RICE UNIVERSITY

Hardware Transactional Persistent Memory

by

Ellis Giles

A Thesis Submitted
In Partial Fulfillment of the
Requirements for the Degree
Doctor of Philosophy

Approved, Thesis Committee:

[Signatures]

Peter Varman, Chair
Professor of Electrical and Computer
Engineering and Computer Science

Joseph R. Cavallaro
Professor of Electrical and Computer
Engineering and Computer Science

Christopher Jermaine
Professor of Computer Science

Eugene Ng
Professor of Computer Science and
Electrical and Computer Engineering

Kshiraj Doshi
Senior Principal Engineer, Intel
Corporation

Houston, Texas
January, 2019
ABSTRACT

Hardware Transactional Persistent Memory

by

Ellis Giles

Recent years have witnessed a sharp shift towards real-time data-driven and high-throughput applications, impelled by pervasive multi-core architectures and parallel programming models. This shift has spurred a broad adoption of in-memory databases and massively-parallel transaction processing across scientific, business, and industrial application domains. However, these applications are severely handicapped by the difficulties in maintaining persistence on typical durable media like hard-disk drives (HDDs) and solid-state drives (SSDs) without sacrificing either performance or reliability.

Two emerging hardware developments hold enormous promise for transformative gains in both speed and scalability of concurrent data-intensive applications. The first is the arrival of Persistent Memory, or PM, a generic term for byte-addressable non-volatile memories, such as Intel’s 3D XPoint™ technology. The second is the availability of CPU-based transaction support known as Hardware Transactional Memory, or HTM, which makes it easier for applications to exploit multi-core concurrency without the need for expensive lock-based software.

This thesis introduces Hardware Transactional Persistent Memory, the first union of HTM with PM without any changes to known processor designs or protocols, allowing for high-performance, concurrent, and durable transactions. The techniques
presented are supported on three pillars: handling uncontrolled cache evictions from the processor cache hierarchy, logging to resist failure during persistent memory updates, and transaction ordering to permit consistent recovery from a machine crash. We develop pure software solutions that work with existing processor architectures as well as software-assisted solutions that exploit external memory controller hardware support. The thesis also introduces the notion of relaxed versus strict durability, allowing individual applications to tradeoff performance against robustness, while guaranteeing recovery to a consistent system state.
Dedication

This thesis is dedicated in memory of my father Vick Forrest Giles and as an inspiration to my children Alden Samuel Giles and Meredith Elise Giles.
Acknowledgments

First and foremost, I would like to thank my advisor Dr. Peter Varman for the many meetings and countless hours helping me develop research ideas and pointing out interesting areas for more exploration. Only with his guidance and support has this research, including this thesis and the many papers, been possible.

I would also like to thank Dr. Kshitij Doshi from Intel Corporation for all of the guidance, insight, and assistance into the area of research along with the hours of help spent with papers and research ideas. Additionally, I would like to thank Dr. Joseph Cavallaro, Dr. Chris Jermaine, and Dr. Eugene Ng for providing valuable feedback and taking the time to serve on my thesis committee.

Finally, I would like to thank all of my family. Without the continuing support from the entire family, I would not have been able to attend graduate school or immerse myself in research. My wife Mary-Helen was invaluable at providing support and helping with the children when graduate school made demands on my time. May this thesis inspire my children Alden Samuel Giles and Meredith Elise Giles to pursue higher education. My father Vick F. Giles also provided daily support for the children and guidance on my education. He would have wished to have seen the completion of the work and attended graduation. My mother and stepfather, Linda and Douglas Rowlett, also provided invaluable help and guidance along the way as well as inspiration with their Ph.D.’s from Rice University.

I would also like to acknowledge the financial support from the National Science Foundation and the Software and Services Group at Intel Corporation.
Contents

Abstract ii
Dedication iv
Acknowledgments v
List of Illustrations xi
List of Tables xvii
List of Algorithms xviii
List of Theorems xix

1 Introduction and Overview 1

1.1 Introduction 1
1.2 In-Memory Data Management Challenges 2
1.3 Persistent Memory Overview 4

1.3.1 Persistent Memory and Big Data 6
1.4 Hardware Transactional Memory Overview 7
1.5 HTM + PM Overview 10
1.6 System Architecture Overview 12
1.7 Contributions 15
1.8 Thesis Organization 18

2 Persistent Memory Transactions 20

2.1 Persistent Memory Access 20

2.1.1 Persistent Memory Access Transaction Properties 22
2.2 Durability for Persistent Memory 24

2.2.1 Persistent Memory Models 26
2.3  Atomicity for Persistent Memory Transactions
     2.3.1  Undo Logging
     2.3.2  Block Copy-on-Write
     2.3.3  Redo Logging
     2.3.4  Versioning
     2.3.5  Summary

2.4  Isolation for Concurrent Persistent Memory Transactions
     2.4.1  Two-Phase Locking
     2.4.2  Software Transactional Memory
     2.4.3  Hardware Transactional Memory
     2.4.4  Checkpoints
     2.4.5  Summary

2.5  Related Persistent Memory Transaction Work
     2.5.1  Hardware Supported Transaction Work
     2.5.2  File System Work
     2.5.3  Software and Concurrent Transaction Work

2.6  Summary

3  Hardware Transactional Memory + Persistent Memory

3.1  HTM + PM Basics
     3.1.1  Fallback Paths

3.2  Challenges Persisting HTM Transactions
     3.2.1  Logging Challenges
     3.2.2  Ordering Challenges

3.3  Related Work

3.4  Hardware Transactional Persistent Memory Model
     3.4.1  Ordering HTM Transactions
     3.4.2  System Checkpointing Model
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.4.3</td>
<td>Transaction Durability Model</td>
<td>78</td>
</tr>
<tr>
<td>3.4.4</td>
<td>Transactional Durability Extensions</td>
<td>79</td>
</tr>
<tr>
<td>3.4.5</td>
<td>Logging and Recovery</td>
<td>80</td>
</tr>
<tr>
<td>3.4.6</td>
<td>Composability</td>
<td>81</td>
</tr>
<tr>
<td>3.4.7</td>
<td>Example</td>
<td>82</td>
</tr>
<tr>
<td>3.4.8</td>
<td>HTM Performance Tuning for Durability</td>
<td>84</td>
</tr>
<tr>
<td>3.5</td>
<td>Summary</td>
<td>92</td>
</tr>
<tr>
<td>4</td>
<td>Continuous Checkpointing HTM for PM</td>
<td>93</td>
</tr>
<tr>
<td>4.1</td>
<td>Background on Variable Aliasing for PM Transactions</td>
<td>93</td>
</tr>
<tr>
<td>4.2</td>
<td>Continuous Checkpointing HTM for PM Approach</td>
<td>95</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Alias Table Model</td>
<td>99</td>
</tr>
<tr>
<td>4.3</td>
<td>Algorithm and Implementation</td>
<td>102</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Blue and Red Queues</td>
<td>103</td>
</tr>
<tr>
<td>4.3.2</td>
<td>Logging and Recovery</td>
<td>104</td>
</tr>
<tr>
<td>4.3.3</td>
<td>Alias Table Implementation</td>
<td>105</td>
</tr>
<tr>
<td>4.3.4</td>
<td>Lock Free Algorithm</td>
<td>107</td>
</tr>
<tr>
<td>4.3.5</td>
<td>Data Structure Properties</td>
<td>113</td>
</tr>
<tr>
<td>4.4</td>
<td>Evaluation</td>
<td>114</td>
</tr>
<tr>
<td>4.4.1</td>
<td>Benchmarks</td>
<td>116</td>
</tr>
<tr>
<td>4.4.2</td>
<td>Hash Table</td>
<td>121</td>
</tr>
<tr>
<td>4.4.3</td>
<td>Red-Black Tree</td>
<td>125</td>
</tr>
<tr>
<td>4.5</td>
<td>Summary</td>
<td>128</td>
</tr>
<tr>
<td>5</td>
<td>Hardware Transactional Persistent Memory Controller</td>
<td>130</td>
</tr>
<tr>
<td>5.1</td>
<td>Background on WrAP Controller</td>
<td>130</td>
</tr>
<tr>
<td>5.2</td>
<td>Persistent Memory Controller Approach</td>
<td>132</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Transaction Lifecycle</td>
<td>133</td>
</tr>
<tr>
<td>5.2.2</td>
<td>Persistent Memory Controller</td>
<td>136</td>
</tr>
</tbody>
</table>
Illustrations

1.1 Persistent Memory provides a large quantity of fast, byte-addressable persistent memory alongside DRAM on the main memory bus. 5

1.2 Logical layout of an Intel E5 Series High-Core Count processor that supports HTM transactions with 18 cores and two memory controllers with both DRAM and PM. 13

1.3 Logical layout of two Intel Xeon processors on a motherboard with a total of four memory controllers accessing both DRAM and PM with HTM support. Processors and cores have shared and cache-coherent access to all memory locations. 14

2.1 Persistent Memory can be accessed directly using normal loads and stores, bypassing the file system interface and page caches. 21

2.2 Writes to persistent memory from a core require moving through multiple buffers, complicating persistence. 24

2.3 Writes to PM may be evicted out of the cache hierarchy in any order. 29

2.4 Undo Logging approach for transactional atomicity of persistent memory. 30

2.5 Redo Logging approach for transactional atomicity of persistent memory. 32

2.6 Checkpointing Methods of Concurrent Transactions. 41

3.1 Problems with logging outside HTM + PM transactions. 62
### 4.8 SSA2 Compute Kernel Execution Transactional Throughput with Varying Maximum Allowable Lag for 8 Threads

- Page: 120

### 4.9 Vacation Benchmark Execution Time as a Function of the Number of Parallel Execution Threads

- Page: 121

### 4.10 Vacation Benchmark Execution Time with Various PM Write Times for Four Threads

- Page: 122

### 4.11 Millions of Transactions per Second for Hash Table Updates of 10 Elements versus Concurrent Number of Threads

- Page: 122

### 4.12 Hash Update of 10 Element Transaction Throughput for 6 Threads with Varying Maximum Allowable Lag

- Page: 123

### 4.13 Average Transactions per Second for Increasing Group Sizes of Atomic Hash Table Updates with 6 Concurrent Threads

- Page: 123

### 4.14 Average Transactions per Second for Increasing Write Percentages of 10 Atomic Hash Table Operations with 6 Threads

- Page: 124

### 4.15 Average Response Time in us for varying Transaction Arrival Rate for Red-Black Tree Element Inserts with 4 Threads

- Page: 126

### 4.16 Millions of Transactions per Second for Atomic Red-Black Tree Element Inserts versus Number of Concurrent Threads

- Page: 126

### 4.17 Red-Black Tree Element Insert Transaction Throughput for 4 Threads with Varying Maximum Allowable Lag

- Page: 127

### 4.18 Red-Black Tree Element Insert Transaction Throughput for 4 Threads with Varying PM Write Times

- Page: 127

### 5.1 WrAP Controller intercepts cache evictions and cache misses

- Page: 131

### 5.2 PM Controller with Volatile Delay Buffer, Current Open Transactions Set, and Dependency Wait Queue

- Page: 136

### 5.3 Flow of a transaction with implementation using HTM, cached write sets, timing, and logging

- Page: 142
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.4</td>
<td>SSCA2 Benchmark Compute Graph Kernel Execution Time as a Function of the Number of Parallel Execution Threads</td>
<td>152</td>
</tr>
<tr>
<td>5.5</td>
<td>SSCA2 Benchmark Compute Graph Kernel Speedup as a Function of the Number of Parallel Execution Threads</td>
<td>153</td>
</tr>
<tr>
<td>5.6</td>
<td>SSCA2 Benchmark Compute Graph Kernel HTM Aborts as a Function of the Number of Parallel Execution Threads</td>
<td>153</td>
</tr>
<tr>
<td>5.7</td>
<td>Vacation Benchmark Execution Time as a Function of the Number of Parallel Execution Threads</td>
<td>154</td>
</tr>
<tr>
<td>5.8</td>
<td>Vacation Benchmark Execution Time with Various Persistent Memory Write Times for Four Threads</td>
<td>155</td>
</tr>
<tr>
<td>5.9</td>
<td>Millions of Transactions per Second for Hash Table Updates of 10 Elements versus Concurrent Number of Threads</td>
<td>156</td>
</tr>
<tr>
<td>5.10</td>
<td>Millions of Transactions per Second for Hash Table Updates of 20 Elements versus Concurrent Number of Threads</td>
<td>157</td>
</tr>
<tr>
<td>5.11</td>
<td>Average Txps for Increasing Transaction Sizes of Atomic Hash Table Updates with 6 Concurrent Threads</td>
<td>158</td>
</tr>
<tr>
<td>5.12</td>
<td>Average Txps for Write / Read Percentage of Atomic Hash Table Updates with 6 Concurrent Threads</td>
<td>158</td>
</tr>
<tr>
<td>5.13</td>
<td>Millions of Transactions per Second for Atomic Red-Black Tree Element Inserts versus Number of Concurrent Threads</td>
<td>159</td>
</tr>
<tr>
<td>5.14</td>
<td>PM Controller simulation is performed using extensions to the McSimA+ simulator and DRAMSim2</td>
<td>161</td>
</tr>
<tr>
<td>5.15</td>
<td>Average Atomic Hash Table 10 Element Update with Various Persistent Memory Write Times with 8 Concurrent Threads</td>
<td>162</td>
</tr>
<tr>
<td>5.16</td>
<td>Maximum FIFO Queue Length for Atomic Hash Table 10 Element Update with 4 ConcurrentThreads</td>
<td>163</td>
</tr>
<tr>
<td>5.17</td>
<td>Average B-Tree Atomic Element Insert with Various Persistent Memory Write Times with 8 Concurrent Threads</td>
<td>164</td>
</tr>
<tr>
<td>Section</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>5.18</td>
<td>Maximum FIFO Queue Length for B-Tree Atomic Element Insert with 8 Concurrent Threads</td>
<td>164</td>
</tr>
<tr>
<td>A.1</td>
<td>Example with Local Alias Table Batched</td>
<td>178</td>
</tr>
<tr>
<td>A.2</td>
<td>Redis Benchmark Set Throughput for Byte-Addressable Persistence Methods and Increasing PM Write Times</td>
<td>180</td>
</tr>
<tr>
<td>A.3</td>
<td>Redis Benchmark Throughput for Various Byte-Addressable Persistence Methods</td>
<td>181</td>
</tr>
<tr>
<td>B.1</td>
<td>Container based storage with byte-addressable, fast Persistent Memory</td>
<td>184</td>
</tr>
<tr>
<td>B.2</td>
<td>Containerized Persistent Memory System Design</td>
<td>185</td>
</tr>
<tr>
<td>B.3</td>
<td>Data Layout on /dev/pmem0 for CPM</td>
<td>187</td>
</tr>
<tr>
<td>B.4</td>
<td>Redis Benchmark of Service Requests Per Second</td>
<td>189</td>
</tr>
<tr>
<td>B.5</td>
<td>Average Number of B-Tree Random Element Insert Transactions Per Second</td>
<td>190</td>
</tr>
<tr>
<td>C.1</td>
<td>SSCA2 Benchmark Compute Graph Kernel Performance Relative to HTM Only as a Function of the Number of Parallel Execution Threads</td>
<td>197</td>
</tr>
<tr>
<td>C.2</td>
<td>SSCA2 Benchmark Compute Graph Kernel Speedup as a Function of the Number of Parallel Execution Threads</td>
<td>197</td>
</tr>
<tr>
<td>C.3</td>
<td>SSCA2 Benchmark Compute Graph Kernel HTM Aborts Relative to HTM Only as a Function of the Number of Parallel Execution Threads</td>
<td>198</td>
</tr>
<tr>
<td>C.4</td>
<td>SSCA2 Compute Kernel Normalized Transactional Throughput with Varying Maximum Allowable Lag for 8 Threads</td>
<td>198</td>
</tr>
<tr>
<td>C.5</td>
<td>Vacation Benchmark Transaction Rate Relative to HTM Only as a Function of the Number of Parallel Execution Threads</td>
<td>200</td>
</tr>
<tr>
<td>C.6</td>
<td>Transaction Throughput Relative to HTM Only for Hash Table Updates of 10 Elements versus Concurrent Number of Threads</td>
<td>201</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>-------</td>
<td>------</td>
</tr>
<tr>
<td>C.7</td>
<td>Speedup Over PTL-Eager for Hash Table Updates of 10 Elements versus Concurrent Number of Threads</td>
<td>201</td>
</tr>
<tr>
<td>C.8</td>
<td>Transaction Rate Relative to HTM Only for Increasing Transaction Sizes of Atomic Hash Table Updates with 6 Concurrent Threads</td>
<td>203</td>
</tr>
<tr>
<td>C.9</td>
<td>Speedup Over PTL-Eager for Increasing Transaction Sizes of Atomic Hash Table Updates with 6 Concurrent Threads</td>
<td>203</td>
</tr>
<tr>
<td>C.10</td>
<td>Transaction Rate Relative to HTM Only for Write / Read Percentage of Atomic Hash Table Updates of 20 Elements with 6 Concurrent Threads</td>
<td>204</td>
</tr>
<tr>
<td>C.11</td>
<td>Speedup Over PTL-Eager for Write / Read Percentage of Atomic Hash Table Updates of 20 Elements with 6 Concurrent Threads</td>
<td>204</td>
</tr>
<tr>
<td>C.12</td>
<td>Transaction Rate Relative to HTM Only for Atomic Red-Black Tree Element Inserts versus Number of Concurrent Threads</td>
<td>205</td>
</tr>
<tr>
<td>C.13</td>
<td>Transaction Speedup Over PTL-Eager for Atomic Red-Black Tree Element Inserts versus Number of Concurrent Threads</td>
<td>206</td>
</tr>
</tbody>
</table>
# Tables

1.1 Transfers A, B, and C for Accounts W, X, Y, and Z .......................... 8

2.1 Transactions A, B, and C for variables w, x, y, and z to PM .................. 28

2.2 Comparison of atomic update methods for a transaction of \( n \) variable writes ................................................................. 34

2.3 Concurrency techniques for parallel transactions using locking, both coarse and fine-grained ......................................................... 36

2.4 Software and Hardware Transactional Memory for Concurrency ........ 39

2.5 Comparison of concurrency methods used for durable transactions .... 42

3.1 Example HTM Transactions ......................................................... 58

5.1 Example Contents of the Persistent Memory Controller Current Open Transactions Dependency Set and FIFO Queue ................................. 149
Algorithms

1. HTPM Transaction Structure ........................................... 71
2. HTM With Fallback Locking and Timestamp Counters ............... 85
3. cc-HTM Transaction Structure .......................................... 97
4. cc-HTM Transaction Implementation ................................... 107
5. Lock-Free Persistence Management Queue Helper Routines ....... 109
6. Lock-Free Persistence Management Queue Retirement Thread .... 111
7. Concurrent WrAP User Software Library ............................. 143
8. Hardware WrAP Implementation ....................................... 146
Properties

2.1 Definition (Transaction) .............................................. 22
2.2 Definition (Conflict) ...................................................... 35

3.1 Definition (Competing) .................................................. 73
3.2 Definition (Commit) ...................................................... 73
3.3 Definition (Precedes) ..................................................... 73
3.4 Definition (Precedence Set) ............................................. 73
3.5 Definition (Retirement) ................................................... 73
3.6 Definition (Strict Durability) ............................................ 73
3.7 Definition (Relaxed Durability) ......................................... 74
3.8 Definition (Lag) ........................................................... 74
3.9 Lemma (HTM Transaction Ordering) ................................. 75
3.10 Definition (TXFence) .................................................... 80
3.11 Definition (TXBarrier) .................................................... 80

4.1 Lemma (Transaction Retirement Wave) ............................... 98
4.2 Lemma (Aliasing with HTM) ............................................. 100
4.3 Lemma (Global Aliasing Preserves HTM Correctness) ............. 100
4.4 Lemma (Safe Alias Table Pruning) ..................................... 102
4.5 Lemma (Bounded Transaction Management Size) .................... 113
4.6 Lemma (Constant Transaction Start Time Cost) ...................... 113
4.7 Lemma (Amortized O(1)) ............................................... 114
5.1 Lemma (Controller Ordering) ........................................ 140
5.2 Lemma (Controller Cache Line Updates) ....................... 140
5.3 Lemma (Controller Recovery Routine) ......................... 140
5.4 Lemma (Controller PM Updates) ................................. 141
Chapter 1

Introduction and Overview

1.1 Introduction

In today’s data centric society, data storage and compute power are crucial to high performance industrial applications including cutting edge data science and deep learning algorithms. Over the past several decades, computer systems and data processing power has rapidly increased. Historically, this increase has met demands for real-time data-driven and high-throughput applications, many employing in-memory database technology for access efficiency. While demand for improved data processing keeps increasing, no longer are applications benefitting from increasing chip densities and processor clock rates. A half-century trend in increasing application performance has been stymied by the recent ending of both Moore’s Law and Dennard Scaling. This crippling has forced developers of these applications to explore other avenues for performance gains.

Fortunately, two emerging technologies have the potential to revive increases in application performance and usher in a new era with transformative gains in data speed and scalability. First is the arrival of byte-addressable and large non-volatile memories called Persistent Memory or PM, such as Intel/Micron’s 3D XPoint® technology. PM simplifies management of durability by eliminating the I/O operations and bottlenecks common to many data management applications. Second is the availability of CPU-based transaction support with Hardware Transactional Memory or HTM. HTM simplifies parallel programming by providing hardware support for concurrency management in pervasive shared-memory multiprocessors.

However, both PM and HTM, in addition to having their own limitations, when
brought together present an entirely new set of challenges. To enable PM and HTM cooperation, researchers recently thought extensive changes to well-known processors and their protocols were required. Changing well-known processors and protocols is a problem not to be taken lightly, as these changes can take a decade or longer, if ever, to realize.

This thesis introduces Hardware Transactional Persistent Memory, the first union of Hardware Transactional Memory with Persistent Memory without requiring any changes to processors or their underlying communication protocols. The thesis further introduces the model of \textit{relaxed} vs. \textit{strict durability}, whereby Persistent Memory operations can tradeoff processing speed with recoverability guarantees. This Chapter presents the high-level view of both technologies and the overall system architecture. The basic challenges for each technology are presented along with combined challenges. Subsequent chapters present detailed challenges and solutions when combining PM with HTM.

1.2 In-Memory Data Management Challenges

Data management applications and database management systems (DMBS) have traditionally stored their data on hard-disk drives (HDDs) and solid-state drives (SSDs) or a combination of the two. These non-volatile devices ensure that data is durable and persisted for protection against loss of power or system failure. Transactions, which read or update the data, bring relevant data into main dynamic memory (DRAM or Dynamic Random Access Memory), operate on them, and write them back to the non-volatile storage. However, these traditional non-volatile devices are block-based; they require data to be read and written in large continuous chunks (usually 4KB or larger) to amortize device delays. Several system-level performance optimization mechanisms like prefetching, main-memory caching, log-structured data organization, and transaction grouping have been designed and implemented to reduce the performance penalty of these slow, block-based backend devices.
Despite these clever techniques, application throughput still suffers as a result of slower persistence operations. To alleviate any persistence delays, many systems employ main-memory caches (such as Memcached) or In-Memory Databases (IMBDs) and data structure stores. Some of these in-memory solutions include Redis [4], SAP HANA [5], DB2 BLU [6], IBM solidDB [7], voltDB [8], and Neo4J [9]. Other high-throughput memory based applications used today throughout business, scientific, and industrial application domains include SciSpark [10], an in-memory parallel computing framework for scientific applications, MLlib [11], a scalable machine learning library built on Apache Spark’s cluster computing platform, and GridGain [12], an in-memory computing platform. These solutions hold all or a large fraction of the data in main memory. In-memory access has many advantages including query and update speed and use of in-memory data structures for fine-grained pointer following. Although in-memory processing delivers fast data accesses helping support data processing demands, there remains a risk for loss of data and system integrity on a system crash or power failure.

To mitigate the loss of data on system failure, many in-memory data management systems offer several options. One approach, such as Whole-System Persistence [13], utilizes redundant servers for backup on failure. While this approach guards against the loss on a single server, power or system failure on multiple servers remains a risk. Also, doubling the number of servers essentially doubles the cost of the system.

Another approach to guard against in-memory data loss involves logging all updates or periodic checkpointing of the data to a non-volatile medium like HDDs or SSDs. Logging all updates can be slow and synchronous, limiting application performance. Periodic checkpoint can be faster while guarding against data loss, but checkpointing suffers many disadvantages. First, many of the in-memory data systems process a million operations a second, so even a loss of one second of data loses millions of records. Additionally, checkpoints take time and processor and disk power, reducing overall application performance and increasing power requirements. Both
SAP HANA \[5\] and Redis \[4\] offer options for persistence, including checkpointing and logging.

Additional disadvantages of in-memory data stores include space limitations and power consumption. Typical DRAM sizes on commercial servers are in the range of hundreds of gigabytes. DRAM also has to be constantly refreshed by the processor several times per second. Very large data stores, on the order of Terra Bytes or more require multiple servers or backing by HDDs or SSDs. Both of these solutions to large data drive up costs and slow access speeds.

Fortunately, a new type of computer memory, called Persistent Memory or PM, can address the challenges faced by large, in-memory data stores. Persistent Memory is non-volatile, so a slower persistence mechanism to HDDs and SSDs is not required. Instead in-memory data applications can persist data directly, in place in main computer memory. Additionally, PM has a much larger storage capacity than DRAM with a much lower passive power requirement, as it also does not require constant refresh. Finally, PM is much faster than persistent SSDs, and almost as fast as DRAM making it an ideal choice for in-memory data storage.

1.3 Persistent Memory Overview

Emerging Persistent Memory technologies \[14\] like Memristors, Spin-Torque MRAM (ST-MRAM), Phase Change Memory (PCM) and the release of Intel/Micron’s 3D XPoint\textsuperscript{®} \[1\] provides applications the best of both worlds: fast, cache-line granularity access of DRAM with the persistence of disk or SSD. Persistent Memory, or PM, is a new class of memory that is both byte-addressable and non-volatile, providing direct access to vast amounts of persistent memory through familiar byte/word grained LOAD and STORE instructions, hitherto employed to access the volatile memory hierarchy. This access mechanism provides performance and programmability benefits by eliminating the I/O operations and bottlenecks common to many data management applications. The combination of direct and cacheable access from CPUs, persistence,
and high capacity means that PM can bridge a long-standing gap and discontinuity between slow block-based persistent storage and fast, byte-addressable volatile memory (DRAM) as shown in Figure 1.1.

Figure 1.1: Persistent Memory provides a large quantity of fast, byte-addressable persistent memory alongside DRAM on the main memory bus.

However, PM also raises problems of *Durability* (power failures leaving pending updates in volatile buffers), memory ordering, *Atomicity*, and *Consistency* (power failures while PM is partially updated with autonomous out-of-order cache evictions). These challenges are addressed in more detail in Chapter 2.

Due to the byte-addressable and high-speed nature of the technology, Persistent Memory is expected to be exposed on the main memory bus and operate alongside DRAM instead of being accessed through an I/O interface like SATA or PCI interface [15, 16, 17, 18, 19, 20]. Accessing PM through a PCI interface would incur a longer latency than accessing PM directly from the main memory bus. PM memory can enable systems with large amounts of persistent, byte-addressable memory that can replace slower, block-based hard disk or flash disk drives.

Replacing slower disks with Persistent Memory gives rise to a new class of applications, making it possible to use algorithms and data structures designed for byte-addressable volatile memory. Currently, applications that require persistence must
periodically save in-memory data to a persistence medium, often times requiring updates to a 4K block of data. Even a one byte update would require a 4,096 bytes to save to a persistence medium with a block size of 4K. With direct processor access to non-volatile PM, there is no need for systems to add complex disk-based checkpointing and logging as separate mechanisms, or endure time-consuming rebuilding of memory structures from disk-formatted data following a system reboot.

Several technologies with different performance, durability, and capacity characteristics are under research and development or on the verge of broad commercial deployment. These include nominally fast and medium sized ReRAM technology by Micron and Sony, slower, but very large capacity Phase Change Memory (PCM) technology by Micron and others, and fast, small spin-torque ST-MRAM technology by Everspin. Phase Change Memory, or PCM, is a promising type of PM as it can achieve a high chip density and speed [16, 21], and it has almost a thousand times greater write endurance over Flash technology. Additionally, Phase Change Memory requires only a fraction of the energy required for reading and writing when compared to Flash (1/10th the energy for reads and 1/100th the energy for writes) [17]. With the high density and low power, PM has the potential to replace disks entirely and reduce system energy requirements, while simultaneously speeding data accesses.

Due to the emerging nature of the technology, the moniker Persistent Memory or PM is not yet standard and may be referred to by a number of different names. Some of these names include Persistent Random Access Memory (PRAM), Byte-Addressable Non-Volatile Memory, Persistent Dual In-line Memory Module (Persistent DIMM), or Storage Class Memory (SCM), though recent terminology refers to SCM as extended DRAM with PM for larger, fast RAM.

1.3.1 Persistent Memory and Big Data

Big Data has recently become a popular term used to describe the size and variety of the growing data sets used by organizations worldwide. Initially characterized
by three components: Volume, Velocity, and Variety [22] [23], Big Data now often encompasses seven V’s like Value, Veracity, Variability, and Visualization [24] as well. The emergence of new, large capacity, fast, and durable memory technologies dovetails perfectly with the requirements of managing Big Data. PM (see Figure 1.1), helps solve some of the Big Data challenges:

- **Large Capacity** - Peta-byte sized PM volumes using Phase Change Memory or PCM, allow for more data to be processed per node and closer to the processor.

- **Persistence** - Non-volatility of PM permits interrupt-free and streamlined computation as threads do not need to schedule or serialize memory objects for IO.

- **Byte Addressability** - Data structures and algorithms can be optimized for cache-friendly, fine-grained memory access to both structured and unstructured data.

- **Speed** - PM will be offered in DIMMs that fit alongside DRAM on the high-speed main-memory bus.

### 1.4 Hardware Transactional Memory Overview

Computer processor manufactures have moved from improving single core processors to adding more cores to multi-core processors. These pervasive multi-core computer architectures have given rise to programmers utilizing these cores for application performance improvements. However, these cores have the potential to update the same data structure simultaneously. Therefore, to avoid data corruption and preserve data integrity, programmers are forced to utilize expensive locking techniques to ensure only one update is performed at a time.

Consider the following example of an accounting system which manages accounts, in this case accounts W, X, Y, and Z. Several example monetary transfers, A, B,
and C, are shown in Table 1.1. In a parallel processing environment, multiple transfers involving different combinations of accounts may be executing concurrently. It is imperative that concurrent transfers, as in this example, do not operate or transfer funds from the same account resource simultaneously, as the account may then be invalid. Consequently, programmers must utilize some exclusive access mechanism, like locking techniques, to ensure updates to accounts maintain validity across execution.

Locking techniques require programmers to edit software code, create software locks, and ensure that the locks are taken in a specific order to avoid deadlock. A coarse-grained approach taking a single global lock is much easier to implement but locks out other concurrent operations that might be updating entirely different sets of data, thus sacrificing application performance. A coarse-grained lock might be taken at the Begin and released at the End statement, thus preventing all other concurrent accounting updates, even those that do not update similar accounts. Alternatively, a fine-grained locking approach divides data into subsets, where each subset can have its own lock. With more locks, other threads of execution can update unrelated data sets simultaneously, thus increasing performance through application parallelism.

In some situations, an update may require multiple data elements and many separate locks. These locks must then be taken and released in a specific order. If another
operation or reservation takes the same locks in a different order, a deadlock situation can occur. Consider a transfer D, which updates accounts Z and W. If D tries to lock Z before W, and C has a lock on W, waiting on Z, then deadlock between transfer C and D can occur. With fine-grained locking, partitioning data around specific locks and ensuring the locks are taken in a specific order is extremely challenging and prone to error.

Another important property of the accounting transfer is that the transfer needs to happen entirely or not at all. If transfer A only completes the update to W, then the accounting system is out of balance. We refer to this as an atomic update, and we refer to the transfers as transactions. Additional properties are discussed in Chapter 2.

Taking a software lock on a section of data requires processing time and resources. With very large data sets, fine-grained locking storage requires even more space, while the probability of locking collisions, threads of execution operating on the same data, decrease. While the numbers of locks increase, the complexity and number of locks an application updating multiple data elements also increases. Software Transactional Memory systems or STMs attempt to perform fine grain locking with the ease of only instrumenting loads and stores, by backing out of transactions that might deadlock, at the cost losing some performance. STMs are discussed in more detail in Chapter 2.4.2.

The timely release of a new hardware mechanism to handle locking offers the best of both worlds, as it simplifies programming and provides high performance similar to fine-grained locking. This hardware mechanism, called Hardware Transactional Memory or HTM, allows programers to program locks with ease, similar to taking a simple, global protection lock around shared data sections without having to worry about deadlock and fine-grained lock order. The processor hardware manages the fine-grained locking behind the scenes using well established protocols built upon the dynamic cache hierarchy. When utilized, HTM allows for high performance, concur-
rent applications and data structures. HTM as a concurrency control mechanism is discussed in more detail in Chapter 2.4.3.

HTM was designed for processor attached volatile main memory, and is not compatible as a persistent control mechanism for disk resident data. Nevertheless, emerging Persistent Memory has brought persistence into the processor-managed domain and amenable to HTM control similar to the control of DRAM. However, there exist several difficult challenges for Hardware Transactional Memory controlled Persistent Memory.

1.5 HTM + PM Overview

Historically, high performance multi-core processors have been designed for dynamic operations, such as high throughput arithmetic operations and updating data in DRAM. These processors contain large dynamic data caches and buffers, all of which can be lost on a system or power failure. The data caches and processor hardware buffers are unmanaged, autonomous subsystems and out of software control. To protect data in these dynamic areas from loss on failure, durability has been heavily managed by software executing on these processors to transfer data from the memory domain to the persistent disk domain.

For instance, in-memory data structures typically persist data at some granularity or slow or halt concurrent operations to create snapshots or checkpoints of the data structure in memory by creating a copy of the data to persistent disk. Additionally, many in-memory data structures incorporate software locking techniques to make use of high performance multi-core processors. However, these structures have been slow to adopt the emerging HTM as a concurrency control mechanism, creating a window of opportunity for new concurrent data structure implementations.

Emerging Persistent Memory has brought persistence into the processor-managed domain and amenable to HTM control similar to the control of DRAM. As used for dynamic memory, the use of HTM as a control and isolation mechanism for trans-
actions to PM seems a natural fit. However, HTM provides a challenging scenario for PM transactions. The HTM mechanism, built on established cache-coherence protocols for dynamic, unmanaged memory, aborts a transaction when data flow to memory is managed. Persistence management operations inside HTM sections to ensure data is correctly ordered for PM therefore aborts the transactions.

Additional problems occur when persisting data outside of an HTM transaction. Any processor delay could postpone the saving of a prior, dependent transaction or include only partial updates of completed transactions. For instance, in the example from Table 1.1 if transaction transfers occurred in order A, B, and C, and B was delayed in saving state or persisting values entirely to back-end PM, then any failure may see values from C and only partial updates from B. These challenges are described in detail in Chapter 3.

This thesis introduces Hardware Transactional Persistent Memory or HTPM. HTPM couples HTM for concurrency isolation control with PM for durability without any hardware changes, while maintaining fast execution speed and preserving PM consistency across any possible system failure. Furthermore, this thesis introduces relaxed durability, whereby HTM operations proceed in the foreground and updates to PM are made durably in the background in a persistent wavefront that may lag foreground cache operations.

