Minimum Epoch ID (Min)* keeps track of the oldest epoch that modifies the cache line while *Maximum Epoch ID (Max)* keeps track of the latest epoch id that modifies the cache-line. *Sync Bit (S)* is set if a store that is annotated with a persist barrier modifies the cache-line while *Persist Bit (P)* tag necessary cache-line for persistence.

5.0.3.2 SyncMap and SyncEngine

To keep track of 1-way persist barriers, RP implements a new table, called *SyncMap*. This is a fast lookup table which maps the addresses of synchronization variables into their respective epoch id and one-sided barrier semantics. Except some operations that we will discuss at the

end of this section, stores with 1-way barrier semantics get appended to the SyncMap without any further actions (or conflict). In addition to SyncMap, there is another hardware module inside L1 cache controller, called *SyncEngine*, which we use to flush cache-lines and resolve persist conflicts. See Figure 5.1(c).

5.0.3.3 L1 Cache Controller Modifications

Besides SyncEngine, we use two other modules inside the L1 controller. **Ack Counter.** For each cache-line writeback in the L1 cache, this counter is incremented by 1. Once it receives an acknowledgement for a (any) writeback, it is decremented by 1. Unlike other buffered persistency models, in our model, writebacks to memory and acknowledgement from memory do not need to send epoch information, but only core Id. This is because RP performs most write-backs lazily and do not maintain a speculative state of an epoch throughout the cache hierarchy. Thus RP use a mechanism to enforce the required order from L1-cache which is independent from the number of MCs and LLC-banks.

Queues. We have two outflow queues, Sync queue and Data queue which hold cache-line headers before actually ordering them to memory. Sync queue is for the cache-lines in which the S bit is set (Sync Bit=1). Data queue is for cache-lines that contain only ordinary data without any synchronization variables (Sync Bit=0). Before writing-back a cache-line, L1 first sends the line's header (metadata and cache tag) to these queues by inspecting the P bit.

5.0.4 Persist Operations

5.0.4.1 Execution Phase

RP is a BEP model. However, unlike other persistency models, RP does not have strong epoch boundaries and allow reordering stores across persist boundaries. Nevertheless, RP assigns a store to a particular epoch when it is being committed (i.e visible). Each thread has a separate *Epoch Counter (EC)*, and for each pbarrier, this counter gets incremented by 1. If the barrier is a release, then the thread increment the epoch counter and then commits the release such that the barrier is assigned to a new epoch while all stores before release are assigned to the old epoch. In contrast, if the barrier has the acquire semantics, thread first commits the acquire and then increments the epoch counter. Therefore, all stores after acquire get allocated to a new epoch. As a result, if a store is annotated with both acquire and release (i.e. full barrier), it is assigned to a new epoch of size of 1 which contains only that particular store. There are two different ways of using EC inside a cache-line when the line is being modified. First, if the

cache-line is being modified for the first time, value of the EC (i.e. *epoch id*) is recorded in both Min and Max Epoch Ids of the particular cache-line. Then, if a cache-line is being modified for the second time or more, the epoch id is only recorded in the Max Epoch Id of the line. If a store has persist barrier semantics, it is then recorded in the SyncMap and SyncBit (Sync=1) is set at the execution time.

5.0.4.2 Stores

Cache-lines that contain only ordinary stores (i.e. SyncBit=0) get evicted into the PM without any conflict by incrementing the Ack. If evicted from L1, cache-line is invalidated and written back lazily to PM without staling the program execution.

5.0.4.3 Release

To enforce correct semantics of the release, similar to self-downgrade in lazy coherence protocols, we can eagerly flush(i.e. invalidate) the entire L1 cache on release operation [37, 39] in the critical path of execution. Alsop et al. shows a mechanism to lazily enforce this semantics by tracking acquire from another thread [2](i.e. Lazy-RC). However, both approaches compromise correctness required for RP. Therefore, to enforce the release semantics, we choose to selectively flush required cache-lines using Min.

In an eviction of a cache-line with a release, L1-cache needs to flush all cache-lines that retain stores prior to release. As shown in our epoch creation algorithm, cache-lines that are modified before the release has *Min Epoch Id* that is less than or equal to the Epoch Id just before release. Therefore, in order to selectively flush cache-lines, we traverse through L1 cache and check this simple **condition**: *Min Epoch Id ¡ Sync Epoch Id* (i.e. Epoch Id of the release). Any cache-line that satisfies the required condition is flushed to PM by incrementing the Ack counter.

