Architecting Persistent Memory Systems

by

Aasheesh Kolli

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2017

Aasheesh Kolli

akolli@umich.edu

ORCID iD: 0000-0001-5355-5542

To Venkata Subbamma, my nayanamma, Madan Mohan Rao, my tathayya, and

Krishnamurthy, my mavayya.

ACKNOWLEDGEMENTS

My PhD experience was full of ups and downs, it was rarely flat. For all the fun times when bugs were solving themselves, reviewers were increasing their scores, and papers were getting accepted, there were just as many, if not more, dark times when papers were getting rejected, Reviewer-C was being unreasonable, and I was doubting myself. However, the many wonderful people in my life made the dark times surmountable and the fun times truly memorable and I'd like to express my deepest gratitude to them.

I would first like to thank my parents, Ravi Kumar and Nagamani, and my sister, Sahithi, for their unconditional love, for encouraging me to pursue my dreams, and for making sure that I never had to worry about anything other than achieving my goals. My father is an educator and my mother is a computer scientist, so, looking back, its not very surprising that I chose to get a PhD in computer science. Thanks also to my uncle and aunt, Subbarao and Krishna, for helping me immigrate to the US. Immigrating to the US unlocked opportunities to me that were not available to many of peers back in India.

This thesis would obviously not have been possible without the mentorship from my advisor, Thomas Wenisch. Tom's technical acumen, generosity with time, and irrepressible enthusiasm made working with him an absolute blast. I will forever be indebted to him for all that I have learnt from him.

When I was a confused undergrad wondering what to do next, several people helped me realize that I want to earn a PhD. Without them, I would not have even applied to a PhD program. My cousins, Vamsi, Prabha, Srihari, and Sowjanya inspired me by example and with their work ethic. Rajeev Balasubramonian generously offered me an internship in the UtahArch lab at the University of Utah and helped me gain necessary technical skills. During the internship, I learnt about the life a grad student from Rajeev's grad students Nil, Manu, Kshitij, Ani, and Manju.

I worked closely with Peter Chen and Satish Narayanasamy during my time at Michigan. Pete's critical analysis of my work and patient answers to all my questions, about work or life, were instrumental in me having a fulfilling PhD experience. Satish brought a programmer-centric perspective to my research which markedly improved the quality of my work. Satish also greatly helped me navigate the challenging waters of teaching an undergraduate course with over five hundred students. I would also like to thank Luis Ceze and Karthik Duraisamy for serving on my dissertation committee and their invaluable feedback on my dissertation.

I was fortunate to work with many amazing collaborators, Ali Saidi, Steven Pelley, Vaibhav Gogte, Stephan Diestelhorst, Joe Izraelevitz, Jeff Rosen, Andreas Hansson, Dhruva Chakrabarty, and Terence Kelly, who helped me formulate new research projects, debug code, write papers, and deal with reviewers.

Thanks to all of my lab mates and department friends, Faissal Sleiman, Neha Agarwal, Richard Sampson, Prateek Tandon, Akshitha Shriraman, Amlan Nayak, David Meisner, Andrew Lukefahr, Nilmini Abeyratne, Shaizeen Aga, Mehrzad Samadi, and David Devecsery, who assisted me with my projects, brainstormed ideas with me, listened to my problems over coffee, gave me advise, and attended my many (bad) practice talks.

A special thanks to my cousins Bobby and Sireesha, for helping me get my first laptop and my first car, the two best purchases I have made in my life.

I would also like to thank all my friends who made life in Ann Arbor an absolute pleasure; Shruti, for all the half-eaten cookies, kitchen runs, and impromptu conversations; Vimal, for all the parties, board games, and introducing me to Ann Arbor Ultimate; Shamitha, for sharing my interests in whiskey, politics, and philosophy; Miloni, for your genuineness, sympathetic ear, and for hosting me on my trips to California; Shankar, for being my tolerant roommate; Navneet Sarathy, for all the jokes and fun conversations; Daya, Ankit, Gaurav, Abhay, Divya, and Ritesh, for all the parties, tea, and card games; Mike, Ben, and Biruk, for all the basketball games.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

Architecting Persistent Memory Systems

by

Aasheesh Kolli

Chair: Thomas F. Wenisch

The imminent release of 3D XPoint memory by Intel and Micron looks set to end the long wait for affordable *persistent memory*. Persistent memories combine the persistence of disk with DRAM-like performance, blurring the traditional divide between a byte-addressable, volatile main memory and a block-addressable, persistent storage (e.g., SSDs). One of the most disruptive potential use cases for persistent memories is to host *in-memory recoverable data structures*. These recoverable data structures may be directly modified by programmers using user-level processor load and store instructions, rather than relying on performance sapping software intermediaries like the operating and file systems.

Ensuring the recoverability of these data structures requires programmers to have the ability to control the order of updates to persistent memory. Current systems do not provide efficient mechanisms (if any) to enforce the order in which store instructions update the physical main memory. Recently proposed *memory persistency models* allow programmers to specify constraints on the order in which stores can be written-back to main memory. While ordering constraints are necessary for recoverability, they are expensive to enforce due to the high write-latencies exhibited by popular persistent memory technologies. Moreover, reasoning about recovery correctness using memory persistency models in addition to ensuring necessary concurrency control in multi-threaded programs drastically increases programming burden. This thesis aims at increasing the adoption of persistent memories through a) improving the performance of recoverable data structures and b) simplifying persistent memory programming.

Software transaction abstractions developed using recently proposed memory persistency models are expected to be widely used by regular programmers to exploit the advantages of persistent memory. This thesis shows that a straightforward implementation of transactions imposes many unnecessary constraints on stores to persistent memory. This thesis also shows how to reduce these constraints through a variety of techniques, notably, deferring transaction commit until after locks are released, resulting in substantial performance improvements.

Next, this thesis shows the high cost of enforcing ordering constraints using recent x86 ISA extensions to enable persistent memory programming, an ordering model referred to as *synchronous ordering*. Synchronous ordering tightly couples enforcing order with writing back stores to main memory, but this tight coupling is often unnecessary to ensure recoverablity. Instead, this thesis proposes *delegated persist ordering*, wherein ordering requirements are communicated explicitly to the persistent memory controller via novel enhancements to the cache hierarchy. Delegated persist ordering decouples store ordering from processor execution and cache management, significantly reducing processor stalls, and hence, the cost of enforcing constraints.

Finally, existing memory persistency models have all been specified to be used in conjunction with ISA-level memory models. That is, programmers must reason about recovery correctness at the abstraction of assembly instructions, an approach which is error prone and places an unreasonable burden on the programmer. This thesis argues for a *language-level persistency model* that provides mechanisms to specify the semantics of accesses to persistent memory as an integral part of the programming language and proposes a concrete model, *acquire-release persistency*, that extends C++11s memory model to provide persistency semantics.

CHAPTER I

Introduction

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

Ensuring recoverability of these data structures requires constraints on the order writes become persistent [4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16]. At the same time, it is desirable that PM accesses are cacheable, both to hide access latency and to enable write coalescing to conserve write bandwidth and lifetime for devices subject to wearout. Conventional memory systems delay, combine, and reorder writes to memory at multiple levels, and do not enforce any particular correspondence between the order stores become visible in multi-core systems and the order in which they are written back to main-memory.

Recent work proposes that programming systems be extended with a *memory persistency model*; an explicit specification and programming interface to constrain the order writes to PM become persistent [5, 14, 4, 8]. A memory persistency model is analogous to the memory consistency model [17] that governs the ordering of reads and writes to shared memory, but instead constrains the order PM writes become persistent, an operation referred to as a *persist* henceforth. And, constraints on the order of persists are referred to as *persist dependencies*.

While PMs offer exciting new possibilities, the following challenges, if not addressed, will hinder them from being widely adopted:

- Whereas specifying and honoring persist dependencies is essential for recovery correctness, enforcing persist dependencies is likely to be expensive. PM technologies are expected to be slower than DRAM [18], and will only be able to keep up with CPU speeds through techniques that exploit parallelism, batching, and reordering [9], all of which are possible only in the absence of persist dependencies. Mechanisms to reduce the cost of enforcing persist dependencies, and hence, improve the performance of recoverable data structures are necessary.

- Writing correct parallel programs is hard. Ensuring recovery correctness requires programmers to reason about persist dependencies (via the memory persistency model) in addition to concurrency control mechanisms, exacerbating programming burden. Developing simple, precise, and high-performing programming abstractions to ease programming burden could go a long way towards increasing the adoption of persistent memories.

This thesis comprises of three thrusts, all aiming to address one or both of the challenges described above. The three thrusts are:

- **High-performance transactions for persistent memories:** Software transactional libraries are expected to be widely used by regular programmers to maintain recoverable data structures in PM. We expose the unnecessary persist dependencies resulting from a straight-forward implementation of transactions. We further show how to reduce these unnecessary persist dependencies, resulting in significant performance gains.

- **Delegated persist ordering:** The cost of enforcing individual persist dependencies plays a major role in determining the performance of recoverable data structures. We show how the semantics of the recently proposed x86 ISA extensions for persistent memory programming cause substantial overheads to enforce individual persist dependencies. Subsequently, we also propose an alternative implementation strategy to reduce the cost of enforcing persist dependencies.

- **Language-level persistency:** All of the recently proposed memory persistency models [5, 14, 4, 8] have been specified in conjunction with the ISA-level memory consistency models. However, programmers predominantly reason about parallel programs using language-level memory models exposed by high-level programming languages (e.g. C++11). We propose to expose persistency semantics at the language level, allowing programmers to reason about recovery correctness without concerning themselves with the ISA-level memory consistency and persistency models.

The rest of this chapter provides a brief overview of the proposals in each of the three thrusts mentioned above.

1.1 High-performance transactions for persistent memories

Ensuring that persistent data is consistent despite power failures and crashes is difficult, especially when manipulating complex data structures with fine-grained accesses. One way to ease this difficulty is to access persistent data through atomic, durable transactions, which make groups of updates appear as one atomic unit with respect to failure. Because of the power and convenience of transactions, they are expected to be widely used by regular programmers [11, 7, 6, 19]. Please note that transactions also provide mechanisms for controlling concurrency; in this thesis, we assume that transactions use locks in volatile memory for concurrency control.

Implementing transactions requires the ability to order writes to the PM. For example, in write-ahead logging [20], the commit record for a transaction may persist only after after all log records for that transaction have been persisted; otherwise a failure may cause the system to recover to a state in which only some of the updates are present.

We consider how to implement transactions for PM in a way that minimizes persist dependencies. We show that a simple transaction system design enforces many unnecessary persist dependencies and that these dependencies greatly slow down common transaction workloads. And, that most of the unnecessary dependencies arise as a consequence of performing the commit step of the transaction while locks are held, and how to remove these dependencies by deferring this commit until after locks are released.

Deferring commits leads to the new challenge of correctly ordering the deferred commit operations across all outstanding transactions. To ensure transaction serializability, commit order must match the order in which locks were originally acquired during transaction execution. We show how to minimize persist dependencies through a combination of techniques, including distributed logs [21], deferred commit [22, 23], Lamport/vector clocks to serialize transactions [24], a subtle epoch-based mechanism to recycle log storage, and memory persistency models [5].

While prior works like like *NV-Heaps* [11] and *Mnemosyne* [7] implement transactions for persistent memories, they focus on optimizing transaction design for a particular persistency model. We instead take a more fundamental approach to transaction design that can be applied to many different persistency models. We evaluate the performance of a transaction system that defers commits on simulated PM with a range of device speeds. For two transaction-processing workloads, we show that performance improves from 50%-150% for various memory persistency models [5].

1.2 Delegated persist ordering

Apart from the total number of persist dependencies, the cost of enforcing individual persist dependencies also plays a major role in determining overall performance. We quantify the high costs of enforcing persist dependencies using recent x86 ISA extensions for PM and also proposes an alternative persistency model implementation strategy to reduce said costs.

To complement upcoming memory technology offerings, Intel [4] has announced instruction set extensions to enable programmer control of data persistence. The `clwb` instruction allows programmers to initiate write back of specific addresses to PM, and the `pcommit` and `sfence` instructions enable order enforcement among these writebacks and subsequent execution. we describe the persistency model implied by the semantics of these instructions. This new persistency model is referred to as *synchronous ordering* henceforth.

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

Synchronous ordering couples two orthogonal operations: prescribing an order of persists and ensuring that persists complete (i.e. the corresponding store has been written back to PM). However, coupling these operations is often unnecessary for software system recoverability, as data structure consistency depends principally upon the order of persists [25, 10, 5, 14]. In many contexts, volatile execution may proceed ahead of properly ordered persists without compromising recoverability or waiting for the persists to complete, allowing PM latency to be hidden. When rare failures occur, some writes may be lost, but data structure consistency is maintained (e.g., journaling file systems maintained on disks [10]).

We explore a new implementation approach to enforcing persist dependencies. Instead of enforcing order through stalls, the proposed solution, *delegated persist ordering*, communicates partial ordering requirements mandated by the persistency model explicitly to the PM controller. Delegated persist ordering decouples persistency model implementation from both volatile execution and cache management. Execution and communication via shared memory proceed while PM writes drain. Caches remain volatile and may communicate through cache coherence and evict blocks at will. Instead, our approach maintains writes to PM alongside the cache hierarchy in per-core *persist buffers*.

Using annotations added to coherence transactions, the persist buffers observe and track persist dependencies mandated by the persistency model. Together, they serialize PM writes into a partially ordered buffer at the PM controller, which may then schedule PM writes to exploit available bank concurrency. Execution does not stall unless buffering resources in the persist buffers and at the PM controller are exhausted or the programmer explicitly requests the stall (e.g., before issuing an irrecoverable action). An evaluation of delegated persist ordering demonstrates that it improves performance by 3.73× on average over synchronous ordering for PM-write-intensive benchmarks, coming within 1.93× of volatile execution without order enforcement.

1.3 Language-level persistency

All of the persistency models that have been proposed until now [5, 14, 1, 8, 4, 26] have been specified at the instruction set architecture (ISA) level. That is, programmers must reason about recovery correctness at the abstraction of assembly instructions, an approach which is error prone and places an unreasonable burden on the programmer. The programmer must invoke ISA-specific mechanisms (via library calls or inline assembly) to ensure persist order, and often must reason carefully about compiler optimizations that may affect the relevant code. Since the ISA mechanisms differ in sometimes subtle ways, it is hard to write portable recoverable programs.

We argue for a *language-level persistency model* that provides mechanisms to specify the semantics of accesses to PM (including with respect to program failures) as an integral part of the programming language, just as language-level memory consistency models enable precise specification of the semantics of memory accesses from concurrent threads. A language-level persistency model provides a single, ISA-agnostic framework for reasoning about persistency and can enable portability of recoverable software across language implementations (compiler, runtime, ISA, and hardware). Furthermore, a language-level model prescribes precise requirements on the implementation, allowing implementers to reason about the correctness of compiler and hardware optimizations.

We explore a taxonomy of guarantees that a language-level persistency model might provide. Stronger guarantees (e.g., *failure-atomicity* of critical sections) make writing recoverable software easier but impose substantial requirements on the implementation, which entail performance penalties. Weaker guarantees complicate reasoning about recovery, but provide greater implementation freedom and performance. The weaker guarantees relax atomicity of critical sections and instead provide only *ordering* guarantees for individual persists. Ordering guarantees on individual persists allow synthesis of higher granularities of atomicity via logging.

Based on our taxonomy, we propose a concrete model, *acquire-release persistency*, to extend the C++11 memory model. We describe how to compile ARP to an existing ISA-level persistency model [1]. Ideally, the language and ISA persistency models work in concert to enforce only the minimal guarantees required for correct recovery. However, we find that mismatch between ARP and the ISA-level model lead to extra constraints that hamper performance. We then propose modifications to the C++11 language, compiler, ISA, and hardware to resolve these mismatches, increasing available persist concurrency and scheduling flexibility. The greater flexibility allows the PM controller to reduce page miss rates, improving application performance by up to 33.2% and by 19.8% on average.

1.4 Summary

Soon to be available persistent memories have the potential to transform how software manages persistent data. However, before persistent memories can be widely adopted, a couple of challenges need to be addressed: 1) improve the performance of in-memory recoverable data structures and 2) develop simple and precise programming abstractions to enable persistent memory programming. In this thesis, we identify three different ways to address one or both of these challenges. First, we develop high-performance transactions for persistent memories by reducing the number of unnecessary persist dependencies enforced. Second, we reduce the cost of enforcing individual persist dependencies by decoupling the enforcement of persist dependencies from execution at the core. And finally, we expose the persistency model at the language-level (similar to language-level memory models) to alleviate programmers of the burden to reason about ISA-specific memory consistency and persistency models.

In the reminder of this thesis, Chapter 2 provides the necessary background to better understand the contributions of this thesis. Chapter 3 describes how we improve transaction performance. Chapter 4 details how we reduce the cost of enforcing individual persist requests. Chapter 5, presents the design of a language-level persistency model. Chapter 6 presents a brief summary of related works and Chapter7 the conclusions from this thesis.

CHAPTER II

Background

This chapter provides details necessary to understand the following chapters in this thesis. First, we talk about different persistent memory technologies, followed by an overview of the various different memory persistency models that have been recently proposed.

2.1 Persistent memory technologies

Various memory technologies offer both byte-addressable accesses and non-volatility (or durability). For example, phase change memory (PCM) [18] uses a chalcogenide glass whose resistence can be programmed by varying electrical inputs to the cell. Resistive RAM (ReRAM) [27] is similar to PCM except that instead of using a chalcogenide glass, ReRAM uses metal oxide, whose ressistence can also be programmed via varying electrical inputs. Spin-transfer torque memory (STT-RAM) is a magnetic memory that stores state in electron spin [28]. Storage capacity increases by storing more than two states per cell in Multi-level Cells (MLC) (e.g., four distinct resistivity levels provide storage of 2 bits per cell).

While it remains unclear which of these technologies will eventually gain traction, many share common characteristics. In particular, PMs will likely provide somewhat higher access latency relative to DRAM. Furthermore, several technologies are expected to have asymmetric read-write latencies, where writes are much more expensive than reads [29]. Write latency worsens with MLC, where slow, iterative writes are necessary to reliably write to cells.

Resistive technologies suffer from limited write endurance, that is, memory cells may be written reliably only a limited number of times. While write endurance is an important consideration, proposed hardware mechanisms (e.g., Start-Gap [30]) are effective in distributing writes across cells, mitigating write endurance concerns.

2.2 Ordering constraints

The ability to order writes is critical to all software that uses persistent storage. Constraining the order that writes persist is essential to ensure correct recovery, and minimizing these constraints is key to enabling high performance.

Formally, we express an ordering relation over memory events *loads* and *stores*, which we collectively refer to as *accesses*. The term *persist* refers to the act of durably writing a store to persistent memory. We assume persists are performed atomically (with respect to failures) at 8-byte granularity. By “thread”, we refer to execution contexts—cores or hardware threads. We use the following notation (originally presented here [31]):

- L_a^i: A load from thread i to address a
- S_a^i: A store from thread i to address a

- M_a^i: A load or store by thread i to address a

We reason about two ordering relations over memory events, *volatile memory order* and *persist memory order*. Volatile memory order (VMO) is an ordering relation over all memory events (loads and stores) as prescribed by the memory consistency model for multiprocessors [17]. Persist memory order (PMO) deals with the same events but may have different ordering constraints than VMO. [5] uses the term *memory persistency model* to describe the types of constraints that hardware allows software to express on the persist memory order.

We denote these ordering relations as:

- $A \leq_v B$: A occurs no later than B in VMO

- $A \leq_p B$: A occurs no later than B in PMO

An ordering relation between stores in PMO implies the corresponding persist actions are ordered; that is, $A \leq_p B \rightarrow B$ may not persist before A.

2.3 Memory persistency models

In currently shipping processor architectures, persist dependencies must be enforced either by using a write-through cache or by explicitly flushing individual cache lines (e.g., using the *clflush* instruction on x86). Moreover, these flush operations must be carefully annotated with fences to prevent hardware and compiler reorderings (details appear in [32]). These mechanisms are quite slow because they give up much of the performance benefits of CPU caches. Because cache flushes are so slow, Intel has recently announced extensions to its x86 ISA to optimize cache line flushing [4].

Researchers have proposed other means to express persist dependencies. Condit and co-authors propose an *epoch barrier*, which ensures writes before the barrier are ordered before writes after the barrier [8]. Pelley and co-authors liken the problem of ordering persists to the problem of ordering memory accesses in a multiprocessor [5]. Just as there is a design space for multiprocessor memory consistency models, Pelley lays out a design space for memory *persistency* models. We use *sequential consistency* (SC) as the underlying consistency model in this section. We briefly summarize all of the three persistency models proposed by Pelley.

2.3.1 Strict persistency

Under strict persistency, PMO is identical to VMO. So, for any two stores ordered by the consistency model, the corresponding persists are also ordered. Formally,

$$M_a^i \leq_v M_b^j \leftrightarrow M_a^i \leq_p M_b^j \tag{2.1}$$

Whereas strict persistency is the most intuitive persistency model, it is not the best performing. By ordering persists per VMO, strict persistency enforces orderings typically not required for recovery correctness [5]. Thus, researchers have proposed more relaxed persistency models, in which PMO may have fewer ordering constraints than VMO.

2.3.2 Epoch persistency

The *epoch persistency model* introduces a new memory event, the "persist barrier" (different from memory consistency barriers). We denote persist barriers issued by thread *i* as PB^i. Under epoch persistency, any two memory accesses on the same thread that are separated by a persist barrier in VMO are ordered in PMO.

$$M_a^i \leq_v PB^i \leq_v M_b^i \rightarrow M_a^i \leq_p M_b^i \tag{2.2}$$

Persist barriers separate a thread's execution into ordered epochs (persists within an epoch are concurrent). While persist barriers order persists from one thread, epoch persistency relies on another property, *strong persist atomicity*, to order persists from different threads.