Some solutions [25, 26, 27, 28] advocate changing the established HTM semantics to support persistent transactions in HTM. However, this can introduce new complexities in the processor that are not known or understood fully, as implementations of HTM even with dynamic memory have proven difficult. The solutions presented in this thesis run on existing Intel processors and persistent memory modules without any changes to the cache hierarchy or their protocols.
1.6 System Architecture Overview

Recent advances in multi-core computing have moved from just one to two cores on a single chip to hundreds of cores on a single chip. Hundreds of cores allows for highly-parallelized concurrent applications to exhibit considerable performance gains. When PM is coupled with multiple cores, the result can support highly parallelized, in-memory applications that no longer have to persist data to a slow, block-based device. This multiple core architecture is coupled with Persistent Memory alongside DRAM on the main-memory bus as shown in Figure 1.2.

The Intel Many Integrated Core Architecture (Intel® MIC Architecture) includes the new Xeon Phi™ processor. Two examples of the Intel® Xeon Phi™ processor are the 7210 Processor with 64 cores and the more recent 7290F Processor which contains 72 cores. Each Intel® Xeon Phi™ processor core supports 4-way Intel® Hyper-Threading, yielding a usable 256 and 288 cores on the 7210 and 7290F processors respectively. Both processors contain 64GB of cache along with the cores on a single 14 nm processor die [29]. Each of these processors supports 6 memory channels and supports over 100 GB/s in memory bandwidth to over 384 GB of addressable memory, allowing for high performance in-memory computation. However, these newer processors trade more cores at the expense of some features such as 1) a limited cache size of 32 MB L2 only cache, 2) no Intel® Virtualization Technology Extensions (VT-x), such as no support for Directed I/O (VT-d) or Extended Page Tables (EPT), and 3) no support for Hardware Transactional Memory.

Other Intel® processors such as the Intel® Xeon® Processor E5 v4 Family support Intel® Transactional Synchronization Extensions New Instructions (Intel® TSX-NI), which are instructions that support Hardware Transactional Memory for synchronization lock removal and parallel operations. These instructions replace expensive synchronization locks used by multi-threaded programs used for concurrent access to program variables with a single hardware instruction. Using the instruction signals the hardware to perform fine-grained transactional locking at the cache-line level.
The Intel® Xeon® Processor E5-2686 v4 (Broadwell Microarchitecture) is a High-Core Count (HCC) processor, which contains 18 cores and supports 2-way Hyper-Threading on 14nm technology, with 7.2 billion transistors and die size of 456.12 mm$^2$. This processor supports not only Hardware Transactional Memory but also contains a 45 MB L3 cache and 4 memory channels that support a total memory size of 1.54 TB with high memory bandwidth of over 70 GB/s.

Figure 1.2: Logical layout of an Intel E5 Series High-Core Count processor that supports HTM transactions with 18 cores and two memory controllers with both DRAM and PM.

When these advanced processors are combined with Persistent Memory, the result is a system that can support highly parallelized access to fast, byte-addressable persistent storage. This architecture can support concurrent access to terabytes of persistent data. Figure 1.2 shows a logical diagram of the layout of an Intel® E5-2686 v4 Processor that has 18 cores and integrated memory controllers. Both Dynamic Random Access Memory, DRAM, and Persistent Memory, PM, are shown in the figure as byte-addressable main-memory. The cache system is partitioned among the
Figure 1.3: Logical layout of two Intel Xeon processors on a motherboard with a total of four memory controllers accessing both DRAM and PM with HTM support. Processors and cores have shared and cache-coherent access to all memory locations.

18 cores as follows: separate 32 KB 8-way set associative instruction and data Level 1 caches, Level 2 cache size 256 KB 8-way set associative caches, and Level 3 cache size of 2.5 MB per core and 45 MB total. Hardware Transactional Memory support is contained within the processor and operates on top of the cache-coherence protocol. The Intel® Xeon® series of processors utilize a variant of the MESIF (Modified, Exclusive, Shared, Invalid, and Forward) cache-coherence protocol.

The Xeon® family of processors can also be included on a multi-processor motherboard node, such as two E5-2686 v4 processors on a single motherboard. When two of these processors are used in a single motherboard or system node, the capacity of the system is greatly enhanced. An example system is shown in Figure 1.3 where two
processors can access PM and DRAM. In this configuration, a memory access by one processor A might take longer to fetch a word of memory if that word is contained on a memory controller on the adjacent processor socket B than compared to if the memory word was on a memory controller on the processor A. This is referred to as Non-Uniform Memory Access or NUMA. Full Hardware Transactional Memory support crosses system processors, safely ordering transactions that update shared data across the combined system.

1.7 Contributions

Leveraging Hardware Transactional Memory for concurrency control while guaranteeing the durability of transactions to Persistent Memory is not straightforward. Recent research has stated that to create durable HTM transactions to PM require major changes to the semantics of the transaction isolation and HTM mechanisms [25, 26, 27, 28]. HTM implementation has proved difficult in of itself as Intel® rolled back implementations of HTM in the first release late in the production cycle and even after shipping most Haswell and Broadwell cores [30, 31]. Adding additional requirements to caching and rolling back decades of work on established HTM and cache-control mechanisms is difficult.

Prior solutions for adding isolation mechanisms to PM transactions include two-phase locking [32, 33], utilizing software in an STM [20], or specialized data structures such as a persistent lock-free queue [34]. This thesis is the first work to couple HTM with PM to create high-performance, concurrent, durable transactions to PM without any changes to hardware or front-end cache or coherency mechanisms. This thesis makes the following contributions:

- **Ordering HTM Transactions for PM:** We show that HTM transactions can be ordered using existing instructions without any hardware changes. Processor delays after a transaction has started or completed a concurrency section or a delay before persistence to PM leads to complicated ordering scenarios.
A completed transaction might depend on another in-flight transaction that has completed an HTM section but before values have been persisted in PM. Other solutions require significant changes to the established HTM mechanisms \cite{25, 26, 27, 28} to allow for shared counters and multiple operations to occur on transaction completion.

- **Persisting HTM Transactions to PM:** Leveraging HTM for transaction isolation, we create a method for persisting PM transactions. Our method uses aliasing and redo-logging for durability and atomicity, and persists completed transactions after ordering. Persisting transactions using HTM for isolation is challenging since HTM aborts any persistence operation within its transaction boundary. Cache evictions after a HTM transaction completes or failure before transactional values are completely persisted after the boundary can leave partial transaction updates on PM. Other solutions require extensive changes to the established processor cache-coherency mechanisms to allow values to flow to PM within HTM \cite{25, 26, 27, 28}.

- **Strict and Relaxed Durability:** We introduce a method that allows for transactions to trade performance for durability guarantees. A transaction, even though having persisted or logged values, might have dependencies on transactions that have not completely persisted all the way to PM and are only in the cache hierarchy due to processing delays. *Strict Durability* is what we call transactions that need to complete and wait until a guarantee that the transaction will be replayed if a failure were to occur. We introduce *Relaxed Durability* where transactions can complete and know that they will be strictly durable at some point (shortly) in the future and that their values, even though not strictly durable, will not corrupt any other transactions or the system. We show that Relaxed Durability can have large performance benefits. No other solutions offer a Relaxed Durability option for transactions.
• **Software-Only Implementation:** This thesis is the first to develop a working implementation of Durable HTM transactions to PM using existing Intel® hardware without any changes or proposed changes to hardware. Additionally, the software is implemented using a novel and efficient, lock-free data structure with bounded memory space. The software builds upon an established and state of the art software aliasing method \cite{35,32} to catch cache-evictions after an HTM section. The statically bounded memory space allows for optimizations such as lock-free meta-data management of transactions and implementation of strict and relaxed durability. No other solution operates on existing commercial processors and hardware.

• **Back-end PM Controller Implementation:** This thesis introduces a back-end PM controller that, when coupled with a lightweight software library, effectively allows HTM transactions to utilize PM for persistence. The controller may be implemented on processor or motherboard, or entirely on a back-end PM device as it is decoupled from other controllers and scales independent of the number or existence of other controllers. A previous controller \cite{36,37,33} did not work with HTM and required locking for concurrency control. Our improvements on this controller not only introduce HTM support, but also simplifies the controller algorithm and alleviates performance bottlenecks such as required log processing. We evaluate the hardware controller performance and buffer sizes using a full micro-architecture level simulator using extensive testing. We implemented the user-level library and show that it runs on existing Intel® hardware. Other solutions require changes to the processor itself \cite{25,26,27,28} which might take a years to realize since just the HTM proposal \cite{3} took over a decade and a half to successfully reach production processors \cite{38}.

• **Full Evaluation on Existing Hardware:** We empirically validate our solu-
tion on existing Intel hardware. We evaluated our solutions using both micro-
benchmarks and benchmarks from the Stanford Transactional Applications for
Multi-Processing [39], or STAMP, benchmark suite. We compare our solutions
to volatile cache-based HTM concurrency and a Persistent Transactional Lock-
ing implementation by extending the technique of [40] for persistence. We show
that our approaches compare well with standard (volatile) HTM transactions,
and that they yield significant gains in throughput and latency in comparison
with Persistent Transactional Locking.

- **Extended PM Case Studies:** We present an extended case study on PM used
  as an in-memory data-structure store utilizing Redis, an in-memory data-structure
  store [41]. Additionally, Docker Containers [41] present an interesting challenge
  for PM for extending isolation mechanisms. We present a new method to access
  PM from within a container for virtualized access to PM.

### 1.8 Thesis Organization

The remainder of this thesis is organized as follows. In Chapter 2 Persistent Mem-
ory transactions are discussed. Related work and various approaches to persisting
transactions to PM are presented and discussed in detail. Additionally, approaches
for providing isolation to concurrent transactions is discussed along with related work.

Chapter 3 introduces Hardware Transactional Memory, or HTM, as a concurrency
control isolation mechanism for transactional persistence of data structures to Persis-
tent Memory. HTM, which was designed for use with volatile memory, is discussed
in detail. The problems with using HTM as an isolation mechanism for concurrent
transactions to PM are detailed. A novel conceptual solution for persisting HTM
transactions to PM is introduced and a detailed example is shown.

Next, in Chapter 4 a software only based solution that effectively utilizes HTM
for concurrent transactions to PM is presented. This solution relies on no additional
hardware changes to the processor, cache hierarchy or memory controller and can execute on existing computer systems. The solution is presented as a software library with lock-free data structures and algorithms. A full evaluation of the software is performed on multi-core architectures using the latest concurrency benchmarks, the STAMP [39] benchmark suite.

In Chapter 5, a software-hardware architecture based on a back-end Persistent Memory Controller solution is discussed and evaluated. This solution is a technique for preventing unwanted cache evictions from corrupting PM before the end of a transaction. A simulation based evaluation of the architecture is presented which shows efficiency of the algorithm and that the software front-end can execute on existing Intel® hardware. A full evaluation of the front-end software is performed on existing Intel hardware. The evaluation which shows the method performs nearly as fast as HTM cache based transactions, gaining persistence for extremely low cost.

In the Appendix we show several important contributions. First, we present a modification to Redis [4], a popular in-memory data structure store. The modification allows for Redis to transactionally update PM data directly without utilizing file based accesses. We discuss and evaluate a novel transaction update mechanism that combines benefits from various logging techniques. Next, we present a mechanism for accessing PM in containers, specifically Docker [41], that allows for direct memory accesses while providing the security and portability required by virtual machines. This solution is presented along with performance evaluations that show an order of magnitude speedup over Docker based accesses. Finally, we evaluate our solutions on an early-access, invitation only machine provided by Intel. The multi-core HTM machine is equipped with unreleased PM modules. Our results show data trends in our initial evaluation are validated by the early-access platform.

Finally, in Chapter 6, conclusions from both methods are presented. A detailed discussion on areas for future work is also presented.
Chapter 2

Persistent Memory Transactions

This thesis makes a sharp distinction between traditional applications using file or block based persistence and new applications that persist data structures directly in Persistent Memory safely. This chapter discusses the problems in shifting from these traditional systems to systems that use processor attached, byte-addressable Persistent Memory. This chapter shows how Persistent Memory can be accessed and utilized in a fail-safe, durable manner for in-memory data structures. Solutions for updating Persistent Memory are discussed along with the state of related work.

This chapter also discusses the challenges associated with concurrent access to Persistent Memory. Several solutions for accessing Persistent Memory concurrently are discussed. Leveraging multi-core architectures for concurrent access to PM will allow for high performance, persistent data structures.

2.1 Persistent Memory Access

Traditional applications access persistent data residing on a block-based hard-disk drive or solid-state disk using a file system interface. This access is shown on the left side in Figure 2.1. Once initialized, data may be copied into the user application’s data space and accessed using loads and stores, as shown in the black hashed lines. However, this approach faces two problems as data is 1) duplicated in page caches backed by dynamic memory and 2) updated in blocks even for a small or single byte write. Persistent Memory has the potential to solve these two problems through direct access.

Various types of PM and PM placement on the main-memory bus in the system
architecture were discussed in Chapter 1.6. Accessing Persistent Memory is fairly straightforward. Developed by Intel® PMFS [42] and more recently PMDK [43], provide a mechanisms to easily manage and access PM located on processor attached memory slots. Persistent Memory may be accessed through a traditional file system interface or may be mapped into the virtual memory address space of a program [44]. Figure 2.1 shows how PM may be accessed through a traditional file system interface or accessed through a driver. PM may then be mapped into memory and accessed directly using normal loads and stores, as shown in the black hashed lines on the right in Figure 2.1.

![Figure 2.1](image)

Figure 2.1 : Persistent Memory can be accessed directly using normal loads and stores, bypassing the file system interface and page caches.

In Linux, new Linux kernel Direct Access (DAX) support and eXecute In Place (XIP) support allow user applications to access block based devices and non-volatile devices directly [45] [46]. This direct access bypasses the Virtual File System interfaces and operating system internal page caches. Bypassing internal page caches creates a direct conduit to Persistent Memory without the need to copy data into internal
operating system software buffers. Internal page caches no longer need to be filled with an entire data block (typically 4KB) to load or store a single byte of information. This new DAX support is added in file systems such as ext2 and ext4.

Several general purpose persistent memory file systems built on Persistent Memory have been proposed that can allow easy adoption of Persistent Memory. In addition to PMFS [42], these include BPFS [18], SCMFS [47], Aerie [48], and NOVA [49]. Since these are general purpose file systems, they can support existing applications that are built to persist data to block devices. However, applications using file-based persistence are subject to file system guarantees. Saving data into a file might still fail, and therefore applications must build additional protections on top of the file system layer, such as database log files. This additional protection may result in multiple layers of logging to preserve the consistency of the data if a failure were to occur. Logging to preserve data integrity in case of failure is compounded for virtualization technologies, such as containers, which use multiple layers of file accesses. In Appendix B we show how cascaded copying for PM consistency in containers can be removed. This thesis utilizes new DAX support for PM, thereby allowing loads and stores to operate on persistent, high performance in-memory data-structures.

2.1.1 Persistent Memory Access Transaction Properties

For a single operation on a data structure, such as inserting an element into a binary tree, multiple individual elements may be updated or accessed. These updates must be preformed as a single operation to preserve the integrity of the data structure. Section 1.4 showed a transaction as a group of operations that should perform as a unit. Here we define a transaction as follows:

Definition 2.1 (Transaction). A transaction refers to a program unit that is executed atomically without interference from other transactions that may access the same data.

To preserve data integrity, we would like transactions to have several properties, known as ACID properties [50]:
• **Atomicity**: A transaction behaves as if it has executed completely or not at all. Multiple operations occur as a group. Atomicity for PM is discussed in Section 2.3.

• **Consistency**: Global invariants of the system state are preserved in the presence of updates. Applications utilizing transactional statements correctly, will move from one consistent state of the data structure to the next, even if a failure were to occur. This thesis examines applications and develops micro-benchmarks that utilize transactions correctly, e.g. with Begin and End statements around data sections, to maintain application consistency. Consistency is therefore guaranteed by ordering and replaying transactions in order upon failure.

• **Isolation**: A transaction may not be affected by other transactions executing concurrently. Transactions may share reads of data, but transactions should not write the same data or attempt to read in-flight data. This thesis adapts a *Serializable Isolation* policy, whereby transactions can be replayed in-order, serialized, to create a consistent view of data and reads will not return data for in-flight transactions, e.g. those that have not completed. Isolation for PM is discussed in Section 2.4.

• **Durability**: The updates made by a committed transaction are permanent on non-volatile media. Individual updates to PM must also be ensured to flow to durable PM. Durability is discussed in Section 2.2.

Similar to block-based systems, a PM transaction, executed by an application such as an accounting system, moves a data structure from one *Consistent* state to another. The following sections discuss the *Durability*, *Atomicity*, and *Isolation* challenges in providing ACID properties for PM transactions.
2.2 Durability for Persistent Memory

Storing even a single value into Persistent Memory is not straightforward. Programmers historically save data to block-based disks explicitly and use processor memory for temporary storage operations. This temporary storage may never reach the main system memory and simply reside in the processor cache. Utilizing computer memory for persistence, changes the view and methods for achieving durability for data structures.

A system with a persistent, or non-volatile, PM DIMM attached to a memory bus alongside a volatile DRAM DIMM is shown in Figure 2.2. The Figure shows the complexity of the location of data values, as the most recent values may be in any of the caches or buffers in either core.

![Diagram: Writes to persistent memory from a core require moving through multiple buffers, complicating persistence.](image)

Consider a store or write to just a single memory-based variable \(x\). The latest value of variable \(x\) might actually never flow to memory, DRAM or PM, even after the instruction, or application program for that matter, completes. Instead, the value...
may be caught in a number of places such as the write buffer to the cache, the cache hierarchy, or various buffers. Additionally, the value may be evicted or written back from the cache, and located in either main memory or a transient memory buffer. Even when using a locking technique, such as a concurrency technique described in Chapter 1.4, the new value of \( x \) should be visible and accessible to other threads. However, the method by which to obtain this new value by threads might vary depending on the hardware, as the actual location of the variable \( x \) is not guaranteed to be in main memory and may likely reside in caches.

For DRAM based transient variables, if a power failure occurs, being located anywhere in the hardware is not problematic as the variable value is cleared on restart. However, for persistent variables where the value is located in Persistent Memory and a power failure occurs, the result of the computation may not be committed to and made durable on Persistent Memory. Even if a value is written to a memory location using a cache-line flush, \( CLFLUSH \), to force the value out of the cache into main memory, it still may be buffered in the lower buffer of Figure 2.2 and not stored persistently. Therefore, even flushing values during or after a transaction does not guarantee the values are persisted on Persistent Memory reliably. Additionally, \( CLFLUSH \) has the unfortunate side effect of also invalidating the entire cache line. Invalidation means the entire cache line will have to be re-read from slower main memory on a subsequent load or store to the cache line.

The new Intel architecture specification [51] specifies several new instructions including \( CLFLUSHOPT \) and \( CLWB \) [52]. \( CLWB \), or cache-line write-back, writes a cache line out to the write buffers but doesn’t invalidate the cache line. \( CLFLUSHOPT \), is like \( CLFLUSH \), but can be optimized by the Intel® hardware [53], as it is more weakly ordered. Both \( CLFLUSH \) and \( CLFLUSHOPT \) are ordered with respect to memory fences and memory stores to the same cache lines and locked read-modify-write instructions. However, even with flushing, cache-line write-backs, or write-through stores, a value might still be in the final memory write
buffer and not safely persisted on Persistent Memory.

Another instruction \textit{PCOMMIT}, or persistent memory commit, was introduced which solved this problem by making all visible memory stores complete or persist to their home locations. However, this instruction has already been depreciated by Intel\textsuperscript{{\textregistered}} and should not be used \[54\]. \textit{PCOMMIT} required the use of a prior \textit{SFENCE} to make prior stores visible and enforce a strong instruction ordering. Once the \textit{PCOMMIT} instruction was retired, then all prior globally-visible stores were guaranteed to have persisted. Additionally, since \textit{PCOMMIT} was weakly ordered, a final \textit{SFENCE} was required so that subsequent instructions depending on the values being persisted to Persistent Memory were not reordered before the \textit{PCOMMIT}. Due to the necessity of \textit{SFENCE} with \textit{PCOMMIT}, Intel\textsuperscript{{\textregistered}} depreciated \textit{PCOMMIT} and made the \textit{SFENCE} instruction perform the work of the combination. That is, once the \textit{SFENCE} instruction has retired, then prior visible PM stores in store buffers can be assumed to be committed to Persistent Memory.

\subsection*{2.2.1 Persistent Memory Models}

Persistent Memory complicates traditional memory models developed for dynamic memory consistency. Dynamic memory consistency has been a widely debated and discussed topic for the past several decades \[55, 56, 57, 58, 59\]. These models for dynamic memory were developed to show expectations for variable visibility and ordering between variable writes and between cores, such as those shown in Figure 2.2. However, a snapshot of memory contents on a memory module attached to the main-memory bus may be subject to 1) the semantics of the cache coherence protocol of the underlying system, 2) the memory consistency model of the system, and 3) application level behavior that includes language consistency guarantees, spurious cache evictions, forced cache evictions, and non-temporal or streaming stores.

As the contents of dynamic memory and memory consistency are lost and reinitialized after a system crash, established models were not concerned with con-
sistency after failure. Some work, such as Memory Persistency [60] analyze the consistency models for Persistent Memory. It presents two classes of models, strict persistency and relaxed persistency, whereby an observer can see memory operations is program order or out-of-order respectively. Other work presents Acquire-Release Persistency, [61] or ARP, which is a language-level persistency model developed for C++11. ARP is based on epoch-ordered persist, whereby a compiler can reorder writes to individual memory locations within an epoch, but not across epochs.

Our work is based on Intel® Persistent Memory ordering of stores and a Serializable Isolation level for concurrent transactions. Fortunately, Intel® architecture performs stores in First-In-First-Out (FIFO) order, commonly called Total Store Order or TSO [62]. Additionally, groups of stores will be visible in order and we persist the entire group of stores atomically as described in the following Section.

2.3 Atomicity for Persistent Memory Transactions

As described in the previous section, the new Intel® instructions can be utilized to ensure data is durably written out to Persistent Memory. However, as described in the accounting system example in Chapter 1.4 and Table 1.1, groups of stores in a single transaction need to be persisted atomically, all or none. These transactions must utilize additional techniques to ensure the entire group of operations is atomic. Without some type of mechanism to guarantee atomicity to PM, any failure may cause an inconsistency of the durable data. This section discusses several types of approaches that can be implemented in existing hardware systems. There are additional proposed hardware mechanisms that detail methods to achieve transactional atomicity to Persistent Memory. These proposals are discussed in Section 2.5.

Table 2.1 shows a sequence of stores for several transactions that must be performed atomically, all or none, to provide consistency by avoiding corrupting an in-memory data structure in case a failure were to occur. Updates to the persistent variables $x$, $y$, or $z$ are updated along with $w$ contained within Begin and End
Table 2.1: Transactions A, B, and C for variables w, x, y, and z to PM statements inside individual functions. After the stores in transaction A have executed, assume the values in Persistent Memory for w, x, y, and z are [1, 1, 0, 0]. In transaction B, w=w+1; and y=w, must be performed as a single group of stores.

Consider Figure 2.3, where during transaction B, before completion of the routine at End, a cache eviction of a line containing variable y occurs, followed by a subsequent system crash. This leaves Persistent Memory in an inconsistent state [1,1,2,0] as shown in the figure and not in either [1,1,0,0] (the state prior to the failure) or state [2,1,2,0] the state after B completes.

Forcing values directly through the cache can leave persistent memory in an inconsistent state as well. Suppose that w was forced through the cache hierarchy and updated to the value 2. If a failure were to occur before y was updated, then we would still have an inconsistent state [2,1,0,0].

There are multiple methods to guarantee atomicity for transaction from hardware support to well known software approaches. These methods all employ some type of logging mechanism to roll back to a previous version should a failure were to occur. This topic was discussed in detail in [63] where several new approaches were presented. In Section 2.5, several persistent memory approaches are discussed, such as specialized data structures and hardware support for atomicity of transactions. The following subsections describe several traditional logging or versioning approaches for transactional atomicity. In these sections, we refer to a log as a group of log records,
2.3.1 Undo Logging

An Undo Log approach involves a synchronous operation which first makes a persistent copy of a value before writing a new value. Before each write however, a value must be made persistent on the underlying medium. In PM, this can be accomplished by creating a log, adding a log entry of the address of the variable and original value of the variable to the log, followed by an SFENCE, then writing the new value of the variable to memory. The writes to the log may be written to cache followed by a CLWB, or may be streamed using streaming stores. Once new values are written to the original variable, the values must be flushed (if they haven’t been streamed) to home locations and committed before the log can be removed. A persistent SFENCE between writing a new value before saving an old value cannot be
removed or the ordering for the write to the main persistent memory location may be caught in the final store buffer in Figure 2.2.

Consider the transaction shown in Figure 2.4. Before a variable is updated to a new value in a PM transaction using an Undo Log, the old value is persisted to a log. The address of \( w \) and old value of 1 are saved to a persistent log before \( w \) is updated to 2. Similarly, \( y \) is logged before being updated with \( y=2 \). However, if a crash were to occur before the transaction end, the Undo Log, as shown in the Figure, contains the old values \([w=1, y=0]\). Even if PM contained an inconsistent state, such as the state \([1,1,2,0]\) due to a cache eviction as mentioned above, a recovery mechanism plays the Undo Log and the state is returned to the original state before the transaction, \([1,1,0,0]\).

Undo Logging solutions include PMDK [43], ATOM [64], ATLAS [65], and others. These solutions are discussed further in Section 2.5.
2.3.2  Block Copy-on-Write

Similar to Undo Logging, a Block Copy-on-Write method must first copy existing data to another area before new data is written. However, with Block Copy-on-Write new data is written into data copies until the transaction is complete. Once the transaction is complete, the new data may be swapped for the old data. Block Copy-on-Write may copy data at different granularities instead of individual variables. The efficiency of the copying technique would depend on the data structure and application. For instance, data may be copied at cache-line (64 bytes) or even page (4 kilobytes) granularities.

In block or page copies, the block or page is copied to a new area, and a new value is written into the area. Then, with an atomic pointer switch, the new block or page is swapped as a replacement for the old data. This method is fast on reconstructing reads, but may suffer from extra writes if only updating a small amount of the block or page. BPFS \[18\] uses a copy technique to copy sub-trees of updated data.

2.3.3  Redo Logging

Similar to Undo Logging, a Redo Logging solution also creates a log. However, unlike Undo Logging which writes new values in-place after logging, a Redo Log writes new values out-of-place before writing the new values to the transaction variables. This out-of-place write can be the log, shadow update area, or combination of the two. A Redo Log approach may be faster than an Undo Log in write heavy workloads, but for reads, the out-of-place write area must be consulted to fetch a possible new value of the variable. If the variable isn’t present in this area, then the variable value must be fetched from persistent memory. Expensive SFENCE operations do not need to be performed until the end of a transaction as it is ok to lose the log if a failure occurs. After the new values have been safely persisted to the log, the new values can be written to home locations immediately or delayed until a future point.

On transaction reads, the out-of-place update area, such as the log, must be
consulted to retrieve the most recent value. One optimization is to create a shadow area for out-of-place updates, which can be in DRAM for speed. A shadow area, as shown in Figure 2.5, may be located in volatile DRAM. If a cache eviction to DRAM of a transient value happens during a transaction, it causes no harm to the persistent values if an unexpected crash occurs.

Consider the transaction shown in Figure 2.5. Suppose transaction B was updating \( w \) and \( y \), it would actually be updating DRAM variables \( w' \) and \( y' \) respectively. First, the transaction must read \( w \) to perform \( w = w + 1 \), however, it reads \( w' \) the DRAM shadow copy instead. The transaction then logs what it will do, e.g. update \( w = 2 \), to a Redo Log in persistent memory and then saves \( w' = 2 \) in the shadow DRAM. Similarly, for \( y \), the new value of 2 is logged and \( y' = 2 \) is updated in DRAM. If a cache
eviction of any variable, like $y$ or $y'$ occurs there is no problem. The variable $y'$ is located in DRAM, so any crash will not corrupt the persistent data structure, and the value of $y$ is still the original value of zero before the transaction. If a crash occurs before completion, the log can simply be removed. After the transaction completes, the log is marked as complete and values may be copied from the log or saved from the shadow locations. Before the log can be removed, the values must be updated in persistent memory.

An efficient shadow version which uses a DRAM alias table was presented in [35, 63] and a multi-threaded concurrent version in SoftWrAP [32]. NV-heaps [66], REWIND [67], WrAP [36, 37, 33], and HAPT [68] (a hardware based version of SoftWrAP [32]) also use Redo Logging. These Redo Logging solutions are discussed in Section 2.5.

### 2.3.4 Versioning

Versioning based mechanisms may use a variety of logging, copying, or shadow techniques. A version or snapshot guarantees some subset of completed transactions is durable on and recoverable from a persistent medium. In this method, the most recent completed transactions may be lost, but consistency is guaranteed up until some completed prior transaction.

Consider a sequence of transactions from Table 2.1, such as $A \rightarrow B \rightarrow C$, whereby the cached memory state is $[3,1,2,3]$ at completion. A versioning solution might use a combined Undo or Redo Log to group transactions A and B in Version 1 and transaction C in Version 2. If Version 2 remains open while there is a failure, even if transaction C completed successfully, then the system would return to the state after Version 1, e.g. $A \rightarrow B$ or $[2,1,2,0]$ and not $[3,1,2,3]$.

A version may persist a state of a data object, set of objects, or system state. Dalí [69] keeps versions of a key-value store data structure in PM and can rollback to a previous version on failure. In database systems, a version or checkpoint may
be applied after a certain number of transactions have completed and saved their contents into a Redo or Write Ahead Log. The contents of the log are then marked as closed and copied into the main database. Redis \[4\] uses a versioning mechanism for transaction updates into the data-structure store. It can be configured to save versions after a configurable number of transactions have completed or at a certain time interval.

2.3.5 Summary

There are numerous methods to perform groups of atomic writes. Table 2.2 summarizes the differences in the various methods. The number of SFENCE operations is also shown in the table for a transaction of \( n \) writes. Undo Logging requires at least one SFENCE between saving the old value to a log and writing the new value. Redo Logging only requires a fence after the log is persisted and after new values are written. Block copying is similar in that it only needs one persistent fence after the block is updated and one after the swap of the block. For versioning, a version could be made very large to reduce fences, however, if a failure were to occur, a rollback would have to be to a state far in the past, losing potentially many thousands of transactions.

Table 2.2: Comparison of atomic update methods for a transaction of \( n \) variable writes

<table>
<thead>
<tr>
<th>Method</th>
<th>Advantage</th>
<th>Disadvantage</th>
<th>SFENCEs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Undo Logging</td>
<td>Fast reads, performed in place</td>
<td>Writes are slow and synchronous to log</td>
<td>( n+1 )</td>
</tr>
<tr>
<td>Redo Logging</td>
<td>Fast writes, write combining log</td>
<td>Reads must be rebuilt from write set</td>
<td>2</td>
</tr>
<tr>
<td>Block Copy-on-Write</td>
<td>Write combining blocks and atomic swap</td>
<td>Large number of unnecessary writes</td>
<td>2</td>
</tr>
<tr>
<td>Versioning</td>
<td>Can combine Undo/Redo Logging</td>
<td>Data loss, failure rolls back to some prior version</td>
<td>2 or more per version</td>
</tr>
</tbody>
</table>
There is no single method that works the best under all workloads. Therefore, applications may choose the best atomicity method based on the data structure and expected workload. A study on Persistent Memory Transactions [70] performance using the SoftWrAP [63] micro-benchmark shows no single method is best. Additionally, methods may also be combined. For instance, one may combine Undo and Redo logging, commonly used in database systems [71], to gain speed of reads with the low cost of writes. This method was explored for PM in MARS [72] and high-performance Big Data operations using Redis [73] and presented in Appendix A.

2.4 Isolation for Concurrent Persistent Memory Transactions

This section describes several isolation techniques to ensure that concurrent transactions read and write valid data. Serializable Isolation is a strict policy that guarantees a transaction is only visible to other transactions once the transaction is complete. This provides strict consistency as transactions can move a durable data structure from one consistent state to another.

Referring to Table 2.1 if transactions A, B, and C are operating concurrently, then, without any isolation control, they can each update w simultaneously. However, simultaneously updating a shared variable can lead to data inconsistency. For instance, suppose that both transactions A and B load the shared variable and value w=0 concurrently. Then, A stores w=1 and x=1 and B stores w=1 and y=1. This is an invalid state of the system with [w, x, y, z] = [1, 1, 1, 0]. A valid state would be either [2, 2, 1, 0] or [2, 1, 2, 0]. To ensure consistent data, we want to prevent conflicting transactions from updating common data concurrently.

**Definition 2.2 (Conflict).** Two transactions are defined to conflict if they both access a common variable and at least one of them is a write.

The following subsections describe several concurrency techniques such as two-
phase locking and transactional memory. These techniques prevent conflicting transactions from performing concurrent updates to common variables and corrupting data. Two-phase locking takes and releases locks on variables at various granularities to prevent conflicting transactions. Transactional memory approaches can track read and write sets of individual transactions and abort those that conflict. When coupled with Persistent Memory, isolation techniques on concurrent transactions can move a persistent data structure from one valid consistent state to another.

<table>
<thead>
<tr>
<th>Listing 2.4: Coarse Lock</th>
<th>Listing 2.5: Durable</th>
<th>Listing 2.6: Fine-Grained</th>
</tr>
</thead>
<tbody>
<tr>
<td>( B() ) { \text{Lock;} \text{\hspace{.2cm}} \text{w = w+1;} \text{\hspace{.2cm}} \text{y = w;} \text{\hspace{.2cm}} \text{Unlock;} } )</td>
<td>( B() ) { \text{Lock;} \text{\hspace{.2cm}} \text{Begin Durable;} \text{\hspace{.2cm}} \text{w = w+1;} \text{\hspace{.2cm}} \text{y = w;} \text{\hspace{.2cm}} \text{End Durable;} \text{\hspace{.2cm}} \text{Unlock;} } )</td>
<td>( B() ) { \text{Lock w;} \text{\hspace{.2cm}} \text{Lock y;} \text{\hspace{.2cm}} \text{w = w+1;} \text{\hspace{.2cm}} \text{y = w;} \text{\hspace{.2cm}} \text{Unlock y;} \text{\hspace{.2cm}} \text{Unlock w;} } )</td>
</tr>
</tbody>
</table>

Table 2.3: Concurrency techniques for parallel transactions using locking, both coarse and fine-grained

2.4.1 Two-Phase Locking

Locking can be applied to any atomic technique for persistent memory as locks are taken at the application level and can postpone execution of competing threads. Locks will neither interfere with atomicity logging or durability with flushing and synchronous fences. Locking provides the isolation mechanisms that can be applied to atomic and durable transactions to create ACID transactions to Persistent Memory.

Figure 2.3 shows an example of two-phase locking, where locks are taken before a critical section and released afterwards. Two-Phase Locking, or 2PL, is sometimes called dual phase locking. In coarse-grained locking, programmers can take
a global lock, which is easy to code, but makes all transactions serialize execution. Fine-grained locking allows much more parallelism as transactions that do not share variables can execute in parallel. However, as shown in Listing 2.6, it is much more difficult to program. Locks must be taken in a global, pre-defined order to prevent deadlock. Otherwise, if locking order were reversed by another thread, then one thread could hold lock y and be waiting for lock w, while another thread was holding lock w waiting for lock y.

In a much more complicated data structure such as a B-Tree, ordering locks for fine-grained locking is extremely difficult. New nodes might be created by other threads concurrently and nodes that need to be locked might not be visible at another transaction start. A programmer might also not know which nodes a transaction might visit during its execution, which precludes ordering locks before start. Ideally, a programmer would like the ease of programming with a simple or coarse-grained locking manner, but have the high-performance of fine-grained locking.

2.4.2 Software Transactional Memory

Software Transactional Memory, or STM, implementations alleviate the complexities of ordering locks for fine-grained locking while providing higher performance than coarse-grained locking. There are a plethora of STM systems that can be found in the literature. STM techniques use a variety of logging and shadow copying mechanisms to enforce atomicity requirements. As noted in [74], these techniques have been improved over the past several decades, but STM slows critical sections by a factor of 3 to 5 and TM has had slow progress due by few benchmarks. Many STM implementations are not compiler integrated and therefore do not support an atomic keyword or section. They therefore require the use of library calls into an API similar to that in Listing 2.7 in Figure 2.4. Each read and write is accessed via the API along with methods to manage the overall transaction.

