Architectural Support for Persistent Memory Systems

Arpit Joshi

Doctor of Philosophy
Institute of Computing Systems Architecture
School of Informatics
University of Edinburgh
2018

Abstract

The long stated vision of persistent memory is set to be realized with the release of 3D XPoint memory by Intel and Micron. Persistent memory, as the name suggests, amalgamates the persistence (non-volatility) property of storage devices (like disks) with byte-addressability and low latency of memory. These properties of persistent memory coupled with its accessibility through the processor load/store interface enable programmers to design in-memory persistent data structures.

An important challenge in designing persistent memory systems is to provide support for maintaining crash consistency of these in-memory data structures. Crash consistency is necessary to ensure the correct recovery of program state after a crash. Ordering is a primitive that can be used to design crash consistent programs. It provides guarantees on the order of updates to persistent memory. Atomicity can also be used to design crash consistent programs via two primitives. First, as an atomic durability primitive which guarantees that in the presence of system crashes updates are made durable atomically, which means either all or none of the updates are made durable. Second, in the form of ACID transactions that guarantee atomic visibility and atomic durability.

Existing systems do not support ordering, let alone atomic durability or ACID. In fact, these systems implement various performance enhancing optimizations that deliberately reorder updates to memory. Moreover, software in these systems cannot explicitly control the movement of data from volatile cache to persistent memory. Therefore, any ordering requirement has to be enforced synchronously which degrades performance because program execution is stalled waiting for updates to reach persistent memory. This thesis aims to provide the design principles and efficient implementations for three crash consistency primitives: ordering, atomic durability and ACID transactions.

A set of persistency models have been proposed recently which provide support for the ordering primitive. This thesis extends the taxonomy of these models by adding buffering, which allows the hardware to enforce ordering in the background, as a new layer of classification. It then goes on show how the existing implementation of a buffered model degenerates to a performance inefficient non-buffered model because of the presence of conflicts and proposes efficient solutions to eliminate or limit the impact of these conflicts with minimal hardware modifications. This thesis also proposes the first implementation of a buffered model for a server class processor with multi-banked caches and multiple memory controllers.

Write ahead logging (WAL) is a commonly used approach to provide atomic durability. This thesis argues that existing implementations of WAL in software are not only inefficient, because of the fine grained ordering dependencies, but also waste precious execution cycles to implement a fundamentally data movement task. It then proposes ATOM, a hardware log manager based on undo logging that performs the logging operation out of the critical path. This thesis presents the design principles behind ATOM and two techniques that optimize its performance. These techniques enable the memory controller to enforce fine grained ordering required for logging and to even perform logging in some cases. In doing so, ATOM significantly reduces processor stall cycles and improves performance.

The most commonly used abstraction employed to atomically update persistent data is that of durable transactions with ACID (Atomicity, Consistency, Isolation and Durability) semantics that make updates within a transaction both visible and durable atomically. As a final contribution, this thesis tackles the problem of providing efficient support for durable transactions in hardware by integrating hardware support for atomic durability with hardware transactional memory (HTM). It proposes DHTM (durable hardware transactional memory) in which durability is considered as a first class design constraint. DHTM guarantees atomic durability via hardware redo-logging, and integrates this logging support with a commercial HTM to provide atomic visibility. Furthermore, DHTM leverages the same logging infrastructure to extend the supported transaction size, from being L1-limited to the LLC, with minor changes to the coherence protocol.

Lay Summary

Modern computer systems consist of two separate tiers to store data. The fast but volatile memory tier, which is also known as random access memory (RAM), stores data that programs can directly operate on. The slow but durable storage tier, which includes devices like hard disks and solid state drives (SSDs), stores durable data in files that users require accesses to across system restarts. Many important applications, that people interact with daily, need to perform frequent updates to their data in the storage tier. Consider a banking system for example, a large number of customers perform multiple banking transactions daily. All the updates because of those transactions need to applied to some database in the storage tier. Because the storage tier is slow, such applications have poor performance as opposed to applications that operate only on the data in memory.

Intel and Micron have recently announced the release of 3D XPoint memory. 3D XPoint belongs to a new class of memories known as persistent memory, as it combines the durability property of storage with low latency of memory. Persistent memory can significantly improve the performance of applications, like banking, which perform frequent updates to durable data. However, writing applications to operate on durable data in persistent memory is not straightforward. In the absence of appropriate support from the system, an application interrupted by a power failure could corrupt the durable data in persistent memory. For example, consider a system that is executing a banking transaction which consists of debiting money from account A and crediting it to account B. If a power failure happens while this transaction is executing, then it is possible that money is debited from account A but not credited to account B.

To avoid such scenarios, systems with persistent memory need to provide primitives that allow programmers to design programs in a fail safe manner. Such fail safe programs would keep all the data in persistent memory in a consistent state, even in the presence of system crashes. In this instance, a primitive called *atomic durability* is required which guarantees that either all updates are performed or none are performed. Let us consider the same example of the banking transaction discussed earlier. If the system supports atomic durability, it will guarantee that either the money is debited from A and credited to B, or it is neither debited from A nor credited to B. This thesis presents efficient designs of primitives like atomic durability, which allow programmers to write crash consistent programs for systems with persistent memory.

Acknowledgements

I would like to begin by thanking *Para Brahman* for being the guiding light through every step of the way. This thesis would not have been possible without the contributions of many people, including the people I may not know or remember. I wish to thank all of them for, directly or indirectly, helping me in this journey.

Informatics has been a marvellous place for me to learn and grow as a researcher primarily because of the people around here. My advisor, Vijay Nagarajan, has been a wonderful mentor. This thesis would not have been possible without his guidance and generosity with time. Every time I would go to him with an idea, Vijay would ask the most pertinent and fundamental questions, answering which have helped me tremendously in shaping this thesis. Apart from learning the ropes of how to conduct and disseminate research, the most important lesson I have learnt from Vijay is to understand everything from first principles. This has not only helped me in grasping concepts at the most fundamental level but has also opened up a world of non-intuitive and fascinating connections between seemingly different areas.

I would also like to thank Marcelo Cintra and my co-advisor Stratis Viglas for their time and mentorship, for the stimulating discussions and most importantly for their critical feedback. It was an absolute delight working with Marcelo and Stratis. I am fortunate to have had the opportunity to interact with and learn from Boris Grot, Michael O'Boyle, Murray Cole and my thesis examiners Pramod Bhatotia and Margaret Martonosi. I would also like to thank everyone in ICSA for their help and support, especially Marco and Cheng-Chieh for the numerous technical discussions and for their invaluable help with gem5.

I would not be who I am today without Pinki my friend, philosopher and guide. Had it not been for her, I would not have been studying computer architecture, let alone pursuing a PhD. It was because of her unwavering support and constant nudging that I managed to apply on time and secure admission to the PhD program at The University of Edinburgh. Her companionship made my time in Edinburgh and the process leading up to this thesis an absolute delight. Her unshakable belief in me was critical, but even more important was her resolve to not let me lose focus of other important things in life, like health and family. Pinki taught me to aim for the stars and was always there to support me during difficult times. This thesis would not have been possible without those summer solstice dinners in the meadows or those hot meals delivered to the office during deadlines but more importantly, without those liberating and rejuvenating conversations while taking one of our numerous strolls through the beautiful alleys and parks of serene Edinburgh. Pinki deserves equal credit, if not more, for this thesis and I can only hope to make her as happy as she makes me. A new chapter in my life started on the day that Naisha was born and just looking at her face has been the highlight of my day ever since. A source of sheer happiness, Naisha has taught me that pursuing a PhD is much simpler compared to other things in life (like raising a child).

I would like to thank my teachers Monika Shah, Manish Chaturvedi and T.P. Singh who introduced me to the beauty of computer during my undergraduate studies at Nirma University. Special thanks to my friend Vishal for teaching me C programming, which was probably the first step in the journey leading up to this thesis. I would also like to thank my masters advisor Madhu Mutyam and my mentors Shankar Balachandran and V. Kamakoti for their help and guidance during my time at IIT Madras and for being a role model for me to look up to. A special thank you to all my colleagues at Intel, who taught me the practical side of things which helped me tremendously in identifying and formulating problems tackled in this thesis. And how can I forget my mentor Dipankar Nagchoudhuri whose practical advice has been invaluable to me.

Our time in the United Kingdom has been made special with the friendship and the company of Chahna and Dhvanil. Our friendship seems like the most natural thing and I would like to thank them just for being around. I am also grateful to have friends like Gaurang, Viral, Hardik, Kunal and Kalyan with whom you can discuss things and come back with a clarity of thought. Special mention for Dhara, who apart from sharing some delicious recipes has been a delightful friend. I would also like to thank Abhirup, Antonios, Manick, Praveen, Priyank, Raj, Rakesh, Saumay, Siddharth and Vasileios for all the interesting times and/or discussions in Informatics.

Finally, I would like to thank my family. I would like to thank my parents-in-law for having faith in me and for introducing me to satsang. I would like to thank Abhijit for being the mature little brother that he is and for always supporting me in whatever I wanted to do. And last but not the least, I would like to thank my parents for all their sacrifices and teachings. I am here because of them and anything good in me stems from them.

Declaration

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

- Arpit Joshi, Vijay Nagarajan, Marcelo Cintra and Stratis Viglas, ”Efficient Persist Barriers for Multicores”, in 48th International Symposium on Microarchitecture, Waikiki, Hawaii, December 2015.
- Arpit Joshi, Vijay Nagarajan, Stratis Viglas and Marcelo Cintra, ”ATOM: Atomic Durability in Non-volatile Memory through Hardware Support for Logging”, in 23rd International Symposium on High Performance Computer Architecture, Austin, Texas, February 2017.
- Arpit Joshi, Vijay Nagarajan, Marcelo Cintra and Stratis Viglas, ”DHTM: Durable Hardware Transactional Memory”, in 45th International Symposium on Computer Architecture, Los Angeles, California, June 2018.

(*Arpit Joshi*)

To Pinki.

Table of Contents

1 Introduction **1**

1.1 Efficient Persist Barriers for Multicores 4

1.2 Hardware Support for Atomic Durability in Persistent Memory 5

1.3 Durable Hardware Transactional Memory 6

1.4 Integrating Primitives in the System Stack 7

1.4.1 Programming Model 7

1.4.2 Compositionality of Primitives 8

1.5 Summary 8

2 Background **11**

2.1 Persistent Memory Technologies 11

2.2 Memory Consistency Models 12

2.3 Memory Persistency Models 13

2.3.1 Buffering 14

2.4 Atomic Durability 16

2.5 Hardware Transactional Memory 18

2.6 ACID Transactions 19

2.7 System Architecture and Evaluation 20

2.7.1 System Architecture 20

2.7.2 Evaluation 21

3 Efficient Persist Barriers for Multicores **23**

3.1 Introduction 23

3.2 Motivation 24

3.2.1 Buffered Epoch Persistency 24

3.2.2 System Configuration 25

3.3 Persist Barrier Design 25

3.3.1 Resolving Inter-thread Conflicts with IDT 26
3.3.2 Resolving Intra-thread Conflict with PF 29
3.3.3 Epoch Deadlocks and their Avoidance 30
3.4 Persist Barrier Implementation 31
3.4.1 Epoch Flush Protocol 32
3.4.2 IDT and PF Implementation 35
3.4.3 Hardware Extensions 36
3.5 Enforcing Persistency Models 37
3.6 Experimental Methodology 38
3.7 Results 39
3.7.1 Impact of Optimizations 39
3.7.2 Epoch Conflicts 40
3.8 Related Work 41
3.9 Summary 42

4 Atomic Durability in Non-volatile Memory through Hardware Logging 43

4.1 Introduction 43
4.2 Motivation 45
4.2.1 Traditional Undo Logging 45
4.2.2 Undo Logging with NVM 46
4.3 ATOM Design 48
4.3.1 Programming Model 48
4.3.2 Baseline Design 49
4.3.3 Posted Log Optimization 51
4.3.4 Source Log Optimization 52
4.4 ATOM Architecture 53
4.4.1 Overview 53
4.4.2 Log Write Initiate (LogI) Module 54
4.4.3 Log Manage (LogM) Module 54
4.4.4 Recovery 57
4.4.5 Log Allocation and Overflow 58
4.5 Hardware Checkpointing 58
4.6 Experimental Setup 59
4.7 ATOM Evaluation 61
4.7.1 Designs 61

4.7.2 Evaluation 62

4.8 Evaluation of Hardware Checkpointing 67

4.8.1 Designs 68

4.8.2 Evaluation 69

4.9 Related Work 72

4.10 Summary 74

5 DHTM: Durable Hardware Transactional Memory 75

5.1 Introduction 75

5.2 Related Work 76

5.3 DHTM Design 78

5.3.1 Logging for Durability 79

5.3.2 Integrating Logging with HTM 83

5.3.3 Handling Overflow 86

5.4 Putting it Together 89

5.5 Experimental Setup 94

5.6 Results 97

5.6.1 Transaction Throughput 97

5.6.2 Sensitivity to the size of the log-buffer 99

5.6.3 TPC-C and TATP Throughput 99

5.6.4 The Cost of Atomic Durability 100

5.7 Summary 101

6 Conclusions and Future Work 103

6.1 Critical Analysis 104

6.2 Discussion 105

6.2.1 When to persist? 105

6.2.2 How to buffer? 106

6.2.3 Undo vs. Redo 107

6.2.4 Intelligent Memory Systems 108

6.3 Future Work 108

Bibliography 111

List of Figures

1.1 Memory hierarchy with and without persistent memory. 2

2.1 A flag synchronization example where the final state is not valid for both SC and TSO. 13

2.2 Dekker's algorithm where the final state is valid for TSO but not for SC. 13

2.3 Timeline for completion of memory requests for various persistency models. 15

2.4 An example to demonstrate the concept of atomic durability. 17

2.5 System Architecture. 20

3.1 System Configuration: Multiple cores (C), a volatile shared multi-banked last level cache (LLC) and multiple memory controllers (MC) connected by an on-chip interconnection network. 26

3.2 Examples illustrating epoch conflicts. (a) Highlights inter-thread conflict where epoch E_{01} tries to read cache line Y modified in epoch E_{11} (b) Highlights intra-thread conflict where epoch E_{02} tries to modify cache line B modified in epoch E_{00}. 27

3.3 Example showing the benefit of IDT optimization. (a) Shows an example of how completion of conflicting requests is delayed waiting for persist of source epochs to complete. (b) Shows with the same example that by reducing the completion time of conflicting request and allowing the source epoch to persist offline, while enforcing persist ordering constraints, IDT improves performance. 28

3.4 (a) Shows an example of persistent epoch deadlock. Epoch E_i and E_j belonging to threads T_0 and T_1 respectively have a circular dependence. E_j reads cache line A modified by E_i and E_i reads cache line X modified by E_j. (b) Possible epoch deadlock between epochs E_i and E_j is avoided by splitting the ongoing epoch E_i into epochs E_{i1} and E_{i2} on detecting conflict with epoch E_j. 30

3.5 Line diagram explaining the handshaking protocol for Epoch Flush implementation in a multicore with monolithic last level cache. 33

3.6 (a) Shows an example of how epoch ordering constraint is violated. Cache line C belonging to epoch E_2 persists before cache line B belonging to the previous epoch E_1. (b) Shows the correct enforcement of epoch ordering constraints. LLC_{B1} delays persisting cache line C belonging to epoch E_2 until all the cache lines belonging to the previous epoch E_1 have persisted. 34

3.7 Line diagram explaining the handshaking protocol for Epoch Flush implementation in a multicore with multi-banked last level cache. . . 35

3.8 Hardware extensions. 36

3.9 (a) Pseudo-code for a queue insert operation using persist barriers for recovery in case of a system crash. (b) Example illustrating the status of the queue on completion of different epochs within the insert function. 37

3.10 Transaction throughput normalized to LB. 40

3.11 Percentage of conflicting epochs (out of the total number of epochs). . 41

4.1 Sequence of actions to be performed for undo logging in various scenarios. 46

4.2 Undo Log Programming Model. 48

4.3 Sequence of actions of store queue (SQ), cache, memory controller (Mem Ctrl) and non-volatile memory (NVM) for undo logging in NVM based systems. 51

4.4 ATOM Components. 55

4.5 Transaction throughput normalized to BASE for micro-benchmarks. . 63

4.6 SQ full cycles normalized to BASE for micro-benchmarks with small dataset size. 64

4.7 Transaction throughput for REDO and ATOM-OPT designs normalized to ATOM-OPT for benchmarks with small dataset size. 66

4.8 Transaction throughput variation (ATOM-OPT vs REDO) with varying memory latency. 67
4.9 Execution time with varying epoch sizes normalized to NP. 69
4.10 Execution time normalized to NP. 70

5.1 Working set sizes for transactions (a) without including durability log and (b) with durability log. 79
5.2 Redo logging in hardware. 82
5.3 Transaction States. A core can start executing subsequent non-transactional instructions after reaching *Commit/Abort* and can start a new transaction after reaching *Commit Complete/Abort Complete*. . 84
5.4 Flow of a transaction - Part 1. 91
5.5 Flow of a transaction - Part 2. 92
5.6 Transaction throughput normalized to SO. 98
5.7 Normalized transaction throughput sensitivity towards log-buffer size for hash benchmark. 100

List of Tables

3.1 System Parameters. 38

3.2 Micro-benchmarks used in our experiments. 39

4.1 System Parameters. 60

4.2 Micro-benchmarks used in our experiments. 61

4.3 % of source logged cache lines for ATOM-OPT 65

4.4 TPC-C throughput normalized to BASE. 67

5.1 Classification of techniques supporting ACID updates on persistent memory. (* Leverage hardware support for ordering to provide atomic durability.) 77

5.2 Hardware Overhead. 93

5.3 System Parameters. 95

5.4 Benchmarks used in our experiments along with their descriptions and write set sizes (# cache lines). 95

5.5 Abort rates for sdTM and DHTM designs. 98

5.6 Transaction throughput for ATOM and DHTM normalized to SO for TPC-C and TATP benchmarks. 99

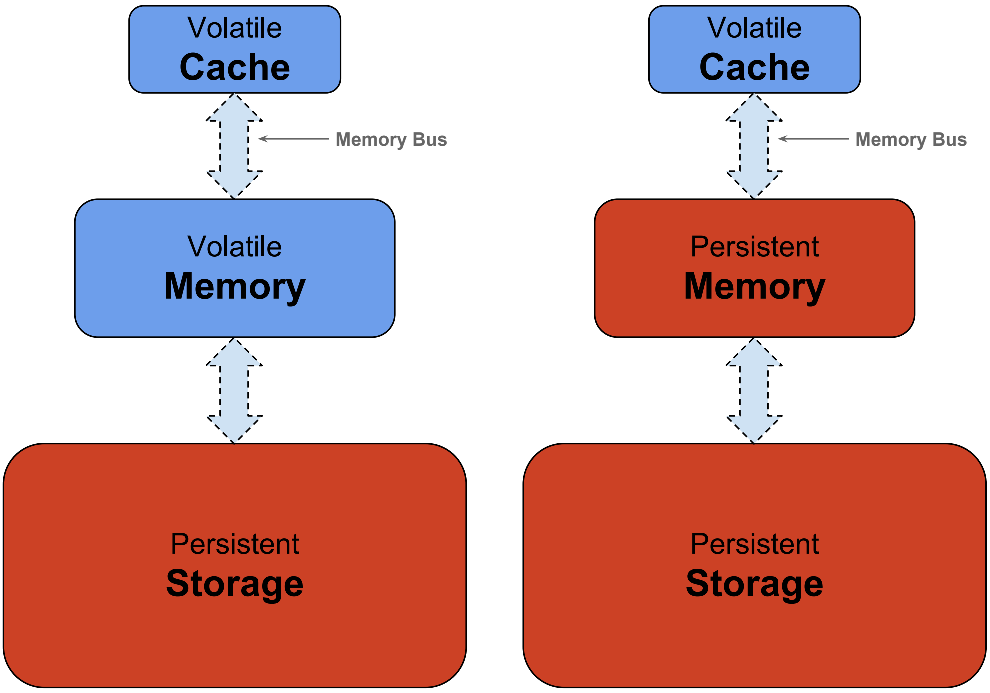
5.7 Transaction throughput for NP and DHTM normalized to SO for hash benchmark with varying memory bandwidth. 101

Chapter 1

Introduction

Traditionally, computer systems have been designed to store data in two different tiers: storage and memory. The storage tier is characterized by the properties of non-volatility and higher density. Non-volatility guarantees that the data in the storage device will be retained even if power supply to the device is disconnected. Higher density means that, compared to memory, a higher number of bits can be stored per unit area. The memory tier on the other hand is characterized by two different properties: low latency and fine granularity. Low latency means that, compared to storage, it takes lesser time to access memory. Finer granularity means that, compared to storage which is typically accessed at a block or a page granularity, the contents of memory can be accessed at a finer granularity like a cache line. This two tier architecture has lead to the design of computer systems where programs typically operate on byte-addressable data in memory while storage is accessed with block based abstractions such as files and operating system managed paging. In summary, traditional computing systems have been designed and optimized for such a two tier architecture because neither memory nor storage possess all the desired properties.

The emergence of multiple viable memory technologies like 3D XPoint [1], PCM [2], STT-MRAM [3], etc., has lead to a promising class of memory known as *non-volatile memory (NVM)* or *persistent memory* which combines the best properties of both storage and memory. It has non-volatility and density properties similar to storage while having access latency and access granularity characteristics similar to memory. Owing to these properties, persistent memory is widely expected to replace or complement the existing memory technology (DRAM) in future computing systems [4, 5]. We envision that a traditional memory hierarchy shown in Figure 1.1(a) with volatile caches, volatile memory and persistent storage would evolve into a sys-



(a) Traditional Memory Hierarchy

(b) Emerging Memory Hierarchy

Figure 1.1: Memory hierarchy with and without persistent memory.

tem shown in Figure 1.1(b) where only caches are volatile while memory and storage are persistent. In the rest of this thesis, we consider this new hierarchy (Figure 1.1(b)) as a representative memory architecture for emerging computer systems with persistent memory.[1]

Persistent memory, by virtue of being attached to the memory bus, allows programmers to access a non-volatile medium through the processor load-store interface. Thereby it opens up a new world of low overhead durability. Durability, also known as persistence, is the property that guarantees that data has safely been written to a non-volatile medium. Access to durable data in memory enables applications like in-memory persistent data structures, low overhead checkpointing, fast databases and key value stores, etc. Irrespective of the application, programmers need guarantees on what updates would have happened to durable data in the event of a system crash. This guarantee is necessary to maintain a property called *crash consistency*, which is required to ensure the correct recovery of program state and durable data after a system crash. For example, consider a program that is operating on an in-memory durable linked list. While adding a new node to the list, the program first writes to the new node and then

[1] However, it is important to note that the principles presented in this thesis will equally apply to an architecture which has both volatile and persistent memory, without loss of generality.

updates a pointer to point to the new node. At the time of failure, the cache might have flushed the pointer update to persistent memory, but the write to the newly created node might still be in the volatile cache leading to an inconsistent state of the durable linked list. This inconsistency can be avoided by ensuring that the write to the newly created node reaches persistent memory before the pointer update. Therefore, to support crash consistency, systems need to provide guarantees on the order in which updates will reach persistent memory. We call this guarantee as the *ordering* primitive.

Ordering is a fundamental primitive that can be used to durably compose and maintain any arbitrary data structure. But it is not the only primitive, as many classes of programs including databases and file systems reason about crash consistency using higher level primitives like *atomic durability*. Atomic durability guarantees that, in the presence of system crashes, either all the updates will be made durable or none of them will be made durable. Consider a scenario where more than one cache line needs to be modified to perform a consistent update to a durable data structure. This is a common scenario applicable to many data structures including the previous example of linked-list. While adding a new node to the linked-list, the write to the new node and the write to update the pointer can potentially map to two different cache lines. If the system crashes after only the pointer update reaches persistent memory, then the linked-list is left in an inconsistent state because of the partial update. However, if the writes to both the new node and the pointer are made durable atomically then this inconsistency can be avoided. Therefore, it is necessary to have primitives like atomic durability which guarantee that a group of updates will be made durable atomically.

ACID transactions is another higher level primitive which is widely used in databases [6]. ACID which stands for Atomicity, Consistency, Isolation and Durability reasons about atomic visibility and atomic durability of updates. Atomic visibility deals with concurrency and provides guarantees on the order in which updates will become visible to other threads. In summary, atomic visibility reasons about consistency with respect to other threads in the system whereas atomic durability reasons about consistency with respect to system crashes. All of these higher level primitives not only ease the burden of reasoning about crash consistency on programmers but also provide them with powerful abstractions to compose structured programs.

Existing memory systems do not provide any sort of ordering guarantees, let alone atomicity or ACID. In fact, these systems implement various optimizations which deliberately reorder updates to memory, at multiple levels in the memory hierarchy, to improve performance. For example, the order is which dirty lines (which have been written to) are replaced from the cache to memory is governed by replacement policies that are solely designed to maximize locality. Other optimizations like write coalescing, that reduce pressure on the memory bandwidth, can also lead to reordering if writes to different cache lines happen between multiple writes to the same cache line. Therefore, while implementing primitives for crash consistency it is desirable to allow reordering to the extent possible.

A key challenge with persistent memory systems is that software cannot explicitly control the boundary between volatile caches and persistent memory. In other words, modified data can move from volatile caches to persistent memory without the permission and even knowledge of software. Therefore, any ordering requirement has to be enforced synchronously. This means that if the software wants to ensure that all the updates till a given point have been made durable, it has to explicitly flush the modified data from caches to persistent memory before performing any further updates. If the software does not do so, a subsequent update might become durable before some of the previous updates and therefore violate the ordering requirement of crash consistency. Enforcing the ordering requirement synchronously stalls program execution waiting for updates to reach persistent memory and can therefore significantly degrade performance. In summary, the problem in persistent memory systems is that the information on when data moves from volatile to persistent domain is not available at the level of instruction set architecture abstraction (software). And therefore, we argue that for performance efficiency, the functionality of moving data from volatile to persistent domain should be implemented at a lower level of abstraction (in the hardware).

In the presence of fast persistent memory, this thesis aims to answer the question: *How to design architectural support for guaranteeing various crash consistency primitives while maximizing performance?* The rest of the chapter provides a brief overview of the main proposals and contributions of this thesis.

1.1 Efficient Persist Barriers for Multicores

To ensure consistency, a set of persistency models [7] have been proposed in the literature that specify the order in which updates can be made durable. This order can be enforced using what is known as a *persist barrier*. A persist barrier ensures that all the updates that happened before the barrier will reach persistent memory before any of the updates that happen after the barrier. Implementing a persist barrier requires periodic flushing of cache lines from volatile caches to persistent memory.

We extend the taxonomy of persistency models by adding buffering, which allows the hardware to enforce ordering in the background, as a new layer of classification. Buffering decouples program execution from persistence by allowing cache line flushes to happen out of the critical path. But, we show that current persist barrier implementations for buffered models [8] can add cache line flushes back to the critical path in the presence of conflicts. A conflict occurs when the same cache line is updated or accessed across an ordering point leading to an ordering dependency. We categorize these ordering dependencies as inter-thread and intra-thread conflicts and propose two solutions to mitigate them. We propose an Inter-thread Dependence Tracking (IDT) mechanism for dynamically tracking inter-thread dependencies in hardware, which allows us to reduce the overhead of preserving ordering in the presence of inter-thread conflicts. We also propose a Proactive Flushing (PF) scheme to write back cache lines proactively as opposed to the reactive approach of existing implementations. PF reduces the probability of encountering inter-thread and intra-thread conflicts in the future.

Our main contribution is an efficient persist barrier that integrates IDT and PF mechanisms and therefore reduces the number of cache line flushes happening in the critical path. We detail the complete implementation of this efficient persist barrier for a server class processor with multi-banked caches and multiple memory controllers. We evaluate our proposed persist barrier by using it to enforce a persistency model known as buffered epoch persistency. Experimental evaluations show that using our persist barrier reduces the probability of encountering conflicts by 15% and improves the performance by 22% on average over the state-of-the-art.

1.2 Hardware Support for Atomic Durability in Persistent Memory

One of the common ways of supporting atomic durability is to employ recovery mechanisms like write-ahead logging [9] as has been detailed in many proposals [10, 11, 12, 13, 14]. Write-ahead logging writes undo or redo log entries for all data updates and requires the ordering constraint that log writes happen before data updates (log → data ordering). Current proposals for implementing logging, rely on software instructions to create log entries and enforce ordering. Since the software has no control over when a cache line is flushed out of the cache, any data update cannot be performed until the corresponding log entry is made durable. This brings logging operations in the critical path and can result in significant performance degradation.

We aim to provide efficient support for atomic durability by moving logging operations out of the critical path. We observe that logging is, fundamentally, a data movement task associated with stores in a program and therefore it can be efficiently supported in hardware. Towards this end we propose ATOM: a hardware log manager to guarantee atomic durability through transparent and efficient logging which manages log allocation, ordering and log truncation in hardware. Our logging design is in many ways similar to the data movement tasks offloaded to a DMA (direct memory access) engine. Offloading logging to a log manager in hardware frees up core execution resources, and relieves the programmer from explicitly implementing the logging logic.

In particular, we implement two performance enhancing optimizations in ATOM. The posted log optimization allows ATOM to efficiently enforce the log → data ordering constraint at the memory controller level, and thereby moves the ordering overhead out of the critical path. We also propose an optimization called source logging in which the memory controller eagerly performs logging for read exclusive requests, thereby eliminating wasteful data movement. We evaluate ATOM on a server class processor and show that it can improve performance by 27% to 33% for micro-benchmarks and by 60% for large-scale transactional workload (TPC-C) over a baseline undo log design. ATOM also compares favorably with a competing approach [15] which provides support for redo logging.

As an application of the primitives, we propose a new mechanism which couples the efficient persist barrier from the previous section with ATOM, for efficiently checkpointing programs in persistent memory systems. Our experiments using a subset of PARSEC, SPLASH and STAMP benchmarks show that doing so enables checkpointing of applications with only a 30% execution time overhead over a non-persistent execution which does not create any checkpoints.

1.3 Durable Hardware Transactional Memory

In an ACID transaction, the updates within a transaction are made both visible as well as durable in an atomic manner. Existing proposals for supporting ACID transactions either support atomic visibility [10, 11, 13, 16] or atomic durability [16, 17, 18] or both in software and therefore suffer from significant performance overheads.

Here we ask the question, can we support both atomic visibility and durability in hardware for persistent memory systems? A promising approach to hardware ACID is to leverage commercially available Hardware Transactional Memory (HTM) to support atomic visibility. However, current commercially available HTM systems support only small transactions [19, 20, 21, 22, 23]; typically limited by the size of the L1 cache. Moreover, existing systems that support ACID by leveraging HTMs also perform log writes to support atomic durability [16, 17, 18]. This worsens the transaction size problem of HTMs by adding log writes to the transaction write set that the HTM has to track. Alternatively, proposals that do not add log writes to the transaction write set introduce significant changes to the cache hierarchy [24].

We propose the design of a Durable Hardware Transactional Memory (DHTM) which provides hardware support for both atomic visibility and atomic durability to overcome the above stated limitations. For atomic visibility DHTM employs an RTM [21] like HTM, whereas for achieving durability DHTM employs a hardware logging mechanism. Logging is supported in hardware via a bandwidth conserving implementation of a redo log, which enables faster commits. DHTM also extends the supported transaction size from being L1 limited to the size of the last level cache by leveraging the same logging infrastructure and without adding additional hardware.

Our evaluation shows that DHTM outperforms the state-of-the-art by an average of 26% on a set of micro-benchmarks and by a minimum of 21% for TATP and TPC-C. We believe DHTM is the first complete and practical hardware based solution for ACID transactions that has the potential to significantly ease the burden of crash consistent programming.

1.4 Integrating Primitives in the System Stack

Programmers need mechanisms to employ the proposed primitives for writing crash consistent programs. In this section we first present the programming model for these primitives followed by a discussion on how they compose with each other.

1.4.1 Programming Model

The proposed primitives can be integrated into the system stack by either extending the instruction set architecture (ISA) or by extending the semantics of existing instructions. The ordering primitive, for example, can be enforced by extending the ISA to include a new instruction, namely *persist barrier*. A persist barrier will guarantee that the updates that happened before the barrier will reach persistent memory before any of the updates that happen after the barrier. The atomic durability primitive can be implemented by extending the ISA with two instructions, namely *Atomic_Begin* and *Atomic_End* (§4.3.1), to demarcate the region of code that needs to be made durable atomically. Similarly, ACID transactions can be demarcated in programs by either extending the semantics of existing instructions like *XBEGIN* and *XEND* provided for HTMs [21] or by extending the ISA with two new instructions like *Begin_Transaction* and *End_Transaction* (§5.3). In addition to the HTM semantics of atomic visibility, these instructions also provide the atomic durability guarantee.

Reasoning about crash consistency is not straightforward and the usage of these primitives will depend on the properties of the application being made crash consistent. Therefore, in most cases, application programmers will either use these primitives directly or through a library [25] to compose crash consistent programs. However, in certain cases, the compiler might be able to insert them automatically [10].

1.4.2 Compositionality of Primitives

The proposed primitives can either be used standalone, or the can be composed using other primitives to provide crash consistency. For example, a fundamental primitive like the ordering primitive can be used to guarantee atomic durability. Many mechanisms to provide crash consistency implement logging in software [11, 13]. These mechanisms leverage the ordering primitive, to enforce the log → data ordering requirement of logging, to guarantee atomic durability. Similarly, as described in Chapter 5, atomic durability primitive can be leveraged to support ACID transactions in conjunction with a concurrency control mechanism (locks or transactional memory). Therefore, the ordering primitive can also be used to support ACID transactions. However, as we highlight in this thesis, composing a primitive using an existing primitive is inefficient and therefore we need to provide architectural support for each of these primitives.

