RICE UNIVERSITY

WrAP: Hardware and Software Support for Atomic Persistence in Storage Class Memory

by

Ellis Giles

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE

Master of Science

APPROVED, THESIS COMMITTEE:

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
Associate Professor of Computer Science

Houston, Texas
April, 2015
ABSTRACT

WrAP: Hardware and Software Support for Atomic Persistence in Storage Class Memory

by

Ellis Giles

In-memory computing is gaining popularity as a means of sidestepping the performance bottlenecks of traditional block-based storage devices. However, the volatile nature of DRAM makes these systems vulnerable to system crashes, while the need to continuously refresh massive amounts of passive memory-resident data increases power consumption. Emerging storage-class memory (SCM) technologies, like Phase Change Memory and Memristors, combine fast DRAM-like cache-line access granularity with the persistence of storage devices like disks or SSDs, resulting in potential 10x - 100x performance gains, and low passive power consumption.

This unification of storage and memory into a single directly-accessible persistent storage tier is a mixed blessing, as it pushes upon developers the burden of ensuring that SCM stores are ordered correctly, flushed from processor caches, and if interrupted by sudden machine stoppage, not left in inconsistent states. The complexity of ensuring properly ordered and all-or-nothing updates is addressed in this thesis in both a software-hardware architecture and a software-only based solution.

This thesis extends and evaluates a software-hardware architecture called WrAP, or Write-Aside Persistence, for atomic stores to SCM. This thesis also presents Soft-WrAP, a library for Software-based Write-Aside Persistence, which provides lightweight
atomicity and durability for SCM storage transactions. Both methods are shown to provide atomicity and durability while simultaneously ensuring that fast paths through the cache, DRAM, and persistent memory layers are not slowed down by burdensome buffering or double-copying requirements.

Software-hardware architecture evaluation of trace-driven simulation of transactional data structures indicates the potential for significant performance gains using the WrAP approach. The SoftWrAP library is evaluated with both handcrafted SCM-based micro-benchmarks as well as existing applications, specifically the STX B+Tree library and SQLite database, backed by emulated SCM. Our results show the ease of using the API to create atomic persistent regions and the significant benefits of SoftWrAP over existing methods such as undo logging and shadow copying. SoftWrAP can match non-atomic durable writes to SCM, thereby gaining atomic consistency almost for free.
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 point out interesting areas for more exploration. Only with his guidance and support has this research, including this thesis and papers, been possible. I really cannot thank him enough.

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 revising papers and running system memory traces. Additionally, I would like to thank both Dr. Joseph Cavallaro and Dr. Chris Jermaine 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 them, day after day, I would not have been able to attend graduate school or immerse myself in research.

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
Acknowledgments ........................ iv
List of Illustrations ....................... viii
List of Tables ............................. xi
List of Algorithms ......................... xii

1 Introduction and Overview ............. 1
   1.1 Introduction .......................... 1
   1.2 Problem Overview ..................... 5
   1.3 Example Transactions .................. 7
      1.3.1 Relation to Database Systems and Transactional Memory ... 12
   1.4 Contributions ........................ 14
   1.5 Thesis Organization .................. 16

2 Background and Related Work ......... 18
   2.1 Storage Class Memory ................ 18
   2.2 Memory Architecture ................ 20
   2.3 Atomic Persistence ................... 23
      2.3.1 Approach 1: Non-Atomic with Persistence Instructions .... 26
      2.3.2 Approach 2: Cache Ordering Primitives ................. 26
      2.3.3 Approach 3: Block Copy with Atomic Pointer Updates .... 28
      2.3.4 Approach 4: Transaction Memory .................. 29
      2.3.5 Approach 5: Undo Logging ................... 30
   2.4 Related Work ........................ 34
3 Software-Hardware WrAP Architecture

3.1 WrAP Architecture ................................. 38
  3.1.1 Background Operations ......................... 42
  3.1.2 Restart and Recovery ......................... 45

3.2 Evaluation ........................................ 46
  3.2.1 Performance Model .............................. 47
  3.2.2 Experimental Results .......................... 49

3.3 Summary ........................................... 59

4 SoftWrAP .............................................. 61

4.1 SoftWrAP Approach ................................. 61

4.2 SoftWrAP API and Implementation ................. 66
  4.2.1 Alias Table Design ............................... 70
  4.2.2 Aliasing Alternatives ............................ 73
  4.2.3 SoftWrAP Architecture ........................... 74
  4.2.4 Application Program Interface .................... 76
  4.2.5 Simulation and Tools ............................. 77

4.3 Evaluation ........................................... 81
  4.3.1 Experimental Results ........................... 83
  4.3.2 Model Validation ................................ 84
  4.3.3 Micro-Benchmarks ............................... 88
  4.3.4 B+Tree Benchmark ............................... 101
  4.3.5 TPC-C Database Benchmark ....................... 105

4.4 Summary ............................................ 107

5 Forward Work and Conclusions ......................... 109

5.1 Forward Work ...................................... 109
5.1.1 Hardware WrAP ........................................... 109
5.1.2 SoftWrAP .................................................. 111
5.2 Conclusions .................................................... 114

Bibliography ..................................................... 116
Illustrations

1.1 Storage Class Memories are fast, byte-addressable, and persistent.  
Adapted from [7]. ......................................................... 2
1.2 SCM will sit alongside DRAM on the main memory bus. ............ 3
1.3 Uncontrolled cache evictions can result in arbitrary mixes of new and  
old values in SCM. ....................................................... 6
1.4 Various methods where groups of stores may be written to main  
memory. ................................................................. 10

2.1 Write Combining Buffers and Streaming Stores to SCM .......... 21
2.2 Persistent memory mmaped objects A and B. ....................... 24
2.3 Example Undo Log used to write a variable. ....................... 30

3.1 WrAP Architecture From [38] ..................................... 39
3.2 Log structures maintained by WrAP from [38] ...................... 44
3.3 Trace Generation for STL Benchmarks ............................ 48
3.4 Comparison of WrAP and Undo Log for a simple benchmark of  
Execution Time vs CPU Delay ....................................... 50
3.5 Comparison of WrAP and Undo Log for a simple benchmark of  
Speedup vs CPU Delay for hot and cold caches .................... 51
3.6 Example time spent for a transaction of N=2 for Undo Log and WrAP. 52
3.7 Comparison of WrAP and Undo Log for a simple benchmark of  
Speedup vs Cache Size ............................................... 53
3.8 Comparison of execution time for WrAP and Undo Log for
STL-Based Benchmarks ............................................. 54
3.9 Comparison of WrAP and Undo Log for STL-Based Benchmark:
Execution time vs Transaction size ............................... 55
3.10 Comparison of WrAP and Undo Log for STL-Based Benchmark:
Number of store operations vs Transaction size .................. 56
3.11 Execution time for B-Tree vs Transaction size .................. 57
3.12 Benchmark Performance versus Cache size ...................... 57
3.13 STL-Based Benchmarks (a) WrAP Log Size (b) VPC Size ......... 58

4.1 SoftWrAP managing paths through foreground cache hierarchy and
background streaming stores to log. ............................... 63
4.2 Updates to an atomic region using SoftWrAP .................... 65
4.3 The Alias Table implementation for global aliasing is a
double-buffered lock-free hash table implementation. It handles reads
and writes to both primitive data types and object data. Values are
retired directly from the table. ........................................ 70
4.4 The lifecycle of Hash Table states within the Alias Table structure
used for global aliasing with a single open wrap at a time. .......... 72
4.5 SoftWrAP Library Architecture ................................. 75
4.6 Program Simulator with SCM and SoftWrAP ..................... 79
4.7 Average time for an ntstore for different group sizes. Write Buffer
size: 8 cache-line entries, Cache Line size: 16 words, SCM write
delay: $T_w=1\mu s$. .................................................. 85
4.8 Average number of pcommits and ntstores per transaction of sizes $n$
=10, 15, and 20 after performing 100,000 transactions. ............ 89
4.9 Response time and throughput for varying arrival rates of transaction
size $n=10$, alias table size of max 8k entries, and SCM $T_w=1\mu s$. .... 91
4.10 Maximum Throughput for various Alias Table sizes. Arrival rate = 50k wraps per sec, n=10, \( T_w=1\mu s \) ........................................ 92
4.11 Response Time and alias table entry size over time. Arrival rate = 50k wraps per sec, n=10, Alias Table: 8k entries, \( T_w=1\mu s \). .............. 94
4.12 Response time using a thread-local alias table for copying to home locations and for only to cache over time with arrival rate of 50k txps of size n=10 and SCM \( T_w=1\mu s \) ........................................... 96
4.13 Response time for various SCM \( T_w \) times with arrival rate of 50k txps of size n=10 and alias table size of max 8k entries. ..................... 97
4.14 Response time for various transaction sizes with arrival rate of 1,000 per second, alias table size of max 8k entries, and SCM \( T_w=1\mu s \). .......... 98
4.15 Maximum throughput for various percentage of data reuse across transactions with size n=10, alias table size of max 8k entries, and SCM \( T_w=1\mu s \) ......................................................... 100
4.16 Average insert throughput of random elements into a B+Tree with alias table size of max 4k entries and SCM \( T_w=1\mu s \) ..................... 102
4.17 Average insertion time of a single integer element (sequential or random) into a B+Tree with alias table size of max 4k entries and SCM \( T_w=1\mu s \) ......................................................... 103
4.18 SQLite Pluggable Virtual File System Implementations using SCM for A) block writes and B) SoftWrAP .................................................. 105
4.19 Throughput in Transactions Per Second for the TPC-C Benchmark with SQLite ................................................................. 107

5.1 SoftWrAP alias table with fine-grained locking ........................................... 112
## Tables

4.1 Subset of the SoftWrAP Application Program Interface .................. 78
4.2 Time to perform a wrap of \( n \) 4-byte words. .......................... 83
4.3 Performance of Array Update Benchmark. \( T_w = 1\mu s \) and \( T_s = 220ns. \)
   \( T_{alias} \) calculated is 450ns. ...................................... 88
4.4 Average number of reads and writes required along with the average
   time needed to perform an insert into a B+Tree for various
   persistence methods. .................................................. 104
Algorithms

1 Example Financial Transaction Routines ........................................ 8

2 Transactional Routines for Linked List Manipulation ....................... 25

3 Persistent atomicity using an Undo Log ...................................... 32

4 Programmer annotated atomic region ........................................ 62

5 SoftWrAP API - wrapOpen, wrapLoad, and wrapStore ....................... 66

6 SoftWrAP API - wrapRead, wrapWrite, wrapClose ........................ 69
Chapter 1

Introduction and Overview

1.1 Introduction

This thesis examines the use of byte-addressable persistent memory (also called Storage Class Memory [1] or SCM) as a replacement for traditional non-volatile storage (hard disks or SSDs) in data intensive applications. A growing number of these applications employ in-memory database technology that operate almost entirely from DRAM (e.g. [2, 3, 4, 5]) for access efficiency and to meet throughput requirements [6]. DRAM-based data access has two main advantages: one is the fast speed of DRAM that enables the high throughputs and real-time response times required of new classes of web applications [6]; second is the fine-grained memory word (or cache-line) accessibility of DRAM, which facilitates applications involving traversing sparse data structures or graph processing. However, the volatile nature of DRAM makes these systems vulnerable to system crashes, often requiring ad-hoc checkpointing techniques to maintain a persistent copy of the data on non-volatile storage. This incurs runtime overheads, and long recovery times following a crash or scheduled maintenance to rebuild in-memory structures.

Emerging Storage Class Memory technologies [1] like Memristors, Spin-Torque MRAM (ST-MRAM), or Phase Change Memory (PCM) raise the possibility of having the best of both worlds: fast, cache-line granularity access of DRAM with the persistence of disk or SSD. As shown in Figure 1.1, SCM is a new class of mem-
ory that is both byte-addressable and non-volatile. This gives rise to a new class of applications, making it possible to use algorithms and data structures designed for byte-addressable volatile memory without either the programming and performance overheads of blocking and unblocking these structures for disk access, or having to deal with the potential loss of data due to a system crash.

Phase Change Memory, or PCM, is a promising SCM technology, as it can achieve a high chip density and speed [8, 7], and it has almost a thousand times greater write endurance over Flash technology. Additionally, it 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) [9]. Due to the byte-addressable and high-speed nature of the technology, PCM is expected to be exposed on the main memory bus and operate alongside DRAM instead of being accessed through a disk or PCI interface [10, 8, 9, 11, 12, 13]. Accessing PCM through a PCI interface would take more time than to access PCM directly from the main memory bus. Figure 1.2 shows how storage class memory can be placed on the main memory bus alongside DRAM.

Figure 1.1: Storage Class Memories are fast, byte-addressable, and persistent. Adapted from [7].

SCM memory can enable systems with large amounts of persistent, byte-addressable
memory that can replace slower, block-based hard disk or flash disk drives.

Figure 1.2: SCM will sit alongside DRAM on the main memory bus.

Since SCM is nonvolatile it raises the issues of consistency faced by all storage systems. However, the ground rules change significantly when using SCM. First, the techniques developed by database and operating systems use complex software intervention for buffering and logging to exploit the slow block storage interface and access latencies [14] of traditional storage devices. Secondly, SCM is attached to the memory subsystem and interacts directly with the processor cache hierarchy, leading to new problems caused by uncontrolled cache evictions. In the ideal situation, a program simply reads or writes locations in SCM just as it reads or writes locations
in volatile memory, with the assurance that its writes into SCM locations are durable and consistent. Durability refers to the property that a store to an SCM location is not left buffered in a volatile tier (such as a processor cache or a memory controller buffer), while consistency refers to the property that a compound update which spans multiple locations in SCM is not incompletely reflected in the SCM after a machine restart. Atomicity refers to an update spanning multiple locations as either all updating or none of the updates occurring. Ensuring durability and consistency is not straightforward for a number of reasons recounted briefly in section 1.2, and in recent years a number of approaches to durable and consistent updates have been proposed. These are summarized and compared with our approach in Chapter 2.

This thesis examines the problem of providing durability and atomicity in a system using SCM for persistence, and describes a novel solution in both a software-hardware architecture and a software-only library. It describes and evaluates WrAP (Write-Aside Persistence), a software-hardware architecture which is based on propagating updates simultaneously through the fast cache hierarchy as well as along a slower asynchronous channel to SCM. The solution avoids changes to the front-end of the well-understood cache hierarchy, and leverages the SCM controller to obtain the desired behavior in a non-intrusive manner. Additionally, this thesis also presents a software-based approach for SCM atomic persistence based on a software library, which we call SoftWrAP (Software-based Write Aside Persistence). The SoftWrAP approach requires no hardware changes, beyond building upon the expectation that store-fencing capabilities of current machines will be extended naturally to fencing writes to SCM. In fact, Intel recently introduced the PCOMMIT instruction [15] that does just this – the instruction retires when pending stores to SCM locations have drained out of intermediate buffers. The SoftWrAP implementation is designed to
be portable, and its performance is amenable to tuning by compiler optimizations. The approach specifically allows an application to benefit from high speed processor caches while simultaneously achieving consistent updates into SCM. We show that both methods outperform other methods that perform atomic updates to SCM and can approach the speed of direct writes to SCM that do not guarantee atomicity at all.

1.2 Problem Overview

The problem addressed in this work is that of ensuring a sequence of stores (writes) to arbitrary addresses of persistent memory is performed atomically, even in the presence of machine failure. A good solution must exploit the processor cache hierarchy to communicate updates within and across transactions.

There are two complementary problems that arise in trying to achieve atomic writes in the presence of failures. The first, which has a direct analog in disk-based transactions, is to ensure that writes made by a program have actually been committed to the non-volatile medium. In the SCM scenario the problem arises because the default memory store semantics supported by processors make no guarantees of when the target location of a store instruction will actually be reflected in the back-end device (typically DRAM). Even memory fence instructions (like SFENCE) only guarantee that the store has been made visible to other processors at the barrier, but do not guarantee that memory has been updated. So also, cache flush instructions only guarantee eventual update of the backing memory, which will typically be done asynchronously from volatile store buffers. In all of these situations a machine restart implies that an update is lost since there is no persistent copy of the data.

The second problem that arises is that the cache subsystem evicts cache lines and
stores their values to the backing memory autonomously based on its specific cache management policies. There are no guarantees regarding the order in which these evictions may occur; an arbitrary subset of a sequence of atomic stores may have been written to the memory at the time of a machine crash, while the remainder are held back in volatile cache and memory buffers. Figure 1.3 shows a transactional sequence of stores where some values get evicted out of the cache hierarchy to persistent SCM before the end of the transaction.

![Diagram of cache hierarchy and persistent memory](image)

**Figure 1.3**: Uncontrolled cache evictions can result in arbitrary mixes of new and old values in SCM.

Even if a programmer were to flush cache lines synchronously and use persistent fence instructions, like SFENCE followed by PCOMMIT, to ensure ordering, it does not guarantee atomicity of the write sequence; this would require resuming execution
of the transaction precisely from the point of failure necessitating all volatile state to have been checkpointed as well [16]. To achieve resilient operation, a programmer must consider employing additional metadata (e.g., a journal) also in persistent memory, to track which sequences completed and which did not, and use that metadata for effecting recovery. Accessing data safely from multiple programs while maintaining durable data structures becomes a significant software engineering challenge. Conventional systems today don’t face a severe coordination problem since blocks on a durable medium are accessed indirectly through a file system or a database storage manager, which effectively coordinates the sequencing of updates and manages recovery from torn updates.