***Algorithm**:* When evicting a cache-line with release, if the Sync bit is set, then the cache-line's header is sent to Sync queue and does not flush from the L1 cache until all depending cache-lines are flushed. SyncMap then performs a lookup and returns the address, epoch id (i.e. Sync Epoch Id) and semantics of the barrier. Then, SyncEngine traverses through L1 cache and flush required cache-lines from the cache. While evicting cache-lines, we follow the same step to all the cache-lines where S bit is set. cache-lines that do not contain any one-sided barriers are directly flushed to higher level of the memory hierarchy. For each cache-line writeback, Ack Counter gets incremented by 1 and once a Ack is received from the memory controller, this counter get decremented by 1. SyncEngine does not flush cache-lines in Sync

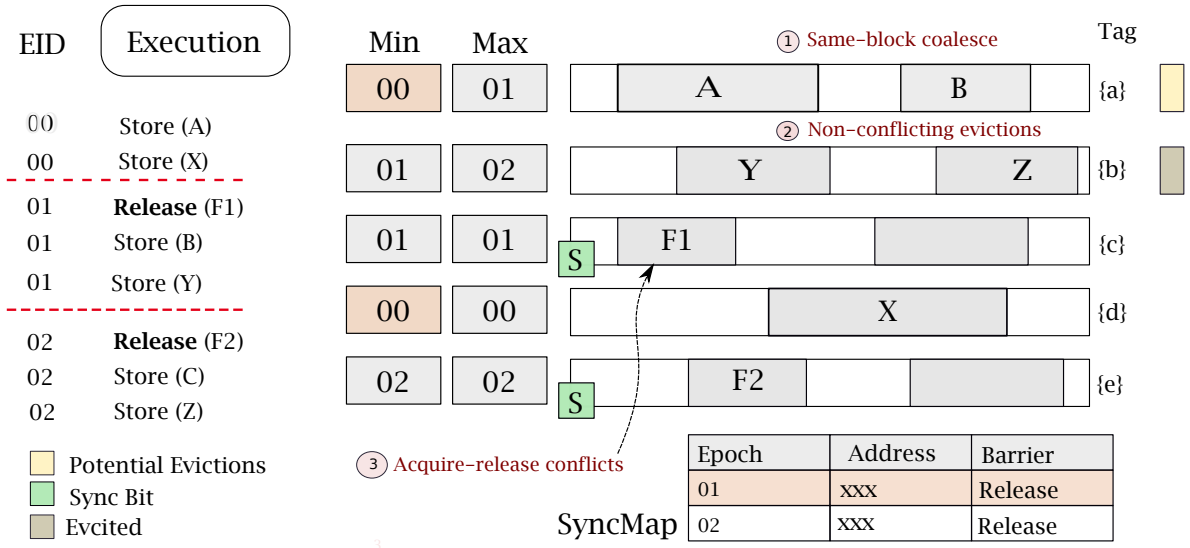

Figure 5.2: One-sided Barrier: Release. (A single store may represent multiple stores)

queue until the ***Ack Counter* is 0**. This makes sure all the required stores get persisted before the corresponding release. Then, we need to make sure all prior releases persist in an order defined by the consistency model. Once the *Ack Counter* is 0, SyncEngine flushes all the cache-lines that contain prior releases in order. If there are further conflicts, SyncEngine resolve them before flushing to the memory. This enforces Invariants 1,4 and 3.

Example: Figure 5.1 shows a simple example[1] of release. When the execution happens, Epoch Counter gets incremented on each release operation, then, as shown in the diagram, max and min epoch ids of the cache-lines are assigned with the value of epoch counter. During the execution, the store-release (or RMW-release) is stored inside cache-lines with Sync bit set (e.g. F1) and placed in the SyncMap as release. To show how same-block coalescing works in RP, we assume both A and B (also Y and Z), two updates belongs to two distinct epochs, get coalesced inside cache-line (a) (and (b)). Therefore, when cache-line (b) get evicted, it does not create any epoch conflict (non-staling). When the cache-line is being evicted, the Ack counter get incremented by 1 (i.e. Ack=1).

Now lets assume, as a result of a read-acquire from another thread (or self eviction), a coherence request approaches the L1-cache for release. When the cache-line (c) get evicted, the metadata header is first buffered inside Sync queue as the Sync bit of the cache-line is already set. Then, SyncEngine perform a SyncMap lookup for the cache-line and collect the required address, epoch id (i.e. *Sync Epoch Id*) and pbarrier semantics of the corresponding release. According to this example, Sync Epoch Id is equal to 1. Therefore, SyncEngine

[1] In this example, single store can be realized as multiple stores as well

traverses* through the L1-cache and flush all the cache-line that has *Min Epoch Id* that is less than the *Sync Epoch Id*. In this example, both (a) and (d) get flushed into the memory by incrementing Ack by 2 (i.e. Ack=3). SyncEngine then waits for 3 Acks from corresponding MCs and decrements the counter. Once the Ack counter is 0, then the required cache-line (c) with release is flushed by incrementing Ack by 1. In the coherence protocol, this cache-line is later written back to the PM before serving an acquire. However, SyncEngine does not need to wait until Ack becomes 0. These steps are same if the cache-line gets evicted as a part of the self-eviction (i.e. intra-thread conflict), but occurs in the critical path of the thread that resides on the flushing L1-cache. To see the effect of multiple release barriers, lets assume cache-line (e) get evicted first. As a result, SyncEngine performs same steps as above and also flush the cache-line (d). Then SyncEngine flush both cache-line (c) and (e) in the correct order of release operations according to their *Sync Epoch Ids*.

As an important advantage of our hardware is that enforcing RP is independent from the number of distributed MCs and LLC banks.