2.3.2.1 Strong persist atomicity:

Memory consistency models often guarantee that stores to the same address by different processors are serialized (this is called *store atomicity*). Pelley argues persistency models should similarly provide *strong persist atomicity* (SPA), to preclude non-intuitive behavior, such as recovering to states unreachable under fault-free execution [5]. SPA requires that conflicting accesses (accesses to the same address, at least one being a store) must persist in the order they executed.

$$\begin{aligned} S_a^i \leq_v M_a^j &\rightarrow S_a^i \leq_p M_a^j \\ M_a^i \leq_v S_a^j &\rightarrow M_a^i \leq_p S_a^j \end{aligned} \tag{2.3}$$

2.3.3 Strand persistency

Strand persistency divides program execution into *strands*. Strands are logically independent segments of execution that happen to execute in the same thread. Strands are separated by the *new strand* (*NS*) memory event. New strand events from thread i are denoted as NS^i. The new strand event clears all prior PMO constraints from prior instructions, effectively making each strand behave as if it were a separate thread (with respect to persistency). Memory accesses within a strand are ordered using persist barriers (Eq. 2.2). Under strand persistency, two memory accesses on the same thread separated by a persist barrier are ordered in PMO only if there is no intervening strand barrier. Memory accesses across strands continue to be ordered via SPA (Eq. 2.3).

$$(M_a^i \leq_v PB^i \leq_v M_b^i) \wedge (\nexists NS^i : M_a^i \leq_v NS^i \leq_v M_b^i) \rightarrow M_a^i \leq_p M_b^i \tag{2.4}$$

2.3.4 Buffering

Pelley further suggests that buffering persists in hardware will expose more opportunities to re-order and coalesce, thus improving performance. Buffering implies that some stores, which have already been executed by a processor and are visible to other processors, might not yet have been persisted to PM. However, the hardware (memory hierarchy) guarantees that, eventually, all these stores will persist *in the order dictated by the PMO*.

Buffering improves performance by ensuring that the memory hierarchy does not have to persist an executed store immediately, but can perform the persist eventually, as long as the correct order is maintained [5, 14]. Volatile execution and cache coherence may proceed while the persist operation is drained lazily to PM.

```
Thread 1    Thread 2
lock L
store A
PB
unlock L
            lock L
            PB
            store B
            unlock L
```

(a) Lock

```
Thread 1    Thread 2
lock L
store A
unlock L
            lock L
            load A
            PB
            store B
            unlock L
```

(b) Observe

Figure 2.1: ***Ordering persists across threads/strands:*** *Common coding patterns to specify inter-thread persist dependencies.*

2.4 Coding patterns to order persists

The previous section described different persistency models that software can use to specify ordering constraints among persists. We next discuss a few canonical examples to show how software can use these interfaces. Persists on a single thread (or strand under strand persistency) are ordered either by VMO (strict persistency) or by persist barriers (epoch and strand persistency).

Enforcing persist order across threads is more complex; as with VMO, these orderings must be established using conflicting accesses.

Figure 2.1 illustrates coding patterns to establish order for a simple scenario under each model. Consider two stores to addresses A and B, executed on different threads (or strands), which are protected by a single lock L, we assume thread 1 wins: $S_A^1 \leq_v S_B^2$. Our objective is to use the persistency models to ensure that the persists of A and B follow the same order: $S_A^1 \leq_p S_B^2$.

The definition of strict persistency (Eq. 2.1) ensures the desired order of persists. Below, we describe two techniques *lock* and *observe*, employed under epoch and strand persistency models to achieve the desired persist ordering.

Lock: The central intuition is to leverage the conflicting accesses of the concurrency control mechanism (i.e., locks), which establish required constraints (e.g., mutual exclusion) in VMO, to also establish the required ordering constraints in PMO. Figure 2.1a shows how to order persists to A and B under epoch persistency using persist barriers PB^1 and PB^2. We denote the unlock operation on thread 1 as S_L^1 and the lock operation on thread 2 as S_L^2. The program orders of thread 1, thread 2 and the ordering property of persist barriers (Eq. 2.2) ensures that:

$$S_A^1 \leq_v PB^1 \leq_v S_L^1 \rightarrow S_A^1 \leq_p S_L^1 \tag{2.5}$$

$$S_L^2 \leq_v PB^2 \leq_v S_B^2 \rightarrow S_L^2 \leq_p S_B^2 \tag{2.6}$$

From conflicting accesses to lock L and SPA (eq 2.3)

$$S_L^1 \leq_v S_L^2 \rightarrow S_L^1 \leq_p S_L^2 \tag{2.7}$$

By transitivity and Eqs. 2.5-2.7, we ensure that $S_A^1 \leq_p S_B^2$. This same reasoning extends to strands (instead of threads) under strand persistency.

Observe: Instead of relying on lock L for conflicting accesses, we can explicitly *observe* (using loads) the specific addresses after which subsequent persists should be ordered, and then issue a persist barrier. Figure 2.1b illustrates this pattern. S_A^1's persist is unordered with respect to any other persist on Thread 1 or (absent L_A^2 and PB^2 we have included) Thread 2. Note that the lock L still ensures mutual exclusion and ordering of the (volatile) execution of the critical sections but, by itself, will not order the persists of A and B. Since thread 1 acquires lock L first, from VMO and SPA (Eq. 2.3), we have:

$$S_A^1 \leq_v L_A^2 \rightarrow S_A^1 \leq_p L_A^2 \tag{2.8}$$

From the program order of Thread 2 and the ordering property of persist barriers (Eq. 2.2), we have:

$$(L_A^2 \leq_v PB^2 \leq_v S_B^2) \rightarrow L_A^2 \leq_p S_B^2 \tag{2.9}$$

By transitivity and Eqs. 2.8 and 2.9, we have $S_A^1 \leq_p S_B^2$. Again, the above reasoning extends to strands as well. In fact, by placing all persists on their own strands and using the observe technique, it is possible to enforce only the required ordering constraints, even under SC.

2.5 Synchronous Ordering

Intel's recently announced extensions [4] provide mechanisms to guarantee recovery correctness and improve upon the performance deficiencies of `clflush`. *Synchronous ordering* (SO) is our attempt to describe the persistency model implied by the semantics of Intel's ISA extensions [4]. We briefly describe the most relevant of these new instructions:

- `clwb`: Requests writeback of modified cache line to memory; clean copy of cache line may be retained.

- `pcommit`: Ensures that stores that have been accepted to memory are persistent when the `pcommit` becomes globally visible.

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

A store operation to cacheable ("write back") memory is assured to be "accepted to memory" when a `clwb` operation ordered after the store becomes globally visible ([4] p. 10-8). However, since `pcommit` is not ordered with respect to the completion of `clwb` operations, an intervening `sfence` is needed to ensure the `clwb` is globally visible. Similarly, a fence operation is required after the `pcommit` to order its global visibility with respect to subsequent stores.

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

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

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

2.6 Formalism for non-multi-copy-atomic models

In this section, we detail the semantics of buffered strict persistency when applied to ARMv7 consistency, yielding a model that we call *relaxed consistency buffered strict per-*

sistency, or RCBSP. Because ARMv7 already allows store reordering between memory fences, RCBSP enables concurrency among persist operations similar to what is allowed under epoch persistency in sequentially consistent systems, without the need to introduce new fence instructions for persists. Moreover, to account for the fact that ARMv7a is a non-multi-copy-atomic model (unlike total store order in previous sections), we introduce new notations to formally specify RCBSP. Next, we precisely specify RCBSP using nomenclature from Pelley [5] and notation similar to that presented in chapter 2.2.

First, we add a new memory event a *fence* to the set of *accesses* (i.e. *loads* (L_a^i) and *stores* (S_a^i)). Note that we use a full strength *dmb* [33] as our *fence*, details later in this section. Further, we use the following notation for dependencies between memory events:

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

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

Local memory order (LMO^i) is an ordering relation over all memory events (loads and stores), observed by thread i, prescribed by the memory consistency model [17]. In relaxed consistency models, especially non-multi-copy-atomic models like ARMv7 [34, 35], different threads may legally disagree on the order in which stores become visible. It is important to note that, *no thread disagrees* with at least a subset of ordering relations, for example, coherence order and orderings enforced by fence cumulativity [35, 34, 36, 37].

In order to account for the fact that ARMv7a is a non-multi-copy-atomic model, we use a more precise definition of Volatile memory order (VMO). VMO is an ordering relation over all memory events as observed by a hypothetical thread that atomically reads all contents of persistent memory at the moment of failure (defined as "recovery observer" in [5]). Note that VMO agrees with all other threads w.r.t. coherence order and fence cumulativity. Persist memory order (PMO) is an ordering relation over all memory events but may have different ordering constraints than any LMO^i or VMO. PMO is governed by the "memory persistency model" [5].

We denote these ordering relations as:

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

An ordering relation between stores in PMO continues to imply that the corresponding persists are ordered; that is,

$$A \leq_p B \rightarrow B \text{ may not persist before } A.$$

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

The motivation for relaxed persistency arises because of the use of conservative consistency models such as sequential consistency (SC) and total store order (TSO). These consistency models require a strict order (in VMO) for all stores and allow little re-ordering or coalescing. Pelley shows that following the same strict order for persists (each of which could take 100s of nano-seconds [18]), hinders performance, much like synchronous ordering. Relaxed persistency models allow programmers to impose a different set of constraints for the PMO than the VMO, thereby allowing more re-order-ing and coalescing in the PMO. Pelley shows that the additional parallelism afforded to persists by relaxed persistency models significantly improves performance.

Even though relaxed persistency models improve performance by exposing additional parallelism, they increase the burden on the programmer by forcing her to reason about two different memory models. ARMv7 consistency already enables parallelism among memory accesses and requires reasoning about proper ordering of shared memory accesses (including non-multi-copy-atomic stores). In this context, we consider the alternate choice of using strict persistency. This choice of relaxed consistency and strict persistency exposes persist parallelism but does not saddle the programmer with an additional memory model. Instead, reasoning about recovery is akin to reasoning about an additional thread.

2.6.1 Relaxed Consistency Buffered Strict Persistency

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

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

$$M_a^i; F^i; M_b^i \rightarrow M_a^i \leq_{li} F^i \leq_{li} M_b^i \tag{2.10}$$

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

$$M_a^i \xrightarrow{d} M_b^i \rightarrow M_a^i \leq_{li} M_b^i \tag{2.11}$$

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

$$S_a^i \xrightarrow{rf} L_a^j \rightarrow S_a^i \leq_{lj} L_a^j \tag{2.12}$$

$$L_a^i \xrightarrow{fr} S_a^j \rightarrow L_a^i \leq_{lj} S_a^j \tag{2.13}$$

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

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

$$\forall (S_a^i, S_a^j), (S_a^i \leq_v S_a^j) \vee (S_a^j \leq_v S_a^i) \tag{2.14}$$

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

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

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

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

$$\forall (M_a^i \in G_A, M_b^j \in G_B), M_a^i \leq_v M_b^j \tag{2.15}$$

The example in Figure 2.2 (a variant of the ISA2 litmus test from [34]) highlights fence cumulativity. A FENCE (F^0) instruction is executed on Core-0. So, S_X^0, preceding F^0, is

$$\begin{array}{llll}
 & \underline{\textbf{Core-0}} & & \underline{\textbf{Core-1}} \\
S_X^0: & St\ X = x & & \\
F^0: & FENCE & & \\
S_Y^0: & St\ Y = y & & \\
 & & L_Y^1: & r1 = Ld\ Y \\
 & & S_Z^1: & St\ Z = r1
\end{array}$$

Figure 2.2: ***Fence cumulativity example:*** *This example shows shows how memory events on different cores may be ordered via fence cumulativity.*

placed in G_A. Note that S_X^0 is the only member of G_A. S_Y^0 is placed in G_B. We assume that L_Y^1 "reads from" S_Y^0, and hence gets added to G_B. The data dependency between L_Y^1 and S_Z^1, requires that S_Z^1 gets added to G_B. So, from Eq 2.15, we have that $S_X^0 \leq_v S_Z^1$, implying that all threads can only observe S_Z^1 after S_X^0. Interestingly, S_Y^0 and S_Z^i are not ordered in VMO as they both belong to the G_B.

Specifically under RCBSP, strict persistency (Eq 2.1), allows two behaviors:

1) Two stores to the same persistent address on different threads will persist in coherence order.

2) Two stores to persistent addresses, one belonging to G_A, and the other belonging to G_B of a $FENCE$ (on any thread), will persist in order (G_A before G_B)

2.6.2 Discussion

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

Next, we present how to implement high-performance transactions for persistent memories implemented using the persistency models described in this chapter.

CHAPTER III

High-performance transactions for persistent memories

3.1 Introduction

Ensuring that persistent data is consistent despite power failures and crashes is difficult, especially when manipulating complex data structures with fine-grained accesses. One way to ease this difficulty is to access persistent data through atomic, durable transactions, which make groups of updates appear as one atomic unit with respect to failure. Because of the power and convenience of transactions, many prior works propose providing them on top of PM [11, 7, 6, 19]. We focus our analysis on static transactions (transactions for which lock sets are known a priori), as detailed in Section 3.2.

Implementing transactions on PM requires the ability to order writes to the NVRAM.[1] Memory persistency models described in the previous chapter allow developers to specify the desired order of persists, and its the responsibility of the hardware to enforce the persist order. This chapter considers how to implement PM transactions in a way that minimizes persist dependencies using various persistency models. We show that a simple transaction

[1]Ensuring recoverability without transactions also requires the ability to order writes.

system design enforces many unnecessary persist dependencies and that these dependencies greatly slow down common transaction workloads. And, that most of the unnecessary dependencies arise as a consequence of performing the commit step of the transaction while locks are held, and how to remove these dependencies by deferring this commit until after locks are released.

We first derive the minimal persist ordering requirements to implement correct transactions under an idealized programming interface that can specify arbitrary ordering constraints to hardware (Section 3.2). However, such a programming interface is unrealistic; we use practical persistency models from Chapter II to express persist dependencies. We then analyze a straightforward transaction implementation, synchronous commit transactions (SCT), demonstrating how it overconstrains persist ordering (Section 3.3). Instead, we propose deferred commit transactions (DCT), which can achieve minimal ordering constraints under sufficiently expressive interfaces (Section 3.4). We evaluate our transaction implementations using the TPCC and TATP transaction processing workloads (Section 4.4) and end with a survey of related work (Section VI).

3.2 Transactions under Idealized Ordering

It is not easy for software to express persist dependencies. Simply ordering the instructions that store data to PM is not sufficient: writes to memory (including PM) are cached and may not be written from the CPU cache to PM in the same order the corresponding instructions were executed [32].

In this section, we suppose that software has the ability to specify precisely the persist dependencies for all writes to PM. While this is unrealistically expressive, it provides a useful baseline upon which to build an idealized transaction system that minimizes persist dependencies. In later sections, we implement transactions built on more realistic interfaces and show how a naive implementation of transactions on these interfaces introduces unnecessary ordering constraints.

The most precise way to specify persist dependencies is as a partial order over all persists. This partial order can be expressed as a directed acyclic graph (DAG), where a node in the graph represents a persist, and an edge exists from node A to node B iff the persist represented by node A must occur no later than the persist represented by node B (note that this condition can be satisfied by performing the two persists atomically). In a system with idealized ordering, the software can express a constraint between any two persists, including persists that occur on separate threads.

We next describe how to build a simple transaction system, given the ability to express general partial orders over all PM writes.

3.2.1 Transaction design

There are many ways to implement transactions [39], with one basic design choice being which version to log of the data being modified in a transaction: the data before the modification (undo logging [40, 11, 41]), the data after the modification (redo logging [7]), or both (e.g., ARIES [20]). In this chapter, we implement transactions with undo logging. We believe this design fits well with storing data directly in PM: both committed and uncommitted data are stored in place, so software can always read the most recent data directly from the in-place data structure (assuming appropriate locks are held). In contrast, if transactions are implemented with a redo log, reads of uncommitted data must be intercepted and redirected to the redo log.

We further implement several common optimizations required to achieve high transaction concurrency. We implement per-thread, distributed logs [7, 21], to avoid the scalability constraints of a centralized log. Our undo log records a copy of data (physical undo records) before it is mutated rather than a "synchronous log-and-update" approach (like PMFS [41]), as the latter requires more persist ordering constraints. We leverage checksum-based log entry validation [42] so that non-atomic writes to a log entry can proceed in parallel, but recovery software can deduce whether a log record was fully written without requiring a separate "valid" bit. This optimization eliminates one persist ordering constraint and is similar to the torn-bit optimization in Mnemosyne [7] and eager commit [19]. We assume concurrency control via arbitrarily fine-grain locking—a transaction must hold all required locks before executing (i.e. static transactions). Requiring a transaction to hold all locks before executing implies that all the data that can possibly be modified by the transaction is known a priori. If such knowledge is not available, a program must execute a read phase to identify all regions it might touch and acquire all locks, and then begin execution (similar to the approach used to implement deterministic transactions [43]).

Figure 3.1(a) depicts the high-level steps of an undo-logging transaction. Steps outlined in a dotted box modify only volatile memory locations; those outlined in a solid box write to persistent memory. We briefly describe each step:

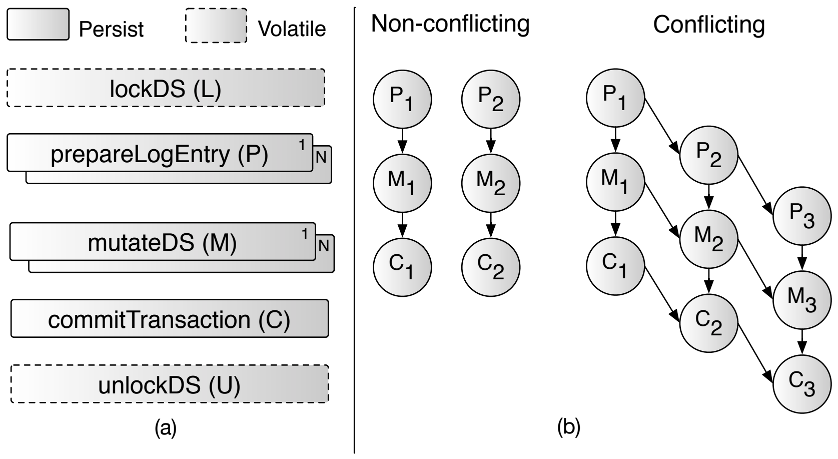


Figure 3.1: ***Ideal undo-loggins transactions:*** *(a) Steps in an undo transaction. (b) Persist dependencies in a transaction sequence.*

- **lockDS (L):** Acquire all locks to ensure mutual exclusion of the transaction. Locks are held in volatile memory.
- **prepareLogEntry (P):** Allocate log space and copy the prior state of all data that will be mutated to the log.
- **mutateDS (M):** Modify the data structure in place.
- **commitTransaction (C):** Commit the transaction by marking the undo log entry invalid; the transaction will no longer be undone during recovery.
- **unlockDS (U):** Release all locks acquired by lockDS.

We represent transactions with three persist nodes, corresponding to the three steps that perform durable writes, *prepareLogEntry* (P), *mutateDS* (M) and *commitTransaction* (C).

3.2.2 Minimal Persist Dependencies

We next analyze the minimal persist dependencies required for correct recovery of an undo-logging transaction. We consider two transactions, T_m and T_n, which acquire lock sets $Locks_m$ and $Locks_n$, respectively. The transactions conflict if their lock sets intersect (i.e., they mutate overlapping data). We require order across conflicting transactions (the order in which they acquire locks); the subscripts indicate this order—in our example, $m < n$. $transactionStep_m$ indicates completion of a particular step in the transaction T_m (and all its associated persists). Recovery correctness requires the following order relationships:

$$prepareLogEntry_m \leq_p mutateDS_m \tag{3.1}$$

$$mutateDS_m \leq_p commitTransaction_m \tag{3.2}$$

$$\forall (m,n):$$

$$(unlockDS_m \leq_v lockDS_n) \wedge (Locks_m \cap Locks_n \neq \phi)$$

$$prepareLogEntry_m \leq_p prepareLogEntry_n \tag{3.3}$$

$$mutateDS_m \leq_p mutateDS_n \tag{3.4}$$

$$commitTransaction_m \leq_p commitTransaction_n \tag{3.5}$$

- Within one transaction, the log entry must be complete before data structure mutation (Eq. 3.1), and mutation must be complete before the transaction commits (Eq. 3.2). These dependencies ensure that any incomplete transaction can be rolled-back during recovery.

- Between conflicting transactions, preparing the log, mutating data, and commit must be ordered (Eqs. 3.3, 3.4, 3.5). These dependencies ensure that: (1) Mutations from conflicting transactions persist in lock-acquisition order (Eq. 3.4). (2) During recovery, active log entries from conflicting transactions can be undone in the appropriate order (Eqs. 3.3, and 3.5). Note that no dependencies exist between non-conflicting transactions.

Next, we discuss how the above constraints ensure recovery correctness.

Intra-transaction: Once a transaction begins, if failure occurs, one of two scenarios arise.

- No valid log entry: As per Eqs. 3.1 and 3.2, the transaction has either successfully committed or hasn't finished preparing a log. In either case, the data structure is in a consistent state; no recovery is required.

- Valid log entry: In this case, the data structure is recovered to a consistent state using the log entry, which undoes all mutations.

Note that it is unnecessary to distinguish a partially from a fully written log entry, since an incomplete undo record has no effect.

Inter-transaction: Since persists from multiple conflicting transactions may be in-flight at the same time, a situation might arise where a particular region of the data structure has multiple associated log entries during recovery. We analyze the scenario with two conflicting log entries, which readily generalizes to additional entries. As above, let T_m and T_n be the conflicting transactions, such that $m < n$ and let Log_m and Log_n be the corresponding log entries. Since we know log entries of conflicting transactions are created and committed in order (Eqs. 3.3 and 3.5), one of three possible scenarios might arise during recovery:

- Both Log_m and Log_n exist: If multiple undo logs exist, the recovery system should undo the transactions from youngest to oldest, according to the order locks were acquired. It determines this order by consulting lock-acquisition timestamps, which are recorded in the log entries. Log_n must be undone before Log_m. Therefore, the recovery mechanism must know the order of the transactions (i.e., the order locks were acquired).

- Only Log_n exists: In this case, only one log entry has to be undone. Eq. 3.5 precludes the case where Log_m exists but transaction n has committed.

- Neither exist: In this case, both the transactions have successfully committed and hence no undo operations are required.

The "mutate in order" constraint (Eq. 3.4) ensures that the data structure mutations are performed in the serialized order of the conflicting transactions, which is required in the common case where no failure occurs. Hence, we have shown that the all the constraints mentioned in this section are required by the recovery mechanism.