Some proposals to change the cache hierarchy to enforce specified cache eviction orderings have been advocated [11, 12]. Controlled ordering of cache evictions reduces, but does not eliminate, undesirable execution scenarios. They also involve a significant change to existing well-understood processor cache hierarchies.

1.3 Example Transactions

Consider a simple, single-threaded software program that might perform updates to account balances such as transferring money between two accounts or accruing monthly interest. Algorithm 1 shows two separate in-memory account update operations for transfer and interest calculation.
Algorithm 1: Example Financial Transaction Routines

```
transferMoney ()
begin
    Trans_Begin
    balanceAccountA += money;
    balanceAccountB -= money;
    Trans_Commit

addMonthlyInterest ()
begin
    Trans_Begin
    for int i=0; i < size; i++ do
        balance[i] = (1 + rate/12);
    Trans_Commit
```

The group of operations contained between the `Trans_Begin` and `Trans_Commit` comments is called a transaction. If the simple program was running in traditional volatile memory and did not need to persist any data, then there is no problem. However, if the program needs to persist account balances, it is crucial that the data is consistent in case of system failure. Instead of saving the account data to a back-end, block-based disk store, suppose the program maintains the account balances directly in SCM. The account balances might appear to be persistent in memory, but several things might happen.

First, in the case of a balance transfer, if the system were to fail after adding money to account A but before subtracting from account B, then the overall consistency in the amount of money being tracked by the program is not preserved. The transfer of money between the two accounts needs to be atomic, either the money is transferred
or it is not. The same problem exists when updating a large number of accounts in the interest calculation example. Next, even if the system does not fail and the program finishes an update and reaches the commit comment, some or all of the new account balances might be stuck in the cache and not written to SCM. The state of the data should be consistent and durable, and the transaction should not be lost. Finally, if the program were to flush all the updated balances from the cache to SCM after completing the transaction, a system failure could happen during this flush making the system inconsistent, e.g. the group of balance updates would no longer be atomic, since only some memory locations might have been updated.

The program might attempt to implement some type of logging mechanism to preserve the integrity of the data. As in a balance transfer above, the intermediate operations between the Trans_Begin and Trans_Commit sections must be performed in an atomic manner. A copy of the data could be updated or the old data could be copied and saved in case of a system failure. However, copying data requires additional overhead. Additionally, if the program is multi-threaded, then there are other concerns such as isolating transactions that havent completed from other operations, e.g. if balance queries on accounts A and B were being performed during the middle of a balance transfer, the amount of money reported might be incorrect. Databases management systems provide transactional guarantees by implementing locking as well as logs, and this relationship is described in the Section 1.3.1.

Figure 1.4 depicts several ways in which a transaction containing a group of stores can be written to memory. A transaction start is denoted by Start, several writes are shown as small lines, a commit is denoted by Commit, and the transaction ends and is present in persistent memory at time End. Figure 1.4(a) shows a typical transaction that suspends all writes until a Commit. This method incurs a large
Figure 1.4: Various methods where groups of stores may be written to main memory.
number of write operations all at once, hence a Write Storm, after a commit is issued at time Commit. The large number of writes all close together in time to persistent memory can fill up write buffers, especially when the write delays are long in the case of SCM, and this can significantly affect performance. It is important to note that this large delay could be avoided if the writes were performed asynchronously. An asynchronous Write Storm could be performed by a separate thread of execution that had access to the list of variables in the update group. However, the Write Storm must be atomic to persist in SCM as described above or the state of the system will still be inconsistent. This update method will be addressed in our approach to atomic persistence.

Figure 1.4(b) shows an Undo Log approach that requires copying of old values and placing them in a persistent memory based log structure before a write is made. The old values must first be read from persistent memory before being written to a log entry. This synchronous copy operation also incurs extra long delays, as each log entry must also be flushed to persistent memory to preserve ordering and consistency. Only after the log entry is safely in persistent memory may the new value be written. New values must also be made sure to have been pushed all the way to persistent memory through any cache hierarchy before the log entries for a transaction may be removed.

A more efficient approach is shown in Figure 1.4(c) where a transaction lets writes proceed asynchronously without stalling, and at the end of the transaction time Commit, only the remaining outstanding memory writes are flushed to persistent memory. This approach uses background writes to allow operations in a thread to proceed while avoiding the synchronous delay of waiting. This is not an easy task as the asynchronous writes must all be atomic to ensure the consistency of the
data. The method to make groups of write operations to persistent memory atomic in this asynchronous manner is the main focus of this thesis. This method is introduced as a WrAP, or write-aside-persistence, and is described and evaluated in both a architecture approach and software only approach.

1.3.1 Relation to Database Systems and Transactional Memory

The fundamental problem addressed is that of obtaining atomic updates of an arbitrary sequence of writes to byte-addressable persistent memory in the presence of arbitrary hardware or software induced machine restarts. This issue lies at the boundary between two well-studied and still-evolving fields: database systems and transactional memory. Transaction processing database systems provide a superset of the failure atomicity and persistence problems considered here: that is, they guarantee all four ACID properties for database transactions.

The well-known ACID properties are: Atomicity (a transaction behaves as if it has executed completely or not at all), Consistency (global invariants of the system state are preserved in the presence of updates), Isolation (a transaction may not be affected by other transactions executing concurrently), and Durability (the updates made by a committed transaction are permanent on non-volatile media) [17]. The issues faced by supporting these properties are tied together by the DBMS into sophisticated software-managed solutions [14] that are tailored for slow secondary storage devices like disks.

In contrast to DBMS, transactional memory systems deal with main memory resident data, but are only concerned with the issues of atomicity and isolation (requirements A and I). Transactional memory systems do not deal with the durability (D) requirement, i.e. guaranteeing persistence in the presence of failures. If there
is a machine restart, all state can be legitimately discarded. If a transaction aborts then the isolation mechanisms that prevent partial views from becoming visible are sufficient to allow rollback.

Providing failure atomicity for SCM straddles these two domains: it deals with devices accessed like main memory at speeds orders-of-magnitude faster than disk; yet, it needs to handle arbitrary system failure and transaction rollbacks while maintaining the consistency of persistent storage (requirements A and D). Additionally, the fast speed of SCM precludes expensive software intervention that could handle the failure atomicity. Recently proposed solutions like Mnemosyne [13] advocate obtaining atomic persistence within a software transactional memory (STM) system. The Mnemosyne approach ties together atomicity considerations for concurrency control (Atomicity and Isolation) with the orthogonal issue of persistence atomicity (Atomicity and Durability). Coupling these two issues shoehorns application developers into using a single concurrency control mechanism just to obtain persistence. It is often far more efficient to implement concurrency control for a long-lived database transaction using fine-grained locking rather than treating it as a monolithic STM transaction.

So also, high end file systems may explicitly guarantee A, C, and D characteristics at the storage interface, by protecting against torn writes and by preventing overlapped writes and reads to blocks of data, but the freedom and responsibility for properly ordering reads and writes is left to the application developer.

Our approach is to keep the two issues independent, so that application developers can blend any concurrency control policies and methods they choose, with means for achieving atomic and durable updates to an SCM device.
1.4 Contributions

Guaranteeing transactional execution in persisting data to durable storage while exploiting the cache hierarchy is not straightforward.

In traditional computer systems there is a sharp distinction between accesses made to volatile and persistent storage. Applications can directly access the former using regular memory instructions (loads and stores), while accesses to durable storage are arbitrated by the file system or a database system. The arbitration allows the system to ensure that the ACID (atomicity, consistency, isolation and durability) requirements of transactions are met, by coordinating the component and concurrent activities. In addition, the interposed access provides protection of the underlying data from spurious writes by malicious or erroneous processes.

Three issues that need to be addressed for correctly implementing persistence include **Ordering**, **Atomicity**, and **Protection**. The techniques needed to handle these efficiently are different when the non-volatile store is a memory-bus-based SCM device that is accessed using load and store instructions, compared to traditional disk-based stores. In disk-based stores, all accesses to disk are arbitrated by intervening system software and performed in large blocks that may contain additional information about the underlying transaction instead of just a single store of data alone.

- **Persistence Ordering.** Updating persistent data structures imposes additional constraints on the *ordering* of statements, due to the possibility of failure at arbitrary points in the program. For instance, setting a persistent pointer variable to the address of an uninitialized block of storage can result in an undetectable error if the system crashes between the pointer update and the initialization of the block. However, switching the order of updates, so that the
block is initialized before the pointer, maintains consistency even after a failure-induced reboot. Note that persistence ordering requires that the updates must be propagated all the way to the persistent memory in the specified order. It is not sufficient to just order the global visibility of these updates as in typical memory consistency protocols. Additional hardware support may be needed to ensure this memory behavior, as discussed in Chapter 2.

- **Persistence Atomicity.** Transactional semantics require that updates to a set of related records must always occur as a group; either all the records must be updated or none of them should. Examples in Section 1.3 include the updates to account balances when transferring funds between bank accounts. Additional examples include maintaining dynamic data structures requiring coupled changes to sets of pointers. Since failure may occur at any time, the system must have some way of backing out from a partial set of updates, or must defer the updates until all values have been safely recorded in a fail-safe region. Traditional software systems perform transactional updates by making system calls to an underlying file system or database, which uses disk-based record logging or copy-on-write-based mechanisms to ensure that the updates are applied indivisibly, and are always recoverable once the transaction has committed.

- **Persistence Protection.** Programming without bugs in a persistent memory system can be extremely challenging. Not only does the persistent nature of the changes make it impossible to simply reboot to a consistent memory state, but subtle pointer dependencies between data structures spread over volatile and non-volatile memory regions of memory increase the challenge of robust programming tremendously [18].
This thesis is directed towards solving these issues, mainly providing atomicity guarantees to groups of writes to persistent memory. It describes and evaluates WrAP (Write-Aside Persistence), a software-hardware architecture which is based on propagating updates simultaneously through the fast cache hierarchy as well as along a slower asynchronous channel to SCM. It also presents a new software-only approach called SoftWrAP (Software-based Write-Aside Persistence) that requires no hardware additions whatsoever. This thesis shows that both methods outperform other methods that perform atomic updates to SCM, and the methods also approach the speed of direct writes to SCM that do not guarantee atomicity.

1.5 Thesis Organization

The remainder of this thesis is organized as follows. In Chapter 2, the problem background and related work solutions to the problem are discussed in detail. The state of Storage Class Memory and new Instruction Set Architecture (ISA) is discussed before presenting a more detailed example that builds on the examples in Section 1.3. Related work and various approaches to solving the problems presented are discussed in detail.

In Chapter 3, a software-hardware architecture solution, called WrAP, or Write-Aside Persistence, is discussed and evaluated. This solution is a technique for preventing spurious cache evictions from occurring before the end of a transaction. A simulation based evaluation of the architecture is also presented in Section 3.2.

Next, in Chapter 4, a software only based solution, called SoftWrAP, or Software-based Write-Aside Persistence, is presented that relies on no additional hardware changes. The solution is presented as a software library along with an evaluation of the software, which is compared to several other techniques. The evaluation is
performed using software emulation of storage class memory running in executing applications.

Finally, in Chapter 5, conclusions from both methods are presented. A detailed discussion on continuing and future work is also presented.
Chapter 2

Background and Related Work

In this Chapter, I describe the background and related work in the area of atomic persistence mechanisms for Storage Class Memory. I first briefly describe the rapidly changing area of SCM technology, and a background for implementations to support atomic persistence in SCM follows. I describe instructions that are introduced as additions to the Instruction Set Architecture from Intel Corporation for supporting persistent writes to SCM. Next, I expand with an additional transaction example to the financial transaction example from Section 1.3 and describe various approaches that might be used to achieve atomic persistence. These approaches are then described along with research in the related work currently in development for atomic persistence for SCM.

2.1 Storage Class Memory

Persistent, byte addressable memory has been gaining popularity recently. While not a new idea, battery backed DRAMs have also gained popularity recently in the area for caching persistent data for larger back end databases. These Non-Volatile DIMMS, or NVDIMMs, are attached to the main memory bus and have batteries to persist data in case of machine failure. However, having to constantly refresh DRAM lines with battery power, only allows for data to be held for up to several hours or days. With Storage Class Memory, byte addressable memory on the main memory
bus is persistent similar to disk or SSD.

Phase Change Memory, or PCM, is a promising SCM technology, as it can achieve a high chip density and speed [8, 7]. It is made from a type of glass that can be switched between two phases by the application of heat using electric pulses. Its two phases, crystalline and amorphous, have different resistances, which can be used to represent a binary 1 or 0 [7, 19, 20]. PCM cannot withstand as many write operations as DRAM, only being able to endure about $10^8$ writes as compared to $10^{18}$ writes for DRAM [9]. However, it has almost a thousand times greater write endurance over Flash technology. Research exists to reduce the number of writes by wear-leveling techniques such as lazy writes [7], managed data structures [9, 11], and NVM allocators [10, 12]. PCM buffer reorganizing [19], write cancellation and adaptive write cancellation [20], and techniques to pre-set bits prior to incurring a long latency write [21] have all been shown to improve the overall latency for PCM.

Early Phase Change Memory modules shipped by Micron Corporation have been recently put on hold while 3D-NAND technology has been favored. The PCM modules were put on hold since it was used in a market exclusively for certain mobile phones that collapsed. However, several research groups are still actively researching Storage Class Memory at Micron. With a recent partnership between Micron and Sony, the pair is working on a resistive random access memory (ReRAM) [22]. This ReRAM is 27nm technology with a 16Gb data capacity, supporting up to 200MB/s writes.

Everspin Technologies, is a promising venture specializing in Spin Torque Magnetoresistive Random Access Memory (ST-MRAM) and is planning to scale up to gigabit density and higher speeds [23]. Everspin has announced a DDR3 module utilizing banks of its ST-MRAM chips, but it is not yet publicly available. The speeds are much faster than the ReRAM announced by Sony and Micron, with the
ST-MRAM executing 1,600 million transactions per second. However, the density is not as high as the announced ReRAM, and the DDR3 DIMM is only 64Mb.

Storage Class Memory technologies are rapidly changing. Favored Phase Change Memory, PCM, is possibly being replaced with an alternative technology, either ST-MRAM or ReRAM. The hardware and software solutions presented in this thesis are agnostic to the type of Storage Class Memory, as many of the problems of atomicity in byte addressable, persistent memory are the same regardless of the underlying persistence medium.

2.2 Memory Architecture

Caching plays a pivotal role in the performance of persistent in-memory data structures [24]. The introduction to the SCM architecture in Chapter 1 is extended to show additional detail. Figure 2.1 shows a memory write and write-combining buffer along with a non-temporal or streaming store path to main memory that bypasses the cache hierarchy. This path will be shown to be crucial to high performance software and hardware architectures needing to persist data in an atomic manner.

Non-temporal or streaming stores are stores to memory locations that bypass the cache hierarchy without polluting caches, but will update the cache if the memory location is present. These store intrinsics are in the family of `mm_stream` instructions [25], primarily created for multi-media applications to stream data that need not be cached. Additionally, data written in this manner can take use of the Intel Write-Combining (WC) Buffer to reduce the memory bandwidth [25]. The streaming store path is useful for persistence as it forces writes to main memory. Without this path, writes must be performed into the cache hierarchy and then may be forced out of the cache through flush commands at a later time.
The recent Intel Architecture Instruction Set Extensions Programming Reference [15] describes several new features to support persistent or Non-Volatile Memory (NVM) technologies such as Storage Class Memory or SCM. Some instructions are noted to have been optimized, such as non-temporal stores that bypass caches and place data directly in write-buffers. New features have been added to the processor Instruction Set Architecture (ISA), most notably:

• **Atomic Stores.** The write atomicity, or guaranteed write granularity, of a store to SCM is a 64-bit access. The write must be contained within a single cache-line but may be located anywhere within the line.

• **Cache Flushing.** An older CLFLUSH operation was useful to force data
from the processor cache into store buffers to main system memory. However, this instruction also invalidated the cache-line which made the instruction not attractive when using it to force cached entries to a persistent memory location. Intel is introducing two new instructions for optimized cache flushing: CLFLUSHOPT and CLWB. CLFLUSHOPT is an optimized version of CLFLUSH, useful in flushing multiple cache-lines to persistent memory. CLWB, or Cache Line Write Back, is useful for keeping data in the cache, but forcing it into store buffers for persistent memory writes. Care must be taken with a store fence to make sure a variable has been written to the cache before flushing or the variable might be in a store buffer and not visible. Flushed values will need to be committed to main persistent memory locations with the following new instruction.

