UNIVERSITY OF CALIFORNIA SAN DIEGO

Providing Easy to Use and Fast Programming Support for Non-Volatile Memories

A dissertation submitted in partial satisfaction of the
requirements for the degree
Doctor of Philosophy

in

Computer Science (Computer Engineering)

by

Amirsaman Memaripour

2019

DEDICATION

To those who have been there with me since the beginning of this journey.

And to those closest to me whose strength, kindness and patience

made this, at times, laborious path joyful and worthwhile.

EPIGRAPH

Home is behind, the world ahead,

and there are many paths to tread.

Through shadows to the edge of night,

until the stars are all alight.

Then world behind and home ahead,

we'll wander back and home to bed.

Mist and twilight, cloud and shade,

Away shall fade! Away shall fade!

— J. R. R. Tolkien

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I am grateful for the love and support of many incredible people who helped me throughout the long journey that led to writing this dissertation.

Writing this dissertation would not have been possible without the guidance of my adviser, Steven Swanson. I am especially thankful for his patience and kind support. His mentorship has helped me to grow academically and advance towards my career goals. I am grateful for numerous opportunities that came my way as a result of his hard work and dedication to nurturing students like me.

I want to express my appreciation to my committee members, Geoffrey Voelker, Dean Tullsen, Ranjit Jhala, and Farinaz Koushanfar, for their constructive feedback and guidance. Your suggestions have been an invaluable resource throughout this process.

I want to thank my colleagues at the Non-Volatile Systems Laboratory (NVSL). This dissertation is the product of hard work by many individuals, and their share in its success is no less than mine. I am thankful to Meenakshi Sundaram Bhaskaran and Michael Wei for their advise and feedback through the early years of my Ph.D. I am also grateful to Joseph Izraelevitz for his help and support on the Pronto and NVHooks projects. I wish to also thank Jian Xu, Jian Yang, Yiying Zhang, Lu Zhang, Juno Kim, Yi Xu, and the rest of the NVSL.

I wish to also express my gratitude to many wonderful people at UC San Diego, Microsoft Research, and Micron Technology. I mention a few as I find it difficult to name them all, but please know you are all immensely appreciated. Thanks to Anirudh Badam, Amar Phanishayee, and Karin Strauss for their mentorship and support during my time at Microsoft Research. Thanks to Ameen Akel, Ken Curewitz, Sean Eilert and many others at Micron Technology for their mentorship and friendship.

Finally, I want to express my great appreciation to my family and friends for sticking by my side regardless of the ups and downs. I am forever indebted to my family for their unconditional love and support. I would not have accomplished anything if it wasn't for their sacrifices. A very special word of thanks goes to Mia, who has been kind, patient, understanding, and supportive throughout the toughest days of this journey. I wish to also thank Hana for her genuine friendship over the years. Thank you, everyone, for believing in me!

Chapter 1, Chapter 2, and Chapter 3 contain material from "Breeze: User-Level Access to Non-Volatile Main Memories for Legacy Software", by Amirsaman Memaripour and Steven Swanson, which appeared in the 36th International Conference on Computer Design (ICCD 2018). The dissertation author is the primary investigator and first author of this paper. The materials are copyright © 2018 by Institute of Electrical and Electronics Engineers (IEEE).

Chapter 1, Chapter 2, and Chapter 4 contain material from "NVHooks: A Flexible, Optimizing Compiler for Non-Volatile Memory Programming", by Amirsaman Memaripour, Yi Xu, Steven Swanson, and Joseph Izraelevitz which is submitted to the 25th Architectural Support for Programming Languages and Operating Systems (ASPLOS 2020). The dissertation author is the primary investigator and first author of this paper.

Chapter 1, Chapter 2, and Chapter 5 contain material from "Pronto: Easy and Fast Persistence for Volatile Data Structures", by Amirsaman Memaripour, Joseph Izraelevitz, and Steven Swanson which is submitted to the 25th Architectural Support for Programming Languages and Operating Systems (ASPLOS 2020). The dissertation author is the primary investigator and first author of this paper.

VITA

2010	Bachelor of Computer Engineering, Shahid Beheshti University
2012	Master of Computer Engineering, Iran University of Science and Technology
2013-2019	Graduate Student Researcher, University of California San Diego
2014	Internship, Micron Technology
2015	Internship, Microsoft Research
2016	Internship, Microsoft Research
2017	Candidate of Philosophy, University of California San Diego
2017	Internship, Microsoft Research
2017	Technology Management and Entrepreneurism Fellowship Program, Rady School of Management
2019	Doctor of Philosophy in Computer Science (Computer Engineering), University of California San Diego

PUBLICATIONS

Jian Xu, Juno Kim, Amirsaman Memaripour and Steven Swanson, "Finding and Fixing Performance Pathologies in Persistent Memory Software Stacks", In the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2019.

Amirsaman Memaripour and Steven Swanson, "Breeze: User-Level Access to Non-Volatile Main Memories for Legacy Software", In the 36th International Conference on Computer Design (ICCD), 2018.

Daehyeok Kim, Amirsaman Memaripour, Anirudh Badam, Yibo Zhu, Hongqiang Harry Liu, Jitu Padhye, Shachar Raindel, Steven Swanson, Vyas Sekar and Srinivasan Seshan, "Hyperloop: Group-Based NIC-Offloading to Accelerate Replicated Transactions in Multi-Tenant Storage Systems", In the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM), 2018.