***Traversing L1 Cache**: In RP cache-line flushes are mostly happening out of the critical path of the execution, meaning that the thread that is running on top of the corresponding L1 cache does not stop the thread's execution. Therefore, flushing L1 cache should not stall the program execution. To overcome this problem, as shown in the Figure 5.1(d), we use per-set flushing unit to flush modified cachelines. This reduces the traversing overhead of searching L1 cache from top to bottom [35]. Per-set traversing time overlap with the cache-line flushing time, therefore becomes insignificant.

5.0.4.4 Acquire

As we discussed in the section 4, we assume only one buffered acquire (per-thread) in the L1 cache (e.g. no nested or interleaved locks). Also, acquire become release once the critical section is complete. However, it can get evicted from L1 cache at any time after the execution. For example, L1 cache may self-downgrade the cacheline if some other threads spin on the same lock.

In order to enforce correct one-sided acquire barrier semantics, we need to make sure that no store after acquire persists before the acquire is persisted. First, stores after acquire can get evicted due to self-eviction. Also, as an outcome of the 1-way pbarriers, preceding stores can get coalesced with stores after acquire. Therefore, if a cache-line that contains a prior stores get evicted, there is a possibility that the stores after an acquire also get evicted.

With our epoch creation, we saw all stores after acquire have epoch ids greater than that of

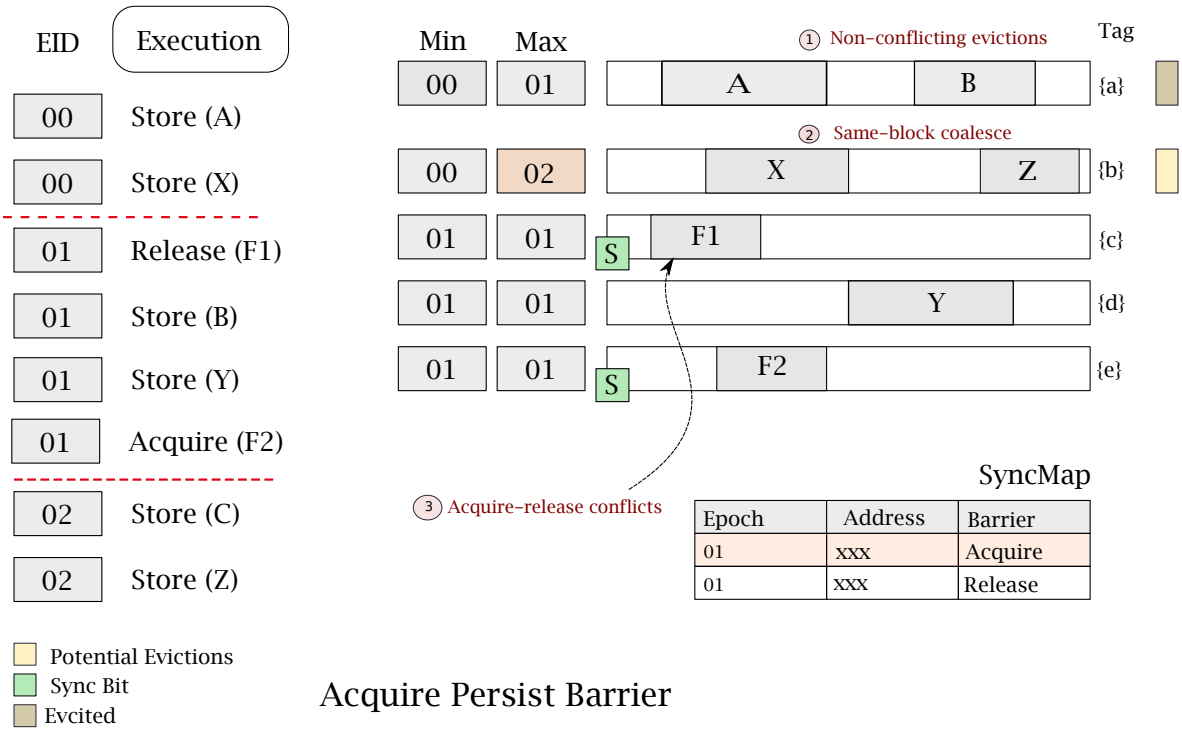

Figure 5.3: One-sided Barrier: Acquire

the acquire. Therefore, all cache-lines that contain stores following acquire have *Max Epoch Id* larger than or equal to that of the epoch id just after acquire. Therefore, for each evicting cache-line, if an *acquire barrier* exists in the SyncMap, we check the following **condition**: *Max Epoch Id ¿ Sync Epoch Id*, i.e. if the SyncMap contains an acquire with an epoch Id that is less than the *Max Epoch Id* of a cache-line being evicted, then the cache-line that retains the acquire must be persisted before persisting the corresponding cache-line.