- **Persistent Commit.** PCOMMIT is a new persistent commit instruction introduced by Intel. It commits data that might be queued in write buffers to persistent memory locations, ensuring that all writes of globally visible data from non-temporal stores, cache evictions, and cache flushing is made persistent and written to memory. This instruction is very expensive, as it waits until persistent memory ranges in all write buffers are flushed to persistent memory before becoming globally visible. Additionally, proper fencing must be used as PCOMMIT only guarantees writing visible stores to memory.

The above instructions must be correctly ordered with proper fencing to ensure that stores written before or after the instruction are performed in the order desired. For instance, writing to a variable X and then performing a CLFLUSH and a subsequent PCOMMIT does not necessarily ensure that the value of X has been safely
copied to main memory. Therefore, a store fence, or SFENCE, is required after the
write to X to ensure that there is a proper ordering and the store has been performed.

Additions to the Intel Instruction Set Architecture were a needed foundation by
much of the related work outlined in Section 2.3 and Section 2.4. These instructions
allow for a new path of fast atomic persistence which is presented in this thesis.

2.3 Atomic Persistence

In our model, persistent objects are stored in non-volatile memory (SCM) and man-
aged by an object-management layer that handles the naming and space allocation
requirements. Applications use familiar `mmap` primitives to map the object into
their virtual address space. Subsequently, memory load and store instructions are
used to access the object for reading and writing the objects; these accesses are in-
tercepted by the cache controller and moved to and from the cache hierarchy just as
accesses to regular DRAM.

Figure 2.2 shows two objects A and B in persistent memory that are mmaped
to regions of a processes address space. A cache line from object A is shown within
the cache hierarchy from the virtual address space to the persistent memory ad-
dress. Reads and writes to these virtual addresses are simply handled as normal
loads and stores to the physical addresses corresponding to the persistent memory
devices. Cache misses and cache evictions directed to these physical addresses are
handled by the SCM controller, analogous to a traditional DRAM memory controller.

Within a transactional section of code, a sequence of store operations to persistent
memory must be performed atomically even if the write sequence is interrupted by a
machine restart. Following a restart, the state of persistent memory should reflect all
or none of the updates. A good solution must exploit memory reuse by utilizing the
processor cache hierarchy to transmit values of a variable both within a transaction as well as between transactions.

Atomic persistence is illustrated with another simple example shown in Algorithm 2. The routine `moveNode` transfers a node between singly-linked lists `freeList` and `workList`. If the store of step 3 has reached persistent memory and the store from step 2 has not reached memory, if the machine fails, then the entire `workList` is unreachable and lost. If only the store of step 2 makes it to persistent memory before a failure, then most of the `freeList` is lost since the nodes of `freeList` beyond the first are unreachable. Implementation of simple data structures becomes a complicated programming challenge, and redundancy needs to be built in to account for different store ordering scenarios.
Algorithm 2: Transactional Routines for Linked List Manipulation

addNode(node *list, node *newNode)
begin
    Trans_Begin
    1. newNode -> next = list;
    2. list = newNode;
    Trans_Commit;

moveNode(node *freeList, node *workList)
begin
    Trans_Begin
    1. temp = freeList -> next;
    2. freeList -> next = workList;
    3. workList = freeList;
    4. freeList = temp;
    Trans_Commit

When the data structure is implemented in non-volatile memory, the data loss can be catastrophic since the memory copy is the durable record of all the changes. The problem is compounded when the updates are made directly using regular store instructions, since it unpredictable which of the completed store instructions have actually been written to persistent memory and which are still within the cache hierarchy or pending in a store buffer.

The following sections describe several methods to achieve persistence.
2.3.1 Approach 1: Non-Atomic with Persistence Instructions

In the simplest scenario, suppose a machine failure occurs after a transaction commit. The variables updated by the transaction may not have been written to persistent memory, which would be catastrophic. Before the transaction commits, the updated variables must be explicitly flushed from the cache. A memory fence is needed to ensure that the flushed values are not pending in store buffers. Memory fence instructions only guarantee that the pending stores are made globally visible before execution proceeds (for providing consistent ordering [26]) but not necessarily written to the backend memory. Therefore, after a memory fence, a PCOMMIT instruction is needed to perform a commit of all globally visible values in write buffers to persistent memory. As an alternative to using flushing instructions, such as CLFLUSH, CLFLUSHOPT, and CLWB, after a store to a variable, the stores may be streamed using the non-temporal or streaming stores as previously mentioned.

While persistent fences and cache flushes can ensure data is written to SCM, they are insufficient to handle more complicated scenarios. At the time a failure occurs, it is generally unknown which of the preceding writes have made it to persistent memory and which have not. Therefore, this method achieves persistence, but is non-atomic in the sense that some of the stores may be made to persistent memory and not just all or none of the stores.

2.3.2 Approach 2: Cache Ordering Primitives

Figure 1.3 showed an example atomic region that includes writes to four persistent variables in the order: A, B, C and D. Suppose that the system crashes immediately after the store to C. At that point the updates to A, B and C have been recorded somewhere within the cache hierarchy, but may or may not have reached persistent
memory. A possible execution sequence is where a cache line corresponding to B has been evicted to persistent memory due to normal cache management operations, while A and C are still present only in the cache. This is the complementary problem to a persistent update not having been evicted to persistent memory at the time of the crash.

In the example of the addNode transaction from Algorithm 2, it is possible for store 2 to be evicted from cache into persistent memory before store 1 updates persistent memory. A crash would result in the loss of the original list along with a dangerous dangling pointer from the newly inserted node.

Proposals to change the cache hierarchy to enforce ordering of cache evictions have been proposed [11, 12]. While differing in specifics, the broad idea is to tag cache lines with a sequence number indicating the write order, and ensure that cache evictions do not violate this prescribed order. Such an ordering mechanism would solve the addNode problem since updating the head of the list would not become persistent until the new node was linked in. While the proposals for such ordering primitive are interesting, they involve a significant change to existing well-understood processor cache hierarchies. It is unlikely to be easily adopted unless overwhelming advantages can be shown, and the impact of such a disruptive change on existing software is evaluated.

Now consider the slightly more involved transaction shown in moveNode, where a freeList node is removed from the start of the list and added to the start of the workList. Suppose the failure occurred just preceding step 4. If the store of step 3 has reached persistent memory while 2 has not, then the entire second list(workList), originally passed as an argument, is unreachable and lost. If the store of step 2 makes it to persistent memory, then most of the first list is lost since the nodes of freeList
beyond the first are unreachable. Implementation of simple data structures becomes a complicated programming challenge, and redundancy needs to be built in to account for different store ordering scenarios.

Controlled ordering of cache evictions reduces, but does not eliminate, undesirable execution scenarios. In the moveNode example, ensuring that the update of step 3 does not occur unless the update of step 2 is also completed will ensure that the first scenario, in which workList is lost, does not occur, but does not prevent the loss of most of freeList.

Similarly, in the interest calculation example from Algorithm 1 in Section 1.3, should the update loop be interrupted by a machine restart, it is impossible to know which of the account values in persistent memory represent updated values, even if program-visible state (like program counter and the loop index variable) had been saved at the time of interruption.

With enforced orderings on cache evictions, the uncertainty is reduced to tracking the index of the last value evicted from cache, but still requires additional mechanisms to implement properly.

2.3.3 Approach 3: Block Copy with Atomic Pointer Updates

Using Copy-On-Write mechanisms [11] or timestamp-based multiversioning [12] to maintain before and after versions of updated persistent memory has been proposed. When combined with hardware-supported atomic writes to 8-byte words (sufficient to hold a pointer value), this provides an effective mechanism for atomic updates of an important but restricted class of block-based tree-structured data structures. Small (8-byte or less) updates are made by directly writing the word into the persistent memory location. For larger-sized updates, a copy of the block is made and
the changes applied to the copy; the new block replaces the existing block by atomically switching its parent pointer to the new location. When the updates of the transaction are spread over several distinct blocks, then all the nodes on the paths to the least-common-ancestor of the affected blocks are copied, their relevant portions updated, and the pointer to the new ancestor node is written into its parent with an atomic pointer write. The approach is most useful when updates are isolated to single blocks and involve updating a significant portion of the block. However, transactions involving writes to scattered addresses incur large copying overheads.

2.3.4 Approach 4: Transaction Memory

A different approach to providing atomicity in persistent memory is the Mnemosyne system [13], based on software transactional memory (STM). Essentially all transactions are handled by an STM system that intercepts all the writes and reads within a transaction. Internal copying and logging is used to provide both concurrency control and atomicity of transactions. The approach has the advantage of using a single framework for meeting ACID requirements, and can potentially leverage advances in STM technology to improve performance. On the other hand it forces the application developer to a single concurrency control model (TM), and is difficult to fit to legacy software applications, which support different isolation models or employ lock-based concurrency control. Coupling the very distinct concerns of providing atomicity to a group of operations (like the coupled pointer switches to a data structure) and controlling concurrent accesses to the data structure is not desirable, as applications demand more and more autonomy in their implementation (for instance the noSQL approaches to big data and databases like voltDB [4] that employ statically serialized transaction sequences to reduce concurrency control overheads).
2.3.5 Approach 5: Undo Logging

A classic method of providing ACID guarantees in transaction management systems is the use of a software-arbitrated Undo Log. In this approach, before an update is applied to an object, a copy of its current value is made and saved in non-volatile storage. This approach is sometimes referred to as Copy-On-Write, since a copy is made before a writing a value to a variable. In case the transaction aborts or there is a system failure before all the updated values are committed to durable storage, then the system is rolled back to the instant before the start of the transaction using the original values in the Undo Log.

![Figure 2.3: Example Undo Log used to write a variable.](image)

An Undo Log based approach has the advantage that once a copy of the original has been saved to non-volatile memory, the update and any subsequent reads can be done directly on the object. Figure 2.3 shows an Undo Log approach for a persistent variable with virtual address \(a\) and physical persistent address \(\phi(a)\), with initial value 0. When a value 5 is written to \(a\), the Undo Log saves a record \((\phi(a), 0)\) in the Undo
Log area of persistent memory before updating the cached value to 5. Alternatively, the value may be streamed directly to home locations. In case of a crash, the Undo Log can be replayed. At the end of a transaction, all of the values can be flushed from the cache hierarchy followed by a persistent commit, PCOMMIT. Only then may the log be deleted followed by a final persistent commit.

Algorithm 3 describes an implementation of an Undo Log. A persistent store within a transaction invokes Trans_Store. The routine first reads the current value of the variable and appends its value along with its persistent memory address to an undo log. The persistent memory address corresponding to the virtual address $\text{addr}$ of the mmapped object is denoted by $\phi(\text{addr})$. The log record needs to be committed to persistent memory before the update of the new value can be allowed to proceed. This is shown by the PCOMMIT call that is used to force all pending stores (and loads) to complete before execution continues. While similar in spirit to the common memory fence instruction provided for memory synchronization in modern processors, a PCOMMIT following an store fence provides a persistence guarantee; that is, all pending stores will have been committed to memory and not merely made visible to other processors using the coherence mechanism.

Note there is no benefit to caching the log record since it is simply written once. Instead, we use streaming non-temporal store instructions for this purpose. These instructions bypass the cache and, whenever possible, employ write combining, which can be used to optimize the sequential write usage of the undo log. The PCOMMIT instruction following the Append is necessary to force the log contents in the write buffer to persistent memory. Finally, the new value is written to the memory address in the cache in write through mode to allow the update to asynchronously trickle to persistent memory. Alternatively, like writing to the log, the new value can be
Algorithm 3: Persistent atomicity using an Undo Log

\[\text{Trans\_Store} \ (\text{TransactionToken } \text{token}, \text{ address } \text{addr}, \text{ value } \text{newVal})\]
begin
    currentVal = *addr; —— Read current value of variable
    Append (φ(addr), currentVal) to UndoLog[w];
    SFENCE;
    PCOMMIT; —— Commit log record to persistent memory
    writeThroughStore(*addr, newVal); —— alt. streaming/non-temporal store

\[\text{Trans\_Commit} \ (\text{TransactionToken } \text{token})\]
begin
    Remove log entries associated to \text{token};
    —— Commit all updated values to persistent memory
    SFENCE;
    PCOMMIT;

written using a streaming or non-temporal store, which will update any associated cache-lines. When the transaction is closed and committed as in Trans\_Commit, the program must ensure that all the updated values have reached persistent memory, which is also accomplished using a PCOMMIT instruction. In the absence of a write through mode or using streaming, non-temporal stores, the updated values must be explicitly flushed from the cache and then written back before the transaction can close.

As noted earlier, the virtual addresses generated by the program need to be mapped to their physical persistent memory addresses and recorded in the undo log; otherwise their location can be lost along with the page tables in a system crash. Using base and offset addressing to access persistent objects permits a simple imple-
mentation: only the mapping $\phi$ of the base address needs to be known, and frequent operating system invocation can be avoided.

The second point to be noted is the performance impact due to many synchronous memory operations. Before any persistent variable can be updated, the variable needs to be read and a log record with the old value must be $synchronously$ written to persistent memory. Also, the new value needs to be written to persistent memory before the transaction is committed. If the updates are cached in write-through mode, the updated values can trickle through to persistent memory in parallel with the transaction execution while retaining their cache accessibility. In a write back cache, the updates will need to be explicitly flushed from cache (using the CLFLUSH instruction for instance) and then persisted to memory using a PCOMMIT. Note that CLFLUSH actually invalidates the corresponding cache line as well, which has the undesirable effect of destroying the fast-path communication of the variable through the cache hierarchy. Alternatively, on supported instruction sets, the new CLWB instruction may be used that will not invalidate cache-lines. Deferring the flushes to the end of the transaction can increase intra-transaction communication efficiency, but this will create a write storm of updated values to persistent memory at the end of the transaction, creating a synchronous stall until they are all written into their persistent memory locations, as mentioned in Chapter 1.3 and depicted in Figure 1.4(b). Since the write-through option is often only an advisory hint and may not be guaranteed based on the processor implementation, software may be forced to use the slower CLFLUSH alternative.
2.4 Related Work

Mechanisms to enforce persistent atomicity in the literature based on software mechanisms that build upon simple hardware support: atomic 8-byte writes, memory fences, and instructions with PSYNC semantics. Analysis of consistency models for persistent memory was considered in [27]. Memory controller designs for persistent memory haven been proposed in [7, 28, 29, 30]. Adding a small DRAM buffer in front of SCM to improve latency and coalesce writes was proposed in [7].

Hardware approaches that specialize processor and cache behaviors to achieve atomicity have been proposed [11, 18, 12, 9, 30, 16]. These approaches change the front-end architecture with additional cache hardware and policies. Copy-On-Write mechanisms (BPFS [11]) and timestamp-based multiversioning [9, 12] to maintain before and after versions of updated persistent memory have been proposed for ordering cache evictions. 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. Research into new data structures such as in NV-heaps [18], which uses logging and copying, provide support for ACID components in software applications using SCM. CDDS [9] provides a versioning method that copies data and uses sequences of fences and flushes to provide transaction support. BPFS [11] and NV-heaps [18] require changes to the system architecture to support the atomicity and consistency of data. A non-volatile victim cache to provide transactional buffering was proposed in [30], with the added property of not requiring logging at all; by comparison, our approach achieves efficiency through software-based non-temporal write-combining streaming of log records. Unlike [30] which tracks pre- and post-transactional states for cache lines in both volatile and persistent caches and atom-
ically moves them to durable state on transaction commits, our approach does not change the behavior of hardware and instructions and does not require synchronous cacheline writebacks out of processor caches on completions. Whole-system persistence [16] snapshots the entire micro architectural state at the point of a failure to allow for in memory databases, but relies on batteries to power non-persistent memories on system failure. The hardware based WrAP architecture allows an application to handle its own concurrency control such as in RVM [31], but requires hardware changes to the cache backend [29].

Some solutions [13, 32] combine concurrency control with persistence in an integrated framework. Mnemosyne [13] uses software transactional memory (STM) based interception of all writes and reads within a transaction, and uses internal copying and logging to achieve both concurrency control and atomicity. The approach has the advantage of using a single framework for meeting ACID requirements, and can potentially leverage advances in STM technology to improve performance. On the other hand, it forces the application developer to a single concurrency control model (TM), and is difficult to fit to legacy software applications, which support different isolation models or employ lock-based concurrency control. ATLAS [32] uses a compiler pass to automatically generate transactional regions for atomic writes utilizing a synchronous undo log. Coupling the very distinct concerns of providing atomicity to a group of operations (like mutually dependent pointer switches) and controlling concurrent accesses to the data structure is not desirable, as applications demand more and more greater autonomy in their implementation (for instance the noSQL approaches to big data and databases like voltDB [4] that employ statically serialized transaction sequences to reduce concurrency control overheads). Our approach explicitly decouples concurrency control from durable atomicity requirements.
Research in persistent file systems built on SCM is also a promising area that might quickly enable software applications to take advantage of SCM. PMFS [33] is a complete file-system implementation built for SCM. This will be advantageous for legacy file based programs to easily take advantage of SCM, however, new classes of applications can benefit from optimized byte-addressability. SCMFS uses sequences of \texttt{mfence} and \texttt{clflush} operations to perform ordering and flushing of load and store instructions and requires garbage collection [34]. A technique for ensuring atomicity of \texttt{sync} was discussed in [35].