The most comprehensive system is called RSTM, for Rochester Transactional
Memory, and has been extended to support 13 different STM implementation techniques. DSTM introduced dynamically sized data structure support in an obstruction-free technique with a doubly indirect pointer locator for data accesses to an object in a transaction [75]. When an object is opened, the read set must be validated to make sure that other objects opened in the transaction have not changed values. Commits can be performed atomically after validating the read set by using a compare-and-swap instruction to change the status word of the open transaction.

Implementation of obstruction-freedom for STM was shown to be not as fast as locking techniques [76]. Ennal’s method [76] uses a two-phase revocable locking for writes, by acquiring locks on commit and aborting if it cannot acquire a lock, and optimistic concurrency control for reads, by validating that the read version of an object has not changed during a transaction. Transactional Locking, TL, [77] and TL2 [70] are similarly based on locking techniques. TL has two different modes, a commit mode and an encounter mode. In a commit or lazy mode, locks are taken near the transaction end, and variables utilized in the transaction are compared to current values. In encounter or eager mode, locks are taken as variables are accessed, initially preventing transactions from conflicting and having to re-try or rollback.

TL2 improves the implementation of TL by using a global version-clock that is incremented by one for each transaction as it writes to memory. The version-clock is read by all transactions to verify the read set, and is similar to global time stamped transactions where a global transaction manager assigns unique timestamps transactions. This method avoids inconsistent memory states which reduces aborts and is shown to enhance performance.

Since all reads are writes are moderated through the STM to detect and resolve conflicts, logging for persistent memory durability can be performed within the STM itself. Mnemosyne [20] provides a full STM for durable transactions to persistent memory using logs with a torn-bit technique and Write-Ahead Logging with a full virtual memory system. DudeTM [27] uses TinySTM [78] with undo logging to
decouple the concurrency phase from the persistence phase, using a shadow area similar to \cite{35,32}. For a comparison to other methods, we utilize the TL2 source \cite{79} and modify it to support durable transactions to Persistent Memory. We do so by utilizing the encounter, eager undo sets of the transactions and persist them to a log before writing new values.

<table>
<thead>
<tr>
<th>Listing 2.7: Using STM</th>
</tr>
</thead>
<tbody>
<tr>
<td>B()</td>
</tr>
<tr>
<td>{</td>
</tr>
<tr>
<td>STMBegin();</td>
</tr>
<tr>
<td>t=STMRead(&amp;w);</td>
</tr>
<tr>
<td>t = t+1;</td>
</tr>
<tr>
<td>STMWrite(&amp;w, t );</td>
</tr>
<tr>
<td>STMWrite(&amp;y, t );</td>
</tr>
<tr>
<td>STMEnd();</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Listing 2.8: Using HTM</th>
</tr>
</thead>
<tbody>
<tr>
<td>B()</td>
</tr>
<tr>
<td>{</td>
</tr>
<tr>
<td>HTMBegin();</td>
</tr>
<tr>
<td>w += 1;</td>
</tr>
<tr>
<td>y = w;</td>
</tr>
<tr>
<td>HTMEnd();</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>

Table 2.4: Software and Hardware Transactional Memory for Concurrency

2.4.3 Hardware Transactional Memory

Hardware Transactional Memory approaches for concurrency attempt to combine the ease of programming coarse-grained locks with the high performance of fine-grained locking. Hardware Transactional Memory, or HTM, was introduced in \cite{2} as a new, easy to use method for lock-free synchronization supported for hardware. The initial instructions for HTM included load and store transactional instructions in addition to transactional management instructions. The idea is based on extending an underlying cache-coherency protocol to handle detection of transactional conflicts during program execution.

Further implementations by Intel and IBM have removed the additional load and store instructions and, in some implementations, require only begin and end notations
for the transactions. The hardware system performs a speculative execution on a region of code defined by begin and end notations. The basic usage for HTM is shown in Listing 2.8 which is similar to an atomic section.

Intel uses a technique based on Speculative Lock Elision [3] called Intel Transactional Synchronization Extensions in two forms [38]. The first, an older Hardware Lock Elision, or HLE is compatible with existing cores by appending high order bits to lock management instructions. If the hardware doesn’t support the transaction, then a fallback lock is taken. This older architecture requires an explicit abort handler and is more cumbersome to program than simple begin and end instructions. A newer Restricted Transactional Memory, or RTM, is available on some Haswell architectures with micro-code updates and newer chips such as the 6700 series 6th generation processors based on the Skylake architecture. This newer HTM implementation is much easier to utilize, as it can be programmed similar to Listing 2.8, but a fallback lock (discussed in the next chapter) is required. According to [80], the Intel cores use the L1 for cache conflict detection and cache buffering, allowing for only 22KB of writes for transactional error detection.

HTM mechanisms are excellent for both ease of programming and execution speed, with conflict detection performed by the hardware. However, HTM, since it is based on cache-coherency, aborts transactions that perform writes through the cache [38]. This is problematic for applications wishing to perform durability, in that a log cannot be made durable to persistent memory during a transaction execution phase. Without the ability to log within a transaction, transactions durability is complex. External logging can be delayed by other threads or the processor, and subsequent, dependent transactions might have completed logging, making recovery difficult. Several approaches attempt to modify the well-known cache-coherency mechanisms to support HTM [25, 26, 27, 28, 81]; however, as reasonable as this may sound, it can be disrupting to processor and hardware manufactures as mentioned previously. Additional details on this problem and our solution are discussed in Chapter 3.
2.4.4 Checkpoints

(a) Checkpoint of multiple concurrent transactions

(b) Non-quiescent checkpoint of multiple concurrent transactions

Figure 2.6: Checkpointing Methods of Concurrent Transactions

A checkpoint can be used with multiple threads operating concurrently to create a version of the system, application, or data structure state across several threads. Figure 2.6a shows multiple transactions operating concurrently. As shown in a), if a checkpoint is initiated after transaction P completes, then all future transactions are delayed and after current open transactions B and Q are complete, the first checkpoint is saved and processing can continue. Failures that happen after this time can roll back to a completion point that contains transaction A, B, P, and Q. However, this delays transaction processing.

Non-quiescent checkpointing allows for concurrent transactions to proceed and make updates to a new checkpoint log, but must consult the current checkpoint that is being processed, which can complicate implementations and provide unnecessary overhead. In Figure 2.6b, the checkpoint also starts after transaction P completes, but only transactions A and P have completed and are included in the checkpoint, while in-work transactions B and Q will be included in the next checkpoint. How-
ever, transaction \( R \) doesn’t have to wait until the checkpoint is finished and can start with updates applied to the next checkpoint. Dalí \cite{69} presents a method for checkpointing a key-value store using versions to periodically persist the data structure. ThyNVM \cite{82} proposes a hardware mechanism to save previous checkpoint values in the background which collecting current transaction writes.

### 2.4.5 Summary

Table 2.5: Comparison of concurrency methods used for durable transactions

<table>
<thead>
<tr>
<th>Method</th>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coarse Locking</td>
<td>Easy to program</td>
<td>Poor parallelism and performance</td>
</tr>
<tr>
<td>Fine-Grained Locking</td>
<td>Excellent parallelism and performance</td>
<td>Difficult to Program</td>
</tr>
<tr>
<td>Software Transactional Memory</td>
<td>Good parallelism and performance</td>
<td>All reads and writes must be arbitrated by software</td>
</tr>
<tr>
<td>Hardware Transactional Memory</td>
<td>Easy to program; Excellent parallelism and performance</td>
<td>Cannot make durable logs within a transaction</td>
</tr>
<tr>
<td>Checkpoints</td>
<td>Moderate programming and performance</td>
<td>Loss of transactions on rollback to prior checkpoint version on recovery</td>
</tr>
</tbody>
</table>

Table 2.5 summaries the advantages and disadvantages of several concurrency techniques used for isolation. Ideally, to isolate **durable, atomic** transactions for Persistent Memory, we would like high performance and parallelism with ease of implementation programming. Hardware Transactional Memory is a great choice to reach the goal of programming ease with high performance. However, without the ability to perform durable logging within a transaction, logging is forced outside of the transaction. Logging post transaction completion can be delayed due to processor
delays, while other dependent transactions have completed logging. In Chapter 3, our solution is presented which describes how to create durable, atomic transactions for persistent memory with HTM as an isolation mechanism.

2.5 Related Persistent Memory Transaction Work

This section discusses related work in relation to the atomicity methods (Undo and Redo Logging, Copy-on-Write, and Versioning) and concurrency isolation methods (Locking, Software and Hardware Transactional Memories, and Checkpointing) described above. Some related work solutions propose obtrusive changes to processor level and forward cache level hardware, while others propose back-end memory controller solutions. This section also discusses several PM supported file systems, software solutions, and specialized data structures for PM.

To enforce atomicity in PM, software approaches in the literature rely on simple hardware support: Atomic 8-byte writes, memory fences, and instructions with persistent memory fencing semantics. As discussed in Section 2.2, recent Intel® instructions, such as CLWB and CLFLUSHOPT, have been introduced to aid in efficient persistent memory operations [51, 52]. The Intel® instruction PCOMMIT, which ensured all visible stores in buffers were durable in persistent memory, was recently deprecated in favor of adding persistent memory durability guarantees to the store fence instruction SFENCE [54]. Our approaches similarly rely on these hardware capabilities.

2.5.1 Hardware Supported Transaction Work

Many of the approaches for atomicity and durability of Persistent Memory transactions rely on the logging and versioning techniques previously described. This section divides the proposed hardware-supported PM transaction space into two broad categories, those that require processor or cache-level hardware changes and those proposals that can rely on back-end hardware changes.
Processor Level Hardware Changes:

Approaches for hardware changes to the front-end architecture with additional cache hardware and policies have been proposed \cite{BPFS, NV-heaps, Consistent and Durable Data Structures, NVMalloc}. These approaches can have significant implications to processor design by advocating changes to the front-end cache for ordering cache evictions \cite{BPFS, NV-heaps, Consistent and Durable Data Structures, NVMalloc}. For instance, BPFS \cite{BPFS} and NV-heaps \cite{NV-heaps} require changes to the system architecture to support the atomicity and consistency of data. The summary of each contribution and hardware change is as follows:

- **BPFS** \cite{BPFS} proposed *epoch barriers* as an addition to the cache hierarchy to control eviction order. This order is important for the proposed file system to avoid corruption of data structures. It utilizes a Copy-on-Write approach by copying changing portions of the file structure. When combined with hardware-supported atomic writes to 8-byte words, this provides an effective mechanism for atomic updates of an important, albeit restricted class of block-based tree-structured data structures, by using pointer flipping.

- **NV-heaps** \cite{NV-heaps} creates a persistent object system with garbage collection for fast NVMs that allows for safe and familiar pointer access by programmers. It relies on epoch barriers for consistency \cite{BPFS}, or combinations of persistent flush operations, and utilizes Write-Ahead Logging for atomicity and durability.

- **Consistent and Durable Data Structures** \cite{Consistent and Durable Data Structures}, or CDDS, uses timestamp-based multi-versioning. It requires a *flush* primitive to assert control of update order. CDDS \cite{Consistent and Durable Data Structures} provides a versioning method that copies data and uses sequences of fences and flushes to provide transaction support.

- **NVMalloc** \cite{NVMalloc} is also based on a timestamp-based multi-versioning technique. It proposes a modification to the cache for cache line counters, a set of counters in the CPU for id association, and a new instruction. A cache line contains a
tag to an identifier, and when the line is flushed, the identifier is referenced and decremented. Using a proposed new instruction, an application may create an association group, associating the identifier to a group of cache-lines. Groups of writes that need to be atomic are then in cache and only durable when the identifier, which associates the cache-lines, becomes zero.

- Kiln \[83\] is a proposed non-volatile victim cache to provide transactional buffering. It has the added property of not requiring logging, but requires disruptive changes to the front-end cache controller. Kiln \[83\] tracks pre- and post- transactional states for cache-lines in both volatile and persistent caches. It atomically moves cache-lines to durable state on transaction commits.

- Whole-system persistence \[13\], or WSP, is designed for in-memory databases. It requires hardware support to snapshot the entire micro-architectural state at the point of a failure. WSP relies on batteries to power non-persistent memories on system failure.

- ATOM \[64\] proposes a hardware log manager for managing logs off of the critical application execution path. To enforce ordering, ATOM is implemented across first level caches and memory controllers. It utilizes Undo Logging to allow for in-place updates and access.

- Hands-Off Persistence System \[84\], or HOPS, utilizes both front-end and back-end persist write buffers. It requires new ordering and durability fences for the buffers, similar to those introduced in efficient persist barriers \[87\]. Programmers use these fences when creating transactions and can implement an Undo Log.

- Strand Based Persistent Memory System \[85\], or SPMS, implements a relaxed persistent memory model based on strand persistency \[60\]. It modifies the front-end cache to track dependencies and introduces a new instruction. The
proposal also requires an ultra-capacitor to push write buffers to PM. SPMS was inspired by WrAP [33], however SPMS was implemented in the front cache unlike WrAP, which is implemented on the back-end near the memory controller.

- **Transaction Cache** [86], or TC, introduces a persistent memory accelerator into the processor. The persistent memory accelerator is a side data path for PM operations, the main component being a persistent Transaction Cache (TC) directly off of the CPU. The persistent TC caches data and sends it to the memory controller transactionally to avoid corrupting data. TC requires modifications to the cache hierarchy to contain a persistence flag and last level cache misses to consult the TC for persistent memory locations.

- **Hardware-Accelerated Persistent Transactions** [68], or HAPT, was inspired by SoftWrAP [63]. The solution implements a shadow DRAM area for Persistent Memory transactions, similar to the software alias table in SoftWrAP. However, HAPT uses a proposed hardware mechanism to update the TLB with aliased PM addresses to an area in DRAM. HAPT utilizes both a Software Alias Table and Hardware Alias Table, which is managed by an Alias Management Unit (AMU) between the processor and TLB.

The techniques used in this thesis for atomicity achieves efficiency through non-temporal write-combining streaming of log records. Our approach neither requires changes at the processor level nor to the front-end cache controller.

**Memory Controller Support:**

Memory controller support for transaction atomicity for Persistent Memory have been proposed [21, 88, 89, 36, 37, 33, 90]. These methods may be preferred by hardware implementors as they do not interfere with the well-known cache interface and coherency mechanisms.
• Adding a small DRAM buffer in front of Persistent Memory to improve latency and to coalesce writes was proposed \cite{21}. This approach greatly reduces write latency of Phase Change Memory to the speed of DRAM and provides enhanced write endurance. However, it does not address or support atomicity or durability of write sequences.

• Low-level memory scheduling to improve efficiency of persistent memory access was studied in \cite{88}. The presented modifications improve Phase Change Memory Quality-of-Service, but also does not address atomicity of groups of writes.

• FIRM \cite{89} describes techniques to differentiate persistent and non-persistent memory traffic, and presents scheduling algorithms to maximize system throughput and fairness. It places memory requests into different categories based on uses and schedules the accesses across multiple buffers to improve throughput.

• ThyNVM \cite{82} is a hardware-assisted mechanism that proposes hardware changes to allow dual-scheme checkpointing. This method writes previously checkpointed values in the background while collecting current transaction writes. It integrates tightly with the memory controller, incorporating address translation tables for both cache-line blocks and pages. ThyNVM requires integrating an additional checkpointing controller into hardware.

• WrAP \cite{36, 37, 33} advocates use of a volatile victim cache to prevent uncontrolled cache evictions from reaching persistent memory during an atomic group of stores to PM. WrAP utilizes an efficient Redo Logging technique that write- Combines log entries to reduce memory bandwidth. The controller supports concurrent transaction execution. It further handles log pruning and victim cache space management in an integrated manner. The hardware-based WrAP architecture allows an application to handle its own concurrency control, like software locking, such as in RVM \cite{91}.
Our approach effectively uses HTM for concurrency control and does not require changes to the font-end cache controller or use logs for replaying transactions to Persistent Memory.

2.5.2 File System Work

Research in file systems built on Persistent Memory is also a promising area that might quickly enable software applications to take advantage of PM. This will be advantageous for legacy file-based applications, as these applications can then easily take advantage of byte-addressable non-volatile memory. If a system failure were to occur, applications using file-based persistence are subject to file system guarantees for recovery, which can lose partial updates to data. Therefore, applications must build additional protections, like database log files, on top of the file system layer. These additional protections can result in multiple layers of logging for persistence recoverability.

Some Persistent Memory file systems include:

- The Persistent Memory File System [42], or PMFS, is a complete file-system implementation developed by Intel® for Persistent Memory. PMFS provides a mechanism to easily manage and access PM located on processor attached memory slots. It doesn’t address virtualization, either full or container-based. PMFS also includes methods for transactional access to PM data. PMFS has been replaced by PMDK since support for direct access to PM has been included in Linux-based filesystems.

- BPFS, or Block-Persistent File System, [18] uses Copy-on-Write techniques for ordering of cache evictions but requires changes to the hardware. However, as described in Section 2.5, BPFS requires hardware support added to the processor cache hierarchy.

- DAX support for file systems, including Ext2 and Ext4, provide direct, un-
buffered access to files. This support does not scale when used with container-based virtualization. Transaction mechanisms must be built on top of DAX file systems to provide guarantees for groups of PM updates.

- SCMFS uses sequences of \texttt{mfence} and \texttt{clflush} operations to perform ordering and flushing of load and store instructions and requires garbage collection [47].

- Aerie [48] exposes a Persistent Memory file system without kernel interaction in user mode.

- NOVA [49] is a hybrid file system for Persistent Memory and DRAM that provides consistency guarantees.

- Failure-Atomic Msync [92], is a technique for ensuring atomicity of \texttt{msync}. Though not developed for PM, this technique ensures that data written into the file system page caches, or buffers, is atomic, all or none, with respect to programmer calls to \texttt{msync}. This technique is developed for the Linux ext4 file system to enable support for atomic updates to memory mapped file data.

2.5.3 Software and Concurrent Transaction Work

To maintain the consistency of Persistent Memory, existing non-HTM solutions [20, 65, 67] tightly couple concurrency control with durable, atomic writes. Software solutions that employ these approaches generally extend the duration of critical sections for durability operations. Extension of critical sections leads to longer times to hold locks, consequently reducing concurrency, expanding transactional duration. Specialized data structures [66, 69, 93] can be customized for PM durability. These data structures are specific implementations and not applicable as a general purpose or integrated framework for PM. Other work [32, 27] decouples concurrency control so that post transactional values may flow through cache hierarchy and reach PM asynchronously. However, in these solutions, the transaction log must be persisted synchronously to PM before transaction close.
• PMDK [43], is the Persistent Memory Development Kit led by by Intel®. PMDK provides several software libraries that include support for Persistent Memory management and transactional access to PM. Transactional access utilizes Undo Logging. Additional libraries are provided that allow for overloaded dynamic memory allocation in legacy applications with PM placement. Other libraries support atomic PM blocks of data and transactional object stores for data.

• Mnemosyne [20] is a full Software Transactional Memory system that supports Persistent Memory with keywords and primitives for persistent variable support. To achieve both concurrency control and atomicity, Mnemosyne uses internal copying and logging through sequences of write through stores, fences, and flushes. The implementation provides a compiler with an atomic keyword for marking sections atomic, intercepting all writes and reads within a transaction. The atomic section allows for both concurrent accesses and persistence. Mnemosyne can use different modes for consistency updates, such as shadow or in-place updates, depending on the underlying data structure. However, it uses a full virtual PM sub-system, protecting PM locations from other processes and swapping them to files. While this can protect data areas, it also reduces performance.

• ATLAS [65] is a tool which uses a compiler pass and a Copy-on-Write based method for atomic durability. In its approach, an area that is locked is identified as a Failure-Atomic Section or FASE. A durability ordering on every FASE is induced by happens-before relationships. This method provides a good approach for existing systems which utilize locks for concurrency control, but can be limited in performance due to the synchronous copying of data and atomic pruning of the log.

• NVthreads [94] is a library that is a drop-in replacement for the pthreads
Unix multi-threading library. Similar to ATLAS [65], NVthreads uses critical sections to determine atomic regions that need crash recovery. The library uses page level tracking and Redo Logs for recovery after a crash.

- REWIND [67] utilizes a custom atomic doubly linked list structure. This linked list structure is used to minimize the flushes to a write-ahead log for consistency. Locks are used in a fine-grained manner to serialize log access for thread-safe access during checkpointing. However, this method doesn’t benefit from write-combined logging and requires checkpointing and user locks. It is a library that also supports transactional writes to main memory similar to early an early presentation of Software-based Write-Aside Persistence [35].

- Techniques for efficient logging of NVM using cache-coherence protocols [95] modifies TinySTM [78] for persistent memory. To reduce the log traffic, the approach compresses log entries based on visible writes in the transaction write set before persisting the log.

- NV-heaps [66] supports ACID transactions for software applications through a full STM utilizing write-ahead logging. It requires the use of epoch barriers, a processor and cache addition to the hardware, as described previously.

- Dalí [69] presents a method for checkpointing a key-value store safely to NVM using versioning, periodically persisting the data structure. Versions of each record are kept in a bucket entry, and depending on the prior crashes of the system, a safe value is found for a key from prior versions. Dalí defines the periods of checkpoints as epochs, and uses the epoch counter to determine if a record version has been persisted on NVM, currently being persisted, or is not-yet persisted.

- Persistent Lock-Free Queue [93] presets several versions of a concurrent, lock-free queue that persists values to Non-Volatile Memory. It is based on pervious
versions of concurrent queues and uses a persistent flush to ensure atomicity of concurrent inserts and deletes.

- DudeTM [27] effectively decouples transactional persistence from transactional execution similar to SoftWrAP [32]. It also employs the use of TinySTM [78] for implementation and transaction isolation. More discussion on DudeTM is in Chapter 3.3.

- SoftWrAP [35, 32], or Software-based Write-Aside Persistence, utilizes an in-volatile memory area alias table for fast shadow variable accesses and Redo Logging. The approach lets reads proceed through DRAM while log processing can be performed from the DRAM based alias table. Consecutive log entries are streamed and write-combined into PM. Multiple writes to the same variable become one final merged update into its Persistent Memory home location at a later time. Concurrency control mechanisms, like two-phase locking, can be applied by the application developer.

- Dual Phase Locking can be combined with many of the hardware atomicity solutions like WrAP [63]. Finer locking granularity combined with atomicity control can result in higher parallelism and performance. However, fine-grained locking has the added cost of lock ordering and development.

Combining logging and locking techniques, other work creates a full in-memory database for accessing Persistent Memory. Persistent Memcached [96] creates a full in-persistent memory version of the popular key-value store Memcached [97]. The authors discuss tradeoffs and challenges in identifying and implementing atomic sections in the full application. A Persistent Memory data-structure store [73] based on Redis [4] was created using techniques from SoftWrAP. It utilizes local alias tables and shows a significant improvement over file-based persistence.

New HTM mechanisms for concurrency control have recently been proposed [25, 26, 27, 28, 81]. However, all of these solutions require making significant changes to the
existing HTM semantics and implementations. Large changes to well-known semantics can be problematic for implementation on modern processors. These approaches are discussed in more detail in Chapter 3. Our solution to utilizing HTM for PM requires no changes to processor cache coherence protocols and can be run on existing Intel® systems with a novel software solutions.

2.6 Summary

This chapter discussed the challenges in utilizing Persistent Memory for durable data structures that will reside in memory instead of on block-based media like hard disk and solid state drives. Atomic based persistence of groups of stores to PM is an exciting area of research in both computer architecture circles and software groups. Existing logging and versioning mechanisms like Undo and Redo logging, Copy-on-Write, and Checkpoints were described. Research into proposed hardware changes, both on the front-end and back-end, were presented.

Adding concurrency controls for isolation to existing atomicity methods for durable PM transactions is not straightforward. Existing concurrency control mechanisms from two-phase locking and checkpoints to Software and Hardware Transactional Memory systems were described in relation to PM along with current research. The following chapter successfully adds HTM as an isolation mechanism for durable PM transactions without any of the complex hardware modifications described. Our solution safely uses HTM in the foreground and can persist transactions in the background off of the critical execution path.
Chapter 3

Hardware Transactional Memory + Persistent Memory

This chapter describes the problem of persisting HTM transactions to Persistent Memory and presents a solution model. HTM was designed for volatile memory transactions, where all values are instantly visible in the cache hierarchy after transaction completion. Coupling HTM with PM is challenging, as PM might see transaction stores, which were instantly visible in the cache, evicted in no particular order. These unordered stores makes PM consistency vulnerable to corruption on a system crash.

A generic solution model for creating durable HTM transactions to PM is presented. The solution allows for hardware to manage concurrency control while preserving durability to PM. Furthermore, the solution introduces relaxed durability, whereby transactions can perform other work and do not have to wait for other transactions to complete. The solution requires no changes to existing processor hardware or protocols and executes on existing commodity processors. A detailed example is presented along with related work. Chapters 4 and 5 present novel software and memory control based solutions respectively using the model discussed in this chapter.

3.1 HTM + PM Basics

Hardware Transactional Memory, or HTM [2], first discussed in Chapter 2.4.3, is a hardware provided concurrency control mechanism, by which applications can use ordinary load and store instructions, along with transactional management instructions, to gain efficient parallelism in an easy to use manner. Most HTM implementations extend an underlying cache-coherency protocol to handle detection of transactional
conflicts during program execution. The hardware system performs a speculative execution on a demarcated region of code similar to an atomic section. Independent transactions (those that do not write a shared variable) proceed unrestricted through their HTM sections. Transactions which access common variables concurrently in their HTM sections, with at least one transaction performing a write, are serialized by the HTM. That is, all but one of the transactions is aborted; an aborted transaction will restart its HTM code at the beginning.

Updates made within the HTM section are hidden from other transactions and are prevented from writing to memory until the transaction successfully completes the HTM section. This mechanism provides atomicity (all-or-nothing) semantics for individual transactions with respect to visibility by other threads, and serialization of conflicting, dependent transactions. However, HTM was originally designed for volatile memory systems (rather than supporting database style ACID transactions) and therefore any power failure leaves main memory in an unpredictable state relative to the actual values of the transaction variables.

Persistent Memory, or PM, introduces a new method of persistence to the processor. PM, in the form of persistent DIMM, resides on the main-memory bus alongside DRAM. Software can access persistent memory using the usual LOAD and STORE instructions used for DRAM. Like other memory variables, PM variables are subject to forced and involuntary cache-evictions and encounter other deferred memory operations performed by the processor.

For Intel® CPUs, CLWB and CLFLUSHOPT instructions provide the ability to flush modified data (at cacheline granularity) to be evicted from the processor cache hierarchy. These instructions, however, are weakly ordered with respect to other store operations in the instruction stream. Intel has extended the semantic for SFENCE to cover such flushed store operations so that software can issue SFENCE to prevent new stores from executing until previously flushed data has entered a power-safe domain; i.e., the data is guaranteed by hardware to reach its locations in the PM media. This
guarantee also applies to data that is written to PM with instructions that bypass the processor caches. While in an HTM transaction, however, a CPU cannot exercise CLWB, CLFLUSHOPT, non-cacheable stores, and SFENCE instructions since the stores by the CPU are considered speculative until the HTM transaction completes successfully.

Even though HTM guarantees that transactional values are only visible on transaction completion, hardware manufacturers cannot simply utilize a non-volatile processor cache hierarchy or battery backed flushing of the cache on failures to provide transactional atomicity. First, transactions that do not complete before a software or hardware restart produce partial and therefore inconsistent updates in non-volatile memory, as there is no guarantee when a machine halt will occur. The halt may happen during XEnd execution leaving only partial updates in cache or write buffers which can corrupt in-memory data structures. Additionally, any solution must preserve fallback paths for transactions executing outside HTM semantics. Transactions executing in a fallback path have cached values visible, so persisting cached values to PM on failure can result in partial updates of transactions and data corruption.

3.1.1 Fallback Paths

Transactional concurrency control mechanisms built using HTM do not only use hardware instructions, such as XBegin and XEnd, to protect critical sections for several reasons. First, the hardware can abort any transaction at any time. With transactions competing for processor time, one transaction may never get executed and starve. Additionally, a transaction can have a large write set that overflows transactional buffers aborting the transaction, and such a transaction only using HTM instructions could never succeed. Other situations such as transactions that are unusually time consuming or involve system calls or events may also be aborted by the processor.

To overcome these limitations, HTM based concurrency controls use mechanisms
to catch and re-try aborted transactions and allow for a fallback path for transactions that have been aborted a number of times greater than a configurable threshold. The fallback path for aborted transactions involves a software based locking mechanism that does not use HTM. Variable values inside the locked portion of code in a fallback path are thus visible to other threads and the cache and memory hierarchy. Since these values are visible to the memory hierarchy, they are subject to write-back and out-of-order cache evictions as mentioned previously. If a power failure were to occur during a software-based locked region of a fallback path for HTM, then values inside a transaction may only be partially complete, and any flushing of the cache to PM or values remaining in a persistent cache can corrupt home PM variable values.

3.2 Challenges Persisting HTM Transactions

Consider transactions A, B and C shown in Listings 3.1, 3.2 and 3.3. Assume that $w$, $x$, $y$, $z$ are persistent variables initialized to zero in their home locations in PM. The code section demarcated between the instructions XBegin and XEnd will be referred to as an HTM transaction or simply a transaction. The HTM mechanism ensures the atomicity of transaction execution. Within an HTM transaction, all updates are made to private locations in the cache, and the hardware guarantees that the updates are not allowed to propagate to their home locations in PM. After the XEnd instruction completes, all of the cache lines updated in the transaction become instantaneously visible in the cache hierarchy.

1. Atomic Persistence: The first challenge is to ensure that the transaction’s updates that were made atomically in the cache are also persisted atomically in PM. Following XEnd, the transaction variables are are once again subject to the normal cache operations like evictions and the use of cache write-back instructions. There are no guarantees regarding whether or when the transaction’s updates actually get written to PM from the cache. This can create a problem if the machine crashes before all these updates are written back to PM. On a reboot, the values of these
variables in PM will be inconsistent with the pre-crash transaction values. This leads to the first requirement:

- *Following crash recovery, ensure that all or none of the updates of an HTM transaction are stored in their PM home locations.*

A common solution is to log the transaction updates in a separate persistent storage area before allowing them to update their PM home locations. Should a crash interrupt the updating of the home locations, the saved log can be replayed. When transactions execute within an HTM there is a problem with this solution since the log cannot be written to PM within the transaction and can be done only after the `XEnd`. At that time the transaction updates are also made visible in the cache hierarchy and are susceptible to uncontrolled cache evictions into PM. Hence there is no guarantee that the log has been persisted before transaction updates have percolated into PM. We describe our solution in Section 3.4.

2. **Persistence Ordering:** The second problem deals with ensuring that the *execution order* of dependent HTM transactions is correctly reflected in PM following crash recovery. As an example, consider the dependent transactions $A, B, C$ in Listings 3.1, 3.2 and 3.3. The HTM will serialize their execution in some order: say $A$, $B$ and $C$. The values of the transaction variables following the execution of $A$ are given by the vector $V_1 = [w, x, y, z] = [1, 1, 0, 0]$; after the execution of $B$ the
vector becomes $V_2 = [2, 1, 2, 0]$ and finally following $C$ it is $V_3 = [3, 1, 2, 3]$. Under normal operation the write backs of variables to PM from different transactions could be arbitrarily interleaved. For instance suppose that $x$ is evicted immediately after $A$ completes, $w$ after $B$ completes, and $z$ after $C$ completes. The persistent memory state is then $[2, 1, 0, 3]$; should the machine crash, the PM will be contain this meaningless combination of values on reboot. A consistent state would be either the initial vector $[0, 0, 0, 0]$ or one of $V_1, V_2$ or $V_3$. This leads to the second requirement:

- **Following crash recovery, ensure that the persistent state of any sequence of dependent transactions is consistent with their execution order.**

If individual transactions satisfy atomic persistence, then it is sufficient to ensure that PM is updated in transaction execution order. With software concurrency control (using STM or two-phase transaction locking), it is straightforward to correctly order the updates simply by saving a transaction’s log _before_ it commits and releases its locks. In case of a crash, the saved logs are simply replayed in the order they were saved, thereby reconstructing persistent state to a correctly-ordered prefix of the executed transactions.

When HTM is used for concurrency control the logs can only be written to PM _after_ the transaction **XEnd**. At that time other dependent transactions can execute and race with the completed transaction, perhaps writing out their logs before the first. Solutions like using an atomic counter within transactions to order them correctly are not practical since the shared counter will result in HTM-induced aborts and serialization of all transactions. Some papers have advocated that processor manufacturers alter HTM semantics and implementation to allow selective writes to PM from within an HTM [25, 26, 27]. We describe our solution without the need for such intrusive processor changes in Section 3.4.

### 3. Strict and Relaxed Durability

In traditional ACID databases, a committed transaction is guaranteed to be durable since its log is made persistent before it commits. We refer to this property as **strict durability**. In HTM transactions the
log is written to PM after the **XEnd** instruction some time before the transaction commits. A natural question is to characterize the time it is safe for a transaction requiring strict durability to commit.

It is generally not safe to commit a transaction \( Y \) at the time it completes persisting its log for the same reason that it is difficult to ensure persistence ordering. Due to races in the code outside the HTM it is possible that an earlier transaction \( X \) (on which \( Y \) depends) to have completed but not yet persisted its log. When recovering from a crash that occurs at this time, the log of \( Y \) will not be replayed since earlier transaction \( X \) cannot be replayed. This leads to the third requirement:

- **Following crash recovery, strict durability requires that every committed transaction is persistent in PM.**

We define a new property known as **relaxed durability** that allows an individual transaction to opt for an early commit immediately after it persists its log. Requiring relaxed or strict durability is a local choice made individually by a transaction based on the application requirements. Transactions choosing relaxed durability face a window of vulnerability after they commit, during which a crash may result in their transaction updates not being reflected in PM after recovery. The gain is potentially reduced transaction latency. However, irrespective of the choice of the durability model by individual transactions, the underlying persistent memory will always recover to a consistent state, reflecting an ordered atomic prefix of every sequence of dependent transactions.

### 3.2.1 Logging Challenges

Another challenge faced with HTM based transactions for PM involves logging transactional values. Within an HTM transaction, values that are forced outside of cached stores, such as with streaming or non-temporal stores or instructions that flush cache lines, such as CLFLUSH or CLWB, commonly used for persistent logging operations, will cause HTM based transactions to immediately abort. For these reasons, to make
HTM based transactions durable on PM, transactions must be separated into a concurrency phase for variable updates, with normal load and store operations, and a durability phase to ensure transactional durability and data consistency.

Solutions that attempt to couple these two phases, require changes to the well-known HTM mechanisms to allow stores to flow through the cache and not abort transactions, or involve software-based STM. Software Transactional Memory based solutions require instrumenting every load and store operation while managing locking and conflict resolution all through slower software mechanisms. These approaches use software arbitration and locking that increase the amount of time spent in the critical section by introducing more work which in turn increases the probability of transactional conflicts and aborts.

In this section we visit the challenges faced by both Undo and Redo Logging outside HTM sections. We return to our example, when transactions A, B, C are executed by HTM in that order and assume PM atomicity. If the system crashes after C finishes executing, the values in PM can only be one of the states \( S_0, S_1, S_2, S_3 \). Similarly, a crash after B executes could result in any of the PM states \( S_0, S_1, S_2 \), but not a state like [1, 0, 2, 0] that has some updates from A and some from B.

Consistency of PM updates forces an order in which transaction updates become durable in PM. Specifically, when any transaction \( T \) becomes durable, all earlier transactions that conflict with \( T \) must be durable; otherwise the durable state of the system can be inconsistent with the execution order in volatile memory.