1.5 Summary

Emerging persistent memory, which combines the best properties of memory and storage, has the potential to enable new applications such as in-memory persistent data structures, low overhead checkpointing, fast databases and key-value stores, etc. To realize this potential, persistent memory systems need to provide support for primitives to guarantee crash consistency while minimizing restrictions on existing performance enhancing optimizations like reordering and write coalescing. In this thesis, we present efficient designs of three primitives: ordering, atomic durability and ACID transactions.

The rest of this thesis is structured as follows. We begin with the necessary background material in Chapter 2. In Chapter 3, we show how to design persist barriers to efficiently support the *ordering* primitive for server class processors. Then in Chapter 4, we present the design of an *atomic durability* primitive by way of hardware undo logging that not only moves log writes out of the critical path but also reduces redundant data movement in certain cases. We then show the design of a primitive to support *ACID transactions* in hardware that employs a hybrid version management mechanism and also extends the write set size of hardware transactions from being L1 limited to being LLC limited in Chapter 5. Finally, we conclude and provide perspectives on future directions in Chapter 6.

Chapter 2

Background

This chapter presents the background material necessary to understand the main contributions of this thesis.

2.1 Persistent Memory Technologies

We define persistent memory as a device that is non-volatile, that sits on the memory bus of the processor and that provides access latency and access granularity characteristics similar to DRAM main memory. There are multiple candidate technologies that can be used to realize persistent memory. These technologies can be broadly classified into two categories based on the mechanism they use to store data. Resistive memories, like Phase Change Memory (PCM) [2, 4] and Resistive Random Access Memory (ReRAM) [26, 27], store data by varying the resistance of the material that is used to make the memory cell. Magnetoresistive random access memory (MRAM) [28] and Spin-Transfer Torque MRAM (STT-MRAM) [3] on the other hand store data by changing the magnetic polarity of the material used to make the memory cell.

Although these technologies are promising, they pose challenges like limited endurance and high write power and/or latency [29]. Owing to these challenges, persistent memory is likely to have limited endurance and higher access latency compared to DRAM. Access latency is also expected to be asymmetric with write latency being higher than read latency. However, many of these technologies are in a very advanced state of development and in fact Intel and Micron have already announced products based on a new technology which they call 3D XPoint [1]. Additionally, multiple techniques have already been proposed to mitigate the endurance problem [30, 31, 32]. Therefore, we believe that persistent memory is highly likely to be available in computer systems in the near future.

2.2 Memory Consistency Models

Most modern multicore systems support shared memory in hardware which allows multiple cores to operate on a single shared address space. To be able to write correct parallel programs in such systems, programmers need guarantees on the order in which memory accesses from a given core appear to perform (i.e. take effect) from the perspective of other cores in the system. A *memory consistency model* specifies this order. Four kinds of orderings are possible:

1. *load* ⟶ *load*
2. *load* ⟶ *store*
3. *store* ⟶ *store*
4. *store* ⟶ *load*

Sequential consistency (SC), for example, requires that all the memory accesses happen in program order. So it enforces all the above memory orderings. Total store order (TSO) on the other hand, variants of which are supported by Intel, AMD and SPARC processors, enforces all but the last (*store* ⟶ *load*) ordering constraint. By relaxing this constraint, TSO allows processors to commit stores by writing them to a local store queue and without waiting for them to be performed with respect to other cores. Let us better understand the differences between SC and TSO with examples. Consider the flag synchronization example shown in Figure 2.1. The final state where *rax* has the value 1 and *rbx* has the value 0 is not valid for both SC and TSO because it violates *store* ⟶ *store* ordering. On the other hand, the final state shown for Dekker's algorithm in Figure 2.2 is valid only for TSO and not for SC. This is because TSO relaxes *store* ⟶ *load* ordering, and therefore the values of x and y can be read before the stores to those variables are performed with respect to other cores.

For the purposes of this thesis, we are less concerned with the precise definition of memory consistency models; we are rather interested in one specific aspect of memory consistency models which is that they reason about when a store from one core will be performed (made visible) with respect to other cores. We will use this aspect as a tool to introduce and explain memory persistency models in the next section. For a broader perspective on memory consistency models, the reader is referred to the book by Sorin et al. [33].

```
//Initial State
flag=0; data=0; rax=0; rbx=0;

Core 0          Core 1
data=1;         rax=flag;
flag=1;         rbx=data;

//Final State
rax=1; rbx=0;
```

Figure 2.1: A flag synchronization example where the final state is not valid for both SC and TSO.

```
//Initial State
x=0; y=0; rax=0; rbx=0;

Core 0          Core 1
x=1;            y=1;
rax=y;          rbx=x;

//Final State
rax=0; rbx=0;
```

Figure 2.2: Dekker's algorithm where the final state is valid for TSO but not for SC.

2.3 Memory Persistency Models

Our focus in this thesis is on leveraging persistent memory technologies to enable fast persistence of programs. In order to enable correct recovery, program state in persistent memory needs to be in a consistent state. The definition of what constitutes a consistent state depends on the programming model or more specifically, on the *memory persistency model* [7]. One easy way to understand memory persistency models is to think about them in relation to memory consistency models. Just as consistency models allow us to reason about visibility of stores, persistency models allow us to reason about durability of stores. In other words, a memory persistency model defines the behaviour of an ordering primitive which specifies the order in which stores become durable. Pelley et al. [7] introduce three persistency models: Strict, Epoch and Strand persistency. Here we focus only on Strict and Epoch persistency.

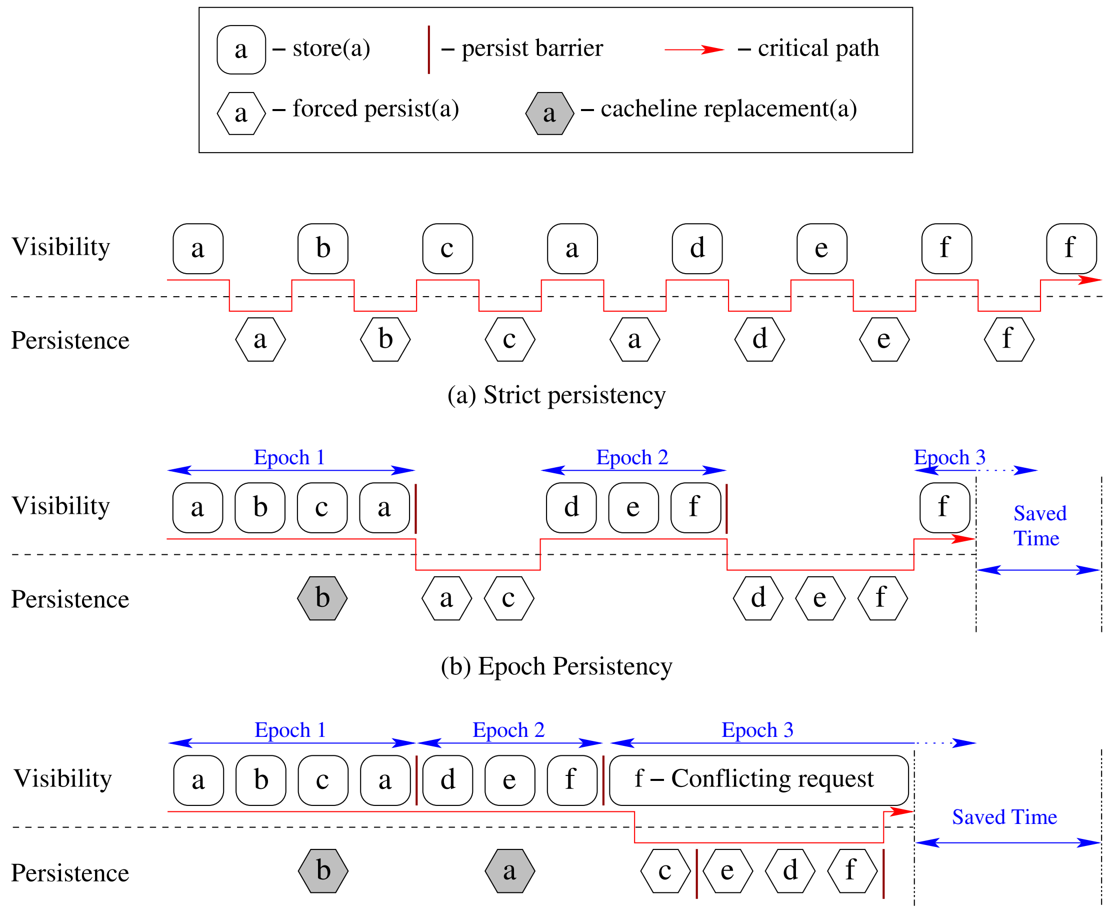
Strict persistency (SP) couples memory persistency with memory consistency. So at the point of failure, whatever updates are visible are guaranteed to have been persisted. For example, TSO systems under strict persistency operate under the following rules: **S1.)** stores persist in program order and **S2.)** a store cannot be made visible until the previous store (in program order) has persisted. A sequence of stores to different cache lines under SP is shown in Figure 2.3(a). As shown in the figure, SP creates persist ordering constraints at the level of each store operation. Hence, caches effectively have a write-through behaviour. Essentially, these fine grained persist ordering constraints conflict with two key optimizations employed in most modern processors. First, multiple stores to a cache line are coalesced in the caches and only written back to memory on a cache line replacement. Under SP since a store operation cannot be issued until the previous store operation persists (rule S2) multiple stores to a cache line cannot be coalesced (as shown in Figure 2.3(a) for cache line *a*). Second, processors reorder cache line persists to improve performance by exploiting temporal and spatial locality. This reordering happens in caches as well as in memory controllers. But under SP, cache lines have to be flushed in program order (rule S1), eliminating any possible performance gain from reordering of writes to memory.

Epoch persistency (EP)[1]relaxes persist ordering constraints compared to SP and enforces ordering at the granularity of *epochs* [8]. An epoch is a contiguous group of instructions which are demarcated using a primitive known as a *persist barrier*. A system with EP operates under the following rules: **E1.)** stores belonging to different epochs persist in the order of their respective epochs and **E2.)** a new epoch cannot begin until all stores belonging to the previous epoch have persisted. Thus, EP allows coalescing of stores and reordering of persists for stores belonging to the same epoch. A sequence of stores to different cache lines under EP is shown in Figure 2.3(b). As shown in the figure, EP allows coalescing for cache line *a* which reduces the overall time taken to complete the sequence of accesses compared to SP. Moreover, since cache lines belonging to the same epoch can persist out of order, cache line *b* can persist before cache line *a*. In EP, persist operations are in the critical path of execution upon completion of an epoch. Even though EP allows write coalescing and reordering of persists within an epoch, it still has a high performance overhead over volatile execution.

2.3.1 Buffering

The fundamental reason for overhead in SP and EP is that persist operations are in the critical path of execution (because of rules S2 and E2 respectively). To overcome this limitation, Pelley et al. [7] propose buffering as an optimization. With buffering,

[1] Pelley et al. [7] do not explicitly differentiate between epoch persistency and buffered epoch persistency that we introduce in §2.3.1.



(a) Strict persistency

(b) Epoch Persistency

(c) Buffered Epoch Persistency

Figure 2.3: Timeline for completion of memory requests for various persistency models.

program execution is allowed to carry past the persist ordering point by only communicating the persist ordering dependency to hardware and not necessarily enforcing it. The hardware is expected to enforce these persist dependencies asynchronously in the background.

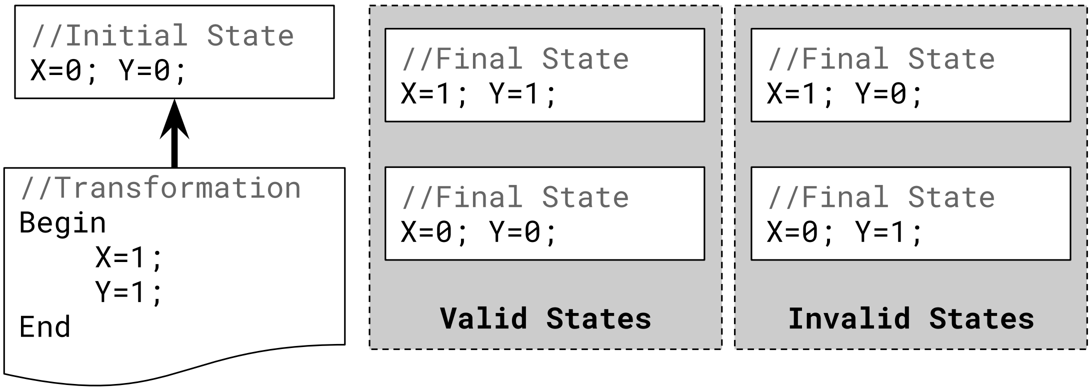
With the buffering optimization there are no guarantees on what updates would have been written to persistent memory at the time of a system crash. It only guarantees that any update that would have reached persistent memory, would have happened in the order specified by the underlying persistency model. In the absence of such guarantees programs need to be remodeled to ensure crash consistency with buffering. Therefore, we take a different approach and view buffering as mechanism that enables a new class of persistency models. The goal of buffered persistency models is to decouple program execution from persistence and thereby improve performance by moving persist operations out of the critical path.

Buffered strict persistency (BSP) [7] is a result of relaxing constraint S2 from strict persistency. Although it removes persistence from the critical path, the problems of being unable to coalesce writes and reorder persists would still remain. These problems in turn would trigger frequent conflicts resulting in a larger percentage of persist operations being in the critical path. We present an optimized implementation of BSP in bulk mode with logging support in Chapter 4.

Buffered epoch persistency (BEP) is a result of relaxing constraint E2 from epoch persistency. Thus, BEP only requires that stores belonging to different epochs persist in the order of their respective epochs. BEP allows program execution to continue across epoch boundaries without waiting for previous epochs to persist. In this case, the cache sub-system has to ensure that epochs are flushed in the correct epoch order. Figure 2.3(c) shows the timeline for a sequence of stores under BEP. Persist barrier after $Epoch1$ does not prevent $Epoch2$ from executing before all the cache lines modified by $Epoch1$ persist. While the program execution continues, modified cache lines can persist naturally because of replacement as shown in the figure for cache lines b and a. In BEP, persist operations are not in the critical path of execution as long as there are no epoch conflicts. An epoch conflict is a scenario where a memory request triggers an epoch flush. In Figure 2.3(c) a store to cache line f conflicts with $Epoch2$ because cache line f has been modified in $Epoch2$ which has not yet persisted. The store request has to wait until all epochs up to $Epoch2$ are flushed. Only epoch conflicts bring the persist operation in the critical path of execution for BEP.

2.4 Atomic Durability

Atomic durability guarantees that for a group of writes, either all writes will be made durable or none of them. Consider the example shown in Figure 2.4. In the initial state, the durable variables X and Y have the value 0. If the shown transformation needs to be applied to the initial state atomically, then the only valid outcomes are either both X and Y have the value 1 or both of them have the value 0. That is, either all of the updates from the transformation are applied or none are applied. The states where partial updates from the transformation are applied are not valid. Storage systems use the atomic durability primitive as a tool to design crash consistent programs. There are multiple ways to implement atomic durability, and the most common among them are write-ahead logging (WAL) [9] and shadow paging [34]. While shadow paging is useful if writes belonging to an atomic update happen at page granularity, WAL works better for atomic updates consisting of scattered writes which happen at a cache line or



```
//Initial State
X=0; Y=0;
```

```
//Transformation
Begin
    X=1;
    Y=1;
End
```

```
//Final State
X=1; Y=1;
```

```
//Final State
X=0; Y=0;
```

```
//Final State
X=1; Y=0;
```

```
//Final State
X=0; Y=1;
```

Figure 2.4: An example to demonstrate the concept of atomic durability.

finer granularity [10, 11, 12, 13, 14, 35].

WAL is based on the principle of physical logging: maintaining a persistent copy of the old and new versions at all times during the atomic update so that state can be recovered to either of the versions. To implement physical logging, a log entry is created which consists of the address and the old or the new value of the data being modified. Additionally, a constraint is imposed that log entries should be made persistent before data entries. To enforce this constraint WAL requires support in the form of an ordering primitive. WAL can be implemented by using either a redo log or an undo log. When the system crashes in the middle of an atomic update, the atomic update can either be reapplied (for a redo log) or undone (for an undo log). There are different trade-offs in implementing a redo or undo based logging mechanism which we discuss next.

Redo Log. In a redo log based design, a log entry contains the address and the new value of the data being modified. Therefore, the in-place data needs to retain the old value to maintain the principle of physical logging. When all the redo log entries have been made durable only then can the in-place data be updated with new value. If any in-place data is durably overwritten with new value before all the redo log entries have been made durable, then principle of physical logging is violated. At this point, the system does not have a persistent copy of the old version of the one data item which was overwritten nor does it have the new version of all data items (redo log entries). If the system were to crash at this point of time, the state can neither be recovered to the old or the new version, thereby violating crash consistency. However, if physical logging is not violated, then the atomic update can be considered as complete as soon as all the redo log entries have been made durable, without waiting for all the in-place data updates to be made durable.

Undo Log. In an undo log based design, a log entry contains the address and the old value of the data being modified. Unlike a redo log based design, the in-place data update can be made durable as soon as the corresponding log entry is made durable without waiting for the log entries of the entire atomic update to be made durable. If the system were to crash before completing the atomic update, then the state can be recovered to the old version as a durable undo log entry would exist for any data that has been updated in-place. However, in an undo log based design the atomic update can be considered complete only when all the in-place data updates have been made durable.

2.5 Hardware Transactional Memory

Hardware Transactional Memory (HTM) is a primitive that allows programmers to write concurrent programs without using explicit synchronization mechanisms like locks. In other words, HTMs provide support for atomic visibility which guarantees that either all the updates of a transaction will be applied and made visible to other transactions and threads or none will be applied. If all the updates of a transaction are applied then the transaction is said to have committed and if none of the updates are applied it is said to have aborted. From idea inception [36] to mainstream commercial adoption, HTMs have come a long way. An important parameter of HTMs is the size of the transaction that they support. Although some prior works have explored unbounded transactions, current commercial HTMs predominantly provide only a best effort service with transaction sizes being limited by the size and associativity of the L1 cache. Below, we briefly describe an HTM system which is similar to state-of-the-art commercial HTM designs [19, 20, 22] and is specifically modelled on Intel's *Restricted Transactional Memory* (RTM) [21] design. For a broader perspective, the reader is referred to Harris et al.'s book [37].

Commercial HTMs. HTMs primarily provide support for three functionalities: buffering the speculative state, tracking read and write sets and detecting conflicts. Commercial HTMs typically buffer speculative state in private caches (typically L1). Each L1 cache line is associated with a *write bit* to keep track of the write set of a transaction. If a cache line belonging to the write set of a transaction is evicted from the L1, the transaction is aborted. Thus, the supported write-set size is limited by the size and the associativity of the L1 cache. Commercial HTMs avoid supporting overflows from the private L1 caches to reduce the design complexity, and in particular that of the LLC.

Similar to the write bit, a *read bit* is also associated with each cache line in the L1 cache. This bit is set when the corresponding cache line is read within a transaction. When such a cache line is evicted, the transaction is typically not aborted, but the address of the cache line is added to a *read-set overflow signature* (also maintained in the L1 cache). Thus, the read set of a transaction is tracked using both the read bits in the L1 cache and the read-set overflow signature.

Conflict detection happens at the L1 cache, with help from the cache coherence substrate. Specifically, when the L1 receives an invalidate request for a cache line in the read set, or an invalidate/data forwarding request for a cache line in the write set, a conflict is detected, triggering an abort of one of the transactions. What transaction must abort is determined by the conflict resolution policy. Two of the commonly used policies are the requester wins policy [21] and the (first) writer wins policy [22].

Overflow Support. Multiple techniques [38, 39, 40] have been proposed to support write set overflows from private caches. Techniques with lazy version management [39, 40] allow the write set to overflow into a redo log. On a commit, these values need to be copied in-place. Consequently, these techniques stall any transaction that conflicts with a committed transaction that is still copying its updates in-place. Techniques with eager version management, such as LogTM [38], allow the write set to overflow in-place in memory but maintain an undo log that is applied in case of an abort. Therefore, they have to stall transactions that conflict with an aborting transaction that is applying its undo log. Stalling adds significant design complexity as it requires support for retrying requests using a NACK based coherence protocol. Our goal with DHTM (Chapter 5) is to support overflows from the L1 cache to the LLC while maintaining the simplicity of an RTM like protocol (§5.3.3).

2.6 ACID Transactions

ACID transactions have been widely deployed in databases as a crash consistency mechanism [6]. ACID stands for atomicity, consistency, isolation and durability. Atomicity provides an all or nothing guarantee, indicating that either all the updates of the transaction are applied or none are applied. Consistency ensures that a successful transaction preserves consistency of the database. Isolation ensures that the intermediate results or partial updates of the transaction are not visible to other concurrently executing transactions. Finally, durability guarantees that all the updates of a committed

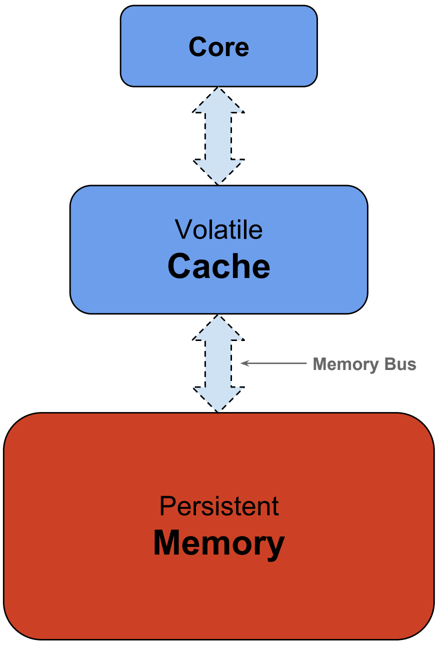


Figure 2.5: System Architecture.

transaction have been made durable such that they will survive any subsequent system crash. Programmers can leverage support for ACID transactions to perform updates to in-memory persistent data structures in a consistent manner. We view ACID transactions as supporting two properties: atomic visibility and atomic durability. Therefore we envision that ACID transactions can be supported by employing HTMs for atomic visibility in conjunction with a suitable write-ahead logging mechanism for atomic durability. We explore this direction in Chapter 5.

2.7 System Architecture and Evaluation

In this section we briefly discuss the system architecture including the failure model and then provide an overview of the evaluation mechanism.

2.7.1 System Architecture

In this thesis we consider the architecture as shown in Figure 2.5, where persistent memory is accessible to the processor (via the load/store interface) over the memory bus. This architecture is an abstraction for multiple alternate architectures which can either have DRAM as memory at the same level as persistent memory or have a DRAM cache as an additional (volatile) caching layer between volatile caches and persistent memory. In a model where DRAM and persistent memory are at the same level in the memory hierarchy, application programs can map non-persistent data that is not required across system crashes to DRAM and persistent data to persistent memory. Alternatively, in a model where DRAM is an additional layer of cache, it will be transparent to the application programmers and the operating system like any other hardware managed cache. Therefore, the proposed architecture is largely representative of architectures for systems with persistent memory.

An alternative model, where persistent memory is not accessible via the processor load/store interface but is rather exposed as a block based device is not covered by the proposed architecture. However, such a model is beyond the scope of this thesis as it would largely leverage existing mechanisms for providing crash consistency that have been proposed for systems with secondary storage based persistence.

2.7.1.1 Failure Model

The crash consistency primitives proposed in this thesis are designed to handle failures like power failures and software crashes. Such failures are modelled using a fail-stop failure model [41], which models a scenario where a failure would halt the system or the application and they will have to be restarted. In other words, a failure would prevent the application from performing any further updates to persistent memory. Recovery in these systems is performed using backward error recovery schemes, wherein the persistent state is recovered to known pre-failure safe state.

2.7.2 Evaluation

For evaluating all the proposals of this thesis we use the gem5 [42] simulator with the Ruby memory model. We model persistent memory in gem5 by modifying the access latency and the bandwidth of existing memory model to represent the characteristics of persistent memory. We evaluate all the proposals on a multicore processor with a two level cache hierarchy consisting of private L1 caches (split between instruction and data) and a shared multi-banked last level cache. The precise system configurations are detailed in the evaluation section in all the chapters.

It is important to note that all designs for crash consistency primitives proposed in this thesis have always been compared to a hardware baseline design in addition to other designs. This comparison is important as it improves the fairness of evaluation and offsets the impact of error margins of the simulator.

2.7.2.1 Benchmarks

We evaluate all the proposals in this thesis on a set of micro-benchmarks and some existing workloads. The micro-benchmarks implement operations on data structures which are typically used in applications that operate on persistent data. The set of micro-benchmarks we use in our evaluations are similar to those in the benchmark suite used by NVHeaps [12] and [7]. For evaluating the combination of ordering and atomic durability primitive to implement checkpointing of applications, we use a subset of PARSEC [43], SPLASH-2 [44] and STAMP [45] benchmark suites (§4.6). Finally, for evaluating the primitives for atomic durability and ACID transactions we also use in-memory implementations of TPC-C and TATP, which are traditional online transaction processing workloads. All of the above benchmarks are typically used to evaluate crash consistency proposals for persistent memory systems [7, 12, 46, 47, 48].

Chapter 3

Efficient Persist Barriers for Multicores

3.1 Introduction

To ensure consistency of persisted data, an *ordering primitive* like a persist barrier is needed to enforce the correct ordering of writes to persistent memory. A persist barrier will ensure that stores appearing before the barrier persist before the stores appearing after the barrier. One way to implement a persist barrier is by using existing instructions like *clflush* and *mfence*[1] [10, 11, 12, 13, 14]. However, this implementation tightly couples visibility and persistence. This coupling forces persistence (e.g., cache line flushes) to happen in the critical path of execution, which can lead to significant performance degradation [7].

Condit et al. [8] propose hardware support for realising an improved persist barrier, that enforces persist ordering lazily. We refer to this barrier as *Lazy Barrier* (LB). Their key idea is to decouple visibility from persistence, allowing program execution to continue beyond the persist barrier, without waiting for stores from previous epochs[2] to persist; their memory system ensures that stores persist in the correct order, out of the critical path.

In LB, the memory system delays the flushing of cache lines; a cache line is only flushed, either due to natural eviction (e.g., replacement), or due to a forcible eviction. Forcible evictions are required in case of *epoch conflicts*, where old epochs need to be flushed in the critical path to ensure consistency of data in persistent memory. For example, before a dirty cache line belonging to a newer epoch can be replaced, cache

[1] Although these instructions ensure the correct order of cache line flushes, they provide no guarantees on the order in which these cache lines persist (are written to persistent memory by the memory controller) [35]. For this, additional instructions like the new *pcommit* [21] need to be used.

[2] Epochs are instruction groups divided by persist barriers.

lines written in older epochs need to be flushed first. Thus, in case of an epoch conflict, the conflicting request has to wait until all the relevant epochs have been flushed to memory. Epoch conflicts bring persist ordering constraints, and consequently cache line flushes, back in the critical path. This again couples visibility with persistence for the duration of conflicts.

In this chapter, we design and implement an efficient persist barrier (LB++) which improves upon LB. We first categorize epoch conflicts into inter-thread and intra-thread conflicts. We then propose optimizations to reduce the overhead due to the conflicts. We propose an Inter-thread Dependence Tracking (IDT) mechanism for dynamically tracking inter-thread dependencies in hardware, which allows us to reduce the overhead of inter-thread conflicts. We then propose a Proactive Flushing (PF) scheme to flush epochs proactively as opposed to the reactive approach of LB. Once an epoch completes, the values of all its cache lines are final. PF exploits this property and starts flushing cache lines on completion of epochs. A related issue in multi-threaded programs is that deadlocks can occur on inter-thread epoch conflicts if the programmer does not correctly place persist barriers. We present a solution to break persistence deadlocks, by splitting epochs, on detecting scenarios which could potentially lead to deadlocks. Finally, we propose a detailed protocol for flushing epochs in the correct order for a system with multi-banked caches.

We demonstrate the efficacy of LB++ by employing it to enforce Buffered Epoch Persistency [7]. Using micro-benchmarks we show that using LB++ (as opposed to LB) improves performance by 22%.

3.2 Motivation

In this section, first highlight the limitations of current implementation [8] of buffered epoch persistency. We then present the system configuration that we consider for our work.

3.2.1 Buffered Epoch Persistency

As described in Chapter 2, buffering allows program execution to be decoupled from persistence. Hence, buffered epoch persistency (BEP) allows program execution to continue past epoch boundaries without waiting for previous epochs to persist. The persist barrier proposed by Condit et al. [8] (LB) is basically an implementation of

BEP. To track the current epoch, each core is extended with an *epoch ID counter* which is incremented by one each time a persist barrier is encountered. Whenever a store completes, it is tagged with the value of current epoch ID and core ID[3]. To track the status of cache lines, cache tags are extended to include epoch ID and core ID fields. Core ID identifies the core that last modified the cache line and epoch ID identifies the epoch in which the cache line was modified. Using these hardware extensions, cache controllers can track and enforce persist ordering dependencies between epochs belonging to the same core.

We call persists happening in the critical path as *online persists* and persists happening out of the critical path as *offline persists*. Online persists have a direct performance impact because they delay program execution while waiting for persist operations to complete. Offline persists on the other hand have no direct performance impact since they are not in the critical path. *The current implementation LB delays persist operations by buffering epochs and relies on offline persists in the form of natural cache line replacements. Although this design improves performance, this is not optimal for two reasons. First, conflicts trigger online persists thus delaying program execution. Second, this design does not actively reduce the number of online persist operations.* We elaborate on these limitations and present solutions in Section 3.3.

3.2.2 System Configuration

We consider a multicore system as shown in Figure 3.1. In this system each core (C) has a private cache and all the cores share a multi-banked last level cache (LLC). All the caches are volatile. It has multiple memory controllers (MC) to provide sufficient memory bandwidth for large number of cores. These memory controllers are connected to persistent memory. This system is similar to most modern server processors, with the only difference being that memory in our system is non-volatile memory.

3.3 Persist Barrier Design

The goal of LB is to decouple persistence from visibility, which allows persist operations to happen out of the critical path. LB achieves this goal as long as there are no epoch conflicts. In this section, we first describe two types of conflicts and propose op-

[3] Condit et al. [8] partition the epoch ID counter and use the high order bits as core ID and remaining bits as epoch ID. We show them as 2 fields for the sake of simplicity.

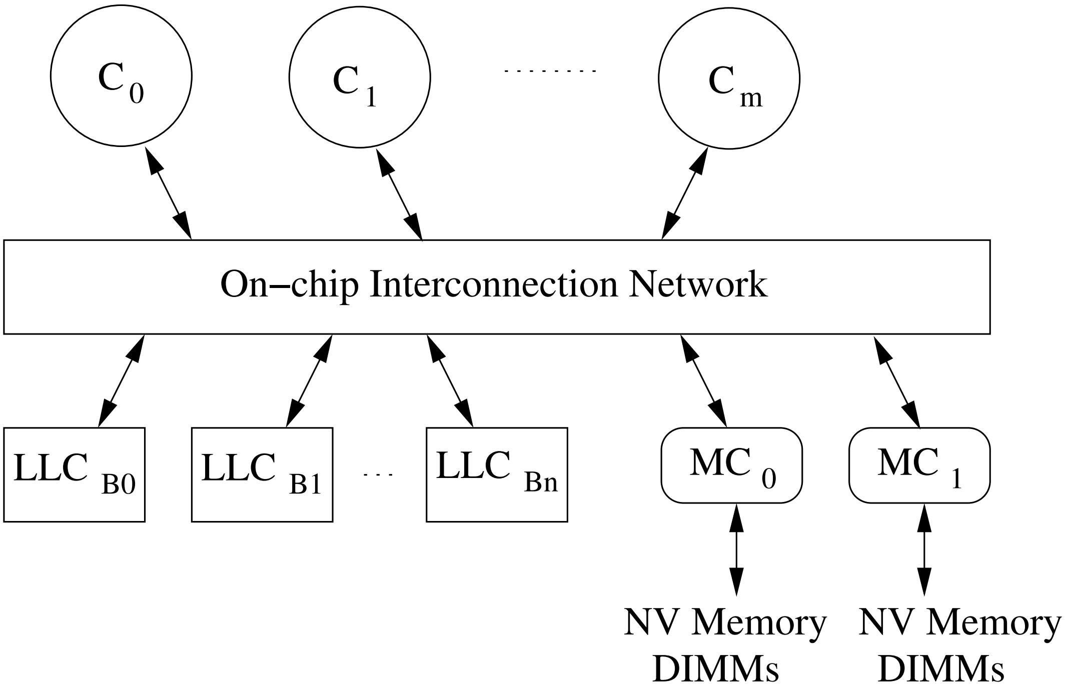


Figure 3.1: System Configuration: Multiple cores (C), a volatile shared multi-banked last level cache (LLC) and multiple memory controllers (MC) connected by an on-chip interconnection network.

timizations to reduce the overheads because of conflicts. We also illustrate the problem of epoch deadlocks and present a solution for the same.