Algorithm: For each cacheline that is being evicted we perform a SyncMap lookup against the acquire. To make this lookup efficient, we keep an extra *bit* in the SyncMap to flag the existence of an acquire in the SyncMap (i.e. *hasAcquire*). If corresponding acquire exists, SyncEngine first flushes the acquire from L1 cache and increment the Ack by 1 (then Ack ¿= 1). SyncEngine then waits until Ack becomes 0 and evicts the particular cache-line by incrementing the Ack. However, SyncEngine does not wait for the Ack from MC. Until, the cache-line is evicted, its metadata is held by the Data queue. However, if these are (buffered) releases in the SyncMap, to correctly enforce the Invariant 3, SyncEngine performs the release algorithm for the last release in the program order before the corresponding acquire.However, in non-clocking algorithms, read-acquire dominates and these patterns are rare. This enforces Invariants 2 as well as 3.

Example: Figure 5.3 shows an example of how apbarrier works. This extends the previous example, but we change the second synchronization operation to a RMW-acquire instead of Write-release. Therefore, after the execution, SyncMap contains an entry for apbarrier and our extra flag bit is set to inform that the SyncMap has an apbarrier. If write-apbarrier get evicted first, then there is no conflict. Lets assume first few steps are as same as the previous example. First, cache-line (a) get evicted without a conflict. Now, Lets assume as a self-eviction cache-line (b) get evicted from the cache. While SyncEngine performs the lookup, it catches there is an apbarrier that has a larger *Max Epoch Id* than the evicting cache-line. Therefore, the cache-line that has the apbarrier, i.e. cache-line (e) get flushed to PM first and Ack is incremented (i.e. Ack=2). SyncEngine waits until Ack to become 0 and then flush the cache-line (b) by incrementing the Ack by 1. If, apbarrier creates more conflicts, it has to be solved as described in the rpbarrier. If there are no other conflicts, we just need to persist the cache-line that contains the write-apbarrier. Also, Overwriting a sync write-rpbarrier with write-apbarrier to the same address creates an epoch conflict.

Chapter 6

Discussion

We made several simplifying assumptions that allowed us to focus on the technical contribution of this paper better. But these assumptions are not binding to our contributions. Here we briefly discuss what these assumptions are why they are not biding.

Assumption of RC. This paper assumes that the ISA model is a variant of RC. This in iteself is not a very constraining assumption as some important ISAs such as ARM and RISC-V are converging to the simplicity of a globally ordered (i.e. multicopy atomic RC). Having said that our contributions are applicable to other consistency models too, as long as there is ISA support for one-sided persist barriers. In fact, in our experiments because of the constraints of the simulator, the consistency model is a variant of TSO. Therefore we needed to add ISA support for release and acquire one sided barriers.

Persistency Model. Language model and ISA model are the same. In reality, language models are much more complex. There is no global order. Release sequences are more complex. RMWs that are relaxed have more semantics when they are part of the release sequence. These issues need to be handled as they are handled today for consistency models: the mapping from language to ISA primitives.

Persistent Stores. All stores are implicit persist their values. Again this assumption is not tied to our contribution. With ISA support (or which explicit writeback instructions), it is possible to mark a subset of stores are persisting. Correspondingly, our microarchtiecutal implementations tags only these data as persistent-ready using special P bit.

Chapter 7

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 exeuction that provides no persistency guarantees? Before we go to the results, we first discuss our workloads and methodology.

7.0.1 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 [16], which is a collection of nonblocking data structures. Specifically, we used the linkedlist [17], hashtable [29], binary search tree (balanced tree) [33] and skip-list [43] workloads. We also implemented the lock-free queue from Michael and Scott [30]. 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.

7.0.2 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). Recall that SB blocks until all the cache lines modified by the writes before the barrier have been persisted. SB also has an inter-processor component; when a shared memory dependency is detected via the coherence protocol, the target processor blocks until the writes in the the ongoing epoch of the source processor 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-processor persist ordering is captured. (I.e, to ensure that when an acquire synchronizes with a release the acquire correctly blocks until the release persists.)
BB. This represents an RP enforcement approach using the the state-of-the-art full barrier [23]. As discussed in §2, the barrier enforces the persist orderings (both intra-processor and inter-processor) 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-processor persist ordering (I.e, to ensure that when an acquire synchronized with a release all writes following the acquire should persist after the release persists).
NOP. Finally, we also compare against volatile execution which does not enforce any persistency model (NOP).

7.0.3 Simulator

Our hardware implementation is built on top of the pin-based [28] PRiME [13] simulator, with 64 out-of-order-cores processor (single thread per core), a logically shared LLC and multiple memory controllers. Table 7.1 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) con-

Processor	64-Core (out-of-order) 2.4-2.5 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 15 cycles
On-chip Network	2D-Mesh 32 bit flits and links
Coherence	Directory-based, MESI
NVRAM (PCM)	cached mode: 120 cycles uncached mode: 350 cycles
SyncMap (private)	32 Entries

Table 7.1: Simulator Configuration

sistency 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 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.

LRP outperforms BB and SB. Figure 7.1 shows the execution times of LRP, BB and SB normalized to NOP with 32 worker threads and 64K elements. We first observe that BEP outperforms SP, showing a 26%-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 16%-55% (average 33%) improvement over BB.

LRP is 5%-12% within NOP. Figure 7.1 also reveals that LRP is only 5%-12% (average 9%) within volatile execution which suggests that the persistency-related overheads incurred by RP is nominal for these workloads.