**Undo Logging:** Traditionally, Undo Logging is performed as encounter mode logging; variables old values are logged as they are encountered just before the new values are written. Thus, a programmer need not know all the variables that will be update before a transaction begins. However, since with HTM a programmer must create a persisted log outside of the critical section, creating an Undo Log of all variables that will or may be updated within a transaction must be performed before the HTM atomic section is entered. A programmer must know all variables a priori that will
Undo Logging. Undoing HTM transactions in PM after a failure can cause partial undoing of completed transactions causing an inconsistency.

Redo Logging. For recovery, logs must be replayed in HTM completion order, but logging operations might be delayed past other transactions.

Figure 3.1: Problems with logging outside HTM + PM transactions.

or may possibly be updated in a transaction before a critical HTM section begins. Therefore, additional programming logic must be implemented to log all variables that will or could be updated before a transaction HTM section begins, which can drastically affect the performance of programs and legacy code. This can greatly complicate programming as many transactions may be nested or rely on a plethora of helper functions and branches, with variables and pointers not known until run-time.

Additionally, if an Undo Log is created before the HTM section begins, another transaction may start and completely finish, making the persistent undo state invalid. See Figure 3.1a which shows Undo Logging. Returning to our example of transactions A, B, and C, if they use Undo Logging coupled with HTM an additional problem can occur. Consider transaction A creates an Undo Log of \([w = 0, x = 0]\), B of \([w = 0, y = 0]\), and C of \([w = 0, z = 0]\). After A completes, the PM state is \([1, 1, 0, 0]\). If a crash occurs at time \(t_1\), when transaction B is writing new values of \(w\) and \(y\), then the system might play Undo Logs for transaction B and C and not transaction A, leaving the system in an in-consistent state \([0, 1, 0, 0]\).

Redo Logging: In the example, when B becomes durable (i.e., the PM values are
$w = 2$ and $y = 2$), we require that $A$ must also be durable (i.e. $x = 1$ in PM). If not, a crash at this time will lead to the inconsistent global state $[2, 0, 2, 0]$. For Redo Logging, if the state is restored from saved logs during crash recovery, then the logs must be replayed in the order of memory execution and there should be no gaps in the sequence. A transaction must save enough log metadata to assure this information is available to the recovery routine. A recovery routine must replay the logs in timestamp order to maintain consistency.

Cache evictions immediately after HTM end must also be caught when Redo Logging. Consider Figure 3.1b at time $t1$ if an uncontrolled cache eviction evicted $y = 2$ and an immediate failure occurred, then we might have the state of $A$ and part of $B$, e.g. $[1, 1, 2, 0]$, an invalid state.

However, there is an additional subtlety that still needs to be addressed. Specifically, one needs to ensure that all conflicting transaction logs that precede the log being replayed have also been replayed. Since transactions execute independently and freely once they are outside the HTM, the order in which they actually complete their log writes in PM is not related to the order of transaction completions in memory. In the example shown in Figure 3.1b after $A$, $B$ and $C$ have finished their HTM section, it is possible that $A$ and $C$ have finalized their logs in PM, but the logging of $B$ is delayed. If there is a failure at time $t1$, a recovery routine might replay the logs of $A$ and $C$ in that order (based on their timestamps and existence of the log) yielding the inconsistent state $[3, 1, 0, 3]$. A solution must therefore detect that $B$ executed before $C$ and, since $B$ does not have its logs saved, restore only $A$, so that the restored state is $S_1$ (a consistent state). Note this means that $B$ and $C$ are not durable but the system is restored to a consistent state where only $A$ executed.

It is perfectly acceptable for a thread completing execution of transaction $C$ in Figure 3.1b to complete and continue execution on other tasks. If a system failure were to occur, the state of PM would reflect the state of transaction $A$ only. However, if the thread required that transaction $C$ be durable (i.e. recoverable on failure
after completing), then the thread must delay until the dependent transaction \( B \) has completed writing its logs at time \( t2 \). At that point, the thread may continue execution on tasks, ensured that it’s transactional writes were durable and recoverable on failure.

Additionally, even at time \( t2 \) when \( B \) has completed, it must need to know which values of \( w \) and \( y \) to correctly persist to PM. Transaction \( A \) cannot simply flush its own values of \( w = 1 \) and \( x = \) to PM during its logging and durability phase since \( w \) is different, having been updated by \( B \) during its HTM section. This can be true for any transaction, in that an in-flight transaction that has completed concurrency assignment but not persisted to PM can overwrite later concurrent assignments that persisted earlier.

We describe a solution that avoids the need for locking and serializing the replaying of logs to PM in Section 3.4. Our solution also allows for both relaxed and strict durability; that is a transaction may proceed after completing and writing its logs or wait until other dependent transactions that started prior have also completed.

### 3.2.2 Ordering Challenges

Ordering HTM transactions can be challenging, for instance sharing a common updated sequence variable will abort concurrent HTMs for conflicting write sets. Additionally, reading a system time can also be costly and abort some transactions that trap into the operating system call. Adding logging outside the HTM section complicates the situation even further. Consider the challenges of using Redo Logging presented in the previous section.

Catching cache evictions permits Redo Logging, discussed later, but a consistent ordering on completed transactions must be determined. The question is to determine a correct, consistent point in transactions that can provide a correct overall ordering of all potential parallel transactions. A transaction must preserve serializability in that all the transactions can be represented in some serial order in time that preserves
(a) Ordering with times taken before or after the critical section can be inconsistent with actual ordering.

(b) Ordering with times taken inside the HTM concurrency section.

Figure 3.2: Challenges with Transaction Ordering

the correct persistence.

Consider the two transactions $A$ and $B$ shown in Figure 3.2a. If we choose a time before the critical sections, such as times $t_{1\text{before}}$ and $t_{2\text{before}}$, then, in the situation shown in the figure, these times are actually in reverse order of the actual assignment of variables in the critical sections in $A$ and $B$. Persisting values in this order can lead to an inconsistency. Choosing a point right after the critical section with times $t_{1\text{after}}$ and $t_{2\text{after}}$ has a similar problem. The critical section of $B$ happens before $A$, but due to system delays, the recorded times are in reverse order. Using these times for a serial ordering of the transactions can put the system in an inconsistent state. Therefore, we need to choose a time within the critical section.

However, an arbitrary time within a transaction cannot be used, even though the HTM will abort with conflicting read and write sets. An arbitrary time would work with non-conflicting variable sets, since replaying transactions of non-conflicting variables may be performed in any order, as they won’t conflict. However, due to intra-transaction delays and write ordering, we cannot assume that an arbitrary order will allow for correct ordering of transactions that do have overlapping variables even if the HTM will abort. For example, refer to Figure 3.2b. Let $t_{1\text{start}}$ and $t_{2\text{start}}$ denote the immediate starting time of the transactions as soon as the transaction has
started. If these times are used to order the transactions A and B, they can also create an out of order situation. Suppose the writes to w and y are delayed within the transaction B to time \( t_{\text{middle}} \). If logs are then replayed in the order B, A, then this also leaves the system in an inconsistent state.

Using the transaction end times within the HTM section before the XEnd for ordering can fix the aforementioned scenarios. Consider times \( t_{1\text{end}} \) and \( t_{2\text{end}} \) that are read just before an HTM section is completed as shown in Figure 3.2b. At the end of the transaction, the full and completed read and write sets are dependencies to other transactions. Therefore, if there is any conflict in these sets then the transaction will abort the HTM section. Using the end times lets us know that the transaction has completed its full read and write sets. If a delay is experienced by a transaction after reading the time and before the XEnd command and the transaction still completes, then we know that no other pending transaction completed that had overlapping read write sets during the period before the \( t_{\text{end}} \), and the transaction can be safely retired by the ending time.

### 3.3 Related Work

Recent work [25, 26, 27, 28] aims to exploit processor-supported HTM mechanisms for concurrency control instead of traditional locking or STM-based approaches. However, all of these solutions require making significant changes to the existing HTM semantics and implementations. One difficulty with proposals for changing hardware is the uncertainty around whether and when such proposals are embraced by processor manufacturers. These proposals were the only ones to our knowledge that attempt to utilize HTM with PM in an ACID framework.

- PHTM [25] and PHyTM [26], propose a new instruction called **Transparent-Flush** which can be used to flush a cache line from within a transaction to persistent memory without causing any transaction to abort and without invalidating the cache line. They also propose a change to the xend instruction that
ends an atomic HTM region, so that it atomically updates a bit in persistent memory as part of its execution, allowing for a log indication record. A global log map is used to lock out current transactions so that only one transaction is updating an uncommitted value at a single time. This has the disadvantage of holding locks longer than an HTM section and performing persistence within an HTM. PHTM and PHyTM require transactional stores to use an API to lock out other stores until transactions are persisted.

- DUDETM \[27\] also requires changes to the coherency mechanisms. Similar to SoftWrAP \[63\], it uses shadow DRAM for execution. However, DUDETM requires a shadow memory wish is a logical mirror of the PM space, which can be quite large. For DUDETM to use HTM, it requires that designated memory variables \textit{within a transaction} be allowed to be updated globally and concurrently without causing an abort. It also requires both Undo Logging in DRAM integrated with TinySTM \[78\] and Redo Logging when reproducing transactions to PM.

- Durable Hardware Transactional Memory or DHTM \[81\] requires significant changes to the cache hierarchy. DHTM proposes adding a log buffer to the level one cache. The log buffer is fully associative, keeping track of cache line addresses used in the transaction. At the end of a transaction, additional control logic ensures cache lines recorded in the log buffer are persisted to PM before the transaction is complete. The control logic added to the processor also track and record dependent transactions and record them in a sentinel log entry in transactions. This log entry is used in a recovery manager to manage conflicting updates and transaction order.

- PTM \[28\] extends HTM designs by adding several components to the processor cache. It adds a on-chip scoreboard to track dependencies among non-persisted transactions and a global transaction id register. As an item gets evicted out
(a) Solutions that mix concurrency with persistence hold locks for a longer period of time.

(b) Some solutions tightly couple persistence with concurrency which causes delay of future concurrency sections until persisted.

(c) Our solution loosely couples persistence to the transaction while still preserving tight persistence consistency, allowing for higher concurrency.

Figure 3.3: Concurrency with persistence

of the last level cache, PTM has to first scan the scoreboard to see if any transactions depend on the item being evicted, then examine the entire cache to flush any data to undo logs of prior dependent or current transactions before proceeding. It allows sparse transactions committal, as transactions need not be retired in order if they don’t depend on each other.

In addition to changing the cache coherency mechanisms, these solutions tightly couple concurrency with persistence to maintain transactional atomicity and the consistency of PM as shown in Figure 3.3. For instance, software based locks are held within the HTM concurrency section by PHTM [25] and PHyTM [26] in a global log
map. This tight coupling extends the transaction concurrency dependencies until a persistence point, which pushes dependent transactions further into the future and consequently lowers throughput, as shown in Figure 3.3a. In comparison to the volatile memory transactions which are often fine-tuned for cache-friendly execution, the persistent memory versions have to incur cache flushing synchronous writes to PM in order to commit either log records or data cache-line updates before releasing locks. Generally this means they must extend the duration for which they remain in critical sections (i.e., leading to raised lock hold times) which both reduces concurrency and expands transactional duration. These solutions too have the property, as depicted in Figure 3.3b that, the overall transaction cannot complete until the writes to PM have completed. Our solution on the other hand loosely couples the two sections while still preserving persistence consistency. As shown in Figure 3.3c, our method allows for lightweight concurrency which leads to increased throughput.

3.4 Hardware Transactional Persistent Memory Model

Our solution separates concurrency control from the persistence portions of the transaction. The concurrency portions execute in the foreground, serializing themselves and exchanging data as controlled by the HTM mechanism. The persistence operations occur outside the HTM and can occur in the background, safely persisting values to PM and ensuring that, in the event of failure, PM is restored to a consist state. HTM abort and retry occur exactly as they would within a HTM transaction in volatile memory, leaving PM unchanged.

Specifically, we persist an HTM transaction onto durable PM consistently by splitting it into three phases as shown in Figure 3.4 a parallel execution or concurrency phase that completes under HTM provisions (where its updates are limited to the volatile cache hierarchy), a logging phase using a Redo Log, and a decoupled, ordered durability phase that follows. A volatile Redo Log constructed during the HTM execution is persisted and then used in the ordered durability phase to cover deferred
updates of values transactionally. The figure also shows how arbitrary processor delays can occur between any of the phases.

![Figure 3.4: HTPM Transaction model splits transaction into three phases: concurrency, logging, and durability](image)

The order in which the updates get committed to PM home locations is established by using a fine-grained monotonic persistence-timestamp from within the HTM execution phase, obtained without the danger of causing inter-thread memory collisions. To prevent spontaneous cache evictions that may occur after the XEnd from corrupting PM with partial transactional updates, transaction variables are handled either by our software solution using a novel aliasing mechanism based on SoftWrAP [63], presented in Chapter 4 or our Persistent Memory controller based on WrAP [33], presented in Chapter 5.

The structure of a transaction is shown in Algorithm 1. The original HTM transaction is marked by the XBegin and XEnd instructions and embedded within a transaction wrapper marked by HTPM primitives Transaction Begin and Transaction End.

A transaction can be viewed as progressing through five states: OPEN, COMPUTE, LOG, CLOSE and COMMIT as shown in Algorithm 1. Execution begins in the OPEN state. The transaction obtains a transaction number txId allocates and initializes space in PM for its log and updates the startTime record of the log. During this state the transaction also notifies a controlling routine that a new transaction has started. The writeSet is a sequence of (address, value) pairs in the log that will be filled with the locations and values of the updates made by the HTM transaction.
Algorithm 1: HTPM Transaction Structure

Transaction Begin

——— State: OPEN ————————
1 txId = GetTransactionID();
2 Allocate a Log structure in PM;
3 Notify Open
4 startTime = ReadStartTimeStamp();
5 WriteSet = {};
6 Persist startTime to Log

——— State: COMPUTE ——————
XBegin;
// Transaction Body of HTM
// Transactional reads and writes
// Writes appended to the WriteSet
7 persistTime = ReadPersistTimeStamp();
XEnd;

——— State: LOG —————————
8 Persist the persistTime to Log in PM
9 Persist the WriteSet to Log in PM

——— State: CLOSE ————————–
10 Notify Close (txId)

——— State: COMMIT ————————
11 if strict durability required then
12 Wait until dependent transactions CLOSE
13 Transaction End

The log with its recorded startTime is then persisted using cache line write-back instructions (CLWB) and SFENCE.

The transaction then enters COMPUTE state by executing XBegin and entering the HTM code section. Within the HTM section, the transaction updates writeSet with the persistent variables that it writes. Note that the records in writeSet will be held in cache during the COMPUTE state since it occurs within an HTM and cannot be propagated to PM until XEnd completes. Immediately before XEnd the transaction obtains a second timestamp persistTime that will be used to order the
transactions correctly. This persist timestamp is obtained using the \textit{RDTSCP} instruction, a platform-wide timestamp counter provided in Intel machines [53], described further in the following section.

We do not make any assumptions about the order in which two transactions read the counter and complete their \texttt{XEnd}. It is entirely possible for a transaction to receive a smaller timestamp than a second transaction, but complete its \texttt{XEnd} after the second one. That is, there may be arbitrary delays between the last real transaction instruction and the read of the timestamp counter, and between the counter read and \texttt{XEnd}. The total ordering is discussed next in Section 3.4.1.

After executing \texttt{XEnd}, a transaction next enters the \texttt{LOG} state. It flushes its log records from cache hierarchy into PM using cache line write-back instructions (CLWB or CLFLUSHOPT), following the last of them with an \texttt{SFENCE}. This ensures that all the log records have been persisted. In addition to \texttt{startTime}, a log includes the \texttt{persistTime} time stamp that was set just prior to completing the transaction. After the \texttt{SFENCE} following the log record flushes, the transaction enters the \texttt{CLOSE} state, where the transaction signals a controlling routine its log has been persisted in PM.

The transaction is then complete and enters the \texttt{COMMIT} state. In this state, the transaction could apply its \texttt{writeSet} to PM home locations of the variables without fear of a machine crash. We refer to the updating of home locations, by copying their updated values from the log or the shadow locations, as transaction \textit{retirement}. However there is an ordering constraint that must first be satisfied.

A transaction can be retired when it is in the \texttt{COMMIT} state and has the smallest persistence timestamp among \textit{all} transactions in the system. It is not enough to have the smallest persistence timestamp of completed transactions, because there may be (laggard) transaction threads with earlier persistence timestamps that have not yet entered the \texttt{CLOSE} state.

The persistence order is maintained by a controlling routine that compares the persistence timestamp to the earliest start timestamp of all in-flight transactions. If
the transaction requires \textit{strict durability}, that is ensured that it will be replayed on an immediate and subsequent failure, it waits until the controlling routine signals the transaction is recoverable. Otherwise, the transaction immediately commits and leaves the system, and durability can lag until some future point. We call this \textit{relaxed durability}, that is an immediate failure might not replay the transaction until other in-flight transactions have completed. Management and recovery provides a continuous checkpoint of transactions that become durable and can recover to the last durable transaction.

We refer to several definitions as follows:

\begin{definition}[Competing] HTM transactions $T_1$ and $T_2$ are said to be competing if they have one or more variables in common and at least one of them is changed by one or both of the transactions.
\end{definition}

\begin{definition}[Commit] The end of a transaction. An HTM transaction is said to commit when it completes execution of XEnd.
\end{definition}

\begin{definition}[Precedes] If transaction $T_1$ commits before competing transaction $T_2$ we say $T_1 \prec T_2$.
\end{definition}

\begin{definition}[Precedence Set] The precedence set of a transaction $T$, denoted by $\text{prec}(T)$, is the set of all dependent transactions that executed their HTM before $T$. Since the HTM properly orders any two dependent transactions the set is well defined.
\end{definition}

\begin{definition}[Retirement] A transaction retires when it copies values from its write set of (address, value) pairs to home PM address locations.
\end{definition}

\begin{definition}[Strict Durability] A transaction $T$ requiring strict durability must wait in the COMMIT state until it is safe to retire; that is, $T$ waits until all preceding transactions $\text{prec}(T)$ have entered the CLOSE state. The transaction is guaranteed to be replayed on a failure after completing the COMMIT state if strictly durable.
Definition 3.7 (Relaxed Durability). A transaction $T$ may COMMIT and not wait for preceding transactions, $\text{prec}(T)$, to enter the COMMIT state, and thread execution can go about other tasks. There might be dependent lagging transactions that have not completed writing a Log and the transaction might not be replayed on failure. This is called relaxed durability; the transaction will become durable at some later point.

Definition 3.8 (Lag). A transaction $T$ with relaxed durability is guaranteed replay on failure behind subsequent execution of some number $L$ (Lag) of open transactions that have not entered the COMMIT state.

3.4.1 Ordering HTM Transactions

To create a consistent persist order among HTM transactions, we leverage a new global system time stamp counter using the new Intel instruction RDTSCP [53], or read time stamp counter and processor ID, to provide the end transaction time ordering. The RDTSCP instruction provides access to a global monotonically increasing processor clock across processor sockets [98], while serializing itself behind the instructions that precede it in program order. Older versions of the read time step counter, RDTSC, had to preceded by a serializing CPUID instruction so that previous instructions did not get reordered after the RDTSC instruction. Additionally, these serializing instructions had the unfortunate side effect of aborting an HTM transaction, rendering it useless for ordering.

The new RDTSCP instruction has the nice benefit of not reading the time stamp counter until all previous instructions have been completed. However, subsequent instructions may get reordered before the RDTSCP. To prevent the reordering of an XEND before reading the timestamp counter into registers, we save the resulting time stamp into a volatile memory address. Since all stores preceding an XEnd become visible after XEnd, and the store of the persist timestamp is the last store before XEnd, that store gets neither re-ordered before other stores nor re-ordered after the end of
the HTM transaction. We note that RDTSCP has also been used to order HTM transactions in novel transaction profiling \cite{99} and memory version checkpointing \cite{100} tools.

**Lemma 3.1** (HTM Transaction Ordering). *If two conflicting transactions $T_1$ and $T_2$ satisfy $T_1 \prec T_2$ (i.e. $T_1$ completes its XEnd before $T_2$) then the persistence timestamp of $T_1$ must be less than that of $T_2$.***

*Proof.* Let $I_j$ denote the interval between $XBegin$ and $XEnd$ for transaction $T_j$. The claim holds trivially when $T_1$ and $T_2$ are non-overlapping in time *i.e.* $I_1 \cap I_2 = \emptyset$ (Figure 3.5(a)). If $T_1$ and $T_2$ overlap in time, then $T_2$ could not have accessed any of the variables it shares with $T_1$ in the interval $I_1 \cap I_2$ (Figure 3.5(b)); otherwise one of $T_1$ or $T_2$ must have been aborted by the HTM (see Figure 3.5(c)). Hence the timestamp of $T_2$, which is taken after the last instruction of $T_2$ before $XEnd$, must occur after $T_1$ gets its persistent timestamp. Note that a similar claim cannot be made if $T_1$ and $T_2$ are non-conflicting (Figure 3.5(d)). In this case, the timestamps of $T_1$ and $T_2$ can be in the opposite order of their completion. \hfill \qed

### 3.4.2 System Checkpointing Model

In this section we describe the checkpointing model and the role of the lag parameter $L$ in more detail. Informally, transactions are allowed to complete the concurrency
portions of their transactions at the speed of the HTM. The transaction is held in a wait state for its updates to be reflected consistently in PM, but its updates in volatile memory are available for use by other transactions. In the default policy, a transaction waits to commit until its updates are persistent. In practice, transactions may be allowed to complete early even before their updates are persisted, with the understanding that the updates may be lost in an inopportune machine crash.

The lag parameter \( L \) reflects how many transactions can be allowed to complete prior to being persisted; \( i.e. \) how far ahead of the persistence wavefront we allow the computation wavefront to advance. \( L = 0 \) reflects the default conservative policy, see the following section on Transaction Durability. Different applications may choose different threshold values of \( L \) depending on the perceived benefit of early completion versus guaranteed persistence. Note that in the event of failure the system will always recover to a consistent past state, and system consistency is not affected by the early commit of some transactions.

**Formal Checkpoint Model:**

The execution of concurrent conflicting transactions are serialized by the HTM. For any two conflicting transactions \( T_1 \) and \( T_2 \) one can define a relation \( \prec \) such that \( T_1 \prec T_2 \) if \( T_1 \) precedes \( T_2 \) in the execution order defined by the HTM. Define a valid order to be a sequence of completed transactions \( T_1, T_2, \ldots, T_n \) in which \( T_i \) occurs earlier than \( T_j \) in the sequence if \( T_i \prec T_j \). A consistent checkpoint is any prefix \( T_1, T_2, \ldots, T_k, \ k \leq n \) of a valid order. On recovery from a failure our algorithm will restore the state of PM to a consistent checkpoint by replaying the log of the retired transactions. A consistent checkpoint will reflect a prior, valid transaction state that may differ (in an inconsequential way) from any state of volatile memory during transaction execution.

**Example:** Consider conflicting transactions \( T_1, T_2, T_3 \) (with all variables initially 0) performing the following transactional updates \( \{x = 1, y = 10\}, \{y = 11, z = 20\} \)
and \( \{x = 2, z = 21\} \) respectively, and suppose they are serialized by the HTM in the ordered sequence \( S_1 = (T_1, T_2, T_3) \). Similarly, assume conflicting transactions \( T_4, T_5 \) with transaction updates \( \{a = 5, b = 6\}, \{b = 7, c = 8\} \) are serialized as \( S_2 = (T_4, T_5) \).

Any interleaving of \( S_1 \) and \( S_2 \) that maintains individual sequence order is a valid order: for instance \( (T_1 \ T_4 \ T_2 \ T_3 \ T_5) \). Note that the definition does not insist on an ordering between two non-conflicting transactions like \( T_2 \) and \( T_4 \) due to the inherent ambiguity in defining their relative order. That is the orderings of volatile memory and persistent memory for non-conflicting transactions may differ. However, this cannot happen for conflicting transactions.

Figure 3.6: Transaction checkpoint model allows memory to recover to a valid state.

**Example:** Figure 3.6 shows the situation where non-conflicting transactions can be recovered to a valid, but unobserved, state. Suppose that the actual execution of transactions completed in the order \( (T_1 \ T_2 \ T_4) \) as shown in the Figure. At \( t_1 \) the state of volatile memory is \( x = 1, y = 11, z = 20, a = 5, b = 6 \) and subsequently the machine crashes at time \( t_2 \). It is possible that the system recovers to a consistent checkpoint \( (T_1 \ T_4) \), which will restore the state to \( x = 1, y = 10, z = 0, a = 5, b = 6 \), since the
non-conflicting transactions $T_2$ and $T_4$ do not impose any correctness ordering. Even though unobserved by memory directly, this is a perfectly acceptable valid state of memory. This state is possible due to a delay after $T_4$ read its persistent timestamp at $t_0$ with RDTSCP before ending the concurrency section. The transactions are still ordered by the persistent timestamp with RDTSCP, which is the actual completion time of all of the work in the transaction.

3.4.3 Transaction Durability Model

Our model can retire a transaction $T$ when it has persisted its log and $T$ has the smallest persistence timestamp among all transactions in the system. However, there may be (laggard) transactions with earlier persistence timestamps that have not yet persisted their logs and entered the CLOSE state, e.g. their persistence timestamp is unknown. To manage persistence for transaction durability, we use two service queues, colored Blue for start and Red for finish.

A Blue Queue indicates the order in which transactions have started via the start timestamp of the transactions. When a transaction starts, it is placed in the Blue Queue. After a transaction has recorded its persist timestamp and completes the concurrency section, it is added to a Red Priority Queue, ordered by increasing persist timestamps. It is then removed from the Blue Queue. After the transaction has completed writing its log, it is marked as complete in the Red Priority Queue.

When the head of the Red Priority Queue is marked complete, the priority (transaction concurrency section end time) is compared to the time value of the head of the Blue Queue, the earliest transaction (smallest start timestamp) that has not recorded its persist timestamp. If the persist timestamp of the transaction at the head of the Red Priority Queue is less than the transaction start timestamp of the head of the Blue Queue, then the transaction can be retired. There are no other transactions from the Blue Queue that can possibly go to the head of the Red Priority Queue. Lemma 3.2 indicates durability by persistence timestamp ordering.
A transaction requiring **strict durability** waits until it is the head of the Red Priority Queue and has the smallest persist timestamps in the system, Lemma [3.2.](126x672) A transaction can request **relaxed durability** and immediately COMMIT and continue processing; that is, it need not wait until it has the smallest persist timestamp in the system.

**Lemma 3.2** (HTM Transaction Durability). *If the transaction $T$ at the head of the priority queue (PQ) has an persist timestamp that is smaller than the minimum start timestamp of all transactions not in the priority queue (PQ), then $T$ has the smallest persistence timestamp of all transactions in the system.*

**Proof.** Partition the transactions be into two sets $R$ and $B$ consisting of transactions in the PQ (i.e. those in the COMMIT state) and those not in the PQ respectively. Suppose transactions record a start timestamp $\text{startTS}$, then a persistence timestamp, followed by an ending $\text{endTS}$. Since the PQ is ordered in increasing order of persistence timestamps, the element at the head of the PQ has the smallest persistence timestamp among all transactions $R$. If the $\text{endTS}$ of a transaction $P$ is smaller than the $\text{startTS}$ of a transaction $Q$, then the persistence timestamp of $P$ (which is less than its $\text{endTS}$ must be less than the persistence timestamp of $Q$ ((which will be greater than its $\text{endTS}$). Hence the persistence timestamp of $T$ must be less than the persistence timestamps of all transactions in $B$.  

### 3.4.4 Transactional Durability Extensions

Some transactions might require **strict durability**, after the transaction end routine completes, the transaction must be durable (fully recoverable on failure). This means that the transaction durability cannot lag, e.g Lag must be set to $L = 0$. This ensures that the transaction is durable before subsequent program instructions are executed. In like fashion, some applications might have needs similar to a memory fence, e.g. an application may require a) that all transactions be completed before proceeding or b) all transactions currently open must close before proceeding but new transactions
may open. We propose two new software primitives for these handshakes between the persistence thread and the remainder of software:

**Definition 3.9 (TXFence).** A transaction fence - this returns only after all open transactions are durable but new transactions may start in other threads.

**Definition 3.10 (TXBarrier).** A transaction barrier - this disallows opening a new transaction until all transactions are durable.

TXFence is useful when reading variables non-transactionally, but ensuring that if variables have been potentially updated in preceding transactions, then their updated values have been copied to home locations. Values may be present only in cache with a recoverable log or be present in home PM locations with a purged log. TXBarrier allows for multiple threads with concurrent transactions to converge on a single point and read shared values non-transactionally after completing a parallel section.

### 3.4.5 Logging and Recovery

Each completed transaction saves its log in PM. The log holds records $startTime$ and $persistTime$ obtained by reading the platform timer using RDTSCP instruction. We refer to these as the start and end timestamps of the transaction. The start timestamp is persisted before a transaction enters its HTM. This allows the recovery routine to identify a transaction that started but had not completed at the time of a failure. Note even though such a transaction has not completed, it could still have finished its HTM and fed values to a later dependent instruction which has completed and persisted its log. The end timestamp and the write set of transaction updates are persisted after the transaction completes its HTM section, followed by an end of log marker. There can be an arbitrary delay between the end of the HTM and the time that its log is flushed from caches into PM and persisted.

The recovery procedure is invoked on reboot following a machine failure. The routine will restore PM values to a consistent state that satisfies persistence ordering
by copying values from the writeSets of the logs of qualifying transactions to the specified addresses. A transaction \( \tau \) qualifies for log replay if and only if all earlier transactions on which it depends are also replayed.

### 3.4.6 Composability

Transaction composability refers to multiple transactions that can be composed or used together in a single transaction that operates atomically. For instance, two transactions can compose together and become part of a higher-level transaction. In many systems transactions roll-up and become sub-routines in a higher-level transaction. Consider a transactional routine which creates and inserts an element into a linked-list. If a user needed to insert two elements into a linked list atomically, e.g. neither or both, then a higher transaction could be created that used the transaction insert routine twice, one for each elemental insert. See Figure 3.7a. If \( T_1 \) and \( T_2 \) are used inside a higher transaction \( T \) and are blindly rolled up, then, in our model, the persistence portion of \( T_2 \) will cause \( T_1 \) to abort, since the persistent cache flushing or logging would be performed inside the HTM concurrency section of \( T_1 \).

![Figure 3.7: Composability of Transaction T which calls sub-transactions T1 and T2.](image)

We roll up the concurrency and persistence sections of sub-transactions separately as shown in Figure 3.7b. The concurrency sections of \( T_1 \) and \( T_2 \) are rolled up into the concurrency section of \( T \). We keep a transaction depth counter and as sub-transactions are opened and closed the counter is incremented and decremented. New values for variables in the sub-transactions \( T_1 \) and \( T_2 \) are added to the log for \( T \),
which are persisted in correct encounter-time order when the outermost transaction T1 is closed.

### 3.4.7 Example

We illustrate with an example to show the correct processing order of completed transactions and persistence. It illustrates situations where transactions that have started earlier but may have been delayed before performing concurrency sections or may have been delayed for a long period before saving a timestamp or completing logging. Figure 3.8a shows an example set of four concurrent transactions, T1-T4, happening concurrently. In this example, we show T1-T4 split into states, specifically, HTM concurrency (COMPUTE), shown in vertical lines, and LOG, depicted with slanted lines. A transaction is shown in either the Blue Queue with only a start timestamp recorded or in the Red Queue, ordered by persist timestamps. Bold and underline indicate the log is written, and a circled transaction indicates that the transaction is durable, it can be persisted or is recoverable on failure. Table 3.8b illustrates the contents of the logical Red and Blue queues at each time step indicated in the figure.

First, at time $t_1$, T1 opens and is placed with the start time in the Blue Queue. At times $t_2$ and $t_3$, transactions T2 and T3 also open and are added to the Blue Queue with their start timestamps.

T2 then completes its concurrency section and records its persist timestamp at time $t_4$ and begins writing its logs. It is removed from the Blue queue and added to the Red Priority Queue. Transaction T4 starts at time $t_5$ and is added to the Blue queue. T3 completes concurrency section and records its persistence timestamp at time $t_6$, also moving from the Blue to the Red queue. T3 is now ordered after T2 for recovery, though neither have completed persisting their logs.

At time $t_7$, transaction T3 has completed writing its logs and is marked completed in the Red queue, denoted by bold and underline in Table 3.8b. However, T3 is not
(a) Example Transactions T1-T4 with HTM Concurrency and Logging Phases

(b) Transaction Durability with Blue Queue and Red Priority Queue

Figure 3.8: Example Transactions T1-T4 and Blue and Red Persistence Queues

durable at this point since it is not the head of the Red queue, being ordered behind T2. Additionally, T1, the head of the Blue queue, has a smaller start timestamp than the persist timestamp of T3, which could potentially place T1 at the front of the Red queue. When T2 completes writing of logs at time $t_8$, it is marked as complete in the Red queue. However at this time $t_8$, both T2 and T3 are not durable (recoverable) as shown in 3.8b as T2 has a persist timestamp that is greater than the start timestamp of T1. It would be unknown by a recovery process if T1 simply had a delay in persisting its log and T2 had transactional values dependent on T1.

T1 finally completes its concurrency section and writes its persist timestamp at $t_9$. Since the persist timestamp of T1 is safely persisted and known at recovery time, transaction T2 is now fully durable (recoverable) as shown circled in 3.8b. A routine can safely copy (or retire) the write set of T2 to home PM locations. However, at this time, T3 is not fully durable since it is waiting on T4, which started before T3 completed its concurrency section, and T4 has not yet recorded the persist timestamp.

At time $t_{10}$, T4 records its persist timestamp and moves from the Blue to Red
Queue. Note that T4 has a persist timestamp before T1, so it is placed before T1 in the Red Priority Queue. At this time, T3 is now fully durable and its transactional writes may be copied to PM immediately or recovered on failure. T1 completes log writing at \( t_{11} \) and is marked as complete, but is behind T4, which has not finished writing its logs, in the Red Priority Queue. At time \( t_{12} \), T4 completes writing its logs and both T4 and T1 can be orderly retired; all transactions are durable.

**Strict Durability:** Suppose a transaction requires *strict durability* during its COMMIT state, ensuring that once complete, the transactional writes will be reflected in Persistent Memory if a failure were to occur. If T4 requires strict durability, it is simply durable at the end as there are no open transactions when it completes. However, T1, T2, and T3, have other constraints. A transaction requiring strict durability is only durable when it is fully recoverable. T1 must wait until step \( t_{12} \) if it requires strict durability as it might have dependencies on T4. T2 is strictly durable at time \( t_{9} \) when T1, which started earlier, records its persist timestamp. At time \( t_{10} \), T3 is strictly durable when T4, which started before T3 completed its concurrency section and could have introduced transactional dependencies, records its persist timestamp, indicating T4 completed the HTM section later than T3.

3.4.8 HTM Performance Tuning for Durability

Several factors affect HTM performance including transaction size, data structure size, and fallback path parameters. Additionally, the method of timestamp acquisition for PM ordering can affect performance. For instance, we show a shared counter in the HTM section drastically reduces concurrent transaction throughput.

Transactions utilizing HTM for concurrency control typically require a fallback path. A fallback path is a way for a transaction to execute outside of the HTM section while still ensuring isolation. This path is needed as a transaction may abort for any number of reasons including too many writes, updating conflicting data, certain system calls, I/O, or other hardware deemed reasons. To ensure isolation in a fallback
path, a global lock is typically used to enable ease of programming. More complicated fallback locking mechanisms are possible, but they require complex programming to track loads and stores in software and utilize an STM, while also tracking in the HTM section.