REWIND [36] is a library that also supports transactional writes to main memory similar to an early presentation of Software-based Write-Aside Persistence [37]. However, REWIND uses an in-persistent-memory version of a log structure (which saves DRAM), but log records have to transverse the cache-hierarchy on variable reads and not just log processing.

2.5 Summary

In this chapter, I described recent various Storage Class Memory technologies and new support for atomic persistence to SCM by additions to the Instruction Set Architecture by Intel Corporation. The problem of atomic persistence was presented in detail with several different approaches to persistence, and these were related to current research efforts. The current research in atomic persistence requires synchronous operations or heavyweight software libraries or programming efforts. In the following Chapters, we will show that with both hardware WrAP and SoftWrAP, failure atomicity is lightweight, as reads and writes proceed through the fast cache hierarchy, with logging taking place concurrently in the background with write-combined stores to SCM.
Chapter 3

Software-Hardware WrAP Architecture

SCM technology provides an ideal solution that combines the cache-line access of DRAM with the persistence of disk. This makes it possible to use fine-grained RAM algorithms and data structures, without worrying about either the need for blocking these structures for disk access, or the loss of data due to a system crash. However, when writing data, the integrity of the data structures must be maintained in an atomic manner to avoid corruption and loss of data. In the previous chapters we showed the problem of atomic persistence and attempts at creating transactional updates to SCM that were either heavyweight in terms of software overhead or slow synchronous operations, such as Undo Logging or Copy-On-Write, or required changes to the front end cache hardware.

In this Chapter, we develop on the WrAP concept introduced in [38]. WrAP, or Write-Aside Persistence, a Software-Hardware Architecture, is based on propagating updates simultaneously through the fast cache hierarchy as well as along a slower asynchronous channel to SCM. This approach performs atomic writes to SCM in a more ideal method as shown in Figure 1.4 part (c) as opposed to the synchronous copying shown in part (b) or a method that doesn’t guarantee atomicity like in part (a). The WrAP solution avoids changes to the front-end of the well-understood cache hierarchy and leverages the SCM controller to obtain the desired behavior in a non-intrusive manner. The architecture is described in detail in Section 3.1 with detail on the background operations required and the restart and recovery process. In
Section 3.2.1 the performance model and simulation is discussed. Finally, Section 3.2.2 provides the results of a simulation-based empirical evaluation that showing that WrAP-based atomic persistence is significantly faster than traditional software-based approaches based on Undo Logs.

3.1 WrAP Architecture

In this section, we will develop on the wrap concept introduced in [38] as an abstract conduit through which a thread funnels its writes to persistent memory. A wrap ensures the atomicity and durability of the transaction that it shepherds. However, the wrap is not directly involved in communicating the values of persistent variables between writers and readers. Reads and writes proceed independently through the system cache hierarchy, with minimal interference from wrap operations. The interaction between the cache hierarchy and the wrap occurs only at the back-end at the SCM memory controller. All of these interactions will occur in background or asynchronous mode that are off the critical execution path. This contrasts with most proposals [11, 19, 9, 13, 12] in which cache and logging operations are coordinated at the processor or cache.

A wrap has several different functions:

- Acts as a lightweight firewall that prevents arbitrary writes to persistent memory. Changes to protected areas of persistent memory are only possible through a wrap operation. Like a file system that protects a storage device from arbitrary updates, all changes to persistent memory are orchestrated by a wrap.

- Provides an ordered log of all updates to persistent memory made by transactions, permitting rollback or recovery in case of process or system failures.
• Provides a **non-intrusive interface** for interaction between the cache system and persistent memory while permitting relatively independent operations.

Figure 3.1 shows a high-level view of the WrAP architecture that is responsible for providing various wrap services. It is made up of several components: a **victim persistence cache** (VPC); a Log area of SCM that is used to keep a **log of update** operations; and an asynchronous channel used to propagate log records to persistent memory.

![WrAP Architecture](image)

Figure 3.1 : WrAP Architecture From  [38]

A thread will do a **wrapped write** when it wants to update persistent storage
in an atomic manner*. At the start of an atomic region, the thread opens a wrap and obtains a token, which is used to uniquely identify this wrap. Writes within the atomic region result in two actions: a wrap record is created to log this update (similar to a redo log record) and write it to a reserved area in the Log structure allocated by the wrap. Simultaneously, a normal store instruction to the persistent memory address is issued. At the end of the atomic region the thread closes the wrap.

A log record is a key and value pair, consisting of the memory address that the transaction is updating and the value being written. Log records are write-once records used only for logging purposes. Hence, they are not constrained by memory consistency requirements and do not benefit by caching. In addition, while the underlying writes may be to scattered persistent memory addresses, the log records of an atomic region will all be stored contiguously in a bucket associated with this wrap. This makes them ideal candidates for using the non-cached write-combining modes present in many modern processors (referred to as non-temporal writes [25]). This mode bypasses the cache on stores and uses a write buffer to combine writes in a cache line before flushing the buffer to memory, greatly speeding up sequential writes. When the transaction commits, a single persistent fence operation is needed to make sure that any remaining log records have been written out to the corresponding bucket.

The normal store instruction that is issued concurrently gets written to the cache as usual. This cached item is used to communicate the value of the variable to any reads (loads) made to it. The read may be from the same thread that did the write or from a different thread that is allowed to do so by the isolation policy in effect. Note

*Programmer or compiler support is needed to identify writes that must be wrapped.
that if the value had not been written to the cache, then these reads would involve a slow, software-arbitrated lookup of the Log in order to satisfy the read. While such arbitrations are common when using disk-based logging, they are not compatible with the load and store characteristics of SCM accesses. Writing only to the cache is clearly insufficient and will lead to the Write Storm situation described in Figure 1.4(a). The updated cache lines must be prevented from updating the persistent memory locations until the transaction commits, and all the updated values must then be flushed to persistent memory before the transaction ends.

As noted above, persistent memory locations should not be updated until the transaction commits. The log write does not update the actual memory locations referenced by the thread. However, there is no guarantee that as part of its normal activity, the cache hierarchy will not evict such a cached write to persistent memory before the transaction commits. One approach is to simply mark these updates as clean, so the cache lines are never written back to memory. If the cache reclaims one of these cache lines for a different memory block, it simply overwrites its contents. In this case, future reads to the variable will need to be trapped and the latest value returned from its saved value in the Log.

To address the problem of premature cache evictions, the wrap controller implements and manages a victim persistence cache (VPC) to hold persistent memory entries that are evicted from the last-level cache. The VPC will serve as the backing store for these evicted variables until the SCM controller writes them to their persistent memory locations. Unlike the usual spillage of dirty cache lines which results in updating main memory, these cached values will not be written to their persistent memory locations. Instead, they will be saved in the VPC which acts as a logical extension of the cache hierarchy. The VPC may be implemented in volatile DRAM
memory, since its entries need not be persistent. The maximum size of the of the VPC depends on the number of live persistent variables that overflow the last-level cache. The number of these variables is bounded by the number of distinct variables in currently open wraps, i.e. variables in open transactions that have not yet committed. To keep the VPC from growing too large, items that have been safely committed to persistent memory should be deleted from the VPC. Once deleted, the next read of that variable will result in a cache read miss, and will be serviced in the normal way by reading that variable from its persistent memory location. Deletions of entries from the VPC can be performed as part of the background operation by the wrap controller when it copies values from the log records to their actual memory locations. Different designs of the VPC are possible: these range from pure hardware-controlled solutions to software cache implementations. The latter would be triggered by a cache miss exception that would return the value from the VPC rather than from persistent memory.

The use of write coalescing in a DRAM buffer in front of PCM to reduce the number of writes and improve reliability was used in [7]. In contrast, the proposed VPC stops writes from percolating to the backing SCM, and is used to service memory requests made by the processor.

3.1.1 Background Operations

As mentioned earlier, the actual persistent memory locations referenced by a write operation (called home locations) are not updated immediately. A copy is made in the cache in order to facilitate normal program functioning, and a log record carries the new value to the log bucket associated with the wrap.

A possible organization of the Log is shown in Figure 3.2. When a wrap is opened,
it is allocated a bucket in the Log area. A bucket implements a Key-Value store to hold the log records being written in that atomic region. The figure shows four buckets A, B, C and D. Of these, C and D are buckets belonging to wraps that are currently open. Buckets A and B belong to wraps that have already closed. No new records will be added to a closed wrap. When a wrap closes, it is added to a Log, which is a circular First-In-First-Out queue of closed wraps. Each entry in the Log points to the bucket of a closed wrap. When a wrap closes, its bucket is atomically added to the tail of the Log queue by appending a pointer and incrementing the tail indicator. Methods to implement a robust Log in the presence of failures are presented in [8, 13], and we can easily adapt those ideas for our log structure as well. As discussed below, the entries in the Log are periodically processed and deleted after the associated updates are made persistent. Note that a transaction is allowed to complete only after its bucket has been added to the Log.

The update to the home locations will be made independently by a COPY module that is part of the wrap controller. The module operates as a background task that is periodically invoked to trim the log. It operates on the Log entries in order from the head towards the tail. This module will perform an update operation on one wrap at a time, in the order that they closed. Its task is to update each persistent memory variable referenced in the wrap’s bucket, by copying the new value to the address noted in the log record.

The frequency of invocation of the COPY operation is constrained by the space available in the VPC. If too many items belonging to closed transactions remain in the VPC it may overflow. In our design, these items will be deleted when the COPY module updates the corresponding persistent memory location from the log. We note that overflowing the VPC should be a very rare event with proper sizing. In any case,
it is not a fatal issue since the system can always switch to a degraded mode where items not found in the cache hierarchy are first searched for among the buffered log records before returning the value stored in the SCM. Clearly, this has a significant performance impact, but is provided only as a safe exception path should the rare event actually materialize.

A couple of issues regarding the COPY operation should be noted. First, it is possible that a variable that is being copied from a log record to its home location has since been updated by another transaction. In fact this more recent transaction may still be live or may itself have completed. In either case the VPC entry, which
contains the most recent value for that variable, should not be blindly deleted. A check to make sure that this is the last transaction that wrote to the variable is necessary, and the item should be deleted only if the copying is being performed by the most recent transaction. An alternate strategy is to check that the value of the item being copied as part of the COPY operation is the same as that in the VPC. In this case, the item in the VPC can be safely deleted, even if it is not the last transaction that wrote it. This can happen if two transactions wrote the same value to the variable. In this case, the premature deletion of the entry in VPC is unnecessary, but can cause no harm.

The second issue is that of failure occurring during a COPY operation. A system failure that occurs before all the locations have been updated will leave the log record and its bucket unchanged, so that on reboot the recovery module can redo all the updates in the bucket without problem.

3.1.2 Restart and Recovery

Finally, we discuss the requirements for recovery from system failure, using the **RE-STORE** module. On a system reboot, the RESTORE module is invoked. Its task is to make sure that all pending updates found in the Log are applied to their actual home locations before the restart operation completes. This is a necessary step, since it is possible that a system crash occurred while a COPY operation was in progress that could have left variables of a wrap in an inconsistent state. Since the pending updates recorded in the Log entries collectively leave the system in a consistent state, this is also sufficient to ensure the system is restarted from a correct initial state. The RESTORE module also flushes all entries in the VPC and reinitializes the structure. In fact, since the VPC can be implemented in volatile DRAM, its contents may have
been lost in the system crash anyway.

Note that partially written buckets that were not attached to the Log at the
time of system crash can be safely discarded, since their transactions are treated
as not having completed. Of course, none of the variables that these transactions
wrote have had their home locations updated either. Finally, employing a robust, yet
lightweight, implementation of the Log structure (using any number of torn writes
detection techniques mentioned in the literature) ensures that a failure that occurs
during the update of the Log while a bucket is being added, can be detected by the
RESTORE module.

3.2 Evaluation

In this section we present a performance evaluation of the WrAP approach to persis-
tence. We present the results of a microbenchmark that performs variable numbers of
transactional operations (Insert, Find, Delete) on several data structures implemented
in persistent memory. We compare the performance in three different scenarios. In
the first scenario, the benchmark is applied in non-transactional mode using PCM as
the backing store; in the second, transaction semantics for each operation are enforced
using a traditional implementation based on an Undo Log. The third implementation
uses the WrAP approach. As expected, adding transactional semantics slows down
program execution significantly. However, using WrAP there is a significant perform-
ance improvement over a traditional transactional implementation. In Section 3.2.1
we describe the experimental methodology used; empirical results are presented in
Section 3.2.2.
3.2.1 Performance Model

A benchmark program was created which can perform a variable number of random Insert / Find / Delete operations on a configurable data structure while tagging a group of operations as a WrAP transaction. The data structure can be initialized with any number of elements and the number of operations in a transaction and the number of transactions can be varied. Traces are used to simulate the WrAP architecture. The procedure to generate the traces is outlined in Figure 3.3.

A trace of the memory accesses is obtained using Pin tools [39] to instrument the benchmark program. For the evaluation we need to distinguish accesses to persistent memory from accesses to volatile DRAM. Furthermore, there needs to be a mechanism to recognize transaction boundaries by examining the trace.

We use the C++ Standard Template Library (STL) for the implementations of several data structures that are allocated in persistent memory. To track these persistent addresses, we instrument the STL memory allocator to obtain a trace of the ranges of virtual memory locations it allocates to the calling program. The memory trace generated by the Pin tool is then analyzed offline, and addresses of the allocator and benchmark are reconciled; the persistent memory locations so identified are flagged for the simulation. To identify transaction boundaries, the benchmark program writes to a specific known memory location immediately before beginning a transaction and a different known address immediately before the transaction ends. In addition, a program start location is saved to ignore initializations such as shared library loading captured by Pin. These memory locations are saved to combine with the virtual memory accesses and CPU delays generated by the Pin analysis along with the persistent memory address ranges from the STL memory allocator. The final trace file contains a sequence of memory access records with several fields: the
number of instructions since the previous memory access, memory address, and a set of identifiers indicating whether the address is persistent or volatile, the operation is a read or a write, and whether it is the start or end of a transaction. In this evaluation we only consider non-overlapping transactions.

The memory simulation is performed using a process based simulator Yacsim [40]. The system model consists of a single processor that executes the benchmark program. A single-level, direct-mapped processor cache is modeled, and all reads and writes (write allocate policy) are fielded by the cache. Reads and writes that do not occur within a transaction are handled as normal memory reads and writes, and accessed from the cache. A cache miss takes a variable amount of time depending on whether the access is to a volatile DRAM or to a persistent memory location. Evictions from the cache are either written to the victim cache or are written to their home location (which may be either in DRAM or persistent memory), depending on whether the access belongs to a transaction or not. Access within a transaction (wrapped reads
and writes) are handled by the WrAP protocol; in addition to accessing the cache, a wrapped write operation creates a log record that is pushed to an asynchronous queue to be flushed to the persistent memory log bucket. Cache misses for wrapped reads are served by the victim cache. System parameters (from [19]) normalized to a cache hit time are: DRAM accesses 100 cycles, and persistent memory read are and write accesses are $T_r = 440$ and $T_w = 1200$ cycles respectively.

The persistent memory controller arbitrates between flushes of transactional log records for transactional writes, and servicing transactional reads and non-transactional writes. It gives priority to non-log accesses. The current implementation does not take advantage of the sequential nature of the log writes, whereby an entire cache line could be flushed in a single write operation. It also does not take advantage of modeling store buffers. Modeling these would improve the performance of this evaluation of the WrAP architecture.

### 3.2.2 Experimental Results

The first study of our simulation model was to verify predictions of the expected behavior of the WrAP architecture compared to an Undo Log implementation. A persistent write operation in a cold cache using an Undo Log approach incurs the following delays: a read followed by two writes which must complete in persistent memory to ensure atomicity and consistency. The WrAP approach requires the same read delay, but can place the store in a write back buffer that can drain asynchronously. With a store buffer size of one, the average time to perform a persistent write with $T_c$ CPU time between writes, is given by: $T_U = T_r + T_w + \max(T_c, T_w)$ for an Undo Log implementation and $T_W = T_r + \max(T_c, T_w)$ for a WrAP implementation.