Jian Xu, Lu Zhang, Amirsaman Memaripour, Akshatha Gangadharaiah, Amit Borase, Tamires Brito Da Silva, Steven Swanson and Andy Rudoff, “NOVA-Fortis: A Fault-Tolerant Non-Volatile Main Memory File System”, In the 26th Symposium on Operating Systems Principles (SOSP), 2017.

Amirsaman Memaripour, Anirudh Badam, Amar Phanishayee, Yanqi Zhou, Ramnatthan Alagappan, Karin Strauss and Steven Swanson, “Atomic In-Place Updates for Non-Volatile Main Memories with Kamino-Tx”, In the 12th European Conference on Computer Systems (EuroSys), 2017.

Yiying Zhang, Jian Yang, Amirsaman Memaripour and Steven Swanson, “Mojim: A Reliable and Highly-Available Non-Volatile Memory System”, In the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2015.

ABSTRACT OF THE DISSERTATION

Providing Easy to Use and Fast Programming Support for Non-Volatile Memories

by

Amirsaman Memaripour

Doctor of Philosophy in Computer Science (Computer Engineering)

University of California San Diego, 2019

Professor Steven Swanson, Chair

Non-Volatile Memory (NVM) technologies, such as 3D XPoint, offer DRAM-like performance and byte-addressable access to persistent data. NVMs promise an opportunity for fast, persistent data structures, and a wide range of applications stand to benefit from the performance potential of these technologies. These potential benefits are greatest when applications access NVM directly via load/store instructions rather than conventional file-based interfaces. Directly accessing NVM presents several challenges. In particular, applications need guaranteed consistency and safety semantics to protect their data structures in the face of system failures and programming errors.

Implementing data structures that meet these requirements is challenging and error-prone.

Existing methods for building persistent data structures require either in-depth code changes to an existing data structure or rewriting the data structure from scratch. Unfortunately, both of these methods are labor-intensive and error-prone.

Failure-atomicity libraries and programming language extensions can simplify this task. However, all the proposed solutions either require pervasive changes to existing software or incur unacceptable overheads to runtime performance. As a result, porting legacy applications to leverage NVM is likely to be prohibitively difficult and time-consuming.

This dissertation first presents Breeze, an NVM toolchain that minimizes the changes necessary to enable legacy code to reap the benefits of directly accessing NVM. In contrast to PMDK and NVM-Direct, Breeze reduces the programming effort of porting Memcached and MongoDB by up to $2.8\times$, while providing equal or superior performance.

Second, it introduces NVHooks, a compiler that automatically annotates NVM accesses and avoids disruptive and error-prone changes to programs. NVHooks reduces the cost of these annotations by applying novel, NVM-specific optimizations to their placement. For our tested benchmarks, NVHooks matches the performance of hand-annotated code while minimizing programmer effort.

Finally, it presents Pronto, a new NVM library that reduces the programming effort required to add persistence to volatile data structures. Pronto uses asynchronous semantic logging (ASL) to allow adding persistence to the existing volatile data structure (e.g., C++ Standard Template Library containers) with minor programming effort. ASL moves most durability code off the critical path. Our evaluation shows Pronto data structures outperform highly-optimized NVM data structures by a large margin.

Chapter 1

Introduction

For decades, computer architects have been in the pursuit of a fast, persistent, cost-effective storage technology that would fill the gap between the volatile memory hierarchy (e.g., CPU caches and memory) and the persistent storage, mainly comprising of hard-disk and solid-state drives. Non-Volatile Memory (NVM) technologies provide the means to fill this gap by offering comparable latency and bandwidth to volatile memory (e.g., DRAM) while delivering persistence, higher density, and lower cost per gigabyte.

NVMs expose user applications to low-cost persistence, essentially invalidating many assumptions concerning the storage hierarchy. In particular, NVMs are byte-addressable and can sit beside DRAM on the memory bus, allowing applications to avoid expensive system calls and directly access persistent data. As a result, programmers are no longer required to serialize data before sending it for persistence or deserialize it after reading the data back. Furthermore, writing to NVMs is orders of magnitude faster than solid-state drives and only takes a few microseconds to complete, while the performance of reading is almost equivalent to that of DRAM. The performance boost that NVMs offer obviates many buffering and grouping techniques (e.g., group commit) vastly used in many popular storage and database applications.

By directly accessing NVMs and bypassing the storage stack, programmers can take the most advantage of the performance potential of these memories. Direct access allows applications to issue load and store instructions to access persistent data; however, it introduces other challenges. In particular, programmers must carefully construct persistent data structures to ensure the durability and consistency of persistent data in the face of programming errors and system failures (e.g., power outages).

Due to the volatility of CPU caches, writes to persistent memory are not immediately durable unless flushed from the volatile caches (e.g., using cache-flush instructions). As discussed in [9] and [16], flushing cache-lines is an expensive operation, and applications often postpone such flushes to mitigate the performance penalties. Delaying flushes, however, could result in unfortunate outcomes; a power outage, for instance, could tear writes to persistent memory, resulting in partially updated data structures. Computer scientists have proposed several approaches to either reduce the cost of these cache-flushes or make them unnecessary [16, 84]. To date, persistent caches are yet to be adopted by the processor manufacturers, while NVM-optimized cache-flush and memory-fence instructions are the only tools available to programmers to construct persistent data structures for NVMs [47].