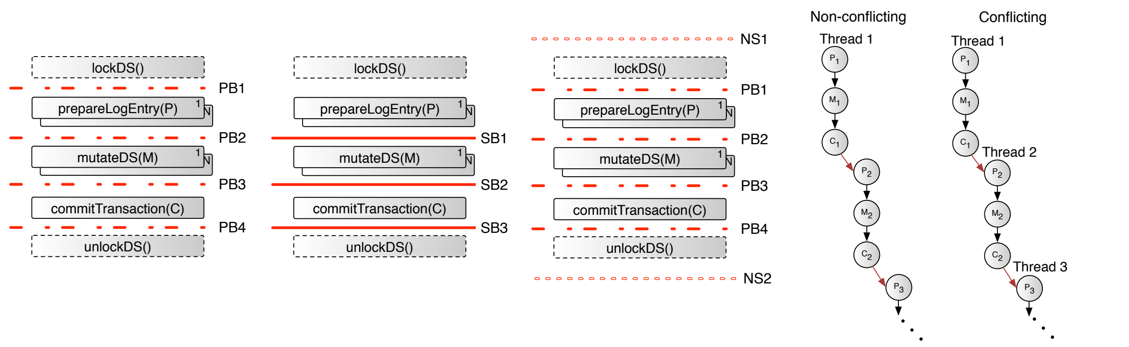
3.2.3 Persist critical path analysis

In later sections, we evaluate alternative transaction implementations by comparing their persist dependency critical path to the ideal persist dependency DAG. Conflicting transactions incur additional dependencies that are absent among non-conflicting transactions. Hence, we characterize the critical path under two extreme scenarios, one where all transactions are non-conflicting, and one where all transactions conflict. Figure 3.1(b) depicts the ideal DAG for conflicting and non-conflicting transaction sequences. Nodes in this figure correspond to the (concurrent) sets of persist operations performed in each transaction step (we omit steps that modify only volatile state). Edges indicate persist dependency between nodes (more precisely, pairwise persist dependencies between all persists represented by each node).

Under each scenario, we assume x transactions are performed, and t threads concurrently execute those transactions. In the non-conflicting scenario, the x transactions all acquire disjoint locks and modify disjoint data. Therefore, there are no persist order dependencies across threads; the critical path is determined solely by persist ordering constraints that arise on a single thread. In this scenario, the ideal persist critical path length is 3—the intra-transaction ordering constraints—independent of x or t.

In the conflicting scenario, we assume all x transactions mutually conflict (they all require a lock in common). Therefore, the persist critical path follows the total order of these x transactions, as established by the order the locks are acquired. In this case, the persist critical path propagates through the commit node of each transaction, resulting in a critical path length of $x+2$ persist operations. Again, the critical path is independent of the number of threads t.

While persist critical paths for an ideal DAG are quite short, achieving this ideal is difficult with currently proposed programing interfaces, as we show next.



(a) SCT under epoch persistency (b) SCT under synchronous ordering (c) SCT under strand persistency (d) Persist critical path (epoch persistency)

Figure 3.2: ***Synchronous-commit transaction designs:*** *Synchronous-commit transactions under epoch persistency, synchronous ordering, and strand persistency. The red arrows in (d) represent the unnecessary dependencies enforced when compared to the minimal dependencies shown in Figure 3.1(b).*

3.3 Synchronous commit transactions (SCT)

Section 3.2 showed how to implement transactions under an idealized programming model allowing arbitrary persist dependencies. We next examine how to implement transactions using more realistic mechanisms.

We first discuss an intuitive transaction implementation, which we call synchronous commit transactions (SCT). However, as we will show, SCT enforces unnecessary persist dependencies and overconstrains the persist critical path. Below, we describe and analyze SCT under epoch persistency, synchronous ordering, and strand persistency (we omit analysis under strict persistency; all our designs will work under strict persistency).

3.3.1 SCT under Epoch Persistency

Epoch persistency enforces intra-thread persist dependencies via persist barriers, and inter-thread dependencies (for conflicting transactions) via persist barriers and SPA (Eq. 2.3). Figure 3.2(a) depicts a synchronous-commit transaction annotated with the four persist barriers required for correctness.

Intra-transaction dependencies: $PB2$ and $PB3$ ensure proper intra-transaction ordering of *prepareLogEntry*, *mutateDS*, and *commitTransaction* (Eqs. 3.1 and 3.2).

Inter-transaction dependencies: Conflicting transactions (T_m and T_n) are synchronized through the common locks in their lock sets and hence through $unlockDS_m$ and $lockDS_n$. Since T_m happens in VMO before T_n, from SPA (Eq. 2.3), we have:

$$unlockDS_m \leq_v lockDS_n \rightarrow unlockDS_m \leq_p lockDS_n$$

The VMO of *prepareLogEntry*, $PB2$ (or $PB3$ or $PB4$), and *unlockDS* in T_m imply:

$$prepareLogEntry_m \leq_p unlockDS_m$$

The VMO of *lockDS*, $PB1$, and *prepareLogEntry* in T_n imply:

$$lockDS_n \leq_p prepareLogEntry_n$$

Applying transitivity to the above three equations, we observe that conflicting transactions prepare their log entries in order, satisfying Eq. 3.3. It is important to note that $PB1$ is critical to ensuring the correct order. Similarly, the VMO of *lockDS*, $PB1$ (or $PB2$), *mutateDS*, $PB3$ (or $PB4$), and *unlockDS* ensures that conflicting transactions mutate the data structure in order, satisfying Eq. 3.4. VMO of *lockDS*, $PB1$ (or $PB2$ or $PB3$), *commitTransaction*, $PB4$, and *unlockDS* ensure that conflicting transactions commit in order, satisfying Eq. 3.5.

Thus, the four persist barriers in Figure 3.2(a) are necessary and sufficient to ensure transactional persist ordering requirements. Unfortunately, these four persist barriers create a persist critical path longer than the path that would be possible had the software been able to specify the precise dependencies between all persists (Section 3.2).

From the VMO of *commitTransaction*, *PB*4, and *unlockDS* in T_m and Eq. 2.2, we have:

$$commitTransaction_m \leq_p unlockDS_m$$

Similarly, VMO of *lockDS*, *PB*1, and *prepareLogEntry* in T_n implies:

$$lockDS_n \leq_p prepareLogEntry_n$$

We have already shown:

$$unlockDS_m \leq_p lockDS_n$$

Applying transitivity to the above three equations, we have:

$$commitTransaction_m \leq_p prepareLogEntry_n$$

So, under epoch persistency, conflicting transactions are serialized. Moreover, transactions on the same thread are always serialized, even if they do not conflict. Figure 3.2(d) shows the persist critical path under epoch persistency: $3x$ for conflicting transactions and $3(x/t)$ for non-conflicting transactions. Both are longer than the minimal critical path (Figure 3.1(b)). Whereas SCT under epoch persistency is simple and intuitive, performing all steps of a transaction while holding locks overconstrains the persist dependency graph and lengthens the persist critical path.

3.3.2 SCT under Synchronous Ordering

Synchronous ordering enforces both intra-thread and inter-thread (for conflicting transactions) persist dependencies via sync barriers. Figure 3.2(b) depicts a synchronous-commit transaction annotated with the three sync barriers required for correctness. We also assume that all the $CLWB$s required to be issued before the sync barriers are issued along with the stores in the functions $prepareLogEntry$, $mutateDS$ and $commitTransaction$.

Intra-transaction dependencies: $SB1$ and $SB2$ ensure correct intra-transaction ordering of $prepareLogEntry$, $mutateDS$, and $commitTransaction$, satisfying Eqs. 3.1,3.2.

Inter-transaction dependencies: We again consider conflicting transactions T_m and T_n. $SB3$ ensures $unlockDS_m$ is not globally visible until $commitTransaction_m$ persists. $prepareLogEntry_n$ cannot be executed until T_n acquires its locks ($lockDS_n$), which happens after $unlockDS_m$ becomes globally visible. By stalling the global visibility of $unlockDS_m$ until $commitTransaction_m$ persists (because of $SB3$), we ensure that:

$$commitTransaction_m \leq_p prepareLogEntry_n$$

It is important to note that a sync barrier between $LockDS$ and $prepareLogEntry$ is not required to achieve the above dependency. The above dependency is the same (over-constraining) dependency incurred under epoch persistency, which serializes all conflicting transactions. $SB3$ also enforces that non-conflicting transactions within the same thread are serialized (as under epoch persistency). SCT under synchronous ordering enforces the same ordering constraints as SCT under epoch persistency, resulting in the same persist critical path (Figure 3.2(d)).

3.3.3 SCT under Strand Persistency

Strand persistency makes it possible to remove unnecessary persist dependencies between transactions on the same thread (left graph of Figure 3.2(d)) by placing the transactions on different *strands*. Our implementation of SCT is shown in Figure 3.2(c). We start and end every transaction on a new strand ($NS1$, $NS2$ in Figure 3.2(c)). As a result, each transaction behaves as if on its own logical thread (from the perspective of the persistent memory, from Eq. 2.4). Such a design removes the dependence between successive non-conflicting transactions on the same thread. Conflicting transactions continue to be ordered due to the dependencies caused by the lock/unlock operations (as under epoch persistency).

It is important to note that in the SCT design for strand persistency (Figure 3.2(c)), $NS1$ ensures that each transaction starts on a new strand, and $NS2$ ensures that memory events executed after the transaction (but prior to the next transaction), don't end up serializing transactions on the same thread. For example, transaction systems may perform some book-keeping (say, update transaction count) at the end of every transaction. Without $NS2$, such bookkeeping could end up causing conflicts between otherwise non-conflicting transactions.

To achieve high concurrency, our SCT implementation uses per-thread (distributed) logs. In practice, log space is limited, and must ultimately be recycled. As a consequence, transactions that share no locks may nonetheless conflict if they reuse the same log space. We enforce the necessary ordering by adding a lock for the log entry to the transaction's lock sets.

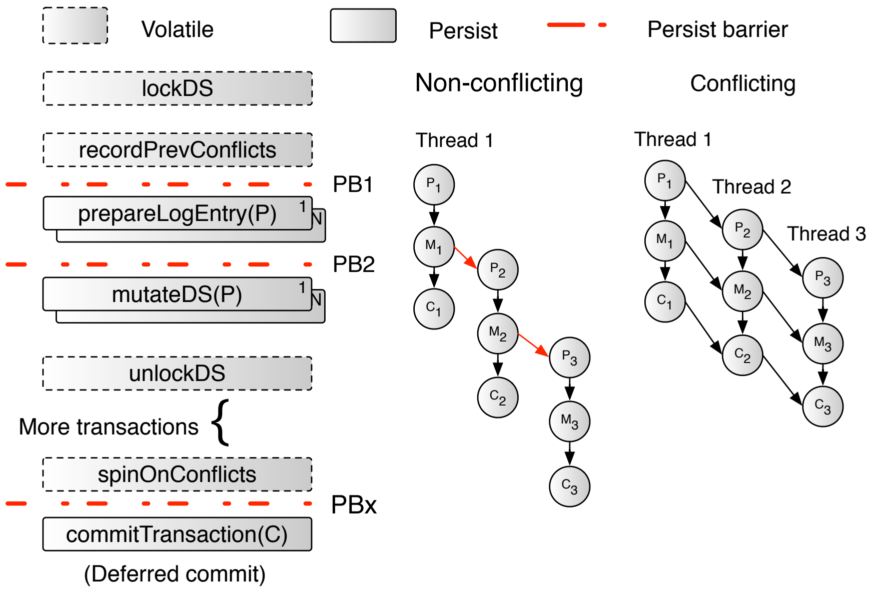
Under ideal implementations of strand persistency, the achievable persist concurrency is limited only by the available log space. In practice, we expect future systems to limit strand concurrency. In a system with t threads and s strands, the maximum concurrency under strand persistency is similar to a system with $s \times t$ threads under epoch persistency. Under strand persistency, the SCT persist critical path for non-conflicting transactions improves to $3(x/st)$ (the persist critical path for conflicting transactions remains $3x$). Whereas strand persistency improves the SCT persist critical path, it remains longer than the theoretical minimum.

3.4 Deferred commit transactions (DCT)

SCT generates longer critical paths than needed when implemented on realistic persistency models. In this section, we describe deferred commit transactions (DCT), which generate shorter critical paths than SCT.

The key idea in DCT is for a transaction to release locks after mutating the data structure and to defer commit until later. This idea has been explored as a mechanism for managing lock contention for transaction systems with a centralized log [22]. We use this idea to break the persist order dependence between *commitTransaction* and *prepareLogEntry* of consecutive conflicting transactions. However, performing the commit after the lock release implies that the persists from *commitTransaction* are no longer synchronized by the respective locks and could result in conflicting transactions committing out of order. [2] To ensure that conflicting transactions commit in order (Eq. 3.5) we modify transactions to explicitly track (in volatile memory) their predecessor conflicting transactions and commit

[2]This is not a problem with a centralized log, as they are serialized by the lock for the log.



(a) DCT under epoch persistency (b) Persist critical path

Figure 3.3: ***Deferred-commit transaction design for epoch persistency:*** *Deferred-commit transactions under epoch persistency and the resulting persist ordering constraints.*

after all predecessors have committed. Next, we describe DCT implementations under the three persistency models.

3.4.1 DCT under Epoch Persistency

Figure 3.3(a) shows the implementation of DCT and the associated "deferred-commit" block.

Intra-transaction dependencies: Persist barrier $PB2$ helps satisfy Eq. 3.1 by guaranteeing that $prepareLogEntry$ is ordered before $mutateDS$. The commit-after-mutate rule (Eq. 3.2) is ensured by PBx (a barrier from a subsequent transaction).

Inter-transaction dependencies: For conflicting transactions T_m and T_n, the persist barriers $PB1$ and $PB2$, along with the SPA guarantees of $unlockDS_m$ and $locksDS_n$, ensure that the log entries are prepared in order, satisfying Eq. 3.3. SPA (Eq. 2.3) of the conflicting regions of the data structure ensure that Eq. 3.4 is satisfied. DCTs need to explicitly track prior conflicting transactions to ensure the commit-in-order rule (Eq. 3.5). We achieve this order by having the transaction $spinOnConflict$ (conflicts are recorded in the log entry) after the lock release and then $commitTransaction$ following a persist barrier *PBx*. It is important to note that, instead of having a designated barrier to order the commit, we rely on a barrier from a subsequent transaction. As a result, the $commitTransaction$ step may occur concurrently with persists from a subsequent transaction and does not add to the persist critical path. Next, we describe the challenges that arise from deferring commits and the bookkeeping required to address them.

3.4.1.1 Inferring undo order during recovery

By allowing transaction commits to be deferred, we can arrive at a state where multiple conflicting uncommitted transactions must apply undo log entries at recovery. The recovery protocol must infer the order of these log entries and perform the undo operations in reverse order. As we use distributed logs, deducing this order is non-trivial. Mnemosyne [7] uses a single global atomic counter to assign each new transaction an incrementing global timestamp (log entries can be undone in the decreasing order of timestamps). However, such an approach implies that all transactions conflict (they all update the global counter), and results in an artificially conflated persist critical path. One might consider recording a timestamp in each log entry, but reliably ordering nearly concurrent events via wall-clock timestamps is challenging, especially if execution is distributed over multiple cores/chips.

Since we only need to order log entries for conflicting transactions, we extend all locks to contain logical time stamps (i.e., Lamport clocks [24]). When a transaction acquires a lock, it records and increments the current lock timestamp, ensuring subsequent conflicting transactions will see a higher timestamp. Timestamps are logged in the new *recordPrevConflicts* function, shown in Figure 3.3(a). If a transaction acquires multiple locks, all of their timestamps must be recorded. Recovery uses these timestamps to deduce the correct undo order.

3.4.1.2 Enforcing correct commit order

To ensure correct recovery, conflicting transactions must commit in order. DCT requires an explicit software mechanism to track and enforce this order. We extend each lock with a pointer to the log entry of the last transaction to acquire that lock. When a transaction acquires all of the locks in its lock set, it records the pointers to previous conflicting transactions (one per lock) in volatile memory, shown as *recordPrevConflicts* in Figure 3.3(a). Then, it records a pointer to its own log entry in each lock.

At commit, a transaction must verify that preceding conflicting transactions have committed. Using the recorded pointers, it examines each preceding log entry for a commit marker, spinning until all are set (*spinOnConflicts* in Figure 3.3(a)). However, if a log entry is recycled, its commit marker is now stale. Along with the commit marker, *recordPrevConflicts* records a log generation number associated with every log entry. The log generation number is incremented if the log entry is recycled. The combination of the commit marker and the log generation number is used to deduce whether an earlier transaction has committed.

Once all of the conflicting log entries are committed, the transaction may commit (*commitTransaction* in Figure 3.3(a)). (Note that, rather than simply spinning, an intelligent transaction manager could instead further defer commit and execute additional transactions on this thread). The spin loop prior to commit orders conflicting transactions in VMO. A persist barrier is required between *spinOnConflicts* and *commitTransaction* (*PBx*) to ensure the conflicting transaction commits are also ordered in PMO.

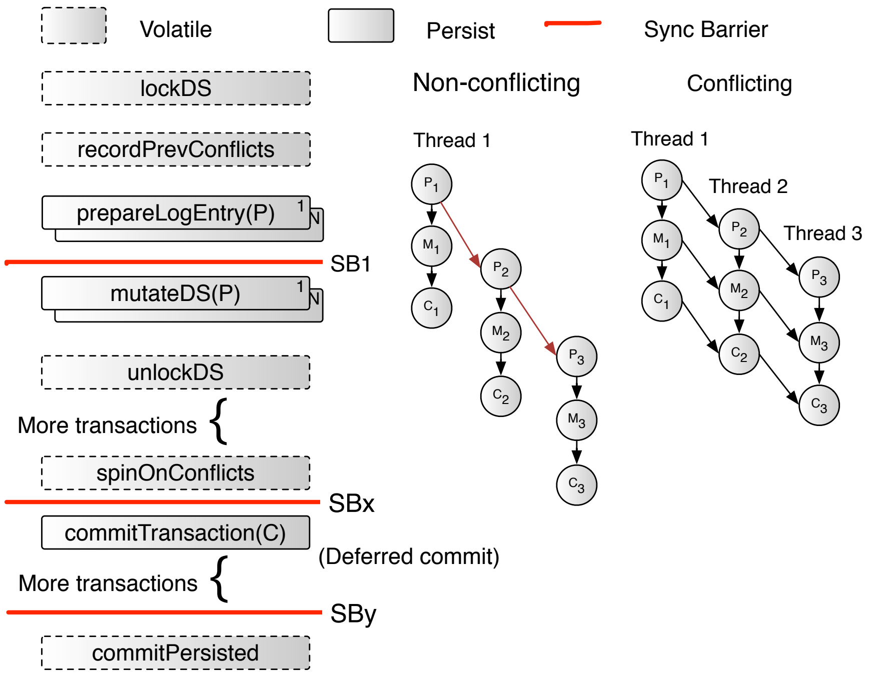
3.4.1.3 Persist critical path analysis

Figure 3.3(b) shows the persist critical path for DCT under epoch persistency. DCT succeeds in matching the critical path length of the ideal dependence DAG for conflicting transactions as derived in Section 3.2. For transactions on a single thread, it reduces the critical path by allowing commit operations to be batched.

For non-conflicting transactions on the same thread, the *prepareLogEntry* and *mutateDS* steps remain (unnecessarily) serialized. The *commitTransaction* step overlaps with the *prepareLogEntry* step of the subsequent transaction. Hence, the non-conflicting persist critical path length is $(2(x/t)+1)$. For conflicting transactions, the persist critical path traverses the *commitTransaction* step of each transaction, and its path length is $x+2$.

3.4.2 DCT under Synchronous Ordering

The implementation of DCT under the synchronous ordering persistency model is shown in Figure 3.4(a). While similar to the DCT implementation under epoch persistency, we require some subtle changes to account for the differences between a persist barrier and a sync barrier detailed in Section 2.5. It is important to note that we require only one sync barrier within the transaction, rather than the two persist barriers required for DCT under epoch persistency.



(a) DCT under synchronous ordering (b) Persist critical path

Figure 3.4: ***Deferred-commit transaction design for synchronous ordering:*** *Deferred-commit transactions under synchronous ordering and the resulting persist ordering constraints.*

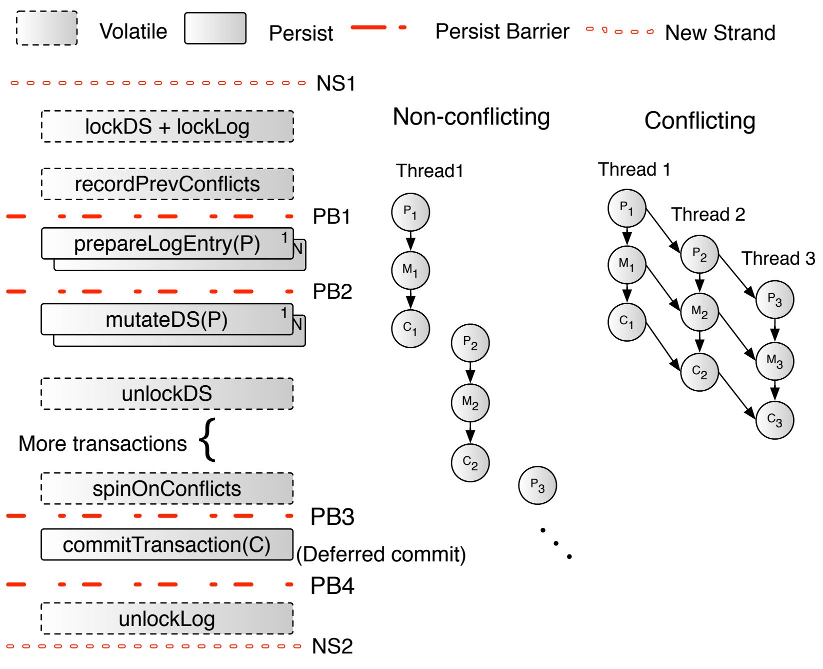
Intra-transaction dependencies: The sync barrier $SB1$, ensures order between $prepareLogEntry$ and $mutateDS$, satisfying Eq. 3.1. The sync barrier *Sbx* (which belongs to a subsequent transaction) ensures that $mutateDS$ and $commitTransaction$ are ordered correctly, satisfying Eq. 3.2.

Inter-transaction dependencies: We discuss inter-transaction dependencies using two conflicting transactions T_m and T_n. Within T_m, $SB1$, ensures that $unlockDS_m$ doesn't become globally visible until $prepareLogEntry_m$ becomes persistent. In transaction T_n, $prepareLogEntry_n$ cannot be executed before the locks are acquired using $lockDS_n$. However, $lockDS_n$ cannot be completed until $unlockDS_m$ becomes globally visible. Transitively, $SB1$, ensures that log entries are prepared in order, satisfying Eq. 3.3. Cache coherence ensures that at any given time, only the latest values of any conflicting regions of the data structure persist, satisfying Eq. 3.4.

Since SPA (Eq. 2.3) is not provided under eager sync, we cannot use the same coding pattern as used in the epoch persistency DCT implementation to ensure conflicting transactions commit in order. Instead, we have a *commitPersisted* bit associated with every log entry, which is set after $commitTransaction_m$ is guaranteed to be have persisted (ensured by *SBy* in Figure 3.4(a)). We modify the *spinOnConflict* function to spin on the *commitPersisted* bits of conflicting transactions, rather than the log entries. Once a transaction observes that the *commitPersisted* bit of earlier conflicting transactions have been set, it can be committed and be certain that the correct commit order has been followed.