**Algorithm 2: HTM With Fallback Locking and Timestamp Counters**

```plaintext
HTMBegin ()
numRetries = 0;

Read and Record Start Timestamp;
while (1) {
  if ( ++numRetries > maxRetries)
    break; // Reached max retry, take the fallback lock.
  status = XBEGIN();
  if (status == XBEGIN_STARTED) {
    if (!fallback.isLocked())
      return; // Success in starting HTM transaction.
    XABORT() // Someone has the fallback lock. Retry.
  }
  if (status indicates explicit and transaction can retry) {
    // Wait until the fallback lock is free.
    while(fallback.isLocked()) pause();
    continue; // Try and start HTM section again.
  }
  if (status indicates cannot retry)
    break; // Break and take the fallback lock.
  pause(); // Pause for conflicts and loop to try again.
}
// Serialize and take the fallback lock.
fallback.Lock();

HTMEnd ()

Read Persist Timestamp;
If fallback is locked then it is held by this transaction.
if (fallback.isLocked())
  fallback.unlock();
else
  XEND();
Return Persist Timestamp;
```
Algorithm 2 shows an HTM interface that utilizes a global fallback lock. The algorithm also includes a placeholder for recording start and persistence timestamps. A transaction attempts several HTM acquisitions up to maxRetries before taking the fallback lock.

To begin an HTM transaction, a loop is entered until either a transaction successfully enters a hardware managed HTM section of code or exceeds its maxRetries attempts to do so. If exceeding the maxRetries, the transaction exits the loop and acquires the fallback lock, serializing behind any other transactions that might currently hold the lock. If the HTM section is successfully entered, then the transaction still has to check to make sure that no other transaction is holding the fallback lock. If no transaction holds the fallback lock, the HTM section, which is speculatively managed by the processor hardware, is entered. If another transaction does hold the fallback lock, then the HTM transaction must explicitly abort. Then, after waiting until the lock is released, the transaction may retry until succeeding or ultimately exceeding the maxRetries. While waiting to retry or for the fallback lock to be released, the transaction pauses to allow for the conflict to complete.

For ending an HTM transaction, first the fallback lock has to be examined. Consider a transaction that is in an HTM section managed by the hardware and another transaction has taken the fallback lock after the HTM transaction started. When examining the fallback lock at HTM end will force the HTM transaction to abort the speculative execution and retry. This performance is correct, as the fallback lock does not guard against individual writes and the two transactions could be conflicting in their write sets. Therefore, if the transaction observes that the fallback lock has been taken, it is guaranteed to be the holder of the fallback lock and releases the lock. Otherwise, the HTM section completed successfully by the hardware and the transaction can end the HTM.

For persisting transactions to PM, our solution, as detailed in section 3.4, requires two timestamps used for ordering. The first timestamp is recorded as the start
timestamp, before any other operation executes. Recording is performed before HTM section is entered so as persisting the start timestamp for recovery will not abort the transaction. The persist timestamp, shown in the HTM End routine, shows the reading of the timestamp as the last operation before the transaction end. As it is the last operation, as described in section 3.4.1, all processor read and write sets are complete, and any conflicting transactions have been aborted. After the ending routine completes the ending timestamp can be persisted.

Performance Experiments:

To evaluate the transaction size, data structure size, fallback path choices and timestamp mechanisms, we implemented the HTM begin and ending routines as described above and created a test application. We tested on a machine equipped with an Intel® Xeon E5-2686 v4 (Broadwell) processor with 36 hyper-threaded cores, as described in Chapter 1.6, and 512 GB of memory running Red Hat Enterprise Linux 7. HTM transactions were implemented with Intel Transactional Synchronization Extensions (TSX) [38] using a global fallback lock as described.

The test application simply creates a hash table in main system memory and creates a number of threads. Each thread performs a number of transactions, where each transaction inserts a configurable number of elements into the hash table. The transaction time is recorded along with HTM metrics such as aborts and fallback lock rates. We built our software using g++ 7.3.1. Each measurement reflects an average over ten repeats with small variation among the repeats.

Figure 3.9 shows the performance of HTM transactions versus the data size on which the transactions operate. A hash table is created of a given size and 4, 8, and 16 threads (shown by the separate lines) are created and perform transactional inserts into the table. Each thread performs 1M transactions, where each transaction consists of inserting 30 random elements. The HTM is configured to try 16 times before taking the global fallback lock. Figure 3.9a shows as the data size increases, trans-
(a) HTM Transactions per second for increasing hash table data size.

(b) Average number of aborts per HTM transaction and hash table data size.

Figure 3.9: HTM performance versus hash table data size.

Actional throughput increases up until a point about 3M elements where transaction throughput decreases. Additionally, it shows as the number of threads increase, so does the throughput. The reason for the performance trends is shown in Figure 3.9b.

As the data size increases, the number of HTM aborts decreases also up to a size of 3M elements. With an decreasing numbers of HTM aborts, the HTM transactions don’t have to retry as many times and can complete faster. A smaller data size means that threads have a higher probability that they will be conflicting. That is, when inserting the 30 elements per transaction, a smaller data size increases the chances that two threads update the same cache line.

The point 3M is significant as the size of the last level cache is 45 MB. Each element in the hash table is 8 bytes, yielding a total size of 24 MB. At the next point, 10M, the total data structure size is 80 MB, which exceeds the processor cache capacity. Since the processor cache capacity is exceeded, transactions have increasing probability (with increasing data size) that it will have to fetch a value from main memory. When a transaction reads from main memory and not cache, it increases the execution time and increases data collisions and aborts.
Figure 3.10 shows the effect of increasing the transaction size on the number of aborts and fallback lock taken rate. The HTM is configured to try 16 times before taking a global fallback lock, and the hash table size is 1M elements. Each thread performs 1M transactions, where each transaction inserts the specified number of elements. Figure 3.10a shows the average number of HTM aborts per transaction for increasing the write set size. As the number of threads are increased from 4 to 8 and 16, the number of aborts per transaction also increase. Increasing the write set from 10 to 50 increases the number of aborts by slightly more than 5x. If write sets become too large, then a transaction can never complete under the speculative execution of an HTM, as reads and writes are tracked in hardware buffers with limited capacity. Figure 3.10b depicts the increase in the fallback lock taken rate with increasing write set size. After the failed 16 attempts at entering the processor HTM section, a transaction takes the fallback lock. This means that subsequent transactions might also be delayed and increase their abort rate, thus increasing the rate of the fallback lock.
Figure 3.11: Transaction throughput and fallback locking versus max number of HTM attempts.

(a) Fallback lock taken rate for number of transactional retries.

(b) Transaction throughput versus number of retries before fallback lock is taken.

transaction retries before taking the fallback lock. Each thread performs 1M transactions, where each transaction inserts 30 elements into a hash table of 1M entries. Figure 3.11a shows the average fallback lock taken rate versus the number of HTM attempts. Figure 3.11b shows the transaction throughput versus the HTM attempts before the fallback lock is taken. At zero attempts, all transactions take the fallback lock (rate of 1) and the throughput is very low at 1M transactions per second. At this point increasing the number of threads decreases performance since competition on the single lock has increased. It is interesting to note that the number of retries to reach maximum transactional throughput increases with the number of threads. As the number of retries is increased, fewer threads take the fallback lock allowing for more threads to utilize HTM. With higher HTM usage, transactional throughput increases. If the number of retries is too high, then a thread can retry for a long time (or forever) and transactional throughput can start to decrease.

The timestamp acquisition method can also affect performance. Consider again Algorithm 2 and the reading of the start and persist timestamps. The start timestamp in the HTMBegin function is read outside of the HTM section, so it can be a shared
variable counter that is updated with an atomic fetch and increment instruction or an RDTSCP instruction, as described in Section 3.4.1. The persist timestamp in the HTMEnd function is inside the HTM section. Therefore, simultaneous updates to a shared counter can sharply increase the transaction conflict rate.

Figure 3.12 shows several methods for the timestamp acquisition as the number of threads are increased. Similar to the previous experiments, the hash table is constructed with 1M elements and each thread performs 1M transactions, each of which insert 30 elements into the hash table. Transactions retry 16 times before taking the fallback lock. The figure shows HTM performance with no counters and combinations of RDTSCP and shared variables for the start and persist timestamps. As shown in the figure, using RDTSCP for just the persist or even both timestamps has little to no impact on performance. This configuration is utilized in the Persistent Memory Controller described in Chapter 5.
Figure 3.12 also shows how a single shared counter in the HTM section can reduce throughput by over 50 percent. Since the counter is shared in the HTM, all transactions that overlap in time also have the potential to abort due to the conflicting write set, which includes the counter. A shared counter outside of the HTM section and RDTSCP within the section, shown as Atomic + RDTSCP, has slower performance when compared to no counters. However, as will be shown in the next chapter, implementing the start timestamp as an atomic counter update is advantageous. The start timestamp as a monotonically increasing counter can be used to order transaction starts, as an index into a lock-free queue, as an indicator of a lagging transaction, and function as a minimum transaction durability wavefront.

3.5 Summary

Hardware Transactional Memory improves parallelism in applications and simplifies concurrent programming, making all updates visible in cache memory on transaction completion. However, this instant update creates a window of vulnerability for concurrent transactions that can corrupt Persistent Memory data structures on failure. In this chapter, we presented an approach to create durable, concurrent transactions that utilize HTM for concurrency control and PM for durability. By introducing a small lag parameter, we create a continuous wave of transactions that are persisted to PM fronted by transactional HTM concurrency sections. Additionally, we presented relaxed and strict durability, whereby a transaction can choose to let durability lag for increased performance or require durability on completion for ensured recovery.

Our model is the first solution to unify HTM with PM without any changes to the processor or cache hierarchy. In the following chapters we present two implementations of the solution approach. In Chapter 4 we present a software only solution which executes on existing Intel® hardware and has a novel lock-free management algorithm. In Chapter 5 we describe and evaluate a Persistent Memory controller that works with HTM for ACID transactions to PM.
Chapter 4

Continuous Checkpointing HTM for PM

This chapter presents a software based approach and implementation of the Hardware Transactional Persistent Memory model presented in Chapter 3. The approach creates a continuous checkpoint for durability, or persistence wavefront, of HTM transactions to PM. The implementation utilizes variable aliasing so that after an HTM concurrency section ends, transactional values that were updated do not get written back to Persistent Memory until it is safe to do so. The approach and implementation is the first solution that couples HTM with PM on existing hardware, as no costly changes to well established protocols, caches, or processor hardware are required.

First, the background on variable aliasing is presented. Next, the overall software approach as it relates to the model from Chapter 3 is discussed in Section 4.2. The implementation is then presented in Section 4.3 with a full discussion on the lock-free algorithms. An extensive software evaluation is performed on Intel® hardware and the results presented in Section 4.4. Finally, future optimizations and a summary follows.

4.1 Background on Variable Aliasing for PM Transactions

There are numerous methods of persisting transactions atomically to PM as discussed in Chapter 2. Consider the Redo Logging approach as discussed in Section 2.3.3. A Redo Log is created while values are written during transaction execution. Variable writes are aliased into a DRAM shadow area, or alias table, to prevent unwanted cache evictions after transaction end from spilling into PM out of transaction order. After a transaction completes, the updated values may be copied from the log or
shadow area to home PM locations. We earned a patent on this Persistent Memory transactional atomicity technique [101].

With concurrent transactions, two-phase locking (2PL) can be used for concurrency control as discussed in Section 2.4.1. There are two options for aliasing with 2PL, thread-local or global aliasing.

Local aliasing is a technique where each thread has its own alias table [102], or Local Alias Table or LAT, as shown in Figure 4.1a. After taking appropriate locks to ensure isolation, each thread performs loads and stores into its own table. On a load, if a variable is not present in the table, it is loaded from PM. When completing the transaction, before locks are released, variables must be copied into the cache hierarchy to allow for other threads to read the most recent value. Before removing the transaction log, values must be written back to PM.

In global aliasing all threads load and store values into the same alias table. A thread does not have to store values back to the cache hierarchy or PM before releasing locks since the most up to date value will be read by other transactions consulting the alias table. However, transaction logs cannot be removed until values from the
In SoftWrAP, a two-phase, double-buffered retirement scheme is used [32]. In the SoftWrAP approach, two global alias tables are utilized as shown in Figure 4.1b. An active table records the stores to all variables in open transactions. A second table retires values from completed transactions concurrently in the background. On a transaction load, if a value is not found in the active table, the retiring table is next checked before finally checking PM if not found. Once the retiring table is complete it is marked empty and associated logs for the transactions in the table may be removed. Once the size of the active table crosses a configurable threshold, the active table can start retiring and the empty table be switched to the active table.

In our software approach to couple HTM with PM, we utilize aliasing to prevent unwanted cache evictions from corrupting PM immediately after the HTM section ends. As shown in the following section, when utilizing HTM for the concurrency control, a global aliasing table with a common aliasing transpose must be used.

### 4.2 Continuous Checkpointing HTM for PM Approach

Our approach persists an HTM transaction onto durable PM consistently by splitting it into three phases as described in 3.4. A parallel execution phase that completes utilizing HTM for concurrency control (with updates in volatile cache hierarchy), a Redo Logging phase, and a decoupled, ordered-durability phase. A persistence timestamp is read at the end of the HTM phase using the RDTSCP instruction as described in Chapter 3.4.1. A persistence management thread manages transaction durability and provides a persistence wavefront or continuous checkpoint of HTM transactions to PM. We call our approach cc-HTM. The model is adapted to execute on existing Intel hardware as follows.

The persistence timestamp is bracketed between two other timestamps that are obtained outside the HTM phase - a start timestamp and an end timestamp. As described shortly, this bracketing provides the required temporal reasoning about a
consistent order for committing potentially overlapping updates from HTM transactions that may be partially concurrent with one another. The timestamped logs are funneled into a persistence management thread for durability ordering. Threads that complete transactions may either wait until updates are durably accepted into PM in the right order, or proceed to perform other operations without waiting. The model gives rise to consistency in PM with the flexibly for delayed and efficiently parallel transactions in the volatile cache hierarchy.

To prevent spontaneous cache evictions that may occur after XEnd from corrupting PM with partial transactional updates, all transaction variables are aliased to shadow volatile locations within the HTM so the updates do not directly percolate to PM. This accomplished using an Alias Table that implements the shadow locations. Further details are discussed in this section and the implementation is described in detail in Section 4.3.3.

The structure of a transaction is shown in Algorithm 3. The original HTM transaction is marked by the \texttt{XBegin} and \texttt{XEnd} instructions and embedded within a transaction wrapper marked by cc-HTM primitives \texttt{Transaction Begin} and \texttt{Transaction End}. \texttt{TXQueue} is the main data structure that tracks the progress of a transaction.

A transaction goes through five states in the course of its execution. The \texttt{COMPUTE} state is where it performs the specified HTM transaction. The \texttt{COMMIT} state where it waits for its updates to be persisted to PM before returning. A transaction maintains three timestamps in the course of its execution: \texttt{start} timestamp (startTS) and \texttt{end} timestamp (endTS) are values of a monotonically-increasing global transaction counter that assigns a unique id to each transaction; the \texttt{persistence} timestamp is obtained by reading the platform-wide timestamp counter provided in Intel machines [53].

Execution begins in the \texttt{OPEN} state. The transaction obtains a unique transaction number \texttt{myId} that serves as the start timestamp of the transaction, which is recorded
Algorithm 3: cc-HTM Transaction Structure

Transaction Begin
——— State: OPEN ————————
1 myId= FetchAndIncrement(TxCounter);
2 myIndex= myId % Qsize;
3 TXQueue[myIndex].startTS= myId;
——— State: COMPUTE ——————
XBegin;
// Transaction Body of HTM
// All reads and writes to transaction variables are aliased to shadow locations.
// The final value of any persistent variable is recorded in a log.
4 myPersistTime= RDTSCP();
XEnd;
——— State: LOG —————————
5 TXQueue[myIndex].endTS= Read(TxCounter);
6 TXQueue[myIndex].persistTime= myPersistTime;
7 Persist the transaction log in PM;
——— State: CLOSE ————————–
8 TXQueue[myIndex].logAddress=PM Address of log;
9 Place TXQueue[myIndex] into a priority queue ordered by their persistTime fields
——— State: COMMIT ————————
// Wait for transaction to be retired
// Retirement thread updates PM home locations when safe, and notifies waiting thread when it is done.
Transaction End

in the entry (myIndex) of TXQueue (steps 1-3).

The transaction then enters the COMPUTE state with an XBegin instruction, which starts the execution of the HTM controlled transaction. To sidestep problems of uncontrolled cache evictions of transaction variables following XEnd, updates are made to shadow DRAM locations implemented as an Alias Table. Writes are also logged in a private log that will be written to PM after XEnd.

After reading the persist timestamp via RDTSCP (step 4) and executing XEnd,
the transaction enters the LOG state. The current value of the transaction counter TxCounter is recorded as its end timestamp, and the RDTSCP timer value that was previously read is its persistence timestamp (steps 5, 6). The transaction then flushes its log records to PM using cache line writebacks together with an SFENCE as recommended by Intel (step 7). The transaction log is now persistent, and the transaction enters the CLOSE state. Ordering is updated by inserting the finished transactions in a priority queue ordered by their persistence timestamps (steps 8, 9).

The transaction can be retired when it is in the COMMIT state and has the smallest persistence timestamp among all transactions in the system. This condition can be ensured by checking that all transactions not in the priority queue began after the transaction at the head of the priority queue ended i.e. by comparing the end timestamp (endTS) of the transaction at the head of the priority queue with the smallest start timestamp (startTS) of in-flight transactions i.e those not in the priority queue.

A separate persistence thread is responsible for retiring transactions when it is safe to do so, and signaling the waiting transaction. To prevent PM write buffers on the back end from filling up and stalling the front end, our retirement thread copies entries in the log into the cache hierarchy and does not force persistence onto PM. The persistence management thread creates a retirement wavefront whereby transactions with smaller start timestamps are guaranteed to be durable. See Lemma 4.1 below. Periodically, the persistence management thread after completing a configurable set of logs or time, flushes the cache, which persists PM values after a persistent memory fence, and safely removes processed logs.

In Section 4.3.4, we describe an efficient concurrent lock-free data structure to implement the priority queue and safety checking operations, along with details of managing of the shadow variables with the Alias Table.

Lemma 4.1 (Transaction Retirement Wave). All transactions with a start time less than the minimum start time of all transactions in the management queue (including
those that have not been retired) have been retired.

Proof. From Lemma 3.2 a transaction can only be retired if 1) it has the minimum persist timestamp and 2) its persist timestamp is less than the start timestamp of all open transactions without a persist timestamp. Assume that there exists a transaction T that has a start timestamp less than all start timestamps for transactions in the management queue, TXQueue. If T had not been retired it would either have a persist timestamp and be queued for orderly retirement or not have a persist timestamp and be in the set of open transactions. In both cases, open transactions and those queued for retirement, the transactions are still contained within the management queue. Therefore T, with a start timestamp less than all start timestamps in the management queue, has been retired.

\[\square\]

4.2.1 Alias Table Model

In this section we present the Alias Table Model and show aliasing preserves transaction isolation and correctness. We also present the model for reclaiming space. A DRAM based Alias Table is used to alias all loads and stores in the HTM concurrency section to prevent unwanted cache evictions from corrupting PM immediately after XEnd. The alias table may be implemented as a key-value store, where the key is the PM address of the variable. An entry in the Alias Table holds a variable value, PM address, and transaction start time for the stored value.

As long as the same aliasing transpose or key lookup for all transactions are used, then isolation for HTM concurrency sections is preserved. Suppose two transactions have conflicting read-write sets and both use the same transpose to alias their read-write sets. Then the two transactions will still have conflicting read-write sets. See Lemma 4.2. Additionally, if all of the transactions in an application use the same common global alias transpose, then HTM correctness is preserved across all transactions. See Lemma 4.3 below.
Lemma 4.2 (Aliasing with HTM). Transactions $T_i$ and $T_j$ with conflicting read-write sets $S_i$ and $S_j$ respectively, will have conflicting read-write sets, $S'_i$ and $S'_j$, using a common aliasing transpose, $A(S_n) = S'_n$.

Proof. Let $S_i \cap S_j$ indicate two conflicting read-write sets. If $S'_n = A(S_n)$ indicates a common aliasing transform, then $A(S_i) \cap A(S_j)$. For example, let transaction $T_1$ write to variables $X$ and $Y$ and read from variable $Z$, then it will access $X'$, $Y'$, and $Z'$. Let transaction $T_2$ accesses $X$, $Y$ and $Z$, then it will similarly access $X'$, $Y'$, and $Z'$ since it uses the same alias mapping. Therefore $T_1$ and $T_2$ which conflicting read-write sets also conflict when using the same alias mapping. □

Lemma 4.3 (Global Aliasing Preserves HTM Correctness). If all concurrent transactions $T$ utilize the same aliasing transpose $A$, then HTM correctness is preserved.

Proof. It follows that, under the same timing scenarios, if transactions $T_1$ and $T_2$ abort when running using un-aliased variables and a variable always aliases to the same location across concurrent transactions, then transactions $T_1$ and $T_2$ are also guaranteed to abort by the hardware. Therefore, global aliasing will preserve correctness. However local aliasing will not guarantee aborts as one transaction’s alias of $X$ to $X_1'$ would not be the same as another transaction’s alias of $X$ to $X_2'$. □

Reclaiming Alias Table Space:

Once a transaction is safely committed to PM, values may be pruned from the Alias Table to make space for future writes. However, clearing up space or pruning variables in the Alias Table faces several challenges:

- An external thread to a transaction which removes or updates an element in the Alias Table can cause a transaction to abort. If both a transaction and the external thread access the same cache line, the transaction can abort. Additionally, concurrently removing elements from the Alias Table while executing...
transactions limits implementation choices to data structures that support concurrent delete operations.

- Relying on update time of a variable when deciding to prune is not sufficient. Suppose a write to a variable might occur at time \( t_1 \) in transaction T1 and a second transaction T2 may subsequently complete and be ready for persisting to variables at time \( t_2 \). If T1 has published completion times but not finished writing logs, then removing variables belonging to T1 from the Alias Table could cause corruption, since the PM would not have the new values from T1 safely persisted.

- Simply comparing the value in the Alias Table to the value being retired is not correct. Consider a variable X that was written with a value 4 in transaction T1, then updated to 5 in T2, then back to 4 in T10. When T1 completes, simply comparing the value of X=4 to \( X'=4 \) is not sufficient for removal from the Alias Table. If \( X'=4 \) is removed from the table, when T2 is complete and saves X=5 into PM, subsequent transactions T3-T9 will read the incorrect value of X, as X will not be in the Alias Table and they will reach back to main PM and retrieve X=5 and not X=4.

We solve the challenges above by tagging entries in the Alias Table with the start time of the transaction that updated the variable most recently. If a variable is written to and already exists in the table, then we update the variable with the new transaction start time that wrote the variable. Note that this start time tag in the table can actually go backwards if a transaction that started earlier writes a variable value later. This is fine since values are written in real-time order into the table; any conflicting concurrency writes are handled by the underlying hardware and will abort if necessary.

All transactions that started before the minimum start time of all open transactions have been retired. Therefore, any entry in the Alias Table with a start time
less than the minimum of all open transactions can be removed from the table. See Lemma 4.4.

The variables updated in a transaction need not be deleted from the Alias Table when the transaction completes. Instead, the variables and can be kept as long as there is free space as the values in the table reflect the most up-to-date value. If the Alias Table runs out of space and there are no free entries in the table, then a transaction requiring a new entry must abort. However, with a large, associative table or multiple hashes, the probability is very low; a transaction can retry after an abort since the minimum blue or red time may have moved up after a transaction has been retired and allowed one of the existing entries to be removed.

**Lemma 4.4** (Safe Alias Table Pruning). A variable from the Alias Table can be removed if the variable in the Alias Table was written with a transaction start time that is less than the minimum of the blue or red queues transaction start times.

*Proof.* Since all transactions and all variables in the transactions have been retired with start times less than the minimum start times of un-retired transactions in the blue or red queues, then we can remove variables from the alias tables with retired variables. These variables will have been retired if tagged with transactions start times from prior transactions.

### 4.3 Algorithm and Implementation

In this section we describe our implementation of cc-HTM. There are four major components: (1) and a pair of queues designated as Blue and Red queues, to order persistent writes consistently; (2) a protocol for logging and recovery; (3) an Alias Table used as shadow memory for transactions to prevent corruption due to cache spillage; and (4) a lock-free algorithm with a background thread that retires completed transactions in order of HTM completion.
4.3.1 Blue and Red Queues

A Blue Queue holds transactions that have not entered the COMMIT state (called *blue* transactions). The Red Priority Queue holds those that are in the COMMIT state (called *red* transactions).

Our software implementation makes use of the following:

- The Blue Queue size is bounded by the maximum number of concurrent transactions.
- The Red Priority Queue size is bounded by the maximum number of transactions that have not yet been retired.
- Since the queue sizes are bounded, we use a static memory array and therefore do not have to perform dynamic memory allocations.
- The Blue Queue may have elements removed in any order, as transactions may enter the COMMIT state, moving from the Blue to Red Queue, in any order.
- A transaction thread that inserted itself in the Blue Queue will eventually remove itself from the Blue Queue.
- Only one retiring thread needs to examine the head of the Blue Queue and compare it to the head of the Red Priority Queue.
- The Red Priority Queue can be a linked list Priority Queue.
- The Red Priority Queue can have multiple inserters (transactions moving from Blue to Red), but only needs a single retirement thread to remove entries.

We use an atomic counter to indicate the start and end times and as an index into the Blue Queue. When a transaction completes the concurrency section, it examines the current atomic counter to get a bounds on all current transactions that could be open during the transaction before retirement. This counter is used as the virtual
time token, and a transaction cannot be retired until all transactions opened before this time have entered the COMMIT state (entered the Red Priority Queue).

We combine the Blue and Red Queues:

- The transaction thread inserts into and removes from the Blue Queue and inserts into the Red Priority Queue.
- Inserting into the Red Priority Queue before removing from the Blue Queue allows for the move to not require atomic operations.
- Since both queues are bounded in size, we implement them in a single bounded buffer.
- We can advance the token that points to the head of the Blue Queue until we get to the next element in the buffer that is non-red and non-empty.

The combined Blue-Red Data Structure has several important properties that allow for static allocation and low cost transaction management. These properties are discussed in detail in Section 4.3.5.

4.3.2 Logging and Recovery

The transaction start, published concurrency end time, and log completion flag are stored in the log. The Red Priority Queue and Blue Queue are not saved in PM. As a transaction starts the start time is saved in the log for Blue Queue reconstruction; for the back-end persistence method this log is in-cache only. This allows for easy reconstruction of the queues on failure while not having to have a persistent version of the concurrent queues.

For recovery, we read all logs and rebuild the Blue and Red Queues. We then process the completed transactions at the head of the Red queue that have times less than the head of the Blue queue. This brings the system back into the closest possible point for in-memory consistency.
4.3.3 Alias Table Implementation

The structure of the Alias Table is shown in Figure 4.2. It is implemented as a DRAM-resident, set associative key-value store that holds the values of transaction updates. Access to PM transaction variable X is redirected to a shadow DRAM variable X’ allocated in the Alias Table. The aliased entry holds the address of X, its value, and a timestamp field $T_{Start}$ that is used to reclaim the space after X has been persisted in PM. Most recent values of variables are found in either the Alias Table or the home location (if retired from the log and reclaimed from the Alias Table).

![Diagram of Alias Table and Continuous Checkpointing of HTM Transactions]

Figure 4.2: Continuous Checkpointing of HTM Transactions (cc-HTM) Using DRAM Aliasing and Persistent Management Queues
For example, suppose two transactions T1 and T2 as shown in Figure 4.2. T1 starts at time $t_1$ and writes X and Y into the alias table. T2 starts at time $t_2$ and writes Y and Z into the alias table. The start times of the transactions are recorded as the tags in the table. When the T1 retires at some point in the future, a new minimum start time will be in the transaction queue. Since this new time will be greater than $t_1$, other transactions can safely remove X from the table if space is needed.

Transaction loads and stores are implemented as calls to a user library shown in Algorithm 4. The Load library call first checks the Alias Table for the requested address using the function $getAliasTableEntry$. The function returns the entry in the table that matches the passed address or, if no entries match, returns the entry in the associative set with the smallest timestamp as a possible candidate for reclamation. The Load returns the value in the table if found; else it performs a normal LOAD from the PM address of the variable and returns the value.

The Store library call similarly checks the Alias Table. If the variable is found in the table the entry is updated with the new value; else if there is an empty entry or stale entry that can be reclaimed, then the address, new value, and the start timestamp of the transaction doing the write are entered in the reclaimed entry. If no entry is available, the transaction explicitly aborts. Note we do not try to evict entries from the Alias Table to make space, but instead use a simple aging mechanism to reclaim stale entries, as described in Section 4.2.1 and Lemma 4.4. This reduces contention for the Alias Table that would otherwise cause many unwanted HTM aborts due to accesses for table management. To facilitate conflict-free reclamation each entering transaction sets a private variable $myObservedMin$ to the lowest start timestamp of all transactions in the system. If an Alias Table entry has a timestamp smaller than this value, the last transaction that wrote that entry has retired and the space can be safely reclaimed.
4.3.4 Lock Free Algorithm

**Algorithm 4: cc-HTM Transaction Implementation**

**TransactionBegin ()**

\[
myId = \text{QueueNotifyStart();}
\]
\[
myObservedMin = \text{QueueGetMin();}
\]
\[
\text{Open Log and persist startTS=myId;}
\]
\[
\text{XBegin();}
\]

**TransactionEnd ()**

\[
\ast myPersistentTS = \text{RDTSCP();}
\]
\[
\text{XEnd();}
\]
\[
\text{QueueNotifyEnd(myId, \ast myPersistentTS);}
\]
\[
\text{Persist cached Log and timestamps to PM;}
\]
\[
\text{QueueTxCommit(myId, Log);}
\]

**Load (address)**

Entry *e = getAliasTableEntry(address);
if (e->address = address))
    return e->Value;
else return *address;

**Store (address, value)**

Entry *e = getAliasTableEntry(address);
    // Abort if entry not found and min entry not retired:
if (e->Address ≠ address AND
    (e->id > myObservedId)) XAbort();
    // Save the write and tag with myId start time:
    e->Address = address;
    e->Value = value;
    e->TStart = myId;
    Append (address, value) to cached Log;

**getAliasTableEntry (address)**

line = getLine(address);
for (entry = each element in line) {
    if (entry->address = address)
        return entry;
    if (entry->TStart < minTStart) {
        minTStart = entry->TStart;
        minEntry = entry;
    }
}
return minEntry;
We present the details of the transaction lifecycle described in Section 4.2. Transaction interface routines are shown in Algorithm 4 and queue management routines in Algorithm 5 and retirement thread and queue management in Algorithm 6. In the OPEN state, the transaction invokes the library function TransactionBegin of Algorithm 4. The function QueueNotifyStart from Algorithm 3 returns the (unique non-decreasing) transaction id that also serves as its start timestamp startTS. The current minimum start timestamp of all transactions currently in the system is computed by the function QueueGetMin also from Algorithm 5 and is saved locally in myObservedMin. This value helps facilitate conflict-free reclamation of space on writes. The transaction then enters the COMPUTE state by executing XBegin and begins its HTM section.

Load and store operations within the HTM section are implemented by accesses to the Alias Table as described in Section 4.3.3, with stores additionally appending to the log. Additionally, for stores, if an Alias Table entry has a timestamp smaller than the myObservedMin, the last transaction that wrote that entry has retired and the space can be safely reclaimed.

After the last instruction of the HTM section the library function TransactionEnd is invoked. This reads the platform timer using RDTSCP, ends the HTM transaction with XEnd, and enters the LOG state. The transaction obtains the end timestamp, records the persistence timestamp, and writes its log to PM. The log is persisted by a sequence of CLWB (cacheline writebacks) instructions that write back the values cached during the HTM execution, followed by a persistent memory fence. The function QueueNotifyEnd moves the transaction to its correct location in the Red Priority Queue, moving the transaction to the CLOSE state. The transaction then enters the COMMIT state by calling function QueueTxCommit. The transaction will be released from the COMMIT state and will commit when it is signaled by the persistence management thread. As discussed previously this can be controlled with the choice of lag parameter (higher L allows transactions to commit early).
When a transaction enters the system its transaction id is used to allocate a slot in the Blue Queue. When a transaction enters the Red Priority Queue it marks itself as deleted in the Blue Queue. The head of the Blue Queue points to the earliest transaction that has not been marked as deleted, and the tail is the last entry that has been allocated. If the transaction at the head of the Blue Queue is marked as deleted, the head pointer is advanced sequentially until it points to an entry that has not been marked as deleted or reaches the tail of the queue. The amortized time
per advancement of the head pointer, and marking an element in the Blue Queue as deleted, is a constant.

The Red Priority Queue is organized as a singly-linked list arranged in increasing order of the persistence timestamps, permitting simple concurrent insertion and deletion using Compare-and-Swap instructions [103]. When moving from the Blue to Red queue in QueueNotifyEnd, the persistence timestamp of the transaction is used to find its position in the Red Priority Queue using function FindInsertNode. Using the head pointers of the Blue and Red queues, entering transactions can update myObservedMin in constant time by comparing the start timestamps of the entries at the head of the two queues.

**Retirement Thread:**

Transaction retirement is managed by a Persistence Management Retirement Thread shown in Figure 4.2 and implementation in Algorithm 6. The thread continuously checks for available logs to retire, as shown by the RetirementThread and CheckRetirement functions in Algorithm 6. When it is safe to retire a transaction, the thread reads log records of the entry at the head of the Red Priority Queue and writes them to their home locations in PM. The thread monitors the Red Priority Queue, and retires the transaction at the head when its startTS is smaller than the startTS of the entry at the head of the Blue Queue. This ensures that it is the transaction with the smallest persistence timestamp in the system (Lemma 3.2 in Chapter 3.4.3). After retirement the element at the head of the Red Priority Queue must be deleted from the list. A lock-free implementation in which the element is marked with a flag for removal from the linked list before removal is used to prevent concurrent insertions from swapping into the next pointer simultaneously similar to [103].
Algorithm 6: Lock-Free Persistence Management Queue Retirement Thread

CheckRetirement()

\[ ri = redHead; \]
\[ \text{if } ((Q[ri].endTS > 0) \text{ and } (Q[ri].logPointer \neq 0) \text{ and } (Q[ri].endTS \leq \text{UpdatedBlueHead}())) \{ \]
\[ \text{next} = Q[ri].nextRedIndex; \]
\[ \text{if } (!\text{CAS}(&(Q[ri].nextRedIndex), \text{next}, \text{next} + \text{FLAG})) \text{ return; } \]
\[ \text{if } (\text{CAS}(&\text{redHead}, ri, \text{next}) \{ \]
\[ \text{Retire}(Q[ri].logPointer); \]
\[ \text{if } (Q[ri].startTS == TxMin) \]
\[ \text{CAS}(&\text{TxMin}, Q[ri].startTS, \text{UpdatedMin}()); \]
\[ Q[ri].startTS = Q[ri].endTS = 0; \]
\[ Q[ri].logPointer = Q[ri].priorityTS = 0; \]
\[ Q[ri].nextRedIndex = \text{EMPTY}; \text{sfence(); } \]
\[ \text{AtomicAdd}(&\text{numElements}, -1); \]
\[ \} \text{ else } Q[ri].nextRedIndex = \text{next}; \]
\[ \}

UpdatedMin()

// Walk from TxMin to next non-empty Q[i].startTS

UpdatedBlueHead()

static blueMin = 0;
\[ \text{if } (\text{blueMin} == \text{TxCounter}) \text{ return } \text{blueMin; } \]
\[ \text{while } (Q[\text{blueMin} \& \text{MASK}] != 0) \]
\[ \text{blueMin}++; \]
\[ \text{return } \text{blueMin}; \]

RetirementThread()

while (!\text{done OR } (\text{numElements} > 0))
\[ \text{CheckRetirement(); } \]

Retire(log)

\[ \text{walk log and write } *\text{address}=\text{value; sfence(); } \]

Example:

An example is shown in Figure 4.3. Both Blue and Red queues share the same statically allocated circular buffer of transaction entries. The logical Blue Queue
contains elements 11 and 14, while the logical Red Priority Queue has 12, 9 and 13. The minimum of both queues has a TxMin of 9. When an element is added to the Blue Queue, it is simply placed in the tail, the TxCounter. When one of the elements is ready to move from the Blue to the Red queue, the endTS time is saved along with the persistTime. The element is added to the Red Priority Queue by Atomic Compare-and-Swap after finding the index in the Red Priority Queue. When the TxMin index element is retired, e.g. removed from the Red Priority Queue, in this case element 9 at index 1, the new TxMin is found by walking until the next non-empty element, 11 in this example.
4.3.5 Data Structure Properties

Lemma 4.5 (Bounded Transaction Management Size). The size of the Blue-Red data structure is bounded by $2(M+N)$ where $M$ is the limit on the number of outstanding transactions that have not been retired in the Red Priority Queue and $N$ is the maximum number of open transaction threads.