To validate the simulator, we produced a synthetic trace that performs ten WrAP
store operations on sequential cache lines. The traces were then run through the simulation for varying CPU delays from 0 to 10,000 cycles. Figure 3.4 shows the simulation time versus the CPU delay for three scenarios assuming a cold cache. The base case is for an application that simply ignores all transactional semantics and performs writes directly to the cache (called Non-Transactional). This is compared to both WrAP and Undo Log implementations that guarantee failure atomicity. Since the non-transactional approach does no logging (neither an undo or redo log) nor flushing to memory, it has the fastest simulation time. However, as the CPU delay (the time between writes in a transaction) increases, we see that the WrAP architecture performs as well as a system that provides no persistent memory consistency.
Figure 3.5: Comparison of WrAP and Undo Log for a simple benchmark of Speedup vs CPU Delay for hot and cold caches

In a cold cache implementation, the cache is empty at the start. In the hot cache implementation, the trace preloads the cache by accessing all elements that will be future WrAP stores in the simulation. The speedup of WrAP is calculated as $T_U/T_W$. For $T_c$ between 0 and $T_w$, the speedup is $1 + T_w/(T_r + T_w)$. Using the values for $T_r$ (400) and $T_w$ (1200), the maximum speedup for a cold cache is 1.75. For a hot cache, where all the WrAP data is already in the cache, reads are a cache hit for both WrAP and the Undo Log implementations. However, the latter performs two stores to memory that must complete before the transaction can continue. The WrAP architecture allows for stores to lazily write to the log in the background, overlapping the reads and other processing. This can produce a speedup of slightly greater than
2.0. Figure 3.5 shows the speedup obtained for both cold and hot caches. A cold cache with no hits and little delay between consecutive write operations within a WrAP will have a speedup close to 1.75. As the time between persistent stores within a WrAP increases, the speedup drops off as the persistent writes in an Undo Log approach have more opportunity to overlap operations. Finally, the computation becomes heavily processor bound and both implementations require the same time.

\[ \text{UNDOS LOG: } T_{\text{ULOG}} = (T_R + T_W + \max(T_C, T_W)) \cdot N \]

\[ \text{WRAP: } T_{\text{WRAP}} = (T_R + \max(T_C, T_W)) \cdot N \]

Figure 3.6: Example time spent for a transaction of N=2 for Undo Log and WrAP.

Figure 3.6 shows an example of execution time of a wrap of N=2 elements where the CPU time \( T_c \) is equal to the write time for SCM \( T_w \) in a cold cache. The time taken using WrAP is slightly less than half the time required for an Undo Log approach. Reads are in a cold cache and take a full read time from SCM. Undo Log must synchronously read the old value and then write and commit the old value before storing the new value in the home location. WrAP on the other hand, can let the stores flow to memory while processing other instructions.

Continuing with the simple trace, we verified that larger caches provide for an
Figure 3.7: Comparison of WrAP and Undo Log for a simple benchmark of Speedup vs Cache Size

Increasing speedup of WrAP compared to an Undo Log implementation. Figure 3.7 shows speedup increasing for increasing cache size. For large cache size, the expected 1.75 speedup is reached. In addition, it is important to note that larger transactions will incur more cache misses, which slow the speedup.

**STL Benchmarks**

After verification of the expected behavior using the simple sequential benchmark, we generated numerous traces using the method described in Section 3.2.1. To generate the traces, for each of the four benchmark data structures Set, MultiSet, B Tree, and B+ Tree, we varied the transaction size from one to ten. Each transaction consists of an insert, find, and delete of randomly generated data values into the data structure.
The structure is initialized to contain 100 random elements, and the number remains constant through the trace. The traces were simulated for various CPU delays and cache sizes to generate the performance results.

Figure 3.8: Comparison of execution time for WrAP and Undo Log for STL-Based Benchmarks

Figure 3.8 shows the time required by different methods on the benchmark memory traces. We see that WrAP performs nearly as well as a method that does not guarantee transactional atomicity and significantly better than the use of an Undo Log. Figure 3.9 compares the speedup of the WrAP architecture over an Undo Log for each of the benchmarks as the size of the transaction is increased. The cache size is set to 8192 blocks and the CPU delay is one cycle between instructions. The speedup for different data structures varies between 1.25 and 1.85. The B-Tree benchmark takes
full advantage of the cache as the transaction set size increases, thereby pushing the speedup above that of a purely cold cache. Figure 3.10 shows the number of persistent store operations for each of the data structures as a function of the transaction size. The B-Tree implementation has almost double the number of persistent store operations per WrAP for the benchmark when transaction size is equal to five.

Figure 3.11 shows B-Tree simulation time for various benchmark transaction set sizes for each of the WrAP, Undo Log, and non-transactional approaches. The simulation was performed with a cache size of 8192 blocks and a CPU delay of one cycle between instructions. Even on small set sizes, the WrAP method performs almost as well as simply writing directly to fast cache.
Figure 3.10: Comparison of WrAP and Undo Log for STL-Based Benchmark: Number of store operations vs Transaction size

Figure 3.12 shows how each benchmark performs with increasing cache size. In this set of simulations, the transaction size is set to five and each of the benchmarks and models are simulated and compared using the appropriate memory trace. With an increasing set size, the increased number of persistent stores contained within a WrAP can impact the speedup. As shown in the models, more transactions, more cache hits, and less time between persistent stores within a WrAP all increase the speedup.
Figure 3.11: Execution time for B-Tree vs Transaction size

Figure 3.12: Benchmark Performance versus Cache size
Figure 3.13: STL-Based Benchmarks (a) WrAP Log Size (b) VPC Size
Finally, for the benchmark memory traces we compare the WrAP Log Buffer and Victim Cache sizes. We simulated various cache sizes from 128 to 8192 blocks for each benchmark with five Insert, Find, and Delete operations per WrAP transaction and a CPU delay of one cycle between each CPU instruction, where consecutive memory operations have numerous instructions of one cycle each between them.

Figure 3.13(a) shows the maximum WrAP Log Buffer size for each of the benchmarks. The maximum log buffer size occurs when the cache size is the largest in the group of simulations, as that is the point where the operations are performing the fastest with no cache misses and the slow, persistent memory might not keep up with a fast cache. The buffer only grows to a size of 50 in the maximum case for a B-Tree implementation that is conducting hundreds of persistent stores per transaction. The victim cache size was also examined in Figure 3.13(b). A large victim cache is seen when the memory cache is smaller, as more data is evicted from the cache into the victim cache. In the simulations, the WrAP architecture flushes all wrap log records and all entries in the victim cache to persistent memory before closing a WrAP transaction. The victim cache flush is a pessimistic assumption, since these can occur in the background without hurting correctness.

### 3.3 Summary

Uncontrolled writes to persistent memory locations can leave a system in an inconsistent state if there is a failure. Traditional transaction systems have dealt with the problem in the context of block-based I/O accesses to disks and more recently to SSD storage. However, it is only now that these issues have begun to be addressed for cache-line-granularity accesses to fast persistent memory systems. Conventional approaches are unsuitable because they interpose a software layer between the trans-
action and the memory system, thereby failing to fully exploit the advantages of fast, byte-addressable persistence.

In this Chapter we evaluated WrAP, an approach to using SCM in a memory-bus-based load and store architecture. WrAP propagates the writes within an atomic section along two paths: the fast path through the cache hierarchy and a slow path used for background updating of persistent memory and recovery from system crashes. A last-level persistent victim cache is used to prevent premature spilling of cache contents to persistent memory locations, while simultaneously avoiding costly look-up operations along the critical path.

Evaluation using synthetic micro-benchmarks of transactional operations on STL data structures (sets, multi-sets and B-Trees) shows significant performance gains over traditional techniques for persistence based on the Undo Log. We have shown in many instances that the WrAP architecture can perform as fast as a system that provides no consistency. The gains are expected to increase with transactions that have greater variable reuse and with concurrent transactions that access shared data using weak consistency semantics, an increasingly common mode of access in many current database applications. Continuing work involves evaluating the performance of the approach for different design alternatives, hardware and software tradeoffs, structure sizes, and different application domains.
Chapter 4

SoftWrAP

As shown in Chapter 1, the problem of atomicity is complicated by premature cache evictions. Proposed hardware changes to solve the problem of atomic writes to SCM are disruptions of the existing well-understood cache subsystems. The changes being incorporated into real systems is speculative and will certainly not be available in any form for the near future. Hence, we seek a solution that does not rely on changes to existing cache mechanisms yet can still catch premature cache evictions while ensuring the atomicity of a group of stores to persistent memory. The only hardware support assumed is a PCOMMIT instruction and an atomic write of 64 bits. Both have already been announced by Intel [15].

In this chapter, we present and describe Software-based Write-Aside Persistence, or SoftWrAP, a library to provide atomicity for a sequence of persistent-memory writes that may be interrupted by a machine restart or failure. We evaluate the performance of several benchmarks using SoftWrAP and compare it to other techniques for persistence.

4.1 SoftWrAP Approach

In this section, we describe the approach used by SoftWrAP, first showing an example of an atomic region in a program that requires atomic persistence to SCM. The major components of the SoftWrAP approach are then described in context of the example.
Algorithm 4: Programmer annotated atomic region

// x, p and pmalloc array are persistent.

wrapOpen();
begin
  \textbf{x} = 1; \quad \textbf{\textend{wrapStore}(&x, 1)};

  ........
  \textbf{p} = \textbf{pmalloc}(100);

  \quad \textbf{temp} = \textbf{pmalloc}(100);

  \quad \textbf{\textend{wrapStore}(&p, temp)};

  ........
  \textbf{\textend{for} (i = 0; i < 25; i++)}

  \begin{verbatim}
  \textbf{begin}
  p[i] = i; \quad \textbf{\textend{wrapStore}(p+i, i)};
  \end{verbatim}

  ........
\textbf{\textend{wrapClose}();}

Algorithm 4 shows an example program fragment. It comprises a single atomic region using two static persistent variables \textbf{x} and \textbf{p} and a dynamically allocated region of persistent memory. The variable \textbf{x} is a persistent integer value that is assigned a value. The variable \textbf{p} is a persistent pointer variable that points to an area of persistent memory obtained by a call to \textbf{pmalloc}, a persistent memory version of the usual \textbf{malloc} function.

A programmer identifies the atomic region within \textbf{wrapOpen} and \textbf{wrapClose} tags. The programmer (or a preprocessor) translates accesses to persistent memory within the atomic region to calls into the SoftWrAP library as shown in the comments. The
wrapOpen call marks the beginning of an atomic region that ends with a wrapClose. The atomic region will also be referred to as a *wrap*, and a call to *wrapStore* will be referred to as *wrapping a variable*. In our current implementation, the programmer must manually wrap the variables in the atomic section.

Figure 4.1: SoftWrAP managing paths through foreground cache hierarchy and background streaming stores to log.

The basic idea in SoftWrAP is shown in Figure 4.1 to simultaneously propagate updates made within an atomic region along two paths: a foreground path through the cache hierarchy that is used for value communication within and between wraps, and an asynchronous background path to persistent memory to log the updates. By
creating these two paths, SoftWrAP effectively decouples value communication for transaction execution from persistent memory logging for recovery.

The SoftWrAP approach has three logical components: **logging**, **alias handling**, and **retirement**. The **logger** maintains a sequential log in persistent memory that is updated using efficient, cache-line-combined streaming writes. The log is only used to recover from a crash. During normal operation, the log is updated efficiently in an append-only fashion using cache-line size combining writes. The log is periodically pruned by deleting entries of retired transactions (i.e., those whose updates have been retired to their home persistent memory locations). To handle the problem of spurious cache evictions, SoftWrAP employs a software **aliasing mechanism**. This redirects updates to persistent memory locations made by calls using the `wrapStore` function to stores located in a managed area of DRAM referred to as the **shadow DRAM**. The stores to shadow DRAM allow the updates to be freely communicated via the cache hierarchy (as is done for normal variables) but uncontrolled cache evictions can do no harm. The **retirement** component copies the values of aliased variables from the shadow DRAM to the persistent memory home locations when it is safe to do so. This step is done in the background, asynchronously and concurrently with foreground transactions. When a portion of shadow DRAM has been retired, it can be reused, and all log records from retired transactions can also be deleted.

Figure 4.2 shows a contiguous region $P$ of persistent memory that is mapped into the virtual address space $V$ of the application. The shadow DRAM is mapped to a different range of the address space $V'$ that is a small fraction of the size of $V$. $V'$ only maintains the recently updated wrapped variables in persistent memory and is regularly emptied by retiring its contents. An access to persistent address $a$ is redirected to address $a'$ by the aliasing mechanism. Following the first access to $a$
until it is retired, reads and writes to $a$ are performed from location $a'$. If evicted from the cache, the updated value of $a$ updates only the shadow DRAM location $\phi(a')$ rather than its persistent home location $\phi(a)$. Thus, value communication takes place via the cache hierarchy using the aliased location. The record of updates is streamed to a log area in persistent memory asynchronously and concurrently in the form of log records. The figure shows that following the wrapped write $a = 5$, the cached value 5 is backed by a DRAM address $\phi(a')$ corresponding to the aliased address $a'$ and the Redo Log stores a copy of the new value as the record ($\phi(a), 5$).
Algorithm 5: SoftWrAP API - wrapOpen, wrapLoad, and wrapStore

wrapOpen \((options \, o = \text{default})\)
begin
  if \(\text{WrapNestingDepth} == 0\) then
    Acquire handle to AliasTable;
    Open RedoLog;
    WrapNestingDepth += 1;

wrapLoad \((address \, a)\)
begin
  if Entry for \(a\) in AliasTable then
    Return value from AliasTable;
    Return value from SCM address \(a\);

wrapStore \((address \, a, \, value \, newVal)\)
begin
  if No entry for \(a\) in AliasTable then
    Add \((a, newVal)\) to AliasTable;
    Append \((\phi(a), newVal)\) to RedoLog;

4.2 SoftWrAP API and Implementation

Algorithms 5 and 6 present the basic implementation of the SoftWrAP library. The data structure to implement the alias table is a hash table based Key-Value store. A non-blocking implementation is used to avoid excessive locking and unlocking. For scalar variables the alias table itself stores the latest value of the variable, while for large objects only the pointer to the copy of the object is stored in the alias table.
Figure 4.3 (see Hash Table A) shows scalar persistent variables M, Z, N with values 1, 3 and 2 respectively in the alias table along with a size qualifier. In Hash Table B, object A (a 4K page) is stored in the alias table as a pointer to the shadow DRAM area where the copy of A is stored.

Wraps may be nested. For instance, an application may require that the insertion of an element into a B-tree data structure along with the deletion of some other element to be performed transactionally by enclosing the operations in a wrap. The persistent B-tree operations may themselves need to be wrapped to maintain data structure integrity while multiple internal tree pointers are updated. In our implementation, nested wraps are rolled up into the parent wrap. Hence calls to wrapOpen and wrapClose increase and decrease the nesting level by 1 respectively. A wrapClose with nesting level 0 indicates the closing of the top-level wrap and requires the log to be made persistent in SCM.

The outermost wrapOpen call, shown in Algorithm 5, registers a thread-specific handle to the alias table and to a RedoLog area allocated for the wrap. Subsequent nested calls by this thread are simply rolled up to its outermost call, since atomicity needs to be preserved at the outermost level.

Algorithm 5 also shows wrapLoad and wrapStore calls which are used for reading and writing scalar persistent variables within the wrap. The wrapStore implementation inserts or updates an existing entry for the passed variable in the Alias Table with the new value being stored. It also appends a log record with the persistent memory address and value of the variable to the end of the redo log for this thread.

Reading and writing object data is performed using wrapRead and wrapWrite, shown in Algorithm 6. This is useful for applications that read and write data in large extents. These objects are allocated space in the shadow DRAM and accessed
indirectly via pointers in the alias table. To simplify space management and support legacy database applications, objects are broken up into units of fixed-size pages (we use 1KB pages but this is a tunable parameter), and one entry is maintained per page in the alias table. A *wrapWrite* whose destination spans multiple pages that have all been already inserted in the alias table simply updates the data pages in shadow DRAM. Otherwise, if the data size being written spans an entire page, it is written to a newly allocated DRAM page. The worst-case occurs if the new data spans only part of a newly allocated page. In this case the updated data needs to be merged with missing bytes from persistent memory. A record containing the new data being written is also appended to the redo log in persistent memory and also benefits from write-combining.

Finally, *wrapClose* decrements the nesting depth of the wrap. If the current close indicates no more wraps are open, then all logs must be committed to persistent memory using a PCOMMIT. In our implementation of PCOMMIT, we also make sure that proper ordering is performed on memory stores using a memory store fence. Not shown in the description of the *wrapClose* function is the clearing of a pointer to the Alias Table.

Management of the Alias Table is described in the following section.
Algorithm 6: SoftWRAP API - wrapRead, wrapWrite, wrapClose

wrapRead \((address \ src, \ size \ n)\)

begin

Break up \(n\)-byte address range of \(src\) into

aligned page sequences \(S_i\);

for (each page \(S_i\)) {

if Entry for \(D_i\) in AliasTable then

Return \(D_i'\);

Return \(D_i\);

}

end

wrapWrite \((address \ dest, \ address \ src, \ size \ n)\)

begin

Break up \(n\)-byte address range of \(src\) and \(dest\) into

aligned page sequences \(S_i\) and \(D_i\) respectively;

for (each page \(D_i\) in the destination) {

if (No entry for \(D_i\) in AliasTable) then

Allocate Shadow DRAM page \(D_i'\) for \(D_i\);

Add \((D_i, D_i')\) to AliasTable;

Update page addressed by \(D_i'\) from \(S_i\) // Write DRAM

Append \((\phi(D_i), D_i)\) to RedoLog;

}

end

wrapClose ()

begin

WrapNestingDepth -= 1;

if WrapNestingDepth == 0 then

// Commit all log records to persistent memory

PCOMMIT; // Stall until pending persistent memory

writes are committed to SCM
4.2.1 Alias Table Design

The alias table is the key data structure in SoftWrap and needs to be managed carefully for performance. We implemented the alias table as a hash table based key-value store that supports update and lookup operations. Entries are never deleted from the table so we do not need to support a delete operation. Instead, the entire hash table is recycled after the home SCM locations of the variables are updated from the alias table.