3.3.1 Resolving Inter-thread Conflicts with IDT

An inter-thread conflict is a scenario where a thread tries to read or write to a cache line which has been modified by some other thread in an epoch which has not yet been flushed. Consider the example shown in Figure 3.2(a). Thread T_0 consists of epoch E_{00} and E_{01}. Thread T_1 consists of epoch E_{10} and E_{11}. T_0 tries to read address Y in epoch E_{01} after T_1 has written to it in epoch E_{11}. This creates a new persist ordering constraint that epoch E_{11} of thread T_1 should persist before epoch E_{01} of thread T_0. The epoch tracking hardware in LB can only track persist ordering constraints between epochs from the same core. Since this is an inter-thread ordering constraint, before completing *Ld Y* request from T_0, epoch E_{11} needs to be persisted; if not, epoch E_{01} might persist before epoch E_{11}, leaving persistent data in an inconsistent state. It is important to note here that a read request *Ld Y* creates an epoch conflict since it leads to persist ordering constraints which LB cannot track.

Inter-thread conflicts can lead to significant performance degradation as shown in

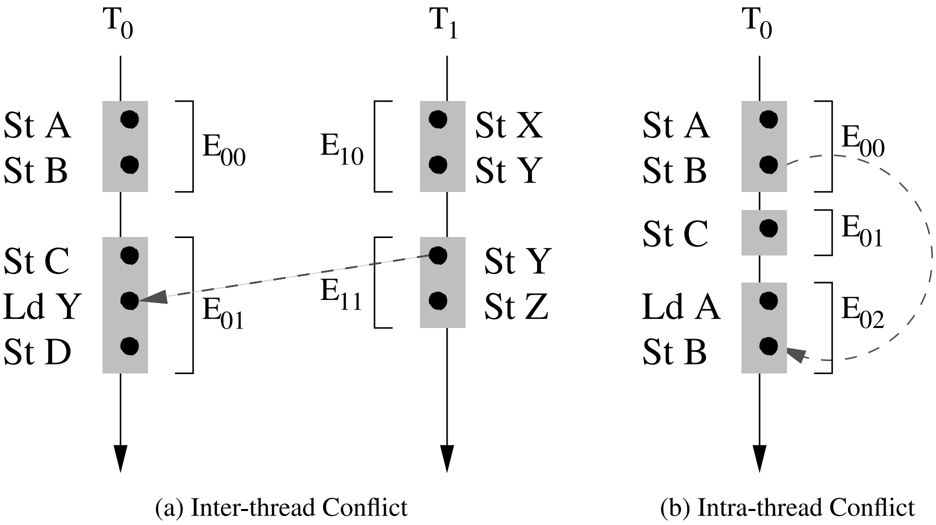


Figure 3.2: Examples illustrating epoch conflicts. (a) Highlights inter-thread conflict where epoch E_{01} tries to read cache line Y modified in epoch E_{11} (b) Highlights intra-thread conflict where epoch E_{02} tries to modify cache line B modified in epoch E_{00}.

Figure 3.3(a), which shows memory requests issued by 2 threads T_0 and T_1. T_1 issues request R_B to read cache line B. Since B has been modified by T_0 in epoch E_{00}, it triggers an inter-thread conflict which triggers online persist of E_{00}. This delays the completion of request R_B.

Inter-thread Dependence Tracking (IDT). If hardware support is provided for tracking inter-thread ordering constraints, the impact of inter-thread conflicts can be reduced. We define the epoch from which a request triggered an inter-thread conflict as *dependent epoch* and the epoch which last modified the requested cache line as the *source epoch.*

To avoid online persists of epochs in case of inter-thread conflicts, we propose a mechanism called inter-thread dependence tracking (IDT). On detecting a conflict, instead of waiting for the conflicting epoch to flush, IDT records source and dependent epochs and enforces this dependence offline. Thus a conflicting request does not have to wait for older epochs to persist. Figure 3.3(b) illustrates the possible performance improvement by using IDT. Request R_B from thread T_1 does not wait for the persist of epoch E_{00} to complete. IDT records the dependence between epochs E_{00} and E_{11} and allows request R_B to complete. When epoch E_{11} completes, cache line E is not allowed to persist until E_{00} has persisted. Thus the overall completion time is reduced while enforcing the correct persist ordering constraints.

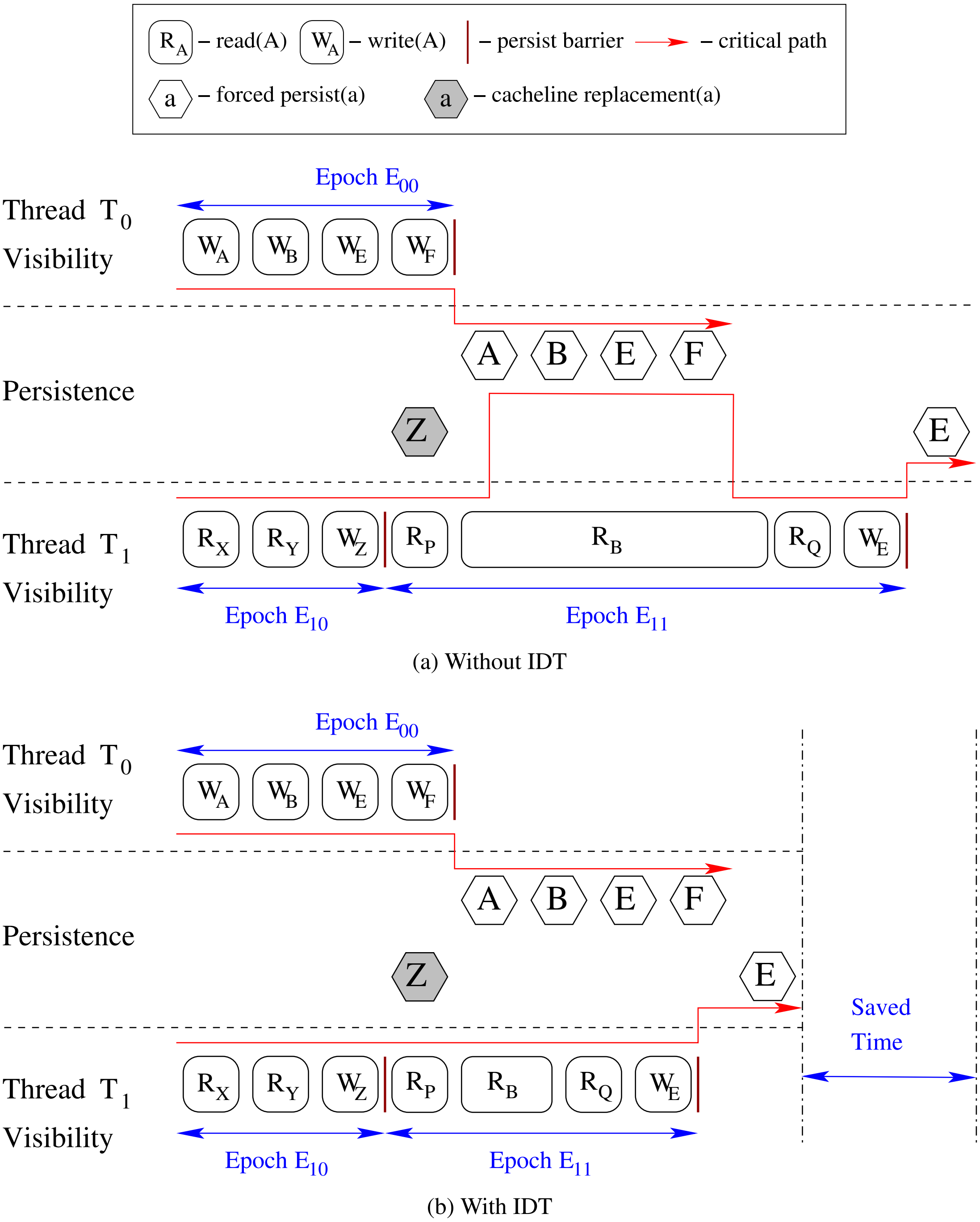


Figure 3.3: Example showing the benefit of IDT optimization. (a) Shows an example of how completion of conflicting requests is delayed waiting for persist of source epochs to complete. (b) Shows with the same example that by reducing the completion time of conflicting request and allowing the source epoch to persist offline, while enforcing persist ordering constraints, IDT improves performance.

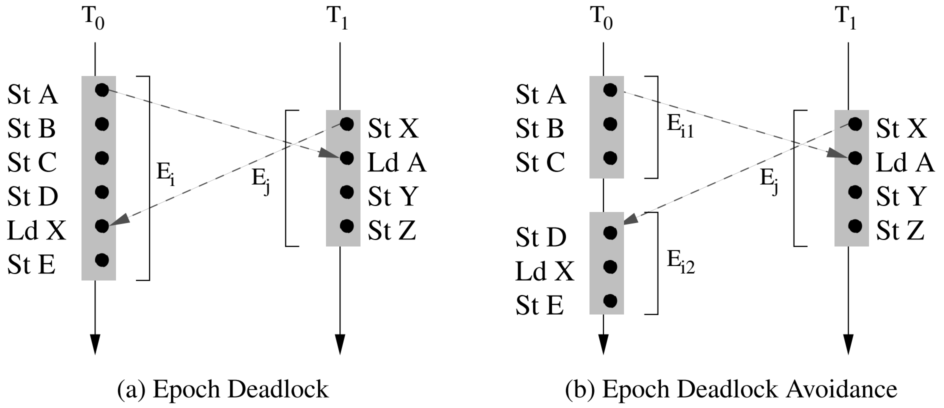
3.3.2 Resolving Intra-thread Conflict with PF

An intra-thread conflict is a scenario where a thread tries to write to a cache line which it has already modified in some prior epoch and the cache line has not yet been flushed. Consider the example shown in Figure 3.2(b). Thread T_0 writes to address B in epoch E_{00}. It then again tries to write to the same address in epoch E_{02}. At this point in time the previous value of B has not yet persisted. If operation *St B* in epoch E_{02} completes, then the value of B will be overwritten. Now if the system crashes after persisting epoch E_{00} but before persisting epoch E_{01} then it will lead to an inconsistent state. To prevent this scenario, before completing *St B* in epoch E_{02}, epoch E_{00} needs to be persisted. It is important to note here that a read request (*Ld A*) does not create a conflict, as the persist ordering constraint between epochs within a thread is already being tracked by LB. On an intra-thread conflict, the epoch that last wrote to the conflicting cache line (B in the example) and all the epochs before it need to be flushed. The only way to minimize the performance impact of this type of conflicts is to minimize the number of such conflicts.

Proactive Flushing (PF). To mitigate the problem of intra-thread conflicts, we propose to persist epochs proactively. Proactive flushing would increase the number of epochs persisting offline. An intra-thread conflict happens because a cache line modified by some older epoch has not yet persisted because of natural cache line eviction. By flushing a cache line proactively we reduce the probability of a conflict arising out of a subsequent access to it. This decrease in the probability of an intra-thread conflict results in improved performance. *It is worth noting that proactive flush will similarly reduce the probability of inter-thread conflicts too.* For ease of explanation, we have introduced it in relation to intra-thread conflicts.

While persisting epochs proactively, care also needs to be taken to ensure that we do not increase the number of flushes to memory. In other words, a cache line should be persisted only when its value is final. This will avoid multiple writes to memory for persisting the same cache line. Therefore, we propose persisting epochs proactively after epochs complete. Naturally, we cannot start proactive persist of an epoch if the previous epoch is not yet fully persisted.

An epoch persist operation consists of durably writing all the modified cache lines, belonging to the epoch being persisted, to persistent memory. An important aspect when persisting epochs proactively should be to ensure that epoch persist operation does not invalidate cache lines. If hardware employs mechanisms similar to *clflush*



(a) Epoch Deadlock

(b) Epoch Deadlock Avoidance

Figure 3.4: (a) Shows an example of persistent epoch deadlock. Epoch E_i and E_j belonging to threads T_0 and T_1 respectively have a circular dependence. E_j reads cache line A modified by E_i and E_i reads cache line X modified by E_j. (b) Possible epoch deadlock between epochs E_i and E_j is avoided by splitting the ongoing epoch E_i into epochs E_{i1} and E_{i2} on detecting conflict with epoch E_j.

instruction, cache lines are also invalidated while being written back to memory. This will have a negative impact on the overall system performance as the persist operation will start evicting working sets from the cache. We implement a non-invalidating flush operation similar to the new *clwb* instruction [21]. This mechanism will not invalidate the cache line being persisted, thus avoiding any negative impact on performance.

3.3.3 Epoch Deadlocks and their Avoidance

In multi-threaded applications, epochs belonging to different threads are independent and can persist in parallel, as long as there are no inter-thread dependencies. In the presence of dependencies, epochs need to persist in the order specified by the dependencies. For the example shown in Figure 3.2(a), epochs E_{00} and E_{10} are independent and can persist in parallel. Whereas, epoch E_{01} is dependent on epoch E_{11}, so E_{01} cannot persist before E_{11} to ensure consistent state of memory. This dependence is enforced by the epoch persistence mechanism by either flushing epoch E_{11} before completing *Ld Y* request of epoch E_{01} or by tracking the inter-thread dependence relation between epochs E_{01} and E_{11} and ensuring that E_{11} persists before E_{01}.

The epoch persistence mechanism can enforce epoch ordering constraints when the dependence relation between epochs is linear. Consider the example shown in Figure 3.4(a). Epochs E_i and E_j have a circular dependence between them. On encountering *Ld A* request by T_1 LB identifies E_i to E_j dependence and will try to flush E_i before completing the request. But a flush for epoch E_i cannot complete because the epoch is ongoing. Then on encountering *Ld X* request by T_0 the epoch persistence mechanism tries to flush E_j before completing the request. This leads to a deadlock.

To prevent deadlocks, a scenario where a circular dependence relation arises needs to be avoided. We propose a solution to epoch deadlocks by conservatively preventing a scenario which can lead to circular dependence. This solution is based on the observation that circular dependence can only occur if a request triggers an inter-thread dependence with an ongoing epoch. By ongoing epoch we mean an epoch whose persist barrier has not yet occurred – in other words, an epoch which has not yet completed. If a request triggers an inter-thread dependence with a completed epoch, then there is no chance of having an inverse dependence since no memory operations are pending in the completed epoch. On detecting an inter-thread dependence with an ongoing epoch, our proposal is to divide the source epoch into two parts: the first part includes all the operations completed at the time of detection and the second part is the remaining portion of the epoch. Without IDT we would have had to flush the first part of the epoch, whereas with IDT it suffices to register the inter-thread dependence in hardware, before completing the request. It is worth noting that, by breaking the source epoch, we have ensured that there is no chance of having an inverse dependence. Figure 3.4(b) shows the solution with an example. When the first dependence is detected with respect to ongoing epoch E_i, E_i is split into epoch E_{i1} which consists of the part of the epoch that has already completed (until *St C*) and the remaining epoch which is called E_{i2}.

Discussion. To prevent epoch deadlock scenarios from happening, persist barriers need to be placed appropriately. One way to prevent epoch deadlocks from happening is to place persist barriers at the start and end of each critical section [49]. However, there can be programs where it is difficult to identify where to place epoch barriers to prevent deadlock (e.g., lock-free programs, programs with user-defined synchronisation, etc.) The deadlock avoidance scheme described above can be useful in such programs.

3.4 Persist Barrier Implementation

In this section, we describe the implementation of our persist barrier. We first present a detailed epoch flush protocol that enforces persist ordering correctly in systems with multi-banked caches. We then describe how IDT and PF are implemented. We summarize by highlighting the additional hardware required for our persist barrier implementation.

3.4.1 Epoch Flush Protocol

To ensure consistency of data in persistent memory, epochs need to be persisted in the correct order. The union of the intra-thread program order and inter-thread shared memory dependencies define this epoch happens-before order. The goal of the epoch flush protocol is to ensure that order in which epochs are persisted is consistent with this happens-before order.

In the system presented in Section 3.2.2 caches are volatile, so persistence happens only when epochs have been written back to persistent memory. Hence, it is sufficient to ensure that for any two epochs E_1 and E_2 such that E_1 happens-before E_2, the last level cache (LLC) will not flush a cache line belonging to epoch E_2 until all the cache lines belonging to epoch E_1 have persisted.

To satisfy the above constraint LLC has to identify two pieces of information: first, the set of all the cache lines belonging to each epoch; second, it has to know when a cache line has persisted. These two pieces of information will help in identifying when an epoch has persisted, based on which LLC can potentially start persisting its successor(s). LLC needs to identify when the L1 cache has written back all the cache lines belonging to an epoch. To convey this information the L1 controller sends an *epoch completion* message (*EpochCMP*) to LLC after writing back all the cache lines belonging to an epoch. This informs the LLC that it has seen all the cache lines belonging to that epoch. It is important to note here that receiving an *EpochCMP* message for a given epoch is a prerequisite for completing the flush of that epoch. If LLC has to flush an epoch but has not received *EpochCMP* message for the same, it can request L1 to flush all the cache lines belonging to that epoch.

Monolithic LLC. If the LLC is monolithic, it can flush epochs independently after receiving *EpochCMP* messages for the epochs being flushed. Figure 3.5 illustrates the protocol. After flushing the epoch preceding epoch E, LLC in step ① starts flushing cache lines belonging to epoch E. Memory controllers respond with a *PersistACK* message after durably writing cache lines to persistent memory in step ②. On receiving *PersistACK* messages for all the cache lines flushed, LLC registers epoch E as having persisted and can start persisting the subsequent epoch. It is important to note that LLC has already received *EpochCMP* message for epoch E and hence it can consider the

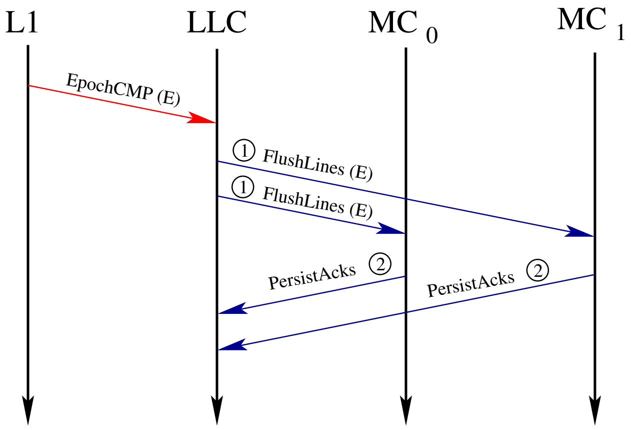
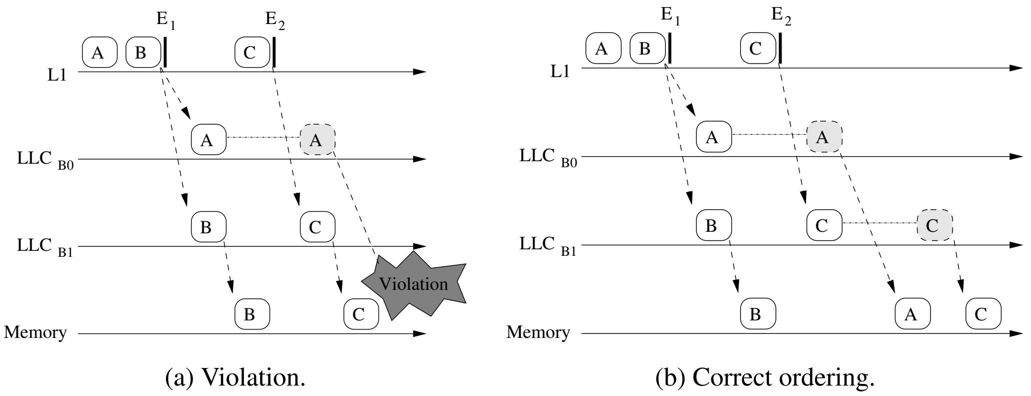


Figure 3.5: Line diagram explaining the handshaking protocol for Epoch Flush implementation in a multicore with monolithic last level cache.

flush of E as having been completed.

Multi-banked LLC. The two step protocol presented above works for monolithic caches, but when extended to a system with multi-banked caches it might not work. Consider the example shown in Figure 3.6(a). The system consists of an L1 cache and two banks of *LLC*. Epoch E_1 consist of two cache lines A and B mapping to LLC_{B0} and LLC_{B1} respectively. Epoch E_2 consists of cache line C mapping to LLC_{B1}. L1 first flushes epoch E_1 and then epoch E_2. LLC_{B1} decides to flush epoch E_1 and hence flushes cache line B. Meanwhile LLC_{B0} delays flushing cache line A. When LLC_{B1} receives cache line C belonging to epoch E_2, it flushes the cache line. LLC_{B1} is allowed to flush epoch E_2, because all its cache lines belonging to previous epoch E_1 have already been flushed. This leads to a violation of epoch ordering constraints since a cache line belonging to epoch E_2 persists before the previous epoch E_1 is flushed completely. If the system crashes at this point, persistent memory will be left in an inconsistent state. The violation shown in Figure 3.6(a) happened because, in a multi-banked cache organisation, each bank only handles a range of addresses and has no information about the status of cache lines outside that range. In the example, LLC_{B1} had no information about the pending cache line belonging to epoch E_1 in LLC_{B0}. To avoid this scenario, a bank of LLC should not start flushing a cache line until all the banks have completed persisting the previous epoch. With this constraint, LLC_{B1} will not flush cache line C until LLC_{B0} has also flushed all the cache lines belonging to epoch E_1. This scenario where epoch ordering constraint is correctly enforced is shown in Figure 3.6(b), where LLC_{B1} does not flush cache line C belonging to epoch E_2 until LLC_{B0} has flushed cache line A belonging to epoch E_1.



(a) Violation.

(b) Correct ordering.

Figure 3.6: (a) Shows an example of how epoch ordering constraint is violated. Cache line C belonging to epoch E_2 persists before cache line B belonging to the previous epoch E_1. (b) Shows the correct enforcement of epoch ordering constraints. LLC_{B1} delays persisting cache line C belonging to epoch E_2 until all the cache lines belonging to the previous epoch E_1 have persisted.

To persist epochs in correct order all the banks of the LLC need to communicate with each other to co-ordinate flushing of every epoch. If all the banks send messages informing epoch completion to each other directly, it will require $O(n^2)$ messages, where n is the number of banks. This can be a prohibitively large overhead, especially considering the fact that this will have to be incurred for each epoch that is being flushed. Instead we propose using an arbiter module to control flushing of epochs. All the banks will inform the arbiter when they have completed persisting an epoch. The arbiter in turn on receiving messages from all the banks will broadcast a message indicating that the relevant epoch has persisted. After receiving this message, LLC banks can start flushing the next epoch. Using an arbiter in this way requires $O(n)$ messages only. In a multicore, the arbiter can become a bottleneck if there is a single arbiter for persisting epochs belonging to all the threads. Instead we propose using a per thread arbiter, which is responsible for coordinating the persist operations belonging to a single thread. This per thread arbiter is placed along with private L1 caches in all the cores and is responsible for coordinating the persist of epochs belonging to the thread executing on that core.

We propose a handshaking protocol by using an arbiter module sitting in the L1 cache to orchestrate epoch flush. The protocol is shown in Figure 3.7. The arbiter in the L1 cache will start epoch flush by first flushing all the cache lines, belonging to the epoch being flushed, from L1. In step ①, L1 will flush all the cache lines belonging

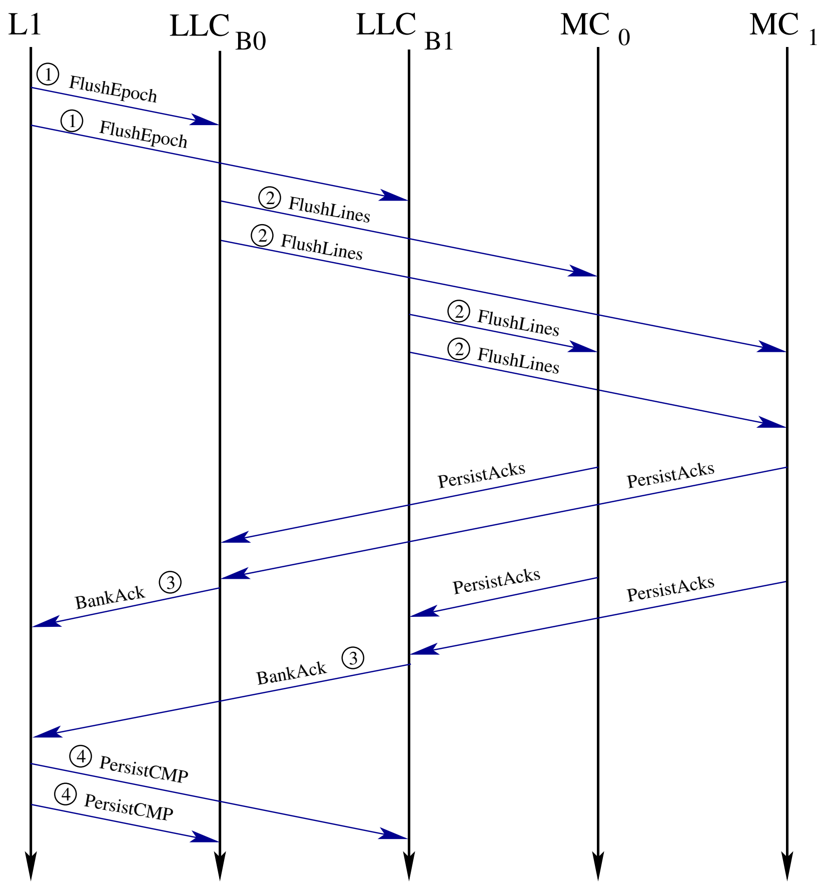


Figure 3.7: Line diagram explaining the handshaking protocol for Epoch Flush implementation in a multicore with multi-banked last level cache.

to that epoch to LLC and also send a *FlushEpoch* message to all LLC banks. In step ② each LLC bank will start flushing all the cache lines belonging to the epoch being flushed. On receiving *PersistAcks* for all the cache lines flushed, each LLC bank will send a *BankAck* message to the arbiter in the L1 controller in step ③. Finally in step ④, after receiving *BankAck* from all LLC banks, the arbiter will signal flush completion (*PersistCMP*) to all the LLC banks. This final step will update the state corresponding to last flushed epoch in all the banks.

3.4.2 IDT and PF Implementation

Enforcing inter-thread persist ordering constraints out of the critical path requires two things. First, preventing the dependent epoch from persisting before the source epoch persists. For this, an entry called dependence register is created in the arbiter corresponding to the dependent epoch. The arbiter before persisting an epoch will check (in addition to its predecessor epochs in program order) the dependence register to see

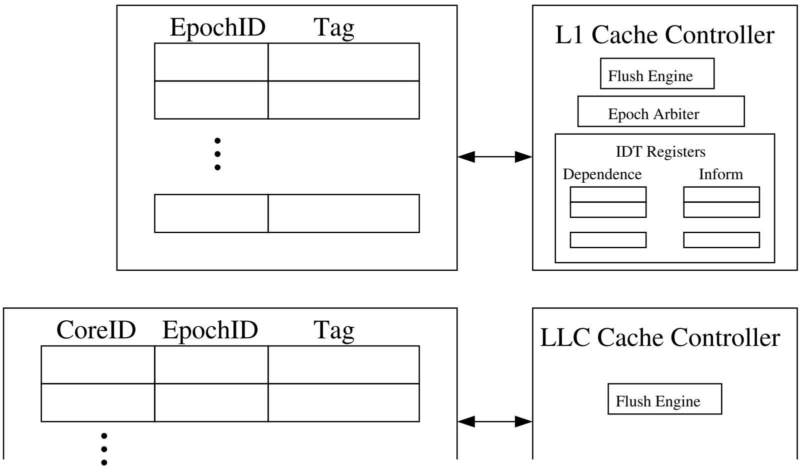


Figure 3.8: Hardware extensions.

if that epoch is dependent on an epoch belonging to some other thread. If so, the arbiter will not flush until the source epoch has been flushed. The second aspect is to inform the dependent epoch when source epoch has been flushed. For this, an entry called inform register is created in the arbiter corresponding to the source epoch. On completing the persist for an epoch, the arbiter will send an epoch persist completion message to the dependent epochs listed in the inform registers.

To implement a proactive flushing scheme, once an epoch completes, a request is sent to the corresponding arbiter to start flushing the epoch. The arbiter starts flushing the completed epoch after ensuring that all its predecessor epochs have persisted.

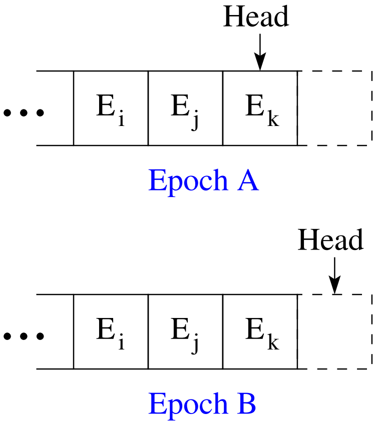
3.4.3 Hardware Extensions

Hardware extensions required to implement a persist barrier are shown in Figure 3.8. To track the epoch status of each cache line, cache tags in both L1 and LLC are extended with EpochID. Cache tags in shared LLC need to be extended with CoreID information to detect inter-thread conflicts. Apart from the cache tags we add a *flush engine* in each cache controller to flush epochs as and when required. Flush engine will trigger a flush for the dirty cache lines of the epoch being flushed. An epoch arbiter is added in the L1 cache controller to co-ordinate epoch flush operation in the presence of multi-banked caches. To track inter-thread dependencies as proposed in IDT, we add a pair of registers called dependence and inform registers in L1 cache controller to identify the source and dependent epochs (using a combination of EpochID and CoreID). These registers are added per in-flight epoch.

In our implementation we support 8 in-flight epochs in a 32-core machine. So

```
QUEUE_INSERT (Head, Entry)
    1. Persist Barrier
A [ 2. Copy (data[Head],Entry)
  [ 3. Persist Barrier
B [ 4. Head = Head + EntryLen
  [ 5. Persist Barrier
END
```

(a) Queue insert pseudo-code



(b) Example

Figure 3.9: (a) Pseudo-code for a queue insert operation using persist barriers for recovery in case of a system crash. (b) Example illustrating the status of the queue on completion of different epochs within the insert function.

EpochID is 3 bits wide and CoreID is 5 bits wide. The overhead of tagging cache lines is 5 bits and 8 bits per cache line in L1 and LLC respectively. We add 4 pairs of IDT registers per epoch to allow tracking of as many inter-thread dependencies per epoch. The overhead of adding these registers is 64 bytes in each L1 cache. The per core arbiter contains a 5-bit counter to track *BankAck* messages received from all the banks. Even though multiple epochs can be in flight, only one of them will be flushed at a time; so one counter is sufficient. Our flush engine maintains bookkeeping information similar to [8], in order to reduce the overhead of searching. In our implementation, we maintain a bitmap per epoch, where each bit corresponds to 64 sets in the cache, amounting to an overhead of 512 bytes for 16-way 1MB LLC bank.

3.5 Enforcing Persistency Models

In this section, we illustrate how our efficient persist barrier can be used to implement Buffered Epoch Persistency (BEP). In BEP [7], programmer inserted persist barriers divide the program into epochs and persist ordering is enforced at epoch granularity. Consider the sample queue insert function (similar to [7]) shown in Figure 3.9(a). Queue insert operations consist of two epochs. In *Epoch A* from lines 2 to 3, a new entry is copied in the queue at the location pointed to by *Head* pointer. In *Epoch B* from lines 4 to 5, *Head* pointer is updated to point to the next empty location. Figure 3.9(b)

Cores	32 OoO cores @ 2GHz
ROB Size	192 Entry
Write Buffer	32 Entry
L1 I/D Cache	32KB 64B lines, 4−way
L1 Access Latency	3 cycles
L2 Cache	1MB×32 tiles, 64B lines, 16−way
L2 Access Latency	30 cycles
Memory Controllers	4
Memory Access Latency	360 (240) cycles write (read)
On-chip network	2D Mesh, 4 rows, 16B flits

Table 3.1: System Parameters.

shows the status of the queue on completion of Epoch A and Epoch B. If the system crashes after Epoch A persists but before Epoch B persists then the new entry E_k is ignored on recovery. If the system crashes after persisting Epoch B then on recovery the program will see successful completion of insert operation.

3.6 Experimental Methodology

We evaluate our proposed persist barrier (LB++) using gem5 [42] with Ruby in full system simulation mode. The on-chip interconnect is modelled using Garnet [50]. We evaluate a 32-core multicore (1 thread per core) with multi-banked LLC and 4 memory controllers placed on 4 corners of the chip. Table 3.1 shows the parameters of the system. Our aim is to evaluate BEP for applications that maintain persistent data structures by using micro-benchmarks.

Workloads: We use micro-benchmarks listed in Table 3.2 to evaluate the proposal of using LB++ to implement BEP. These micro-benchmarks implement data structures that are similar to those in the benchmark suite used by NVHeaps [12], except for the queue micro-benchmark which is similar to the copy-while-locked queue presented in [7]. The size of data entry (table entries, tree nodes, queue entries etc.) for each micro-benchmark is 512 bytes. Each benchmark performs search, delete and insert operations on the corresponding data structure. We inserted persist barriers at appropriate points to ensure persistency (as illustrated in Figure 3.9(a)).

Hash	Insert/delete entries in a hash table
Queue	Insert/delete entries in a queue
RBTree	Insert/delete nodes in a red-black tree
SDG	Insert/delete edges in a scalable graph
SPS	Random swaps between entries in an array

Table 3.2: Micro-benchmarks used in our experiments.

3.7 Results

In this section we evaluate our key contribution, which is our proposed persist barrier (LB++). More specifically, we evaluate the additional speedup provided by LB++, over the state-of-the-art persist barrier LB [12], in enforcing BEP (Section 3.5). We also present performance improvements provided by inter-thread dependence tracking (LB+IDT) and proactive flush (LB+PF) optimizations individually on top of the unoptimized barrier. Recall that LB++ is a result of combining both IDT and PF optimizations.