Why LRP outperforms BB? Recall that the expected advantage of LRP over BB is that it

Figure 7.1: Average execution time normalized to No-Persistency (lower the better). 64K elements lock-free data structures.

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

In Figure 7.3, 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 69% of the write backs are in the critical path, whereas for LRP only 15% of the write backs are in the critical path. Since almost all of the write back are due to persistency orderings, this suggests that LRP significantly minimizes intra-processing overheads in comparison with BB.

Figure 7.2 compares the normalized execution time overheads of RP vs BB as the number of worker threads are varied from 1–32. Greater the number of threads, greater the probability of inter-processor conflicts and hence potentially high inter-processor persist ordering overhead for RP. However, as seen in Figure 7.2, this effect is nominal: for RP the persistency overhead remains relatively flat with increasing threads. For BB there is a marginal decrease in performance overhead as the number of threads are increased.

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

7.0.4 Results

Individual workload analysis. Whereas RP consistently outperforms BB, as we can see from Figure 7.1, the gap between RP and BP varies. One trend we observed is that, for read-intensive workloads the gap between RP and BP is smaller than for write-intensive ones. As discussed earlier, BB suffers from intra-processor conflicts and these are more pronounced for write-intensive workloads. Thus, we can observe that linkedlist, a read-intensive workloads owing to read-heavy link traversals, shows lesser gain over BB 19% gain compared to BST, a write-intensive workload which shows a relatively higher 37% 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 7.4 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 7.4, RP is more robust to this change when compared with BB or SB. RP continues to incur a nominal 6%-20% (average 11%) 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 RP. Thus, RP shows a significant 76% improvement over BB in this configuration.

Sensitivity to Data structure size. In order to measure the sensitivity of RP 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 over the 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.