To lookup an address $p$, the table is scanned starting from the index computed by $Hash(p)$ until either $p$ is found or the scan encounters a blank entry in the table. In the first case, an update operation can simply rewrite the value field of the existing

---

**Figure 4.3**: The Alias Table implementation for global aliasing is a double-buffered lock-free hash table implementation. It handles reads and writes to both primitive data types and object data. Values are retired directly from the table.
entry. In the second case, it must fill in the address (key) and value fields of the blank entry. A simple compare-and-swap test of the single entry just prior to the update is sufficient to prevent races. The table does not need to be locked nor does one need to lock extended code sequences.

The home locations in SCM of written variables need to be updated to their new values and the alias table memory freed for new entries. This retirement process could be done either from the values in the logs or from the alias table. Retiring updates from the log requires reading the log and then writing to SCM, while the latter approach can stream DRAM-resident data in the alias table to SCM using efficient memory scatter instructions. We use alias table based retirement in our design. For atomicity, it is necessary that only alias table entries from closed wraps be retired.

To permit retirement of the alias table entries in the background along with foreground activities (new wrap openings and closings, and wrap reads and writes) we use a two-table double-buffered approach. At any time one table is the active table and the other is being retired to SCM. However, we need to be careful to avoid races or unnecessary locking in implementing such an approach.

Figure 4.3 shows the two-table design. In the figure there are two hash tables A and B; B is the currently active table while A is being retired to persistent memory. Additions of new alias table entries are only made to the active table. However, lookups must consult both tables until we are sure that the latest value of a variable has been written to SCM, at which time the lookup can be made from its home location. In the figure, a store to variable \( W \) will lookup the active hash table \( B \) and either update the existing entry or add a new entry to \( B \). A load of \( W \) will look up table \( B \) first, and if found there (as in the figure) return its value. On the other
hand, a load of $M$ will fail in table $B$, and must be followed by a lookup of table $A$. If a variable is not in either hash table, then it must be retrieved from SCM, which is guaranteed to have its last updated value.

Figure 4.4: The lifecycle of Hash Table states within the Alias Table structure used for global aliasing with a single open wrap at a time.

The complete design of the two-table Alias Table design requires states to be maintained for the tables as shown in Figure 4.4. Each table can be in one of four states: Empty (E), Active (A), Closed (C), and Retiring (R) with the following semantics. In the $E$ state the table has no valid entries and is available for use. A table in the $A$ state will be used for making updates and will be given priority in lookups. When the number of entries in the table approaches a threshold and all wraps that were opened while this table was active complete, the table transitions to the $C$ state on the final \texttt{wrapClose} call. In the $C$ state, it is safe to begin retiring the table entries to the SCM home locations. During this time, lookups into the table from other executing wraps can continue safely and without conflict. When the retirement of the entries to SCM is complete, the table transitions to the $R$ state, after performing a persistent memory fence and commit. In this state no further
lookups will be permitted; instead all lookups that are not found in the active table will go to SCM. The table remains in the $R$ state until all the values are cleared and is then transitioned into the $E$ state.

On a $wrapStore$ or $wrapWrite$, if the hash table creates a new entry, an atomic increment is performed on the number of elements in the table. Since in our implementation there is currently only one thread with an open wrap, state transitions need only be handled on a $wrapClose$ when the outermost wrap is closed. On $wrapClose$, the number of entries is compared against a configurable threshold for the ratio of free to used space in the table. If the threshold is exceeded, then the $wrapClose$ transitions the state to the $C$ state. A background processing thread is also notified of the state transition, and can start retiring the entries in the table as previously mentioned. On a $wrapOpen$, if there are no active tables, the thread waits until the background processing thread signals that there is an empty table.

### 4.2.2 Aliasing Alternatives

Different implementations of the basic aliasing scheme described are possible. First, as an alternative to creating aliases in DRAM, one could instead simply alias a variable to its copy in the redo log record. This effectively trades away the cache efficiency which was achieved by treating the log records as non-temporal and not caching them. That is, redo log records would now need to go through the cache hierarchy to allow fast path communication, and cache misses resulting from evictions of these records would have to access the slower SCM rather than DRAM. That has the potential to degrade performance when the cache pressure is high and variable reuse is frequent. Also, the aliased location will change as different transactions access the variable and redirect it to their private log locations. Frequent updates will cause increased
coherency traffic as hash table entries are repeatedly invalidated.

The SoftWrAP library allows programmers to choose between global aliasing (default) for more algorithmic flexibility and local aliasing for the common strict isolation case. In the global (default) case, variables are held within the buffered alias table in a single hash table until it reaches a size threshold.

In the local aliasing case, on a `wrapClose`, all of the variables are flushed from the hash table into the cache hierarchy and write-combining buffers. Persistent memory flushing may be performed immediately after a `wrapClose` in this case or be performed at configurable intervals. Only after a persistent memory flush may log entries be retired.

A final consideration concerns the mapping of a shared persistent object in the address space of multiple threads. As is commonly done for shared libraries, in this implementation we assume a fixed mapping based on common agreement, in preference to more costly dynamic alias conflict handling mechanisms.

### 4.2.3 SoftWrAP Architecture

The SoftWrAP library is implemented in C and C++ and has been built and tested on Linux based systems including Red Hat Enterprise Linux Server 6.5 and Ubuntu 12.04 LTS and later operating systems. The SoftWrAP implementation builds with GCC 4.x and above. The SoftWrAP library can be statically or dynamically linked into software applications. SoftWrAP has an Application Program Interface for C/C++ and tools to automatically identify atomic regions and perform consistency checking.

Figure 4.5 shows the overall architecture of the related software components of the SoftWrAP library. The SoftWrAP API implementation defaults to Software-based Write-Aside Persistence, but may be configured with an environment variable to im-
plement Undo Log or Non-Atomic memory operations. Both SoftWrAP and Undo Log implementations utilize the same log manager and persistent memory management routines such as `pmalloc` and `pfree`. The log manager manages allocating space to new logs and saving pointers to logs of completed transactions in a persistent area. The log entries are comprised of a destination address, size, and value with arbitrary data sized objects being supported. Additionally, our log can be configured to favor one size of data value over another and reduce the log entry size further. Log areas are allocated in configurable sized chunks, with a default of 4KB, that can be chained together. The log manager and logs are optimized similar to Mnemosyne [13], such that memory barriers and flushes are minimized using an alternating protocol between rewrites of the log area to detect corrupted logs. On restart and recovery, the logs that have been marked completed by the log manager are replayed and data copied

Figure 4.5 : SoftWrAP Library Architecture
to the home locations. With SoftWrAP, entries into the log are streamed together and benefit from write combining [32, 36, 37].

The persistent memory management routines can be configured via environment variables on initialization to either write to DRAM or an SCM model that delays stores to model SCM (see Section 4.3), or use memory-mapped files for persistent memory. Emulating SCM uses DRAM for storage, however, streaming store and persistent memory flush operations are funneled through a store buffer as described in Section 4.3. The memory-map file is used for correctness checking, testing, and to persist applications between executions. We ran our micro-benchmarks using all three configurations, and verified the resulting contents of the memory mapped file against expected values. Environment variables can be specified to configure SCM write time and implementation options.

The SoftWrAP implementation utilizes a DRAM based Alias Table by default that is double-buffered with lock-free hash tables. However, the SoftWrAP API allows for the Alias Table implementation to be overridden for application-specific optimized versions.

4.2.4 Application Program Interface

The Application Program Interface, API, has both data primitive and object functions to access persistent memory atomically. Primitive data types, such as 32 and 64-bit integers, are supported directly in the Alias Table. See Table 4.1 for reference. Object data types have values in the table that are pointers to the base of the object and are accessed in 1K chunks. Offsets within an object can also be referenced, as the base pointer is found by aligning the pointer to a 1K boundary.

The implementation utilizes streaming, or non-temporal store instructions, which
allows a consecutive sequence of bytes to be combined into cache-lines and streamed
to memory without disturbing the cache. Stores to the same cache line are combined
into a single entry and written to memory in a single cache-line write operation. For
the implementation of the \textit{ntstore} function, on architectures that do not support the
multimedia streaming instructions, such as \texttt{mm_stream_si32}, we also support directly
writing to the memory location in the cache hierarchy and flushing the line with a
\texttt{clflush} instruction on a subsequent \texttt{pcommit} or \texttt{ntstore} to a different cache-line similar
to \cite{32, 24, 36}. Future Intel instruction sets will support the CLWB, or cache-line
write-back, instruction which will perform a flush of the cache line to main memory
without invalidating the entry. The instruction sets also will support the PCOMMIT
instruction which will flush buffers to main memory \cite{15}. Our implementation of
\texttt{pcommit} performs a memory fence to ensure proper ordering on stores to memory so
they are all visible before calling a commit intrinsic.

4.2.5 Simulation and Tools

A simulator was constructed to perform initial analysis of the SoftWrAP approach
and implementation. The procedure is outlined in Figure 4.6. For a simulation of
an application accessing SCM, we need to distinguish accesses to persistent memory
from accesses to volatile DRAM. A program is instrumented at runtime using Pin \cite{39}.
Pin is a dynamic, binary instrumentation tool that can analyze running programs.
We created a Pin tool, called WrapPin, that instruments all memory accesses and
instruction references along with injecting run-time code to overload functions such as
\texttt{pmalloc, pfree, pcommit, ntstore, and clflush}. The simulation is modeled after a 3
GHz processor, and every instruction incurs a one cycle cost. On each instruction and
memory reference, WrapPin passes the request to the cache hierarchy. By overloading
<table>
<thead>
<tr>
<th>SoftWrAP API</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>void *pmalloc(size_t size);</td>
<td>Allocate persistent memory in bytes.</td>
</tr>
<tr>
<td>void pfree(void *ptr);</td>
<td>Free a pointer in persistent memory.</td>
</tr>
<tr>
<td>void ntstore(void *dest, void *src, int size);</td>
<td>Streaming store to memory.</td>
</tr>
<tr>
<td>void pcommit();</td>
<td>Mem fence and persistent flush.</td>
</tr>
<tr>
<td>iint wrapOpen();</td>
<td>Open a WrAP in the current thread.</td>
</tr>
<tr>
<td>int wrapClose();</td>
<td>Close a WrAP in the current thread.</td>
</tr>
<tr>
<td>void wrapOptions(long options);</td>
<td>Sets options for WrAPs.</td>
</tr>
<tr>
<td>void setAliasTable(AliasTableBase *a);</td>
<td>Override alias table implementation.</td>
</tr>
<tr>
<td>void *wrapRead(void *ptr, int size);</td>
<td>Read a value at ptr of size bytes.</td>
</tr>
<tr>
<td>void wrapWrite(void *ptr, void *src, int size);</td>
<td>Write a pointer src of size bytes to ptr.</td>
</tr>
<tr>
<td>uint32_t wrapLoad32(void *ptr);</td>
<td>Load a 32-bit value pointed to by ptr.</td>
</tr>
<tr>
<td>void wrapStore32(void *ptr, uint32_t value);</td>
<td>Save a 32-bit value to ptr location.</td>
</tr>
<tr>
<td>uint64_t wrapLoad64(void *ptr);</td>
<td>Load a 64-bit value pointed to by ptr.</td>
</tr>
<tr>
<td>void wrapStore64(void *ptr, uint64_t value);</td>
<td>Save a 64-bit value to ptr location.</td>
</tr>
</tbody>
</table>

Table 4.1: Subset of the SoftWrAP Application Program Interface

`pmalloc`, WrapPin notes the virtual address ranges in the user program space which the user benchmark program deems to be persistent. We model our cache hierarchy after an Intel 5660 Processor with split L1 data and instruction caches of 32 Kb each, a L2 unified cache of 2 MB, and an L3 cache of 16 MB with 64 byte cache lines. L1, L2, and L3 access incur 2, 10, and 40 cycles respectively.

On a cache miss, eviction, flush, or `ntstore`, the memory request is checked against a fast data structure to check if the virtual address is contained within the virtual
address ranges obtained from overloading the `pmalloc` routine. If the address is in the persistent range it is sent to a memory model that models SCM, otherwise it is sent to a DRAM model. Both our memory models utilize DRamSim2 [41] as a cycle accurate memory simulator. The DRAM is simulated as a Micron DDR3 666 MHz DRAM memory with burst length of 8. For SCM, we utilize the parameters used in [42] for an experimental setup of PCM. These parameters are modeled after Micron’s DDR2-800 SDRAM and further modeled by [19] which operates at 400 MHz and has a burst length of 4. The overloaded `pcommit` function waits and cycles the memory subsystems until all stores are out of the memory buffers.

Early results utilizing the simulation method showed good speedup for SoftWrAP
over Undo Log or Copy-On-Write based approaches [37, 43]. However, results depend on the accuracy of the CPU simulation, instruction issuing and modeling of the cache hierarchy. Additionally, Pin cannot be used to simulate out-of-order execution, as the analysis phase is performed in-order. The performance of SoftWrAP using aliasing can be increased from out-of-order execution on modern processors.

Further analysis of the SoftWrAP implementation was performed on a DRAM interposer or tracer tool at Intel. A test application running a series of SoftWrAP transactions was executed and a DRAM memory trace was produced. Analysis of the trace showed additional speedups of SoftWrAP due to prefetching and write-combining not modeled by the simulator. Therefore, we opted to also support an emulation method of SCM for evaluation which is described and used in Section 4.3.

The simulator is still very useful as it can automatically find lines of source code that need to use the SoftWrAP API for atomic persistence. It can identify code by a programmer performing two steps. First, data objects that need to be persisted are set to be allocated and freed with the `pmalloc` and `pfree` routines. Next, the tool is launched with Pin and a running instance of the program linked with the SoftWrAP library. Our Pin based tool specifically looks at memory references to memory allocated with `pmalloc`. The tool confirms that the memory access has been preceded by a `wrapOpen` call and that the reference is also included in a call using a `wrapRead`, `wrapWrite`, `wrapLoad`, or `wrapStore` routine. If persistent memory access is not within these boundaries, then our tool accesses the debugging information of the running executable and prints out the associated file name and line number. Since it is a running application, and these lines will, in many cases, be accessed multiple times, the file names and lines are sorted, grouped, and displayed. This allows for easy identification of areas of code that need to use SoftWrAP for atomic persistence.
4.3 Evaluation

In this section we describe the evaluation of SoftWrAP using micro-benchmarks and two applications: STX B+Tree library, and a SQLite database running a scaled TPC-C benchmark. We will compare our SoftWrAP approach with two other methods referred to as Non-Atomic and UndoLog.

In Non-Atomic all the stores in a wrap are made durable on SCM before the next wrap can begin. This ensures the atomicity of completed wraps, but a machine failure during the execution of a wrap can leave SCM in an inconsistent state. This approach provides a lower bound on the performance of methods which provide both durability and atomicity. In UndoLog the idea is to directly update SCM just as in Non-Atomic. However, to preserve atomicity of all wraps, the current values of the variables before the update are made persistent on SCM in an undo log. In the case of failure the state can be rolled back to the state at the start of an incomplete wrap by replaying the undo log. At the end of the wrap, all updated values are retired to their home location by a persistent commit operation.

In SoftWrAP updated values are written to DRAM shadow memory and to a redo log. However, there are two main differences from UndoLog which improve its performance remarkably. First, the log records in SoftWrAP can be write-combined into compact cache lines and streamed to the log asynchronously. In contrast, UndoLog needs to synchronously write the current value of each variable to the undo log before it can update its value. Secondly, UndoLog persists all its updates at the end of the wrap to guarantee the durability of the completed wrap. SoftWrAP merely writes its updates to DRAM shadow memory (the alias table), and only needs to persist the much more compact sequential log structure when the wrap closes. The updates of the persistent memory locations are performed asynchronously in the background.
As our experiments will show, this has two performance consequences. First, both the throughput and response time (time from wrapOpen to the end, wrapClose) in SoftWrAP will be significantly better than UndoLog due to the asynchronous write of a smaller number of cache lines for the log and the the asynchronous writing back to the home locations. Secondly, the throughput of SoftWrAP will be slightly less than Non-Atomic because the latter does not need to write any metadata log records. However, the response time will be slightly better than Non-Atomic because its retirements are done asynchronously in the background while Non-Atomic needs to compete the retirement synchronously before closing.