Both optimized and unoptimized barrier implementations involve cache line flushes. Should the cache line flush be an invalidating (similar to *clflush* instruction) or a non-invalidating flush (similar to recently introduced *clwb* instruction)[4]? We analyzed the performance impact and found that using a non-invalidating flush is significantly faster (around 30% faster). This is not surprising, since an invalidating flush would disrupt locality by evicting lines from cache, which on subsequent accesses need to be fetched again from persistent memory. *For the remainder of the section we only consider using non-invalidating cache line flushes to implement all persist barriers.*

3.7.1 Impact of Optimizations

We study the performance improvement due to the two optimizations, first individually, and then in combination. Figure 3.10 shows the transaction throughput for micro benchmarks, normalized to throughput of LB. On average, LB+IDT improves throughput by only 3%. Recall that IDT improves performance by reducing the latency of memory requests that trigger inter-thread conflict. The primary reason why LB+IDT does not have a high performance improvement is because the performance

[4] The reason for making this comparison is that many processors today only offer flush instructions that invalidate the cache line on completion.

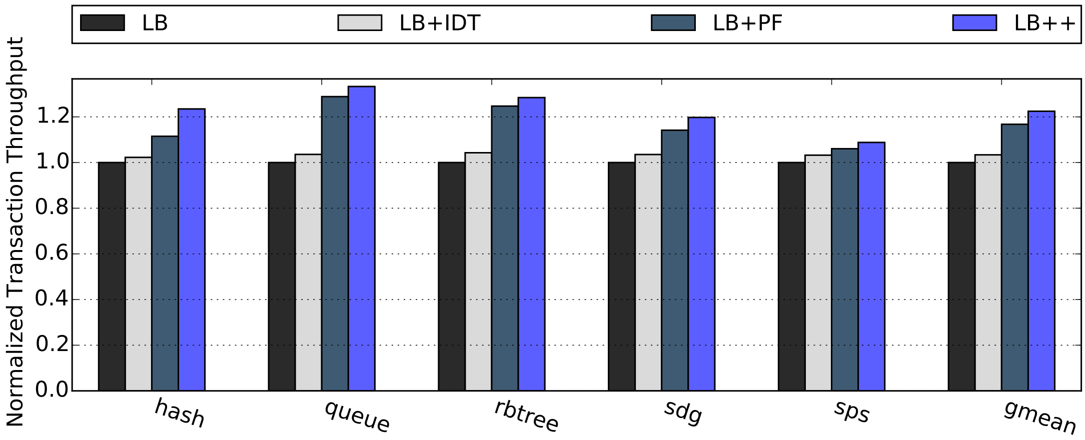


Figure 3.10: Transaction throughput normalized to LB.

is dominated by intra-thread conflicts for these micro-benchmarks (it should be noted that IDT provides significant performance improvement for implementing other persistency models - §4.8). LB+PF on the other hand improves transaction throughput by 17%. This improvement is because LB+PF reduces the number of conflicts, thereby reducing the overall latency of memory requests. LB++, which is obtained by combining IDT and PF optimizations, achieves an improvement in throughput of 22% over LB.

3.7.2 Epoch Conflicts

In the presence of epoch conflicts persist operations happen in the critical path. This is contrary to the objective of LB, which is to perform offline persists. Figure 3.11 shows the percentage of epochs that are flushed because of a conflict. On an average 90% of epochs are flushed because of a conflict in LB. LB+IDT has a similar percentage of epoch conflicts, because IDT optimization does not directly impact the percentage of conflicting epochs but only reduces the latency of conflicting requests. PF optimization, on the other hand, decreases the probability of an epoch conflicting by persisting epochs proactively. Therefore, we can see that, on average LB+PF reduces the percentage of epoch conflicts from 90% to 77%. LB++ (LB+IDT+PF) reduces the epoch conflict percentage further down to 75%, as IDT can help PF. Recall that PF will start flushing an epoch only after the epoch completes. Since IDT allows epochs to complete faster, the scope of flushing epochs proactively increases.

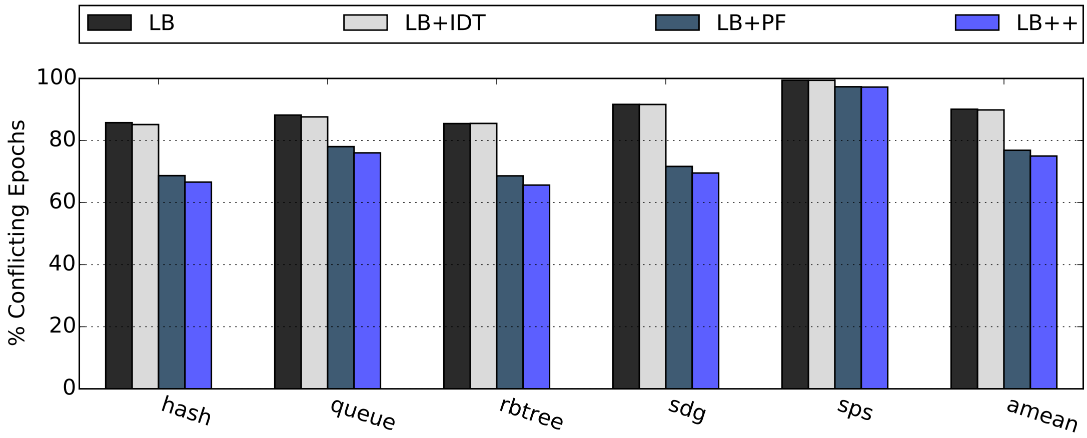


Figure 3.11: Percentage of conflicting epochs (out of the total number of epochs).

3.8 Related Work

There have been many proposals for enabling fast persistence with persistent memory systems [8, 10, 11, 12, 13]. All these techniques provide a programming framework to expose persistent memory to programmers. BPFS and NVHeaps [8, 12] rely on LB for ensuring correct order of persists, whereas others [10, 11, 13] rely on instructions like *clflush* and *mfence* provided by existing processors. It is important to note that these instructions are neither optimal nor sufficient to enforce correct order of persists. Newly proposed *clwb* instruction is optimal because it does not invalidate a cache line while writing it back to memory and another instruction *pcommit* is required to avoid reordering of persists at memory controller level. All of these techniques can seamlessly benefit from our efficient persist barrier implementation.

LOC [51] provides hardware logging support to reduce the overhead of persistence. Kiln [46] proposes a technique to reduce persist latency by using a non-volatile last level cache (NVLLC) along with persistent memory. Using a non-volatile cache also eliminates the requirement of logging by allowing NVLLC and persistent memory to store two versions of a cache line, and one of the versions can be conceptually considered as a log entry. NVM Duet [52], FIRM [53] and DP2 [54] propose optimizations in memory controller to improve the performance of persistent applications. All these proposals broadly help in reducing persist latency which is complimentary to our proposal of efficient persist barrier, in which we reduce conflicts and online persists.

Pelley et al. [55] present designs for implementing transactions for a system with persistent memory. Central to their design is the notion of persisting a batch of transactions together to amortize cost. However, their persists happen in the critical path; in contrast, we seek to move the persists out of the critical path using hardware support. In *memory persistency* [7] various models for persistency including epoch and strict persistency have been proposed. They also identify the possibility of inter-thread dependence tracking. However, they do not discuss how these can be realized. Our work presents a detailed design and implementation of the same.

3.9 Summary

Ensuring consistency of data in persistent memory requires support in the form of an ordering primitive like a persist barrier. We illustrate that even for buffered implementations of persist barrier, persist operations happen in the critical path because of inter-thread and intra-thread conflicts. We proposed an optimization called proactive flush (PF), which eagerly flushes cache lines on completion of an epoch, to reduce the probability of encountering conflicts. We also proposed an inter-thread dependence tracking (IDT) mechanism, which tracks the dependencies between epochs belonging to different threads in hardware, to reduce the overhead of inter-thread conflicts. We presented the design of an efficient persist barrier (LB++) which incorporates PF and IDT optimizations for a server class processor with multi-banked caches and multiple memory controllers. Using LB++ to implement buffered epoch persistency (BEP) for a set of micro-benchmarks reduces the probability of encountering conflicts by 15% and thereby improves the performance by 22%. In the next chapter we will present the design of an efficient hardware checkpointing mechanism that leverages the efficient persist barrier (LB++) proposed here.

Chapter 4

Atomic Durability in Non-volatile Memory through Hardware Logging

4.1 Introduction

In the previous chapter a design for an efficient ordering primitve was presented. In this chapter, we focus on the design for an efficient atomic durability primitive which is important for many classes of applications.

Atomic durability can be supported by employing recovery mechanisms like write-ahead logging [9]. These mechanisms operate on the principle of physical logging: maintaining a persistent copy of the old and new versions at all times during the atomic update so that state can be recovered to either of the versions. Write ahead logging writes log entries for all data updates, and enforces the ordering constraint that log entries become durable before any data update (log $\rightarrow$ data ordering). In systems with non-volatile memory (NVM), log implementations rely on instructions like *non-temporal stores* and *cache-line write backs* to durably write log entries to memory. Moreover, ordering constraints to memory have to be explicitly enforced using instructions like *pcommit* and *sfence* [21, 35, 25].

Support for atomic durability using the above method has a fundamental drawback: durably writing log entries to NVM is in the critical path of execution which can result in significant performance degradation. Our experiments with a set of micro-benchmarks show that durably writing log entries in the critical path degrades throughput by 40% on average and upto 70% (Figure 4.5: BASE vs NON-ATOMIC).

Our goal is to reduce the overhead of logging by moving it out of the critical path. We observe that logging, fundamentally, is a data movement task associated with stores in the original program. Our insight is to perform logging transparently in hardware by: (i) coupling log writes with data stores; and (ii) co-locating data and their corresponding log entries at the same memory controller. In doing so, we not only minimize wasteful data movement, but also enforce log → data ordering constraint in the memory controller (out of the critical path).

We propose ATOM: a hardware log manager to guarantee atomic durability through transparent and efficient logging. ATOM manages log allocation, ordering and log truncation in hardware. At the same time, ATOM is distributed across memory controllers and handles logging for multiple threads on a multicore processor. Our logging design is in many ways similar to the data movement tasks offloaded to a DMA engine. Offloading logging to a log manager in hardware frees up CPU resources, and relieves the programmer from explicitly implementing the logging logic. In ATOM, we expose atomic durable regions to hardware via ISA support (Atomic_Begin and Atomic_End instructions). Stores in this region that require logging (i.e., the first store to a cache line) are detected dynamically and the log write corresponding to the store is performed transparently.

We leverage operating system (OS) support to reserve log space behind each memory controller. ATOM ensures that a log write is sent to the same memory controller as that of the corresponding data. This allows us to efficiently enforce the log → data ordering constraint at the memory controller level, thereby moving the ordering overhead out of the critical path. We also propose an optimization called source logging in which the memory controller eagerly performs logging for read exclusive requests, thereby eliminating wasteful data movement. Finally, we ensure that the log structure is preserved for recovery by forcing every memory controller to flush critical hardware structures (128 bytes per memory controller) to the NVM. Recovery is then ensured through a routine implemented as a system call that undoes all the updates that were incomplete at the time of the crash.

We also propose a new mechanism for checkpointing applications in persistent memory systems. This mechanism implements Buffered Strict Persistency (BSP) (§2.3.1) in bulk mode by combining the efficient persist barrier (LB++) from the previous chapter (§3.3) with ATOM. Here, instead of enforcing ordering constraints at the granularity of each store, it is enforced at the granularity of a group of stores or epochs. However, with the logging support provided by ATOM any epoch which has persisted partially at the time of a system crash will be rolled back, thus satisfying the strict persistency semantics.

In summary the contributions of this chapter are:

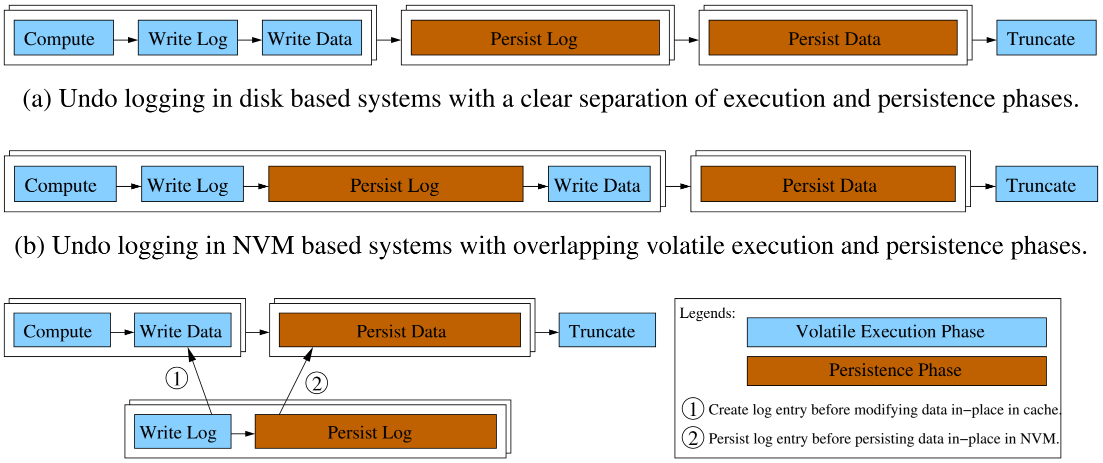
- We propose a log organization that allows us to eliminate log persist operations from the critical path of program execution by enforcing log → data ordering at the memory controller (§4.3.3).
- We propose an optimization to minimize data movement for log entries by dynamically identifying when logging can be done at the source (§4.3.4).
- We propose an efficient log manager in hardware that manages allocating log space, writing log entries and truncating logs transparently with only 3.125 KB overhead per memory controller (§4.4).
- We evaluate ATOM and show that it can improve performance by 27% to 33% for micro-benchmarks and by 60% for large-scale transactional workload (TPC-C) over a baseline undo log design. ATOM also compares favorably with a competing approach which provides support for redo logging (§4.7).
- We propose to implement Buffered Strict Persistency in bulk mode by combining efficient persist barrier (LB++) with ATOM.
- We evaluate BSP implementation and show that it can be used to checkpoint applications with only a 30% runtime overhead compared to an implementation without checkpointing.

4.2 Motivation

As discussed in the background chapter (§2.4), write-ahead logging (WAL) is typically employed to support atomic durability. WAL can be implemented by using either an undo log or a redo log. We consider an undo log based WAL implementation as it enables in-place data writes, so the program can read the latest value without any redirection. In a redo log based implementation, data writes happen in the log area and read requests need to be redirected to the redo log for the latest value. Alternatively, if in-place writes are allowed, cache overflows need to be stored in a victim cache [15].

4.2.1 Traditional Undo Logging

Traditional systems with volatile main memory and persistent secondary storage typically follow the sequence of actions shown in Figure 4.1(a) for implementing atomic durability through an undo log. An atomically durable update using WAL is divided into two phases. The first phase is *volatile execution*: for each data item that is part of



(a) Undo logging in disk based systems with a clear separation of execution and persistence phases.

(b) Undo logging in NVM based systems with overlapping volatile execution and persistence phases.

(c) Undo logging in NVM based systems with decoupling of execution and logging by enforcing two invariants.

Figure 4.1: Sequence of actions to be performed for undo logging in various scenarios.

the atomic update, new values are computed in the compute stage, an undo log entry is written to in the Write Log stage and the data locations are updated in-place in the Write Data stage. The second phase is *persistence*: first the entire log is made durable in the Persist Log stage and then all the data updates are made durable in the Persist Data stage. After updating data, the log is truncated. Note here the clear separation of the volatile and persistence phases, which is justified due to two reasons. First, secondary storage is many orders of magnitude slower than memory and hence making any data durable incurs high latency. Second, secondary storage devices like disks are block based devices, so any update will write an entire page or block to secondary storage. Thus, traditional systems have a separate persistence phase to amortize the cost of performing the atomic update. Moreover, the boundary between volatile memory and persistent storage is software controlled: no data can persist without software's knowledge; this enables the separation.

4.2.2 Undo Logging with NVM

In contrast, in systems with non-volatile memory (NVM), the boundary between volatile caches and non-volatile memory is hardware controlled. Cache line replacements can move data from volatile caches to NVM without software's knowledge. Therefore, such systems cannot completely separate volatile execution from persistence. Moreover, NVM has very different properties than secondary storage. It is expected to have close to DRAM speed, while allowing for updates at a much finer (cache line) granularity. Because of the relatively lower cost (low latency and fine granularity) of persistence there is little need for amortization. Therefore it is not necessary to separate volatile execution from persistence. In fact, it is important to begin persisting data as soon as it is modified to avoid being limited by the write bandwidth, which can happen if all data is simultaneously flushed to persist at the end of the update.

Since we cannot and need not decouple volatile execution from persistence, let us examine the challenges (or constraints) for an undo log implementation in systems with NVMs. Undo log WAL implementation requires that the system maintain a persistent copy of the old version of all data items that are part of the atomic update at all times during the update. Hence, the in-place version of data cannot be modified until the undo log entry of the corresponding data has been made durable. Therefore, it is necessary to persist undo log entries before modifying data structures in-place. Figure 4.1(b) shows the sequence of actions performed for an undo log WAL implementation in NVM. The update process is split into two phases. In the first phase, there is an interaction between volatile execution and persistence operations; the compute stage takes place and an undo log entry is written to; then, the undo log entry is persisted and data is modified in-place. In the second phase the data updates are persisted and finally the log is truncated.

The bottleneck in this approach is that undo log entries have to be made durable in the critical path of execution. Our goal is to decouple log management from volatile execution and move the operation of persisting log entries out of the critical path of execution. As shown in Figure 4.1(c), writing an undo log entry and persisting log entries can be safely moved out of the critical path only if the following two invariants are satisfied.

Invariant 1. *A store should not complete until an undo log entry is created for the data being modified by the store.*

Invariant 2. *In-place data should not be made durable until the corresponding log entry is made durable.*

Invariant 1 ensures that an undo log entry exists for every data that is being modified as part of an atomically durable update. *Invariant 2* ensures that if the atomically durable update fails, undo log entries for all the data items updated in-place are durable. These log entries can be used to undo the partial changes of the failed update.

```
While (!Done) {
    Write Undo Log
    Flush Log
    Write Data
}
Flush Modified Data
```

(a) Traditional Model

```
Atomic_Begin
While (!Done) {
    Write Data
}
Flush Modified Data
Atomic_End
```

(b) Proposed Model

Figure 4.2: Undo Log Programming Model.

4.3 ATOM Design

In this section we introduce the conceptual design for ATOM, a hardware log manager for undo logging. We begin the section by first introducing the programming model with and without ATOM and then go on to establish a baseline design for an undo log manager in hardware. We then propose two optimizations: (i) to eliminate log persist operations from the the critical path, and (ii) to minimize data movement.

4.3.1 Programming Model

A typical approach towards atomic durability in software using an undo log is shown in Figure 4.2(a). An undo log entry is created and flushed before writing to data in-place. After completing the update, the modified data is flushed to NVM to complete the atomic update. In ATOM, we introduce two primitives, *Atomic_Begin* and *Atomic_End* to demarcate the start and end of the code segment performing an atomic update. Using these two primitives, the programmer does not have to create and flush undo log entries, but only write data in-place and flush data on completion of the update (Figure 4.2(b)). In ATOM, the hardware log manager, will create undo log entries and flush them to memory before the in-place data modifications are written to memory.

The *Atomic_Begin* and *Atomic_End* construct only guarantees atomic durability and not isolation in a multi-threaded context. We require software to provide isolation. Specifically, following Chakrabarti et al. [10], we require the durable regions to coincide with outermost critical sections.

4.3.2 Baseline Design

The purpose of ATOM is to provide atomic durability for updates in NVM. Recall that to provide atomic durability, an undo log manager has to perform two tasks. First, creating a log entry consisting of: the old value of the data being modified, and its address. Second, ensuring that the log entry persists before the corresponding data is persisted. For the purposes of our discussion, we consider a generic chip multi-processor with private L1 caches, a multi-banked shared L2 cache and multiple memory controllers. ATOM is implemented as a distributed log manager, that is distributed across L1 caches and memory controllers – with the former responsible for creating log entries and the latter responsible for enforcing log $\rightarrow$ data ordering constraint. Finally, the OS reserves log space behind each memory controller for ATOM to write log entries into.

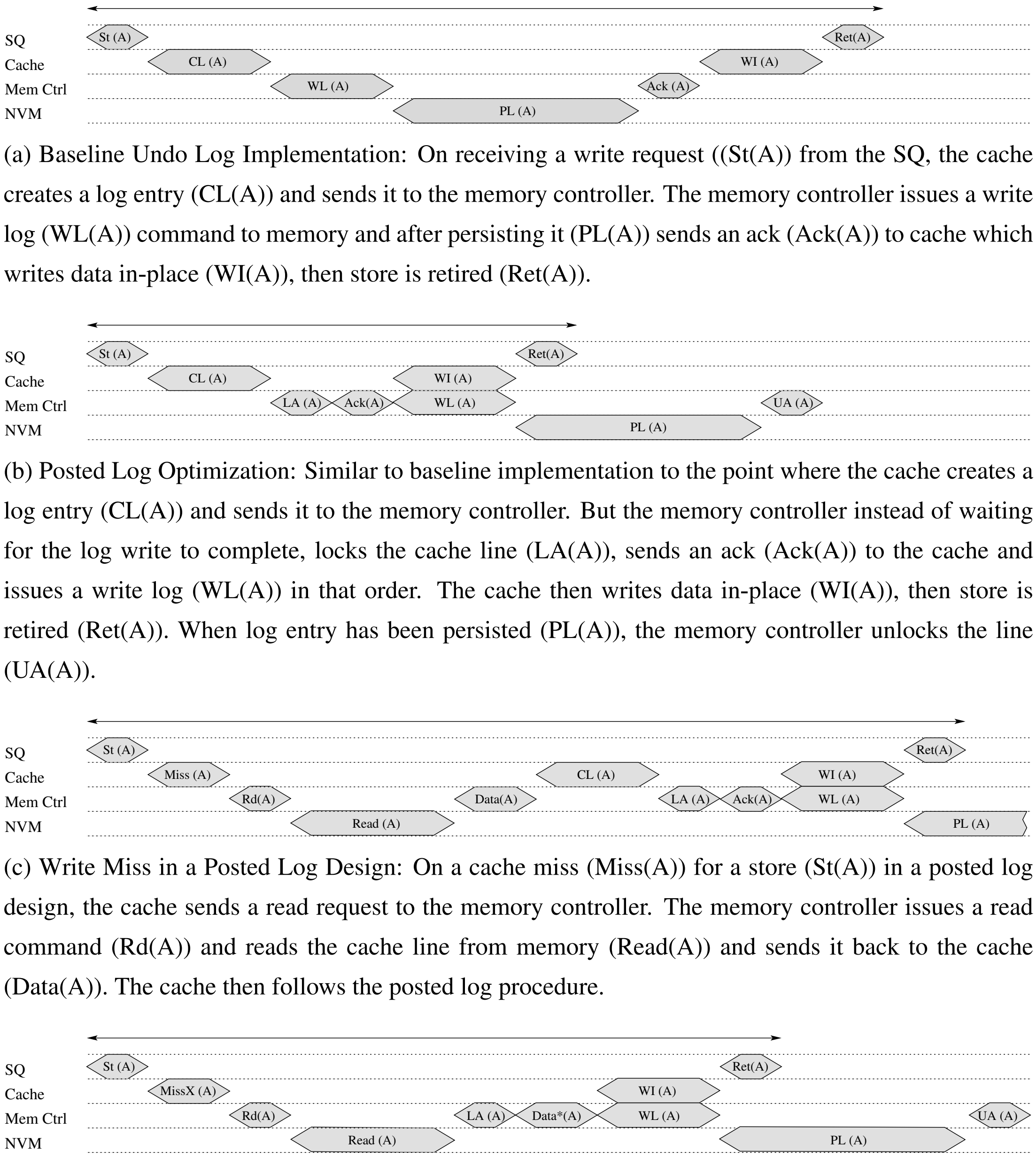
Creating a log entry. A log entry (old value, address pair) has to be created before modifying any data in an atomic update. Hence we use a store operation, belonging to an atomic update, to trigger the creation of log entries. We propose that the log manager in L1 cache couple the creation of a log entry with the processing of a write request from a store operation. Specifically, when the L1 cache controller receives a write request for a cache line, it first sends a log entry to the memory controller by piggy backing on the cache write-back interface. This ensures that the log entry is created before completing the write request, satisfying *Invariant 1*. The memory controller then writes the log entry into the log area in the NVM.

Note that while a cache line can get modified multiple times during an atomic update, it does not have to be logged every time it gets modified. Since an undo log stores the old value, it is sufficient to log a cache line only once: on the first write. To detect the first write to a cache line, we augment all the cache blocks with an additional *log* bit. The log bit is set when a cache line is written to for the first time during an atomic update. It is cleared when the modified value of the cache line is durably written to memory. This mechanism is similar to the log mechanism employed in LogTM [38]. The critical difference is that the log write in this case is not cached but has to be written to NVM. The log bit is only maintained during the lifetime of a cache line in the cache. As soon as a cache line is replaced, information about whether that cache line is logged or not is lost. Therefore, after being flushed to memory, when a cache line is modified again in the same atomic update the log bit is not set and the log manager logs it again. However, this is not a problem for ensuring correct recovery. During recovery the roll backs are applied in the order of newest first. This ensures that, at the end of recovery, the value of a cache line is restored to the one before the atomic update started.

Enforcing log → data ordering. The next task is to enforce ordering between log writes and in-place data writes. Therefore, upon detecting the necessity to log a cache line, the log manager first durably writes a log entry to the log area in NVM. After completing the write, it updates the value of the cache line in the cache and retires the store from the store queue (SQ). This ensures that an in-place data write cannot become durable before the corresponding log entry, thus satisfying *Invariant 2*. Figure 4.3(a) shows the sequence of operations. The log manager in the cache controller, upon receiving a write request from the SQ, checks if the log bit is set for the cache line (A) being updated. If the log bit is not set, the log manager creates a log entry ($CL(A)$) and sends it to the memory controller. The memory controller issues a write request for the log entry ($WL(A)$). After durably writing the log entry to NVM ($PL(A)$), the memory controller sends an acknowledgement ($Ack(A)$) back to the log manager in the cache controller. The log manager then completes the write request by modifying data in-place in the cache ($WI(A)$), which allows the store to be retired from the SQ. Under this baseline design, durably writing the undo log entry is in the critical path of completing the corresponding store operation from the SQ.

Sources of reordering. The log manager cannot allow the update of data in the cache until it receives an acknowledgement that the log entry has been made durable. This is because the cache line containing the modified data can be replaced at any time from the cache and could possibly overtake the log entry to NVM, which in turn will violate *Invariant 2*. This overtaking can happen because of the possible reordering in either the on-chip network (between the cache and NVM) or at the memory controller. Reordering is possible even if we consider the network and the memory controllers to be strictly ordered. It can arise if the log area and the data (cache line) are mapped to *different* memory controllers.

Logging cost. Store operations are typically not in the critical path in modern processors because they employ a queue to buffer store operations. But durably writing the undo log to memory is in the critical path of store operations. This reduces the rate at which store operations are completed from the SQ, which leads to a back pressure that can fill up the SQ and eventually stall the processor pipeline. Thus, it is important to reduce the critical path of store operations.



(a) Baseline Undo Log Implementation: On receiving a write request ((St(A)) from the SQ, the cache creates a log entry (CL(A)) and sends it to the memory controller. The memory controller issues a write log (WL(A)) command to memory and after persisting it (PL(A)) sends an ack (Ack(A)) to cache which writes data in-place (WI(A)), then store is retired (Ret(A)).

(b) Posted Log Optimization: Similar to baseline implementation to the point where the cache creates a log entry (CL(A)) and sends it to the memory controller. But the memory controller instead of waiting for the log write to complete, locks the cache line (LA(A)), sends an ack (Ack(A)) to the cache and issues a write log (WL(A)) in that order. The cache then writes data in-place (WI(A)), then store is retired (Ret(A)). When log entry has been persisted (PL(A)), the memory controller unlocks the line (UA(A)).

(c) Write Miss in a Posted Log Design: On a cache miss (Miss(A)) for a store (St(A)) in a posted log design, the cache sends a read request to the memory controller. The memory controller issues a read command (Rd(A)) and reads the cache line from memory (Read(A)) and sends it back to the cache (Data(A)). The cache then follows the posted log procedure.

(d) Source Logging Optimization: After reading the cache line (Read(A)) on a store miss (MissX (A)), the memory controller locks the cache line (LA(A)). It then sends the cache line back to the cache (Data*(A)) with log bit set, so the cache does not send a log write request. The cache then writes data in-place (WI(A)), then store is retired (Ret(A)). The memory controller, meanwhile issues a write log (WL(A)) request and unlocks the cache line (UA(A)) after persisting the log entry (PL(A)).

Figure 4.3: Sequence of actions of store queue (SQ), cache, memory controller (Mem Ctrl) and non-volatile memory (NVM) for undo logging in NVM based systems.

4.3.3 Posted Log Optimization

Currently, for each store, the critical path includes writing the log durably to memory as shown in Figure 4.3(a). To minimize the performance overhead of enforcing the log → data ordering constraint, we propose to allow the log manager in the cache controller to perform *posted log writes* to the memory controller, where the log manager enforces log → data ordering at the memory controller level. By doing so, we move the performance overhead of durably writing log entry to NVM, out of the critical path.

Figure 4.3(b) shows the sequence of operations for logging with a posted write feature. Upon receiving a write request from the SQ, the log manager in the cache controller sends a log entry to the memory controller. The memory controller locks the cache line (LA(A)) for which the log entry is being persisted and then sends an acknowledgement back to the cache controller. Upon receiving the acknowledgement, the cache controller completes the write request, allowing the store to retire from the SQ (without having to wait for the log write to persist). When the log write eventually completes, the log manager in the memory controller unlocks the cache line (UA(A)). Whenever a write entry is ready to be scheduled out of the memory controller, the log manager is first consulted; only if the cache line is not locked, the write is allowed to go to NVM. In effect, this is a simple and efficient approach to enforcing the log → data ordering at the memory controller.

The posted log optimization cannot be applied if the log and data are mapped to different memory controllers. It can be challenging to ensure log-data co-location in software because an application program might be modifying data scattered behind multiple memory controllers. But because we perform logging in hardware, we are able to ensure that the log entry is sent to the same memory controller as the corresponding data (§4.4.2). Thus, by co-locating log and data behind the same memory controller, we can enable the posted log optimization. With posted log optimization even though a store completes before durably writing the log entry to NVM, log → data ordering is enforced by the memory controller and hence *Invariant 2* is satisfied.

4.3.4 Source Log Optimization

Performing a posted write to the memory controller still incurs the cost of writing to and receiving an acknowledgement from the memory controller in the critical path of the store operation. But this can be further optimized in certain scenarios. Consider the scenario shown in Figure 4.3(c). The cache controller receives a write request for a cache line (*A*). It misses in the cache (Miss(A)), so the cache controller sends a fetch request to the memory controller. When the memory controller responds with the data (Data(A)), the cache controller checks for the log bit, which in this instance is not going to be set since the cache line has just been read from NVM. So the cache sends a log entry for cache line *A* to the memory controller. In a posted log design, the memory controller locks the cache line and responds with an acknowledgement, which completes the write request enabling the SQ to retire the store.

In the above example, however, there is unnecessary data movement from the cache controller to the memory controller in performing the log write. If a cache line is not present in the cache, then the in-place data in NVM is actually the old value of the cache line that needs to be written to the undo log. So the data that the cache controller sent back along with the undo log request is actually the same data that it just received from the memory controller because of its fetch request. This data movement from the cache controller to the memory controller can be avoided if the memory controller itself can write the old value of the cache line in the log area. We call this optimization *source log optimization* and is shown in Figure 4.3(d). The cache controller on detecting a miss on a write request (MissX (A)), sends a fetch exclusive request to the memory controller. The memory controller follows the posted log procedure after reading the cache line from NVM (Read(A)). It first locks the cache line (LA(A)), and then sends a data response to the cache with the log bit set (Data*(A)). On receiving data with the log bit set, the cache controller completes the write to the cache line. The memory controller, after sending the data to the cache, writes the log entry to NVM and eventually unlocks the cache line (UA(A)) on completion of the log write (PL(A)). Thus, this technique completely removes logging out of the critical path for stores that miss in the cache. It also eliminates redundant data movement.

4.4 ATOM Architecture

In this section, we present the architectural and implementation details of ATOM.

4.4.1 Overview

The primary functions of ATOM are initiating log writes, managing log space (log allocation and clearing) and enforcing the log → data ordering constraint. These functions are implemented across two modules. The *log write initiate* module (LogI) and the *log manage* (LogM) module. The LogI module is embedded in the L1 cache controller as shown in Figure 4.4(a) and is responsible for initiating log write requests. The LogM module is embedded in the memory controller as shown in Figure 4.4(a) and is responsible for managing log space and enforcing the log → data ordering constraint. ATOM supports one atomic update per core. But it allows for concurrent execution of atomic updates across different cores by creating multiple (one per core) instances of the tracking structures in the LogM module.