Providing programmers with direct access to NVM also hardens ensuring the consistency of persistent data in the wake of a restart. Existing hardware only offers support for atomically updating 64 bit NVM regions. Although the high-end Intel processors have well-defined constraints on the order in which they flush words of a cache-line, no commercial processor today provides support for arbitrary sized atomic writes to NVM. To ensure writes to persistent memory are atomic with respect to restarts (i.e., failure-atomic), programmers must adopt these hardware primitives to build failure recovery mechanisms (e.g., undo-logging or copy-on-write), a delicate and error-prone task. Furthermore, programmers must take special care in allocating and referencing persistent memory to avoid prevalent programming errors such as memory leaks and dangling pointers. These challenges stifle NVM programming and impede the adoption of these emerging technologies.

Failure-atomicity libraries (e.g., PMDK [44]) aim to reduce the challenges of building programs for NVMs by hiding the complexity and limitations of the hardware primitives under a set of flexible programming interfaces. To avoid the pitfalls of direct access, these libraries provide support for arbitrary sized failure-atomic updates to persistent data and an efficient transactional NVM allocator that prevents permanent memory leaks in the face of system failures. They also offer persistent naming to allow access to persistent data across programs' restarts and support for remapping persistent pools after recovery.

These libraries, unfortunately, introduce new issues that impede the adoption of persistent programming. In particular, they often require programmers to annotate accesses to NVM, specify the boundaries of failure-atomic operations, and use library-specific APIs to manage persistent memory and write recovery code. These annotations require disruptive changes to existing programs, especially those with no persistence semantics, and are often misplaced or overused.

Failure-atomicity libraries require programmers to annotate accesses to persistent memory, and these annotations are cumbersome, introduce new programming errors, and disrupt code reusability. Libraries use these annotations to intercept NVM accesses, identify boundaries of failure-atomic operations, and manage persistent memory using NVM allocators. Annotating NVM accesses, in particular, require altering significant portions of the source code, a laborious and error-prone task.

Furthermore, programmers often misuse these annotations, by either missing to annotate some of the NVM accesses or unnecessarily annotating others. The challenges of using annotations are prevalent enough that has motivated researchers to build NVM debugging tools [65, 53, 41]; however, these tools often fail to identify all instances of unnecessary or missed annotations. Note that missing to annotate a single write to NVM could result in recovering to an inconsistent state after a failure.

These libraries offer library-specific annotations and vary in their support for concurrent, failure-atomic writes to NVM. The differences in the syntax and semantics of these libraries impede retargeting programs to different NVM libraries and inhibit constructing benchmarks that can evaluate multiple libraries. Chapter 2 provides more background on non-volatile memory technologies and studies the challenges of building programs for byte-addressable NVMs in more detail. It also motivates our research on reducing the programming effort of constructing applications for emerging NVM technologies.

This dissertation focuses on addressing the challenges of programming with NVMs and provides a collection of tools and techniques to facilitate building programs that access persistent memory via load/store instructions. These tools enable programmers to transactionally update persistent data while avoiding memory leaks and dangling pointers. Furthermore, they reduce the programming effort of constructing persistent applications with no or minor performance penalty through a series of NVM-specific compilation and optimization passes. Finally, the dissertation introduces a set of techniques that allow adding persistence to volatile, concurrent data structures with only a few lines of code.

In Chapter 3, we introduce Breeze that includes a user-level library as well as a compiler; it offers programmers with transactional access to NVM and reduces the changes to the source code required to enable failure-atomic, direct-access to NVM. Breeze ensures consistency of persistent data in the wake of a restart and reduces changes to the source code by automatically generating code for failure recovery, referential integrity, and garbage collection. Our experiments with both microbenchmarks and real-world applications (e.g., MongoDB [77]) show that Breeze reduces the programming cost of transforming existing programs to NVM-enabled versions. Furthermore, Breeze meets or exceeds the performance of other failure-atomicity libraries when adopted with our benchmark applications.

Next, we present NVHooks in Chapter 4. NVHooks minimizes the need for manual annotation of NVM programs. It automates the annotation of NVM accesses, allows failure-atomicity libraries to intercept those accesses without involving the programmer, and reduces the cost of retargeting NVM programs to new failure-atomicity libraries. Furthermore, NVHooks offers a set of NVM-specific optimization passes that leverage the semantics of NVM annotations to minimize their performance overhead.

We introduce Pronto in Chapter 5, an NVM library that uses asynchronous semantic logging to transform operations on volatile data structures into failure-atomic operations. Instead of recording the details of how updates change persistent data structures, a semantic log records the arguments and completion order of the operations. Pronto plays back semantic logs after a restart to reconstructs the latest consistent state of persistent data structures. It also creates periodic snapshots to limit the overhead of replaying semantic logs during recovery. Our evaluations show that Pronto is easily adoptable by many popular, volatile data structure implementations (e.g., STL containers). Moreover, the resulting persistent data structures provide comparable performance to their volatile counterparts, while outperforming other failure-atomic variants.

Finally, we conclude this dissertation in Chapter 6 by summarizing its contributions, including a user-level library to facilitate adding failure-atomicity to legacy software, a series of NVM-specific compilation and optimization passes to reduce the programming effort of constructing persistent programs, and a new NVM library to transform concurrent, volatile data structures into failure-atomic ones.

Acknowledgments

This chapter contains material from “Breeze: User-Level Access to Non-Volatile Main Memories for Legacy Software”, by Amirsaman Memaripour and Steven Swanson, which appeared in the 36th International Conference on Computer Design (ICCD 2018). The dissertation author is the primary investigator and first author of this paper.