Table 4.2 shows the number of SCM writes and pcommit operations for a wrap of \(n\) word-sized stores using the three methods. Non-Atomic requires the \(n\) words (to scattered SCM locations) to be written to SCM with a single pcommit instruction to ensure their persistence. UndoLog requires a pcommit after writing each log record and another at wrap close to persist the updates. A log record is 12 bytes long for a 4-byte word, 8 bytes for the address and 4 bytes for the value. Each of the \(n\) log records generally requires 1 cache line write, and each of the \(n\) updates requires another write, for a total of \(2n\) writes. The additional terms in the expression are 1 write for an end-of-log marker and a correction term to account for splitting of a log record at a cache line boundary. Cache lines are 64 bytes. Finally, SoftWrAP can close the wrap after writing the \(3n\) consecutive words (12\(n\) bytes) that make up the log, followed by a single pcommit. Due to the write combining of the sequential cache lines, this results in \(12n/64\) cache line writes.
<table>
<thead>
<tr>
<th></th>
<th>SCM Writes</th>
<th>pcommits</th>
<th>Estimated Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-Atomic</td>
<td>n</td>
<td>1</td>
<td>$nT_w + T_s$</td>
</tr>
<tr>
<td>UndoLog</td>
<td>$2n + 1 + \lceil 12n/64 \rceil$</td>
<td>$n + 1$</td>
<td>$n(2T_w + T_s) + \lceil 12n/64 + 1 \rceil T_w + T_s$</td>
</tr>
<tr>
<td>SoftWrAP</td>
<td>$1 + \lceil 12n/64 \rceil$</td>
<td>1</td>
<td>$T_w + \max(n \ast T_{alias}, \lceil 12n/64 \rceil T_w) + T_s$</td>
</tr>
</tbody>
</table>

Table 4.2: Time to perform a wrap of $n$ 4-byte words.

4.3.1 Experimental Results

In this section, we describe our experimental results with SoftWrAP. The absence of readily available systems with persistent memory raises challenges in evaluating the performance of SCM-based software. Simulation on a cycle-accurate processor and memory simulator like DRAMSim2 with NVRAM extensions is time consuming, and the accuracy depends on the fidelity in modeling many details from the processor pipeline, the cache hierarchy, the memory controller, and the DRAM and SCM memory systems. We opted for an approach similar to that used for Mnemosyne [13] based on measuring the execution time of a running application that has been instrumented to add a delay to specific types of persistent memory stores. A description of the emulated system model and its validation is presented in Section 4.3.2. This is followed in Section 4.3.3 by microbenchmark experiments to demonstrate and validate the behavior and advantages of SoftWrAP as predicted in Table 4.2. In Section 4.3.4 we discuss the evaluation of the STX B+Tree library, and in Section 4.3.5 evaluate a SQLite database running a scaled TPC-C benchmark.
4.3.2 Model Validation

The streaming non-temporal store instruction (ntstore) allows a consecutive sequence of bytes to be combined into cache-lines and streamed to memory without disturbing the cache. The emulated memory system to handle ntstore operations has a write-combining buffer implemented by an 8-entry, cache-line-wide queue. Stores to the same cache line are combined into a single entry and written to memory in a single cache-line write operation. The time for a cache-line write from the head of the buffer is denoted by $T_w$. In our experiments this is set at $T_w = 1\mu s$, a conservative write time for SCM [44].

On an ntstore operation, a run-time routine is invoked to emulate the memory system. The routine waits until there is space in the write buffer and then adds the request to the queue, if possible combining it with a pending store to the same cache line. Consequently, when a group of sequential writes are made in quick succession, they are write-combined into a single cache-line effectively increasing the data that can be written in a single memory operation by a factor of 16 over scattered single-word writes (cache line size is 64 bytes and word size is 4 bytes). A pseudo-instruction pcommit is introduced to emulate the forcing of queued requests to memory, which will behave similar to the PCOMMIT instruction recently introduced by Intel. However, our pcommit function also performs a memory store fence to force an ordering on stores so that they are visible to the memory subsystem. This is crucial for the proper behavior of calling the Intel instruction as it only ensures globally visible stores are committed to persistent memory. A pcommit call is implemented by inserting a software delay loop until the completion time of the last queued request. The delay loop is bookended by CPUID instructions to force the serialization of the delay with respect to the rest of the program.
Validation of \textit{ntstore} and \textit{pcommit}

Our tests are performed on an Intel(R) Xeon(R) CPU x5660 at 2.80GHz with 64GB of DDR3 system memory running Red Hat Enterprise Linux Server 6.5. The SoftWrAP implementation and benchmarks were built with GCC 4.4.7. First, we tested the implementations of the streaming non-temporal store (\textit{ntstore}) and persistent memory sync (\textit{pcommit}) operations, which are used to emulate the writes to SCM. A group of $n$ 4-byte writes using \textit{ntstore} instructions in a tight program loop was run and its execution time recorded. The average time for a store in the group is computed and reported in the plots in Figure 4.7. The stores in a group are either to random addresses within a large array or to sequential array locations that can
exploit write-combining. Two backing devices were tested: DRAM and emulated persistent memory. For the DRAM-based experiments the \textit{ntstore} was streamed directly to memory bypassing the emulated write buffer, while for the persistent memory experiments the time for an SCM write $T_w$ was set to $1\mu s$.

The average time for a store in a group is plotted for different group sizes in Figure 4.7. The time for writing a group of $n$ random writes can be estimated as follows: for a non-full buffer the time is a small amount $\epsilon$ equal to the time to insert the request in the buffer; a sequential stream of back-to-back requests to a full buffer will each incur a delay $T_w$ to free up a buffer entry. Hence, the delay incurred by a group of $n$ writes assuming a buffer of $B$ entries is approximately $n\epsilon$ for $n \leq B$ and $B\epsilon + (n - B)T_w = nT_w - B(T_w - \epsilon)$ for $n > B$. For random writes the estimated write time is $T_w(1 - B/n) + B\epsilon/n$ for $n > B$ which is approximately $T_w$ for large $n$.

The measured average write time becomes steady at about $100\text{ns}$ for DRAM (a reasonable measured value for $T_w$ for DRAM writes) and, as predicted, to the delay $T_w = 1000\text{ns}$ for emulated SCM, since once the write buffer is full the writes are written at the average rate of $T_w$.

When the group is made up of sequential writes, they are write-combined into a single cache-line effectively increasing the data that can be written in a single memory operation by a factor of 16, the cache-line size of 16 4-byte words. The reduced time for sequential writes is also shown empirically in Figure 4.7. Note that the drain rate of DRAM for sequential writes is so fast that the measured delay reflects the instrumentation overheads. Nonetheless, the experiments show that our delay model provides a good emulation of the SCM write performance for use as a basis of comparison.
Validation of predicted performance of different methods

To validate the predicted behavior of the three methods, we conducted the following experiment. We executed 100,000 consecutive wraps, each consisting of a group of $n$ 4-byte writes. The arrival rate of the wraps was fixed at 10,000 wraps per second. Writes were made to random word addresses in a large 1GB array to avoid effects of data reuse and caching. The size of each hash table is 8K entries, and an entry is made up of an 8-byte address field, an 8-byte data field and a 4-byte size field. Flushing of the hash table to SCM begins when the table has 500 entries. A sparse table allows rapid insertion and lookup operation but is large enough to amortize the overheads of creating and retiring the table. The performance as a function of cache size is discussed in Section 4.3.3 below.

We recorded the number of writes to SCM, total number of $pcommit$ operations, and the average execution time for each method when $n=10$, $n=15$, and $n=20$. A comparison of the predicted and experimental times are shown in Table 4.3. There is close agreement between them for Non-Atomic and UndoLog. For SoftWrAP there is an additional unknown variable $T_{alias}$. Computing this value from the experimental data for $n = 10$ we use it to predict the execution time for the other values of $n$ and get a good correspondence.

The number of $pcommit$ and $ntstore$ operations are also measured during the experiment and are shown (per wrap) in Figure 4.8. For Non-Atomic and SoftWrAP there is 1 $pcommit$ operation at wrapClose, while UndoLog performs $(n+1) pcommit$ operations. The retirement of the Alias Table only requires 1 $pcommit$ every time a hash table is written to SCM, which occurs at a fixed fraction (determined by the retirement threshold) of the incoming transaction rate. Likewise, the number of SCM stores for Non-Atomic and Alias Table processing is $10^5 n$ total and $n$ on average as
Table 4.3: Performance of Array Update Benchmark. $T_w = 1 \mu s$ and $T_s = 220 \text{ns}$. $T_{alias}$ calculated is $450 \text{ns}$.

shown. SoftWrAP reduces the number of stores by a factor of 16 while UndoLog has roughly twice the number of $ntstores$ compared to Non-Atomic. The experimental results match the predictions from Table 4.2.

4.3.3 Micro-Benchmarks

In the following experiments, we create a large 1GB array and perform a number of experiments using the same machine setup as described above. We test the wrap response time and throughput and their dependence on the size of the alias table. Additionally, we determine the sensitivity to SCM write time and number of elements in a wrap. Finally, we examine reuse of data across wraps, and show additional benefits from the caching performed by SoftWrAP.
Response Time and Throughput

First, we consider wraps made up of 10 random updates to the array. We insert a tunable delay in-between transaction arrivals to simulate different arrival rates. The transaction arrival time and completion times are recorded along with the throughput. The arrival rate is varied from 10,000 to 200,000 wraps of $n=10$ elements each per second. The results are shown in Figure 4.9.

The results are shown on log-log plots to include the performance of UndoLog. The response time at low arrival rates for Non-Atomic for $n=10$ and $T_w=1\mu s$ is slightly more than $10\mu s$. By comparison, UndoLog has more than double the response time, while SoftWrAP is almost a factor of 2 faster than Non-Atomic. The times follow
the service times of the wrap as in Table 4.2 at low request rates. When the arrival
rate exceeds the service rate, all methods show a sharp increase in response time.
However, SoftWrAP can absorb a much higher arrival rate before breakdown. Before
the knee, the alias table can be retired in the shadow of the foreground operations,
without slowing the latter. However, ultimately the rate at which the Alias Table is
filled exceeds the rate at which it can be retired, and the wraps stall waiting for the
alias table to become free.

The maximum throughput of both UndoLog and SoftWrAP is slightly less than
100,000 transactions per second. The maximum throughput of UndoLog is only about
39,000 transactions per second. As discussed previously, each wrap in UndoLog is
slowed down since it does significantly more writes than the other methods. Non-
Atomic must perform a pcommit operation after each wrap (group of 10 writes),
therefore performing slightly less than the maximum possible throughput of 100,000
transactions per second based on memory bandwidth alone. SoftWrAP isn’t limited
in the writing to the table but its throughput is limited by the time to retire the
alias table. It is an interesting property that at reasonable arrival rates in a write-
intensive workload, SoftWrAP can have a service and response time faster than that
of the Non-Atomic method that does not guarantee atomicity, and have only a slightly
less throughput due to the small logging overhead.
Figure 4.9: Response time and throughput for varying arrival rates of transaction size n=10, alias table size of max 8k entries, and SCM $T_w=1\mu s$. 
Alias Table Size

Figure 4.10: Maximum Throughput for various Alias Table sizes. Arrival rate = 50k wraps per sec, n=10, $T_w=1\mu s$.

In the following sets of experiments, the sensitivity of SoftWrAP to the size of the alias table is analyzed. Our implementation of the alias table uses a double buffering technique with two hash tables. Each hash table is array based and walks the hash index looking for a match or free entry. To avoid walking a dense table, we start retiring the table using a retirement thread after reaching a configurable size threshold, currently at 1/8 the capacity. On a wrapClose operation, if the table has reached or exceeded the threshold size, the retirement thread is notified. The retirement thread might be busy processing the other table, in which case a new wrapOpen may have to wait until that table has been completely retired.
Each element in the table requires a key, value, and size; so an entry requires 24 bytes. We test the size of the alias table for \( n = 10 \) elements per wrap and an incoming rate of 50,000 wraps per second. As shown in Figure 4.10, when the table is relatively small (\( 2^{12} \) entries or less) or very large (greater than \( 2^{20} \) entries) the throughput falls. The reason for the drop at large table sizes is that the alias table thrashes in cache causing a sharp drop in throughput. At small sizes the bookkeeping operations in creating, freeing, and updating state of the hash tables require a significant fraction of the retirement time, as discussed below.

Figure 4.11 shows the response time and combined alias table size in the double buffer implementation for a subset of time in the same experiment. Once one hash table fills up, the retirement thread must be signaled and the hash table switched. The spike in response time as measured in the experiment is caused by the filling of a hash table, since the incoming rate of 50,000 wraps per second of 10 writes each, fills a table with threshold 500 every 1ms. The extra time includes signaling the retirement thread, zeroing out the table, and performing a memory barrier. The retirement thread can also zero out the table, but first must copy entries to home locations and perform a barrier to prevent threads from reading stale values.
Figure 4.11: Response Time and alias table entry size over time. Arrival rate = 50k wraps per sec, n=10, Alias Table: 8k entries, $T_w=1\mu s$. 
Thread-Local Alias Table

In thread-local aliasing, variables are only visible to the current thread and a global alias table is not needed. Instead, values are written to a local table. When writing values, log entries of the addresses and values are streamed to the log area just as in global aliasing. When the outermost wrap in the thread closes, the values in the alias table are streamed to store buffers. Alternatively, the values may be copied into the cache hierarchy and later stored to SCM. In either situation, variable values must be committed to persistent memory before the associated log is removed.

In our experiments, we use ntstores to stream all the values in the alias table to persistent memory on a wrap close. The non-temporal store instructions from Intel also update the associated variable in the cache hierarchy while streaming to memory. In this implementation, we perform a pcommit at the end of the wrap and then remove the associated log.

Figure 4.12 shows the average response time of wraps using the thread-local alias table technique. It also shows the time when the values are not retired to SCM but instead copied to the cache hierarchy for later retirement. Each wrap has n=10 elements at an incoming rate of 50,000 wraps per second. Note that the response time is much more flat than the response time for the global aliasing method due to no thread interaction with a retirement thread. The response time includes the time for the wrap aliasing overhead plus the time to store n=10 elements to SCM at $T_w = 1\mu s$ each with a final pcommit. The average response time for thread-local aliasing with full retirement is $16.8\mu s$ compared to only $7.8\mu s$ for global aliasing. Copying from the alias table to the cache hierarchy has a $7.23\mu s$ average response time, but in this test the entries are not persisted by a management thread.

One optimization is to relax the retirement so that it is only performed at certain
Figure 4.12: Response time using a thread-local alias table for copying to home locations and for only to cache over time with arrival rate of 50k txps of size n=10 and SCM $T_w=1\mu s$.

Another optimization is that the stores may be committed in the background before removal of the log entries or flushed from the cache by a management thread. These optimizations will be explored in future work.

**SCM Speed**

Next, we vary the SCM write time $T_w$ in an array update at 50,000 txps in Figure 4.13. On fast $T_w$ times approaching the speed of DRAM, the SoftWrAP overhead is greater than the Non-Atomic time. As the time to write a cache-line to SCM is increased, the benefit of SoftWrAP increases as the aliasing time $T_{alias}$ remains
constant. At SCM $T_w = 600\, ns$, both implementations have equal performance. This is as expected from the values in Table 4.2, setting $n=10$, and $T_w = 600\, ns$, yields a Non-Atomic time of 6,220\, ns, close to the same processing time for SoftWrAP. Additionally, for SoftWrAP, the time required for aliasing, $T_{alias}$, can overlap the time required to stream SCM writes to the log, aside from the final $T_w$ log write. This is verified with SCM $T_w = 1,500\, ns$, as SoftWrAP has a response time measured at 6,640\, ns. SoftWrAP, even with fast SCM, provides atomicity and consistency to a group of writes at speeds close to Non-Atomic.
Figure 4.14: Response time for various transaction sizes with arrival rate of 1,000 per second, alias table size of max 8k entries, and SCM $T_w=1\mu s$.

**Transaction Size**

Figure 4.14 shows the relationship between the size of a transaction and the effect on the overall performance. As the size of the transaction grows, SCM writes in SoftWrAP only grow at 12/64 the rate. When there is just one element in a transaction, obviously any overhead of logging by SoftWrAP, Undo Log, or any transactional method, will perform worse when compared to Non-Atomic. However, with just two elements in a transaction, SoftWrAP can perform as well as Non-Atomic. Certainly, anyone wishing to update just one element would not go to the trouble to create a transaction mechanism. However, even if one does, or the size of an update is not known prior to the start of a transaction, then SoftWrAP still performs exceptionally
Non-Atomic provides a lower bound on direct update methods to random locations in a large data structure. However, if the updates are to be performed within a small, known region or block of persistent memory, like a 1KB block, and a pointer to the data block can be flipped, then a Block-Copy method could be used to atomically commit multiple stores to data elements within the block. In a Block-Copy, an entire block of data is copied from the destination location to a new location and may temporarily remain in cache. The new data values are saved into the copied block. After all of the updates are performed, block data is flushed from the cache and a final \textit{pcommit} is performed to ensure the block is committed to persistent memory. The entire block is contiguous in memory and benefits from write combining. A 1KB block, which could contain 256 4-byte integers, will only require 16 cache-line writes to SCM. If these 256 integers were updated using another method, they might require separate writes.

Once a block is flushed, then an atomic pointer flip may be used to point to the new block of data, but the new pointer must also be committed to persistent memory. Figure 4.14 also shows the response time for a Block-Copy method for atomic persistence with a block size of 1KB. The items in the transaction are randomly dispersed only over a 1KB block, and assumes that the data structure allows a block to be atomically replaced with a new block by a pointer switch. Note that the response time doesn’t vary noticeably with the change in the number of elements in the transaction. Most of the data movement and writes to SCM is for the 1KB block of copied data, regardless of the number of elements in the transaction. New data can be replaced instead of writing the old data to further reduce traffic.