4.4.2 Log Write Initiate (LogI) Module

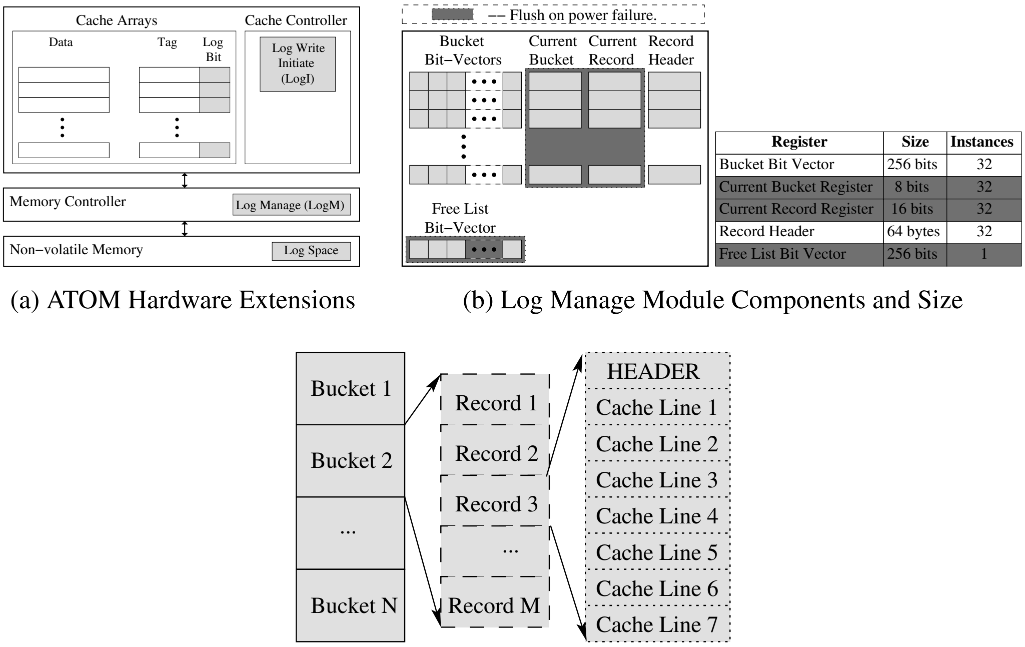
As discussed in §4.3.1 we extend the processor-to-memory system interface to include two new commands, *Atomic_Begin* and *Atomic_End*. These commands signify the start and the end of an atomic update respectively. The memory system, upon seeing the *Atomic_Begin* command, will start logging for the cache lines being modified by subsequent stores. It will stop logging upon receiving an *Atomic_End* command. We handle nested atomic sections by flattening them.

The LogI module looks at the log bit of each cache line before completing a write request. If the bit is set, the write request is immediately serviced. Otherwise the write request is stalled, a miss status handling register (MSHR) is allocated and a log write request is initiated to the memory controller associated with the corresponding cache line. The memory controller associated with the cache line is easily determined from the cache line address. By sending the log request to the same memory controller as the data, we ensure log-data co-location.

4.4.3 Log Manage (LogM) Module

ATOM's LogM module manages a central log space which is shared across all threads and is statically allocated by the OS. ATOM manages this log space in terms of records and buckets as is described next.

Log Record Organization. We consider a system with 64 byte cache lines and ATOM performs logging at a cache line granularity. Therefore, each log entry consists of a cache line as data and address as meta-data. The simplest way to organize logging is to allow all threads to create individual log entries in the central log space. Writing a log entry to NVM in this way would require 2 write requests to memory since the size of a log entry is greater than a cache line. To minimize the overhead of multiple write requests we propose log entry collation (LEC), in which multiple log entries are collated into a single log record. The size of each log record is 512 bytes (or 8 cache lines). A log record can contain up to 7 log entries, and is divided into data (7 cache lines) and header (1 cache line) as shown in Figure 4.4(c). The header contains the addresses



Register	Size	Instances
Bucket Bit Vector	256 bits	32
Current Bucket Register	8 bits	32
Current Record Register	16 bits	32
Record Header	64 bytes	32
Free List Bit Vector	256 bits	1

(a) ATOM Hardware Extensions

(b) Log Manage Module Components and Size

(c) Log Organization

Figure 4.4: ATOM Components.

of all the 7 cache lines, the number of cache lines logged in the current record, and some reserved bits. On receiving a log write request, only the data field is written to memory at first. The meta-data for the log entry (consisting of its address) is added to the *record header*. A log entry is not considered durable until its corresponding record header persists. After logging 7 cache-lines, the header is written to memory, thus persisting the entire log record. When all 7 log entries in a log record are occupied, LEC reduces the overhead of writing a log entry: from 2 write requests for 1 log entry to 8 write requests for 7 log entries, which is a 57% reduction in the number of write requests to memory for logging.

LEC can lead to a violation of Invariant 2 if the cache line containing an in-place update is replaced from the cache and is made durable before the log header corresponding to its log entry. To avoid this, and before writing any data cache line to NVM, the memory address is compared to the addresses in the record header. The data cache line is written to memory only if there is no match in the header. If there is a match, then the header is first made durable to complete the log write and then the data write is allowed to persist in NVM. Adding the address of a cache line in the record header corresponds to the concept of locking the cache line described in §4.3.3. The record header is cleared after persisting it in NVM, which corresponds to unlocking the cache lines.

The centralized shared log space can potentially be managed at a log record granularity. The log manager can maintain a log record head pointer and keep adding new log records to the central log space based on the requests received from different atomic updates. There are two ways to clear such a log. In the first approach, on completion of an atomic update, the log manager can read the log space starting from the beginning of log and clear all records belonging to the corresponding atomic update individually until the corresponding commit record is encountered. Unfortunately this will generate additional memory read requests to read log record headers sequentially and additional memory write requests to clear records corresponding to the completed update. Moreover, this will leave the log space fragmented. In the second approach the log manager – instead of clearing log records on completion of atomic updates – can wait for the completion of all concurrent atomic updates and then clear the log space. This method will avoid fragmentation, but will stall the processing of new atomic updates during the wait.

Log Bucket Organization. To overcome these limitations we propose dividing the shared log space into buckets of log records and managing log space allocation and deallocation at a bucket granularity, resulting in the organization shown in Figure 4.4(c). An atomic update has an associated bit-vector, known as *bucket bit vector*, indicating the buckets allocated to that update. Using a bit-vector alleviates the first problem of requiring additional memory read and write requests to allocate or clear the log as it can be used to identify free buckets and to clear allocated buckets. Along with the bit vector, there is a *current bucket* register that identifies the bucket to which log records are being added currently; a *current record* register that indicates the record in the current bucket being written to; and, finally, a *record header* register that stores the meta-data for the log record currently being updated as shown in Figure 4.4(b). A new bucket is allocated from the *free list bit vector*, which is generated by NORing all the bucket bit vectors.

The bucket bit vector and current bucket, current record and record header registers – together track a single atomic update and are collectively known as an *atomic update structure* (AUS). So to support concurrent atomic updates, these need to be replicated as shown in Figure 4.4(b). We support up to 32 concurrent updates in our system (1 per core). The sizes of all the registers is shown in Figure 4.4(b). The space overhead of LogM module amounts to 3.125 KB.

The bucket organization, by allocating log buckets from a central pool, allows for dynamic sharing of log space by concurrent atomic updates. It also simplifies log clearing on completion of an atomic update. LogM does not have to read the log space, but only has to clear the bit vector corresponding to the atomic update and update the free list bit vector. This is a single cycle operation and will be completed even if a power failure occurs at the moment of clearing the log.

4.4.4 Recovery

After a power failure, the incomplete atomic updates need to be undone to restore the system to a consistent state. The enforcement of *Invariant 2* guarantees that at any point of time during execution, if a log entry has not persisted then the corresponding data would not have persisted either. Hence in the event of a power failure, all the pending log writes in the memory controller store buffers can be safely discarded. Only those log entries that have already persisted need to be considered during recovery. However, on a power failure the information about valid log buckets in the memory controller will be lost. To correctly access the log space we need to be able to identify which buckets are valid (contain valid log records). This can be identified by taking a complement of the *free list bit vector*. Also, some of the valid buckets might be partially filled because log entries were being added to them when the power failure occurred. These partially filled buckets can be identified from the *current bucket register*. And finally the number of valid log records in those partially filled buckets can be identified from the *current record register*. The total size of the above 3 critical structures is 128 bytes. To ensure that these critical structures are preserved, we utilize a feature similar to Asynchronous DRAM Refresh (ADR) [56] supported by Intel. ADR ensures that on a power failure, all the memory controller buffers (24 or more cache lines) are flushed to memory. In our implementation, only the critical structures (amounting to only 2 cache lines) need to be written to NVM on detecting a power failure.

Recovery after a power failure is accomplished in software through a generic recovery routine provided as a system call which relieves the programmer from having to implement custom recovery schemes. The recovery routine will read the bucket bit vectors and current bucket and record information from NVM and reconstruct the state of the log space at the time of the crash. It will then perform undo operations in the reverse order starting from the last log record to the first one for each incomplete atomic update. The recovery routine performs undo operations for all the cache lines recorded in the log even though some of the cache lines may not have been updated in memory at the time of crash. This might impose a performance overhead during recovery but does not affect the correctness.

4.4.5 Log Allocation and Overflow

In our design, a central log space is allocated by the operating system (OS) which is shared between concurrent atomic updates. The OS is aware of the number of physical pages associated with each memory controller. It reserves a proportional number of these pages as the log area. The OS then ensures that no virtual page is mapped to any of these reserved log pages. Recall that the LogI module ensures that each log entry is correctly directed to the memory controller where the corresponding data page resides.

There can be two kinds of overflows in the system. The first type of overflow, known as structural overflow, occurs when the number of concurrent update requests are higher than the number of updates supported by the hardware. An *Atomic_Begin* instruction checks for the availability of an AUS. If an AUS is not available it will stall. Eventually as other atomic updates complete (execute *Atomic_End* instruction), an AUS will free up and will be allocated to the stalled update. The waiting update does not have any resources reserved and hence cannot block any other update. Thus, a structural overflow cannot result in a deadlock.

The second type of overflow, known as log overflow, occurs when a new bucket needs to be allocated behind a memory controller, but no more buckets are available in the corresponding free list bit vector. In other words, all of the reserved log pages in the memory controller have been exhausted. In this scenario, the OS is interrupted to allocate additional log pages for that memory controller, which will be used to store subsequent log records. Because this additional resource (log space) is allocated to the requesting update, it will make forward progress and not block any other update. Hence, a log overflow will also not result in a deadlock. Moreover, dynamically sharing the log space between atomic updates reduces the probability of log overflow as opposed to a design where the log space is statically partitioned.

4.5 Hardware Checkpointing

As an application of the ordering and atomic durability primitives, we propose a new mechanism to efficiently checkpoint programs in this section. This mechanism leverages the persist barrier from Chapter 3 to create checkpoints and combines it with ATOM to guarantee atomic durability of those checkpoints. This is analogous to implementing a strict persistency model albeit in bulk mode.

Recall that, strict persistency couples memory persistency with memory consistency (§2.3). With buffering, although program execution is decoupled from persist operations, stores still need to persist in the same order as they are made visible. So in systems with Total-Store-Order (TSO) as the memory consistency model, buffered strict persistency (BSP) enforces that stores persist in program order. Therefore, even with buffering, there would be frequent conflicts resulting in a larger percentage of persist operations happening in the critical path.

We propose to enforce BSP in bulk mode, to minimize the number of persist operations happening in the critical path. Instead of enforcing persist ordering constraints at memory operation granularity, we enforce them at an epoch granularity. This is similar in spirit to the way Sequential Consistency is enforced by BulkSC [57].

BSP is implemented completely in hardware. Hence, no programmer annotations in the form of persist barriers are required. A hardware persistence engine divides the sequence of stores from a program execution into epochs. Persistency is enforced at the granularity of epochs using the efficient persist barrier (LB++) presented in Chapter 3 (§3.3). At the end of each epoch, along with the modified cache lines, processor state is also saved to persistent memory. This state can be used to restart the process, similar to the way it is done in [58]. It is worth noting that epoch boundaries are the points at which BSP holds. At the time of a crash though, some epochs might have persisted partially and therefore BSP might be violated. To overcome this problem we propose to employ ATOM, which will log all the updates performed in an epoch and undo any partially persisted epochs.

Another issue that can arise in this mechanism is the possibility of epoch deadlocks. Since hardware dynamically creates epochs, it is oblivious to dependencies between threads. This could lead to epoch deadlocks. We use the solution presented in Chapter 3 (§3.3.3) to overcome this problem.

4.6 Experimental Setup

We now describe our simulation infrastructure, system configuration, benchmarks and designs that we evaluate. We implemented ATOM and BSP on gem5 [42] with Ruby in full system simulation mode. The on-chip interconnect is modelled using Garnet [50].

Cores	32 OoO cores @ 2GHz
ROB Size	192 Entry
Store Queue	32 Entry
L1 I/D Cache	32KB 64B lines, 4−way
L1 Access Latency	3 cycles
L2 Cache	1MB×32 tiles, 64B lines, 16−way
L2 Access Latency	30 cycles
MSHRs	32
Memory Controllers	4
NVM Access Latency	360 (240) cycles write (read)
On-chip network	2D Mesh, 4 rows, 16B flits

Table 4.1: System Parameters.

We extend the Ruby memory model to implement the proposed log manager. We evaluate ATOM and BSP on a 32-core multicore (1 thread per core) with multi-banked LLC and 4 memory controllers placed on the corners of the die. We consider a MESI based coherence protocol for our evaluation. Table 4.1 shows the main parameters of the system. The memory write latency that we consider is 10× that of typical DRAM latency. We assume a single memory channel per memory controller unless otherwise stated. The peak memory bandwidth in our setup is 5.3 GB/s per memory channel. We model an address match latency of 1 cycle in the memory controller to check if the data write request has a corresponding log entry pending in the record header.

Workloads. We use the micro-benchmarks listed in Table 4.2 to evaluate ATOM and the proposed optimizations. These micro-benchmarks implement data structures that are similar to those in the benchmark suite used by NVHeaps [12], except for the queue micro-benchmark, which is similar to the copy-while-locked queue of [7]. We evaluate these workloads with two data set sizes (table entries, tree nodes, queue entries etc.): small (512 bytes) and large (4 KB). Each benchmark performs search and atomic insert and delete operations on the corresponding data structure.

We also evaluate ATOM using the TPC-C benchmark where the TPC-C schema is implemented using B$^+$-Trees [11]. We use a scaling factor of 1 and use 32 threads to simulate the 32 terminals issuing new order transactions. Our goal is to measure the overhead in write-intensive operations. Therefore, the new order transaction is the best choice as it is the most write-intensive TPC-C transaction. We slightly modified the

Hash	Insert/delete entries in a hash table
Queue	Insert/delete entries in a queue
RBTree	Insert/delete nodes in a red-black tree
BTree	Insert/delete nodes in a b-tree
SDG	Insert/delete edges in a scalable graph
SPS	Random swaps between entries in an array

Table 4.2: Micro-benchmarks used in our experiments.

benchmark and removed the wait times (implemented using *sleep* system call) to allow us to execute the benchmark in a reasonable amount of time.

We employ BSP for a scenario where long running applications that are periodically checkpointed to reduce the amount of lost work due to system crashes. To evaluate the same, we use benchmarks from PARSEC [43], SPLASH-2 [44] and STAMP [45] benchmark suites. The benchmarks were unmodified; persist barriers are inserted transparently by the hardware to ensure BSP. In our experiments, we model the overhead of checkpointing all general purpose, special registers, privilege registers and floating point registers (non-AVX) as part of the processor state. We ran all the workloads to completion.

4.7 ATOM Evaluation

This section first lists the various designs for atomic durability that we consider and then presents an evaluation of those designs.

4.7.1 Designs

- **BASE**: The baseline hardware undo log which performs logging transparently in hardware (without additional instructions for logging), but the log write happens in the critical path of a store operation (§4.3.2).
- **ATOM**: Proposed design with posted log optimization (§4.3.3).
- **ATOM-OPT**: The above with source log optimization as well (§4.3.4).
- **NON-ATOMIC**: No logging operations are performed, and hence this design represents upper bound on performance for a logging implementation. On completion of each atomic update, all the data modified within the atomic update is still written back to NVM.

– **REDO**: The redo log design of Doshi et al. [15] with a couple of modifications (that actually benefit their design). First, although their implementation requires additional log write instructions in software, we do this in hardware by allowing the cache to issue log writes on receiving a store, for the sake of fair comparison. Second, we consider an infinite size victim cache. Similarly to their design, we implement write combining for log writes.

4.7.2 Evaluation

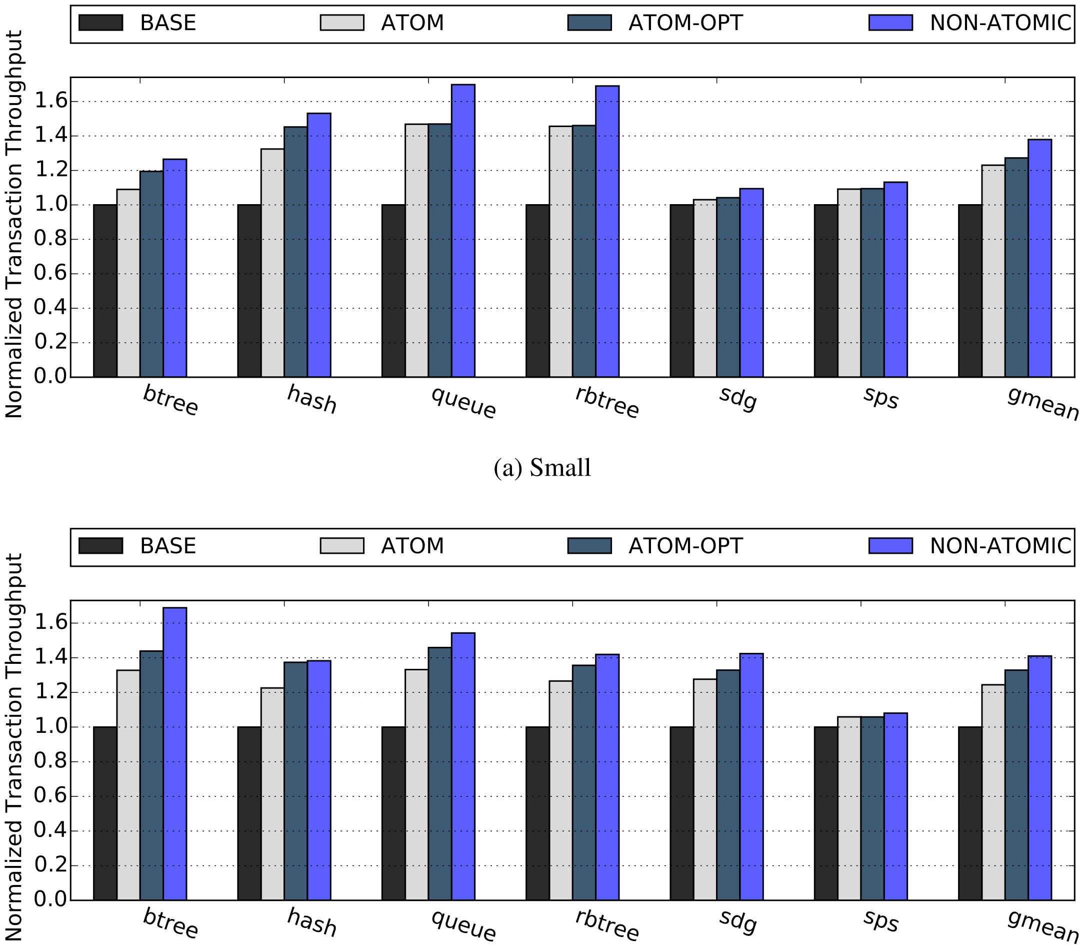
We first present the speed-up due to ATOM and analyze the impact of both posted logging and source logging optimizations. We then show how ATOM reduces the critical path of logging operations by looking at the occupancy of the store queue and also analyze the reasons behind the magnitude of performance improvement due to source logging. We also compare ATOM with a REDO log based design [15] and perform a sensitivity study by varying memory latency. Finally we present the performance of ATOM for the TPC-C benchmark.

4.7.2.1 Transaction Throughput

Figure 4.5(a) shows transaction throughput for the ATOM, ATOM-OPT and NON-ATOMIC designs, normalized to BASE for small dataset sizes. On average, ATOM improves transaction throughput by 23%. Recall that the posted log optimization reduces the critical path of store operations by enforcing log $\rightarrow$ data ordering at the memory controller. ATOM-OPT improves the throughput by 27% on average over BASE which is a 4% improvement over ATOM. Recall that source logging optimization further reduces the critical path of stores that miss in the cache by eliminating the log write request from cache to memory controller. The improvement because of this optimization will depend on the percentage of log writes that are source logged. We analyze this further in §4.7.2.3.

The NON-ATOMIC design has a 38% higher throughput than BASE. The ATOM-OPT design, by improving the throughput by 27%, is able to close about 71% of the performance gap between BASE and the optimal (NON-ATOMIC) design.

Figure 4.5(b) shows the normalized transaction throughput for large dataset sizes. On average, ATOM improves the transaction throughput by 24%, while ATOM-OPT improves it by 33% over BASE which is a 9% improvement over ATOM. For large dataset sizes NON-ATOMIC design improves the throughput over BASE by 41%



(a) Small

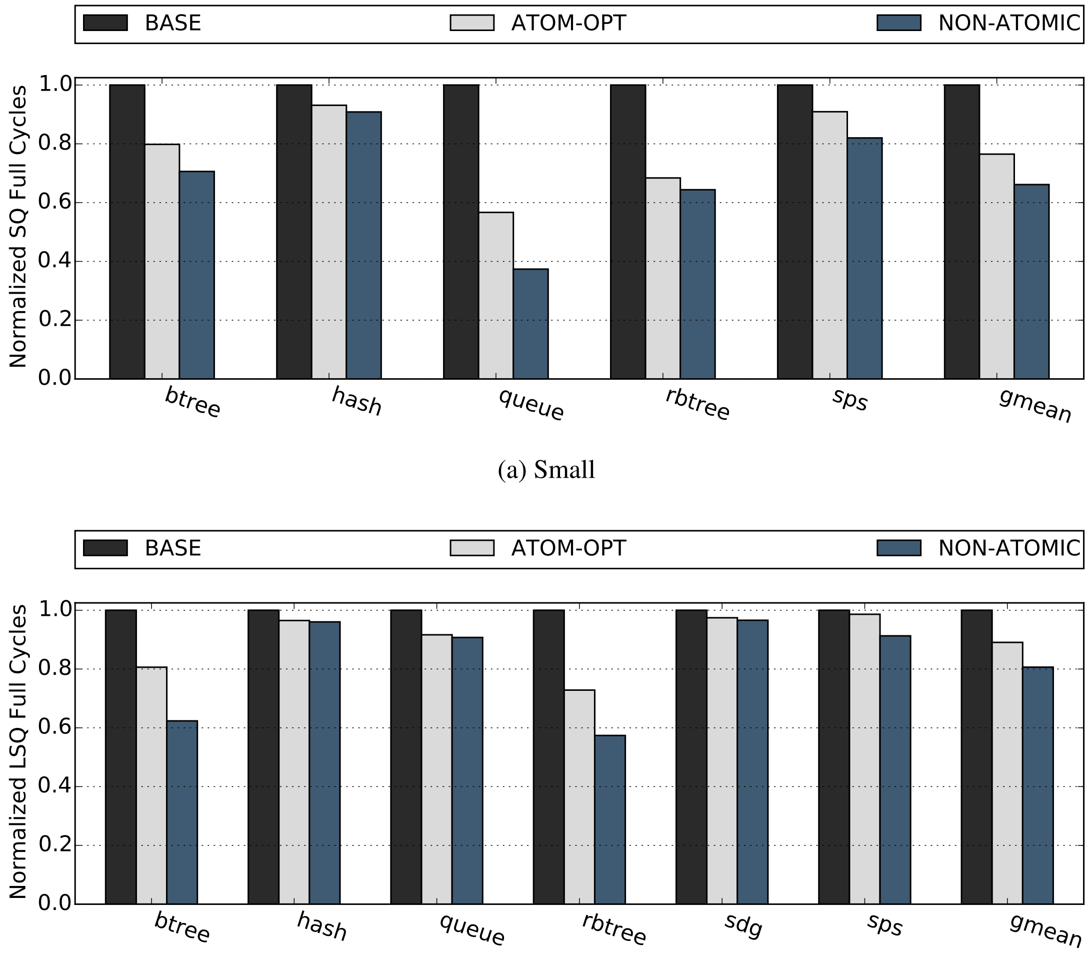
(b) Large

Figure 4.5: Transaction throughput normalized to BASE for micro-benchmarks.

and ATOM-OPT design is able to close 83% of this performance gap between NON-ATOMIC and BASE designs.

4.7.2.2 Impact on Critical Path

Store operations are not typically in the critical path of program execution because most processors employ store queues (SQ) to complete stores out of the critical path. But with logging, writing to NVM is in the critical path of completing store operations from the SQ. This creates a back pressure, which eventually fills up the SQ and stalls the processor pipeline. Figure 4.6(a) shows the number of SQ-full events for ATOM-OPT and NON-ATOMIC designs, normalized to BASE for benchmarks with small dataset sizes. ATOM-OPT reduces the SQ-full cycles by 21% on average which correlates with the increase in throughout. Benchmarks with high reduction, like *queue*



(a) Small

(b) Large

Figure 4.6: SQ full cycles normalized to BASE for micro-benchmarks with small dataset size.

(43%) and *rbtree* (35%) also show high improvement in throughput: 47% and 46% respectively. Similarly, *sps* which has the minimum reduction (1%) shows the minimum improvement in throughput (4%). On average ATOM-OPT has only 10% more SQ-full cycles than NON-ATOMIC.

Benchmarks with large dataset sizes show a similar trend (Figure 4.6(b)). In these benchmarks the average reduction in the number of SQ-full cycles drops to 11% from the high 21% seen for benchmarks with small dataset sizes. With increasing dataset sizes, the number of cache lines to be written back at the end of an atomic update increases. This places additional pressure on the SQ occupancy and hence the scope for reducing SQ-full cycles decreases.

	btree	hash	queue	rbtree	sdg	sps
small	0.12	0.12	0.07	0.01	0.04	0.01
large	0.4	0.4	0.7	0.4	0.07	0.01

Table 4.3: % of source logged cache lines for ATOM-OPT

4.7.2.3 Source Logging

The source logging optimization removes log writes from the critical path for store operations that miss in the cache. ATOM-OPT logs the cache lines for which a fetch exclusive request is received by the memory controller during an atomic update. Table 4.3 shows the percentage of source logging for benchmarks with small and large datasets. We see that even with as little as 0.12% of log writes being source logged, ATOM-OPT provides a transaction throughput improvement of 10% and 13% over ATOM for *btree* and *hash* benchmarks respectively, for small datasets.

As the dataset size grows, the percentage of store operations missing in the cache increases. We see that *queue*, which has the highest percentage (0.7%) of source logging, provides the highest throughput improvement (16%) for ATOM-OPT over ATOM. Moreover, *sps*, which has the lowest percentage of source logged cache lines for both small and large datasets does not show any improvement compared to ATOM.

4.7.2.4 Comparison with Redo Log

We compare ATOM-OPT with the recently proposed REDO log design [15]. In addition to the setup of §4.6, we also evaluate these designs in a configuration with two memory channels at each memory controller (*-2C), where one channel is used for data while the other channel is used for logging, in order to mimic the configuration used by the authors in [15]. Figure 4.7 shows the transaction throughput normalized to ATOM-OPT. In the single channel configuration REDO is only able to achieve 22% of the transaction throughput of ATOM-OPT while in the two channel configuration it is able to achieve 30%. We identified that the disparity in performance between ATOM-OPT and REDO is because of the difference in their memory bandwidth requirements.

REDO generates 19× more log entries than ATOM-OPT. This is because in REDO, every store operation in an atomic section generates a log entry. Whereas in ATOM, a log entry is generated only on the first write to a cache line. These log entries increase the pressure on the memory write bandwidth. Moreover, in REDO the log entries have to be read from memory to perform in-place data updates. These log read requests

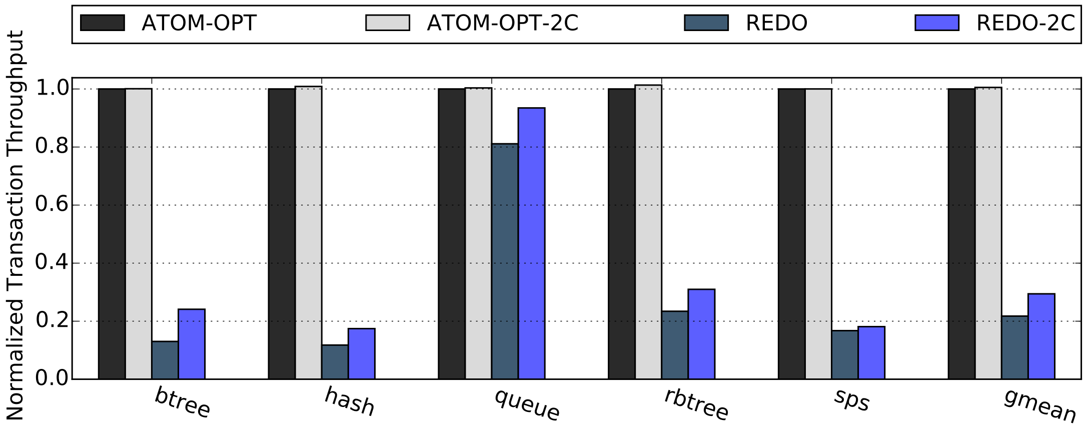


Figure 4.7: Transaction throughput for REDO and ATOM-OPT designs normalized to ATOM-OPT for benchmarks with small dataset size.

interfere with the critical data read requests from the cores, thus slowing down execution. In the two channel configuration, the log and data reads go to separate channels and hence the log read requests do not interfere with the critical data read requests. Therefore, the throughput of REDO-2C increases by 9% over REDO.

4.7.2.5 Sensitivity to Memory Latency

Figure 4.8 shows the transaction throughput for the *rbtree* benchmark with small dataset size for varying memory latencies (as a ratio of DRAM latency). At NVM latencies similar to DRAM, REDO provides higher transaction throughput than ATOM-OPT because of two reasons. First, the low latency memory is quickly able to absorb the large number of log writes generated by REDO. So this is no more a bottleneck for REDO. Second, REDO performs in-place updates of data in the background, whereas ATOM-OPT has to persist all the in-place modifications to NVM at the end of each atomic update. But on increasing the latency, the performance of REDO degrades super-linearly because of the relatively high memory bandwidth requirement. The throughput of ATOM-OPT degrades almost linearly because its memory bandwidth requirement is lower than REDO.

4.7.2.6 TPC-C

As a case study we evaluate ATOM using TPC-C, annotating all the critical sections as atomic regions. Table 4.4 shows the throughput for TPC-C normalized to BASE. ATOM provides a throughput improvement of 58% over BASE whereas ATOM-OPT

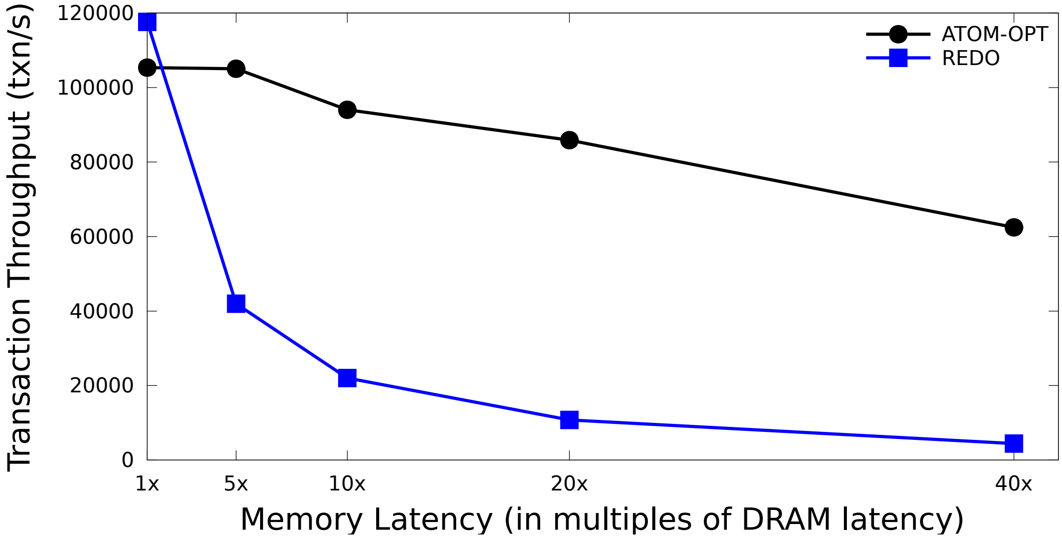


Figure 4.8: Transaction throughput variation (ATOM-OPT vs REDO) with varying memory latency.

	BASE	ATOM	ATOM-OPT	REDO
Throughput	1	1.58	1.6	1.47

Table 4.4: TPC-C throughput normalized to BASE.

provides an improvement of 60% over BASE. ATOM-OPT provides negligible improvement over ATOM because only 0.02% of log operations were source logged. ATOM-OPT reduces the SQ-full cycles by 42%.

REDO on the other hand provides a throughput improvement of 47% over BASE (13% lesser improvement than ATOM-OPT). It is worth noting that both ATOM and REDO provide higher gains for TPC-C as opposed to micro-benchmarks. This is because TPC-C has relatively lower frequency of updates in comparison to micro-benchmarks, and hence memory bandwidth is less of a problem.