This chapter contains material from “NVHooks: A Flexible, Optimizing Compiler for Non-Volatile Memory Programming”, by Amirsaman Memaripour, Yi Xu, Steven Swanson, and Joseph Izraelevitz which is submitted to the 25th Architectural Support for Programming

Languages and Operating Systems (ASPLOS 2020). The dissertation author is the primary investigator and first author of this paper.

This chapter contains material from “Pronto: Easy and Fast Persistence for Volatile Data Structures”, by Amirsaman Memaripour, Joseph Izraelevitz, and Steven Swanson which is submitted to the 25th Architectural Support for Programming Languages and Operating Systems (ASPLOS 2020). The dissertation author is the primary investigator and first author of this paper.

Chapter 2

Background and Motivation

NVMs have introduced a new possibility for designing storage systems: Programs can have byte-addressable access to terabytes of persistent data at DRAM-like performance. In contrast to hybrid solutions such as battery-backed DRAM, NVMs (e.g., Intel DC Persistent Memory) offer higher capacity, lower cost per gigabyte, and marginally lower performance relative to DRAM. A recent study shows NVMs can deliver 76% of DRAM performance when running WHISPER, a benchmark suite comprising a series of persistent micro-benchmarks and applications [78, 51].

Several NVM technologies are available today or are expected to appear in the market in the next few years. These technologies present significant challenges to programmers, and researchers have proposed several systems to simplify programming for NVMs.

The rest of this chapter provides the background of this thesis. Section 2.1 reviews the basics of non-volatile memory technologies. Section 2.2 and Section 2.3 introduce the challenges of non-volatile memory programming and summarize the contributions of failure-atomicity libraries in facilitating NVM programming, respectively. Section 2.4 highlights the limitations of failure-atomicity libraries. Finally, Section 2.5 presents the motivation for NVM-specific compiler support and optimizations.

2.1 Non-Volatile Memories

NVMs promise to fill the gap between volatile memory and disks (both hard and solid-state) by offering byte-addressability, DRAM-like latency and bandwidth, and persistence [6, 89]. NVMs based on battery- or flash-backed DRAM [74] have been available for many years, and cheaper main memory modules based on 3D XPoint [73, 43] have entered the market recently [75, 48]. These emerging NVMs offer higher density, higher latency, and lower bandwidth [68, 13] than DRAM-based devices. Thus, we anticipate hybrid memory systems with both DRAM and NVM.

NVMs are fast enough to sit on the processor's memory bus [4], providing software with direct access to NVM via load/store instructions. Programmers must bypass the storage stack (e.g., file systems) and directly access these devices to avoid the software overhead and fully exploit the performance benefits these devices offer [89]. Using the load/store interface exposes the full performance of NVMs, but it introduces a different set of challenges.

Since CPU caches are volatile, stores to non-volatile memory do not become durable until the cache writes back the affected data. Cache evictions are usually transparent to software, so programmers must use cache flush instructions to trigger write-backs and memory barriers to wait for the write-backs to complete [89, 3, 46]. Cache-flushes and memory barriers are necessary building blocks, but they do not suffice for providing the failure-atomicity that applications need to make use of NVM.

In general, for a user application to access NVM, it maps a corresponding NVM-resident file, also known as the *persistent pool*, to a contiguous region in the program's virtual address space (e.g., via `mmap()`). In order to avoid collisions, ideally the persistent pool can be mapped at an arbitrary virtual address. As the virtual address of this mapping may change across program restarts, the virtual addresses of objects inside the mapped region are susceptible to change and not immutable. Thus, pointers within the pool could become invalid after a restart — relative pointers are a common solution [86].

Programs can use store instructions to persistently write to NVMs, but the store is not persistent so long as it resides in the volatile cache. Moreover, stores need not reach NVMs in program execution order due to the reordering by the cache write-back policy. So, programmers must use special instructions to enforce the correct persist ordering for NVM writes.

Cache-flush and memory-barrier instructions allow programmers to control persistence and ordering of NVM writes. For example, on Intel CPUs, a `clflush` and `sfence` must follow an NVM store to ensure its persistence, and stores are failure-atomic at the 8-byte granularity. Programmers and failure-atomicity library developers can use these hardware primitives to write software that provides more complex and custom persistence semantics.

Ordering writes into NVM is important. In a persistent stack, for example, the newly inserted element through `push()` must become persistent before the head pointer, which represents the top of the stack, points to it. If this order is violated, an inopportune crash could result in the head pointer pointing to uninitiated memory or partially written data.

2.2 Programming with Non-Volatile Memories

While atomic writes and cache control instructions are sufficient, in principle, to unlock NVM's benefits, building complex, fault-tolerant, and highly concurrent data structures using those primitives is very challenging. In particular, programmers must address all the challenges that volatile data structures present, such as memory management and locking. Both of these areas are well-known sources of bugs and the resulting inconsistencies in the data structures will be permanent with NVM. Programmers must also reason carefully about the order in which updates occur and which updates may or may not be visible to other threads and after a failure. In this respect, building persistent data structures resembles building lock-free data structures, a notoriously tricky, subtle, and error-prone discipline.

NVM also introduces new classes of bugs that are at least as pernicious as memory management and locking errors. For instance, pointers from a non-volatile data structure to volatile memory are inherently unsafe, as are pointers between two independent NVM regions (e.g., two mmap'd files) [18]. Finally, NVM complicates locking, since the programmer or the system must ensure that all locks are released after a system failure.

The rest of this section reviews the challenges of NVM programming and highlights the performance and programming cost of constructing persistent applications.