Proof. Assume there is a straggling transaction $T$ that started earlier than all other transactions that have not been retired, and $T$ has not yet saved its completion time. If we limit new transactions from starting when the size of the Red Priority Queue is greater than size $M$, there could be $N-1$ threads waiting to start a transaction. In the worst case, when $T$ saves its completion time, it may be placed as the last element in the Red Priority Queue while also having the smallest start time at the start of the data structure, now of size $M+1$. Therefore, all transactions in the Red Priority Queue can immediately be retired, and the $N-1$ waiting threads can each start new transactions. In the worst case, these $N-1$ transactions complete quickly and persist their logs, entering the Red Priority Queue before the retirement thread can retire a single transaction. Now the data structure size is $(M+1) + (N-1)$ or $(M+N)$. In the worst case, $T$ takes a long time to push its log, and all other $M+N-1$ transactions are retired. This leaves $(M+N-1)$ spaces between $T$ and the next set of transactions. Before $T$ is retired, the $N$ threads fill up another $M+N$ elements in the data structure. At this point, $T$ has to be the head of the Red Priority Queue, since it completed before any transactions $M+N$ that started after it, since they were waiting on $T$ to complete. So, there could be $(M+N-1) + (M+N)$ spaces in the data structure plus $T$ itself.

Lemma 4.6 (Constant Transaction Start Time Cost). The time to update the minimum start time in the combined data structure is amortized to $O(1)$.

Proof. The minimum start time is in either the Blue or the Red queue. If the minimum is in the Blue Queue, then when an element moves from the Blue to Red queue,
the minimum need only be updated if it is the element moving queues. When the
minimum start time element is in the Blue Queue and moves to the Red Priority
Queue, the new minimum can be found by simply walking down to the next valid
element. The element stays as the minimum until the transaction is retired and only
needs to search the next set of non-retired elements in front of it, bounded by M+N,
only if it had waited M+N as the minimum of both data structures. The same hold
for the minimum of the Red Priority Queue. The minimum start time of transactions
in the Red Priority Queue need not be the front of the queue, as transactions may
complete in a different order from which they started. However, in linear order of
start times is the next element is minimum start time in the Red Priority Queue.
When this element is retired, the next new minimum can be found using the same
method, walking down the data structure.

\[\text{Lemma 4.7 (Amortized O(1))}. \] The head of the Blue Queue can be found in amor-
tized O(1) time.

\[\begin{proof}
Using the same update technique for finding the global minimum, we simply
walk to the next Blue Queue element found in the Blue-Red data structure before we
get to the tail. A Compare-and-Swap instruction is not needed. This walk only has
to be completed when the head of the Blue Queue, the earliest transaction start time,
completes. If they complete in order, then each transaction moves the head pointer
to the next element. If transactions complete out of order, then only the earliest
updates the head pointer, but might have to walk up to the number of transactions
that completed before it.
\end{proof}\]

4.4 Evaluation

For evaluation, we employed Intel(R) Xeon(R) E5-2699 v3 series processors, 18 cores
per processor, running at 2.30 GHz, with Red Hat Enterprise Linux 7. We used num-
actl to restrict all threads to a single processor (socket) for repeatable results and
reducing interference from other background activities during measurements. HTM transactions were implemented with Intel Transactional Synchronization Extensions (TSX) [38] using a global fallback lock. Processor micro-code was patched to enable TSX support. A set of runtime environment variables allowed for the easy configuration of Alias Table sizes, persistence methods such as cc-HTM or no persistence, and maximum allowable lag. We built our software using g++ 4.8.2. Each measurement reflects an average over twenty repeats with small variation among the repeats.

Using micro-benchmarks and SSCA2 [104] and vacation, from the STAMP [39] benchmark suite, we compared the following methods:

- **HTM Only**: Hardware Transactional Memory with Intel TSX, without any logging or persistence. This method provides a baseline for transaction performance in cache memory without any persistence guarantees. If a power failure occurs after a transaction, writes to memory locations may be left in the cache, or written back to memory in an out-of-order subset of the transactional updates.

- **cc-HTM**: Continuous Checkpointing of HTM for Persistence in PM. This is our full implementation using the Persistence Management Thread, Queue, Logging, and Alias Table.

- **PTL2**: (Persistent Transactional Locking). In this method, we added PM-persistence to TL2 [40, 79] by generating a persistent UNDO log at the time that a TL2 transaction performs its sequence of writes. The undo-log entries are written with write-through stores and SFENCEs, and once the TL2 transaction commits and the new data values are flushed into PM, the undo-log entries are removed.
4.4.1 Benchmarks

The Scalable Synthetic Compact Applications for benchmarking High Productivity Computing Systems [104], SSCA2, is part of the Stanford Transactional Applications for Multi-Processing [39], or STAMP, benchmark suite. It uses a large memory area and has multiple kernels that construct a graph and perform operations on the graph. We executed the SSCA2 benchmark with scale 20, which generates a graph with over 45 million edges. We increased the number of threads from 1 to 16 in powers of two and recorded the execution time for the kernel for each method.

For cc-HTM, we use an Alias Table of Size 64 MB with 2-way set associativity. We also set the maximum allowable Lag $L$ to 512. Later in this section we explore variations in these parameters.

Figure 4.4 shows the execution time for each method for the Compute Kernel in the SSCA2 benchmark as a function of the number of threads. Each method reduces the execution time with increasing numbers of threads. Our cc-HTM approach has similar execution time to HTM in the cache hierarchy with no persistence and is over 1.5 times faster than a persistence method to PM with PTL2. Figure 4.5 shows the speedup for each method as a function of the number of threads. Even though the HTM (cache-only) method does better in absolute terms as we saw in Figure 4.4, it proceeds from a higher baseline for single-threaded execution. The cc-HTM method matches its speedup. And even-though PTL2 has the advantage of a low single thread number for comparison, it yields a significantly weaker scalability due to the inherent costs of having to perform persistent flushes within its concurrent region.

Figure 4.6 shows the execution time of the SSCA2 Compute Graph Kernel with varying Alias Table sizes and associativity. With increasing Alias Table sizes, the benchmark can take longer to execute, due to having more cache misses for Alias Table entries and having to retrieve entries from main memory. A transaction will abort if it does not find a matching entry in the Alias Table and cannot find a free slot it can reclaim— that is, as we described earlier, the case when there is an alias table
Figure 4.4: SSCA2 Benchmark Compute Graph Kernel Execution Time as a Function of the Number of Parallel Execution Threads

Figure 4.5: SSCA2 Benchmark Compute Graph Kernel Speedup as a Function of the Number of Parallel Execution Threads
collision with a set containing values that were modified in a transaction that is still pending retirement. Therefore a smaller Alias Table or one with lower associativity, can have more conflicts and can cause more aborts. However, too large an Alias Table will suffer from cache performance, having to reach back to main memory, lengthening the time of transactions and also potentially causing more aborts. We find Alias Tables in the size of 64-256 MB with 2-4 way associativity perform well in general.

Next, we investigated the effect of the the maximum $\mathcal{L}$ (we chose this as 512) on parallel execution time. First, we counted the number of non-retired transactions colored Blue and Red over time in a subset of the Compute Graph Kernel in the SSCA2 Benchmark. We used the default graph scale of 20 and performed evaluations with 8 concurrent threads. Initially, we set the maximum allowable Lag $\mathcal{L}$ to 512 and recorded the results in Figure 4.7. The total number of non-retired transactions never grows past 150 over the entire execution time of the benchmark. The number
of Red transactions depends on long running transactions that have not yet published their completion times. A delayed or long running transaction thread might hold up the persistence of several transactions that have completed in each of several threads. Once the long running transaction is complete, changing color from Blue to Red, then other transactions can be persisted. A fast thread might have created numerous outstanding transactions waiting on persistence but is allowed to continue in the foreground path. Figure 4.8 shows how the maximum allowable Lag can affect the overall performance. We use 8 threads and vary the maximum allowable Lag $L$ in the SSCA2 benchmark and show that after $L$ is increased to 125, throughput is insensitive to additional increases in Lag and cc-HTM achieves peak transactional throughput.

Figure 4.7: Transactions Pending Retirement Colored Blue and Red Over Time in the SSCA2 Benchmark with 8 Concurrent Threads

* To put these lag parameters of 125, 256, etc. in perspective: at transaction rates reaching half a million to a million per second, these lag values represent completions that are in the 99.8-99.9th percentile, or, retiring within about a millisecond of completion. That is, the recoverable cc-HTM checkpoint is just 1-2 ms behind the wavefront of completing transactions.
We also evaluated the vacation benchmark which is part of the STAMP benchmark suite. The vacation benchmark emulates database transactions for a travel reservation system. We executed the benchmark with the low option for lower contention emulation. Figure 4.9 shows the execution time for each method for the vacation benchmark as a function of the number of threads. Each method reduces the execution time with increasing numbers of threads. Our cc-HTM approach follows the trends similar to HTM in the cache hierarchy with no persistence, with both approaches flattening execution time after 4 threads. As thread contention increases, all experience more aborts, but PTL2 can spin and retry with individual locks to gain extra parallelism under high contention, and thus it continues to improve, albeit slowly, while cc-HTM and HTM-Only stop reducing their runtimes.

Additionally, we examined the effect of increased PM write times on the benchmark. Byte-addressable, persistent non-volatile memory is characterized by longer write times. To emulate the longer write times for PM, we insert a delay after non-
temporal stores when writing to new cache lines and a delay after cache line flushes. The write delay can be tuned to emulate the effect of longer write times typical of PM. Figure 4.10 shows the vacation benchmark execution time for various PM write times. Our cc-HTM method is less affected by increasing PM write times than the PTL2 approach due to several factors. First, cc-HTM performs write-combining for log entries on the foreground path for each thread, so writes to several transaction variables may be combined into a fewer writes. Additionally, PTL2 transactionally persists an undo log on writes causing a foreground delay for each write.

![Vacation Benchmark Execution Time as a Function of the Number of Parallel Execution Threads](image)

Figure 4.9: Vacation Benchmark Execution Time as a Function of the Number of Parallel Execution Threads

### 4.4.2 Hash Table

Our next series of experiments show how read / write ratios and transaction sizes affect overall performance. We create a 64 MB Hash Table Array of elements in main memory and transactionally perform a number of element updates. For each
Figure 4.10: Vacation Benchmark Execution Time with Various PM Write Times for Four Threads

Figure 4.11: Millions of Transactions per Second for Hash Table Updates of 10 Elements versus Concurrent Number of Threads
Figure 4.12: Hash Update of 10 Element Transaction Throughput for 6 Threads with Varying Maximum Allowable Lag

Figure 4.13: Average Transactions per Second for Increasing Group Sizes of Atomic Hash Table Updates with 6 Concurrent Threads
transaction, we generate a set of random numbers of a configurable size, compute their hash, and write the value into the Hash Table Array.

First, we create transactions consisting of 10 atomic updates and vary the number of concurrent threads and measure the maximum throughput. We perform 1 million updates and record the average throughput with maximum allowable Lag set to 256 and plot the results in Figure 4.11. This experiment shows that throughput increases with the number of threads until transactions start conflicting and affect the overall throughput. For up to 5 concurrent threads cc-HTM achieves roughly 2x throughput over PTL2. Next, we fix the number of concurrent threads to 6 and vary the maximum allowable Lag parameter $L$ over a broad range and record the average throughput. The results in Figure 4.12, like SSCA2, show cc-HTM achieves peak throughput with Lag $L$ at 125-150 even for write intensive workloads.

The transaction write set was then varied from 2 to 30 elements with 6 concurrent threads. The average throughput was recorded and is shown in Figure 4.13.
just 2 elements in the write set, cc-HTM performs slightly slower than PTL2, due to overhead in the Persistence Queue Management, but for all other transaction sizes, cc-HTM has 1.5x to 2x throughput over PTL2.

Finally, the read-write ratio is varied from 10% to 100% for transactions of 10 element updates with 6 concurrent threads. As shown in Figure 4.14, we perform 1 million transactional updates of 10 elements each and record the average transaction throughput. With only 1 to 2 written elements in a heavy read transaction comprised of 80-90% reads, cc-HTM performs slower. However, with only 30% writes, cc-HTM performs better; and with mostly-write workloads cc-HTM has up to 50% higher throughput over PTL2. Additionally, cc-HTM is less sensitive to changes in the read-write ratio due to shorter concurrency sections. Writes are less expensive during the concurrency section of cc-HTM since they do not have to persist to PM, but reside in cache along with the cached log. In PTL2 however, a write to an PM variable is more expensive; therefore increasing the number of writes in a transaction will affect PTL2 more when compared to cc-HTM.

4.4.3 Red-Black Tree

In the final set of experiments, we use the transactional Red-Black tree from STAMP [39] initialized with 1 million elements. We then perform insert operations on the Red-Black tree and record average transaction times and throughput over 200k additional inserts.

Figure 4.15 shows the average response time versus transaction request arrival rate for 4 concurrent processing threads. Each transaction inserts an additional element into the Red-Black tree after it is initialized with 1 million elements. As shown in the Figure, cc-HTM has a much faster response time (by a factor of 4 to 5) over PTL2 and close to non-persistent cache-based HTM Only.

Next, we record the maximum throughput of inserts into the Red-Black tree per second for a varying number of threads. The average throughput over 1 million
Figure 4.15: Average Response Time in μs for varying Transaction Arrival Rate for Red-Black Tree Element Inserts with 4 Threads

Figure 4.16: Millions of Transactions per Second for Atomic Red-Black Tree Element Inserts versus Number of Concurrent Threads
Figure 4.17: Red-Black Tree Element Insert Transaction Throughput for 4 Threads with Varying Maximum Allowable Lag

Figure 4.18: Red-Black Tree Element Insert Transaction Throughput for 4 Threads with Varying PM Write Times
insertions is shown in Figure 4.16. As can be seen in the Figure, cc-HTM has a much higher throughput over PTL2, and thread contention on the data structure causes both methods to fall off. cc-HTM can have almost 4-5x improvement over eager persistence locking. Inserting an element into a Red-Black tree first requires finding the insertion point which can take many read operations. In our experiments, we averaged 63 reads and 11 writes per transactional insert of one element into the Red-Black tree. Next, we vary the Lag parameter $L$ and record the maximum throughput in Figure 4.17. In this more read-heavy workload, cc-HTM achieves peak throughput with $L$ at only 80.

Finally, we vary the PM write time similar to our prior experiment. Figure 4.18 shows the Red-Black tree throughput as a function of the PM write time. As the write time increases both our cc-HTM method and the PTL2 method have decreasing throughput.

4.5 Summary

Systems with large core counts coupled with large, byte-addressable non-volatile memory offer the potential of breakthrough performance on problems with large data footprints, by enabling massively parallel in-memory applications. Hardware Transactional Memory creates the possibility that parallel applications can run with minimal entanglement among threads, impeded only by actual data races, instead of by preventive software synchronization. However, since HTM implementations typically make all changes to memory visible instantly, they create a window of vulnerability for transactions whose updates flow out of order across the memory buses and potentially leave PM based data and structures in a broken state in the event of an untimely machine crash.

In this Chapter we presented cc-HTM, a continuous checkpointing of Hardware Transactional Memory to Persistent Memory, which requires no hardware changes and runs on existing Intel hardware. The Chapter presents light-weight mechanism
to propagate the updates from a completed HTM transaction to PM asynchronously, while preserving atomicity to PM. By introducing a small lag parameter, we create a continuous wave of transactions that are persisted to PM fronted by transactional HTM concurrency sections. We utilize a global alias table to redirect all stores to dynamic memory to prevent unwanted cache evictions from corrupting PM.

We investigated other implementation options for our approach. One variant was to write values back from the table to PM to reclaim space, but found that HTM store sets were increased and caused additional aborts, slowing persistence. We also investigated using multiple threads for retirement which can increase back-end persistence but complicates implementation.

Using micro-benchmarks and benchmarks from the STAMP suite, we showed that the lock-free implementation of our data structures enables fast, parallel atomic persistence to PM. With small values of the lag parameter, cc-HTM achieves its peak throughput, at which it significantly outperforms persistent software transactions in both throughput and response time, and it compares well with volatile HTM transactions.
Chapter 5

Hardware Transactional Persistent Memory Controller

This chapter presents a memory controller approach for the Hardware Transactional Persistent Memory model presented in Chapter 3. The approach utilizes a novel software protocol combined with a Persistent Memory Controller, without requiring changes to processor cache hardware or HTM protocols. In contrast, previous approaches require significant changes to existing processor microarchitectures.

First, the background on using a memory controller for persisting transactions to PM is presented. Next, the Hardware Transactional Persistent Memory Controller as it relates to the model from Chapter 3 is discussed in Section 5.2. The implementation is then presented in Section 5.3 with a full discussion on the software protocol and memory controller algorithms. An extensive evaluation is performed on Intel® hardware and the results presented in Section 5.4. We show the controller yields significant gains in throughput and latency in comparison with persistent transactional locking.

5.1 Background on WrAP Controller

One way to catch unwanted cache evictions from corrupting Persistent Memory is to re-design the memory controller. Several such memory controllers are discussed in Chapter 2.5. We base our HTPM implementation using the WrAP controller from [90, 33, 37] as it does not require any modifications to the cache, protocols, HTM semantics, or the processor core. The WrAP controller is placed between the cache and memory, intercepting unwanted cache evictions before they reach PM.
The original WrAP controller, shown in Figure 5.1, uses a volatile victim cache to prevent cache evictions from reaching persistent memory during an atomic group of stores to PM. WrAP combines the controller with a lightweight software library that utilizes an efficient Redo Logging technique that write-combines log entries to reduce memory bandwidth. The controller supports two-phase locking for concurrent transaction execution.

On opening a wrap transaction, a log is created in PM and the controller adds the transaction to the set of open transactions. Loads and stores proceed normally, and stores are simultaneously propagated to a PM Redo Log area using write-combining stores. All evictions from the cache hierarchy to PM are caught by the controller and held in a volatile victim cache. The eviction is tagged with the set of open transactions, where it remains until erased by the controller. On closing a wrap, the log is closed, notifying the controller that the log can be retired. Log retirement involves reading entries in the log and copying their values to the home PM locations.
Once the log is retired, the transaction identifier is removed from the set of open transactions on the controller and removed from the dependency set in all entries in the victim cache. When an entry in the victim cache has an empty dependency set, it is removed from the victim cache.

While the controller gained notability, it faced several drawbacks. The original design of the controller does not operate with HTM. Required logging operations during transactional execution for PM atomicity aborts HTM used for concurrency control. Additionally, the original controller design requires complicated victim cache implementations. Entries in the victim cache may be added or removed in any order, and updates to existing entries get re-tagged with the current set of open transactions. Finally, after transaction close, the log read and entries copied into main PM. This adversely affects performance due to the synchronous required log processing with reading and writing.

The controller presented in this chapter not only simplifies the design, but also enhances performance and works with HTM. The victim cache is redesigned to be a First-In-First-Out buffer, and log writes are held to the end of the transaction, where they are only needed for recovery.

5.2 Persistent Memory Controller Approach

Our approach achieves durability of HTM transactions to Persistent Memory by a cooperative protocol involving three components: a back-end Persistent Memory Controller, a software library for transaction execution and logging, and a failure recovery procedure. The Persistent Memory Controller intercepts dirty cache lines evicted from the last-level processor cache (LLC) on their way to persistent memory. An intercepted cache line is held in a FIFO queue within the controller until it is safe to write it out to PM. All memory variables are subject to the normal cache operations and are fetched and evicted according to normal cache protocols. The only change we introduce is interposing the external controller between the LLC and memory. Note
that the controller does not require changing any of the internal processor behavior. The controller simply delays the evicted cache lines on their way to memory till it can guarantee safety. It is pre-programmed with the address range of a region of persistent memory that is reserved for holding transaction logs. Addresses in the log region pass through the controller without intervention.

HTM+PM transactions execute independently. Within an outer-envelope that achieves consistency of updates between the volatile cache hierarchy and the durable state in PM, these transactions use an unmodified HTM to serialize the computational portions of conflicting transactions. A transaction (1) notifies the controller when it opens and closes, (2) saves start and end timestamps in PM to enable consistent recovery after a failure, (3) performs its HTM operation, and (4) persists a log of its updates in PM before closing. If a transaction requires strict durability it informs the controller during its closing step, and then waits for the go-ahead from the controller before committing. If it only needs relaxed durability it can commit immediately after its close. The recovery routine is invoked after a system crash to restore the PM variables to a valid state i.e. a state that is consistent with the actual execution order of every sequence of dependent transactions. The recovery procedure uses the saved logs to recover the values of the updated variables, and the saved start and end timestamps to determine which logs are eligible for replay and their replay order.

5.2.1 Transaction Lifecycle

Our approach persists an HTM transaction onto durable PM consistently by splitting it into three phases and five states as described in 3.4. The PM Controller approach adapts the five states: OPEN, COMPUTE, LOG, CLOSE and COMMIT as shown in Listing 5.1. When a transaction begins, it calls the library function OpenWrapC (see Algorithm 7 in Section 5.3.1). This function invokes the Persistent Memory Controller with a small unique integer (wrapId) that identifies the transaction. The controller adds wrapId to a set of currently open transactions (referred to as COT)
that it maintains (see Algorithm 8 in Section 5.3.2). The transaction then allocates and initializes space in PM for its log and updates the \textit{startTime} record of the log. The \textit{startTime} is obtained by reading a system wide platform timer using the \textit{RDTSCP} instruction as described in Chapter 3.4.1. In addition to \textit{startTime}, a log includes a second timestamp \textit{persistTime} that will be set just prior to completing the HTM transaction. The \textit{writeSet} is a sequence of (\textit{address}, \textit{value}) pairs in the log that will be filled with the locations and values of the updates made by the HTM transaction. The log with its recorded \textit{startTime} is then persisted using cache line write-back instructions (\textit{clwb}) and \textit{sfence}.

The transaction then enters the \textit{COMPUTE} state by executing \textbf{XBegin} and entering the HTM code section. Within the HTM section, the transaction updates \textit{writeSet} with the persistent variables that it writes. Note that the records in \textit{writeSet} will be held in cache during the \textit{COMPUTE} state since it occurs within an HTM and cannot be propagated to PM until \textbf{XEnd} completes. Immediately before \textbf{XEnd} the transaction obtains a second timestamp \textit{persistTime} that will be used to order the transactions correctly. This timestamp is also obtained using the same \textit{RDTSCP} instruction.

After executing \textbf{XEnd}, a transaction next enters the \textit{LOG} state. It flushes its log records from cache hierarchy into PM using cache line write-back instructions (\textit{CLWB} or \textit{CLFLUSHOPT}), following the last of them with an \textit{SFENCE}. This ensures that all the log records have been persisted. In addition to \textit{startTime}, a log includes the \textit{persistTime} time stamp that was set just prior to completing the transaction. The \textit{writeSet} records in the log hold (\textit{address}; \textit{value}) pairs representing the locations and the values updated by the transaction. After the \textit{SFENCE} following the log record flushes, the transaction enters the \textit{CLOSE} state.

In the \textit{CLOSE} state the transaction signals the Persistent Memory Controller that its log has been persisted in PM. The controller removes the transaction from its set of currently open transactions \textit{COT}. It also reflects the closing in the state of evicted
Listing 5.1: Transaction Structure

<table>
<thead>
<tr>
<th>Transaction Begin</th>
<th>State: OPEN</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 NotifyPMController(open);</td>
<td></td>
</tr>
<tr>
<td>2 Allocate a Log structure in PM;</td>
<td></td>
</tr>
<tr>
<td>3 nowTime = ReadPlatformCounter();</td>
<td></td>
</tr>
<tr>
<td>4 Log.startTime = nowTime;</td>
<td></td>
</tr>
<tr>
<td>5 Log.writeSet = {};</td>
<td></td>
</tr>
<tr>
<td>6 Persist Log in PM;</td>
<td></td>
</tr>
</tbody>
</table>

---

<table>
<thead>
<tr>
<th>State: COMPUTE</th>
</tr>
</thead>
<tbody>
<tr>
<td>XBegin</td>
</tr>
</tbody>
</table>

// Transaction Body of HTM
// All reads and writes to transaction
// variables are performed and also
// appended to Log.writeSet.

| 7 endTime = ReadPlatformCounter(); |
| XEnd |

---

<table>
<thead>
<tr>
<th>State: LOG</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 Log.persistTime = endTime;</td>
</tr>
<tr>
<td>9 Persist Log in PM;</td>
</tr>
</tbody>
</table>

---

<table>
<thead>
<tr>
<th>State: CLOSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>10 NotifyPMController(close);</td>
</tr>
</tbody>
</table>

---

<table>
<thead>
<tr>
<th>State: COMMIT</th>
</tr>
</thead>
<tbody>
<tr>
<td>11 If (strict durability requested)</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>12 Transaction End</td>
</tr>
</tbody>
</table>

A transaction requiring strict durability informs the controller at this time; the controller will signal the transaction in due course when it is safe to commit i.e. its updates are guaranteed to be durable.

The transaction is then complete and enters the COMMIT state. If it requires strict durability it waits till it is signaled by the controller. Otherwise, it immediately
5.2.2 Persistent Memory Controller

The Persistent Memory Controller is shown in Figure 5.2. While superficially similar to an earlier design [90, 33, 37], this controller includes enhancements to handle the subtleties of using HTM rather than locking for concurrency control, and makes significant simplifications by shifting some of the responsibility for the maintenance of PM state to the recovery protocol.

A crucial function of the controller is to prevent any transaction’s updates from reaching PM until it is safe for it to do so - without requiring detailed book-keeping about which transactions, currently active or previously completed, generated a specific update. It does this by enforcing two requirements before allowing an evicted dirty cache line (representing a transaction’s update) from proceeding to PM: (i) ensuring that the log of the transaction has been made persistent, and (ii) guaranteeing
that the saved log will be replayed during recovery. The first condition is needed (but not sufficient) for atomic persistence by guarding against a failure that occurs after only a subset of a transaction’s updates have been persisted in PM. The second requirement arises because transaction logs are persisted outside the HTM, and there is no relation between the order in which the transactions execute their HTM and the order in which their logs are persisted. To maintain correct persistence ordering, the recovery routine may not be able to replay a log. We illustrate the issue in the example below.

**Example:** Consider two dependent transactions A and B, A: \{w = 3; x=1;\} and B: \{y = w+1; z = 1;\}. Assume that HTM transaction A executes before B, but that B persists its logs and closes before A. Suppose there is a crash just after B closes. The recovery routine will not replay either A or B, since the log of the earlier transaction A is not available. This behavior is correct.

Now consider the situation where \(y\) (value 4) is evicted from the cache and then written to PM after B persists its log. Once again, following a crash at this time, neither log will be replayed. However, atomic persistence is now violated since B’s updates are partially reflected in PM. Note that this violation occurred even though the write back of \(y\) to PM happened after B’s log was persisted. The Persistent Memory Controller protocol prevents a write back to PM unless it can also guarantee that the log of the transaction creating the update will be played back on recovery (see Lemmas 5.2 and 5.4 in Section 5.2.4).

The second function of the controller is to track when it is safe for a transaction requiring strict durability to commit. It is not sufficient to commit when a transaction’s logs are persistent on PM since, as seen in the example above, the recovery routine may not replay the log if that would violate persistency ordering. The controller protocol effectively delays a strict durability transaction \(\tau\) from committing until the earliest open transaction has a \(startTime\) greater than the \(persistTime\) of \(\tau\). This is because the recovery protocol will replay a log (see Section 5.2.3) if and only
if all transactions with startTime less than its persistTime have closed.

Implementation Overview:

The controller tracks transactions by maintaining a COT (currently open transactions) set $S$. When a transaction opens, its identifier is added to COT and when the transaction closes it is removed. The write into PM of a cache line $C$ evicted into the Persistent Memory Controller is deferred by placing it at the tail of a FIFO queue maintained by the controller. The cache line is also assigned a tag called its dependency set, initialized with $S$ the value of COT, at the instant that $C$ entered the Persistent Memory Controller.

The controller holds the evicted instance of $C$ in a FIFO until all transactions that are in its dependency set (i.e. $S$) have closed. When a transaction closes it is removed from both the COT and from the dependency sets of all the FIFO entries. When the dependency set of a cache line in the FIFO becomes empty, it is eligible to be flushed to PM. One can see that the dependency sets will become empty in the order in which the cache lines were evicted, since a transaction still in the dependency set of $C$ when a new eviction enters the FIFO will also be in the dependency set of the new entry. The simple protocol guarantees that all transactions that opened before cache line $C$ was evicted into the controller (which must also include the transaction that last wrote $C$) must have closed and persisted their logs when $C$ becomes eligible to be written to PM. This also implies that all transactions with startTime less than the persistTime of the transaction that last wrote $C$ would have closed, satisfying the condition for log replay. Hence the cache line can be safely written to PM without violating atomic persistence.

Note that the evicted cache lines intercepted by the controller do not hold any identifying transaction information and can occur at arbitrary times after the transaction leaves the COMPUTE state. The cache line could hold the update of a currently open transaction or could be from a transaction that has completed or even committed and left the system. To guarantee safety, the controller must perforce assume
that the first situation holds. The details of the controller implementation will be presented in Section 5.3.2.

5.2.3 Recovery

Each completed transaction saves its log in PM. The log holds records $startTime$ and $persistTime$ obtained by reading the platform timer using RDTSCP instruction. We refer to these as the start and end timestamps of the transaction. The start timestamp is persisted before a transaction enters its HTM. This allows the recovery routine to identify a transaction that started but had not completed at the time of a failure. Note even though such a transaction has not completed, it could still have finished its HTM and fed values to a later dependent transaction which has completed and persisted its log. The end timestamp and the write set of the transaction are persisted after the transaction completes its HTM section, followed by an end of log marker. There can be an arbitrary delay between the end of the HTM and the time that its log is flushed from caches into PM and persisted.

The recovery procedure is invoked on reboot following a machine failure. The routine will restore PM values to a consistent state that satisfies persistence ordering by copying values from the $writeSets$ of the logs of qualifying transactions to the specified addresses. A transaction $\tau$ qualifies for log replay if and only if all earlier transactions on which it depends (both directly and transitively) are also replayed.

**Implementation Overview:**

The recovery procedure first identifies the set of incomplete transactions $I$, which have started (as indicated by the presence of a $startTime$ record in their log) but have not completed (indicated by the lack of a valid end-of-record marker). The remaining complete transactions (set $C$) are potential candidates for replay. Denote the smallest start timestamp of transactions in $I$ by $T_{\text{min}}$. A transaction in $C$ is valid (qualifies for replay) if its end timestamp ($persistTime$) is no more than $T_{\text{min}}$. All valid transactions are replayed in increasing order of their end timestamps $persistTime$. 
5.2.4 Protocol Properties

Recall from Definition 3.4 that the precedence set of a transaction \( T \) is the set of all dependent transactions that completed their HTM before \( T \). We now summarize the invariants maintained by our protocol.

Lemma 5.1 (Controller Ordering). Consider a transaction \( X \) with a precedence set \( \text{prec}(X) \). For all transactions \( Y \) in \( \text{prec}(X) \), \( \text{startTime}(Y) < \text{persistTime}(X) \).

Proof. Let \( Y \) be a transaction in \( \text{prec}(X) \). First let us consider direct precedence, in which a cacheline \( C \) modified in \( Y \) controls the ordering of \( X \) with respect to \( Y \). That is, \( X \) either reads or writes the cacheline \( C \). Since \( Y \) is in \( \text{prec}(X) \), the earliest time that \( X \) accesses \( C \) must be no earlier than the latest time that \( Y \) accesses \( C \), and thus \( \text{persistTime}(Y) < \text{persistTime}(X) \). Next consider a chain of direct precedences, \( Y \rightarrow Z \rightarrow W \rightarrow \cdots X \), which puts \( Y \) in \( \text{prec}(X) \); and by transitivity, \( \text{persistTime}(Y) < \text{persistTime}(X) \). Since \( \text{startTime}(Y) < \text{persistTime}(Y) \) the lemma follows. \( \square \)

Lemma 5.2 (Controller Cache Line Updates). Consider transactions \( X \) and \( Y \) with \( \text{startTime}(Y) < \text{persistTime}(X) \). If a cache line \( C \) that is updated by \( X \) is written to PM by the controller at time \( t \), then \( Y \) must have closed and persisted its log before \( t \).

Proof. Suppose \( C \) was evicted to the controller at time \( t' \leq t \). Now \( t' \) must be later than the time \( X \) completed HTM execution and set \( \text{persistTime}(X) \); by assumption this is after \( Y \) set its \( \text{startTime} \) at which time \( Y \) must have been registered as an open transaction by the controller. Now, either \( Y \) has closed before \( t' \) or is still open at that time. In the latter case, \( Y \) will be added to the dependence set of \( C \) at \( t' \). Since \( C \) can only be written to PM after its dependence set is empty, it follows that \( Y \) must have closed and removed itself from the dependence set of \( C \). \( \square \)

Lemma 5.3 (Controller Recovery Routine). Any transaction \( X \) that writes an update to PM and closes at time \( t \) will be replayed by the recovery routine if there is a crash any time after \( t \).
Proof. The recovery routine will replay a transaction \( X \) if the only incomplete transactions (started but not closed) at the time of the crash started after \( X \) completed; that is, there is no incomplete transaction \( Y \) that has a \( \text{startTime}(Y) \leq \text{persistTime}(X) \). By Lemma 5.2 such an incomplete transaction cannot exist.

Lemma 5.4 (Controller PM Updates). Consider a transaction \( X \) with a precedence set \( \text{prec}(X) \). Then by the time \( X \) closes and persists its logs, one of the following must hold: (i) Some update of \( X \) has been written back to PM and all transactions \( Y \) in \( \text{prec}(X) \) have persisted their logs; (ii) No update of \( X \) has been written to PM and all transactions \( Y \) in \( \text{prec}(X) \) have persisted their logs; (iii) No update of \( X \) has been written to PM and some transactions \( Y \) in \( \text{prec}(X) \) have not yet persisted their logs.

Proof. From Lemmas 5.1 and 5.2 it is not possible to have a transaction in \( \text{prec}(X) \) that is still open if an update from \( X \) has been written to PM.

5.3 Algorithm and Implementation

The implementation consists of a user software library backed by a simple Persistent Memory Controller. The library is used primarily to coordinate closures of concurrent transactions with the flow of any data evicted from processor caches into PM home locations during those transactions. The Persistent Memory Controller uses the dependency set concept from [90, 33, 37] to temporarily park any processor cache eviction in a searchable structure. In our implementation this is a Volatile Delay Buffer (VDB) so that its effective time to reach PM is no earlier than the time that the last possible transaction with which the eviction could have overlapped has become recoverable. The Persistent Memory Controller in this work improves upon the backend controller of [37] by dispensing with synchronous log replays and victim cache management. The library also covers any writes to PM variables by volatile write-aside log entries made within the speculative scope of an HTM transaction; and then streaming the transactional log record into a non-volatile PM area outside the
HTM transaction. These log streaming writes into PM bypass the VDB. A software mechanism may periodically check the remaining capacity of the PM log area and initiate a log cleanup if needed; for such occasional cleanups, new transactions are delayed, and, after all open transactions have closed, processor caches are flushed (with a closing sfence), the logs are removed.