4.8 Evaluation of Hardware Checkpointing

Recall that, we have proposed to implement strict persistency (BSP) in bulk mode as an application of the ordering and the atomic durability primitives. We use this implementation as a mechanism to checkpoint programs in hardware. Our primary goal in this section is to understand the overheads of checkpointing programs. A secondary goal is to use the implementation of strict persistency in bulk mode as a tool to quantitatively evaluate the benefits of buffering, buffering optimizations and hardware logging. In this section, we first present the designs that we consider followed by their evaluation.

4.8.1 Designs

As stated above, one of our goals in this evaluation is to understand the performance benefits of buffering, optimizations for buffering and hardware logging. To evaluate the benefits of ATOM-OPT (hardware logging) we consider two designs of strict persistency (without buffering), where one design implements unoptimized hardware logging (§4.3.2) for atomic durability while the other design leverages ATOM-OPT with both posted log (§4.3.3) and source log (§4.3.4) optimizations. We contrast these designs with designs for implementing buffered strict persistency to evaluate the benefits of buffering. The designs for implementing buffered strict persistency use ATOM, with both of its optimizations, for atomic durability and leverage lazy barrier with and without optimizations to understand their benefits. To understand the overheads of checkpointing, we compare all the above stated designs with respect to a design that does not perform checkpointing. Below, we list the designs along with their brief description.

- **NP**: A design that provides no guarantees on what will be present in persistent memory on a system crash. In other words, this design does not create any checkpoints, so in case of a system crash applications will have to be restarted from the beginning. We use this design as a baseline to understand the overheads of checkpointing.

The following design implements unoptimized hardware logging (§4.3.2), to guarantee atomic durability.

- **SB-BASE**: Strict persistency implementation without buffering where all the updates of an epoch have to be made persistent before executing the next epoch (§2.3).

The following designs implement ATOM-OPT to guarantee atomic durability.

- **SB**: Same as SB-BASE except that logging is not performed in software (§2.3).
- **LB**: Baseline lazy barrier implementation without any optimization (§3.2.1).
- **LB+IDT**: Baseline lazy barrier implementation with inter-thread dependence tracking optimization (§3.3.1).
- **LB++**: Baseline lazy Barrier implementation with both proactive flush and inter-thread dependence tracking optimizations. (§3.3).

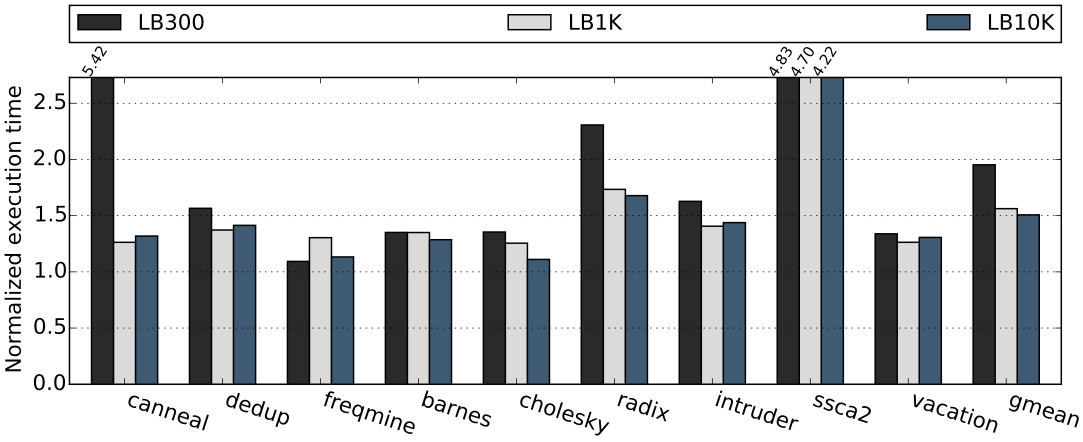


Figure 4.9: Execution time with varying epoch sizes normalized to NP.

4.8.2 Evaluation

In this section, we evaluate the performance overhead of achieving BSP in bulk mode. We target the x86 architecture which supports a variant of TSO, hence the resultant persistency model is also the same. We compare the performance of achieving BSP relative to a baseline that provides no guarantees: No Persistency (NP). It is worth noting that NP also stores its data in persistent memory and hence incurs its relatively high latency. Since persist barriers are inserted dynamically by hardware in BSP, we first perform a study to find the optimum granularity at which to insert these barriers. We then compare the performance of various designs.

4.8.2.1 Epoch Size

In BSP, store operations are divided into epochs dynamically by hardware. How large should the epoch size be? Smaller epochs are desirable as that would mean lesser work lost, whereas larger epochs are expected to be more efficient. Therefore, we analyze the performance impact of varying epoch size; we consider sizes (dynamic stores) of 300 (LB300), 1000 (LB1K) and 10000 (LB10K). We use the unoptimized persist barrier (LB) for this study. Figure 4.9 shows the execution time of benchmarks for designs with varying epoch sizes normalized to NP.

We observe that, on average, performance improves with increasing epoch size. LB300 has an execution time overhead of 1.9×, whereas LB1K has a significantly lower overhead (40% reduction with respect to the baseline). This is because increasing the epoch sizes provides the opportunity for multiple writes to same cache line (belonging to the same epoch) to be coalesced, thereby decreasing the number of persists.

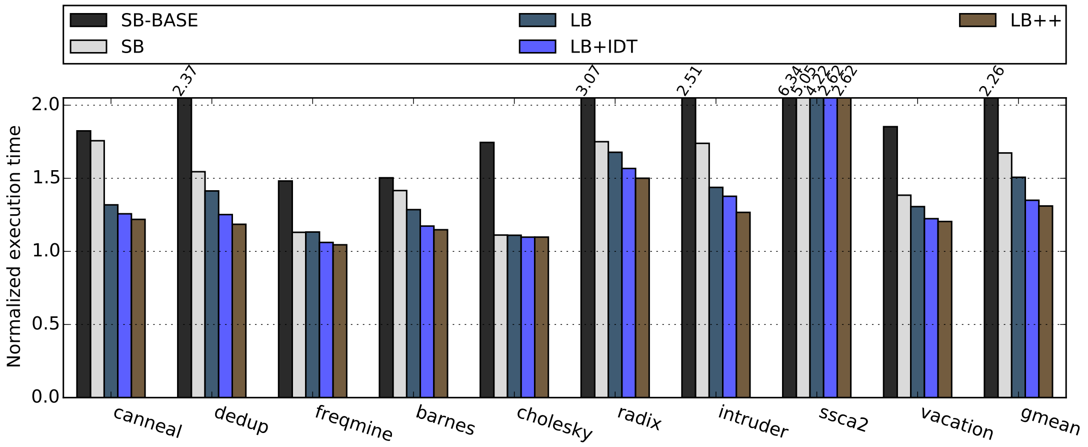


Figure 4.10: Execution time normalized to NP.

For the same reason, we observe that LB10K performs marginally better than LB1K – although interestingly on some benchmarks like canneal, dedup, intruder and vacation LB1K outperforms LB10K. We believe this is because epoch size increases the number of epoch conflicts, so with persist coalescing providing diminishing returns, epoch conflicts starts to dominate. This also explains why performance improvement saturates beyond epoch size greater than 10000 stores (not shown).

4.8.2.2 Performance Analysis of Optimizations.

In this section, we analyze the overheads associated with checkpointing programs and also the performance benefits of buffering, the optimizations proposed for buffering and the impact of ATOM-OPT for enforcing strict persistency in bulk mode. We consider an epoch size of 10000 for this study, as this is what gave the best results. Figure 4.10 shows the execution time of all the designs normalized to the non-persistent design (NP).

We being by looking at the performance of a barrier enforcing strict persistency in bulk mode, without buffering and by employing baseline logging (SB-BASE). As shown in Figure 4.10, SB-BASE has a 126% higher execution time compared to NP. This is primarily because all the updates in an epoch in SB-BASE have to be written to persistent memory before executing the next epoch. Moreover, because of logging each store translates into two writes, one for log and other for data, and the log write has to reach persistent memory before data write can complete. When support for atomic durability in the form of ATOM-OPT is added to a strict persistency barrier (SB) the execution time overhead reduces to 67%, which is 59% lower than SB-BASE. ATOM-

OPT significantly improves performance compared to baseline logging by moving log writes to persistent memory out of the critical path. ATOM-OPT provides the maximum improvement of 138% for ssca2 because it is a write intensive benchmark and thus generates a large number of log writes.

Now we look at the overhead of ensuring strict persistency with a buffered implementation using a lazy barrier combined with ATOM-OPT to guarantee atomic durability. The unoptimized lazy barrier (LB), reduces the execution time overhead to 50%. Buffering without optimization reduces the execution time overhead by only 17% compared to a non-buffered implementation (SB) because on average 88% of epochs persist due to conflicts. However, we observe that adding the IDT optimization (LB+IDT) reduces the overhead of strict persistency to 35%. LB+IDT is able to achieve a 15% improvement over LB because a large number (86%) of conflicts are inter-thread conflicts, which IDT is able to optimize on. LB++ further reduces the overhead to 30%, an improvement of 20% with respect to LB and 37% with respect to a non-buffered implementation (SB). The performance improvement provided by LB++ over LB is much more pronounced for some benchmarks. For instance, ssca2 sees an execution time reduction of 160% because it is a write intensive benchmark with fine grained interaction between threads and the number of epochs that need to persist for it is very high.

Although using our optimized persist barrier provided a significant improvement, there is still an overhead of 30% over NP which we quantify as the overhead of checkpointing. Since our implementation of BSP requires logging, we wanted to understand how much of the residual overhead is due to logging. To this end, analyzed execution time for an implementation of bulk persistency using the optimized persist barrier without performing logging (in software or hardware). We find this overhead to be 16% over NP. From this we can conclude that about half of the residual overhead of 30% on LB++ is due to logging.

We can draw the following conclusions from this evaluation. First, hardware support for logging that moves persist operations out of the critical path is essential for performance as can be seen from the results where SB, which employs ATOM-OPT, improves the performance over SB-BASE by 59% on average. This is primarily because logging introduces fine grained (memory operation granular) persist dependencies which degrade performance even if the underlying persistency model supports relaxed ordering at epoch granularity. Second, buffering can help improve performance by decoupling program execution from performance. But the extent of performance improvement will depend on the design of buffered models and the characteristics of the underlying program. A sub-optimal design that does not avoid or handle conflicts efficiently will provide limited improvement (17% improvement of LB over SB). Although an optimized design can provide further performance improvement (20% improvement of LB++ over LB) it will vary depending on the ratio of conflicting epochs.

4.9 Related Work

Non-volatile memory (NVM) technologies have been studied for various application scenarios, e.g., program checkpointing [59, 60, 61], databases [55, 62, 63, 64], in-memory persistent data structures [10, 11, 12, 13, 14, 60, 65] and file systems [8, 35]. All these scenarios require support for atomic durability, which can be implemented using either WAL or shadow paging.

Systems like Mnemosyne [13], REWIND [11] and Atlas [10] support atomic durability through write ahead logging implemented in software. Hence they rely on the *pcommit* instruction which enforces the log $\rightarrow$ data ordering in the critical path of execution. In [66], the authors propose a software approach to reducing ordering overhead for providing atomic durability, by reducing the number of copy operations and by persisting data in bulk. In [55], the authors propose a group commit mechanism to amortize the cost of persist ordering constraints within a transaction. All these techniques have to persist the log in the critical path.

Many hardware techniques have been proposed to avoid persisting the log in the critical path of execution. NVHeaps [12] relies on *epoch barriers* [7, 8] to persist the log out of the critical path. Implementing *epoch barriers*, however, requires significant changes to the cache hierarchy. Besides, their efficacy is limited (and hence performance sub-optimal) for smaller epoch sizes (§4.8.2.1). In a concurrent proposal [48], the authors propose delegated persist ordering that, similar to our posted log optimization, enforces ordering constraints at the memory controller. However, they only provide ordering but not atomic durability. In LOC [51], the authors provide hardware support for atomic durability through redo logging. Their proposal again requires extensive changes to the cache hierarchy along with support for multi-versioned caches.

Kiln [46] provides atomic durability in presence of a non-volatile cache (NVC). Having an NVC eliminates the requirement of logging by allowing NVC and NVM to hold two versions of a cache line where one of the versions can conceptually be considered as a log. Memory controller optimizations [53, 52, 54] have been proposed to improve the performance by differentiating between log writes and data writes. These proposals are broadly aimed at reducing latency of persist operations and are complementary to our proposal of removing log writes from the critical path.

Pelley et al. propose the concept of memory persistency models in terms of persist ordering constraints [7]. In [47] the authors analyze dependencies that need to be satisfied to implement transactions under various persistency models and propose optimizations to improve performance. These proposals broadly deal with reducing dependencies across transactions and are complimentary to our approach of reducing dependencies within a transaction.

Recently, redo logging for atomic durability was proposed in [15]. After completing an atomic update, the backend controller reads the log entries from the log area in memory and updates data in-place. Reading log entries after each update places additional pressure on the memory read bandwidth and can significantly delay the critical read requests coming from the processor. Another drawback is that it can lead to multiple log entries for the same data if the data gets modified multiple times during an atomic update (§4.7.2.4). They also need a victim cache to avoid spilling dirty cache lines into memory.

NVM cannot be used as a drop-in replacement for disks without modifying the surrounding software stack [55, 63, 67]. In systems with NVM, the synchronization overheads of a centralized log are high and hence there have been proposals for using per-thread distributed logs [62, 63]. In ATOM, however, the log space is centralized and shared across all threads to reduce fragmentation and improve utilization. We overcome the synchronization overhead by partitioning the log space into buckets and managing log space at bucket granularity in hardware.

ATOM provides atomic durability and relies on software locks to provide isolation [10]. But it can be adapted to leverage other ways to provide isolation such as hardware transactional memory (including but not limited to Intel's Transactional Synchronization Extentions [21] and [38]).

Techniques like WSP [58] have been proposed, which save the entire execution state in persistent memory on a power failure. They rely on a small battery backup to flush caches and store processor state. Although this technique works in case of power failure, it is not clear as to how it can be used in case of other failures such as software crashes. In contrast, BSP guarantees persistence and recovery for any kind of failure. TSP [68], on the other hand, discusses tradeoffs in fault tolerance mechanisms, depending on the failure model (software crashes, power failures etc.).

4.10 Summary

Many classes of applications, like databases, reason about crash consistency by using primitives like atomic durability. We highlighted that the common approach to provide support for atomic durability via software logging is inefficient in persistent memory systems. Specifically, we showed that software logging adds log writes to the critical path of programs and can severely degrade performance. We proposed ATOM, a hardware log manager for undo logging, based on the observation that logging is primarily a data movement task. To move log write operations out of the critical path, we proposed a posted log optimization, where the log writes are posted to the memory controller. Additionally, we presented a source log optimization to reduce redundant data movement by allowing the memory controller to perform logging in certain scenarios. Our evaluations showed that ATOM improves performance by 27% to 33% for micro-benchmarks and by 60% for large-scale transactional workload (TPC-C) over a baseline undo log design.

We also presented a hardware checkpointing mechanism by implementing buffered strict persistency in bulk mode. Specifically, we showed that the efficient lazy barrier (LB++) can be used to create checkpoints which can made atomically durable by leveraging ATOM. Our evaluations using a subset of PARSEC, SPLASH and STAMP benchmarks showed that applications can be checkpointed with only a 30% execution time overhead compared to a non-checkpointed execution.

Chapter 5

DHTM: Durable Hardware Transactional Memory

5.1 Introduction

Previous chapters presented the design of an ordering and an atomic durability primitive. This chapter presents the design of a primitive to support ACID transactions, where updates within a transaction are made visible (to other transactions) as well as durable (to non-volatile medium), in an atomic manner. While the database community has developed a plethora of techniques to guarantee ACID efficiently, these techniques have predominantly been developed with slow block based media in mind. When applied to in-memory settings, such techniques tend to spend a significant amount of time on concurrency control [69, 70, 71] and logging [63, 69, 72]. This leads us to ask the question: How fast can we enforce ACID in the presence of fast persistent memory?

Our Approach. Our primary goal is to design an HTM that can support ACID transactions efficiently. A secondary goal is to extend the supported transaction size by supporting overflows from the L1 cache to the last level cache (LLC) without adding significant complexity to the coherence protocol or the LLC.

One way of achieving these goals is to leverage existing unbounded HTM designs [38, 39, 40] that rely on logging to support overflows and make those logs durable [73]. However, such an approach, where durability is treated as a secondary consideration, will have poor performance as persisting the log and/or the data will be in the critical path.

We advocate an alternative approach in which durability is a first class design constraint. We propose Durable Hardware Transactional Memory (DHTM) in which we integrate a commercial HTM like RTM [21] with hardware support for redo logging. DHTM achieves atomic visibility by leveraging RTM. Whereas for achieving durability, DHTM provides architectural support for transparently and efficiently writing redo log entries to a durable transaction log maintained in persistent memory; the key efficiency enabler here is our novel mechanism for collating and flushing log entries without consuming excessive memory bandwidth. The redo log based design allows us to commit a transaction as soon as all the log entries have been written to persistent memory, without waiting for data to be made durable. DHTM then extends the supported transaction size, by leveraging the same logging infrastructure for also supporting L1 overflows. When the write set of a transaction overflows from the L1 cache, DHTM logs the address of the overflowed cache line and leverages the log to commit (or abort) the transaction. DHTM supports this with minor changes to the coherence protocol and without adding any additional transaction tracking hardware to the LLC. In summary, our key contributions are:

- We propose DHTM, the first complete hardware solution for an ACID compliant transactional memory system which is not bound by the size of the L1 cache.
- We enforce ACID efficiently by leveraging RTM [21] for atomic visibility and by providing atomic durability via hardware support for redo logging. We also propose a mechanism for coalescing log entries to reduce the required memory bandwidth.
- We extend the supported transaction size by allowing for the transaction's write set to overflow from the L1 to the LLC by leveraging the same logging infrastructure for handling these overflows. We accomplish this with only minor changes to coherence protocol.
- Our evaluation shows that DHTM outperforms the state-of-the-art [73] by 21% to 25% on average across TATP, TPC-C and a set of micro-benchmarks.

5.2 Related Work

Recently, there have been multiple proposals for providing ACID updates to persistent memory. These proposals are classified in Table 5.1 based on how they enforce atomic visibility and atomic durability. The first class of designs [10, 11, 13, 16] support atomic durability via software logging by employing flushing and ordering instructions. Ensuring atomic durability in software, however, comes at a significant performance cost [15, 51, 53, 48, 74] which motivated the development of the second class of designs that either employ hardware support for atomic durabil-

Designs	Atomic Visibility	Atomic Durability	Transaction Size	LLC Extenstions
Atlas [10], REWIND [11], Mnemosyne [13], DudeTM [16]	Locks or STM	Software	Not limited	None
NVHeaps [12]*, WrAP [15], LOC [51], Kiln [46], [60], DPO [48]*, DCT [47]*, ATOM [73], HOPS [74]*, [75], [76], [77]	Locks or STM	Hardware	Not limited	[12], [15], [51], [46], [60], [75]
DudeTM [16], PHyTM [17], cc-HTM [18], [78]	HTM	Software	L1 limited	None
PTM [24]	HTM	Hardware	L1 limited	Yes
DHTM	HTM	Hardware	LLC Limited	None

Table 5.1: Classification of techniques supporting ACID updates on persistent memory. (* Leverage hardware support for ordering to provide atomic durability.)

ity [15, 51, 46, 73, 75, 76] or leverage hardware support for ordering to guarantee atomic durability [12, 48, 47, 74]. However, both of these classes enforce atomic visibility in software using software transactional memory (STM) or locks.

Another approach to ACID is to leverage commercially available Hardware Transactional Memory (HTM), which is the focus of the remaining classes of designs. However, current commercially available HTM systems have two limitations. First, they efficiently support only small transactions [19, 20, 21, 22, 23]; if a cache line written within a transaction is evicted from the L1, the transaction must abort. The severity of the problem has been highlighted by a recent study which finds that transactions whose write-set size is larger than 128 cache lines (quarter of L1 size) is highly likely to abort [79]. This L1 limitation can significantly limit usability and efficiency for ACID transactions, which tend to have relatively large write working-set sizes (Section 5.5).

Second, HTM systems only provide ACI guarantees, i.e., atomic visibility but not atomic durability. To guarantee ACID, the third class of designs [16, 17, 18] leverages the HTM for atomic visibility and integrates it with software support for atomic durability. The latter requires the writing of a log entry for every modified object within the transaction, thereby increasing the transaction write-set (and the abort rate). The fourth class supports ACID by integrating HTM with hardware support for durability. However, PTM [24] (the only proposal in this class) not only introduces significant changes to the cache hierarchy, but also continues to suffer from the L1 limitation.

5.3 DHTM Design

In this section we present the design of our *durable HTM* (DHTM), that adds support for durability on top of a commercial RTM-like HTM design.

System Model. For the following discussion, we assume a multicore processor with a two level cache hierarchy consisting of private L1 caches and a shared last level cache (LLC). The private L1s are kept coherent using a MESI directory based coherence protocol with forwarding (similar to the one in section 8.2 in [33]). We assume the directory is held in the LLC with each cache line maintaining the coherence state and sharing vector. We assume a baseline HTM similar to Intel's RTM. We assume that the HTM supports strong isolation. Finally, we assume that memory is non-volatile and byte addressable. It is worth noting that the above model is mainly to help anchor our description and as such, none of these choices are fundamental to DHTM.

Overview. At a conceptual level, adding durability to an HTM requires some form of logging. Figure 5.1(a) shows a volatile transaction at the top and the corresponding read and write sets at the bottom. The transaction reads X (read set) and writes to X and Y (write set). One way for this transaction to be made durable is by executing the code sequence shown on the top in Figure 5.1(b), which additionally writes log entries for the data being modified. The resultant read set of the persistent transaction remains the same, but the write set consists of *Log* X, X, *Log* Y and Y. Thus, adding support for durability essentially doubles the write-set size of transactions. This is a challenge on current RTM-like HTM designs which already limit the write-set size. To compound matters, applications that demand ACID tend to have relatively large transaction sizes. Therefore, one of our goals is to support transactions with a larger write-set size rel-

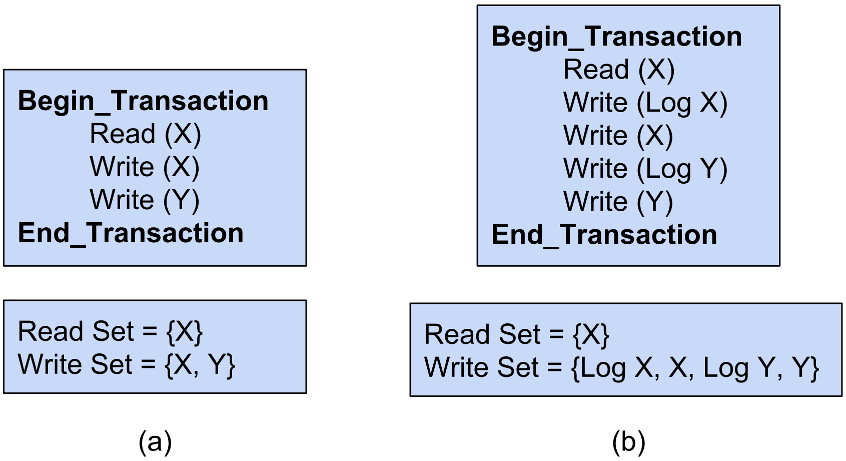


Figure 5.1: Working set sizes for transactions (a) without including durability log and (b) with durability log.

ative to those supported by current commercial HTMs. But in the quest for larger transactions, we do not want to introduce significant hardware complexity; in particular, we do not want to introduce changes to the shared LLC like adding transaction tracking hardware or searching the LLC for cache lines belonging to the write set – something that current HTM designs avoid.

Our approach is to integrate hardware based redo logging to an RTM-like HTM. For atomic visibility DHTM leverages the RTM-like HTM and for atomic durability it employs hardware redo logging. Since logging is performed transparently, DHTM's programming interface is similar to that of volatile transactions (Figure 5.1(a)). DHTM's redo logging mechanism leverages the L1 cache write-back interface to dynamically write redo log entries to persistent memory for cache lines being modified within a transaction. Furthermore, DHTM allows dirty cache lines to overflow from the L1 cache into the LLC without causing an abort. This increases the transaction size with minor changes to the coherence protocol and without adding significant design complexity (in particular, without adding transaction tracking hardware to the LLC). Below, we first describe DHTM's hardware logging mechanism. Then, we describe how logging integrates with the HTM, followed by the description on how DHTM manages overflow.

5.3.1 Logging for Durability

We ensure atomic durability using *write-ahead logging*. The idea is to maintain a persistent copy of the old and new versions at all times during the transaction, so that the state can be recovered to either of the versions. This persistent copy is maintained in the form of *log entries* which consist of the address and the old or new version of data. In this section we provide a design for a redo-log based implementation to work in conjunction with HTM.

Why Redo-Logging? We choose a redo-log based design as it allows us to have both fast commits as well as fast aborts. In volatile transactions, undo-logging supports faster commits because, on transaction completion all the in-place updates would have already taken place (in the cache); commit therefore only requires two simple steps: discarding the undo-log and flash-clearing the speculative write-bits to make the write-set visible to other threads. Durable transactions, however, impose additional constraints. Both the undo-log entries and the write-set (data) have to be written to persistent memory – only then, can the transaction be committed. While techniques have been proposed for minimizing the fine grained ordering overheads while writing log entries [73], flushing the write-set can significantly increase commit time.

Redo-logging, in contrast, requires only the redo-log entries to be written to persistent memory at commit time. This is because the redo-log, in addition to serving as a recovery log in case of a failure, can also provide the up-to-date values on commit. This allows for the data updates to be written to persistent memory in the background, and out of the commit critical path. One traditional drawback of redo-logging is that, because writes are not allowed to overwrite previous values, subsequent reads to those addresses need to be redirected to the redo log. Our proposed hardware based redo-logging mechanism overcomes this limitation by allowing writes to overwrite previous values in the cache. A subsequent read can therefore directly read the updated value from the cache. It is worth noting, however, that the writes do not overwrite the old values in memory but are written to a separate redo-log area. Lastly, aborts are also faster with redo-logging and only require two simple steps: discarding the redo-log and invalidating the modified lines in the cache.

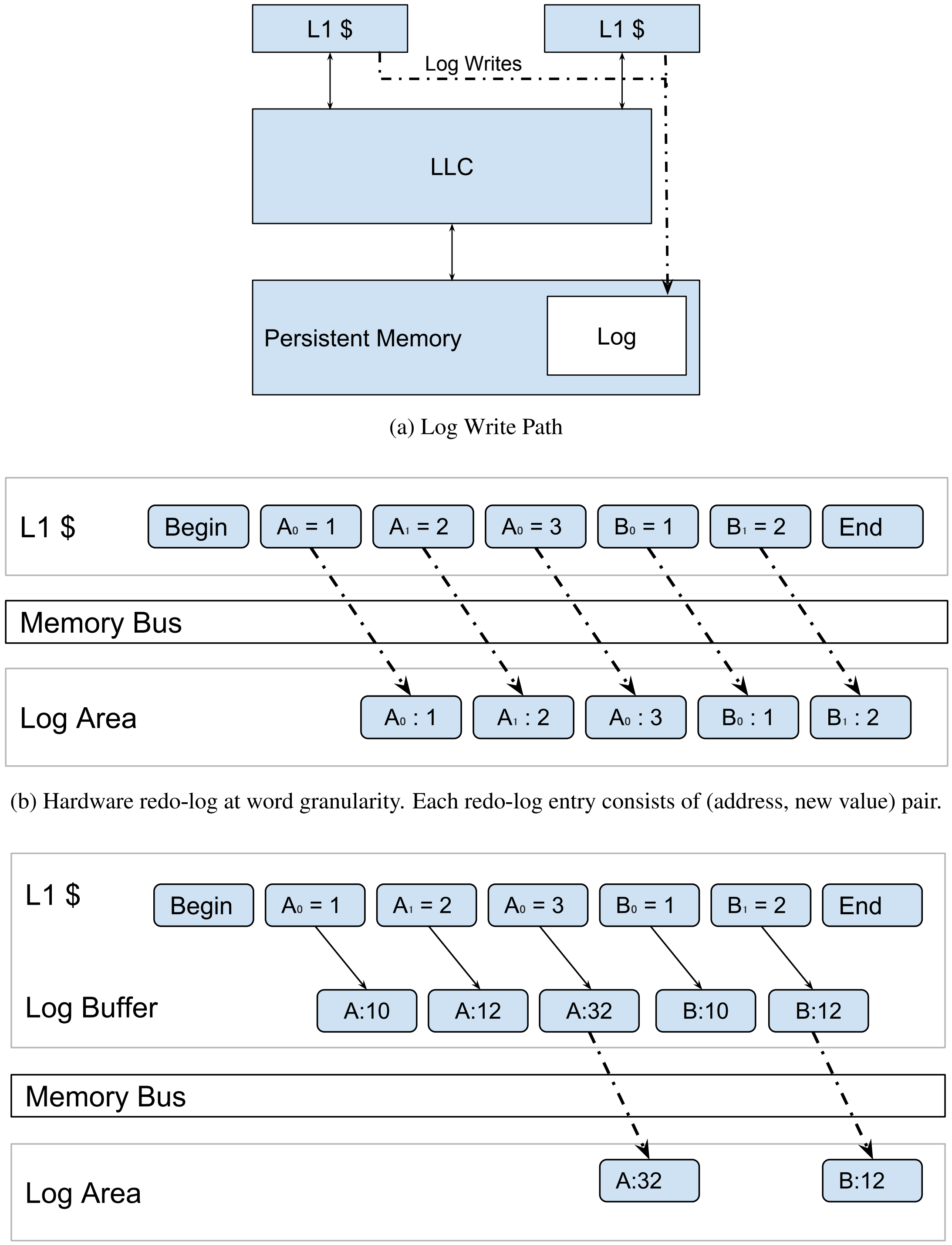
Log management. In the DHTM design, the transaction log space is thread private and is allocated by the operating system (OS) when the thread is spawned. The OS keeps track of all the logs it has allocated so that it can recover transactions from logs in case of a system crash. This per thread transaction log is organized as a circular log buffer similar to Mnemosyne [13]. On a log overflow, DHTM aborts the transaction with an indication that the abort is because of log overflow. The OS in this case allocates a larger log space for the thread and the transaction is retried.

Hardware Support. One of the design goals of DHTM is to write log entries to the transaction log in persistent memory without adding them to the write-set. To this end, logging is performed in hardware in DHTM, allowing DHTM to differentiate between log writes and data writes. The L1 cache controller is modified to enable it to write log entries to persistent memory by bypassing the LLC as shown in the Figure 5.2(a). The L1 cache controller creates these log entries on the fly at a word granularity for every store within a transaction. Figure 5.2(b) shows the log writes that the L1 cache controller performs.

Log coalescing. Writing a word-granular redo-log entry for every store can generate a large number of log entries which can consume significant amounts of memory write bandwidth. Figure 5.2(b) highlights this with an example. Let us assume that each cache line consists of two words (all words belonging to cache lines A and B are initially 0); the subscript for each cache line refers to the word in the cache line that is being modified. Performing word-granular logging generates 5 log writes across the memory bus for 5 store requests to different words in cache lines A and B. The bandwidth consumed can be mitigated to some extent by coalescing multiple log entries into one cache line before writing them to memory. Nonetheless, creating a log entry for every store request is problematic. Recall that each log entry is composed of the data and the address (metadata). The finer the granularity of logging, the greater the amount of metadata, which in turn translates into higher bandwidth consumption. Second, logging for every store request might miss opportunities for coalescing multiple stores to the same word via a single log entry. For example, in Figure 5.2(b) the word A_0 gets written to twice which leads to the creation of 2 log entries, however only the second log entry would have sufficed.

An alternative is to perform logging at cache line granularity. But naively creating a log entry for every store request will only worsen the memory bandwidth consumption. At the same time, the final state of a cache line (at the end of a transaction) must be logged for correctness. If we can predict the final store to a cache line, that would be an opportune moment to log that cache line, since that would minimize the number of entries logged for that cache line. It is important to note that the prediction must be conservative, in that, it must not miss the last store under any circumstance.

We conservatively predict the final store to a cache line via a simple structure called log buffer that is added to the L1. The log buffer is a fully associative structure with a small number of entries that keeps track of cache lines with their cache line addresses (our design has 64 entries). When a store is performed, the corresponding cache line



(a) Log Write Path

(b) Hardware redo-log at word granularity. Each redo-log entry consists of (address, new value) pair.

(c) Hardware redo log at cache line granularity using a log buffer.

Figure 5.2: Redo logging in hardware.