2.2.1 Programming Cost

Bypassing the filesystem to directly access NVM via load/store instructions lets programmers fully exploit the performance benefits of NVMs, but it introduces a series of challenges. Programs could lose part of an update to a persistent data structure during a system failure (e.g., power loss) because existing hardware does not support flushing multiple cache lines atomically: an ill-timed failure could cause permanent data inconsistency.

To avoid this issue, persistent data structures must be able to recover to a consistent state after a crash. NVM transaction libraries [69, 18, 95] embody the most common approach to ensure failure-atomicity of updates to a persistent data structure: fine-grain logging of how the data structure changes. Unfortunately, annotating existing data structures with these libraries is labor-intensive and error-prone as programmers are required both to annotate every persistent data update and reason about failure-atomic update boundaries. As an example, we had to rewrite almost all of a volatile B+Tree to make it persistent using Intel's popular Persistent Memory Development Kit (PMDK) library [44]. For more complicated data structures in use today (e.g., those in the C++ Standard Template Library), adding all these annotations without error would be an extremely invasive change to the code base that is already very complex and highly-optimized for volatile operation. Indeed, the difficulty of correctly adding annotations has spawned research into new debugging tools for finding these errors [65, 53, 41, 100].

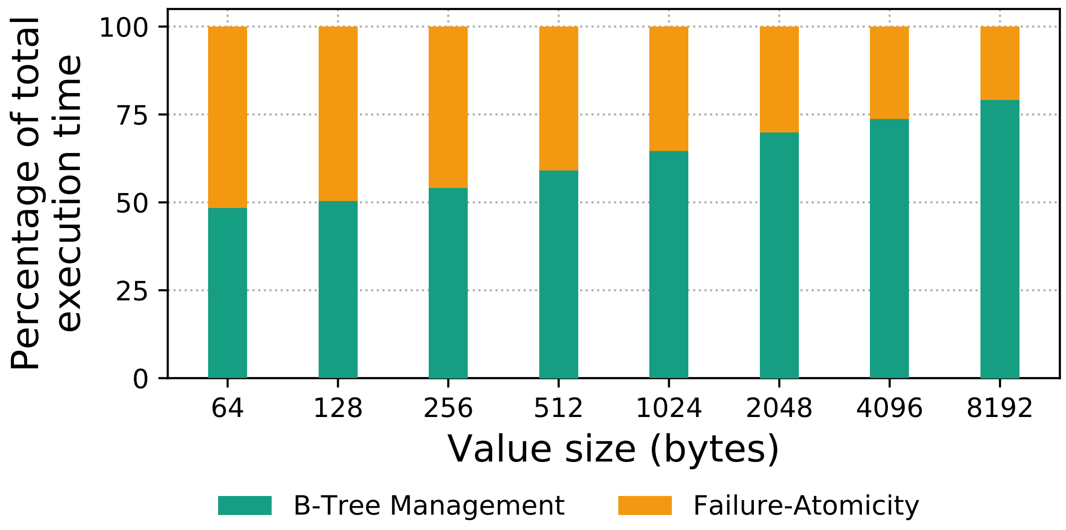


Figure 2.1: Latency breakdown of inserts to PMEMKV

2.2.2 Performance Cost

The cost of enforcing failure-atomic updates for NVM data structures is large. Logging for failure-atomicity libraries adds overhead in the form of stores to transaction metadata, additional cache-flushes, and memory barriers [102, 15]. Moreover, the cost of fine-grain logging scales with the complexity of persistent data structures. Logging also limits the processor's ability to reorder instructions [84], further hurting performance.

To explore this cost, we measured it in PMEMKV [45], a persistent key-value store that uses a B-Tree and stores its last level in NVM [99]. PMEMKV uses the transaction facility in PMDK [44] to transactionally update the B-Tree.

We instrumented PMDK and PMEMKV to gather detailed latency numbers for inserting one million key-value pairs to PMEMKV using traces from YCSB [21]. Figure 2.1 reports the relative latency of managing the B-Tree data structure and ensuring its failure-atomicity (e.g., logging, persistent allocation, and transaction management) for value sizes ranging from 64 to 8192 bytes. Failure-atomicity increases the latency of insert operations by 26% to 106%.

Conventional NVM transaction libraries put all the overhead of ensuring failure-atomicity

Table 2.1: **Concurrency constraints of NVM libraries:** These libraries vary in their support for concurrent NVM access. Programs must comply with the concurrency constraints of a failure-atomicity library to use it.

NVM Library	Locking Scheme	Allows Persistent Writes outside Txs	Atomics Allowed
PMDK [86]	2PL	No	No
Pangolin [100]	2PL	No	No
Kamino-Tx [69]	2PL	No	No
Atlas [14]	FASE	Yes	No
JUSTDO [49]	FASE	No	Yes
iDO [63]	FASE	No	Yes
NVthreads [39]	FASE	No	No

(e.g., logging, cache-flushes, and barriers) on the critical path, so applications bear the full cost.

Next, we modified PMEMKV to disable transaction management and logging. The modified version of PMEMKV does not ensure the durability and consistency of updates to the NVM-resident data, but still adopts PMDK's persistent memory allocator for managing the last level of the B-Tree. Comparing the throughput of the modified and original versions of PMEMKV lets us estimate the performance boost that we can achieve by moving logging and transaction management off the critical path. We observe that the modified version (with no logging and transaction management) runs twice as fast.

2.3 Failure-Atomicity Libraries

NVM libraries aim to mitigate the challenges of direct access, but they introduce new issues that stifle persistent programming. They facilitate NVM programming by providing an interface to name and allocate persistent memory, run failure-atomic operations, and avoid dangling pointers (e.g., via swizzling persistent pointers).