We refer to our implementation as WrAP, for Write-Aside Persistence, and individual transactions as wraps. We first describe the timestamp mechanism, then the user software library, and finally describe the Persistent Memory Controller implementation details.

5.3.1 Software Library

For HTM we employ Intel’s implementation of Restricted Transactional Memory or RTM, which includes the instructions XBegin and XEnd. Aborting HTM transactions retry with exponential back-off a few times, and then are performed under a software lock. Our HTMBegin routine checks the status of the software lock both before and after an XBegin, to produce the correct indication of conflicts with the non-speculative paths; acquiring the software lock non-speculatively after having backed off. HTMBegin and HTMEnd library routines perform the acquire and release of the software lock for the fallback case within themselves. The remaining software library

![Flow of a transaction with implementation using HTM, cached write sets, timing, and logging.](image)
Algorithm 7: Concurrent WrAP User Software Library

User Level Software WrAP Library:

OpenWrapC ()
// ———— State: OPEN ————–
wrapId = threadId;
Notify Controller Open Wrap wrapId;
startTime = RDTSCP();
Log[wrapId].startTime = startTime;
Log[wrapId].writeSet = {};
sfence();
CLFLUSH Log[wrapId];
sfence();
// ———— State: COMPUTE ———
HTMBegin(); // XBegin

wrapStore (addrVar, Value)
Add {addrVar, Value} to Log[wrapId].writeSet;
Normal store of Value to addrVar;

CloseWrapC (strictDurability)
Log[wrapId].persistTime = RDTSCP();
HTMEnd(); // XEnd
// ———— State: LOG ————–
CLFLUSH Log[wrapId].persistTime;
for num cachelines in Log[wrapId].writeSet
  CLFLUSH cacheline;
if (strictDurability)
  durabilityAddr = 0; // Reset Flag;
tAddr = durabilityAddr;
else tAddr = 0;
sfence();
// ———— State: CLOSE ————–
Notify Controller Closed (wrapId, tAddr);
// ———— State: COMMIT ————–
if (strictDurability)
  // Wait for durable notification from controller
  Monitor durabilityAddr;
procedures are shown in Algorithm 7. Various events that arise in the course of a transaction are shown in Figure 5.3, which depicts the HTM concurrency section with vertical lines and the logging section with slanted lines.

Not shown in Figure 5.3 is a per-thread durability address location that we call durabilityAddr in Algorithm 7. A software thread may use it to setup a Monitor-Mwait coordination to be signaled via memory by the Persistent Memory Controller (as described shortly) when a transacting thread wants to wait until all updates from any non-conflicting transactions that may have raced with it are confirmed to be recoverable.

This provision allows for implementing the strict durability for any transaction because the logs of all other transactions that could possibly precede it in persistence order are in PM—which guarantees the replayability of its log. By contrast, many other transactions that only need the consistency guarantee (correct log ordering) may continue without waiting (or defer waiting to a different point in the course of a higher level multi-transaction operation). The number of active HTM transactions at any given time is bounded by the number of CPUs, therefore, we use thread identifiers as wrapIds. In OpenWrapC we notify the Persistent Memory Controller that a wrap has started. We then read the start time with RDTSCP and save it and an empty write set into its log persistently. The transaction is then started with the HTMBegin routine.

During the course of a transactional computation, the stores are performed using the wrapStore function. The stores are just the ordinary (speculatively performed) store instructions, but are accompanied by (speculative) recording of the updates into the log locations, each capturing the address, value pair for each update, to be committed into PM later during the logging phase (after XEnd).

In CloseWrapC we obtain and record the ending timestamp for an HTM transaction into the persistTime variable in its log. Its concurrency section is then terminated with the HTMEnd routine. At this point, the cached write set for the log
and ending persistent timestamp are instantly visible in the cache. Next, we flush transactional values and the persist timestamp to the log area followed by a persistent memory fence. The transaction closure is then notified to the Persistent Memory Controller with the wrapId, and along with it, the durabilityAddr, if the thread has requested *strict* durability (by passing a flag to closeWrapC) – for which, we use the efficient Monitor-Mwait construct to receive memory based signaling from the Persistent Memory Controller. If strict durability is not requested, then closeWrapC can return immediately and let the thread proceed immediately with *relaxed* durability. In many cases a thread performing a series of transactions may choose relaxed durability over all but the last and then request strict durability over the entire set by waiting for only the last one to be strictly durable.

### 5.3.2 PM Controller Implementation

The Persistent Memory Controller provides for two needs: (1) holding back modified PM cachelines that fall into it at any time T from the processor caches, from flowing into PM until at least a time when all successful transactions that were active at time T are recoverable, and (2) tracking the ordering of dependencies among transactions so that only those that need strict durability guarantees need to be delayed pending the completion of the log phases of those with which they overlap. It implements a VDB (volatile data buffer) as means for the transient storage for the first need, implements a durability wait queue (DWQ) for the second need, and implements a dependency set (DS) tracking logic across the open/close notifications to control the VDB and the DWQ, as described next.

**Volatile Delay Buffer (VDB):**  The VDB is comprised of a FIFO queue and hash table that points to entries in the FIFO queue. Each entry in the FIFO queue contains a tuple of *PM address, data*, and dependency set. On a PM write, resulting from a cache eviction or streaming store, to a memory address not in the log area or pass-through area, the PM address and data are added to the FIFO queue and
Algorithm 8: Hardware WrAP Implementation

**Persistent Memory Controller:**

**Open Wrap Notification (wrapId)**
Add wrapId to Current Open Transactions COT;

**Memory Write (memoryAddr, data)**
// Triggered from cache evict or stream store.
if (((memoryAddr not in Pass-Through Log Area) and (Current Open Transactions COT != {})))
   Add (memoryAddr, data, COT) to VDB;
else
   // Normal memory write
   Memory[memoryAddr] = data;

**Memory Read (memoryAddr)**
if (memoryAddr in Volatile Delay Buffer)
   return latest cacheline data from VDB;
return Memory[memoryAddr];

**Close WrAP Notification (wrapId, durabilityAddr)**
Remove wrapId from Current Open Transactions COT;
if (durabilityAddr)
   durabilityDS = COT;
   // Add pair to Durability Wait Queue DWQ;
   Add (durabilityDS, durabilityAddr) to DWQ;
Remove wrapId from Volatile Delay Buffer elements if earliest VDB elements have empty DS
   // Write back entries to memory in FIFO order
   Memory[memoryAddr] = data;
Remove wrapId from Durability Wait Queue elements if earliest DWQ elements have empty DS
   // Notify waiting thread of durability
   Memory[durabilityAddr] = 1;

tagged with a dependency set initialized to the COT. Additionally, the PM address is inserted into the hash table with a pointer to the FIFO queue entry. If the address already exists in the hash table, then it is updated to point to the new queue entry. On a memory read, the hash table is first consulted. If an entry is in the hash
table, then the pointer is to the latest memory value for the address, and the data is retrieved from the queue. On a hash table miss, PM is read and data is returned. As wraps close, the dependency set in each entry in the queue is updated to remove the dependency on the wrap.

Dependency sets become empty in FIFO order, and as they become empty, we perform three actions. First, we write back the data to the PM address. Next, we consult the hash table. If the hash table entry points to the current FIFO queue entry, we remove the entry in the hash table, since we know there are no later entries for the same memory address in the queue. Finally, we remove the entry from the FIFO queue.

On inserting an entry into the back of the queue, we can also consult the head of the FIFO queue to check to see if the dependency set is empty. If the head has an empty dependency set, we can perform the same actions, allowing for O(1) VDB management.

**Dependency Wait Queue (DWQ):** Strict durability is handled by the Persistent Memory Controller using the Dependency Wait Queue or DWQ, which is used to track transactions waiting on others to complete and notify the transaction that it is safe to proceed. The DWQ is a FIFO queue similar to the VDB with entries containing pairs of the *dependency set* and a *durability address*.

When a thread notifies the Persistent Memory Controller that it is closing a transaction (see steps below), it can request *strict* durability by passing a *durability address*. Dependencies on closing wraps are also removed from the *dependency set* for each entry in the DWQ. When the *dependency set* becomes empty, the controller writes to the *durability address* and removes the entry from the queue. Threads waiting on a write to the address can then proceed.

**Opening and Closing WrAPs:** As outlined in Algorithm 8, the controller supports two interfaces from software, namely those for *Open Wrap* and *Close Wrap* notifications exercised from the user library as shown in Algorithm 7. (Implementa-
tions of these notification can vary: for example, one possible mechanism may consist of software writing to a designated set of control addresses for these notifications). It also implements hardware operations against the VDB from the processor caches: Memory Write, for handling modified cachelines evicted from the processor caches or non-temporal stores from CPUs and Memory Read, for handling reads from PM from the processor caches.

The Open Wrap notification simply adds the passed (wrapId) to a bit vector of open transactions. We call this bit vector of open transactions the Current Open Transactions COT. When the controller receives a Memory Write (i.e., a processor cache eviction or a non-temporal/streaming/uncached write) it checks the COT: if the COT is empty, writes can flow into the PM. Writes that target the log range in PM can also flow into PM irrespective of the COT. For the non-log writes, if the COT is nonempty cache line is tagged with the COT and placed into the VDB.

The Close Wrap controller notification receives the wrapId and durability address, durabilityAddr. The controller removes the wrapId from the Current Open Transactions COT bit mask. If the transaction requires strict durability, we save the durabilityDS and COT as a pair in the DWQ. The controller then removes the wrapId from all entries in the VDB and DWQ. This is performed by simply draining the bit on the dependency set bit mask for the entire FIFO VDB. If the earliest entries in the queue result in an empty dependency set, the cache line data is written back in FIFO order. Similarly, the controller removes the wrapId from all entries in the Durability Wait Queue DWQ.

**Software Based Strict Durability Alternative:** As an alternative for implementing strict durability in the controller, strict durability may be implemented entirely in the software library; we modify the software algorithm as follows. On a transaction start, threads save the start time and an open flag in a dedicated cache line for the thread. On transaction close, to ensure strict durability, it saves its end time in the same cache line with the start time and clears the open flag. It then waits
until all prior open transactions have closed. It scans the set of all thread cache lines and compares any open transaction end times and start times to its end time. The thread may only continue, with ensured durability, once all other threads are either not in an open transaction or have a start or persist time greater than its persist time.

### 5.3.3 Example

The example in Chapter 3.4.7 illustrates a set of four transactions, T1-T4, happening concurrently. The example shown here further illustrates the Persistent Memory Controller. At certain time steps we show the contents of the Persistent Memory Controller’s Volatile Delay Buffer, or VDB which is a FIFO Queue, and the Current Open Transactions or COT in Table 5.1.

First, at time $t_1$, T1 opens, notifying the controller, and records its start timestamp safely in its log. The controller adds T1 to the bitmap COT of open transactions, as shown in Table 5.1. At times $t_2$ and $t_3$, transactions T2 and T3 also open, notify the controller, and safely read and persist their start timestamps. At this point in time the Persistent Memory Controller has a COT of \{1,1,1,0\} and only start timestamps have been recorded in the log. In Table 5.1 at time $t_4$, we also show a random cache eviction of a cache line $X$ that is tagged with the COT of \{1,1,1,0\}.

Transaction T4 starts at time $t_5$ persisting its start timestamp and is added to the set of open transactions on the Persistent Memory Controller, now \{1,1,1,1\}. At time $t_6$, we show a random cache eviction of cache line $Y$, and it is placed at the

<table>
<thead>
<tr>
<th>Time</th>
<th>Event</th>
<th>COT</th>
<th>Volatile Delay Buffer</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1 - 3.3</td>
<td>T1,T2,T3 Start, Evict X</td>
<td>{1,1,1,0}</td>
<td>-</td>
</tr>
<tr>
<td>3.4</td>
<td>T4 Starts</td>
<td>{1,1,1,1}</td>
<td>X: {1,1,1,0}</td>
</tr>
<tr>
<td>5.6</td>
<td>Evict Y</td>
<td>{1,1,1,1}</td>
<td>X: {1.0,0.0}</td>
</tr>
<tr>
<td>6.7</td>
<td>T3 Ends</td>
<td>{1,1,0,1}</td>
<td>Y: {1.0,0.1}</td>
</tr>
<tr>
<td>6.8</td>
<td>T2 Ends</td>
<td>{1,0,0,1}</td>
<td>Y: {1.0,0.1}</td>
</tr>
<tr>
<td>6.9</td>
<td>Evict Z</td>
<td>{1,0,0,1}</td>
<td>X: {1.0,0.0}</td>
</tr>
<tr>
<td>6.10</td>
<td>Evict X</td>
<td>{0,0,0,1}</td>
<td>Z: {0,0,0,1}</td>
</tr>
<tr>
<td>6.11</td>
<td>T1 Ends</td>
<td>{0,0,0,0}</td>
<td>Y: {0,0,0,0}</td>
</tr>
<tr>
<td>6.12</td>
<td>T4 Ends</td>
<td>{0,0,0,0}</td>
<td>Z: {0,0,0,0}</td>
</tr>
</tbody>
</table>

Table 5.1 : Example Contents of the Persistent Memory Controller Current Open Transactions Dependency Set and FIFO Queue
back of the VDB on the Persistent Memory Controller and tagged with \{1,1,1,1\} as shown.

At time \(t_7\), transaction T3 has completed and is removed from the dependency set in the controller and cache line dependencies for \(X\) and \(Y\). When T2 completes writing of logs at time \(t_8\), it is removed from the current and dependency sets in the queue of the Persistent Memory Controller as shown in the table. Note that at time \(t_8\), cache line \(X\) is still not able to be written back to Persistent Memory as it is still tagged as being dependent on T1, similarly \(Y\) is tagged with both T1 and T4. We illustrate the eviction of cache line \(Z\) at time \(t_9\). Cache line \(Z\) is tagged with the set of open transactions COT \{1,0,0,1\}.

At time \(t_{10}\), the cache line \(X\) is again evicted into the VDB of the Persistent Memory Controller and tagged with the set of the two open transactions, T1 and T4. Note that there are two copies of cache line \(X\) in the controller. The one at the head of the queue has fewer dependencies (only dependent on T1) than the recent eviction. Any subsequent read for cache line \(X\) returns the most recent copy, the last entry in the VDB. Note how cache lines at the back of the queue have dependency set sizes that are greater than or equal to entries earlier in the queue.

When T1 completes at \(t_{11}\), the PM controller removes T1 from the COT and those in the VDB. The first copy of \(X\) now has no dependencies in the queue and is safely written back to PM as shown in Table 5.1. At time \(t_{12}\), T4 completes and is removed from the dependency sets in the controller, which allows for \(Y\), \(Z\), and \(X\) to flow to PM in that order.

5.4 Evaluation

We evaluated our method using benchmarks directly running on hardware and through simulation analysis (described in Section 5.4.4). Our simulation evaluates the length of the FIFO buffer and performance against various Persistent Memory write times. In the direct hardware evaluation described next, we employed Intel(R) Xeon(R) E5-
2650 v4 series processors, 12 cores per processor, running at 2.20 GHz, with Red Hat Enterprise Linux 7.2. HTM transactions were implemented with Intel Transactional Synchronization Extensions (TSX) using a global fallback lock. We built our software using g++ 4.8.5. Each measurement reflects an average over twenty repeats with small variation among the repeats.

Using micro-benchmarks and SSCA2 and Vacation, from the STAMP benchmark suite, we compared the following methods:

- **HTM Only**: Hardware Transactional Memory with Intel TSX, without any logging or persistence. This method provides a baseline for transaction performance in cache memory without any persistence guarantees. If a power failure occurs after a transaction, writes to memory locations may be left in the cache, or written back to memory in an out-of-order subset of the transactional updates.

- **WrAP**: Our method. We perform all aspects of the protocol such as logging, reading timestamps, HTM and fallback locking, etc. as described in Section 5.2. The volatile delay buffer in the controller is assumed to be able to keep up with back pressure from the cache, as shown in Section 5.4.4.

- **WrAP-Strict**: Same as above, but we implement the software strict durability method as described in Section 5.3.2. Threads wait until all prior-open transactions have closed before proceeding.

- **PTL-Eager**: (Persistent Transactional Locking). In this method, we added persistence to Transactional Locking (TL-Eager) by persisting the undo log at the time that a TL transaction performs its sequence of writes. The undo-log entries are written with write-through stores and SFENCEs, and once the transaction commits and the new data values are flushed into PM, the undo-log entries are removed.
5.4.1 Benchmarks

The Scalable Synthetic Compact Applications for benchmarking High Productivity Computing Systems [104], SSCA2, is part of the Stanford Transactional Applications for Multi-Processing [39], or STAMP, benchmark suite. SSCA2 uses a large memory area and has multiple kernels that construct a graph and perform operations on the graph. We executed the SSCA2 benchmark with scale 20, which generates a graph with over 45 million edges. We increased the number of threads from 1 to 16 in powers of two and recorded the execution time for the kernel for each method.

Figure 5.4 shows the execution time for each method for the Compute Kernel in the SSCA2 benchmark as a function of the number of threads. Each method reduces the execution time with increasing numbers of threads. Our WrAP approach has similar execution time to HTM Only in the cache hierarchy with no persistence and is over 2.25 times faster than a persistent PTL-Eager method to PM.
Figure 5.5: SSCA2 Benchmark Compute Graph Kernel Speedup as a Function of the Number of Parallel Execution Threads

Figure 5.6: SSCA2 Benchmark Compute Graph Kernel HTM Aborts as a Function of the Number of Parallel Execution Threads
Figure 5.5 shows the speedup for each method as a function of the number of threads when compared to a single threaded undo log for the persistence methods and speedup versus no persistence for the in-cache method HTM Only. Even though the HTM (cache-only) method does better in absolute terms as we saw in Figure 5.4, it proceeds from a higher baseline for single-threaded execution. PTL-Eager yields a significantly weaker scalability due to the inherent costs of having to perform persistent flushes within its concurrent region.

Figure 5.6 shows the number of hardware aborts for both our WrAP approach and cache-only HTM. Our approach introduces extra writes to log the write-set, and, along with reading the system time stamp, extends the transaction time. However, as shown in the Figure, this only slightly increases the number of hardware aborts.

![Graph showing Vacation Benchmark Execution Time as a Function of the Number of Parallel Execution Threads](image)

Figure 5.7: Vacation Benchmark Execution Time as a Function of the Number of Parallel Execution Threads

We also evaluated the Vacation benchmark which is part of the STAMP benchmark suite. The Vacation benchmark emulates database transactions for a travel reservation system. We executed the benchmark with the low option for lower contention
Figure 5.8: Vacation Benchmark Execution Time with Various Persistent Memory Write Times for Four Threads

emulation. Figure 5.7 shows the execution time for each method for the Vacation benchmark as a function of the number of threads. Each method reduces the execution time with increasing numbers of threads. The WrAP approach follows the trends similar to HTM in the cache hierarchy with no persistence, with both approaches flattening execution time after 4 threads. We also examine the effect of strict durability, WrAP-Strict in the figure, and show that strict durability only introduces a small amount of overhead. For just a single thread, it has the same performance as WrAP relaxed as a thread doesn’t need to wait on other threads, as it is durable as soon as the transaction completes.

Additionally, we examined the effect of increased Persistent Memory write times on the benchmark. When compared to DRAM, byte-addressable Persistent Memory can have longer write times. To emulate the longer write times for PM, we insert a delay after non-temporal stores when writing to new cache lines and a delay after cache line flushes. The write delay can be tuned to emulate the effect of longer write
times typical of PM. Figure 5.8 shows the Vacation benchmark execution time for various PM write times.

The WrAP approach is less affected by increasing PM write times than the PTL-Eager approach due to several factors. WrAP performs write-combining for log entries on the foreground path for each thread, so writes to several transaction variables may be combined into a fewer writes. Also, PTL-Eager transactionally persists an undo log on writes causing a foreground delay.

5.4.2 Hash Table

Our next series of experiments show transaction sizes and high memory traffic affect overall performance. We create a 64 MB Hash Table Array of elements in main memory and transactionally perform a number of element updates. For each transaction, we generate a set of random numbers of a configurable size, compute their hash, and write the value into the Hash Table Array.

Figure 5.9: Millions of Transactions per Second for Hash Table Updates of 10 Elements versus Concurrent Number of Threads
First, we create transactions consisting of 10 atomic updates and vary the number of concurrent threads and measure the maximum throughput. We perform 1 million updates and record the average throughput and plot the results in Figure 5.9. Our approach achieves roughly 3x throughput over PTL-Eager. Figure 5.10 shows increasing the write set to 20 atomic updates has similar performance. In both figures, adding strict durability only slightly decreases the overall performance; threads wait additional time for the dependency on other transactions to clear before continuing to another transaction.

The transaction write set was then varied from 2 to 26 elements with 6 concurrent threads. The average throughput was recorded and is shown in Figure 5.11. Even with adding strict durability, WrAP performs roughly three times faster than PTL-Eager.

A transaction size of twenty elements was then analyzed using a varied write to read ratio with 6 concurrent threads. The average throughput was recorded and is
Figure 5.11: Average Txps for Increasing Transaction Sizes of Atomic Hash Table Updates with 6 Concurrent Threads

Figure 5.12: Average Txps for Write / Read Percentage of Atomic Hash Table Updates with 6 Concurrent Threads
shown in Figure 5.12. Unlike transactional memory approaches, our approach does not require instrumenting read accesses and can therefore execute reads at cache speeds.

5.4.3 Red-Black Tree

![Figure 5.13: Millions of Transactions per Second for Atomic Red-Black Tree Element Inserts versus Number of Concurrent Threads](image)

We use the transactional Red-Black tree from STAMP initialized with 1 million elements. We then perform insert operations on the Red-Black tree and record average transaction times and throughput over 200k additional inserts. Each transaction inserts an additional element into the Red-Black tree. Inserting an element into a Red-Black tree first requires finding the insertion point which can take many read operations and can trigger many writes through a rebalance. In our experiments, we averaged 63 reads and 11 writes per transactional insert of one element into the Red-Black tree.
We record the maximum throughput of inserts into the Red-Black tree per second for a varying number of threads in Figure 5.13. As can be seen in the Figure, WrAP has almost 9x higher throughput over PTL-Eager, and with strict durability almost 6x faster. Our method can perform reads at the speed of the hardware, while PTL-Eager requires instrumenting reads through software to track dependencies on other concurrent transactions.

5.4.4 Persistent Memory Controller Analysis

We investigated the required length of our FIFO in the Volatile Delay Buffer and performance with respect to SCM write times using an approach similar to [83, 33]. We first describe the modifications to the system simulator and then describe our evaluation setup.

System Simulator

In the absence of readily available memory controllers, we modified the McSimA+ simulator [105]. The system simulator design is shown in Figure 5.14. McSimA+ is a PIN [106] based simulator that decouples execution from simulation and tightly models out-of-order processor micro-architecture at the cycle level. We extended the simulator to support the notifications for opening and closing WrAPs along with extended support for memory reads and writes. Applications are executed in user-space and PIN based traces from an extended McSimA+ are sent to an Out-of-Order simulation core with extensions to simulate the PM Controller hardware and DRAMSim2 for memory accesses.

We then added support for DRAMSim2 [107], a cycle-accurate memory system and DRAM memory controller model library. This allowed for finely tuned memory configuration parameters and easily adding channels and configurations for PM. Write-combining and store buffers were then added with multiple configuration options to allow fine tuning to match the system to be modeled.
We extended the memory controller to support the PM Controller as shown in Figure 5.14. The extended controller includes multiple memory channels as outlined in Figure 5.14: one for DRAM, one for PM, and one for the persistent PM Log area. A configuration option is provided to allow for the PM Log area to be included in the main PM area, with or without a separate channel. Each channel is simulated by an instance of DRAMSim2 and can be configured from a separate initialization file to model different types of PM or have a slowdown factor cycle time as a multiple of
DRAM cycle times.

**Evaluation**

To stress the Persistent Memory Controller, we executed an atomic hash table update without any thread contention by having each thread update elements on a separate portion of the table. In the simulation, we fill the cache with dirty cache lines so that each write by a thread in a transaction generates write-backs to main Persistent Memory. For 8 threads, we recorded the average atomic hash table update time for 10 elements in each transaction. We then vary the Persistent Memory write time as a multiple of DRAM write time. As shown in Figure 5.15, WrAP is less affected by increasing write times when compared to PTL-Eager.

![Figure 5.15](image)

**Figure 5.15 : Average Atomic Hash Table 10 Element Update with Various Persistent Memory Write Times with 8 Concurrent Threads**

Additionally, we record the maximum FIFO buffer size for various Persistent Memory write times and 4 concurrent threads, shown in Figure 5.16. Initially, the buffer size decreases for an increasing PM write time, due to slower transaction throughput.
Figure 5.16: Maximum FIFO Queue Length for Atomic Hash Table 10 Element Update with 4 Concurrent Threads

and less cache evictions into the buffer. As the write time increases, the buffer length increases, but is still less than 1k cache lines or 64KB.

We performed a similar analysis using a B-Tree, where each thread atomically inserts elements on its own copy of a B-Tree. Each insert into the tree required, on average, over 5 times as many reads as writes. As shown in Figure 5.17, our method is less affected by increasing PM write times, due to PTL-Eager instrumenting the large portion of the read operations. In this experiment, we use eight concurrent threads each atomically inserting elements into an initialized B-Tree of 128 elements.

As more reads than writes are generated for each atomic insert transaction, the FIFO buffer length remains small. We also examined the FIFO buffer length in the VDB with 8 concurrent threads. Figure 5.18 shows the length was less than about 100 elements for each write speed due to the large proportion of reads.
Figure 5.17: Average B-Tree Atomic Element Insert with Various Persistent Memory Write Times with 8 Concurrent Threads

Figure 5.18: Maximum FIFO Queue Length for B-Tree Atomic Element Insert with 8 Concurrent Threads
5.5 Summary

In this Chapter, we presented an approach that unifies HTM and PM to create durable, concurrent transactions. Our approach works with existing HTM and cache coherency mechanisms, and does not require changes to existing processor caches or store instructions, avoids synchronous cache-line write-backs on completions, and only utilizes logs for recovery. The solution correctly orders HTM transactions and atomically commits them to Persistent Memory by the use of a novel software and protocol combined with a back-end Persistent Memory Controller.

The software interface provides primitives for the programmer to mark the start and end of transactions and to mark the updates to persistent memory within the atomic region. Such wrapped updates result in the writing of a log record at the end of the concurrency section for each ordinary store into the cache hierarchy. The Persistent Memory Controller fields evictions and misses from the processor caches. It implements a transparent volatile delay buffer to hold evicted values until the values are safely recorded in logs, which are only used for recovery.

The design does not defer visibility of updates, and thereby permits free and immediate propagation of updated values through processor caches. Our approach, evaluated using both micro-benchmarks and the STAMP suite compares well with standard (volatile) HTM transactions. In comparison with persistent transactional locking, our approach performs 3x faster on standard benchmarks and almost 9x faster on a Red-Black Tree data structure.
Chapter 6

Future Work and Conclusions

Emerging Persistent Memory technologies is an exciting area with many research opportunities. The new HTM technology is awaiting broad adoption across new and legacy applications. This chapter discusses future research areas for both the software cc-HTM and Persistent Memory Controller. The chapter also presents conclusions from both methods.

6.1 Future Work

The following sections outline future research opportunities for both the cc-HTM software and applications and the Persistent Memory Controller.

6.1.1 Future Software Work

Future cc-HTM and software work include porting cc-HTM to the IBM POWER architecture with HTM, creating a distributed container of a parallel data structure store, finalizing work on legacy code tools and data structures, and integrations with compilers and operating systems.

IBM POWER Implementation:

The IBM POWER8™ architecture supports HTM [108]. The IBM Blue Gene/Q uses an extension of the L2 caches for HTM support [80]. It supports up to 128 speculation identifiers which can be recycled for different threads immediately after use and can support up 20 MB of writes. The Power 8 can only support 8KB of writes [80].
Future work includes porting the cc-HTM software approach to IBM’s POWER architecture and performing evaluations. This work can be difficult as the memory ordering mechanisms for POWER are different than Intel® ordering. Additionally, ordering HTM transactions on POWER can also prove difficult due to the absence of the RDTSCP instruction.

**Hybrid Transactional Persistent Memory:**

Three drawbacks for HTM are bounded transaction size, required fallback locking, and limited instruction set usage within the boundaries of the HTM transaction. For larger transactions that will cause an HTM section to abort, a fallback lock must be used. The use of a global fallback lock for applications utilizing HTM for concurrency control can result in many of the transactions serializing on the global lock. Instead of the global fallback lock, Hybrid Transactional Memory, or HyTM, utilizes a Software Transactional Memory, or STM, approach [109] for fallback locking. The STM allows for unbounded transaction size and use of additional processor instructions that may cause an HTM section to abort.

Future work includes integrating our HTPM cc-HTM implementation with STM techniques to create a Hybrid Transactional Persistent Memory, HyTPM. This integration would introduce little additional overhead, as the cc-HTM is already utilizing an alias table. In a hybrid mode, when a load or store consults the alias table, the implementation can be extended to examine or take lock bits on the structures. PHyTM [26] describes a persistent hybrid transactional memory, but it requires changes to HTM implementations in the processor. Our extended technique for HyTPM will work on existing Intel processors.

**Distributed Containerized Persistent Memory Data Structures:**

Containers are a virtualization technique that is gaining popularity in cloud data centers. Many cloud data centers offer the Redis [4] data structure store as a container.
This container can be configured to work with other distributed Redis containers. However, for data durability, Redis relies on file based mechanisms. Appendix A presents a version of the Redis data structure store that we adapted for PM. Appendix B presents Persistent Memory Containers. Future work would be to present a Persistent Memory distributed data structure store using our container technique. Additionally, with containers that support HTM, the data structure store can take advantage of the hardware concurrency control for increased performance.

**Tools for Legacy Code and Building Blocks:**

With the release of Persistent Memory, developers of existing applications will face adoption challenges. An existing application can take immediate, though limited, use of PM by migrating the file-based persistence to a PM file system. However, to take full benefit of PM, it might be necessary to make extensive, time-consuming changes to the application. Several tools can help the developer utilize PM effectively, namely move from file-based persistence to byte-based PM. One Pin based tool we started during the thesis examines code at runtime and can report which lines of code access PM for easy modification.

Application building blocks like the Standard Template Library (STL) are built not only for dynamic memory but also for single threaded access. With the emergence of multi-core, HTM, and PM, a template library that supported both concurrent access and PM could help accelerate PM and HTM adoption. Future work includes developing a library with queues, binary trees, etc. using the techniques developed in this thesis for HTM and PM.

**Compiler and Operating System Integration:**

Identifying areas for transactions to PM can be challenging. Some approaches such as Mnemosyne utilize an *atomic* keyword and compiler integrations. Others, such as ATLAS provide ways of determining atomic areas by analyzing thread locks.
Future work would be to include compiler support so that locked regions of code can utilize HTM for concurrency control and have atomic updates to PM.

Operating Systems typically do not give preference to threads executing a transaction. For HTM, if a thread is executing a transaction, the thread execution can be postponed by the OS scheduler switching to another task. This task switch can abort an HTM transaction and decrease overall system throughput. OS integrations could slightly extend or borrow execution time in the scheduler if the thread is executing in HTM sections.

Additional integrations with the Operating System can include aliasing based on memory pages and not individual memory words. Using the TLB, the OS can manage which pages are aliased to locations in PM and perform copies of the memory pages on stores. The OS can also be modified to provide system calls for the transactions and have areas to manage logs for the entire system.

6.1.2 Future Persistent Memory Controller Work

Future work with the Persistent Memory Controller includes developing a hardware block that can be used on custom systems and analysis on optimizing the controller for systems with multiple memory controllers.

**Hardware Implementation:**

Persistent Memory may be an ideal candidate for low power and embedded systems that require data persistence. PM power requirements are on the order of 10 to 100 times more efficient than Flash technologies [17]. PM might be accessed through another channel on the device and not on the memory bus. Future work includes creating an IP block implemented in High Level Synthesis C. The IP block can then be utilized in systems to create atomic transactions to PM.
Distributed Persistent Memory Controller:

We designed the Persistent Memory Controller to function on the memory side of the cache and not require any interface to the processor cores. With multiple memory controllers per processor chip, and potentially multiple processor chips in a system, our PM Controller needs to work in a distributed manner. Currently, to signal the opening and closing of a transaction, we leverage the open and close of a log write sent to the PM Controller. However, for multiple controllers, each controller will need to be signaled by a transaction. This signaling to each controller for open and closing by each transaction can not only degrade performance but also generate additional, unwanted memory traffic. Future work will include developing a lightweight protocol for the distributed memory controllers. Evaluation of the PM Controller on the processor will be compared with placing the controller on the DIMM.

6.2 Conclusions

Systems with large core counts coupled with large, byte-addressable non-volatile, Persistent Memory offer the potential of breakthrough performance gains on problems with large data footprints. Applications requiring these large data footprints can shift storage from slow, block-based disks and SSDs to fast, parallel PM. Hardware Transactional Memory allows parallel applications to execute with minimal entanglement among threads, impeded only by actual data races, instead of by preventive software synchronization. However, since HTM implementations typically make all changes to memory visible instantly, they create a window of vulnerability for transactions whose updates flow out of order across the memory buses. These out of order updates potentially leave PM based data and structures in a broken state in the event of an untimely machine crash.

This thesis presents an approach that unifies HTM and PM to create durable, high-performance concurrent transactions. Concurrency control is managed by the hardware using HTM. Our approach works with existing HTM and cache coherency
mechanisms, and does not require changes to existing processor caches or store instructions, and avoids synchronous cache-line write-backs on completions.

In this thesis we presented cc-HTM, which is a software solution for a continuous-checkpointing of Hardware Transactional Memory to Persistent Memory. The implementation requires no hardware changes and runs on existing Intel hardware and contains a novel lock-free data structure for transaction ordering. cc-HTM is a lightweight mechanism to propagate the updates from a completed HTM transaction to PM asynchronously while maintaining atomicity and serializability across transactions.

This thesis further introduces relaxed durability, whereby transactions can commit and continue on other tasks, thereby increasing overall transaction throughput and performance. With relaxed durability, the durability of a transaction can be configured to lag. By introducing a small lag parameter, we create a continuous wave of transactions that are persisted to PM fronted by transactional HTM concurrency sections.

We further present and evaluate a Persistent Memory Controller with a user-level software library. The controller works with HTM transactions and only utilizes logs for recovery. With the controller, HTM transactions for PM execute near the same speed as cache-based HTM.

Using micro-benchmarks and benchmarks from the STAMP suite, we showed that our Hardware Transactional Persistent Memory approach for HTM + PM enables fast, parallel atomic persistence to PM. With small values of a lag parameter, cc-HTM achieves its peak throughput, at which it compares well with volatile HTM transactions in many scenarios. cc-HTM significantly outperforms persistent software transactional locking in both throughput and response time. Our PM Controller compares well with volatile HTM transactions running in cache memory alone. In comparison with persistent software transactional locking, our PM Controller approach performs 3x faster on standard benchmarks and almost 9x faster on a Red-Black Tree data
structure.
Appendix A

Persisting In-Memory Databases Using PM

In Memory Databases (IMDBs) have become a popular solution to meet the throughput and response time requirements of applications dealing with large amounts of unstructured data. These applications keep as much data as possible in memory for speed of access. However, reliable data saving and accessing large amounts of data that exceed the dynamic memory size pose a challenge for these applications.

Persistent Memory helps solve this challenge by making memory available in large capacity while making changes endure as a seamless continuation of load-store accesses through processor caches. However, when writing values into a PM tier, programmers are faced with the dual problems of controlling untimely cache evictions that might commit changes prematurely, and grouping changes for consistency in the event of failure.