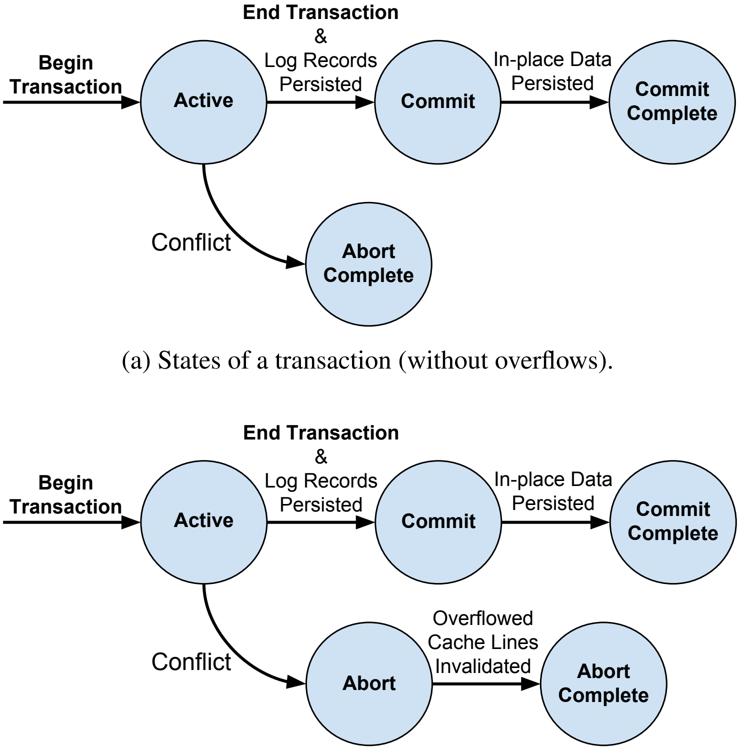
address is added to the log buffer (if not already present). A log entry is written to persistent memory only when an entry is evicted from the log buffer. An entry is evicted from the log buffer under two situations: (a) when the log buffer is full, an eviction has to happen in order to make space for a new cache line address; (b) when an L1 cache line is replaced and the log buffer holds the corresponding address, the address is evicted from the log buffer. Thus, we use eviction from the log buffer as a proxy for predicting the last store to a cache line; in practice, this simple policy works well because write reuse distance (when there is reuse) is typically low for transactional workloads. When an entry is evicted from the log buffer, the redo-log entry for that cache line is created as usual by composing the address with the contents of that cache line from the L1 cache. Then, the redo-log entry is written to persistent memory – in doing so, the stores to one cache line are temporally coalesced, such that all these coalesced stores get only one log write. Finally, at the end of the transaction, all of the cache lines being tracked in the log buffer are logged to persistent memory. It is important to note that this log buffer is different from the log buffer used in LogTM [38]. LogTM uses a buffer to reduce the contention for the L1 cache port and to hide L1 cache miss latency whereas the buffer in DHTM is to coalesce log writes to the same cache line and to predict the last write to a cache line.

Figure 5.2(c) shows the previous example in the presence of a single entry log buffer. Initially the buffer holds cache line *A* while it is being modified. When cache line *B* has to be written to, the updated value of cache line *A* is written to the log area in memory and the buffer now holds cache line *B*. Eventually when the transaction ends, a redo log entry for *B* is also written to the log area. In this simple example, 5 store requests now need only 2 log writes over the memory bus.

5.3.2 Integrating Logging with HTM

In this section, we will describe how to integrate our logging mechanism with an RTM-like HTM. This section will assume that the transaction will abort on a write-set overflow from the L1; we will handle write-set overflows in the next section.

Overview. Committing a volatile transaction requires that the read/write-set tracking structures be cleared and that the speculative state be made visible to other threads. In addition to the above steps, in order to commit a durable transaction (with redo-logging), the redo-log entries must be written to persistent memory. Recall that the data updates (write-set) can be written to persistent memory lazily and out of the commit critical path. Conflict detection works identically to a volatile transaction. Non-transactional accesses also safely integrate with DHTM, similar to RTM, by aborting an ongoing transaction if it conflicts with a non-transactional access. Aborts also are



(a) States of a transaction (without overflows).

(b) States of a transaction (with overflows).

Figure 5.3: Transaction States. A core can start executing subsequent non-transactional instructions after reaching *Commit*/*Abort* and can start a new transaction after reaching *Commit Complete*/*Abort Complete*.

largely identical, with an added step of (logically) clearing the redo-log for the transaction. Thus, a durable transaction can be expressed in the form of a state diagram as shown in Figure 5.3(a), with the following states: *Active*, *Commit*, *Commit Complete* and *Abort Complete*. Below, we discuss these in more detail.

Commit. Upon reaching the end of the transaction, and having written all redo-log entries to persistent memory, the transaction effectively *commits*. To mark that the transaction has committed, DHTM writes a *commit log record* to the log area. The L1 cache controller then starts writing back the cache lines belonging to the write set of the committed transaction via the cache write-back interface. DHTM does not flash clear the write bit associated with cache lines on a commit, instead it clears those bits once a write-back is issued for the corresponding cache line. After writing back all the modified cache lines to persistent memory, DHTM marks the transaction as *completed* by writing a *complete log* record to the log area. Writing a complete log record is not a correctness requirement but reduces recovery time on a failure (as we shall see in the section on recovery). Once a transaction has committed, DHTM can start executing non-transactional code following the transaction. But since DHTM has only one set of write bits per cache line, it cannot start executing a new transaction until the previous transaction has completed. This is because, in order to complete a transaction, DHTM relies on these write bits to identify the modified cache lines that need to be written back to persistent memory.

In the DHTM design, there is a window between the commit point of a transaction and its completion point (when the cache lines modified in the transaction are being written back to persistent memory and are being marked as non-speculative) during which a conflict might be detected incorrectly. For example, consider that a transaction T_A tries to modify a cache line X. But X has already been modified by a committed but not yet complete transaction T_B, and has not yet been marked as non-speculative. In such a scenario a conflict will be detected incorrectly. DHTM sidesteps this problem by also consulting the state of the transaction during conflict detection; as the transaction status of T_B indicates that it has committed, DHTM does not raise a conflict in this situation. Additionally, DHTM inserts a sentinel log entry in the transaction log of both T_A and T_B indicating that transaction T_A is dependent on the updates of transaction T_B. This sentinel log entry enables the recovery manager to decide the correct order of replay for transactions with conflicting updates.

Abort. In DHTM a transaction can be aborted for various reasons including, write-set overflows, context switches etc. However, irrespective of the reason, the abort procedure remains the same. Aborting a volatile transaction requires that the read/write-set tracking structures be cleared and that the speculative state be invalidated. To abort a durable transaction, in addition to the above steps, the log entries need to be cleared. DHTM logically clears the log entries by writing an abort log record, effectively marking the log entries as being part of an aborted transaction.

Recovery. At the time of failure, a durable transaction can be in one of the following states: *Active*, *Commit*, *Commit Complete*, or *Abort Complete*. The recovery manager does not have to do anything for transactions in Active or Abort Complete state as none of the updates of the transactions would have been written back in-place in persistent memory. In other words, persistent memory has the pre-transaction state for those transactions. For committed but not completed transactions (transactions in *Commit* state), the recovery manager reads the log entries and writes the updated values in-place in persistent memory, thus recovering the updates of the transaction. Finally, for completed transactions (in *Commit Complete* state) the recovery manager does not have to do anything, since all of their updates would have already been written back in-place in persistent memory. In the absence of a complete log record, the recovery manager would have had to copy all the updates from the log area to in-place in persistent memory. Thus the writing a complete log record helps reduce the recovery time.

The replay order of committed but not complete transactions does not matter as long as they do not have conflicting updates. For transactions with conflicting updates, the recovery manager infers the required replay order by looking at the sentinel log entries in the relevant transaction logs.

The recovery manager is implemented as an operating system service which is invoked upon system re-start. As described earlier, the OS keeps track of all the logs it has allocated which it also registers with the recovery manager on creation and deregisters when the the log is deallocated. When the recovery manager is invoked, it scans all the registered logs and restores all the committed but not completed transactions.

5.3.3 Handling Overflow

The design described above continues to suffer from the transaction size limitation that is typical of an RTM design. We now describe an extension that allows the write-set of a transaction to overflow the L1 cache without aborting the transaction. Consistent with current commercial HTM designs, our proposed extension also does not require expensive operations at the LLC (e.g., searching for cache lines belonging to a particular transaction), making it amenable to commercial adoption. We first summarize the challenges in supporting overflow efficiently and then describe our approach.

Challenges. Commercial HTM designs like RTM allow the read-set to overflow the L1 cache. Conflicts are detected with the help of the overflow signature maintained in the L1 cache, which tracks the addresses of cache lines that have overflowed. On a transaction commit or an abort, the overflow signature is cleared; importantly, nothing needs to be updated in the LLC. In contrast, RTM-like designs do not support write-set overflows from L1. This is because, aborting a transaction requires that the HTM invalidate all the cache lines belonging to the write-set. Whereas this can be done in private L1 cache by flash invalidating the cache lines, doing this for a shared structure as large as the LLC is expensive and involves non-trivial changes (indexing and searching operations). With durable transactions, a commit would also require a similar operation at the LLC: all the cache lines that have overflowed must be identified and written back to persistent memory.

Overview. Our DHTM design allows for the write set to overflow from the L1, with minor changes to the coherence protocol and without requiring any structural changes to the LLC. Our key idea is to leverage the redo log (which holds the speculative state of the transaction) for handling write-set overflows, thus obviating the need for expensive changes to the shared LLC.

DHTM handles write-set overflow by allowing for cache lines belonging to the write-set to be replaced from L1 to LLC; [1] in order to enable conflict detection the coherence state of the cache line in LLC is kept unchanged, however. This ensures that the LLC continues to show the cache line as being owned by the core executing the transaction. Therefore any coherence message will continue to be forwarded to the owner's L1, wherein a potential conflict can be detected. It is worth noting that our idea of using stale coherence state for conflict detection is similar to the *sticky state* solution used in LogTM [38]. Therefore, the resulting coherence protocol extensions in DHTM are similar to the sticky state extensions of LogTM.

While maintaining stale state in the LLC helps in conflict detection, we also need a mechanism to identify all the cache lines that have overflowed from L1 to LLC for versioning. To this end, DHTM maintains an *overflow list* along with the redo log in memory. When a dirty cache line overflows from L1 to LLC, DHTM writes the address of the overflowed cache line to the overflow list. On a commit or an abort, DHTM uses the overflow list to identify the write-set cache lines that have overflowed, and writes them back to persistent memory (in case of a commit), or invalidates the corresponding LLC cache line (in case of an abort).

The recovery procedure remains the same with overflows, since the cache lines belonging to the write-set that overflowed the L1 cache are already present in the redo log. In summary, a durable transaction with write-set overflows can be expressed in the form of a state diagram as shown in Figure. 5.3(b) with the following states: *Active*, *Commit*, *Commit Complete*, *Abort* and *Abort Complete*. One key difference with overflows is that like commits, aborts now require a completion phase. Below, we discuss this in more detail.

Commit. After writing the commit log record to the log area and issuing write backs

[1] If LLC cache line is already dirty, the old LLC dirty block is first written back to memory.

for all the write-set cache lines in the L1 cache, DHTM reads the overflow list corresponding to the committing transaction (recall that the overflow list contains addresses of all the dirty cache lines that have overflowed from the L1 cache). It then sends write back messages for those cache lines to the LLC. On receiving a write back request, the LLC writes back the relevant cache line in-place in persistent memory and also transitions the cache line to a clean state and clears its sharer vector. After writing back all the cache lines in place in persistent memory, DHTM writes a *complete* log record to the transaction log and then transitions the transaction status to *Commit Complete*. This completion operation ensures that the LLC correctly reflects the status of overflowed write-set cache lines belonging to the transaction and eliminates any need for LLC modifications to clear such state.

Conflict Detection. Recall that conflicts are detected at the L1 controller, by checking coherence requests against the read/write-bits associated with cache lines or the read overflow signature. In order to enable conflict detection in the presence of write-set overflows, we need to ensure that coherence requests for the overflowing cache lines continue to reach L1. To this end, when a dirty block overflows from the L1, the coherence state of the LLC is kept unchanged. Specifically, when an L1 receives a Fwd-GetM request or a Fwd-GetS request for a cache line that is not present in L1, DHTM infers that the request corresponds to a cache line that has overflowed from the L1. Therefore, a conflict is detected and the transaction issuing the request is aborted (because of first writer wins policy).

Abort. To abort a transaction, DHTM first invalidates the cache lines belonging to the write-set in L1 as described in Section 5.3.2. But additionally, the cache lines in LLC that overflowed from the L1 will also need to be invalidated. Therefore, in the presence of overflows, abort also has a completion phase as shown in Figure 5.3(b).

In the completion phase, DHTM reads the cache line addresses from the overflow list in the transaction log and issues invalidate requests for those cache lines to the LLC. The LLC invalidates the cache lines on receiving an invalidate message. Similar to the completion phase of commit (Section 5.3.2), the completion phase for abort can continue in parallel with the execution of other non-transactional instructions, but a subsequent transaction cannot begin until this phase completes. However, differently from a commit, DHTM does not (need to) write an abort complete log record for an aborted transaction as the state that needs to be invalidated is in volatile caches and will anyway be lost on a system crash.

One corner case concerns cache lines that have been reread back into the L1 during the transaction, after overflowing from the L1 to the LLC. When such a transaction aborts, these reread cache lines must be identified as belonging to the write-set and invalidated by the L1. Such reread cache lines are correctly identified by DHTM as follows. When being reread, DHTM will look at the state of the cache line and the sharer vector; if the cache line is dirty and its state is in *modified* state with the requester marked as its owner, DHTM will identify the cache line as belonging to the write-set and will set the write-bit in the L1. This ensures that such reread cache lines are invalidated on an abort.

It is worth noting that, as opposed to existing proposals for supporting overflow (such as LogTM) DHTM does not stall requests from other transactions. Consider the case where transaction T_A has modified cache line X which then overflowed to the LLC. T_A is subsequently aborted because of a conflict. While T_A is in the process of aborting, another transaction T_B issues a read for cache line X. Because of the eager version management of LogTM, the read for X cannot be completed until X has been reverted to a non-speculative state from the undo log. Therefore, LogTM would NACK the read request for X while waiting for the abort process of T_A to end and T_B will have to subsequently re-issue the read. This adds significant complexity to the coherence protocol. DHTM on the other hand has non-speculative data in memory because it maintains a redo log for atomic durability. Therefore, it can immediately complete the read for X by fetching it from memory. In summary, DHTM maintains the simplicity of an RTM like design while allowing for overflows from the L1 to the LLC.

5.4 Putting it Together

In this section we first explain through detailed examples, the life cycle of a transaction. We also quantify the overall hardware overhead and finally describe a software fallback mechanism for transactions that do not fit in DHTM.

Transaction Lifecycle. Figures 5.4 and 5.5 collectively show the life cycle of a transaction. Figure 5.4 shows the initial setup phase (steps **(a)** through **(d)**) of a transaction and Figure 5.5 shows commit (**(e)** and **(f)**) and abort (**(g)** and **(h)** steps. For this example, let us assume a dual-core processor. Each figure shows three views: (i) L1 view from the perspective of core 1, showing L1 cache lines with their read/write-bits, a single entry log buffer, the read overflow signature and transaction status register; (ii) LLC view, showing for each cache line, its coherence state, sharer vector and dirty bit; and (iii) Persistent memory view, showing the overflow list, log area and in-place values of cache lines in persistent memory.

(a) Initial state. The transaction is in *Active* state and has already modified cache line *A* with a value of 15 and has already read cache line *B*. No transactional data has overflowed from the L1. The LLC has cache lines *A* and *B* owned by the core 1 in *modified* state.

(b) Write B. The transaction modifies the value of cache line *B* to 25. Therefore, DHTM sets the write-bit for cache line *B*. Also, cache line *A* needs to be evicted from log buffer (to make space for cache line *B*), so its updated value of 15 is written to the redo log area.

(c) Read C. The transaction reads cache line *C*, because of which cache line *A* gets replaced from the L1 cache. Therefore, cache line *A* is written back to LLC and its dirty bit is set, but the coherence state of cache line *A* is not changed. Also, the address of cache line *A* is written to the overflow list in memory. Cache line *C* is present in the LLC in shared state, its sharer vector is updated to add core 1 as a sharer and the cache line is brought to the L1 with its read-bit set.

(d) Write E. The transaction writes 55 to cache line *E*. This leads to cache line *C* being replaced from L1. Before being replaced, the address of cache line *C* is added to the read overflow signature (but because of inherent imprecision, let us assume that the signature conservatively shows both *C* and *D* as its members). Since cache line *E* is not present in the cache hierarchy, it is brought to the LLC in *modified* state with the sharer vector showing core 1 as the owner. The cache line is also added to L1 cache where it is updated and its write-bit is set. Since cache line *E* also needs to be added to the log buffer, cache line B is removed from the buffer and its updated value is written to the transaction log.

(e) Commit. When the transaction commits, cache line *E* is written from L1 to the transaction log and a commit log record is also written to the log. Simultaneously, the read-bits and the read overflow signature in the L1 are cleared and the transaction state is updated to *Commit*. The transaction commits at this point and core 1 may continue executing non-transactional instructions.

(f) Commit Complete. In the commit completion stage, the L1 writes back cache lines *B* and *E* to LLC and clears their respective write-bits. On receiving the write backs, the LLC updates them and also writes them back to persistent memory. Then the memory controller reads the overflow list and issues a write back request to LLC for cache line *A*. LLC on receiving the request, clears the sharer vector and dirty bit for cache line

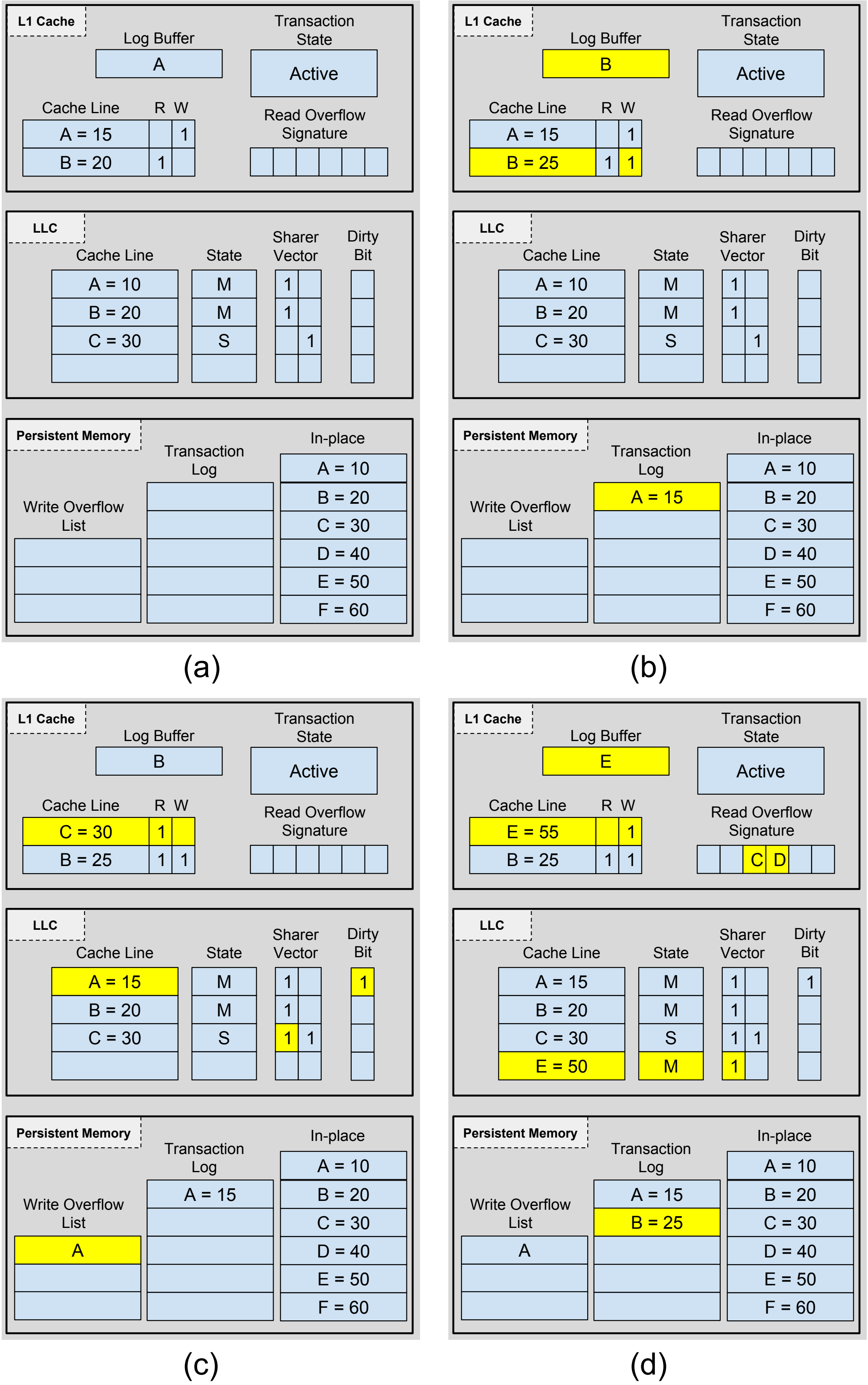


Figure 5.4: Flow of a transaction - Part 1.

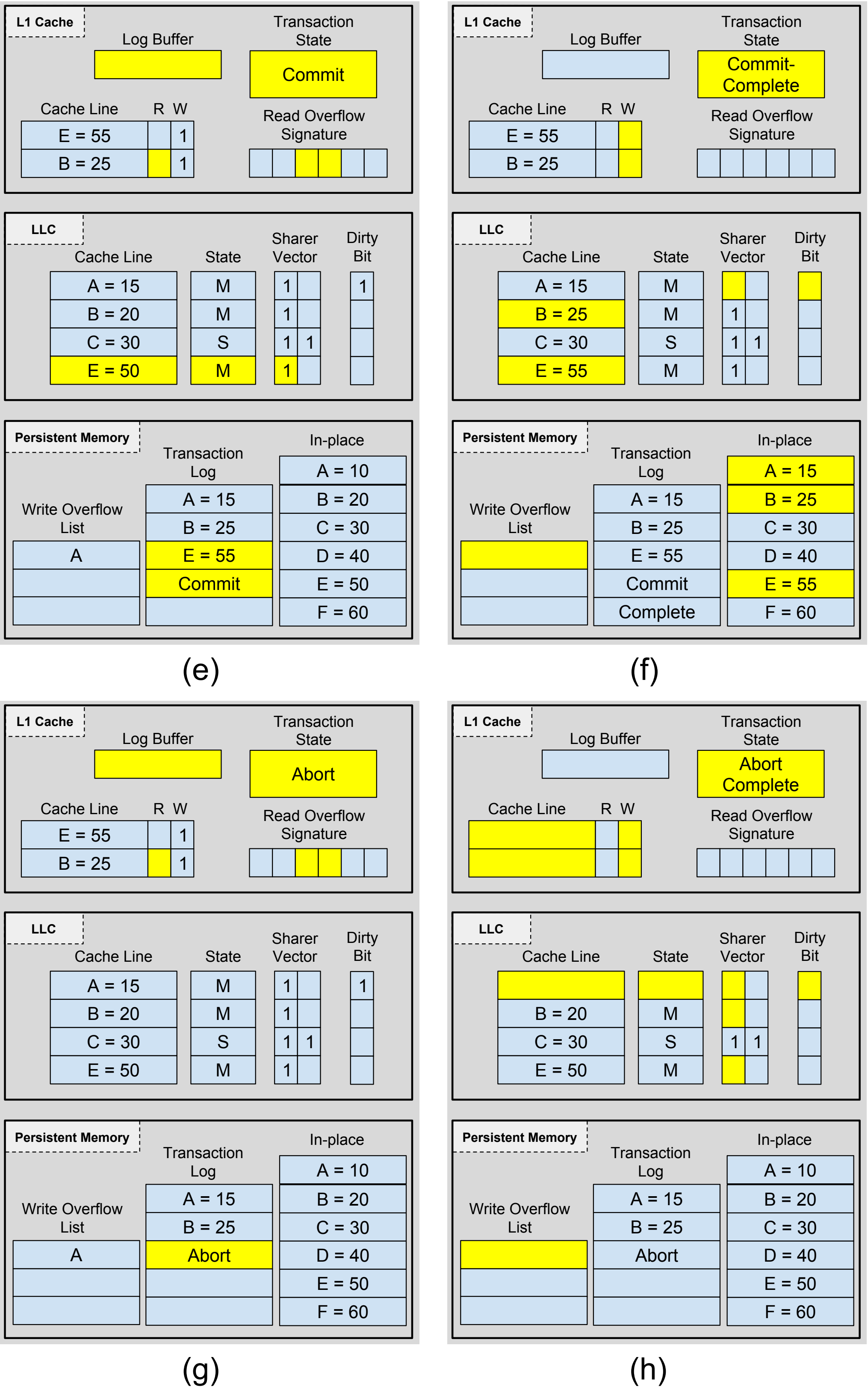


Figure 5.5: Flow of a transaction - Part 2.

Register	**Description**
Log Buffer	Tracks cache lines pending log writes
Transaction State	Identify the state of a transaction
Log Area	
Start Pointer	The start address of the log space
Next Pointer	Address to write the next log entry
Size	Size of the log space
Overflow List	
Start Pointer	The start address of the overflow list
Next Pointer	Address to write the next entry
Size	Size of the overflow list

Table 5.2: Hardware Overhead.

A and writes it back in-place in persistent memory. Finally, a complete log record is written to the log area, the overflow list is cleared and the transaction state is updated to *Commit Complete*. At this point the transaction has completed and core 1 may begin a new transaction.

(g) Abort. Shows the state of the system if the transaction were to abort after (d). An abort log record is written to the log area, the read-bits and the read overflow signature in the L1 are cleared, and the transaction status is updated to *Abort*. The transaction has aborted at this point and core 1 may continue executing subsequent non-transactional instructions.

(h) Abort Complete. In the abort completion phase, the L1 invalidates cache lines *B* and *E* belonging to the write-set and sends invalidate messages to LLC which then clears the sharer vector for cache lines *B* and *E*. Then the memory controller reads the overflow list and issues invalidate message for cache line *A* to LLC. On receiving the invalidate message, the LLC invalidates cache line *A* and clears its sharer vector and dirty bit. Finally the transaction status is updated to *Abort Complete* and at this point, core 1 may begin a new transaction.

Hardware Overhead. Table 5.2 shows the hardware overhead that DHTM adds on top of an RTM-like HTM design. DHTM adds to the L1 cache a full-associate structure called the *log-buffer*, for keeping track of cache lines for which redo log entries need to be written to persistent memory. It also adds a *transaction state* register to identify the current state of the transaction. DHTM also adds two sets of registers to keep track of the log area and the overflow list. The registers in each set consist of a *start pointer* to identify the start address of the corresponding area, a *next pointer* to identify the address where the next entry can be written to and finally a *size* register to keep track of the size of each area so that an overflow can be detected.

Fallback Path. DHTM increases the limit for the transaction size from L1 cache to LLC. However, if a transaction aborts continually because of a overflow from LLC (or due to any other reason) then it might not be able to make forward progress. Therefore, a fallback path must be provided. In principle integrating a software fallback path to DHTM is no different from the ones proposed for RTM [80] because both employ a similar mechanism for atomic visibility. In particular, this software fallback path does not interact with hardware logging because, before initiating the fallback path DHTM would abort the transaction (taking it to abort complete state) which clears the log. The only difference is that our fallback will provide both atomic visibility and durability similar to Mnemosyne [13].

5.5 Experimental Setup

We now describe our simulation infrastructure, system configuration, benchmarks and designs that we evaluate. We implemented DHTM on the gem5 [42] simulator with Ruby. We extend the Ruby memory model to implement DHTM functionality, with a log buffer size of 64 entries. We evaluate DHTM on an 8-core multicore (one thread per core) with each core containing a 32 KB private L1 and a multi-banked LLC. The local L1s are kept coherent using a MESI based directory protocol. DHTM is built on top of HTM that is based on an RTM-like implementation. Conflicts are detected by piggybacking on top of the coherence protocol and the HTM employs a first-writer wins conflict resolution policy similar to IBM POWER8 [22]. However, it is important to note that the choice of conflict resolution policy is not fundamental to DHTM design and it can be implemented with other policies (like requester wins) as well. Table 5.3 shows the main parameters of our system. The peak memory bandwidth in our setup is 5.3 GB/s.

Workloads and their Characteristics. We considered two classes of workloads for

Cores	8 In-order cores @ 2GHz
L1 I/D Cache	32KB 64B lines, 4−way
L1 Access Latency	3 cycles
L2 Cache	1MB×8 tiles, 64B lines, 16−way
L2 Access Latency	30 cycles
MSHRs	32
NVM Access Latency	360 (240) cycles write (read)

Table 5.3: System Parameters.

Workload	**Description**	**Write Set**
TPC-C	Online transaction processing	590
TATP	Mobile carrier database	167
Queue	Insert/delete entries in a queue	52
Hash	Insert/delete entries in a hash table	58
SDG	Insert/delete edges in a scalable graph	56
SPS	Random swaps between entries in an array	63
BTree	Insert/delete nodes in a b-tree	61
RBTree	Insert/delete nodes in a red-black tree	53

Table 5.4: Benchmarks used in our experiments along with their descriptions and write set sizes (# cache lines).

our study. TPC-C and TATP (the first two rows from Table. 5.4) are traditional online transaction processing (OLTP) workloads that require ACID guarantees. We use in-memory implementations of these workloads [11]. It is worth noting that the OLTP workloads have write working-set sizes exceeding or comparable to the size of the L1 cache. Indeed, the write-set size of TPC-C (37 KB) exceeds the L1 cache size (32 KB) and can cause both capacity and conflict L1 misses which in turn can cause aborts when run on an HTM. Although TATP has a write-set size of around 10 KB, because of its access patterns, we find that there are significant conflict misses, which can lead to aborts.

The second class of workloads (the last six rows from Table. 5.4) are micro-benchmarks that perform atomic search, insert and delete operations on the corresponding data structure. The micro-benchmarks are similar to those in the benchmark suite used by NVHeaps [12]. We evaluate each of these micro-benchmarks with a data set size of 3 KB, similar to ATOM [73]. It is worth noting that the write-set sizes of the micro-benchmarks are significantly smaller than the L1 size, in contrast to the OLTP workloads.

Evaluated Designs:

– **SO**: This *software only* design uses locks for atomic visibility and software logging for atomic durability. For the OLTP workloads, we use the default software concurrency control mechanism which uses fine-grained locking. For the micro-benchmarks, we partition the data-structure into coarse-grained partitions with a lock associated with each partition, to allow for concurrency across the different partitions. We use a software logging mechanism similar to Mnemosyne [13], wherein log entries are flushed synchronously as soon as their values are finalized (thus benefiting from coalescing as well).

– **sdTM**: This design (software durability + hardware transactional memory) is based on PHyTM [17], which uses HTM similar to Intel's RTM for atomic visibility. We disable the software concurrency control mechanism in the benchmarks and instead enclose each transaction within a hardware transaction. We use software logging similar to Mnemosyne for atomic durability.

– **ATOM**: This design uses locks (similar to the **SO** design) for atomic visibility. It uses the state-of-the-art hardware undo logging mechanism for atomic durability [73].

– **LogTM-ATOM**: This design uses LogTM [38] like HTM for atomic visibility and integrates it with ATOM [73] for atomic durability, thus employing an eager version management mechanism. It is worth noting that this represents a new design that has not been studied previously.

- **DHTM**: This is our proposed design which supports atomic visibility by using HTM similar to Intel's RTM and atomic durability by using hardware based redo logging. It also allows the write set to overflow from the L1 to the LLC.

5.6 Results

In this section, we quantitatively compare the performance of the evaluated designs on the micro-benchmarks. We then present studies to better understand where the DHTM gains are coming from. One important parameter that could affect the efficacy of DHTM is the size of the log-buffer. Therefore, we quantify its impact. We also evaluate the efficacy of the designs on TPC-C and TATP workloads. Finally, we analyze the overheads of persistence by comparing our design with a non-persistent design.

5.6.1 Transaction Throughput

Figure 5.6 shows the transaction throughput of all the evaluated designs normalized to the software-only (SO) design, on the micro-benchmarks. As we can see, sdTM provides an average throughput improvement of 20% over SO. Recall that sdTM uses HTM for concurrency control which can potentially uncover more concurrency, especially in workloads where locking is coarse-grained. On the other hand, sdTM can suffer the negative effects of rollbacks in situation where the HTM aborts frequently. Table 5.5 shows the abort rates for the workloads for the sdTM design. In general, we can observe a correlation between the abort rates experienced by various workloads and the throughput improvement over SO. In particular, for the *rbtree* workload which experiences a significant 46% abort rate, sdTM provides only a minimal 5% improvement over SO.

We can also observe than ATOM provides a more robust average improvement of 35% over SO. Recall that ATOM uses the same concurrency control mechanism as SO (locks), but provides faster atomic durability by performing undo-logging in hardware. Interestingly, we can see that ATOM has a comfortable 15% advantage over sdTM because of the aborts experienced by the latter.