It is common among NVM libraries to provide custom, library-specific APIs and annotations. For example, PMDK [86] requires programmers to annotate writes to NVM with callbacks to the library, NVthreads [39] relies on the page table to detect writes, and vNVML [17] requires using its `nv_write()` API for all NVM writes. Despite the differences in the interface, they must address the challenges of direct access and provide the following:

- Support for arbitrary sized, failure-atomic updates to persistent data.
- An efficient, transactional memory allocator that avoids permanent memory leaks during system failures.
- Access to persistent data across programs restarts (e.g., by offering persistent naming).
- Support for remapping a persistent pool after restarts, preferably at an arbitrary virtual address to allow persistent heap relocation.

NV-Heaps [18], Mnemosyne [95], and NVM-Direct [12], are examples of these libraries that provide a similar set of core facilities.

First, they provide memory allocation and garbage collection mechanisms that are robust in the face of system failures. These eliminate memory leaks and dangling pointers. Second, NV-Heaps and NVM-Direct provide protections against creating unsafe pointers from non-volatile memory to volatile memory and between independent regions of non-volatile memory. Third, they provide atomic sections that let the programmer specify which operations move a persistent data structure from one consistent state to another, providing a more flexible atomicity primitive than the 64-bit stores that hardware provides natively. The atomic sections also provide a natural replacement for conventional locks and avoid a wide range of locking-based bugs.

All these systems also place strong constraints on how programmers write code. For instance, NV-heaps is a C++ library and requires that all objects inherit from a persistent object base class, and NVM-Direct adds syntax to C in order to distinguish between volatile and non-volatile pointers. These constraints are tolerable for developers writing new code, but it makes adapting existing code to use NVM very labor intensive. This is unfortunate, because there are many legacy applications that could benefit from NVM. These include applications like MongoDB that use memory-mapped files as their persistent data store and applications like Memcached that use volatile data structures but could benefit from making those structures persistent.

These libraries also vary in their support for concurrent, failure-atomic writes to NVM and constrain the synchronization semantics of multithreaded programs. These constraints include the adopted locking scheme, application of atomic instructions on NVM, and updates to persistent data outside transaction boundaries. Table 2.1 summarizes these constraints. "2PL" stands for two-phase locking: the program cannot acquire a new lock after releasing any lock unless all locks are released [11]. A "FASE" (Failure-Atomic SEction) requires the program to never modify NVM without acquiring a lock, but the order of acquiring and releasing locks is not important. A data-structure, for instance, that writes to NVM outside critical sections (i.e., without holding any locks) could adopt Atlas, but it does not qualify to use PMDK.

The library-specific annotations and the difference in persistence semantics prevent programmers from retargeting programs to different NVM libraries. Furthermore, it inhibits constructing benchmarks that work with the majority of these libraries to allow programmers to compare their performance.

2.4 Challenges of Annotations

NVM libraries rely on programmers to annotate programs. These annotations are cumbersome, make programs error-prone, and can suppress code reusability. They enable the libraries to intercept NVM accesses, identify failure-atomic operations, and use persistent allocators to manage NVM. In contrast to the application of library-specific APIs (e.g., to specify the boundaries of failure-atomic operations) that only requires changing small parts of programs, annotating NVM accesses requires rewriting significant portions of the code. For instance, we had to write or modify 720 lines of code to annotate a 629-line volatile B+Tree for PMDK.

```
1  struct Node {
2    int version;
3    char key[32];
4    TOID(struct Node) next;
5  };
6  void update(char *oldKey, char *newKey) {
7    TOID(struct Node) node = get_list_head();
8    while (strcmp(D_RO(node)->key, oldKey)){
9      node = D_RO(node)->next;
10   }
11   if (D_RO(node) == NULL) return;
12   TX_ADD_FIELD(node, key);
13   memset(D_RW(node)->key, 0, 32);
14   D_RW(node)->version++;
15   TX_ADD_FIELD(node, key);
16   strcpy(D_RW(node)->key, newKey);
17 }
```

Figure 2.2: **An example of annotating a volatile linked-list for PMDK:** `TOID` creates a relative pointer of the specified type. `D_RO` and `D_RW` swizzle a pointer and return read-only and read-write pointers of the same type, respectively. `TX_ADD_FIELD` creates an undo-log of the specified field.

2.4.1 Programming Effort

Annotating NVM accesses, in particular, imposes significant changes to programs. These annotations enable libraries to augment NVM accesses with additional logic, such as swizzling pointers and logging.

Figure 2.2 shows an example of these annotations for PMDK. The code is a portion of the update method for a persistent linked-list. PMDK uses the annotations, which are highlighted, to swizzle pointers and create undo-logs for NVM writes.

2.4.2 Correctness

It is easy to misuse these annotations. Programmers could either miss annotating some of the NVM accesses or unnecessarily annotate others. Figure 2.2 shows an example of both scenarios. Line 14 is an example of a missed annotation, which could result in data corruption.

Line 15 shows an instance of over-annotation, where the programmer annotates an NVM write that they already annotated at line 12.

The difficulty of using annotations has motivated research on NVM debugging tools [65, 53, 41]; however, these tools often fall short in identifying all unnecessary or missed annotations.

2.4.3 Code Reusability

Annotations also impede using pre-compiled, third-party libraries. Programmers must ensure every call into a third-party function accompanies the necessary annotations that describe how the function interacts with NVM.