In this Chapter, we present methods to achieve high-performance byte-addressable persistence for an in-memory data store. We chose Redis, a popular high-performance memory oriented key-value data structure store, and modified its source code to use PM such that updates to data and structures are performed in a failure resilient manner. The full details are provided in the full paper [73]. We found that even though Redis uses many PM read operations, it can benefit from highly optimized PM write based approaches.

A.1 Overview

Traditional applications requiring durability typically serialize their data into consecutive segments before persisting it to disks or SSDs. Reliable saving to non-volatile
media typically requires creating a log to guard against failure and, after the log is committed, writing the updates into the the persistent block store before finally removing the log.

In Memory Databases (IMDBs) and data structure stores are popular solutions to support high-performance demands for large amounts of unstructured data. One such solution is Redis [4], a high-performance in-memory data store that offers both in-memory and persistent data operations. Redis can be accessed over traditional networking channels or can be linked into applications.

A fast data structure store such as Redis can persist data to disk or SSD while keeping as much data as possible in memory for speed as allowed by its configuration. Frequent storing of data to a persistent tier greatly affects performance, while infrequent saving can result in losing a considerable amount of updated data. This is particularly a concern for high-speed data stores, as the less frequently they save, the more data that can be lost; however, saving frequently or appending every update to a continuous log can negate the desired performance goals.

Persistent Memory will allow an in-memory data structure store such as Redis to operate directly in persistent memory, without having to check-point operations and serialize the disk data. However, applications are now faced with managing the dual problems of spurious cache evictions and atomic grouping of stores to guarantee consistency in case of failure. A full discussion on the solutions for managing data and atomic transactions to PM is given in Chapter 2.3.

For our approach, we utilize variable aliasing and Redo Logging to persist data structures in Redis. Variable aliasing is discussed in detail in Chapter 4.1. We utilize a variant on Local Alias Tables or LAT [102] that makes it both read friendly and write fast. For persistence transactions which are strictly isolated from one another, as with serialization schemes such as locks, a shared alias table is not necessary, and local aliasing suffices to forward values from writers to readers.
A.2 Redis Enhancements

Redis can be executed with no persistence, where data is only updated in memory. No data persistence may suffice for some usages, but Redis has several persistence options to meet data durability requirements. Redis persistence options combine a Redis database (RDB) file and an Append Only File (AOF). An RDB file can be configured to be saved either periodically or whenever the number of updates crosses a threshold. With millions of updates per second, even a one second loss of data can result in significant data loss. Therefore, the AOF is an attractive option for preserving updates. This option can be configured to append updates to a file and flush that file frequently (on each update or after each batch of updates).

Instead of logging and committing every store to PM, Redis and other key-value data stores need only store the aggregate result of creating a key-value association. Our approach to migrating Redis to an in-persistent-memory data store is conservative. The approach only changes less than 5% of Redis core source. We removed the block based persistence methods and modified Redis to run safely in emulated PM. In effect, we treat an object as durable unless we find it to be a temporary object used for communication (by inspecting, or by instrumenting code). If a transient object is left in PM, the approach would still work (as the object will get freed at some point) but incur needless overhead in logging and synchronous flushing to PM for transient elements.

After marking persistent/temporary data used by Redis, we ran a Pin tool to identify source locations in the Redis codebase from where durable updates are made to data in PM. We then wrapped those lines of source code using wrap-load/wrap-store calls for primitive data types and wrap-read and wrap-write calls for multi-location accesses such as those produced by memcpy calls. An example source code change for resetting a dictionary entry is shown in Listings A.1 and A.2. In resetting a dictionary entry, the update must be performed atomically so as to not corrupt the in-memory data structure in case of failure.
Listing A.1: Before In-Memory Persistence

```c
static void _dictReset(dictht *ht)
{
    ht->table = NULL;
    ht->size = 0;
    ht->sizemask = 0;
    ht->used = 0;
}

int dictRehash(dict *d, int n) { ...
while(d->ht[0].table[d->rehashidx]==NULL)
{
    d->rehashidx++;
    if (--empty_visits == 0)
        return 1;
}
...
}
```

Listing A.2: After Safely Wrapping Operations for PM

```c
static void _dictReset(dictht *ht)
{
    WrapOpen();
    WrapStore64(ht->table, NULL);
    WrapStore64(ht->size, 0);
    WrapStore64(ht->sizemask, 0);
    WrapStore64(ht->used, 0);
    WrapClose();
}

#define GetDictEntryTbl(table, offset)\
    WrapLoad64(((dictEntry**)WrapLoad64(table))[offset])

int dictRehash(dict *d, int n) { ...
while (GetDictEntryTbl(d->ht[0].table, WrapLoad64(d->rehashidx)) == NULL)
{
    WrapStore64PP(d->rehashidx);
    if (--empty_visits == 0)
        return 1;
}
...
```
A.2.1 Local Alias Table Batched

A local aliasing based LAT approach is well matched with a single threaded application like Redis, however, many operations in Redis are characterized by a high read fraction. Redis may generate stores that happen to overlap at a sub-word granularity from different scopes; which complicates the aliasing of the affected locations and makes it challenging to optimize. We develop a variant of LAT for better handling of read-heavy SET operations while minimizing the number of synchronous commits under update operations.

A common step in Redis is to initialize a structure, such as a dictionary hash table, and then to write the entire structure; this creates a block of entries in an alias table. If fields of the structure are touched from elsewhere in the transaction (which is common), care is needed as explained next. A given variable may or may not be aliased at a point in time. Thus for reads, our implementation has to check whether it is in the alias table (so that if it is not, then an access can be satisfied by reading from its original location in PM). For writes, if the destination alias partially overlaps the source alias (due to changes arising from compression or decompression) then a new entry has to be added to the alias table. In general, creating, maintaining and accessing alias table entries for these mutable structures requires checking boundary conditions to distinguish between when such an overlap exists or does not. When an overlap exists, the access may alias data that is complete or only partial, and for partial data reconstruct part of the data from the alias table and part from PM. This generates a high price for mixing a chain of read/write operations as a block, and primitive reads and writes.

When reads occur more numerously in comparison with writes, it is advantageous to shift to a ‘Copy-on-Write (C-o-W)’ approach, since writes are made to a parallel structure, and reads do not therefore need to be aliased at all. We developed a hybrid scheme that has the write benefits of LAT [102], as shown in Figure 2 alongside the read benefits of Copy-on-Write. The scheme we use, LATB (for Local Alias Table,
Local Alias Table Batched

Batched) works as follows. We create an Undo Log in PM but do not guarantee persistence on a store-by-store basis. We allow the LAT to grow to some threshold size before committing it to PM. We switch from a REDO based consistency to a C-o-W based consistency when we detect block (i.e., multivalue) writes.

Figure A.1 shows an example of the LATB and several writes to a data structure X. The first write is to the entire structure X of size 12. Subsequent writes to fields in the structure X, X.A and X.B, grow the alias table in memory. With the regular LAT, when a read of the structure X is executed, the data structure X must be rebuilt from scattered entries in the alias table. However, with LATB, the entries in the log are copies of the original values, so the table may be flushed at any time. With LATB, we flush the table when it reaches a configurable size or when a block wrapped read is executed that contains values inside the table. Therefore, with LATB, we don’t need to scan the entire table to create X, but rather can read the value directly from main PM memory.
A.3 Evaluation

For evaluation, we used an Intel Xeon(R) CPU E5-2697 v2 12 core processor at 2.70 GHz, with 32 GB DDR3 (4 x 8 GB clocked at 1.867 GHz) running RHEL Server 7.2 with the Linux 4.5.3 kernel compiled (per [110]) with NVDIMM, PMEMFS, and Direct Access (DAX) supports. We built Redis and supporting libraries using gcc 4.8.5. PM emulation is performed using 6 GB of DRAM held back from kernel page management, and we used techniques described in [32] to emulate different PM speeds.

We measure in-memory operations without RDB or AOF options, and compare them to file based persistence, including lossy persistence to block media. Below are the two groups of configurations:

Redis In-Memory Only Configurations:

- **C-o-W**: In-memory structures are updated atomically using Copy-on-Write (synchronous, Undo-Logging).

- **LATB**: Local Alias Table Batched method, which updates PM backed by an asynchronous Undo Log.

- **LAT**: Local Alias Table method with Delayed Write-Back and Redo Log.

File-Based Configurations:

- **SSD**: Updates are logged by non-lossy appends.

- **SSD Lossy**: Appends are grouped and reflected into the log every second.

- **Pmem FS**: Non-lossy appending from every update into a Persistent Memory based file.

The Redis Benchmark Suite consists of many groups of operations, such as key-value Set/Get, Auto-incremented elements, list and set operations, and others. Figure A.2 charts the throughput for the SET key-value test across different PM speeds (in
As PM writes get slower, the LATB method outpaces C-o-W significantly (by 60% at 8x DRAM speed). This underscores the point that curbing the number of persistent commit operations is critical for performance under the wider gap between DRAM and PM speeds.

Figure A.3 compares all the persistence options across all of the tests in the Redis benchmark suite for the case when PM speed equals DRAM speed. For the Set operation, LATB has the best non-lossy performance, and the lossy SSD configuration does much better than the (non-lossy). For the Get operation, all methods operate largely from memory (and processor caches, actually); here, the in-memory C-o-W, LATB, and LAT methods underperform slightly as they are instrumented. In all other benchmarks, the LATB method does well due to its ability to keep loads comparable to baseline Redis while performing only a mildly higher number of synchronous stores over the LAT method.
A.4 Summary

Redis is a popular high-performance in-memory data store that offers in-memory and persistence operations for data. Combining Redis with large, fast, byte-addressable Persistent Memory is a natural fit for Big Data applications.

In this appendix, we presented a method to add PM support to Redis. We modified the Redis source code to use PM directly, so that in-memory updates to data structures are performed safely in light of failure. We found that even though Redis uses many PM read operations, it can benefit from highly optimized PM write based approaches, especially when PM write times are longer than DRAM write times. Our LAT Batched method combines a fast atomic write approach using aliasing with the high read performance of Copy-on-Write based approaches.
Appendix B

Persistent Memory Containers

Container based virtualization is rapidly growing in popularity for cloud deployments and applications due to the ease of deployment and high-performance. Containers are an alternative to full virtualization of a host operating system and devices. They offer lightweight virtualization for applications and services running on the same host operating system with near native performance.

Persistent Memory presents a new challenge for container-based applications, which typically access persistent data through layers of slower file isolation. Traditionally, persistent data accesses in containers are performed through layered file access, which slows byte-addressable persistence and transactional guarantees, or through direct access to drivers, which do not provide for isolation guarantees or security.

This chapter presents a high-performance containerized version of PM for applications running inside a container. The full solution and details are provided in the full paper [111]. The solution solves performance challenges while providing isolation guarantees.

B.1 Virtualization Overview

Virtualization was pioneered by Popek and Goldberg who stated that a virtual machine should be "an efficient, isolated duplicate of the real machine" [112]. Virtualization has three key properties including: 1) Isolation - guests should not be able to affect others; 2) Encapsulation - allowing for easy migration as an entire system can be represented as a single file; and 3) Interposition - the system can monitor or intercept guest operations.
Jails \[113\] was introduced for lightweight virtualization of environments to allow for the sharing of machine resources between several customers or users. Jails isolates files and services between concurrent guests on the same machine through the use of chroot and I/O constraints. Chroot changes the root of the file system to a different location for application-level persistence isolation and security. Chroot has many benefits since a system can be shared securely with little performance burden, however services such as CPU and memory were not isolated by Jails and could be abused by users.

Linux Containers (LXC) \[114\] were introduced as the Linux version of Jails. The implementation added features to restrict memory and CPU usage that extended its isolation features. Docker \[41\] is an open-source Linux project which automates the deployment of applications or bundled services inside of Linux Containers. Volatile memory is handled from inside a container using regular virtual memory accesses. Using Linux CGroups, limits on virtual memory can be imposed by the operating system.

Handling access to persistent data inside a Docker Container is not a choice to be taken lightly, as there are several choices available: using a container file or traditional Docker Storage, an external or Docker Volume, or direct access to a device. Each has its own advantages and disadvantages and needs to be specified up-front on container start if requiring a special device or volume.

- 1. **Docker Storage**: Storage to traditional files inside a container are accessed using a pluggable storage driver architecture. File accesses are layered using AUFS (or Another Union File System), Device Mapper, OverlayFS, VFS, or ZFS. A layered access requires slow copying of data through multiple file or persistence layers.

- 2. **Docker Volumes**: Docker Volumes are used to share data between containers or between the container and the host. Volume specifications are passed
as options on startup of a container, and cannot be added later. Data is not isolated, so changes in one environment affect the other.

3. **Direct Device Access:** Direct access to a device is specified and granted on the start of a container, and the exact same driver must be present on restart. Special privilege must also be granted to the container which violates isolation principles, as it allows each container to access or overwrite data in the shared driver among containers launched with the same specification.

![Diagram of container based storage with byte-addressable, fast Persistent Memory.](image)

Figure B.1: Container based storage with byte-addressable, fast Persistent Memory.

Access to Persistent Memory can be presented to a container using one of the options listed above. However, Persistent Memory presents a new situation for container based storage as PM offers byte-addressable memory that is also persistent. See Figure B.1 which also shows traditional container based file storage in relation to PM. Accessing PM through a traditional mapping call to a driver or file system poses a problem for applications running inside an isolated container. An `mmap` of a file through an isolation layer loses the performance benefits of PM due to cascading persistence consistency guarantees in layered file accesses, as with Docker Storage or
Docker Volumes. In addition, mapping a shared volume or device, as with Docker Volumes or Direct Device Access, to multiple containers can remove two important properties of virtualization from the containers: **persistence isolation** and **portability**.

### B.2 Containerized Persistent Memory Approach

To solve the isolation, portability, and performance issues as noted previously, we developed a Containerized Persistent Memory design utilizing a Linux loadable Kernel Module or LKM. Our approach provides isolation through the driver and management and portability through the file layer, while allowing direct access to PM through loads and stores. The approach bypasses the layered access for persistent file access allowing for both high performance and scalability.

![Figure B.2: Containerized Persistent Memory System Design](image-url)
The system design is comprised of three main components shown in Figure B.2:

- **Docker CLI:** The Docker Command Line Client Integrations are simple additions we performed via the Docker API to add functionality when launching or saving a Docker Image.

- **User Library:** The User-Level Library can be any library that provides a Persistent Memory allocator and optional persistence mechanisms or transaction support, such as pmem.io [110] or SoftWrAP [63]. We provide a basic working implementation for accessing PM and flushing updates through the cache hierarchy for persistence consistency and with configurations for evaluation purposes.

- **CPM LKM Driver:** Our Containerized Persistent Memory Linux loadable Kernel Module driver. Most of the work is performed in the driver, as described below, to provide isolated, scalable access to PM through the Docker system.

The Containerized PM Linux loadable Kernel Module (CPM LKM) creates a PM data area for the container and sets up the node and major version number. Our CPM LKM creates and registers our container supported /dev/cpm device for the Linux host operating system and registers the mmap handler on the device.

The Docker container is executed using a privileged device driver option pointed to the device /dev/cpm created from our CPM LKM driver. This configuration gives privileged access to the container to read and write to our device. On container restart the driver is attached through the active device to the persistent PM data. Even though the device access has elevated privileges, our CPM driver performs accesses to PM via loads and stores in isolation as described below.

On CPM LKM driver installation, the /dev/cpm device is created for the Linux host operating system. Applications access the driver by using a regular file open call and then may call mmap to access and map data into user space. On an mmap the pseudo-code shown in Listing B.1 is executed.
Figure B.3: Data Layout on /dev/pmem0 for CPM

<table>
<thead>
<tr>
<th>File System Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXT4 DAX File System</td>
<td>Formatted on /dev/pmem0</td>
</tr>
<tr>
<td>/pmemfs</td>
<td>Mount point for /dev/pmem0</td>
</tr>
<tr>
<td></td>
<td>- &gt; hostdata</td>
</tr>
<tr>
<td></td>
<td>- &gt; volumes</td>
</tr>
<tr>
<td></td>
<td>- &gt; hostpmfs</td>
</tr>
<tr>
<td></td>
<td>- &gt; / var / lib / docker /</td>
</tr>
<tr>
<td></td>
<td>- &gt; dfpmdata</td>
</tr>
<tr>
<td></td>
<td>- &gt; containers /</td>
</tr>
<tr>
<td></td>
<td>- &gt; 230c02125811</td>
</tr>
<tr>
<td></td>
<td>- &gt; images /</td>
</tr>
<tr>
<td></td>
<td>- &gt; 432717234123</td>
</tr>
</tbody>
</table>

Listing B.1: CPM LKM mmap flow

```c
static int pm_mmap( filp , vma){ ... 
  // Get the file system root
  // Detect chroot (is fs root /)
  // Get PM location
  //  If root, /pmemfs/hostdata
  //  else examine /proc/1/cgroup
  // Lock process
  // Open PM File
  // Unlock process
  // Setup VMA generic_file_mmap
}
```
When using `mmap` to access data, our driver executes `pm_mmap`. The function first determines the file system root and container level isolation via OS calls to detect `chroot`. Next, based on the result and operating system type, an offset into the base operating file system is determined. This allows our driver to manage PM data on a file basis for containers for portability, while still serving DAX byte-level persistence through the driver. The driver also determines the generic Virtual Memory Address range, or VMA, and determines if the application is running inside a container and, if so, extracts the container id. Once the container id is determined, the appropriate underlying file is securely accessed by the driver. This secure access by our driver allows for both isolation and portability. The contents are then directly mapped since the file uses Direct Access Extensions underneath. The direct access bypasses the layered accesses, and the page mappings are returned to the original `mmap` call. If the PM data file is not present, it is created on the first access. Figure B.3 shows the Containerized PM Linux Loadable Kernel Module driver and file layout.

### B.3 Evaluation

For evaluation, we used an Intel Xeon(R) CPU E5-2697 v2 12 core processor at 2.70 GHz, with 32 GB DDR3 (4 x 8 GB clocked at 1.867 GHz) running RHEL Server 7.2 with the Linux 4.5.3 kernel compiled (per [110]) with NVDIMM, PMEMFS, and Direct Access (DAX) supports. A 6GB persistent memory emulation space for the pmem driver was created in DRAM and a ext4 file system was created and mounted for persistent memory file system emulation. We used Docker Version 1.10.3 and API for Client Version 1.22 and our CPM LKM driver was built using gcc 4.8.5.

We tested several benchmarks in Redis [4], an in-memory data structure store, that offers lists, sets and hash tables. We configured Redis to operate in memory and integrated it with the CPM user library by modifying the memory allocator Jemalloc that ships with Redis to use our CPM pmalloc routines. We executed the Redis benchmarks for set, get, and list push times and recorded the average
requests serviced per second. The benchmark is scaled to use 40 threads performing simultaneous requests. Figure B.4 shows the CPM driver in a Docker container has equal requests per second throughput as the same benchmark running on the host. Our approach has twice the throughput as the Docker Storage or Volume options since requests do not have to flow through the Docker Storage or Volume drivers. Additionally, CPM allows for the containerized Redis to have near the same performance as executing on the host.

We tested a micro-benchmark that transactionally adds elements to a Persistent Memory initialized transactional persistent B-Tree. An insert into a persistent B-Tree can require safely updating multiple elements from the root node to the leaf. These updates have to be performed atomically to preserve consistency of the data structure. We added synchronization using \textit{pmsync} in the CPM library to the B-Tree tests. The B-Tree is initialized with 200k elements, and we then record the time it takes to add another 200k elements and calculate the average element insert throughput. Figure
Figure B.5: Average Number of B-Tree Random Element Insert Transactions Per Second

B.5 shows the B-Tree element insert throughput. The CPM driver running in the Docker Container has the same throughput as when running on the host. However, when PM is accessed through Docker Volumes, performance degrades by a factor of 10. This is due to the additional level of synchronization required by the volume for the copying. Even slower is the Docker Storage driver which is another factor of 10 slower than the Docker Volume driver, making it two orders of magnitude slower than the host or our Containerized Persistent Memory device. The large slowdown is due to the storage driver going through additional persistence layers when copying values for consistency.

B.4 Summary

Running applications inside containers using Docker is growing in popularity as it presents a low-cost, high performance method for isolating applications and services.
Emerging Persistent Memory presents an interesting challenge for in-memory persistent applications running inside a container.

This appendix presented and evaluated our Containerized Persistent Memory approach, or CPM. Our custom Linux loadable Kernel Module driver provides Docker containers with access to PM using an *mmap* interface for in-memory applications. CPM provides isolated, direct access to PM, where consistency guarantees do not suffer from multiple persistence layers each requiring consistency.

We found our applications using the CPM approach to have the highest in-memory application throughput. For transactional workloads, applications using CPM had orders of magnitude higher throughput over Docker Storage and Docker Volumes while still achieving container persistence isolation.
Appendix C

Evaluation on Intel Optane DC PM DIMMs

This chapter presents the evaluation of the HTPM implementations in this thesis, described in Chapters 4 and 5, on a machine with Persistent Memory DIMMs. Intel has graciously provided us early access to a high-performance server equipped with Intel® Optane DC Persistent Memory DIMMs and a Cascade Lake processor that supports HTM. Intel® has approved the performance results for release and inclusion in the thesis. We found our results followed similar trends as evaluated with emulated PM in prior chapters. Overall, we found our PM Controller to have over 15x speedup and cc-HTM to have over 6x speedup over persistent transactional locking.

C.1 Persistent Memory Configuration

Intel® has graciously provided access to a server equipped with both HTM and early-access PM DIMMs. The machine includes 12 x 128GB Intel® Optane DC Persistent Memory DIMMs for a total PM size of over 1.5 TB. The processor is a Intel® Cascade Lake processor with 24 hardware cores and 2-way hyper-threading support. The processor supports Intel Transactional Synchronization Extensions (TSX) [38] for HTM called Restricted Transactional Memory, or RTM.

The system is configured with Red Hat Linux 7.3.1-2 running Linux kernel 4.15.6 and gcc version 7.3.1. The kernel is configured and built with Persistent Memory and Direct Access (DAX) support per [110]. The modification for the kernel configuration is shown in Listing C.1

With this new configuration, the kernel builds and installs the libnvdimm subsystem as a kernel module. The libnvdimm, or Non-Volatile DIMM (NVDIMM) Library,
Listing C.1: Linux Kernel Configuration for Persistent Memory

```
CONFIG_X86_PMEM_LEGACY_DEVICE=y
CONFIG_X86_PMEM_LEGACY=y
CONFIG_BLK_DEV_PMEM=m
CONFIG_LIBNVDIMM=y
CONFIG_BLK_DEV_PMEM=m
CONFIG_ND_BLK=m
CONFIG_ND_CLAIM=y
CONFIG_ND_BTT=m
CONFIG_BTT=y
CONFIG_ND_PFN=m
CONFIG_NVDIMM_PFN=y
CONFIG_NVDIMM_DAX=y
CONFIG_DAX=y
CONFIG_DEV_DAX=m
CONFIG_DEV_DAX_PMEM=m
CONFIG_NVMEM=m
```

subsystem creates a control interface for Persistent Memory DIMMS or NVDIMMs.

To manage this control interface, we utilized a tool entitled `ndctl`, or Non-Volatile DIMM Control. The `ndctl` tool provides a command line interface to enable and disable the NVDIMMs and create device access points in the Linux kernel for use by applications.

To effectively expose a PM device to applications, we executed the command sequence shown in Listing C.2. The first commands list and then enable all of the NVDIMMs. Once all of the NVDIMMs are enabled, we create a usable region on each of the NVDIMMs. We then created a namespace that spanned the regions on all of the NVDIMMs. This namespace is exposed to applications as the device `/dev/pmem12`. After the device namespace was created, we created the DAX supported XFS filesystem and mounted it for use by applications as the directory `/mnt/pmem`. Now applications are able to create files in the file directory structure under `/mnt/pmem` and map them into memory space. The access is described in Chapter 2.1 Persistent
Memory Access. Once in memory space, PM may be accessed directly with normal loads and stores.

## C.2 Evaluation

To utilize the configured Persistent Memory device on established benchmarks that use regular volatile memory, we had to redirect memory allocations in the benchmarks to use the PM device.

The pmem.io programming framework [110] provides a mechanism through a kernel preload library to redirect all volatile memory management to a PM file. For example, the execution sequence: `LD_PRELOAD=./pmalloc.so command`, loads the library `pmalloc.so` before launching the application command. This pre-loaded library overrides function definitions for memory management routines such as `malloc` and `free`. However, the library provided by pmem.io [110] does not work well with concurrent accesses and aborts HTM transactions. To overcome this limitation, we created a similar overlay library for memory management, but that works well with HTM...
transactions. Our implementation limits shared variable access during management routines that might cause transactions to abort.

We evaluated our HTPM methods including both cc-HTM and the PM Controller user library using benchmarks directly running on the Intel early-access PM DIMMs and HTM supported processor hardware. HTM transactions were implemented with Intel Transactional Synchronization Extensions (TSX) using a global fallback lock. We built our software and all benchmarks using g++ 7.3.1. Each measurement reflects an average over ten repeats.

Using micro-benchmarks and SSCA2, Vacation, and a RB-Tree from the STAMP benchmark suite, we recorded results from the following methods:

- **HTM Only**: Hardware Transactional Memory with Intel TSX, without any logging or persistence. This method provides a baseline for transaction performance in cache memory without any persistence guarantees. If a power failure occurs after a transaction, writes to Persistent Memory locations may be left in the cache, or written back to memory in an out-of-order subset of the transactional updates.

- **cc-HTM**: Continuous Checkpointing of HTM for Persistence in PM. This is our full implementation of HTPM running on existing hardware. The cc-HTM method uses the Persistence Management Thread, Queue, Logging, and Alias Table as described in Chapter 4.

- **PM-Controller**: Our PM Controller. We perform all aspects of the protocol such as logging, reading timestamps, HTM and fallback locking, etc. as described in Chapter 5. The volatile delay buffer in the controller was shown to be able to keep up with back pressure from the cache.

- **PM-Controller-Strict**: Same as above, but we implement the software strict durability method as described in Chapter 5.3.2. Threads wait until all prior-open transactions have closed before proceeding.
• **PTL-Eager**: (Persistent Transactional Locking). In this method, we added persistence to Transactional Locking (TL-Eager) \cite{77, 40, 79} by persisting the undo log at the time that a TL transaction performs its sequence of writes. The undo-log entries are written with write-through stores and **SFENCE**s, and once the transaction commits and the new data values are flushed into PM, the undo-log entries are removed.

After recording measurements from the above methods, we calculated the relative performance of our methods to HTM Only. We also calculated the speedup compared to PTL-Eager. In this manner, we do not release any raw performance numbers.

## C.2.1 Benchmarks

The Scalable Synthetic Compact Applications for benchmarking High Productivity Computing Systems \cite{104}, SSCA2, is part of the Stanford Transactional Applications for Multi-Processing \cite{39}, or STAMP, benchmark suite. SSCA2 uses a large memory area and has multiple kernels that construct a graph and perform operations on the graph. We executed the SSCA2 benchmark with scale 20, which generates a graph with over 45 million edges. We increased the number of threads from 1 to 16 in powers of two and recorded the execution time for the kernel for each method.

Figure \[\text{C.1}\] shows the relative performance to HTM Only for each method for the Compute Kernel in the SSCA2 benchmark as a function of the number of threads. Similar to the emulation in prior Chapters, each method reduces the execution time with increasing numbers of threads. Our PM Controller approach has similar performance to HTM Only in the cache hierarchy and is 50\% to 100\% faster than the PTL-Eager method to PM. Similarly, our cc-HTM method also is close to HTM Only and faster than the more synchronous PTL-Eager. With 16 concurrent threads, the cc-HTM approach falls off due to having only a single thread for durability management.

Figure \[\text{C.2}\] shows the speedup for each method as a function of the number of
Figure C.1 : SSCA2 Benchmark Compute Graph Kernel Performance Relative to HTM Only as a Function of the Number of Parallel Execution Threads

Figure C.2 : SSCA2 Benchmark Compute Graph Kernel Speedup as a Function of the Number of Parallel Execution Threads
Figure C.3: SSCA2 Benchmark Compute Graph Kernel HTM Aborts Relative to HTM Only as a Function of the Number of Parallel Execution Threads

Figure C.4: SSCA2 Compute Kernel Normalized Transactional Throughput with Varying Maximum Allowable Lag for 8 Threads
threads when compared to a single threaded undo log for the persistence methods. PTL-Eager yields a weaker scalability due to the inherent costs of having to perform persistent flushes within its concurrent region.

Figure [C.3] shows the relative number of aborts for each approach when compared to cache-based HTM Only. Our approach introduces extra writes to log the write-set, and, along with reading the system time stamp, extends the transaction time. When cc-HTM utilizes 8 threads, the fallback lock is taken even more frequently, and therefore more transactions are waiting. With more waiting transactions, more end up taking the fallback lock, thereby reducing the number of HTM attempts and aborts. The increase in the number of aborts due to the write-set capture for logging is shown in the PM Controller approach, as this is the only extra work inside the HTM that this approach performs. cc-HTM also includes the write-set saving and also includes aliasing, thereby increasing the relative number of aborts. PTL-Eager has no hardware aborts, but has more expensive software aborts that are managed by slower software.

Figure [C.4] shows how the maximum allowable Lag can affect the overall performance. We use 8 threads and vary the maximum allowable Lag $L$ in the SSCA2 benchmark and show that after $L$ is increased to just 16, throughput is insensitive to additional increases in Lag and cc-HTM achieves peak transactional throughput.

We also evaluated the *Vacation* benchmark which is part of the STAMP benchmark suite. The *Vacation* benchmark emulates database transactions for a travel reservation system. We executed the benchmark with the low option for lower contention emulation. Figure [C.5] shows the relative transaction rate for each method for the *Vacation* benchmark as a function of the number of threads as compared to HTM Only. cc-HTM has twice the performance of PTL-Eager, but after increasing to 8 threads in the vacation benchmark, the performance of PTL-Eager improves over cc-HTM. We also examine the effect of *strict* durability, PM Controller Strict in the figure, and show that *strict* durability only introduces a small amount of overhead.
for the controller. For just a single thread, strict has the same performance as relaxed as a thread doesn’t need to wait on other threads and is durable as soon as the transaction completes. As the number of threads increases, a thread requiring strict durability has to wait on increasing numbers of threads, thereby decreasing relative throughput.

C.2.2 Hash Table

Our next series of experiments show how transaction sizes and high memory traffic affect overall performance. We create a 64 MB Hash Table Array of elements in main memory and transactionally perform a number of element updates. For each transaction, we generate a set of random numbers of a configurable size, compute their hash, and write the value into the Hash Table Array.

First, we create transactions consisting of 10 atomic updates and vary the number of concurrent threads and measure the throughput. We perform 1 million updates of
Figure C.6: Transaction Throughput Relative to HTM Only for Hash Table Updates of 10 Elements versus Concurrent Number of Threads

Figure C.7: Speedup Over PTL-Eager for Hash Table Updates of 10 Elements versus Concurrent Number of Threads
10 elements each and record the average throughput for each method. Figure C.6 shows the relative performance to HTM Only. Since this benchmark is memory write intensive, not much other processor computation work is performed. Therefore, the difference between HTM Only, which can operate in the processor cache, and any persistence method will be high. Since the persistence methods perform extra work, such as persisting the write-set log in-between consecutive transactions, there are fewer concurrent HTM transactions and therefore fewer aborts when compared to HTM Only. With fewer aborts for increasing transactions when compared to HTM Only, our HTM based persistence methods have more room to improve.

Figure C.7 shows the speedup of each of our methods over PTL-Eager. The PM Controller approach achieves over 6x throughput over PTL-Eager. Adding strict durability only slightly decreases the overall performance; threads wait additional time for the dependency on other transactions to clear before continuing to another transaction. cc-HTM also achieves a high speedup over PTL-Eager. Since cc-HTM has only a single thread in the current implementation to process transaction logs, the speedup decreases with increasing numbers of threads beyond 4, but continues to outperform PTL-Eager by a factor of 2.

The transaction write set was then varied from 2 to 30 elements with 6 concurrent threads. The average throughput was recorded and the relative performance to HTM Only is shown in Figure C.8. As the size of the transactions increase, all methods take longer to complete. Since the persistence methods for HTM perform additional logging writes, the relative performance decreases initially. Figure C.9 shows the speedup of our methods over PTL-Eager. Even with adding strict durability to the controller, our methods perform roughly five times faster than PTL-Eager.

A transaction size of twenty elements was then analyzed using a varied write to read ratio with 6 concurrent threads. The average throughput was recorded and the relative throughput to HTM Only is shown in Figure C.10. cc-HTM is roughly unaffected by changes in the write to read ratio, as all reads and writes are performing
Figure C.8: Transaction Rate Relative to HTM Only for Increasing Transaction Sizes of Atomic Hash Table Updates with 6 Concurrent Threads

Figure C.9: Speedup Over PTL-Eager for Increasing Transaction Sizes of Atomic Hash Table Updates with 6 Concurrent Threads
Figure C.10: Transaction Rate Relative to HTM Only for Write / Read Percentage of Atomic Hash Table Updates of 20 Elements with 6 Concurrent Threads

Figure C.11: Speedup Over PTL-Eager for Write / Read Percentage of Atomic Hash Table Updates of 20 Elements with 6 Concurrent Threads
additional work for aliasing. This additional work is more than the work required for logging as writes are write-combined in a log which only slowly increases execution time. Since the PM Controller doesn’t do any additional work for reads, as more writes are introduced in relation to the reads, more work is performed, slowing performance in relation to HTM Only. Unlike software transactional memory approaches, our PM Controller approach does not require instrumenting read accesses and can therefore execute reads at cache speeds. When compared to PTL-Eager as shown in Figure C.11 our controller is 6x faster. Our cc-HTM method performs additional work when compared to PTL-Eager for reads; so as the number of reads decreases in relation to the number of writes, the speedup of cc-HTM over PTL-Eager increases, also close to 6x.

C.2.3 Red-Black Tree

![Red-Black Tree Graph]

Figure C.12: Transaction Rate Relative to HTM Only for Atomic Red-Black Tree Element Inserts versus Number of Concurrent Threads
We use the transactional Red-Black tree from STAMP [39] initialized with 500k million elements. We then perform insert operations on the Red-Black tree and record average transaction times and throughput over 100k additional inserts. Each transaction inserts an additional element into the Red-Black tree. Inserting an element into a Red-Black tree first requires finding the insertion point which can take many read operations and can trigger many writes through a rebalance. In our experiments, we averaged 112 reads and 14 writes per transactional insert of one element into the Red-Black tree.

We record the maximum throughput of inserts into the Red-Black tree per second for a varying number of threads. Figure C.12 shows the relative performance of each of our methods to the HTM Only. HTM Only is inserting elements into the RB tree faster and therefore has a higher chance for aborts. Thus, the persistence methods, which experience a cost of logging, have less contention and can increase performance more with increasing numbers of threads. Figure C.13 shows the speedup of our
methods over PTL-Eager. cc-HTM also maintains a significant improvement over PTL-Eager. As can be seen in the Figure, our PM Controller has over a 15x higher throughput than PTL-Eager. Our PM Controller method can perform reads at the speed of the hardware, while PTL-Eager requires instrumenting reads through software to track dependencies on other concurrent transactions.

C.3 Summary

In this chapter, we presented the evaluation of both the software only and controller assisted user library for HTPM implementations executing on existing Intel hardware. Intel kindly allowed us early access to a high-performance server equipped with Intel® Optane DC Persistent Memory DIMMs and a Cascade Lake processor that supports HTM. Overall, we found our PM Controller to have over 15x speedup and cc-HTM to have over 6x speedup over persistent transactional locking.
Bibliography


[98] W. Ruan, Y. Liu, and M. Spear, “Boosting timestamp-based transactional