When updating a number of data elements in a contiguous region of memory, then
a block update method might be preferred as it takes advantage of write-combining across the entire block. When SoftWrAP retires entries from the Alias Table, the entries, even if contiguous in main memory, will not be contiguous in the Alias Table, since hashed values are ideally spread out evenly over a hash table array. Therefore, on retirement, SoftWrAP can’t take advantage of individual elements written separately that are part of a larger block of data updated with individual stores. However, SoftWrAP does support object updates which take advantage of block updates and can outperform Block-Copy.

**Data Reuse**

![Graph](image)

Figure 4.15: Maximum throughput for various percentage of data reuse across transactions with size n=10, alias table size of max 8k entries, and SCM $T_w=1\mu s$. 


The previous experiments only examine the cases where data was not reused across transactions. However, as in the case of many applications, data is reused both across and within transactions for both reads and writes. We show in Figure 4.15, that with data being reused between transactions, that the overall throughput is increased. With no reuse, SoftWrAP performs equally as well as Non-Atomic. When 10% of the data is reused across the transactions, then the number of writes to SCM that must be performed by the retirement thread is decreased by 10%. In the maximum throughput case, the bottleneck is the processing of a full hash table. Reducing this burden by a percentage allows the throughput to increase proportionately and is validated by the experiment. Undo Log and Non-Atomic do not benefit from data reuse and remain unchanged as expected.

### 4.3.4 B+Tree Benchmark

The previous experiments consisted of all write workloads. In this experiment, we also perform a series of writes into a data structure. However inserting just one element into a B+Tree requires a number of reads and writes that need to be wrapped with atomic semantics to preserve the integrity of the tree. We setup this experiment by slight modifications to the STX B+Tree [45] extension to the C++ STL, wrapping persistent data accesses in the B+Tree elements. We created a STL based memory allocator that allocates memory segments for the B+Tree in an emulated persistent memory area. Nested transactions get rolled up into the top level transaction. For Non-Atomic, just a single `pcommit` is executed once all nested writes are finished in the insertion of an element, but if a failure occurs, the data structure can become corrupted.

To test the performance of the B+Tree, we perform 1M transactions of one ele-
Figure 4.16: Average insert throughput of random elements into a B+Tree with alias table size of max 4k entries and SCM $T_w=1\mu s$.

ment inserts into the B+Tree, recording the number of transactional reads and writes to SCM needed by the internals of the B+Tree data structure along with the average time per transaction. The fastest way to build a B+Tree with a given set of elements is sequentially, as it requires less traversing and modification of internal nodes. Random element insertion takes longer than a sequential insert, but the amount of time required for an insert still approaches an expected time as the tree grows. The throughput for a Random series of insertions is shown in Figure 4.16. The average number of reads and writes per insertion along with the average insertion time is recorded in Table 4.4. The results for both Sequential and Random series of insertions are shown in Figure 4.17. Due to the contiguous nature of data elements
allocated in the B+Tree, many of the writes, even in the Non-Atomic method, get write-combined into a single cache-line write to SCM.

In our Sequential experiment, 1M inserts requires 61.6M reads and 8.1M writes, an average over 61 reads and 8 writes per insert. Non-Atomic averages 7.4μs per insertion which is less than the expected average insert time that would require 8 writes to SCM. Since at $T_w=1\mu s$ with 8 writes, the Non-Atomic method requires approximately $8*T_w$ or 8μs. However, we also measure a write combining effect of the B+Tree of 56% which reduces the Non-Atomic SCM writes to 5. The expected time from Table 4.2 is therefore 5,220ns. The remaining execution time of 2.2μs is needed for reads and traversing the tree.
<table>
<thead>
<tr>
<th></th>
<th>Sequential Inserts</th>
<th>Random Inserts</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Reads</td>
<td>61.6</td>
<td>74.8</td>
</tr>
<tr>
<td>Number of Writes</td>
<td>8.1</td>
<td>20</td>
</tr>
<tr>
<td>UndoLog</td>
<td>24.4µs</td>
<td>51.4µs</td>
</tr>
<tr>
<td>Non-Atomic</td>
<td>7.4µs</td>
<td>10.5µs</td>
</tr>
<tr>
<td>SoftWrAP</td>
<td>7.8µs</td>
<td>13.4µs</td>
</tr>
</tbody>
</table>

Table 4.4: Average number of reads and writes required along with the average time needed to perform an insert into a B+Tree for various persistence methods.

Undo Log requires all 8.1M writes to be synchronous. Referring back to Table 4.2, Undo Log for n=8 writes takes $19T_w + (n+1)T_s$ or 21.3µs. The same additional read and transversal time for the internal data structure as needed by Non-Atomic of 2.2µs is added. This yields 23.5µs for Undo Log which is close to the measured 24.4µs.

For a SoftWrAP of 8 writes and 62 reads, the time required from Table 4.2 is 4.8µs for the writes plus the cost of 62 reads into the alias table structure, for the measured 7.8µs. The cost of the read is then calculated to be approximately 44ns which is higher than a direct read to memory might cost, but reasonable given the complexity and benefit of SoftWrAP. For the Random insertion experiment of 1M elements, 74.8M reads and 20M writes are measured, which produces similar expected times that match the measured experiments for each method as well. SoftWrAP outperforms Undo Log and remains close to Non-Atomic even with heavy read-intensive workloads.
4.3.5 TPC-C Database Benchmark

Finally, we tested the performance of SoftWrAP with the TPC-C Benchmark [46]. The TPC-C benchmark is a popular, complex online transaction processing (OLTP) benchmark. It performs different types of transactions across nine tables with various data sizes.

We chose to test SoftWrAP with SQLite due to its widespread use and ease of extendibility. SQLite is implemented as a compact library and embedded database engine with a pluggable interface for extended media [47]. SQLite performs writes to the main database file atomically by first creating a journal file. The journal file is like a re-do log in that if a crash happens before the main database update has been completed, the journal can be replayed to capture any outstanding changes.

Figure 4.18: SQLite Pluggable Virtual File System Implementations using SCM for A) block writes and B) SoftWrAP.

Figure 4.18 shows how SQLite is extended to use Storage Class Memory using the Virtual File System Interface. This interface requires implementation of Open, Close, Read, Write, Sync, and other methods that closely align with the SoftWrAP API. In Figure 4.18 (A), a SQLite VFS is created that performs all writes to SCM.
When the VFS requests a Sync or Close operation, a persistent memory `pcommit` is performed to ensure that all writes to SCM have been committed. The journal is also updated in SCM just like the main database, and the journal can be discarded once the main database has been updated.

In Figure 4.18 (B), the SQLite VFS utilizes the SoftWrAP library. In the SoftWrAP version, the journal need not be created directly as direct writes to the database are contained within the SoftWrAP based logs. Only the main database file needs to be updated when using SoftWrAP. Therefore, the updates to the main database can be streamed to the SoftWrAP log location and aliased in DRAM. Database reads need not query the SCM based journal, but rather just utilize the SoftWrAP API, which can direct a read to either the DRAM alias table or home SCM location. This reduces the overall number of SCM reads, writes and persistent memory syncs.

To generate the SQL statements that represent a portion of the TPC-C Benchmark, we utilize the PY-TPCC engine, a Python based implementation of the TPC-C Benchmark [48]. The test with SQLite is performed with the SCM VFS from (A), and the SoftWrAP based implementation shown in Figure 4.18 (B). Undo Log and Non-Atomic are also tested with (B).

Figure 4.19 shows the throughput of SQLite for the TPC-C benchmark operations for various SCM write times. Note that SoftWrAP performs almost as fast as a Non-Atomic implementation that can potentially leave the database unstable after a system crash. Execution on disk based media resulted in less than 20 transactions per second, and SoftWrAP is over 30 times that performance at 1 and 2 $\mu$s SCM write times.
4.4 Summary

This chapter presented SoftWrAP, a software library for persistent memory that provides lightweight atomicity and durability. SoftWrAP utilizes an alias table approach to benefit from processor caching while avoiding the problems caused by uncontrolled cache evictions. This is coupled with micro-logging of updates that allow for write-combining of streaming store operations (not possible with synchronous Undo Log methods), and asynchronous background retirement of the updates to SCM (not possible in the Non-Atomic approach). SoftWrAP allows for a fast access path to data through the cache while making sure that persistent memory layers are not slowed
down.

We compared SoftWrAP to implementations that write directly to SCM and do not guarantee atomicity (Non-Atomic approach) and that use well-known Undo Log approaches to guarantee atomicity. Our experiments show significant speedups over Undo Log based atomicity and comparable performance to Non-Atomic implementations that provide no safety guarantees. Just as transactional memory liberates software developers from the mechanics of achieving execution atomicity, Software-based Write-Aside persistence with the SoftWrAP library liberates software developers from the minutiae of delivering atomicity in storage operations while not affecting performance.
Chapter 5

Forward Work and Conclusions

Persisting data atomically in Storage Class Memories is an exciting and fast paced research area with many opportunities. This chapter discusses some of the forward work for both the software-hardware WrAP Architecture and SoftWrAP library approaches and presents conclusions from both methods.

5.1 Forward Work

The following sections outline future research opportunities for both the hardware architecture for WrAP and software enhancements and optimizations for SoftWrAP.

5.1.1 Hardware WrAP

In the hardware-software WrAP architecture, there were many areas for further enhancements and investigations from the Victim Persistence Cache and retiring of logs to multi-threaded considerations.

Controller

The Victim Persistent Cache (VPC) in the WrAP architecture had two areas for future work. First, the size of the VPC was limited and could experience overflow when there were excessive cache evictions of persistent variables. Although the size was analyzed and possible failure alternatives such as overflowing into DRAM were discussed, this situation can be costly, especially in the presence of multi-threaded applications.
Second, except for when there were no open wraps, determining when the VPC could be safely pruned or purged was left for future considerations. Even flushing the VPC when no wraps were open would still leave current persistent variables in the cache hierarchy and subsequent cache evictions would start filling the VPC up again. Forward work will include exploring ways to manage the VPC entries by associating entries with a set of open wraps. Knowing which wraps might be accessing the variables in the VPC can allow for the VPC size to be managed more effectively with additional logic implemented in hardware or software.

**Multi-Threaded Analysis and Isolation Levels**

Multiple wraps may be open across a computer system at a given time in different threads of execution, and the variables used by the wraps might intersect. Multiple open, overlapping or shared wraps can lead to complicated situations in terms of possible overflow of the victim cache, increasing log sizes, and undesired isolation levels. In the current single-threaded evaluation, no safeguards are needed for sharing data between wraps.

Modeling a multi-threaded application for a hardware solution will require a significant change to the trace-based simulation discussed. A real-time approach with Pin or full system simulator might be required.

While traditional databases support various configurable degrees of isolation [17] from dirty to repeatable reads at the application level, the WrAP Architecture, in its current form, would only support dirty reads. Additional research is needed to support a configurable way to manage the access of variables manipulated both within the wrap and in other wraps managed by different threads. The isolation level could be specified when opening a wrap and by global options and defaults.
FPGA Module

Low power requirements for Storage Class Memories are on the order of 10 to 100 times more efficient than Flash technologies [9]. This makes SCM an ideal candidate for use in some embedded systems and applications with low power requirements that need to persist data. One possible solution to persist a group of data values atomically is to use the WrAP approach in such a system. In these systems, the SCM may not be placed on a memory bus, but might instead be accessible through another channel. Future investigations include creating a module that might be implemented in High Level Synthesis C using an approach similar to WrAP or SoftWrAP.

5.1.2 SoftWrAP

There are many areas for future research and optimizations for SoftWrAP. These include areas for extensible alias table implementations and optimizations, checkpointing, hardware testing, and integrations into the operating system and virtualization layers.

Alias Table Optimizations

- **Thread-Local Alias Table Enhancements.** The thread-local alias table presented copied values to SCM on a wrapClose. There are many optimizations that can be performed to hide the latency of direct writes to SCM. One possible implementation might, after copying values to the cache-hierarchy and performing a SFENCE, hand off the table to a retirement thread. The retirement thread would be similar to the retirement thread in the global alias table, except with lower overhead. This implementation could lead to even faster response times.
• **Multi-Threaded Double-Buffered Alias Table.** When different threads are concurrently writing to the same hash-table many things might happen. The hash-table might run out of space and need to be resized. New wraps need to be able to still acquire a handle to a hash-table for aliasing even if the current table is nearing full by concurrent wraps writing to it. Additionally, reading must be performed across multiple tables that would be in different states and might have open wraps. The double-buffered alias table therefore requires additional states to handle multiple concurrent threads. Ideally, minimal locking would be required while still supporting extensibility of the alias table.

• **Fine Grained Locking.**

![SoftWrAP alias table with fine-grained locking.](image)

One possible implementation of an extensible alias table would be to use a fine-grained locking approach as shown in Figure 5.1. This implementation would allow for extending the size of the table without stalling. It might also support concurrent retirement, allowing for a table to decrease in size while concurrently being accessed. This would require the flagging of values for removal and comparing of entries to retiring log values.
NUMA Enhancements

Moving from a Symmetric Multi-Processor (SMP) to a Non-Uniform Memory Access (NUMA) environment such as a multi-socket system poses many challenges. In such a system, there would no longer be a single memory bank that would handle both software aliasing in DRAM and logging and persistent storage of values in SCM. A task executing on a processor might be accessing an SCM location in another memory bank attached to another processor. Additionally, a wrap might access variables spread across disparate memory banks in a single wrap. Special care would need to be taken when considering where to save log entries in order to ensure system durability. Saving logs on a single SCM module that referenced different banks might be problematic. Additional considerations are needed in determining where to place alias table entries. Local aliasing might be preferred to global aliasing, but threads could end up migrating to other processors. A global alias table might have high overhead if accessed from other processors in a different socket.

Other Forward Work

- **Hardware Testing.** The availability of Storage Class Memory DIMMS is rapidly approaching. Everspin Technologies [23] has already announced a 64Mb DDR3 compatible DIMM in testing. Individual SCM chips are currently available for integration into custom applications. When DIMMS are available, quick integration into a testing environment and a performance analysis of SoftWrAP will be performed.

- **Full Database Integration.** SQLite provided a good environment for an initial database evaluation. However, SQLite is single-threaded and the inte-
gration with SQLite was performed as a plugin to the file-system layer. With multi-threaded alias table support, moving to a full multi-threaded database with integrations of SoftWrAP into the core code could achieve much higher performance results. The WrapPin tool discussed in Section 4.2.5 might be used to easily identify persistent areas to be wrapped for a chosen database.

- **Atomic Areas and Compiler Integration.** Identifying areas for atomic persistence can be challenging. Approaches such as ATLAS [32] provide ways of determining atomic areas by analyzing thread locks. Other approaches use compiler integration and keywords such as *atomic* for sections of code that must be performed atomically. Future work would be to include compiler support or support for automatically identifying atomic regions with SoftWrAP.

- **Virtualization and Operating System Integration.** As Storage Class Memories become a reality, support for obtaining handles and permission to persistence regions in virtualization layers and operating systems will be needed. Managing the access to the persistent regions will be challenging as it might require going through multiple software layers. Forward work for determining best locations for the SoftWrAP library will be considered, as it might be beneficial to move the SoftWrAP library into the operating system kernel.

5.2 Conclusions

This thesis extended and analyzed a software-hardware approach called WrAP, or Write-Aside Persistence, for atomically persisting data to Storage Class Memory. The approach uses a novel addition to the memory controller that allows for a fast path to data through the cache hierarchy while atomically persisting data through
a background path to SCM. Spurious cache evictions are handled by a Victim Persistence Cache on the backend of the cache-hierarchy, thereby having no up-front hardware changes. We showed promising performance results for WrAP that have significant speedup over traditional methods of atomic persistence, like copy-on-write or undo log approaches, and can approach the performance of a non-transactional method that only updates to cache.

This thesis also presents a software-only based approach to atomic persistence called SoftWrAP. The software approach relies on no new hardware additions aside those recently introduced by Intel. SoftWrAP also allows for a fast path of data values through the cache hierarchy while taking advantage of write-combining stores to a contiguous log area. SoftWrAP handles cache evictions by utilizing a DRAM based alias table, so that SCM values, which are accessed through the SoftWrAP library, are aliased to other volatile DRAM locations. If these variables are evicted from the cache-hierarchy to home locations, they can pose no harm to the persistent SCM values. The performance of SoftWrAP was shown to approach the speed of a non-atomic method of persistence, thereby gaining atomicity nearly for free. Alias table space was shown to be minimal in achieving high performance.

Both the software-hardware based WrAP architecture and the software-only SoftWrAP approaches were shown to be promising methods for atomic persistence support for Storage Class Memory. They provide atomicity and durability while simultaneously ensuring that fast paths through the cache, DRAM, and persistent memory layers are not slowed down by burdensome buffering or double-copying requirements.
Bibliography