Line 12 of Figure 2.2 provides an example of such annotations for `memset()`, where `TX_ADD_FIELD()` instructs PMDK to create an undo-log of the `key` attribute. So long as these annotations provide an accurate description of how functions of the third-party library access persistent memory, it is safe for programmers to use them. Annotations can satisfy this requirement for many library functions commonly used by programmers (e.g., `strcpy()` and `memset()`).

Existing work offers an alternative to annotating calls into third-party libraries by using hardware support (e.g., page faults to detect NVM writes [39, 83]). However, these solutions impose considerable performance cost.

2.5 Absence of Compiler Support

Existing compilers treat NVM annotations the same as other callbacks into third-party libraries. Not exposing the semantics of these annotations to the compiler limits its ability to optimize them. Figure 2.3 shows how the compiler can use the semantics of NVM annotations to optimize the code. Once the compiler understands the semantics of the pointer swizzling function (i.e., `D_RO()`), it can reuse the swizzled pointer so long as its corresponding relative pointer does not change. In contrast to the original code, the optimized version requires up to 400% fewer calls to the swizzling function.

Acknowledgments

This chapter contains material from “Breeze: User-Level Access to Non-Volatile Main Memories for Legacy Software”, by Amirsaman Memaripour and Steven Swanson, which appeared in the 36th International Conference on Computer Design (ICCD 2018). The dissertation author is the primary investigator and first author of this paper.

This chapter contains material from “NVHooks: A Flexible, Optimizing Compiler for Non-Volatile Memory Programming”, by Amirsaman Memaripour, Yi Xu, Steven Swanson, and Joseph Izraelevitz which is submitted to the 25th Architectural Support for Programming Languages and Operating Systems (ASPLOS 2020). The dissertation author is the primary investigator and first author of this paper.

This chapter contains material from “Pronto: Easy and Fast Persistence for Volatile Data Structures”, by Amirsaman Memaripour, Joseph Izraelevitz, and Steven Swanson which is submitted to the 25th Architectural Support for Programming Languages and Operating Systems (ASPLOS 2020). The dissertation author is the primary investigator and first author of this paper.

```
bool list_contains_duplicate_keys() {
  TOID(struct Node) n = get_list_head();
  // We can reuse D_RO(x) so long as the value of x does not change.
  while ( D_RO(n) ) {
    TOID(struct Node) m = D_RO(n) ->next;
    while ( D_RO(m) ) {
      if (strcmp( D_RO(n) ->key, D_RO(m) ->key) == 0) {
        return true;
      }
      m = D_RO(m) ->next;
    }
    n = D_RO(n) ->next;
  }
  return false;
}
```

```
bool list_contains_duplicate_keys() {
  TOID(struct Node) n = get_list_head();
  const struct Node *np = D_RO(n) ;
  while (np) {
    TOID(struct Node) m = np->next;
    const struct Node *mp = D_RO(m) ;
    while (mp) {
      if (strcmp(np->key, mp->key) == 0) {
        return true;
      }
      mp = D_RO(mp->next) ;
    }
    np = D_RO(np->next) ;
  }
  return false;
}
```

Figure 2.3: **Compilers can significantly reduce the overhead of NVM libraries by optimizing the annotations (highlighted).** This example extends the code from Figure 2.2 with a new function. The code on the top and the bottom are the original and the optimized, respectively.

Chapter 3

User-Level Access to Non-Volatile Memories for Legacy Software

Emerging non-volatile main memory (NVM) technologies, such as Intel and Micron's 3D XPoint [73, 43], offer DRAM-like performance but with higher density and lower cost-per-bit. Applicaitons such as databases and key-value stores could avoid serialization costs and improve response times by leveraging the performance, fine-grain access, and persistence that these memories offer.

Although NVMs promise orders of magnitude better performance compared to conventional hard and solid state disks, unleashing their potential is challenging. To maximize the performance benefits of NVM, applications should access them directly via load/store instructions rather than through conventional file-based interfaces. This requires programmers to construct persistent data structures that are resiliant in the face of system failures, avoid (persistent) memory leaks, prevent the creation of dangling pointers, and support multi-threaded operations.

Building these data structures is hard because it requires careful reasoning about the order in which updates to NVMs will become persistent. Since caches will remain volatile and processors can reorder stores, programmers must issue barriers and/or flush cache lines to enforce the order in which updates become persistent to ensure data consistency. Applying these barriers correctly is challenging: excessive use leads to degraded performance [9, 99], but missing barriers can lead to data corruption.

Researchers have proposed several programming systems [18, 95, 12, 44] to hide this complexity from the programmer by providing library- or language-based mechanisms to manage and allocate NVM, define persistent data structures, and specify the atomic operations that transform those structures from one consistent state to another.

While these systems provide comprehensive solutions to many of the challenges that NVM programming presents, they offer limited support for porting existing programs and data structures to make use of NVM. Applying them to existing codes require pervasive changes and enormous programmer effort.

External libraries pose a particular problem, since, in the near term, they are unlikely to have been built with persistent memory in mind. Given these challenges, and without an alternative solution, it is likely that most legacy code will not fully benefit from the performance that NVM offers. This will, in turn, reduce the rate of adoption of NVM.

To address this problem, we propose Breeze, a toolchain that provides programs with transactional access to NVM and minimizes disruptive changes to the source code. The toolchain includes a user-level NVM library and a C compiler. Breeze minimizes changes to the source code by transparently providing failure recovery, referential integrity (i.e., ensuring that references point to valid data) and garbage collection. In contrast to existing systems for low-level languages (e.g., C), Breeze does not require programmers to explicitly create log entries before touching persistent objects or use specific pointer types to reference persistent memory. Also, it can recover leaked memory regions without help from the programmer. The toolchain offers a set of primitives to manage and update persistent memory regions and guarantees consistency of persistent data in the event of failures.