It is important to note that we do not need dedicated sync barriers, *SBx* and *SBy*, for every transaction. We instead rely on sync barriers from a subsequent transaction, implying that both *mutateDS* and *commitTransaction* are persisted concurrently with later transactions. So, only the persists belonging to *prepareLogEntry* fall on the persist critical path on a single thread, as depicted in Figure 3.4(b). For non-conflicting transactions, the persist critical path traverses all the *prepareLogEntry* steps of each transaction executed on one thread and is $x/t+2$. For the conflicting case, the persist critical path traverses the *commitTransaction* step of all the transactions and is $x+2$. Note that DCT under eager sync incurs a shorter persist critical path than under epoch persistency for non-conflicting transactions, whereas they exhibit the same persist critical path for conflicting transactions.

Discussion: DCT under eager sync and Mnemosyne (asynchronous mode) [7] are similar in that each transaction may add at most one persist epoch delay to the execution critical path. Whereas DCT amortizes the cost of *mutateDS* and *commitTransaction* over subse-



(a) DCT under strand persistency (b) Persist critical path

Figure 3.5: ***Deferred-commit transaction design for strand-persistency:*** *Deferred-commit transactions under strand persistency and the resulting persist ordering constraints.*

quent transactions on the same thread, Mnemosyne offloads log truncation to a separate helper thread.

3.4.3 DCT under Strand persistency

Figure 3.5(a) shows our implementation of DCT under strand persistency. As with the SCT implementation under strand persistency, we expose additional persist concurrency by placing each transaction on a new strand, removing the dependencies between non-conflicting transactions on the same thread. Similar to SCT, we introduce a log entry lock to a transaction's lock set, so that transactions which conflict on a log entry are serialized. The log entry lock is acquired along with all the other locks in a transaction's lock set. However, the log entry is only released after *commitTransaction*, serializing transactions that share log space. Figure 3.5(b) shows the persist critical paths for the conflicting and

	Non-conflicting		Conflicting	
Persistency Model	**SCT**	**DCT**	**SCT**	**DCT**
Epoch	$3x/t$	$2x/t+1$	$3x$	$x+2$
Synchronous ordering	$3x/t$	$x/t+2$	$3x$	$x+2$
Strand	$3x/st$	$3x/st$	$3x$	$x+2$
Notation: x-transactions, t-threads, s-strands/thread				

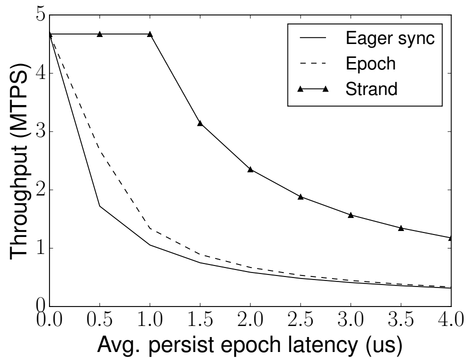
Table 3.1: ***Persist critical path lengths summary:*** *Summary of the persist critical path lengths observed with SCT and DCT for different persistency models.*

non-conflicting scenarios. In the conflicting case, as under epoch persistency, the persist critical path passes through the $commitTransaction$ step of each transaction, leading to the ideal persist critical path length of $x+2$ edges. In the non-conflicting case, the persist dependency critical path improves, but may still fall short of the ideal DAG if the number of strands supported in hardware is limited. The persist critical path for non-conflicting transactions goes through transactions which share log space and is $3x/st$ where t is the number of threads and s the number of strands per thread (similar to SCT under strand). With support for at least two strands per thread, DCT under strand persistency outperforms DCT under epoch persistency.

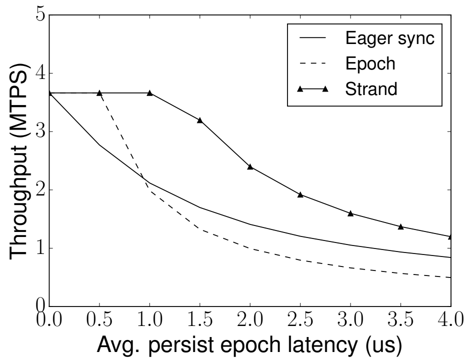
Table 3.1 summarizes the critical paths for SCT and DCT under various persistency models and workloads.

3.5 Evaluation

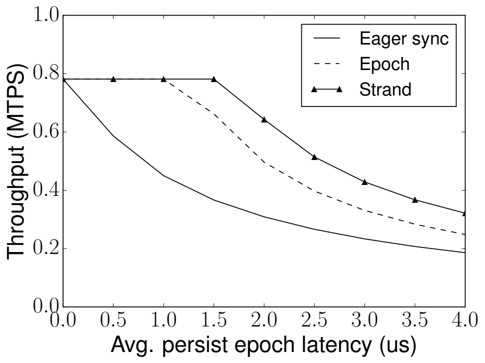
We evaluate transactional systems implementing both SCT and DCT to examine their performance trade-off as a function of persist latency for two transaction processing workloads. In general, we expect DCT to have slower volatile execution performance, due to the bookkeeping overheads required to order commits. However, as persist latency increases, it



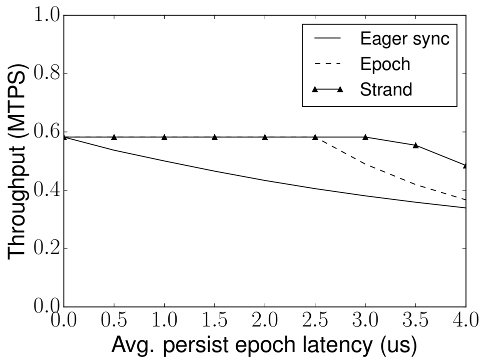
(a) Update Location (SCT)



(b) Update Location (DCT)



(c) New Order (SCT)



(d) New Order (DCT)

Figure 3.6: ***Evaluation of SCT vs DCT:*** *SCT and DCT performance for Update Location and New Order under various persistency models.*

rapidly becomes the performance bottleneck, and DCT overtakes SCT. As the PM programming interface remains unclear, we also compare the performance achieved under different persistency models for both SCT and DCT. Strict persistency performs much worse than all other persistency models, so we omit it from the discussion.

3.5.1 Methodology

We implement our transactional systems as a C++ library providing a simple API comprising only three entry points: `beginTransaction()`, `prepareLogEntry()`, and `endTransaction()`. The system manages bookkeeping, log serialization, commit ordering, and inserting the necessary barriers to enforce persist dependencies.

Because memory-bus-attached PM devices are not yet available, we use a region of DRAM as a proxy. We execute our workloads writing persistent data to DRAM to establish their volatile execution performance. We then re-execute the workloads with lightweight pin instrumentation [44] to record all persist operations and barriers. From these traces, we construct the persist critical path (taking into account ordering within and across threads). We assume 8-byte atomic persists.

Under epoch and strand persistency, overall throughput is limited by the slower of volatile execution and the latency to drain all persists. However, in the case of synchronous ordering, overall throughput is limited by volatile execution, which includes the stalls associated with executing the sync barriers. The overhead of a sync barrier only includes the latency to make the stores persistent and does not include the costs associated with issuing and executing the corresponding *CLWB*s.

As the hardware characteristics, raw device latency, and scheduling limitations of a practical persistent memory system are as yet unknown, we vary our assumption for persist performance and report the resulting throughput. Cumulative persist latency is determined by how fast, on average, an epoch of persists can drain subject to queueing, scheduling, device-level concurrency, and coalescing effects. We abstract these effects as a single average latency per persist epoch (i.e., latency per dependency edge in the critical path). As we expect persist throughput to be the performance bottleneck when transactions are short, load on the persistent memory system will be high and queueing delays substantial. Hence, the average persist epoch latency is likely a small integer multiple of the PM device latency, we study the the range of 0-4μs.

We perform experiments on an Intel Xeon E5645 processor (2.4GHz). We analyze throughput for transactions selected from two widely studied transaction processing workloads. We study the *New Order* transaction from TPCC [45], which is its most frequent write transaction. A New Order transaction simulates a customer buying different items from a local warehouse. The transaction is write-intensive and requires atomic updates to several tables. We also study the *Update Location* transaction from TATP [46], a benchmark that models a mobile carrier database. Update Location records the hand-off of a user from one cell tower to another. In contrast to New Order, Update Location transactions are much shorter, updating a small record in a single table. We execute workloads for 10M transactions running on four threads. We assume four strands per thread under strand persistency.

3.5.2 Performance analysis

Figure 3.6 contrasts SCT and DCT performance across workloads, persistency models, and average persist epoch latencies depicting throughput (millions of transactions per sec (MTPS)) versus persist epoch latency (micro seconds).

Each performance result with epoch and strand persistency (Figure 3.6) comprises a flat region, followed by a curve where throughput rapidly falls off. In the flat regions, where average persist epoch latency is low, overall throughput is limited by volatile execution. At the knee, which we call the "break-even" latency, volatile execution and the persist critical path are equal. Higher break-even latency implies tolerance for slower PM technologies.

Performance then drops off rapidly as average persist epoch latency increases and asymptotically reaches zero. In contrast, for synchronous ordering, since sync barriers cause stalls in volatile execution, performance begins to drop-off at the first non-zero average persist epoch latency.

Volatile execution performance of SCT exceeds DCT. As expected, the additional bookkeeping required to implement DCT penalizes volatile execution speed—SCT transactions are faster than DCT transactions (if persist epoch latency is neglected) by 20%, 25% for Update Location and New Order respectively.

SCT performance across persistency models: Figures 3.6(a) and 3.6(c) show the performance of SCT, for Update Location and New Order, for different persistency models. SCT under synchronous ordering always performs worse than under epoch persistency. This behavior is to be expected as SCT exhibits similar persist critical paths under epoch persistency and synchronous ordering. With similar persist critical paths, the performance under epoch persistency is always better than under synchronous ordering. Under epoch persistency, performance is determined by the slower of volatile execution and time taken to drain the persists. However, in the case of synchronous ordering, the time taken to drain persists (stalls due to sync barriers), slows volatile execution.

Also, as expected, SCT performs better under strand persistency than under epoch persistency, due to a shorter persist critical path. Hence, SCT performs best under strand persistency and the worst under synchronous ordering.

DCT performance across persistency models: The performance trade-offs for DCT are more complex. Figure 3.6(b) shows that the performance of DCT under epoch persistency is worse than under synchronous ordering above 1μs persist latency. DCT incurs a longer persist critical path under epoch persistency than under synchronous ordering, especially for workloads where transactions rarely conflict, like Update Location. Hence, beyond the break-even latency, the performance under epoch persistency declines faster than under eager sync.

For New Order (Figure 3.6(d)), we see that DCT performs better under epoch persistency than under synchronous ordering. This behavior is caused by multiple factors: (1) The break-even latency for epoch persistency is 2.5μs, so epoch persistency performance degrades only for persist latencies above 2.5μs. (2) New Order has more conflicting transactions than Update Location, so the difference in persist critical path between epoch and synchronous ordering is smaller. (3) The crossover point at which synchronous ordering begins outperforming epoch persistency lies beyond 4μs, which is not shown in the graphs. It is not clear that a memory technology that incurs more than 4μs average persist epoch latency is viable as a main memory.

As expected, DCT under strand persistency performs best for both workloads (Figures 3.6(b) and 3.6(d)) as the persist critical path under strand persistency is the shortest.

SCT vs. DCT across persistency models: The performance trade-off (for all persistency models) between SCT and DCT depends upon two competing factors: (1) the better volatile performance of SCT, and (2) the shorter persist critical paths of DCT. As a result, for lower average persist epoch latencies, SCT performs better, but as latency increases, DCT outperforms SCT by up to 50% under epoch and strand persistency and 150% under synchronous ordering.

In Table 3.2, we summarize the average persist epoch latency, where SCT and DCT provide the same performance, under each persistency model. Table 3.2 indicates: (1) DCT

	Update Location	New order
Epoch persistency	0.5	2
Synchronous ordering	0.5	1
Strand persistency	1.5	2.5

Table 3.2: ***SCT vs DCT performance break-even:*** *The average persist epoch latency (in μs), at which DCT breaks even with SCT.*

breaks-even with SCT at higher latencies for New Order than Update Location. New Order is a larger transaction, hiding longer persist delays under volatile execution. (2) Strand persistency exhibits the highest SCT-DCT break-even latencies, as it incurs the smallest difference in persist critical path between DCT and SCT.

CHAPTER IV

Delegated persist ordering

4.1 Introduction

The focus of the previous chapter was to reduce the overall cost of enforcing persist dependencies by identifying and reducing the number of unnecessary persist dependencies enforced. This chapter aims at reducing the overall cost of enforcing persists by reducing the cost of enforcing each individual persist dependency. This chapter quantifies the high costs of enforcing persist dependencies using recent x86 ISA extensions for PM (synchronous ordering, Chapter 2.5) and also proposes an alternative persistency model implementation strategy to reduce said costs.

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

In this chapter, we explore a new implementation approach to enforcing persistency model ordering requirements for PM writes. Instead of enforcing ordering through stalls, we investigate an implementation approach, which we call *delegated persist ordering*, that communicates partial ordering requirements mandated by the persistency model explicitly to the PM controller. Delegated persist ordering represents a fundamental departure from existing persistency implementations and common approaches for enforcing (volatile) write ordering for relaxed memory consistency. Relaxed consistency is often implemented like synchronous ordering, relying on stalling (e.g., at memory fence instructions) to prevent mis-ordering the visibility of memory operations.

We evaluate delegated persist ordering by implementing it for buffered strict persistency [5, 14] (PM writes must reflect the order stores become globally visible, but their persistence may be delayed) over a relaxed consistency model (i.e. RCBSP, as described in Chapter 2.6) and compare its performance to synchronous ordering. We evaluate both approaches in a cache hierarchy that implements ARM's relaxed memory consistency model, which allows reordering and concurrency among stores between ordering points. We implement a series of PM-write-intensive benchmarks, adding minimal fence instructions required for correctness under each model, and evaluate performance using the gem5 simulation infrastructure [47]. We compare both approaches for different PM technologies. In summary, the contributions of this chapter are:

- We analyze the semantics and performance of synchr-onous ordering—the persistency model implied by Intel's ISA extensions for PM [4]—and demonstrate that it results in an 7.21× slowdown on average relative to volatile execution without ordering.

- We propose delegated ordering, an approach to memory persistency implementation that exposes partial ordering constraints explicitly to the PM controller.
- We evaluate delegated ordering and demonstrate that it improves performance by 3.73× on average over synchronous ordering for PM-write-intensive benchmarks, coming within 1.93× of volatile execution without order enforcement.

4.2 Performance of synchronous ordering

We first briefly summarize the semantics of synchronous ordering (more detailed discussion was presented in Chapter 2.5) and then the performance implications of these extensions.

4.2.1 Semantics

Synchronous ordering (SO) is our attempt to describe the persistency model implied by the semantics of Intel's ISA extensions [4]. The most relevant of these new instructions are:

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

A store operation to cacheable ("write back") memory is assured to be "accepted to memory" when a `clwb` operation ordered after the store becomes globally visible ([4] p. 10-8). However, since `pcommit` is not ordered with respect to the completion of `clwb` operations, an intervening `sfence` is needed to ensure the `clwb` is globally visible. Similarly, a fence operation is required after the `pcommit` to order its global visibility with respect to subsequent stores.

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

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

4.2.2 Performance

SO's overhead can be broken into two components:

- The overhead due to `clwb` instructions. `clwb` writes modified data back from the cache hierarchy to the memory controller. Each `clwb` must snoop all caches (including private caches of peer-cores) for a cache block in dirty state and write it back to the PM. Effectively, each `clwb` instruction incurs a worst-case on-chip access latency.

- The overhead due to `pcommit` instructions. A `pcommit` does not complete until *all* writes that have been accepted to memory are persistent (regardless of which processor issued them).

Modern out-of-order (OoO) cores can often hide some access latencies, however the `sfence` instructions (required for correct ordering) preceding and following a `pcommit` will expose the latency of the `clwb` and `pcommit` operations. A PM write may take several times as long as a DRAM write [18] and drastically reduces the effectiveness of OoO mechanisms.

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

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

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

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

Synchronous Ordering (SO): Under this model, the necessary `clwb`, `sfence`, and `pcommit` operations are inserted ensuring data persistence to PM. Note that in case of PWQ, ensuring the data reaches the PM controller in the desired order is sufficient to guarantee correct recovery. Hence, SO requires only `clwb` and `sfence` operations, without any `pcommit`s. The latency of the cache flush operations is exposed on the execution critical path (at the `sfence`), but the PM write itself is not. Increased execution times for SO

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

Table 4.1: ***Synchronous ordering performance:*** *Slowdowns due to SO over volatile execution.*

(over a baseline volatile execution) for our PM-centric benchmarks (please see Section 4.4 for methodology details) are shown in Table 4.1:

4.2.3 Discussion

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

Second, a programmer must explicitly enumerate the addresses for which persistence is needed via flush operations. Although flexible, this interface greatly complicates software development. For example, one cannot easily construct a software library that provides transactional persistence semantics for a user-supplied data structure while hiding the details of the persistency model (the user must supply a list of addresses requiring `clwb` operations). In contrast, fence operations in relaxed consistency [17] and relaxed persistency [5] do not require addresses to be enumerated, facilitating synchronization primitives in libraries.

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

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

Interestingly, ARM has also recently announced a new instruction for persistence support as part of ARM V8.2 [26]. The new instruction `dccvap` (data cache clean virtual address to point of persistence), is similar to `clwb` in that, it forces a writeback of a cache block. However, unlike `clwb`, `dccvap` requires the writeback to become persistent (reach PM), rather than just being accepted at the PM controller, obviating the need for a separate `pcommit`-like instruction. The `dccvap` instruction implies a different synchronous persistency model than SO (which is based on Intel's `clwb`, and `pcommit` instructions). However, a complete ARM V8.2 specification is not yet available.

We can address synchronous ordering's deficiencies with delegated persist ordering. Next, we describe delegated persist ordering (Section 4.3), to implement the RCBSP model (detailed in Chapter 2.6).

4.3 Delegated persist ordering

We now describe delegated persist ordering, our implementation strategy for the R.

4.3.1 Design goals

Delegated ordering is based on four key design goals:

Enforce persist ordering: Under RCBSP, the persist order must match the store order given by the consistency model (as is necessary for strict persistency). Under ARMv7, intra-thread ordering arises from $FENCE$s, which divide the instructions within a thread into epochs. Stores to PM from the same epoch may persist concurrently (assuming they are not ordered by the fence cumulativity property, Eq 2.15, of a remote FENCE), however, stores from successive epochs must be totally ordered. Inter-thread persist ordering arises from coherence order (Eq 2.14) and fence cumulativity (Eq 2.15). When accesses conflict, corresponding persists (and their cumulative dependents) must occur in cache coherence order. Our implementation must observe, record, and enforce these persist dependencies.

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

Express lane for persists: Under SO, persists reach PM via successive writebacks from subsequent cache levels. Such an architecture optimizes for read performance at the expense of write latency—an appropriate trade-off for volatile memory. However, in PM-write-intensive applications, persist latency plays a major role in determining recoverable system performance. Some epoch persistency implementations buffer unpersisted stores in the cache hierarchy [8, 14]. However, buffering unpersisted stores in cache implies that a later store to the same address may not become globally visible until the prior store has persisted, leading to stalls. Moreover, performance-sensitive cache replacement policies may have to be modified to account for the desired persist order. We provide a separate persist path with dedicated storage that reduces persist latency and decouples persist order from cache eviction and store visibility.

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

Prior research has proposed sophisticated schedulers at DRAM controller to jointly optimize access latency, concurrency, and fairness [48, 49, 50]. Unlike DRAM controllers, the PM controller is additionally expected to honor persist ordering constraints. Initial research on PM-aware scheduling is under way [9, 51]. Our goal is to communicate persist ordering constraints to the PM controller as precisely and minimally as possible, providing it maximal scheduling flexibility, and yet without placing burdens on the cache hierarchy.

Delegated ordering succeeds in decoupling persistency enforcement *entirely* from cache management. Nevertheless, the goal of communicating *minimal* ordering constraints is aspirational; to reduce hardware complexity, our design serializes per-core persists into a single, partially ordered write queue at the PM controller, which is insufficient to represent a fully general dependence graph among persists. A single write queue cannot represent a dependence graph where two accesses must be ordered by an epoch boundary, but a third access is unordered with respect to both—we must place the third access in either the

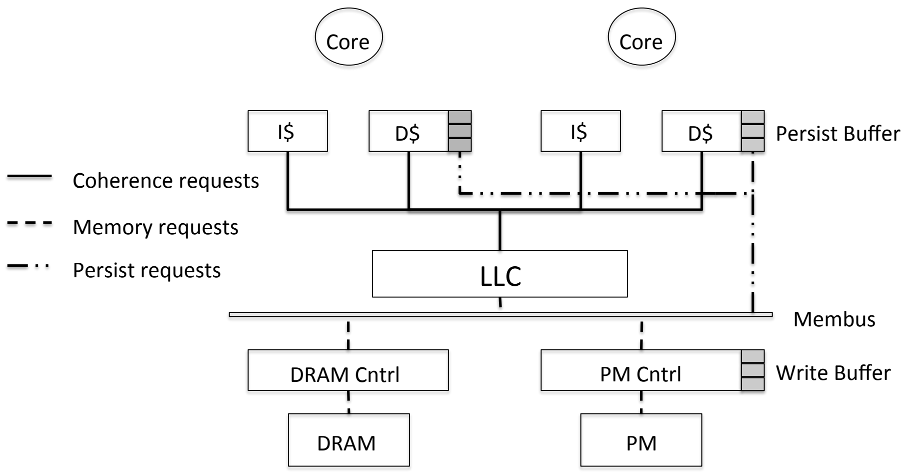


Figure 4.1: ***Proposed architecture design:*** *Our system architecture implementing delegated ordering for RCBSP, with a persist buffer at the L1 D-cache for every core and write queue at the PM controller.*

first epoch or the second, introducing an unnecessary constraint. Nevertheless, our design provides prodigious performance advantage; we believe the remaining gap to the performance of unordered volatile execution does not warrant additional hardware complexity to communicate minimal constraints.

4.3.2 System Architecture

Figure 4.1 shows our system architecture to implement delegated ordering for RCBSP. Responsibility for ensuring proper persist ordering is divided between persist buffers, located alongside each L1 D-cache, and the PM controller, which ultimately issues persist operations. The persist buffers each track persist requests and fences from their associated core to discover intra-thread persist dependencies and monitor cache coherence traffic to discover inter-thread persist dependencies. The buffers coordinate to then serialize their per-core persist operations into a single, partially ordered write queue at the PM controller through the path marked “Persist requests” in Figure 4.1. (We show a dedicated persist path for clarity; persist traffic may be multiplexed on existing interconnects, much like uncacheable memory requests or non-temporal store operations [32]). The persist buffers drain persists into totally ordered epochs. The PM controller may freely schedule within an epoch, but not across epoch boundaries.