Our DHTM design provides the best throughput improvement amongst all compet-

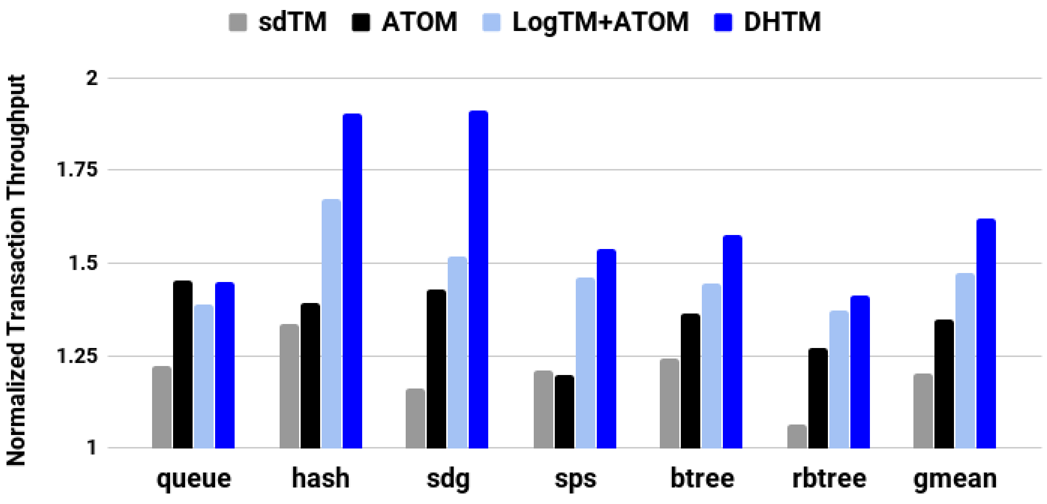


Figure 5.6: Transaction throughput normalized to SO.

	queue	hash	sdg	sps	btree	rbtree	Ave.
sdTM	68	19	23	27	37	46	37
DHTM	46	5	13	16	18	26	21

Table 5.5: Abort rates for sdTM and DHTM designs.

ing designs. On average, DHTM improves transaction throughput by 61% compared to SO. In comparison to SO, it can achieve faster durability by way of hardware logging and can also uncover more concurrency. In comparison with sdTM, DHTM improves transaction throughput by 41%. It not only benefits from faster durability, but also benefits from fewer aborts because DHTM supports write-set overflows from the L1. This is evidenced in Table 5.5, where we can see that DHTM suffers from a relatively lower 21% abort ratio in comparison with the 37% abort ratio of sdTM. In comparison with ATOM, DHTM provides a 26% higher improvement in transaction throughput. DHTM not only benefits from better concurrency (because of HTM), but also because logging is faster in DHTM. Indeed, because ATOM uses undo-logging, it suffers from the overhead of persisting in-place data in the commit critical path. In contrast, because DHTM uses redo-logging, data can be persisted out of the critical path.

Finally, we look at LogTM-ATOM which implements eager version management for both atomic visibility and atomic durability as opposed to hybrid version management on DHTM. On average, DHTM provides 17% higher improvement in throughput compared to LogTM-ATOM. Since both designs implement eager version management for atomic visibility, the difference in performance between the two designs is

	SO	ATOM	DHTM
TPC-C	1	1.67	1.88
TATP	1	1.27	1.53

Table 5.6: Transaction throughput for ATOM and DHTM normalized to SO for TPC-C and TATP benchmarks.

primarily because of difference in atomic durability mechanisms. In other words, the low transaction commit latency in DHTM enabled by the redo log leads to performance improvement over LogTM-ATOM. From this we can also infer that more than half of the DHTM's 26% improvement over ATOM is because of faster durability (and the rest owing to higher concurrency). In summary, DHTM provides a significant performance improvement, because of faster logging and/or integration with HTM, over the state-of-the-art design (ATOM [73]) and over a (novel) design combining LogTM [38] with ATOM.

5.6.2 Sensitivity to the size of the log-buffer

Figure 5.7 shows the impact of the size of log buffer on the performance of DHTM for the hash benchmark (other benchmarks show similar trends). We run the benchmark with buffer sizes ranging from 4 through 128 entries. As the number of entries are increased, the throughput increases, saturates at the size of 64 entries (default configuration in DHTM) and then marginally reduces upon further increase in the size. A small persist buffer size leads to creation of multiple redo log entries which consumes higher amount of memory bandwidth and adversely impacts other memory requests. On the other hand, a larger buffer delays log writes which results in those log writes happening in the critical path of commit. Recall that a transaction cannot commit until all the redo log entries have been made persistent. In summary, we find that a 64-entry log-buffer provides the best coalescing effect.

5.6.3 TPC-C and TATP Throughput

Table 5.6 shows the transaction throughput of ATOM and DHTM normalized to the throughput of SO for the TPC-C and TATP workloads. We have not shown sdTM results because it performs quite poorly. Because of the significant number of HTM aborts (owing to the large write working-set size of the OLTP workloads), these con-

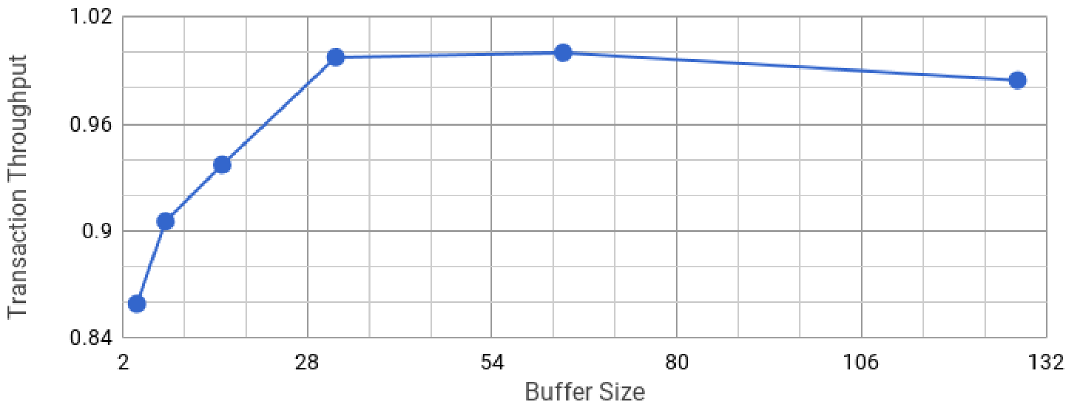


Figure 5.7: Normalized transaction throughput sensitivity towards log-buffer size for hash benchmark.

ventional HTM designs revert to the software concurrency mode often.

As we can see, DHTM continues to provide impressive speedups, not only over the SO baseline, but also over ATOM. Specifically, for the TPC-C workload, DHTM provides a 88% improvement over SO and 21% higher improvement compared to ATOM. For the TATP workload, DHTM provides a 53% improvement over SO and 26% higher improvement compared to ATOM.

5.6.4 The Cost of Atomic Durability

In this section, we wanted to see how close DHTM is compared to a non-persistent (volatile) HTM design that we call NP. For micro-benchmarks, we find that NP provides 2.2× higher transaction throughput compared to SO which is 59% better than DHTM.

Next, we wanted to better understand the reason for the performance gap. Are their inefficiencies in DHTM or is it fundamentally limited by the cost of durability? There are two primary sources of overheads in DHTM compared to NP. First, the overhead of log writes and data writes that are in the critical path. These include log writes that are pending when a transaction execution completes and is waiting to commit and data writes from the committed but yet to complete transaction pending when a core encounters the next transaction. To evaluate the performance impact of these overheads we implemented a DHTM design where these writes happen instantaneously. This design is able to improve performance over DHTM by 16% for micro-benchmarks Therefore log/data writes in the critical path of execution appear not to be the major source of overhead.

	1×	2×	10×
NP	2.9	3.0	3.3
DHTM	1.9	2.4	3

Table 5.7: Transaction throughput for NP and DHTM normalized to SO for hash benchmark with varying memory bandwidth.

The second source of overhead corresponds to a fundamental difference in memory write bandwidth requirements for DHTM and NP. Recall that, in comparison with NP, DHTM needs to flush cache lines for atomic durability. To analyze the impact of this overhead, we performed experiments by varying the available memory bandwidth. Table 5.7 shows the transaction throughput for NP and DHTM designs normalized to SO design for the hash micro-benchmark with varying memory bandwidth. With the baseline bandwidth (5.3 GB/s) the difference between NP and DHTM designs is 100% whereas with 10× the baseline bandwidth the difference is only 30%. Thus in a system with higher memory bandwidth DHTM can achieve performance similar to that of a volatile only (NP) design.

5.7 Summary

ACID transactions are a well-understood and widely adopted programming model. In this chapter we tried to answer the question: How fast can we achieve ACID in the presence of fast persistent memory? We have proposed DHTM, a HTM design in which durability is treated as a first class design constraint. It extends a commercial HTM like RTM with hardware support for atomic durability. It supports atomic visibility by employing an RTM like HTM and atomic durability by employing a hardware logging infrastructure which transparently and efficiently writes redo log entries to persistent memory. A redo-log based design allows us to commit a transaction as soon as all the log entries have been made persistent, without waiting for the data to persist.

One of our design goals was to support larger transactions, since ACID transactions tend to be considerably larger than those supported by current L1-limited RTM like HTM designs. But in supporting larger transactions we did not want to introduce significant hardware complexity. In particular, we did not want to introduce changes to the shared LLC – something that current HTM designs avoid. Our key insight here is to reuse the logging infrastructure that is necessary for durability for also supporting L1 overflows. Our experimental results showed that our proposal outperforms the state-of-the-art ACID design by an average of 26% on a set of micro-benchmarks and by a minimum of 21% on TATP and TPC-C.

Chapter 6

Conclusions and Future Work

This thesis has explored the design space of various crash consistency primitives that are necessary to enable the adoption of persistent memory. These primitives guarantee crash consistency by placing restrictions on the order in which updates reach persistent memory. Additionally, they also require that the updates be made durable periodically. Enforcing these constraints in a performance efficient manner is challenging for two reasons. First, these constraints go against certain performance enhancing optimizations which reorder updates to memory to maximize locality. Second, these constraints need to be enforced synchronously because software cannot explicitly control the movement of updated data from volatile cache to persistent memory.

This thesis has highlighted the fact that buffered models for implementing ordering primitives like persist barrier are important, but they need to be carefully designed. In particular, we have demonstrated that existing implementation of buffered persist barrier largely fails in achieving its goal of enforcing ordering constraints asynchronously because of the presence of conflicts. We incorporated two simple optimizations, to reduce the probability and the overhead of conflicts, in the design of an efficient persist barrier (LB++). We showed that using LB++ to enforce buffered epoch persistency can improve the performance by 22%.

Next, we highlighted that supporting atomic durability via undo logging performs log writes to persistent memory in the critical path. Based on the observation that logging is fundamentally a data movement task, we proposed ATOM: a hardware log manager. We showed that leveraging the existing memory hierarchy to perform posted log writes to the memory controller can remove log writes from the critical path. In fact, by having access to additional information, like whether a read request is for exclusive access, the memory controller can additionally eliminate redundant data movement by performing source logging. ATOM improved the performance by 27% to 33% for microbenchmarks and by 60% for a large scale transactional workload (TPC-C). We also proposed an efficient hardware mechanism for checkpointing programs which leverages LB++ to create checkpoints and ATOM to ensure atomic durability of those checkpoints. This approach enables checkpointing of applications at only a 30% execution time overhead compared to an execution without checkpointing.

Although atomic durability is an important primitive, the ease of programming can be significantly improved if persistent memory systems provide primitives to support ACID transactions which provide both atomic visibility and atomic durability. We argued that to achieve the best performance, ACID primitive has to be completely supported in hardware. We proposed durable hardware transactional memory (DHTM) in which durability is treated as a first class design constraint. Leveraging an RTM like HTM for supporting atomic visibility enables in-place updates and supporting atomic durability by employing a hardware logging infrastructure for a redo log enables faster commits. We also demonstrated that the existing logging infrastructure can be leveraged to support larger transactions, up to the size of the last level cache, without adding any additional hardware. DHTM outperforms the state-of-the-art ACID design by an average of 26% on a set of micro-benchmarks and by a minimum of 21% on TATP and TPC-C.

6.1 Critical Analysis

The proposed implementations of all the primitives in this thesis had one common design principle behind them: moving persist operations out of the critical path. However, in certain scenarios that might not be possible and such scenarios would be adversarial for these primitives. The ordering primitive moved persist operations out of the critical path by buffering epochs and tracking intra-thread and inter-thread persist dependencies in hardware. At the same time, it could maintain only 1 version of any cache line in the cache. Therefore, if all the epochs modified a common set of cache lines, the proposed ordering primitive would not be able to buffer epochs and would have to persist them in the critical path. In such a scenario, the proposed buffered implementation of persist barrier would degenerate to a non-buffered implementation.

ATOM was able to move log persist operations out of the critical path by enforcing log → data ordering at the memory controller level. To enable ordering at the memory controller, ATOM needed to track every ongoing atomic update in an atomic update structure (AUS). If the number of concurrent atomic updates exceeded the number of available AUS's then it would either have to stall the new atomic updates or enforce the necessary ordering requirements for those updates in software, which would bring persist operations in the critical path. Similarly, DHTM could support ACID transaction as long as the transaction write-set size did not exceed the LLC. If the write-set of a transaction overflows from the LLC then the transaction would be aborted and it will have revert to the software fallback path. And, as was the case with ATOM, the software fallback path would bring persist operations in the critical path.

In summary, the designs proposed in this thesis improved the performance of crash consistency primitives in most scenarios by moving persist operations out of the critical path. However, adversarial scenarios do exist in which the proposed designs cannot move persist operations out of the critical path. More importantly though, even in such scenarios, the performance of the proposed designs would not be any worse than a software implementation of these primitives.

6.2 Discussion

This section presents a discussion on certain design decisions and trade-offs that affect the performance of crash consistency primitives. It ends by presenting a common thread that underlines this thesis.

6.2.1 When to persist?

The performance of all crash consistency primitives can be improved by minimizing online persists that lie in the critical path and by maximizing write coalescing. However, these are conflicting requirements which make it difficult to answer the question *when to persist?* Delaying persist operations is necessary to allow write coalescing, but at the same time it is important to be proactive to avoid performing them online.

A proactive flush operation in buffered epoch persistency (BEP) is started after an epoch completes (§3.3.2), thereby maximizing coalescing of writes belonging to that epoch. This results in dirty cache lines remaining in the cache for a longer duration, in turn increasing the probability of a conflict. As a result cache lines persist online, as opposed to a scenario in which proactive flush would have been started sooner by potentially sacrificing some amount of write coalescing. Thus proactive flushing in BEP traded off offline persist operations for maximizing write coalescing. Similarly, ATOM also maximized write coalescing by employing an undo log which requires only one log entry per cache line for each atomic update and by flushing the in-place updates at the end of the atomic update. This was again a trade-off with respect to offline persists in two ways. First, flushing of in-place updates could be started sooner at the cost of sacrificing some coalescing. Second, a redo-log could be employed which would require one log entry for every write in an atomic update but would allow an atomic update to complete without waiting for in-place updates to reach persistent memory.

DHTM strove to balance the trade-off between write coalescing and online persists. It employed a redo log for durability which allowed the transaction to commit as soon as all the redo log entries had been written to persistent memory, without waiting for in-place updates to persist in the critical path. Therefore, in-place updates could be coalesced throughout the transaction and still persist out of the critical path. At the same time, DHTM also employed a log buffer that supported coalescing of log entries by combining multiple log entries for the same cache line into one log entry. In hindsight, a similar approach could have been applied to BEP and ATOM in the form of a data buffer to funnel in-place updates to persistent memory eagerly. Such a design could potentially improve their performance by better balancing the trade-off.

6.2.2 How to buffer?

Buffering (§2.3.1) has been proposed as a mechanism to enable offline (out of the critical path) persists. However, the extent to which offline persists are possible is determined by the frequency of conflicts. As shown in §3.7.2 for a lazy barrier, even with various optimizations, a large percentage of epochs might still encounter conflicts. Conflicts arise in this case because buffering is performed across two epochs which could modify the same cache line.

ATOM proposed an alternate model of buffering, which allowed buffering of undo log entries for an atomic update. This was different from the lazy barrier primitive because buffering here was performed only for log writes which would not overlap to trigger a conflict. Such a design had the twin benefits of eliminating conflicts and of incurring a negligible hardware overhead. In fact for checkpointing applications, ATOM improved performance by 59% while lazy barrier could improve it by only 37% (§ 4.8.2.2). DHTM also proposed a similar model of buffering, where redo log entries were buffered. With a redo log, two log entries could be created for the same cache line but that did not create a conflict because the log entries were part of the same transaction. Rather DHTM exploited this nature of redo log to coalesce multiple redo log entries for the same cache line (§5.3.1) and write the coalesced entry to persistent memory, thereby conserving memory bandwidth.

Employing buffered models, like buffered epoch persistency, could also be challenging for programmers because they do not provide any guarantees on what would have persisted at the time of a system crash. They only guarantee that persist operations would have happened in epoch order. Therefore it is desirable to have support analogous to the *fsync* system call in file systems, which guarantees that all the updates up to the point of the function call have been made persistent. Support for a sync operation was proposed in [74] which we believe is necessary to ease the adoption of models like buffered epoch persistency. It is important to note however, that support for a sync operation is implicitly provided by the *Atomic_End* and *End_Transaction* constructs in ATOM and DHTM respectively.

In summary, buffering is a useful construct but it needs to be carefully designed to avoid conflicts. Additionally, it important for buffered models to support a sync operation for ease of programming.

6.2.3 Undo vs. Redo

Atomic durability via write-ahead logging can be provided by using either an undo log or a redo log and both these approaches have different trade-offs. An undo log requires less memory bandwidth as only one log entry needs to be created per cache line, whereas a redo log allows for faster commits as the transaction can be committed as soon as all the redo log entries persist. However, the impact of these trade-offs will vary with certain design choices. For example, the number of log entries created for both undo and redo log can be the similar if logging is performed at a word granularity because the probability of the same word being modified multiple times in a transaction is low. On the other hand, an optimization like the log buffer (§5.3.1) which was used to allow coalescing of redo log entries, can be used for data writes in a design with an undo log. A small buffer for data writes could be viewed as a mechanism to proactively persist in-place data updates and thereby enable faster commits for transactions implementing an undo log.

Therefore, the choice between implementing an undo log or a redo log based design is not always straightforward. This choice should be carefully made by taking into consideration other design parameters and the scope for optimizations.

6.2.4 Intelligent Memory Systems

This thesis has argued for pushing more work into the memory system to provide efficient support for crash consistency primitives. In designing an efficient persist barrier [60] we moved the support for ordering into the cache hierarchy. In ATOM [73], the memory controller primarily handled the logging functionality in conjunction with the cache hierarchy. Again in DHTM [81], the cache hierarchy and the memory controller together supported the necessary operations to guarantee atomic visibility and atomic durability. In essence, this thesis has argued for the design of *intelligent memory systems* to efficiently support primitives for persistent memory.

6.3 Future Work

This thesis has presented the design of efficient crash consistency primitives for single node systems. However, more work needs to be done to efficiently enable the adoption of persistent memory. We provide our perspective on future directions below.

Addressing Memory Write Bandwidth. Although multiple optimizations have been proposed to enable write coalescing and thus reduce the memory write bandwidth, it is still a major bottleneck (§5.6.4). This is primarily because all the updates in a transaction need to be made durable for that transaction to commit, which significantly increases the number of writes to persistent memory. One of the potential ways of addressing this issue further is to look at optimizations like group commit [55], that commit transactions at the granularity of a group of transactions as opposed to individual transactions. Another promising direction is to look at an in-memory computing model for logging. It has been already observed in this thesis that logging is fundamentally a data movement task. If logging is completely offloaded to memory, then it can significantly reduce the pressure on memory write bandwidth as log writes contribute to more than half of the memory write bandwidth consumption of each transaction.

Verification. This thesis has proposed implementations of various primitives for crash consistency. These primitives place constraints on the order in which updates reach persistent memory and closely interact with memory consistency models and cache coherence. It is well known that verification of memory consistency models and cache coherence is a difficult problem. Adding persistency to the picture makes it even more challenging. But this is an important problem to address and significant work needs to be done in this direction to enable the adoption of persistent memory.

Programming Models. Although multiple programming models have been proposed for systems with persistent memory, there is no consensus on which model is the most suitable. More work needs to be done in this direction to standardize the interface available to the programmer to not only improve the ease of programming these systems but, more importantly, to enable efficient mechanisms to debug these systems.

Scale-Out Persistent Memory Systems. This thesis addresses the problem of designing crash consistency primitives for single node systems with persistent memory. However, the wide adoption of technologies like Remote Direct Memory Access (RDMA) enables low latency access to the memory of remote machines. This has lead to the emergence of scale-out systems for performing distributed transaction processing [82]. These scale-out systems would certainly benefit from persistent memory, not only for supporting ACID transactions efficiently but also to provide high availability via replication. We believe that the design of efficient crash consistency primitives for scale-out persistent memory systems is an important challenge that needs to be addressed in the future.

Bibliography

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

[2] S. Raoux, G. W. Burr, M. J. Breitwisch, C. T. Rettner, Y. Chen, R. M. Shelby, M. Salinga, D. Krebs, S. Chen, H. Lung, and C. H. Lam, "Phase-change random access memory: A scalable technology," *IBM Journal of Research and Development*, vol. 52, no. 4-5, 2008.

[3] T. Kawahara, R. Takemura, K. Miura, J. Hayakawa, S. Ikeda, Y. M. Lee, R. Sasaki, Y. Goto, K. Ito, T. Meguro, F. Matsukura, H. Takahashi, H. Matsuoka, and H. Ohno, "2Mb Spin-Transfer Torque RAM (SPRAM) with Bit-by-Bit Bidirectional Current Write and Parallelizing-Direction Current Read," in *Proceedings of the International Solid-State Circuits Conference*, 2007.

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

[5] E. Kultursay, M. T. Kandemir, A. Sivasubramaniam, and O. Mutlu, "Evaluating STT-RAM as an energy-efficient main memory alternative," in *Proceedings of the International Symposium on Performance Analysis of Systems and Software*, 2013.

[6] T. Haerder and A. Reuter, "Principles of Transaction-oriented Database Recovery," *ACM Comput. Surv.*, vol. 15, no. 4, 1983.

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

[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 Symposium on Operating Systems Principles*, 2009.

[9] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz, “ARIES: A Transaction Recovery Method Supporting Fine-granularity Locking and Partial Rollbacks Using Write-ahead Logging,” *ACM Trans. Database Syst.*, vol. 17, no. 1, 1992.

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

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

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

[13] H. Volos, A. J. Tack, and M. M. Swift, “Mnemosyne: Lightweight persistent memory,” in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2011.

[14] X. Wu and A. L. N. Reddy, “Scmfs: A file system for storage class memory,” in *Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis*, 2011.

[15] K. Doshi, E. Giles, and P. Varman, “Atomic Persistence for SCM with a Non-intrusive Backend Controller,” in *Proceedings of the International Symposium on High Performance Computer Architecture*, 2016.

[16] M. Liu, M. Zhang, K. Chen, X. Qian, Y. Wu, W. Zheng, and J. Ren, “DUDETM: Building Durable Transactions with Decoupling for Persistent Memory,” in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2017.

[17] H. Avni and T. Brown, "PHyTM: Persistent Hybrid Transactional Memory," *Proceedings of VLDB Endowment*, 2016.

[18] E. Giles, K. Doshi, and P. Varman, "Continuous Checkpointing of HTM Transactions in NVM," in *Proceedings of the International Symposium on Memory Management*, 2017.

[19] J. Chung, L. Yen, S. Diestelhorst, M. Pohlack, M. Hohmuth, D. Christie, and D. Grossman, "ASF: AMD64 Extension for Lock-Free Data Structures and Transactional Memory," in *Proceedings of the International Symposium on Microarchitecture*, 2010.

[20] R. Cypher, A. Landin, H. Zeffer, S. Yip, M. Karlsson, M. Ekman, S. Chaudhry, and M. Tremblay, "Rock: A High-Performance Sparc CMT Processor," *IEEE Micro*, 2009.

[21] Intel Corporation, *Intel® Architecture Instruction Set Extensions Programming Reference*.

[22] H. Q. Le, G. Guthrie, D. Williams, M. M. Michael, B. Frey, W. J. Starke, C. May, R. Odaira, and T. Nakaike, "Transactional memory support in the IBM POWER8 processor," *IBM Journal of Research and Development*, 2015.

[23] A. Wang, M. Gaudet, P. Wu, J. N. Amaral, M. Ohmacht, C. Barton, R. Silvera, and M. Michael, "Evaluation of Blue Gene/Q Hardware Support for Transactional Memories," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 2012.

[24] Z. Wang, H. Yi, R. Liu, M. Dong, and H. Chen, "Persistent Transactional Memory," *IEEE Computer Architecture Letters*, 2015.

[25] Intel Corporation, "Persistent Memory Programming." `http://pmem.io/`.

[26] H. Akinaga and H. Shima, "Resistive Random Access Memory (ReRAM) Based on Metal Oxides," *Proceedings of the IEEE*, vol. 98, no. 12, 2010.

[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 *Proceedings of International Symposium on High Performance Computer Architecture*, 2015.

[28] J. Åkerman, “Toward a Universal Memory,” *Science*, vol. 308, no. 5721, 2005.

[29] 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, no. 4, 2008.

[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 International Symposium on Microarchitecture*, 2009.

[31] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, “Scalable High Performance Main Memory System Using Phase-change Memory Technology,” in *Proceedings of the International Symposium on Computer Architecture*, 2009.

[32] 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 International Symposium on Computer Architecture*, 2009.

[33] D. J. Sorin, M. D. Hill, and D. A. Wood, *A Primer on Memory Consistency and Cache Coherence*. Synthesis Lectures on Computer Architecture, Morgan & Claypool Publishers, 2011.

[34] J. Gray, P. McJones, M. Blasgen, B. Lindsay, R. Lorie, T. Price, F. Putzolu, and I. Traiger, “The Recovery Manager of the System R Database Manager,” *ACM Comput. Surv.*, vol. 13, no. 2, 1981.

[35] 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 European Conference on Computer Systems*, 2014.

[36] M. Herlihy and J. E. B. Moss, “Transactional Memory: Architectural Support for Lock-Free Data Structures,” in *Proceedings of the International Symposium on Computer Architecture*, 1993.

[37] T. Harris, J. Larus, and R. Rajwar, “Transactional Memory, 2nd edition,” *Synthesis Lectures on Computer Architecture*, vol. 5, no. 1, 2010.

[38] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood, “LogTM: log-based transactional memory,” in *Proceedings of the International Symposium on High Performance Computer Architecture*, 2006.

[39] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie, "Unbounded transactional memory," in *Proceedings of the International Symposium on High-Performance Computer Architecture*, 2005.

[40] R. Rajwar, M. Herlihy, and K. Lai, "Virtualizing transactional memory," in *Proceedings of the International Symposium on Computer Architecture*, 2005.

[41] R. D. Schlichting and F. B. Schneider, "Fail-stop Processors: An Approach to Designing Fault-tolerant Computing Systems," *ACM Trans. Comput. Syst.*, vol. 1, no. 3, 1983.

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

[43] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 2008.

[44] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The splash-2 programs: Characterization and methodological considerations," in *Proceedings of the International Symposium on Computer Architecture*, 1995.

[45] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun, "Stamp: Stanford transactional applications for multiprocessing," in *Proceedings of the International Symposium on Workload Characterization*, 2008.

[46] 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 the International Symposium on Microarchitecture*, 2013.

[47] A. Kolli, S. Pelley, A. Saidi, P. M. Chen, and T. F. Wenisch, "High-Performance Transactions for Persistent Memories," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2016.

[48] A. Kolli, J. Rosen, S. Diestelhorst, A. Saidi, S. Pelley, S. Liu, P. M. Chen, and T. F. Wenisch, "Delegated Persist Ordering," in *Proceedings of the International Symposium on Microarchitecture*, 2016.

[49] D. R. Chakrabarti and H.-J. Boehm, “Durability semantics for lock-based multithreaded programs,” in *Proceedings of the Workshop on Hot Topics in Parallelism*, 2013.

[50] N. Agarwal, T. Krishna, L.-S. Peh, and N. Jha, “Garnet: A detailed on-chip network model inside a full-system simulator,” in *Proceedings of International Symposium on Performance Analysis of Systems and Software*, 2009.

[51] Y. Lu, J. Shu, L. Sun, and O. Mutlu, “Loose-ordering consistency for persistent memory,” in *Proceedings of the International Conference on Computer Design*, 2014.

[52] 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 and Operating Systems*, 2014.

[53] J. Zhao, O. Mutlu, and Y. Xie, “Firm: Fair and high-performance memory control for persistent memory systems,” in *Proceedings of the International Symposium on Microarchitecture*, 2014.

[54] L. Sun, Y. Lu, and J. Shu, “Dp2: Reducing transaction overhead with differential and dual persistency in persistent memory,” in *Proceedings of the International Conference on Computing Frontiers*, 2015.

[55] S. Pelley, T. F. Wenisch, B. T. Gold, and B. Bridge, “Storage management in the nvram era,” *Proceedings of VLDB Endowment*, vol. 7, no. 2, 2013.

[56] Intel Corporation, “Platform brief Intel Xeon Processor C5500/C3500 Series and Intel 3420 Chipset.” `http://download.intel.com/design/intarch/prodbref/323306.pdf`.

[57] L. Ceze, J. Tuck, P. Montesinos, and J. Torrellas, “Bulksc: Bulk enforcement of sequential consistency,” in *Proceedings of the International Symposium on Computer Architecture*, 2007.

[58] D. Narayanan and O. Hodson, “Whole-system persistence with non-volatile memories,” in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2012.

[59] X. Dong, N. Muralimanohar, N. Jouppi, R. Kaufmann, and Y. Xie, “Leveraging 3D PCRAM Technologies to Reduce Checkpoint Overhead for Future Exascale Systems,” in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, 2009.

[60] A. Joshi, V. Nagarajan, M. Cintra, and S. Viglas, “Efficient Persist Barriers for Multicores,” in *Proceedings of the 48th International Symposium on Microarchitecture*, 2015.

[61] 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*, 2015.

[62] T. Wang and R. Johnson, “Scalable Logging Through Emerging Non-volatile Memory,” *Proc. VLDB Endow.*, vol. 7, no. 10, 2014.

[63] J. Huang, K. Schwan, and M. K. Qureshi, “NVRAM-aware Logging in Transaction Systems,” *Proc. VLDB Endow.*, vol. 8, no. 4, 2014.

[64] J. DeBrabant, J. Arulraj, A. Pavlo, M. Stonebraker, S. Zdonik, and S. Dulloor, “A Prolegomenon on OLTP Database Systems for Non-Volatile Memory,” in *International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures*, VLDB Endowment, 2014.

[65] E. Giles, K. Doshi, and P. Varman, “Bridging the programming gap between persistent and volatile memory using WrAP,” in *Proceedings of the International Conference on Computing Frontiers*, 2013.

[66] Y. Lu, J. Shu, and L. Sun, “Blurred Persistence: Efficient Transactions in Persistent Memory,” *Trans. Storage*, vol. 12, no. 1, 2016.

[67] J. Meza, Y. Luo, S. Khan, J. Zhao, Y. Xie, and O. Mutlu, “A Case for Efficient Hardware-Software Cooperative Management of Storage and Memory,” in *Proceedings of the Workshop on Energy-Efficient Design*, 2013.

[68] F. Nawab, D. R. Chakrabarti, T. Kelly, and C. B. M. III, “Procrastination beats prevention: Timely sufficient persistence for efficient crash resilience,” in *Proceedings of the International Conference on Extending Database Technology*, 2015.

[69] S. Harizopoulos, D. J. Abadi, S. Madden, and M. Stonebraker, "OLTP Through the Looking Glass, and What We Found There," in *Proceedings of the International Conference on Management of Data*, 2008.

[70] T. Karnagel, R. Dementiev, R. Rajwar, K. Lai, T. Legler, B. Schlegel, and W. Lehner, "Improving in-memory database index performance with Intel® Transactional Synchronization Extensions," in *Proceedings of the International Symposium on High Performance Computer Architecture*, 2014.

[71] V. Leis, A. Kemper, and T. Neumann, "Scaling HTM-Supported Database Transactions to Many Cores," *IEEE Transactions on Knowledge and Data Engineering*, vol. 28, no. 2, 2016.

[72] T. Wang and R. Johnson, "Scalable Logging Through Emerging Non-volatile Memory," *Proc. VLDB Endow.*, 2014.

[73] A. Joshi, V. Nagarajan, S. Viglas, and M. Cintra, "ATOM: Atomic Durability in Non-volatile Memory through Hardware Logging," in *Proceedings of the International Symposium on High Performance Computer Architecture*, 2017.

[74] S. Nalli, S. Haria, M. D. Hill, M. M. Swift, H. Volos, and K. Keeton, "An Analysis of Persistent Memory Use with WHISPER," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2017.

[75] E. L. M. Matheus A. Ogleari and J. Zhao, "Relaxing persistent memory constraints with hardware-driven undo+redo logging," tech. rep., University of California, Santa Cruz, 2016.

[76] S. Shin, S. K. Tirukkovalluri, J. Tuck, and Y. Solihin, "Proteus: A Flexible and Fast Software Supported Hardware Logging Approach for NVM," in *Proceedings of the International Symposium on Microarchitecture*, 2017.

[77] A. Joshi, V. Nagarajan, S. Viglas, and M. Cintra, "Architectural Support for Atomic Durability in Non-Volatile Memory," in *Non-Volatile Memories Workshop*, 2018.

[78] E. Giles, K. Doshi, and P. Varman, "Brief Announcement: Hardware Transactional Storage Class Memory," in *Proceedings of the Symposium on Parallelism in Algorithms and Architectures*, 2017.

[79] V. Leis, A. Kemper, and T. Neumann, “Exploiting hardware transactional memory in main-memory databases,” in *International Conference on Data Engineering*, 2014.

[80] I. Calciu, T. Shpeisman, G. Pokam, and M. Herlihy, “Improved single global lock fallback for best-effort hardware transactional memory,” in *Transact Workshop*, 2014.

[81] A. Joshi, V. Nagarajan, M. Cintra, and S. Viglas, “DHTM: Durable Hardware Transactional Memory,” in *Proceedings of the International Symposium on Computer Architecture*, 2018.

[82] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson, “FaRM: Fast Remote Memory,” in *Proceedings of the Symposium on Networked Systems Design and Implementation*, 2014.