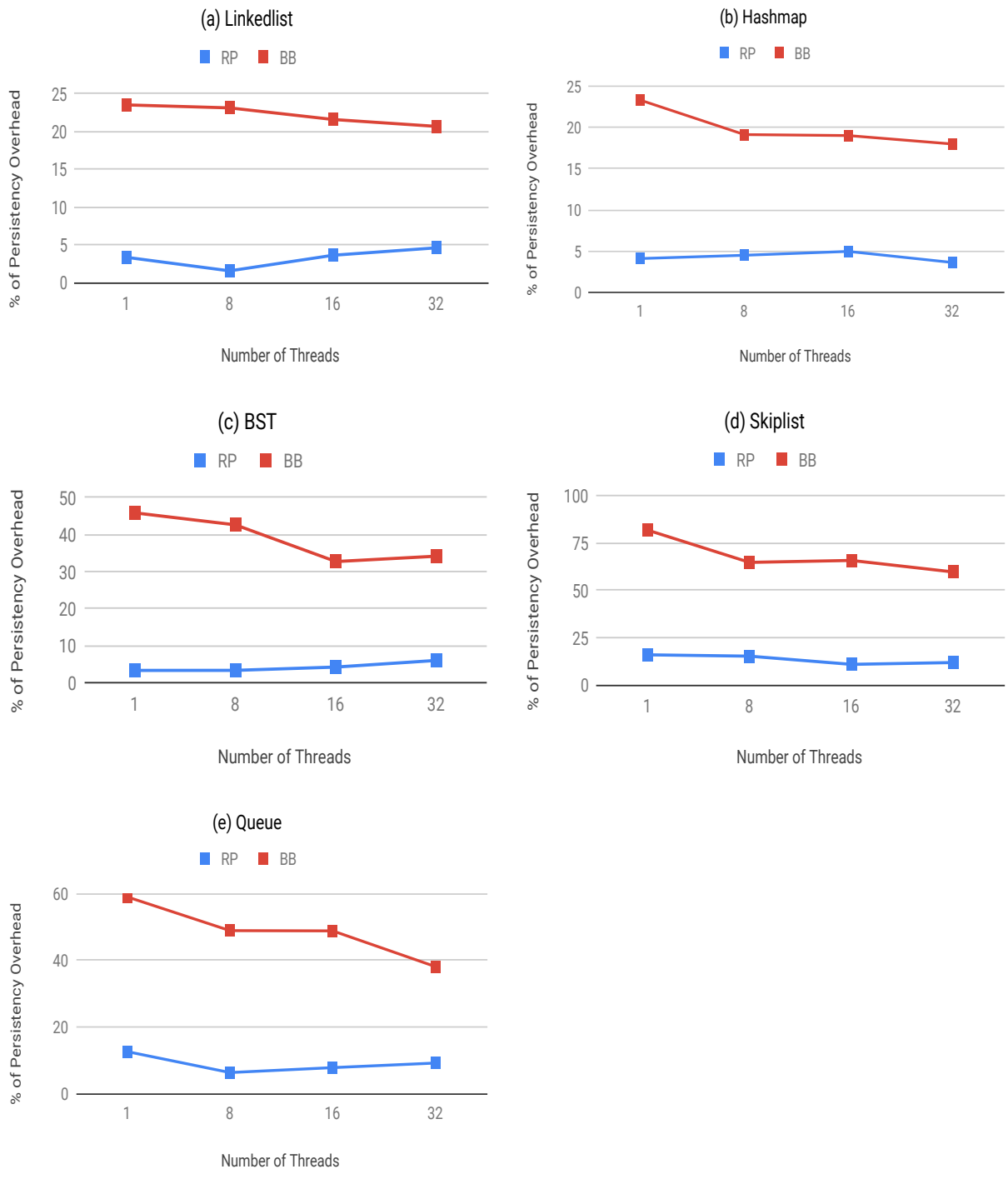

Figure 7.2: % of overhead compared to volatile execution

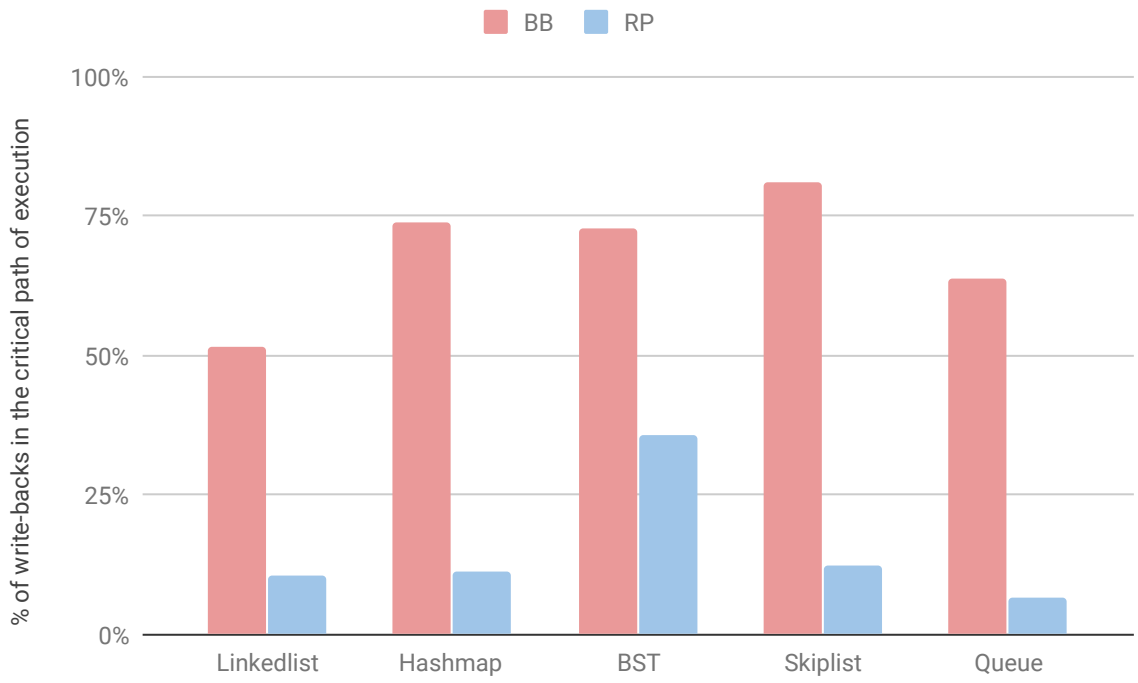

Figure 7.3: Average % of write backs in the critical path

Figure 7.4: Average execution time normalized to No-Persistency in the Uncached mode (lower the better).

Chapter 8

Conclusion and Future Work

8.0.1 Summary

Persistent memory (PM) enables new programming paradigms: fast data persistence and recoverable software that tolerates failures [32, 7, 8, 41, 10, 5]. However, recovery of software programs require a fail-safe state in PM as such data in PM is always in a consistent state after a crash. Therefore, a new programming models are required to keep the data crash-consistent in memory. So far researchers have identified two main persistent programming primitives: *ordering* and *failure-atomicity* [8, 41, 9, 27, 22, 21, 35, 15]. Accordingly, Pelly et al. define *Memory Persistency* to specify a programming framework that order stores to PM [36].

Condit et al. [10] proposed *Lazy Barrier* (LB) by establishing the idea of a persist barrier, such that the programmer can specify the persistent order of stores at a granularity of special regions demarcated by barriers, called *epochs*. LB++ [23] further strengthen the pbarrier semantics by incorporating concurrency control to enforce the correct happen-before order of epochs across many threads. Persist barriers act as a low-level (ISA-level) interface thus programmers have to explicitly barriers to specify the correct. To overcome the programming complexity associated with barriers, DPO [26] proposes a technique that places pbarriers alongside of the consistency fences in relaxed memory models.

Ordering guarantee of pbarriers is often followed by an atomicity technique that guarantee atomic durability of a set of updates [10, 22]. Observing some of the full persist barrier semantics are unnecessary, HOPS [31] proposes two persist barriers: *dfence* and *ofence* to express durability and ordering constraints separately. However, full barrier semantics are still unresolved and have a substantial persistency overhead.

Acquire-release persistency (ARP) [25] investigated the feasibility of a language-level model based on modern DRF-SC programs by studying the implementability of unidirectional fences

in RC for persistency. ARP demonstrated a significant performance gain over full barriers by aggressively reordering persistent stores across unidirectional fences. However, ARP is semantically incorrect thus if properly fixed cannot avoid full pbarriers.