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

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

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

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

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

4.3.3 Enforcing Dependencies

Persist buffers collaborate to jointly drain persists to the PM write queue, constructing unified epochs that are consistent with the persistency model ordering constraints at each core. We first describe at a high level how we ensure a correct drain order with reference to the example code sequences in Figure 4.2. We defer details to Section 4.3.5.

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

Persist-persist		Epoch-persist	
Core-0	**Core-1**	**Core-0**	**Core-1**
$St\ X = x$		$St\ X = x$	
		$FENCE$	
	$St\ X = x'$	$St\ A_L = a$	
			$r1 = Ld\ A_L$
			$St\ Y = r1$

Figure 4.2: ***Examples of persist dependencies:*** *Persist-persist dependency and epoch-persist dependency*

accesses, or between epochs, due to $FENCE$ operations becoming transitively ordered by conflicting accesses.

Figure 4.2 (left) illustrates a dependence between two persists. Persist-persist dependencies arise when two stores to the same persistent address are executed at two different cores. RCBSP mandates that the two stores persist in coherence order (via Eqs 2.14 and 2.1). At a high level, the dependency is discovered as part of the cache coherence transaction that transfers ownership of the cache block from Core-0 to Core-1. Core-0 will include in its write response an annotation with the ID of its persist, indicating that Core-1's persist must be ordered after it. This annotation will prevent the persist from draining from Core-1's persist buffer. When Core-0 drains its persist, Core-1 will observe the drain and resolve the dependency (clear the annotation), allowing its persist to then drain.

Figure 4.2 (right) illustrates a code sequence creating a dependence between an epoch and a persist. An epoch-persist dependency arises when an epoch and a persist on different threads are ordered due to intervening conflicting accesses (here via accesses to A_L) and fence cumulativity. Due to the $FENCE$ instruction on Core-0, we have $S^0_X \in G_A$ and $S^0_{A_L} \in G_B$. Since $L^1_{A_L}$ reads from $S^0_{A_L}$, we have $L^1_{A_L} \in G_B$. Further, since S^1_Y is data dependent on

$L^1_{A_L}$ (via register r1), we have $S^1_Y \in G_B$. Since S^0_X and S^1_Y are in G_A and G_B respectively, via fence cumulativity (Eq 2.15)) and strict persistency(Eq 2.1) we have $S^0_X \leq_p S^1_Y$.

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

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

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

To detect all conflicting accesses that follow a $FENCE$, the persist buffer must keep a record of all addresses read or written by its core until either the $FENCE$ drains or the processor executes a subsequent $FENCE$. Incoming coherence requests must be checked against the read- and write-sets to detect conflicts and discover dependencies. This requirement is similar to the read- and write-set tracking required to implement transactional memory [52]. As in many transactional memory designs, these sets may be maintained approximately, because false positives (identifying a conflict when there is none) introduce unnecessary persist ordering edges, but do not compromise correctness. However, given that the lifetime of a *FENCE* in the persist buffer is much smaller than an entire transaction, a simple design would suffice. We enumerate the steps of this exchange in detail in Section 4.3.5.

Note that persist-epoch and epoch-epoch dependencies may also arise, and are enforced by the hardware structures described in Section 4.3.4. It is also important to note that by tracking dependencies at an individual persist or *FENCE* granularity, our design does not suffer from the epoch dependency deadlocks identified in [14]. We omit examples in the interest of space.

4.3.4 Hardware Structures

Next, we describe the hardware structures required for delegated ordering. At each core, we provision a persist buffer, a pair of bloom filters for tracking read and write sets, and a register for tracking accumulated ordering dependences that must be applied to a yet-to-be-executed *FENCE* or persist. Persist requests and *FENCE* operations drain from persist buffers into the write queue at the PM controller. Figure 4.3 illustrates the hardware and fields in these structures.

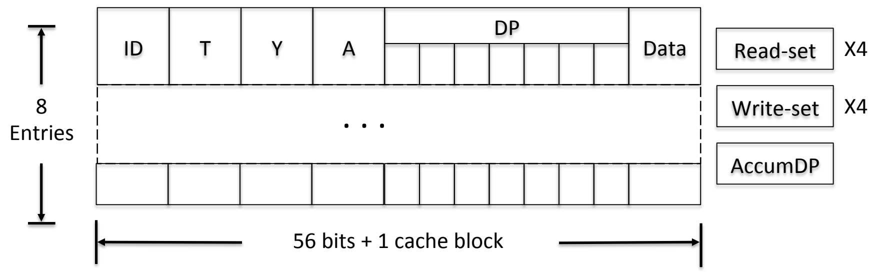


Figure 4.3: ***Persist buffer design:*** *Hardware structures required at each core.*

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

- **T** - The "Type"; persist request or fence.
- **A** - The cache block "Address" of a persist request; supports associative search by address. For $FENCE$s, this field associates the entry with a read/write-set.
- **D** - The "Data" cache block to be persisted.
- **ID** - An "ID" that uniquely identifies each in-flight persist or $FENCE$, comprising the core id and entry index. These IDs are used to track and resolve dependencies across persist buffers. We denote IDs as "{Core index}:{Entry index}".
- **Y** - The "Youngest" bit, marks the youngest persist request to a particular address across all persist buffers. This bit is set when a persist request is appended to the buffer and reset upon an invalidation of the cache block, indicating a subsequent store by another core. When set, this bit indicates this persist buffer must service coherence requests for the address.

- **DP** - An array of inter-thread dependencies for this entry. The number of fields in each "DP" entry is one less than the number of persist buffers, tracking at most one dependency from each other core. An entry can be drained to PM only when all its dependencies have been resolved (drained to PM write queue). The dependencies are tracked via IDs; When an ID drains on the persist bus, matching "DP" fields are cleared.

Read/Write Sets & AccumDP. We provision pairs of bl-oom filters to track addresses accessed by the core after a $FENCE$, as described in Section 4.3.3. Each persist buffer also requires an additional dependence ("AccumDP") register that is not associated with any persist buffer entry. "AccumDP" tracks dependences that are discovered via cache coherence and must be applied as order constraints against the next persist/$FENCE$ issued by the core. When a persist or $FENCE$ is appended to the persist buffer, its "DP" field is initialized from "AccumDP" and "AccumDP" is cleared.

PM Write Queue. The PM Write Queue, like buffers in a conventional memory controller, holds writes until they are issued to the PM storage array. When a $FENCE$ operation is drained from a persist buffer, it creates an epoch boundary across which persists may not be reordered.

Overheads. The storage overhead for each persist buffer entry is 72B. Considering the short duration a $FENCE$ spends in the persist buffer (due to the aggressive draining employed at the persist buffer), we use 32B bloom filters. An AccumDP register of 64B stores dependencies from all other persist buffers. In all, each persist buffer requires 8 persist buffer entries, 8 bloom filters for read/write sets (a maximum of 4 active $FENCE$

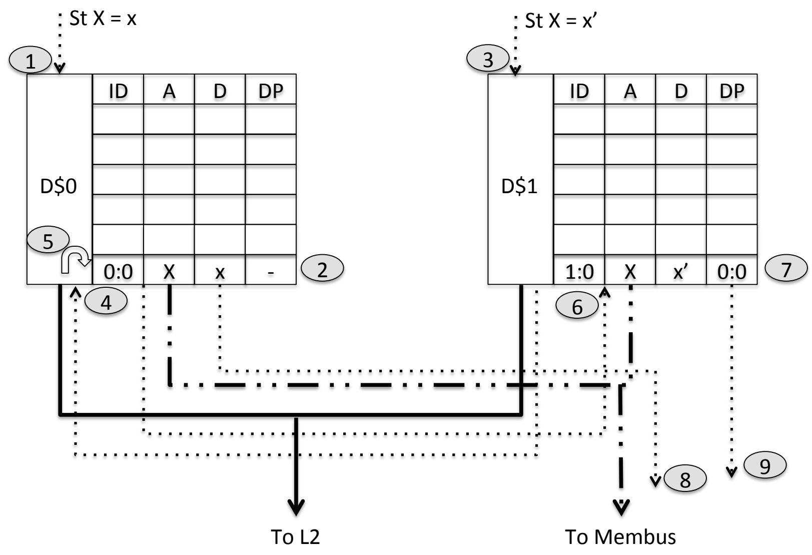


Figure 4.4: ***RCBSP in action - 1:*** *Resolving a persist-persist dependency.*

entries are allowed in a persist buffer) and one AccumDP register, placing the storage overhead at 896B/core. It is important to note that if a persist buffer becomes full (either due to exhaustion of entries or $FENCE$ slots), the corresponding L1-D$ is blocked until an entry drains from the persist buffer. The results presented in Section 4.4 account for all such blockages.

4.3.5 Detailed Examples.

We next walk through detailed examples that illustrate how persist buffers track inter-thread dependencies, with the aid of Figures 4.4 and 4.5

Persist-persist dependency. Figure 4.4 depicts the evolution of the persist buffer state for a persist-persist dependency (see Figure 4.2).

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

Epoch-Persist dependency. Figure 4.5 depicts the evolution of the persist buffer state an epoch-persist dependency.

(1) D\$0 receives a store request to a persistent address X. Assume that it hits at D\$0. **(2)** A new persist request is created for X with ID “0:0”. Assume no dependencies. **(3)** D\$0 receives a $FENCE$ request. **(4)** A new entry is created for the $FENCE$ with ID “0:1”.

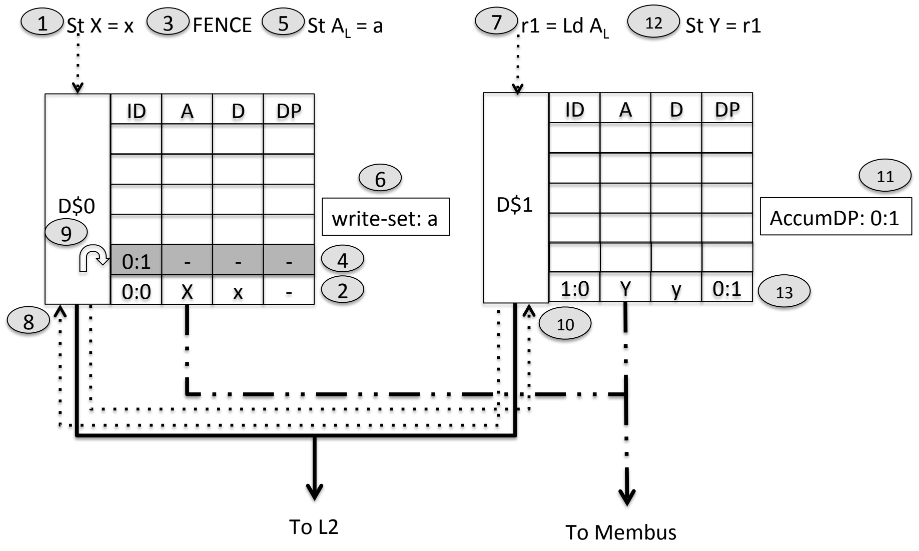


Figure 4.5: ***RCBSP in action - 2:*** *Resolving an epoch-persist dependency.*

(Gray entries indicate a $FENCE$). **(5)** D$0 receives a store request to a volatile address A_L. Assume it hits at D$0. **(6)** The volatile address A_L is recorded in the write-set associated with the $FENCE$. **(7)** D$1 receives a load request for address A_L. **(8)** D$1 sends a read request for address A_L to D$0. **(9)** D$0 snoops its cache and persist buffer, locating its cached copy of A_L. Since it has a pending $FENCE$, it compares address A_L to the write-set of the $FENCE$ and discovers a match, indicating a persist order dependence. The coherence response is annotated to indicate a $FENCE$ with ID "0:1" as a dependence. **(10)** D$1 receives the response with the latest data for A_L and the persist dependency annotation. **(11)** D$1 updates its "AccumDP" register to store the dependence on "0:1". This dependence will be applied to the next persist/$FENCE$ instruction executed at D$1. **(12)** D$1 receives a store request to a persistent address Y. Assume it results in a cache hit at D$1. **(13)** A new persist request is created with ID "1:0". The dependence on "0:1" from the "AccumDP" register is recorded and the register is cleared.

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

We note that our hardware might be substantially simplified under a programming model where conflicting accesses must be explicitly annotated as synchronization accesses, such as the DRF0 model [17]. In such models, only synchronization accesses may create ordering relationships between epochs in properly labeled programs. Unfortunately, ARMv7 does not mandate that racing accesses be annotated, requiring the additional complexity of the read- and write-set tracking.

4.3.6 Coalescing Persists

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

In our design, we drain persist operations eagerly at both the persist buffer and PM write queue, as soon as ordering constraints allow. However, in the absence of a $FENCE$, it may be advantageous to delay persist operations in an attempt to coalesce more persists. The

PM-write-intensive benchmarks we study do not afford additional coalescing opportunity, so we leave such optimizations to future work.

4.4 Evaluation

Core	8-cores, 2GHz OoO 6-wide Dispatch, 8-wide Commit 40-entry ROB 16/16-entry Load/Store Queue
I-Cache	32kB, 4-way, 64B 1ns cycle hit latency, 2 MSHRs
D-Cache	64kB, 4-way, 64B 2ns hit latency, 6 MSHRs
L2-Cache	8MB, 16-way, 64B 16ns hit latency, 16 MSHRs
Memory controller (DRAM, PM)	64/32-entry write/read queue
DRAM	DDR3, 800MHz
PCM	533MhZ, timing from [27]

Table 4.2: ***Simulator Configuration***

We compare delegated ordering against synchronous ordering for three different memory designs: DRAM, PCM, and PWQ (described in Section 4.2.2. We model DDR3 DRAM operating at 800MHz and PCM using timing parameters derived from [27] operating at 533MHz. We use PCM memory assumptions for PWQ. The PWQ design isolates the effect of fences and ordering instructions from PM access latency, while DRAM and PCM provide plausible performance projections. We model an 8-core system with ARM A15 cores in gem5 [47] using the configuration details in Table 4.2. RCBSP uses an 8-entry persist buffer at each core, allowing at most four in-flight $FENCE$s.

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

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

Table 4.3: ***Benchmark descriptions*** *(CKC =* `clwbs` *per 1000 cycles)*

Benchmarks: We study a suite of five PM-centric multi-threaded benchmarks, described in Table 5.3. Our Concurrent Queue is similar to that of Pelley [5]. The Array Swaps and RB Tree are similar to those in NV-Heaps [11]. Our TATP [46] and TPCC [45] benchmarks execute the "update location" and "new order" transactions, respectively the benchmark's most write-intensive transactions. We select these benchmarks specifically because they stress PM write performance; larger applications may amortize slowdown of PM-write-intensive phases over periods of volatile execution. As a heuristic for the "write-intensive"ness of the benchmarks, we report the number of `clwbs` issued per 1000 cycles per core (CKC) in Table 5.3. Array Swaps is our most write-intensive benchmark while RB Tree is the least, so we expect them to show the most and least sensitivity to persistency models, respectively.

4.4.1 Performance Comparison

Figure 4.6 contrasts the performance of RCBSP with SO, for three different memory designs: PWQ, DRAM, and PCM. Execution times in the figure are normalized to volatile

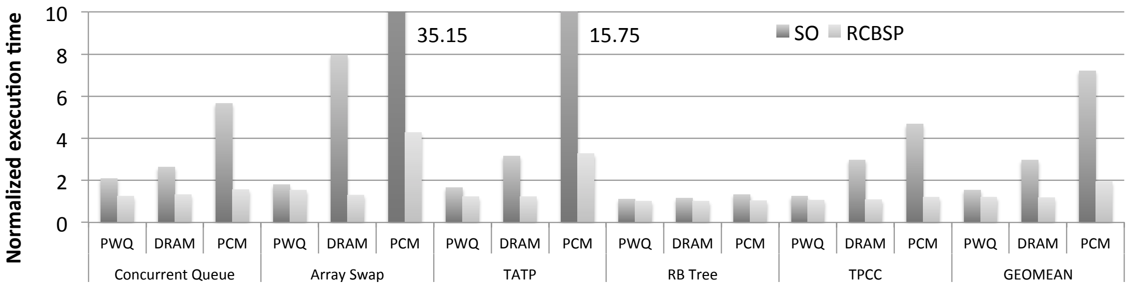


Figure 4.6: ***Evaluating SO vs RCBSP:*** *Normalized execution time for PWQ, DRAM, and PCM.*

execution with the corresponding memory design. The main takeaways from the figure are:

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

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

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

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

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

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

PM wear out: PMs like PCM suffer from wear out due to writes. RCBSP increases the overall writes to PM by 30%, averaged over all the benchmarks. This increase is to be expected as RCBSP is a hardware mechanism that tries to aggressively move persists from persist buffers to the PM write queue, while SO has the advantage of programmer inserted `clwb` instructions as triggers for writebacks, allowing better coalescing. Nevertheless, effective wear-leveling schemes have been proposed [53, 54, 55] to mitigate wear out and those solutions are orthogonal to persistency models and may be deployed with RCBSP.

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

CHAPTER V

Language-level persistency

5.1 Introduction

While the focus of the previous chapters was to improve the performance of persistent memory systems through software (Chapter III) and hardware (Chapter IV) mechanisms, the focus of this chapter is to ease the burden of programming such systems. Various persistency models have been proposed, but all of them have been specified at the instruction set architecture (ISA) level. That is, programmers must reason about recovery correctness at the abstraction of assembly instructions, an approach which is error prone and places an unreasonable burden on the programmer. The programmer must invoke ISA-specific mechanisms (via library calls or inline assembly) to ensure persist order, and often must reason carefully about compiler optimizations that may affect the relevant code. Since the ISA mechanisms differ in sometimes subtle ways, it is hard to write portable recoverable programs.

This chapter proposes a language-level persistency model that provides a single, ISA-agnostic framework for reasoning about persistency and can enable portability of recoverable software across language implementations (compiler, runtime, ISA, and hardware). We consider how to specify a persistency model that extends the *data-race-free* (DRF) consistency model [17] that is espoused by popular high-level programming languages like C++11 and Java. The DRF model is appealing for programmers because DRF guarantees a sequentially consistent (SC) execution for data-race-free programs [56], a guarantee often called "SC for DRF". At the same time, the DRF model admits compiler and hardware optimizations that reorder and optimize memory accesses between (and, in certain cases, across) synchronization. Such reorderings and optimizations are invisible to the programmer, because they cannot be observed without a data race.

One might hope that the simplicity of "SC for DRF" might extend naturally to memory persistency. Unfortunately, DRF is insufficient to define semantics for the PM state that recovery code may observe after a failure. The fundamental problem is that failures, such as operating system crashes, hardware lockups, or power disruptions, may occur *at any time*, and thereby introduce a data race into an otherwise race-free program: loads performed during recovery inherently race with stores before the failure. A failure may interrupt the atomicity of a critical section, exposing a partial (and possibly reordered) set of updates to PM to recovery code.

In Chapter 5.2, we explore a taxonomy of guarantees that a language-level persistency model might provide. Based on our taxonomy, in Chapter 5.3, we propose a concrete model, Acquire-Release Persistency (ARP), to extend the C++11 memory model. Further, in Chapter 5.4, we propose modifications to the C++11 language, compiler, ISA, and state-of-the-art persistency model implementations to improve performance by exploiting all of the available persist concurrency and scheduling flexibility provided by ARP.

In summary:

- We make a case for language-level rather than ISA-level persistency models.
- We explore a taxonomy of guarantees that a language-level persistency model might provide.
- We propose acquire-release persistency as an extension to the C++11 memory model. We demonstrate that writing applications to ARP rather than the ISA-level persistency model improves performance by up to 18.5% (8.9% avg.).
- We show that, with small extensions to C++11 and the ISA-level persistency model, we can eliminate further unnecessary persist dependencies, leading to speedups of up to 33.2% (19.8% avg.).

5.2 Design Exploration

All memory persistency models proposed to date [5, 1, 8, 14, 4, 26] have been specified at the ISA level. These models vary in the semantics they provide. To use these models, programmers must reason about and annotate their programs with assembly instructions to ensure correct persist order. Whereas the challenge of reasoning using assembly instructions might be mitigated by encapsulating assembly annotations in persistency-model-specific libraries, there is no easy way for programmers to develop portable recoverable software. Moreover, without a precise definition of language-level persistency semantics, otherwise legal compiler optimizations could render data structures unrecoverable. These challenges are reminiscent of the motivation for portable language-level memory consistency models [17]. Similarly, we argue for a language-level persistency model, so that programmers do not have to reason about ISA-specific assembly code while developing recoverable software. In this chapter, we explore possible approaches to design a language-level persistency model.

5.2.1 Failure and recovery

A persistency model imposes requirements on the fault-free execution of a program that writes to PM to ensure that, in the event of a failure, a programmer can rely on some set of guarantees on PM state. These guarantees then make it possible to develop recovery software that can repair data structure inconsistencies caused by interrupted updates. Past work on ISA-level persistency models has focused primarily on power failures (since PM state survives power failure). In this work, we consider fail-stop failures more broadly, e.g., program, run-time and operating system crashes, and hardware failures in addition to power failures. Notably, the trivial solution of providing battery backup to drain in-flight persist operations is not sufficient to tolerate all fail-stop failures. For example, an OS crash might expose that the compiler has reordered two persists and may compromise recovery. (We set aside PM media failures, as an orthogonal set of mechanisms are required to tolerate these, e.g., [57].)

After a failure, we assume the contents of all volatile state (processor registers including program counters, cache contents, volatile memory) as well as incomplete persists are lost, but the contents of persistent memory are retained. Recovery software then examines the persistent data structure, repairing it if necessary, so that normal operation may resume. In some cases, normal operation may be able to resume without any recovery. For example,

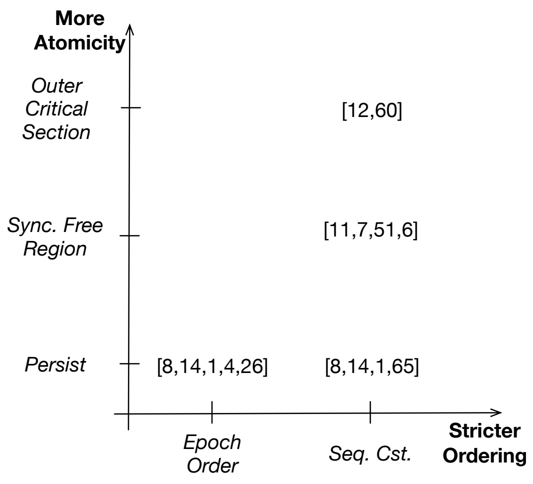


Figure 5.1: ***Design space of persistency guarantees:*** *Persistency guarantees explored along two dimensions, atomicity and ordering.*

prior work has demonstrated that wait-free concurrent data structures are inherently recoverable [58, 59]. We discuss the difficulty of writing recovery code under various persistency guarantees next.