Persist barriers still have unprecedented overhead on fault-free execution which seems like the main drawback of designing fast fail-safe software programs on persistent memory (PM) systems. The questions that we asked were how relaxed a persistency model can be? and can we further relax the persisteny?. To overcome this problem, we propose release persistence (RP).

8.0.2 Lazy Release Persistency (RP)

Release Persistency (RP shows for the first time that the release consistency (RC) can in fact be used to further weaken the relaxed persistency semantics by lifting the ordering the semantics of the persist barrier. RP proposes one-sided persist barriers and efficiently extend the race-free program written using RC semantics in a way that the persistency respect the RC. In doing so, we observe that RC can in fact be used to improve the performance while simplifying the persistent programming model.

One of our objectives was to architect our new persistency model from ground up that support modern hardware with minimal impact of existing cache hierarchy and coherence protocol. Because, modern memory systems are inherently distributed and persistency models must satisfy the need to scalability and performance as well as reliability and fault-tolerance. Inspire by the lazy coherence protocols, we show how to lazily implement RP semantics on modern memory systems by using 1-way barrier semantics. In achieving so, we also reduce the amount of metadata that needs to be stored inside caches. Our observation is that we have achieved better results in all aspects by having almost no overhead on coherence protocols and adding only 3-5% of metadata in any-scale with any number of PMs.

We see an emerging class of log-free algorithms, e.g. log-free programs, that demands a null recovery through ordering-only or atomicity incorporated in to the ordering which preclude unnecessary logging. For such programs, i.e. non-blocking lock-free data structures written using RC semantics, we show our RP model outperforms all existing state-of-the-art persistency models by an average of 43% (upto 53%). RP nor only relaxes the persistency model, but also reduces the programming complexity by leveraging DRC-SC programs which has become the heart of every programming language, e.g. C++, Java

8.0.3 Limitations and Future Directions

This work only investigates a persistency model that gives a programmer a guarantee of what is remaining in the memory in the wake of a crash. However, *null recovery* is not always sufficient. For example, imagine a situation where a client performs a bank transaction and it is crashed in between. In such situation, loosing data might causes problems. Therefore, programmers need more advanced guarantee on top of the persistent happens-before order, e.g. durable linearizability [19] or detectable execution [12]. Release persistency might be well suited for this purpose in the future.

Bibliography

[1] https://newsroom.intel.com/news-releases/intel-and-micron-produce-breakthrough-memory-technology/.

[2] Johnathan Alsop, Marc S. Orr, Bradford M. Beckmann, and David A. Wood. Lazy Release Consistency for GPUs. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-49, pages 26:1–26:13, Piscataway, NJ, USA, 2016. IEEE Press.

[3] ARM Limited. *ARM Architecture Reference Manual ARMv8, for ARMv8-A architecture profile*, 10 2018. Initial v8.4 EAC release.

[4] Hagit Attiya, Rachid Guerraoui, Danny Hendler, Petr Kuznetsov, Maged M. Michael, and Martin Vechev. Laws of order: Expensive synchronization in concurrent algorithms cannot be eliminated. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 487–498, New York, NY, USA, 2011. ACM.

[5] Guy E. Blelloch, Phillip B. Gibbons, Yan Gu, Charles McGuffey, and Julian Shun. The Parallel Persistent Memory Model. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, SPAA '18, pages 247–258, New York, NY, USA, 2018. ACM.

[6] Hans-J. Boehm. Can seqlocks get along with programming language memory models? In *Proceedings of the 2012 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*, MSPC '12, pages 12–20, New York, NY, USA, 2012. ACM.

[7] Hans-J. Boehm and Dhruva R. Chakrabarti. Persistence Programming Models for Non-volatile Memory. In *Proceedings of the 2016 ACM SIGPLAN International Symposium on Memory Management*, ISMM 2016, pages 55–67, New York, NY, USA, 2016. ACM.

[8] Dhruva R. Chakrabarti, Hans-J. Boehm, and Kumud Bhandari. Atlas: Leveraging locks for non-volatile memory consistency. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 433–452, New York, NY, USA, 2014. ACM.

[9] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. NV-Heaps: Making Persistent Objects Fast and Safe with Next-generation, Non-volatile Memories. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 105–118, New York, NY, USA, 2011. ACM.

[10] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. Better I/O Through Byte-addressable, Persistent Memory. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 133–146, New York, NY, USA, 2009. ACM.

[11] Tudor David, Aleksandar Dragojević, Rachid Guerraoui, and Igor Zablotchi. Log-Free Concurrent Data Structures. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 373–386, Boston, MA, July 2018. USENIX Association.

[12] Michal Friedman, Maurice Herlihy, Virendra Marathe, and Erez Petrank. A Persistent Lock-free Queue for Non-volatile Memory. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '18, pages 28–40, New York, NY, USA, 2018. ACM.

[13] Yaosheng Fu and David Wentzlaff. PriME: A parallel and distributed simulator for thousand-core chips. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2014, Monterey, CA, USA, March 23-25, 2014*, pages 116–125, 2014.