5.2.2 Atomicity and ordering

A language-level persistency model has to provide programmers with guarantees on two orthogonal properties: (a) the granularity of failure-atomic regions (i.e., persists from one region are committed to PM atomically) and (b) the ordering of these regions. Programmers need both these guarantees to write correct recoverable software. Figure 5.1 shows the various options that a language may choose to provide for each of these guarantees and places existing academic and industrial proposals for persistent programming within this taxonomy. The granularity of failure atomicity can vary from an individual persist (8-byte atomic writes) to a synchronization free region (code between two synchronization accesses) to an outer critical section (code between the first lock acquired by a thread until the thread holds no locks). It is important to note that if a programmer desires a larger granularity of failure atomicity than what is natively provided by the language, she can achieve it through undo or write-ahead logging mechanisms [15]. Furthermore, the language may guarantee that these atomic units may be ordered sequentially (SC order) or provide a more relaxed ordering mechanism. For example, the language may provide *sequence points* that the programmer can use to break a thread into epochs. Failure-atomic units within an epoch are unordered, but epochs are sequentially ordered (as in [8, 5, 14]).

5.2.2.1 DRF Persistency?

Popular high-level programming languages like C++11 and Java espouse the data-race free (DRF) memory model to enable parallel programming. One of the key advantages of the DRF memory model is that, for DRF programs, it allows programmers to reason about memory access interleaving at the granularity of *synchronization-free regions*, rather than individual accesses. The lack of any data races implies that programmers are assured that the writes in any synchronization-free region will become visible atomically to other threads. Compilers exploit this guarantee to perform optimizations that reorder memory accesses within a region [56], which wouldn't be permissible otherwise.

In addition to its sequentially consistent synchronization operations, C++11 also provides *low-level atomics*, which allow the programmer to label individual synchronization operations with specific memory ordering semantics. A program that uses relaxed atomics has well-defined memory semantics, but loses the "SC for DRF" guarantee. That is, programs with low-level atomics do not necessarily exhibit SC execution [56]. We consider how persistency models might interact with C++11 programs both with and without low-level atomics.

One might argue that it is natural to extend the SC for DRF consistency guarantee to recovery code that executes after failure. That is, it would help programmers in writing recovery code to provide a failure-atomicity guarantee for regions of persists. Such a guarantee would hide compiler or hardware memory access reordering from the programmer, and recovery code need only consider memory states that can arise at synchronization points.

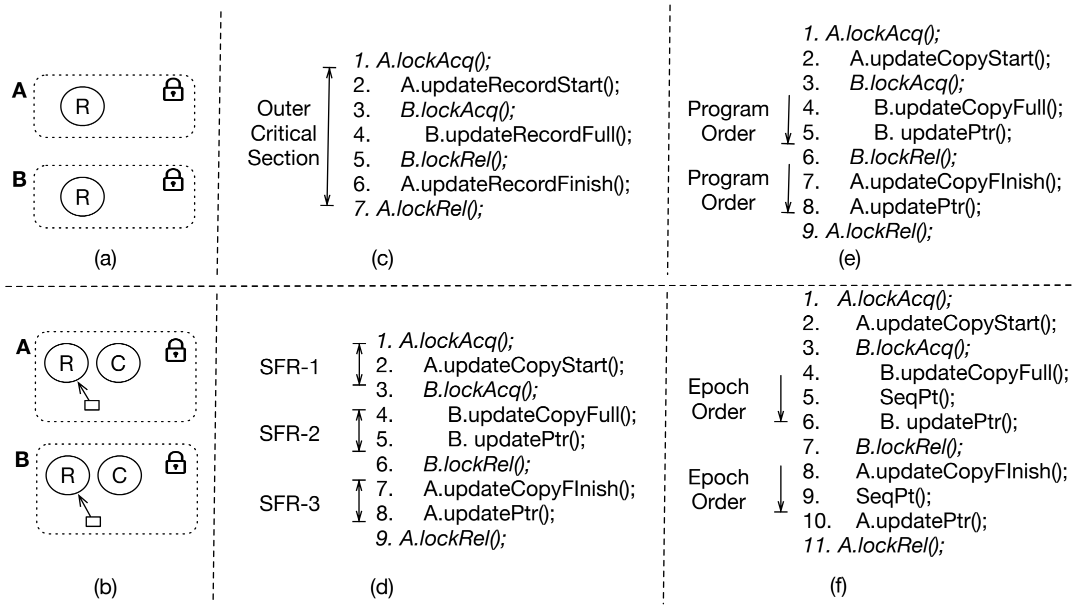
However, arbitrary fail-stop failures make such atomicity challenging to enforce. If writes may persist from a synchronization-free region incrementally, recovery code may observe intermediate memory state within the region, breaking the atomicity guarantee that is core to the DRF model: recovery code is not guaranteed to observe sequentially consistent state.

There are two ways to resolve the conflict between arbitrary failures and DRF:

- **Enforce atomicity:** Programming languages may demand that the implementation provide a programmer-transparent mechanism to ensure failure-atomicity (e.g., undo logging in the hardware or runtime).

- **Forego atomicity and provide only ordering:** Alternatively, languages may forego guarantees that program regions appear atomic to post-fault recovery code, and instead guarantee only the relative order of persists, much like ISA-level persistency models. (Note that, in fault-free execution, the SC for DRF guarantees still apply).

Next, we explore design alternatives and their implications for the programmer, compiler, and implementation.



```
1. A.lockAcq();
2.    A.updateRecordStart();
3.    B.lockAcq();
4.       B.updateRecordFull();
5.    B.lockRel();
6.    A.updateRecordFinish();
7. A.lockRel();
```

```
1. A.lockAcq();
2.    A.updateCopyStart();
3.    B.lockAcq();
4.       B.updateCopyFull();
5.       B. updatePtr();
6.    B.lockRel();
7.    A.updateCopyFInish();
8.    A.updatePtr();
9. A.lockRel();
```

```
1. A.lockAcq();
2.    A.updateCopyStart();
3.    B.lockAcq();
4.       B.updateCopyFull();
5.       B. updatePtr();
6.    B.lockRel();
7.    A.updateCopyFInish();
8.    A.updatePtr();
9. A.lockRel();
```

```
1.  A.lockAcq();
2.    A.updateCopyStart();
3.    B.lockAcq();
4.       B.updateCopyFull();
5.       SeqPt();
6.       B. updatePtr();
7.    B.lockRel();
8.    A.updateCopyFInish();
9.    SeqPt();
10.   A.updatePtr();
11. A.lockRel();
```

Figure 5.2: ***The taxonomy of persistency guarantees analyzed via a running example:*** *(a) Two objects (A,B), each with a record (R) and lock assuming the language provides failure-atomicity of outer critical sections. (b) Two objects (A,B), each with a record (R), a lock, a shadow copy (C), and a pointer to ensure failure-atomicity assuming the language does* not *provide failure-atomicity of outer critical sections. (c) Code and failure-atomic region when the language guarantees sequentially consistent failure-atomic outer critical sections. (d) Code and failure atomic regions when the language guarantees sequentially consistent failure-atomic synchronization free regions. (e) Code and orderings when the language guarantees sequentially consistent persists. (f) Code and orderings when the language guarantees epoch ordered persists.*

5.2.3 A Taxonomy of Persistency Guarantees

We use a running example to highlight how alternative guarantees can be used to ensure recovery correctness. Consider a program with two shared objects, A and B, each with record fields (R) protected by a lock (Fig. 5.2 (a)). Suppose the correctness requirement is that the fields of each object must be updated atomically with respect to failures and with respect to other threads. Now, consider a piece of code that acquires the lock for object A, starts modifying it, and then must also modify B, which it does within a nested critical section (Fig. 5.2 (c)). The two locks assure atomicity with respect to concurrent access from other threads in fault-free execution. The persistency model must enable the programmer to write code that can recover to a correct state (i.e., each object to either its initial or final state) in the event of failure.

For languages which do not guarantee the failure-atomicity of the entire update of objects A and B, we provide alternative designs for A and B that rely on shadow logging, shown in Fig. 5.2 (b). The update is performed on a shadow copy (C) of the object (rather than an in-place update in the object itself). Once the shadow copy has been updated, a pointer is atomically switched to indicate that the copy is committed. For this approach to be correct with respect to recovery, the language must guarantee that the pointer switch persists no earlier than the updates to the shadow copy (assuming appropriate annotations from the programmer). We next consider four different sets of guarantees that a language may provide to enable such recovery. We discuss them in the order of decreasing constraints on persists.

5.2.3.1 Sequentially consistent, failure-atomic outer critical sections

Description: All the persists from an outer critical section (from first lock acquire till no locks are held) are guaranteed by the language implementation to be failure atomic. Further, different outer critical sections must persist in sequentially consistent order.

Example: Fig. 5.2 (c) shows code which updates both objects in nested critical sections. As the entire outer critical section (from line 1 to 7) is failure-atomic, the condition for correct recovery (each element is individually atomically updated) is trivially met.

Programmability: The idea of sequentially consistent failure-atomic outer critical sections was first explored by Chakrabarti [12, 60]. The central appeal of this programming guarantee is that, by ensuring failure-atomicity of entire critical sections, the state of persistent memory post-recovery always reflects a state that would have arisen in fault-free execution and when no threads holds a lock. When no locks are held, shared data structures are always in a consistent state. So, no recovery code is needed; the programmer is assured that her data structures are always in a consistent state post-recovery.

Implementation: Chakrabarti [12] provides a software undo-logging mechanism to ensure failure-atomicity of critical sections. Note that the software logging occurs outside of the language's memory model and must be implemented by the runtime system using ISA-level memory persistency.

Compiler optimizations: Since critical sections persist atomically, any compiler optimizations valid within a critical section under fault-free execution remain valid; optimization is unaffected by the persistency guarantee.

Challenges: While failure-atomic critical sections provide an intuitive guarantee, several challenges must be addressed:

1. *Guarantees for programs without critical sections:* This approach provides no semantics for programs without critical sections (e.g., single-threaded programs). It is unclear how system calls within critical section should be addressed.

2. *Implementation complexity:* Overlapping critical sections introduce considerable complexity to the logging and log-pruning mechanisms. They may cause cyclic dependencies, which must be carefully resolved [12].

3. *Large critical sections:* Providing atomicity guarantees over large regions increases the forward progress loss upon a failure. Large and nested critical sections intro-

duce hardware logging challenges similar to those seen with unbounded transactional memory designs [61].

4. *Alternatives to logging:* Many data structures can be made recoverable without logging (e.g., wait-free data structures [59, 58]). Furthermore, logging can often be optimized for special cases to improve efficiency (e.g., static transactions [15]). A generic, programmer-transparent logging mechanism will miss these optimization opportunities.

5.2.3.2 Sequentially consistent, failure-atomic synchronization free regions

Description: All persists from a synchronization free region (SFR) are guaranteed to be failure-atomic. Regions must persist in a sequentially consistent order. An SFR is defined as code on the same thread separated by two synchronization accesses, or two system calls, or a synchronization access and a system call [62, 63, 64]. For transaction-based code, the outer critical section is from transaction begin to transaction end. Note that, for nested transactions, we assume that inner transactions are flattened into a single outer transaction.

Example: As the modifications to object A span SFRs (due to nested locking), we must use shadow logging to achieve failure-atomicity (Fig. 5.2 (b)). Fig. 5.2 (d) shows the code required to ensure that the pointer switch does not persist earlier than the shadow copy update under sequentially consistent failure-atomic SFRs. For object B, since the shadow copy update and pointer update are in the same SFR (SFR-2), the update is failure-atomic. For object A, since the pointer update cannot persist earlier than the partial copy update in SFR-1 (program order of SFRs) or the partial copy update in SFR-3 (failure-atomicity guarantee of an SFR), failure-atomicity is preserved.

Programmability: For transaction-based programs or programs without overlapping critical sections, SFRs and critical sections are the same. However, for programs which have overlapping critical sections (as in Fig. 5.2 (d)), a critical section may span multiple SFRs. For such programs, partially completed critical sections may be visible post-recovery. While developing recovery software, the programmer must be cognizant of this possibility. If failure-atomicity of outer critical sections is desired, the programmer must add roll-back mechanisms for partially completed critical sections.

Implementations: Various logging proposals can provide failure-atomicity for SFRs. However, most focus only on transaction-based code [6, 51, 11, 7]. While transactions simplify logging, they are not general enough to be provided as a language guarantee [60].

Compiler optimizations: Since SFRs persist atomically, optimizations within an SFR remain valid.

Challenges: Several challenges remain under this model:

1. *Large SFRs:* Large SFRs pose the same challenges as large outer critical sections, as discussed above.

2. *Alternatives to logging:* As with failure-atomic critical sections, the implementation must provide a generic logging mechanism that will miss data-structure-specific optimization opportunities.

5.2.3.3 Sequentially consistent persists (SCP)

Description: Individual stores persist atomically. All stores persist in sequentially consistent order.

Example: Fig. 5.2 (e) shows the code required to ensure that the pointer switch does not persist earlier than the shadow copy update under SCP. Since the shadow copy update precedes the pointer switch in program order (lines 4-5 and 7-8 in Fig. 5.2 (e)), failure-atomicity of the object is preserved.

Programmability: Since only the atomicity of individual persists is guaranteed, the programmer must implement failure-atomicity mechanisms if larger granularities are required. The programmer can rely on sequentially consistent order of persists while implementing the logging mechanisms.

Implementation: Under SCP, the implementation is no longer required to provide a logging mechanism; it is expected that the programmer will implement mechanisms needed for failure-atomicity over multiple persists. Persists drain incrementally to PM, but, the compiler and hardware must ensure that they drain in program order. Under some ISA-level persistency models, stores may need to be flushed individually with explicit instructions [4, 26] or by inserting fence instructions after each store [8, 1]. Hardware can also guarantee SCP via hardware logging [14] or via transparent checkpointing [65].

Compiler optimizations: A consequence of the sequential consistency requirement on stores is that compiler or hardware optimizations that reorder persistent writes are no longer allowed. An implementation may choose to provide atomicity over some regions to allow intra-region reordering, as in speculative consistency implementations [66, 67].

Challenges: While SCP does not require any annotations to ensure persist order, it entails the following challenges:

1. *In-program logging:* The programmer must implement failure-atomicity mechanisms. However, she is also free to leverage data-structure specific recovery optimizations. Notably, some (e.g., wait-free) data structures require no logging at all [59, 58].

2. *ISA-level persistency mismatch:* The ISA-level persistency models proposed to date require persistent stores to be flushed individually and fence/barrier instructions to enforce order. For such ISAs, the compiler must insert copious (and performance-sapping) annotations.

3. *Lost compiler optimizations:* Straight-forward implementation of SCP precludes all compiler and hardware optimizations that reorder writes.

4. *Performance:* Prior works [8, 5] observe that preserving program order is expensive (due to high PM access latencies) and often unnecessary. Instead they argue for an epoch-based ordering of persists, where programmers use special barrier instructions to indicate where ordering is required.

5.2.3.4 Epoch ordered persists (EOP)

Description: This guarantee is derived from ISA-level epoch persistency models [8, 5, 14] proposed in prior research. Special *sequence point* (SP) annotations may be used by a programmer to break a thread into epochs; persists across epochs are ordered, but may be reordered within epochs. Persists on different threads are still governed by synchronization order.

Example: Since the shadow copy update is ordered before the pointer switch via an intermediate SP (lines 5 and 9 in Fig. 5.2 (f)), the failure-atomicity of each object is ensured.

Programmability: Similar to programming under SCP, programmers may have to implement failure-atomicity mechanisms in software. However, the programmer may no longer rely on program order, but instead must issue explicit sequence points when ordering guarantees are required, complicating the implementation of recoverable data structures.

Compiler optimizations: The compiler (and hardware) may reorder persists within epochs (e.g., between two sequence points), but may not allow persists to reorder across epochs.

Implementation: Many approaches to implement epoch-persistency models in hardware have been proposed [8, 14, 1]. Any of these satisfy the requirements of EOP.

Challenges: While EOP alleviates many challenges that arise under SCP, some challenges remain:

1. *In-program logging:* The programmer must use explicit sequence points to ensure recovery correctness rather than simply relying on program order.
2. *Compiler optimizations:* The compiler may not reorder persists across sequence points.

5.2.4 Discussion

Each of the four sets of guarantees analyzed in the previous section have their own advantages and disadvantages. Ignoring performance concerns, programmers would clearly want to choose sequentially consistent failure-atomic outer critical sections as the guarantee that languages should provide, as it requires no logging from the programmer. Instead the compiler, runtime or hardware are responsible for providing failure-atomicity of critical sections. However, indications from hardware vendors (e.g., Intel [4], ARM [26]) are that future processors are only going to guarantee the atomicity of individual persists. Because compiler or runtime logging mechanisms [12] required to ensure failure-atomicity must be general, they cannot take advantage of data-structure-specific optimizations (e.g., wait-free recoverable data structures [58], static transactions [15]).

Given that all the other sets of guarantees would require programmers to implement some in-program logging, we argue that the language should provide the most fundamental atomicity guarantee (individual persists); software solutions (e.g., in expert-crafted libraries) for larger atomic regions can be layered on top to reduce programmer burden. In the rest of this chapter, we focus on analyzing, designing, and evaluating implementations of SCP and EOP.

5.3 Acquire-Release Persistency

We next propose acquire-release persistency (ARP), a persistency model for C++11 based on the EOP approach.

5.3.1 Definition

We formally define ARP as an ordering relation over memory events—loads and stores on data variables, acquire and release operations on atomic variables—and sequence points.

By "thread", we refer to execution contexts—cores or hardware threads. We use the following notation:

- A_x^i: An acquire operation from thread i on an atomic variable x
- R_x^i: A release operation from thread i on an atomic variable x
- SP^i: A sequence point from thread i
- M_x^i: A data load/data store/acquire/release/sequence point by thread i (on variable x)

We use the following notation for ordering dependencies between memory events:

- $M_x^i \xrightarrow{po} M_y^i$: M_x^i is program ordered before M_y^i
- $R_x^i \xrightarrow{sw} A_x^j$: A release operation on atomic variable x in thread i "synchronizes with" [56] an acquire operation on atomic variable x in thread j.

We reason about an ordering relation over all memory events, *persist memory order* (PMO), denoted as $\leq_p$. An ordering relation between stores in PMO implies the corresponding persist actions are ordered; that is,

$$A \leq_p B \rightarrow B \text{ may not persist before } A.$$

Memory events can be ordered in PMO using a combination of *intra-thread* and *inter-thread* ordering relations. A programmer can use the following guarantees to ensure the desired order of events in PMO.

Ensuring intra-thread ordering: Based on the ordering guarantees provided by the language (via sequence points 5.2.3.4) intra-thread ordering can be achieved as follows:

- **Sequence point guarantee:** If two memory events on the same thread are separated by a sequence point in program order, then they are ordered in PMO. Formally:

$$(M_x^i \xrightarrow{po} SP^i \xrightarrow{po} M_y^i) \rightarrow M_x^i \leq_p M_y^i \tag{5.1}$$

Note that we use the existing `std::atomic_thread_fence` instruction in C++11 as our sequence points.

Ensuring inter-thread ordering: Inter-thread ordering is achieved using the "synchronizes with" [56] relationship between a release and a subsequent acquire operation.

- **Synchronization guarantee:** If two memory events are ordered via synchronization accesses, then they are ordered in PMO. Formally:

$$(M_x^i \xrightarrow{po} R_s^i \xrightarrow{sw} A_s^j \xrightarrow{po} M_y^j) \rightarrow M_x^i \leq_p M_y^j \tag{5.2}$$

Furthermore, PMO is a *transitive* (and irreflexive) ordering relationship, that is:

- **Transitivity guarantee:** If A is ordered before B in PMO and B is ordered before C in PMO, then A is ordered before C in PMO. Formally:

$$(M_x^i \leq_p M_y^j) \wedge (M_y^j \leq_p M_z^k) \rightarrow M_x^i \leq_p M_z^k \tag{5.3}$$

A programmer can use the above three guarantees to express the desired order of persists. It is the responsibility of the compiler to translate these constraints to machine code

ARP memory events	RCBSP mapping	Ideal mapping
Data load/store on addr a	`ldr/str a;`	`ld/st a;`
Seq. Pt. (SP)	`dmb ish;`	`full;`
Store Release on addr a	`dmb ish; str a;`	`rel a;`
Load Acquire on addr a	`ldr a; dmb ish;`	`acq a;`

Table 5.1: ***Compiler transformations from ARP to RCBSP:*** *Mapping from ARP memory events to RCBSP [1], which is based on ARMv7a. Ideal mappings from ARP would be to an ISA which supports release consistency.*

using the ISA-level persistency model, and it is the responsibility of the hardware to enforce these constraints. Enforcing constraints on persists is expensive (due to the high access latencies of PMs), so, it important to co-design language-level persistency models, ISA-level persistency models, and hardware implementations such that only the necessary constraints are enforced.

5.3.2 Mapping to ISA-level persistency

While ARP can be translated to any epoch-based ISA-level persistency model [8, 5, 14, 4, 1], in this chapter, we provide mappings to the state-of-the-art RCBSP model [1]. One of the advantages of RCBSP is that it is a *strict persistency* model; that is, if the compiler ensures that two stores are ordered by the ISA-level consistency model, then the corresponding persists are ordered as well.

Table 5.1 lists four important kinds of memory events in ARP and how they map to the machine ISA under RCBSP. Non-synchronization data accesses translate to regular loads and stores. A sequence point is translated to a full fence instruction (`DMB ISH` in ARM). A store release operation is translated to a full fence followed by a regular store instruction. A load acquire operation is translated to a regular load followed by a full fence instruction.

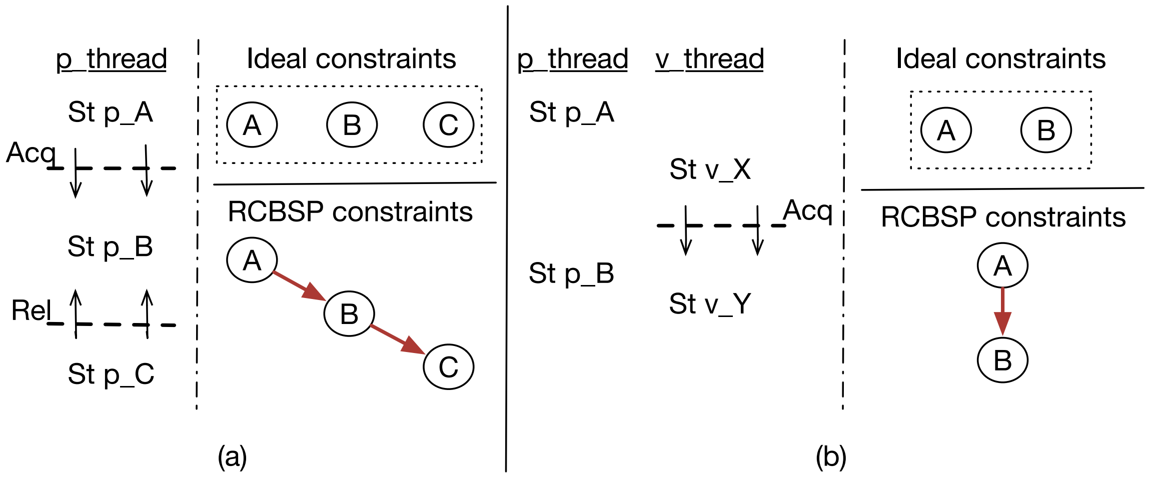


Figure 5.3: ***Unnecessary persist constraints enforced by RCBSP:*** *(a) Unnecessary constraints enforced due to hardware being oblivious to fence directions. (b) Unnecessary constraints enforced due to lack of language level semantics to express volatile fences.*

5.3.3 Fence directionality

We next discuss two sources of unnecessary persist constraints that arise when mapping ARP to RCBSP. The first arises because of the differences between the underlying consistency models of ARP and RCBSP. While ARP (and the C++11 memory model) is based on release consistency [68], RCBSP is based on the more conservative ARMv7 consistency model. Hence, RCBSP is oblivious to uni-directional acquire and release operations that are available in C++11 and ISAs based on release consistency (e.g., ARMv8).

ARP allows programmers to use uni-directional synchronization operations (*acq* and *rel*) to order memory accesses. Both *acq* and *rel* operations are usually used to ensure memory accesses within a critical section do not "leak out", however, they allow memory accesses from outside the critical section to "leak into" the critical section. However, as ARMv7 does not distinguish between an *acq* and a *rel*, compilers are forced to use a full *fence* (`DMB ISH` [69], which precludes memory access reordering in both directions) for both of them. Figure 5.3 (a) shows the unnecessary ordering constraints caused by using a full *fence* instead of uni-directional *acq* or *rel*. A thread performs stores to three persistent addresses, A, B, and C. The stores to A and B are separated by an *acq*, while stores to B and C are separated by a *rel*. As per the semantics of ARP, all three of A, B and C are considered concurrent and may execute and persist in any order. However, replacing the *acq* and *rel* with a full *fence* requires that persists to A, B, and C are serialized. So, persist order is overconstrained by RCBSP. Table 5.2 shows the increase in persists per epoch possible by distinguishing the required directionality of a *fence*. It is important to note that such over-constraints on persist order are not specific to RCBSP, but arise whenever the ISA-level persistency model is not as relaxed as the language-level persistency model.

Table 5.1 also shows the mapping of the four C++11 memory events to an ISA that provides uni-directional acquire and release operations (e.g., ARMv8). A store release is translated to a corresponding release instruction and a load acquire to an acquire instruction.

5.3.4 Conflating concurrency control with recoverability

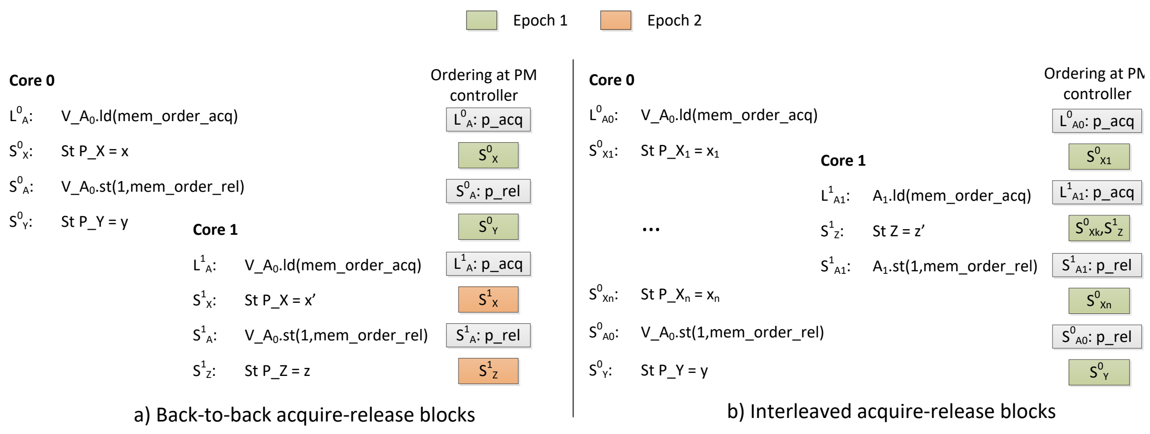
The second set of unnecessary constraints are caused by the lack of mechanisms to allow programmers to annotate constraints that are required for concurrency control, but not for recoverability. Consider the case in Figure 5.3 (b), where two unrelated threads (*p_thread* and *v_thread*) issue memory accesses. RCBSP serializes persists and *fence*s from all cores into the write queue at the PM controller. So, if the *acq* from *v_thread* happens to arrive at the PM controller between the two persists requests from *p_thread*, then the PM controller will place them in different epochs, introducing an unnecessary constraint.

Benchmark	*FENCE* directionality	Volatile annotations
cq	1.4×	2.1×
pc	1.9×	5.7×
sps	1.7×	2.9×
TATP	2.7×	3.6×
TPCC	1.6×	13.1×
YCSB_A	1.8×	5.9×

Table 5.2: ***Effect of removing unnecessary persist constraints:*** *Increase in persists per epoch when the memory controller is aware of FENCE directionality and volatile FENCEs.*

Ideally, we would like the hardware to enforce only constraints required for recovery, however, accurately tracking these persist constraints over multiple cores is challenging. Instead, we observe that programmers can identify *acq* and *rel* memory operations that have no persist semantics (i.e., they are required for concurrency control but were never meant to order persists). For example, some threads may never issue any persist operations and communicate only among themselves [9]. With minor extensions to the C++11 memory model, programmers can annotate *acq* and *rel* that do not have persist semantics as non-persistent or "volatile". And, with appropriate extensions to the machine ISA, this information can be passed to hardware, avoiding unneeded persist constraints to improve performance. Table 5.2 shows the increase in persists per epoch possible by making sure that volatile *acq* and *rel* are not sent to the PM controller.

Discussion: Mitigating the two sources of unnecessary persist constraints allows more persists to join each epoch. Larger epochs in turn provide the PM controller greater flexibility to schedule and batch persist operations, improving persist concurrency, leading to substantial performance gains since PM write latencies are so high.



Epoch 1
Epoch 2
Core 0
Ordering at PM controller
L0A: V_A0.ld(mem_order_acq)
S0X: St P_X = x
S0A: V_A0.st(1,mem_order_rel)
S0Y: St P_Y = y
Core 1
L1A: V_A0.ld(mem_order_acq)
S1X: St P_X = x'
S1A: V_A0.st(1,mem_order_rel)
S1Z: St P_Z = z
L0A: p_acq
S0X
S0A: p_rel
S0Y
L1A: p_acq
S1X
S1A: p_rel
S1Z
a) Back-to-back acquire-release blocks
Core 0
Ordering at PM controller
L0A0: V_A0.ld(mem_order_acq)
S0X1: St P_X1 = x1
...
S0Xn: St P_Xn = xn
S0A0: V_A0.st(1,mem_order_rel)
S0Y: St P_Y = y
Core 1
L1A1: A1.ld(mem_order_acq)
S1Z: St Z = z'
S1A1: A1.st(1,mem_order_rel)
L0A0: p_acq
S0X1
L1A1: p_acq
S0Xk,S1Z
S1A1: p_rel
S0Xn
S0A0: p_rel
S0Y
b) Interleaved acquire-release blocks

Figure 5.4: Allocation of epochs for unidirectional fences in the PM controller.

Algorithm 1 Epoch allocation in PM controller

Input: type of barrier **barrierType**, persists **S**, wait for acquire flag **waitForAcq**

```
1: if waitForAcq && barrierType == p_acq then
2:     epoch = epoch + 1
3:     waitForAcq.reset()
4: else if barrierType == p_rel then
5:     waitForAcq.set()
6: else
7:     S.epoch = epoch
8: end if
```

5.4 Extending RCBSP for ARP

We extend RCBSP [1] to support ARP with unidirectional and volatile fences. The key change to the RCBSP hardware is to allow a single persist buffer entry to represent both a persist and a *fence* for store-release and store-fence operations. However, the PM controller's epoch-based scheduling mechanism must be redesigned to account for the *fence*-directionality that ARP provides.

5.4.1 Enforcing unidirectional fences

In RCBSP, the PM controller tracks persists in epochs and maintains a current epoch number, to which newly arriving persists are assigned. The PM controller drains persists in epoch order. Since RCBSP only supported full *fence*s, the PM controller increments the current epoch number upon each *fence*. We extend RCBSP to support ARP's unidirectional fences by changing the algorithm for incrementing the current epoch number. The current epoch number is incremented only: (1) upon receiving a full *fence*, or (2) upon receiving the first *acq* after a *rel*. A full *fence* always creates a new epoch, since it orders all persists that precede/follow it. However, the respective directionalities of an *acq* and a *rel* mean that only a (*rel* + *acq*) combination disallows persists prior to the *rel* from reordering with persists after the *acq*. Successive *acq*s and *rel*s do not impact the current epoch number. Algorithm 1 provides pseudo-code for the epoch management algorithm, which uses the waitForAcq flag to indicate whether the next acquire operation should open a new epoch. We illustrate the assignment of epochs for the following scenarios:

Conflicting acquire-release blocks: Figure 5.4 (a) illustrates two threads updating a conflicting address `P_X` in persistent memory. Core-0 acquires a lock that resides in volatile memory V_A_0 (L_A^0), sets the persistent location `P_X` (S_X^0), and then releases the lock (S_A^0). It then sets a persistent location `P_Y` (S_Y^0) after releasing the lock. Core 1 then proceeds to acquire lock V_A_0 (L_A^1) and updates location `P_X` (S_X^1). It then releases the lock and sets persistent location `P_Z` (S_Z^1). Assume that the current epoch number is 0 at the start of this code sequence and is incremented to 1 upon the first *acq*.

The dependency tracking mechanism at the persist buffers preserves the happens-before ordering between the release of lock S_A^0 by core-0 and acquire of lock L_A^1 by core-1, and drains the stores to the PM controller in the order shown in Figure 5.4 (a). At the PM controller, upon receiving the lock release by core-0 *rel* S_A^0, the PM sets waitForAcq, indicating that the next acquire must initiate a new epoch. The next persist, S_Y^0, is still assigned to ongoing epoch 1. Upon receiving *p_acq* L_A^1 by core-1, because waitForAcq is set, the current epoch is incremented to 2. Subsequent persists S_X^1 and S_Z^1 arrive and are assigned to epoch 2. Note that persists lying between a release and subsequent acquire may join either epoch. To minimize re-ordering complexity, we assign these persists to the prior epoch. The persists in epoch 2, S_X^1 and S_Z^1, cannot be re-ordered with the persists in epoch 1, S_X^0 and S_Y^0. As a result, the shared address `X` is updated in the persistent memory in the order the stores were executed.

Interleaved acquire-release blocks: The example in Figure 5.4 (b) depicts two threads accessing separate regions of persistent memory by acquiring distinct locks. As in the previous example, the PM controller increments the epoch number to 1 upon receiving *acq* L_{A0}^0 and resets the waitForAcq flag. As core 1 then acquires a different lock, *acq* L_{A1}^1 has no dependency in the persist buffer and drains immediately to the PM controller. Since there has been no release since the last acquire (waitForAcq is clear), *acq* L_{A1}^1 does not increment the epoch number. Upon receiving *rel* S_{A1}^1 from core 1, the waitForAcq flag is set. The subsequent release operation *rel* S_{A0}^0 has no effect; the arriving persist S_Y^0 is assigned to epoch 1. Note that the persists within both critical sections are concurrent and join the same epoch.

5.4.2 Extensions for volatile annotations

To allow programmers to annotate *acq* and *rel* as being "volatile-only", we propose to add an argument to C++11 sync (`std::atomic`) variable accesses. In addition to the memory order argument (`std::memory_order`), we introduce a new argument that identifies if the access has persistent semantics (`bool is_persistent`). By default, sync accesses are labeled as persistent (`is_persistent = true`). For instance, the new definition of a load on an atomic variable (x), is then:

```
x.load(std::memory_order, bool is_persistent = true);
```

This new load operation is synchronized with other variables in the program as per the specific memory order, however, the `is_persistent` flag is used to inform the hardware whether the load operation is intended to have any impact on the order of persists. Similarly, in the machine ISA, we add "volatile" (non-persistent) versions of *acq*, *rel*, and *fence* instructions, allowing the compiler to map persistent/volatile sync accesses to the ISA. The persistent and volatile variants of *acq*, *rel*, and *fence* have the same behavior, except that the volatile versions are not sent to persist buffers and have no effect on subsequent persists.

5.5 Evaluation

We study the relative performance of five different persistency models: (a) SCP (from section 5.2.3.3), (b) ISA-level RCBSP, (c) our hardware design for ARP, (d) our hardware design for ARP with volatile annotations (ARP+VA), and (e) an idealized performance limit model (Ideal). Under the ideal case, we artificially maintain a constant 64 (size of write queue) persists per epoch to estimate an upper bound on performance. Note that, under Ideal, data structures are not recoverable in the event of failure; we include it only as a limit study.

Configuration: Similar to Chapter 4.4, we model persistent memory using the timing model from Xu et al.[27] to represent phase-change memory operating at 533MHz with a 1KB row buffer. We model a persistent memory controller with a 64-entry write queue and schedule persists using an FR-FCFS policy [70], subject to persist ordering constraints. We extend our compiler's `std::atomic` implementation to support our C++11 extensions with volatile annotations in the ARP+VA model. We use the same system configuration as the previous chapter, the details are summarized in Table 4.2.

Benchmarks: We study a suite of three PM-centric multithreaded micro-benchmarks, described in Table 5.3. Our Concurrent Queue (cq) is similar to that of Pelley [5], Array Swap (sps) is similar to that in NV-Heaps [11], and Persistent Cache (pc) is a persistent hash table similar to [14]. In addition, we also consider three write-intensive benchmarks. TATP [46] and TPCC [46] execute "update location" and "new order" transactions, respectively, on top of a transactional storage manger designed for persistent memory, similar to [15]. YCSB `A` [71] (YCSB_A) is a write intensive key-value store workload with 50% reads and 50% updates. It runs on a custom key-value store that has been designed to support all five of our persistency models.

We select these benchmarks specifically because of their PM write-intensiveness, expected to be core tenet of persistent applications [9]. As a measure of the "write-intensive"-ness of the benchmarks, we report the number of persists issued per 1000 cycles (PKC) in Table 5.3. Array swap is our most write-intensive micro-benchmark while concurrent queue is the least, so we expect them to show the most and least sensitivity to different persistency models. Similarly, TATP and TPCC are respectively the most and least write-intensive benchmarks.

All workloads run with eight worker threads that update the underlying persistent data-structure. In all the benchmarks, we run an additional work allocator thread [72] and two volatile antagonistic threads to evaluate the proposed volatile annotations. Each worker

Benchmark	Description	PKC
Conc. queue	Insert/Delete entries in a queue	17.4
Persistent Cache	Persistent hash table	22.7
Array Swaps	Random swaps of array elements	41.8
TATP	Update location trans. in TATP [46]	30.8
TPCC	New Order trans. in TPCC [45]	11.7
YCSB_A	YCSB Workload A [71]	17.4

Table 5.3: ***Benchmark characteristics*** *(PKC = persists per 1000 cycles)*

Benchmark	SCP	RCBSP	ARP	ARP+VA
cq	1	1.7	2.3	3.5
pc	1	2.2	3.9	12.3
sps	1	4.6	8.1	13.2
TATP	1	1.7	4.5	6.0
TPCC	1	1.7	2.7	22.1
YCSB_A	1	2.1	3.8	12.5

Table 5.4: ***Persists per epoch:*** *The persists per epoch observed at the PM controller for various persistency model implementations.*

thread has a 64-entry work queue that resides in the volatile memory. The work allocator thread distributes tasks from a shared work queue to the eight worker threads. Note that since work queue resides in volatile memory, the acquire and release fences required to order accesses to the work queue are volatile fences. This work queue structure represents the request dispatch of a typical network application and illustrates how threads that issue no accesses to persistent memory can nevertheless impact persist performance indirectly due to synchronization operations. Each workload also includes two antagonist threads to simulate the traffic of background threads polling for events. The two threads contend on a lock to a shared counter in volatile memory, increment it, and release the lock. These antagonists represent synchronization activity by unrelated application threads and do not interact directly with the eight worker threads.

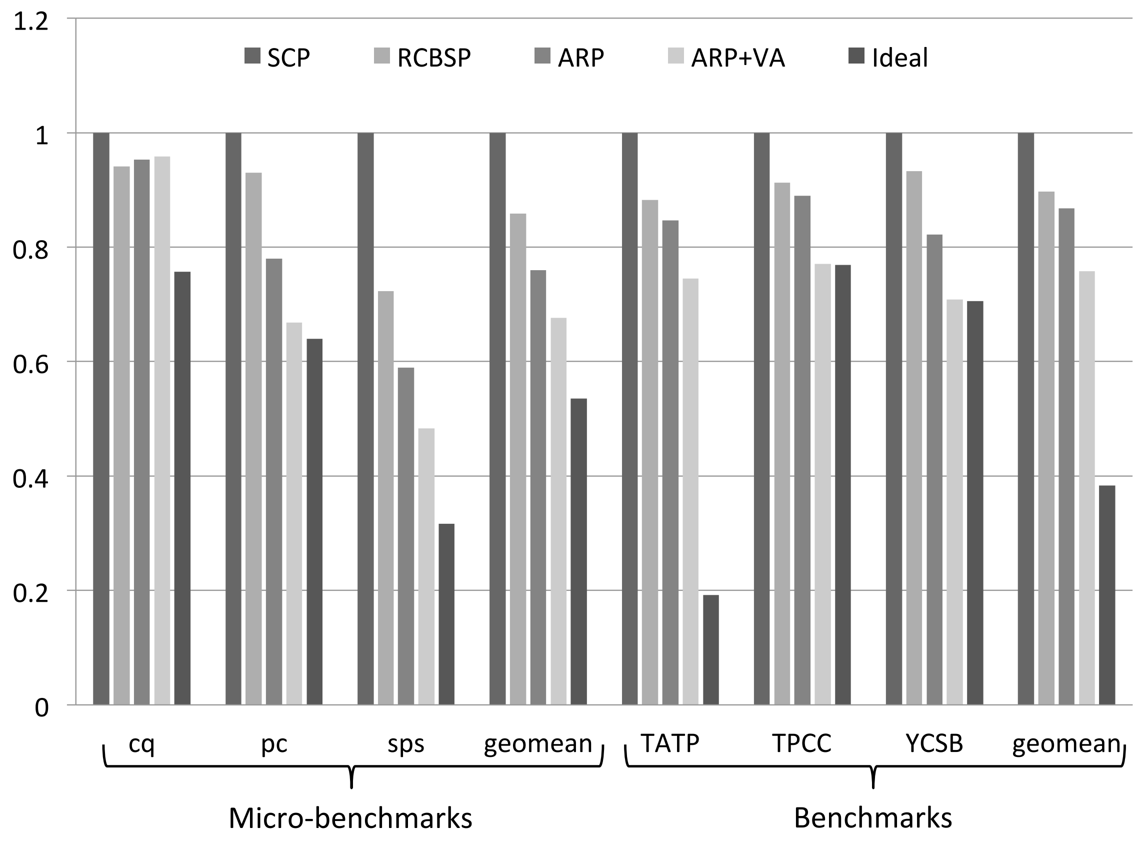


Figure 5.5: ***Execution time normalized to SCP****: The graph compares execution time of ARP and ARP+VA with SCP and RCBSP for micro-benchmarks and benchmarks.*

5.5.1 Performance comparison

We first measure the number of persists per epoch to assess the opportunity of the ARP and ARP+VA models.

Persists per epoch: Table 5.4 shows the persists per epoch under each persistency model. More persists per epoch allow greater reordering opportunity and better persist scheduling at the PM controller. ARP exploits unidirectional acquire and release operations to reduce the number of epochs at the PM controller and increase persists per epoch. ARP provides a 3.9× and 1.8× increase in persists per epoch relative to SCP (which by definition places each persist in its own epoch) and RCBSP, respectively. Further, ARP+VA distinguishes volatile and persistent fences using programmer inserted volatile annotations and achieves a 9.9× and 4.6× increase in persists per epoch relative to SCP and RCBSP.

Micro-benchmarks: The left set of bars in Figure 5.5 contrast the execution time for micro-benchmarks under RCBSP, ARP, ARP+VA, and Ideal ordering models normalized to SCP. Array swap (sps) gains the most from ARP+VA with 51.7% performance improvement over SCP and 33.2% over RCBSP. As evident from the ideal result, array swap is sensitive to the increase in persists per epoch. Concurrent queue (cq) gains the least. In this microbenchmark, entries are pushed or popped from the queue serially by the worker threads; there is limited thread concurrency. As a result, it is not sensitive to the number of persists per epoch and gains little performance even under the ideal case. In fact, due to inopportune read-write bus turnarounds, performance with ARP+VA slightly degrades relative to RCBSP. Overall, ARP+VA improves micro-benchmark execution time by 32.4% as compared to SCP and 21.2% as compared to RCBSP.