[14] Kourosh Gharachorloo, Anoop Gupta, and John L. Hennessy. Two techniques to enhance the performance of memory consistency models. In *ICPP (1)*, pages 355–364, 1991.

[15] Vaibhav Gogte, Stephan Diestelhorst, William Wang, Satish Narayanasamy, Peter M. Chen, and Thomas F. Wenisch. Persistency for Synchronization-free Regions. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, pages 46–61, New York, NY, USA, 2018. ACM.

[16] Vincent Gramoli. More Than You Ever Wanted to Know About Synchronization: Synchrobench, Measuring the Impact of the Synchronization on Concurrent Algorithms. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2015, pages 1–10, New York, NY, USA, 2015. ACM.

[17] Timothy L. Harris. A Pragmatic Implementation of Non-blocking Linked-Lists. In *Proceedings of the 15th International Conference on Distributed Computing*, DISC '01, pages 300–314, London, UK, UK, 2001. Springer-Verlag.

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

[19] Joseph Izraelevitz, Hammurabi Mendes, and Michael L. Scott. 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*, pages 313–327, 2016.

[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. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module. *CoRR*, abs/1903.05714, 2019.

[21] A. Joshi, V. Nagarajan, M. Cintra, and S. Viglas. Dhtm: Durable hardware transactional memory. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 452–465, June 2018.

[22] A. Joshi, V. Nagarajan, S. Viglas, and M. Cintra. ATOM: Atomic Durability in Non-volatile Memory through Hardware Logging. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 361–372, Feb 2017.

[23] Arpit Joshi, Vijay Nagarajan, Marcelo Cintra, and Stratis Viglas. Efficient Persist Barriers for Multicores. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, pages 660–671, New York, NY, USA, 2015. ACM.

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

[25] Aasheesh Kolli, Vaibhav Gogte, Ali Saidi, Stephan Diestelhorst, Peter M. Chen, Satish Narayanasamy, and Thomas F. Wenisch. Language-level Persistency. In *Proceedings of*

the 44th Annual International Symposium on Computer Architecture, ISCA '17, pages 481–493, New York, NY, USA, 2017. ACM.

[26] Aasheesh Kolli, Jeff Rosen, Stephan Diestelhorst, Ali Saidi, Steven Pelley, Sihang Liu, Peter M. Chen, and Thomas F. Wenisch. Delegated Persist Ordering. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-49, pages 58:1–58:13, Piscataway, NJ, USA, 2016. IEEE Press.

[27] Mengxing Liu, Mingxing Zhang, Kang Chen, Xuehai Qian, Yongwei Wu, Weimin Zheng, and Jinglei Ren. DudeTM: Building Durable Transactions with Decoupling for Persistent Memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 329–343, New York, NY, USA, 2017. ACM.

[28] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM.

[29] Maged M. Michael. High Performance Dynamic Lock-free Hash Tables and List-based Sets. In *Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '02, pages 73–82, New York, NY, USA, 2002. ACM.

[30] Maged M. Michael and Michael L. Scott. Simple, Fast, and Practical Non-blocking and Blocking Concurrent Queue Algorithms. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '96, pages 267–275, New York, NY, USA, 1996. ACM.

[31] Sanketh Nalli, Swapnil Haria, Mark D. Hill, Michael M. Swift, Haris Volos, and Kimberly Keeton. An Analysis of Persistent Memory Use with WHISPER. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 135–148, New York, NY, USA, 2017. ACM.

[32] Dushyanth Narayanan and Orion Hodson. Whole-system Persistence. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 401–410, New York, NY, USA, 2012. ACM.

[33] Aravind Natarajan and Neeraj Mittal. Fast Concurrent Lock-free Binary Search Trees. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pages 317–328, New York, NY, USA, 2014. ACM.

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

[35] M. A. Ogleari, E. L. Miller, and J. Zhao. Steal but no force: Efficient hardware undo+redo logging for persistent memory systems. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 336–349, Feb 2018.

[36] Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. Memory Persistency. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, pages 265–276, Piscataway, NJ, USA, 2014. IEEE Press.

[37] Alberto Ros and Stefanos Kaxiras. Complexity-effective Multicore Coherence. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, pages 241–252, New York, NY, USA, 2012. ACM.

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

[39] Hyojin Sung and Sarita V. Adve. Denovosync: Efficient support for arbitrary synchronization without writer-initiated invalidations. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 545–559, New York, NY, USA, 2015. ACM.

[40] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy H. Campbell. Consistent and Durable Data Structures for Non-volatile Byte-addressable Memory. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies*, FAST'11, pages 5–5, Berkeley, CA, USA, 2011. USENIX Association.

[41] Haris Volos, Andres Jaan Tack, and Michael M. Swift. Mnemosyne: Lightweight Persistent Memory. In *Proceedings of the Sixteenth International Conference on Architectural*

Support for Programming Languages and Operating Systems, ASPLOS XVI, pages 91–104, New York, NY, USA, 2011. ACM.

[42] Andrew Waterman, Yunsup Lee, David A. Patterson, Krste Asanovic, Volume I User level Isa, Andrew Waterman, Yunsup Lee, and David Patterson. The risc-v instruction set manual, 2014.

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