Benchmarks: Figure 5.5 also contrasts the execution time of the TATP, TPCC, and YCSB_A benchmarks under each persistency model. YCSB_A is the most sensitive, gaining 17.8% and 29.2% performance, respectively, under ARP and ARP+VA. Further, ARP+VA improves execution time of TATP by 25.5%, and TPCC by 23% as compared to SCP. It is interesting to note that unidirectional fences in ARP do not provide substantial performance gain over RCBSP in TATP even though the ideal case outperforms SCP by 70.2%. TATP includes numerous small critical sections containing full fences to log values before updating the persistent database, limiting potential performance gains. The majority of the gain for TATP is achieved by annotating volatile fences explicitly. Overall, ARP+VA im-

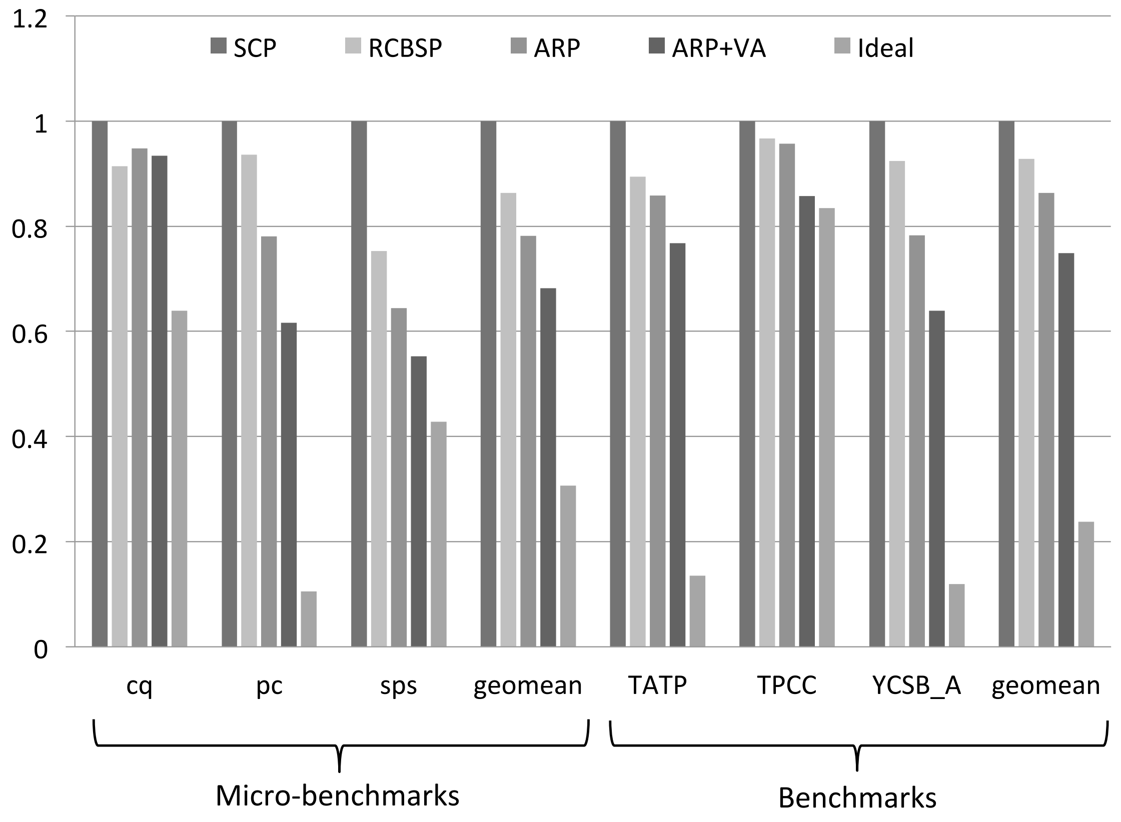


Figure 5.6: Page miss rate normalized to SCP: *Lower page miss rate in the PM controller implies better persist scheduling.*

proves execution time of the three benchmarks by 24.3% and 15.5% over SCP and RCBSP respectively.

5.5.2 Persist scheduling

Finally, we report the impact of PM scheduling policies on the page miss rate of the PM controller in Figure 5.6. The PM controller's FR-FCFS policy seeks to maximize page hits within each persistent memory bank. As described earlier, increasing the number of persists per epoch improves the scheduling flexibility available to the controller. Owing to the unidirectional fences in ARP, the page miss rate drops on average by 17.9% relative to SCP and 8.2% relative to RCBSP. This improvement is the result of the increase in the number of persists that can be scheduled to write to different PM banks concurrently. ARP+VA further relaxes persist ordering constraints by distinguishing volatile and persistent fences, achieving a further 13.0% improvement in page miss rate relative to ARP. The ideal model lowers page miss rate by 76.3% over SCP, indicating the upper bound on PM bank-level parallelism available in these workloads. However, it should be noted that this model does not maintain ordering between the persists and data structures are not recoverable in the event of failure.

CHAPTER VI

Related Works Survey

The proposals in this thesis involve both software (Chapters III, V) and hardware (Chapters IV, V) modifications that would decrease the overall costs of enforcing persist dependencies. Next, we present a survey of related works, we start with related software techniques and then present related hardware works.

6.1 Related works - software

The emergence of new persistent memory technologies has spurred research in many areas of computer science, including file systems [73, 8, 41], databases [40, 74, 75, 76, 21], persistent data structures [11, 7], and concurrent programming [12].

Several systems share our goal of providing a transaction interface to persistent memory. NV-Heaps [11] provides a persistent object system with transactional semantics that prevents persistence-related pointer and memory allocation errors. Mnemosyne [7] allows programmers to declare or allocate persistent data and write this data through special instructions or via transactions. Rio Vista [40] provides transactions on top of flat memory regions.

Prior systems have generally not sought to optimize concurrency of writes to persistent memory. For example, Rio Vista assumes persistent memory is fast enough to not require concurrent accesses [40]. NV-Heaps uses epoch barriers to order persistent writes and assumes that memory accesses execute serially [11]. Mnemosyne uses cache-flush operations to order updates to persistent memory [7].

Unlike these systems, our work focuses on maximizing the concurrency of writes to persistent memory by reducing ordering constraints between persistent memory accesses. We believe that freeing the underlying persistent memory system to reorder, parallelize, and combine writes will be essential to supporting high-performance, transaction-oriented workloads. To our knowledge, our work is the first to explore the implications of various recently proposed persistency models on transaction software.

Recent work by Lu, et al. shares our goal of reducing ordering constraints among persistent writes [19]. Their system distributes the commit status of a transaction among the data blocks to eliminate an ordering constraint within a transaction (similar to the torn bit in Mnemosyne [7]), and uses hardware support (multi-versioned CPU cache and transaction IDs) to enable conflicting transactions to persist out of order. Their techniques are complementary to the ones we propose for reducing ordering constraints. In addition, their system assumes that flushing is required to guarantee ordering (as in eager sync), whereas we explore other memory persistency models.

Our work builds on prior proposals to allow software to communicate ordering dependencies among writes to persistent memory. In shipping systems, order can be enforced by flushing persistent writes from the CPU cache to memory (e.g., via write-through caches or `clflush` instructions) and then issuing a memory barrier (e.g., `mfence`) [77]. However, flushing data to persistent storage is not necessarily the best way to ensure the order in which data is made durable [10]. To relax ordering requirements, Condit et al. propose using *epoch barriers* to ensure an ordering between writes before and after the barrier [8]. Pelley, et al. expand this into a design space for memory persistency models [5].

Others propose hardware support to increase the apparent speed of persistent memory by adding a nonvolatile CPU cache [6] or by assuming sufficient residual power to complete all pending writes [78]. Reducing persist latency makes it less important to allow concurrent writes to persistent memory. Our work makes the more conservative assumption that data must be written to the main persistent memory to be considered durable. Transactions can also be accelerated via other hardware support for persistent memory, such as editable atomic writes [75].

6.2 Related works - hardware

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

No ordering: Apart from durability, cost, scalability, and energy efficiency may make PMs an attractive alternative to DRAM. Some hardware designs focus on PM only as a scalable replacement for DRAM [18, 55] and don't seek to use PM's non-volatility. Deploying PM as a volatile memory alternative requires addressing media-specific issues, such as wear-leveling [53, 54, 55], slow writes [79, 80, 81], and resistance drift [82]. These techniques are essential and orthogonal to our use PM.

Persistent caches: By making the caches themselves persistent, some proposals ensure that stores become durable as they execute, obviating the need for a persistency model. Cache persistence can be achieved by building cache arrays from non-volatile devices [21, 6], by ensuring that a battery backup is available to flush the contents of caches to PM upon power failure [78, 59], or by not caching PM accesses [21]. However, integrating NV devices in high performance logic poses manufacturing challenges, present NV access latencies (e.g., for STT-RAM) are more suitable for the LLC than all cache levels [6], and it is not clear if efficient backup mechanisms are available for systems with large caches. Our approach assumes volatile caches.

Synchronous ordering: SO (see Section 4.2) is our attempt to formalize the persistency model implied by Intel's recent ISA extensions [4]. Without these extensions, it may be impossible to ensure proper PM write order in some x86 systems [41]. Mnemosyne [7] and REWIND [83] use SO to provide transaction systems optimized for PM. Atlas [12], uses it to provide durability semantics for lock-based code. SCMFS [84] uses SO to provide a PM-optimized file system. SO provides few opportunities to overlap program execution and persist operations and Bhandari et al. [32] show that write-through caching sometimes provides better performance. We propose delegated ordering to increase overlap between program execution and persist operations.

Epoch barriers: As proposed in BPFS [8], epoch barriers divide program execution into epochs in which stores may persist concurrently. Stores from different epochs must persist in order. BPFS [8] implements epoch barriers by tagging all cache blocks with the current epochID (incremented after every epoch barrier instruction) on every store, and modifying the cache replacement policy to write epochs back to PM in order in a lazy fashion. This approach allows for more overlap of program execution and persist operations (no need to stall at epoch barriers) than SO. However, BPFS is tightly coupled with cache management, restricting cache replacements and suffers from some other drawbacks of SO, such as discarding write permissions as epochs drain from the cache. Pelley et al. [5] propose a subtle variation of epoch barriers, and show the potential performance improvement due to a better handling of inter-thread persist dependencies. Joshi et al. [14] define efficient persist barriers to implement buffered epoch persistency. However, Joshi does not study persistency models with a detailed PM controller, which is a central theme of our work. Delegated ordering fully decouples cache management from the path persistent writes take to memory and requires no changes to the cache replacement policy.

Other: Kiln [6] and LOC [19] provide a storage transaction interface (providing Atomicity, Consistency and Durability) to PM, wherein the programmer must ensure isolation. Kiln [6] employs non-volatile LLCs and leverages the inherent versioning of data in the caches and main memory to gain performance. LOC [19] reduces intra- and inter-transaction dependencies using a combination of custom hardware logging mechanisms and multi-versioning caches. Pelley [5] explores several persistency models, which range from conservative (strict persistency) to very relaxed (strand persistency) and shows the potential performance advantages of exposing additional persist concurrency to the PM controller. However, Pelley does not propose hardware implementations for the persistency models. FIRM [9] and NVM-Duet [51] optimize memory scheduling algorithms to manage resource allocation at the memory controller to optimize for performance and application fairness while respecting the constraints on the order of persists to PM.

CHAPTER VII

Conclusions

New persistent memory technologies make it possible to store persistent data directly in memory. Achieving the full performance benefits of doing so requires both minimizing the constraints on the order of writes to PM and also minimizing the cost of enforcing individual persist dependencies. Further, simple and precise programming abstractions for persistent memory programming are required to ensure the wide-spread adoption of persistent memories. This thesis addresses the aforementioned challenges on two thrusts and proposes future work on a third . Summaries of the contributions of this thesis follow.

7.1 Summaries of contributions

In Chapter III, we showed how to design transaction systems that specify and communicate these constraints to hardware in a way that reduces the persist dependencies. Our DCT transaction design reduces the persist critical path and improves performance by up to 50% under epoch and strand persistency and up to 150% under synchronous ordering.

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

In Chapter V, we presented a taxonomy of differing failure-atomicity and ordering guarantees that a language-level persistency model might provide. Based on our analysis of this taxonomy, we proposed acquire-release persistency (ARP), a language level persistency model for C++11. We then co-optimized ARP with an underlying ISA-level persistency model, RCBSP, to minimize the number of persist constraints the PM controller must enforce, substantially increasing PM bank-level parallelism and performance. Although we have focused on the C++11 memory model, we believe the insights underlying our work apply more broadly to programming systems, especially when the language mandates a weaker memory model than the underling hardware.

7.2 Future work

This thesis outlines the fundamental research conducted on how to efficiently provide precise recovery guarantees in persistent memory systems using a combination hardware and software techniques. However, much more applied research needs to be conducted in identifying and mitigating the system-specific problems that arise when integrating persistent memories into various computing systems. For example, integrating persistent memories into small energy harvesting devices will likely present different challenges than integrating persistent memories into servers in a data center. The rest of this chapter lists some important directions in which our work can be extended.

Recovery aware wear-leveling for persistent memories: Most candidate persistent memory technologies like PCM and Memristor suffer from “wear-out”. That is, memory cells deteriorate after a certain number of writes to them. So, it is important to make sure that the writes to persistent memory are uniformly distributed to all memory cells, a process called *wear-levelling*. Many wear-levelling techniques have been proposed previously [53, 29], they all involve one or more of the following approaches: a) re-order writes to persistent memory, b) relocate hot persistent memory pages to cold memory locations, and c) in heterogenous memory systems with both persistent memory and DRAM, relocate hot pages in persistent memory to DRAM and cold pages from DRAM to persistent memory.

While such approaches are valid when treating persistent memories as simply DRAM replacement technologies, they may violate recovery guarantees when using persistent memory as the only durable media in the system. While tailoring wear-levelling solutions to ensure recovery correctness, writes to persistent memory may be re-ordered only when such a re-ordering does not violate persistency constraints, relocating pages within persistent memory performed atomically with respect to the program and while relocating pages between DRAM and persistent memory, care should be taken to make sure there is always a consistent version of data in persistent memory.

Programming language support for persistent memories: One of the foci of this thesis is to make programming persistent memory systems easier. To that end, we designed simpler programming abstractions and developed high-level language primitives. However, much more work needs to be done in helping programmers with persistent memory programming. For example, continuing with C++, we need a new type qualifier, say `persistent`, for variables in persistent memory, similar to how `volatile` is used for I/O variables. We need to provide programmers with a library of recoverable data structures they can use in their programs. We also need to develop tools that programmers can use to recover from system failures and debug their persistent memory programs. Programming persistent memory systems is challenging and a robust supporting framework could go a long way in helping the adoption of persistent memory systems.

Persistency models for remote memories: All of the persistency models and their implementations that have been proposed so far assume that persistent memory being accessed by a program is local to the system on which the program is being run. However, with reducing networking latencies and fast remote direct memory accesses (RDMAs), it is reasonable to assume that programs might want to access remote persistent memories. We need to develop the semantics and implementations of our persistency models to handle remote persistent memories, which requires figuring out how the persistency model will interact with network protocols. And, implementations have to specify where, when, and how data gets cached (and hence possible to lose in the event of a failure) when moving data from one system to a remote system, so that programmers are provided with precise guarantees on the status of their data. Persistency models for remote memories will also have to encounter a new kinds kinds of failures emanating from the network and provide programmers with precise guarantees in the event of such failures.

Asynchronous persistency models with durability notifications: Programmers expect mechanisms in their persistency model that they can use to confirm that certain datum has been persisted (for example, before performing an externally visible irrecoverable event). All of the persistency models that have been proposed so far provide synchronous mechanisms to do so, like `pcommit` from synchronous ordering or `persist sync` proposed by Pelley [5]. These synchronous mechanisms block volatile execution of the program until all prior persist operations have been completed. However, if persistent memory access latencies are high, either because of a slow technology or because the memory is in a remote machine, these blocking operations can significantly decrease performance. And, motivate the development of an asynchronous persistency model with durability notifications. Similar to how asynchronous networking protocols operate, these asynchronous persistency models will have mechanisms in place such that programs can poll certain memory locations to confirm if the corresponding data has persisted rather than having to block.

Persistent memories enable a paradigm shift in how we manage recoverable data. For decades, we have used a multi-tiered storage hierarchy with a byte-addressable volatile main memory and a block-addressable persistent storage. With persistent memories we have the ability to unify these tiers into a a single byte-addressable, persistent storage layer. This thesis is one of the initial efforts to redesign software, programming interfaces for this new storage landscape and important work is ahead of us.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] A. Kolli, J. Rosen, S. Diestelhorst, A. Saidi, S. Pelley, S. Liu, P. Chen, and T. Wenisch, "Delegated persist ordering," in *Proceedings of the 49th International Symposium on Microarchitecture*, 2016.

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

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

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

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

[6] J. Zhao, S. Li, D. H. Yoon, Y. Xie, and N. P. Jouppi, "Kiln: Closing the performance gap between systems with and without persistence support," in *Proceedings of 46th International Symposium on Microarchitecure*, 2013.

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

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

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

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

[11] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson, "Nv-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories," in *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2011.

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

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

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

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

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

[17] S. V. Adve and K. Gharachorloo, "Shared memory consistency models: A tutorial," *IEEE Computer*, vol. 29, pp. 66–76, December 1996.

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

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

[20] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz, "Aries: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging," *ACM Transactions on Database Systems*, vol. 17, no. 1, 1992.

[21] T. Wang and R. Johnson, "Scalable logging through emerging non-volatile memory," *Proceedings of the VLDB Endowment*, vol. 7, pp. 865–876, June 2014.

[22] R. Johnson, I. Pandis, R. Stoica, M. Athanassoulis, and A. Ailamaki, "Aether: A scalable approach to logging," in *Proceedings of the VLDB Endowment*, vol. 3, pp. 681–692, September 2010.

[23] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. A. Wood, "Implementation techniques for main memory database systems," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1984.

[24] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communication of the ACM*, vol. 21, pp. 558–565, July 1978.

[25] G. R. Ganger, M. K. McKusick, C. A. N. Soules, and Y. N. Patt, "Soft Updates: A Solution to the Metadata Update Problem in File Systems," *ACM Transactions on Computer Systems*, vol. 18, May 2000.

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

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

[28] G. W. Burr, B. N. Kurdi, J. C. Scott, C. H. Lam, K. Gopalakrishnan, and R. S. Shenoy, "Overview of candidate device technologies for storage-class memory," *IBM J. Res. Dev.*, vol. 52, pp. 449–464, July 2008.

[29] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, "Scalable high performance main memory system using phase-change memory technology," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, 2009.

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

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

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

[33] ARM, "Arm software development tools." `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0802b/CIHGHHIE.html`.

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

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

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

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

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

[39] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, Inc., 1993.

[40] D. E. Lowell and P. M. Chen, "Free transactions with rio vista," in *Proceedings of the 16th Symposium on Operating Systems Principles*, 1997.

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

[42] V. Prabhakaran, L. N. Bairavasundaram, N. Agrawal, H. S. Gunawi, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Iron file systems," in *Proceedings of the 20th Symposium on Operating Systems Principles*, 2005.

[43] A. Thomson and D. J. Abadi, "The case for determinism in database systems," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, 2010.

[44] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, 2005.

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

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

[47] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, pp. 1–7, Aug. 2011.

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

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

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

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

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

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

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

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

[56] H.-J. Boehm and S. V. Adve, "Foundations of the c++ concurrency memory model," in *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2008.

[57] T. J. Dell, "A white paper on the benefits of chipkill-correct ecc for pc server main memory," *IBM Microelectronics Division*, pp. 1–23, 1997.

[58] H. M. Joseph Izraelevitz and M. L. Scott, "Linearization of persistent memory objects under a full-system-crash failure model," in *Proceedings of the International Symposium on Distributed Computing (DISC)*, 2016.

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

[60] H. Boehm and D. R. Chakrabarti, "Persistence programming models for non-volatile memory," in *Proceedings of the ACM SIGPLAN International Symposium on Memory Management*, 2016.

[61] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie, "Unbounded transactional memory," in *11th International Symposium on High-Performance Computer Architecture*, pp. 316–327, IEEE, 2005.

[62] B. Lucia, L. Ceze, K. Strauss, S. Qadeer, and H.-J. Boehm, "Conflict exceptions: simplifying concurrent language semantics with precise hardware exceptions for data-races," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, 2010.

[63] A. Sengupta, S. Biswas, M. Zhang, M. D. Bond, and M. Kulkarni, "Hybrid static–dynamic analysis for statically bounded region serializability," *SIGARCH Comput. Archit. News*, vol. 43, Mar. 2015.

[64] J. Ouyang, P. M. Chen, J. Flinn, and S. Narayanasamy, "...and region serializability for all," in *Presented as part of the 5th USENIX Workshop on Hot Topics in Parallelism*, 2013.

[65] J. Ren, J. Zhao, S. Khan, J. Choi, Y. Wu, and O. Mutlu, "Thynvm: Enabling software-transparent crash consistency in persistent memory systems," in *Proceedings of the 48th International Symposium on Microarchitecture*, pp. 672–685, ACM, 2015.

[66] C. Blundell, M. M. Martin, and T. F. Wenisch, "Invisifence: performance-transparent memory ordering in conventional multiprocessors," in *ACM SIGARCH Computer Architecture News*, vol. 37, pp. 233–244, ACM, 2009.

[67] L. Ceze, J. Tuck, P. Montesinos, and J. Torrellas, "Bulksc: bulk enforcement of sequential consistency," in *ACM SIGARCH Computer Architecture News*, vol. 35, pp. 278–289, ACM, 2007.

[68] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy, "Memory consistency and event ordering in scalable shared-memory multiprocessors," in *Proceedings of the 17th Annual International Symposium on Computer Architecture*, ISCA '90, (New York, NY, USA), pp. 15–26, ACM, 1990.

[69] J. Sevcik and P. Sewell, "C/c++11 mappings to processors," 2011. `https://www.cl.cam.ac.uk/~pes20/cpp/cpp0xmappings.html`.

[70] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens, "Memory access scheduling," in *Proceedings of the 27th Annual International Symposium on Computer Architecture*, ISCA '00, (New York, NY, USA), pp. 128–138, ACM, 2000.

[71] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM symposium on Cloud computing*, pp. 143–154, ACM, 2010.

[72] T. H. Tzen and L. M. Ni, "Dynamic loop scheduling for share-memory multiprocessors.," in *ICPP (2)*, 1991.

[73] P. M. Chen, W. T. Ng, S. Chandra, C. M. Aycock, G. Rajamani, and D. Lowell, "The rio file cache: Surviving operating system crashes," in *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1996.

[74] S. Chen, P. B. Gibbons, and S. Nath, "Rethinking database algorithms for phase change memory," in *Proceedings of the 5th Biennial Conference on Innovative Data Systems Research*, January 2011.

[75] J. Coburn, T. Bunker, M. Shwarz, R. K. Gupta, and S. Swanson, "From aries to mars:transaction support for next-generation solid-state drives," in *Proceedings of the 24thSymposium on Operating System Principles*, 2013.

[76] J. Huang, K. Schwan, and M. K. Qureshi, "Nvram-aware logging in transaction systems," in *Proceedings of the VLDB Endowment*, vol. 8, pp. 389–400, 2014.

[77] S. Venkataraman, N. Tolia, P. Ranganathan, and R. H. Campbell, "Consistent and durable data structures for non-volatile byte-addressable memory," in *Proceedings of the USENIX Conference on File and Storage Technologies*, February 2011.

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

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

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

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

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

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

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