Breeze makes the following contributions:

- The Breeze compiler automatically generates log entries for non-volatile data updates.
- It lets programs use normal pointers to refer to perisstent data rather than “fat” or “swizzled” pointers.
- It supports automatic garbage collection for persistent memory in C.
- It can generate log entries for changes that many third-party functions make to persistent memory.

We evaluate Breeze in terms of how extensively a programmer must modify a code base to make use of NVM and the performance it offers. To quantify this, we port two applications, MongoDB [77] and Memcached [31], as well as a B+Tree and a hash table to use Breeze. We find that Breeze requires fewer, simpler changes to the source code and meets or exceeds the performance of competing systems like NVM-Direct and PMDK.

The remainder of this chapter is organized as follows. Section 3.1 presents the architecture of Breeze while Section 3.2 describes its implementation in detail. Next, we evaluate ease of use and performance of Breeze in Section 3.3. Finally, Section 3.4 discusses related work and Section 3.5 provides a summary of this chapter.

3.1 Design Overview

Breeze provides existing programs with direct access to NVM and maintains consistency of persistent data while minimizing changes to the source code. The toolchain exposes a small set of interfaces to manage and update persistent memory regions transactionally. Programmers employ these interfaces to create/open memory-mapped files, define persistent data structures, specify persistent pointers, and transactionally create and modify instances of those structures.

Breeze exploits the definition of persistent structures and pointers to generate a set of C functions that will maintain reference counts and the referential integrity of persistent objects at

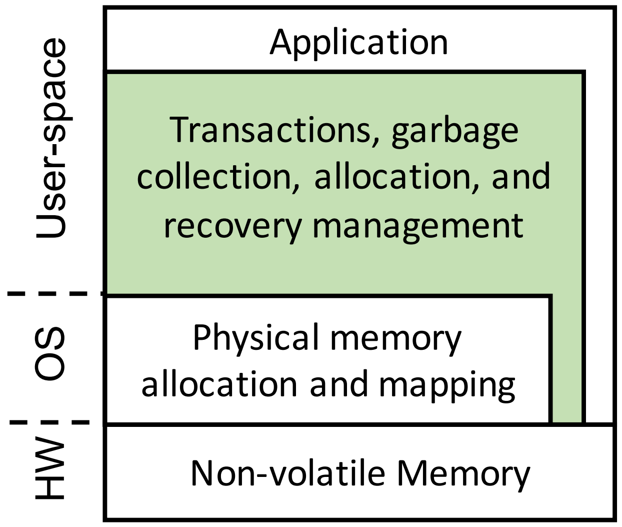

User-space
OS
HW
Application
Transactions, garbage collection, allocation, and recovery management
Physical memory allocation and mapping
Non-volatile Memory

Figure 3.1: **The organization of Breeze** allows programs to bypass the operating system and directly read/write persistent data. Breeze, the green area, is responsible for transaction management, garbage collection, allocation and recovery management.

Table 3.1: Breeze provides a set of macros to declare persistent data types and pointers.

`NVM_TYPE_ID(type)`	Returns type identifier for `type`, a positive integer that is used for allocation purposes.
`NVM_STRUCT`	This is an alternative to `struct` keyword in order to declare persistent structures.
`NVM_PTR(type)`	Declares a persistent pointer of type `type` in a persistent structure.

runtime. In order to provide failure consistency, the Breeze compiler creates undo-log entries for each write before performing it. The toolchain also lets programmers initiate, commit, and abort transactions. Finally, Breeze maintains consistency of persistent data throughout recoverable failures (e.g., power outages) and provides garbage collection.

In contrast to existing NVM programming systems, Breeze automates logging and garbage collection without requiring use of managed pointers or special support from the hardware [44, 12, 95, 18]. Figure 3.1 depicts how programs interact with Breeze during execution. Below, we illustrate Breeze's interface by using it to convert a volatile linked-list (shown in Figure 3.2) into a persistent linked-list.

```c
typedef struct Node {
  struct Node *next;
  char[32] data;
} Node;

void main() {
  Node *head = NULL;
  for (int i = 0; i < 10; i++) {
    Node *t = malloc(sizeof(Node));
    if (t == NULL) {
      break;
    }
    t->next = head;
    strcpy(t->data, "Hello␣world");
    head = t;
  }
}
```

Figure 3.2: This code creates a simple, volatile linked-list with 10 elements by adding new elements to the head of the list.

Applying Breeze to existing code comprises four steps (Figure 3.3). Programmers use the C macros from Table 3.1 to label persistent structures and pointers. Next, programmers define transaction boundaries and replace volatile memory management function calls with persistent counterparts. Then, Breeze's compiler uses this information to log updates to NVM and generate code to recover from failures. Finally, programmers link the object files to Breeze's user-level NVM library. Section 3.1.1, Section 3.1.2 and Section 3.1.3 provide examples of adding annotations, defining transaction boundraies and allocating NVM for the linked-list example. Section 3.1.4 presents the design of Breeze's compiler.

3.1.1 Declaring Persistent Types and Pointers

The first step in adapting legacy code to use Breeze is annotating the data structures that will be persistent using the primitives in Table 3.1. Breeze feeds these annotations to its compiler in order to generate C functions that maintain reference count and referential integrity at

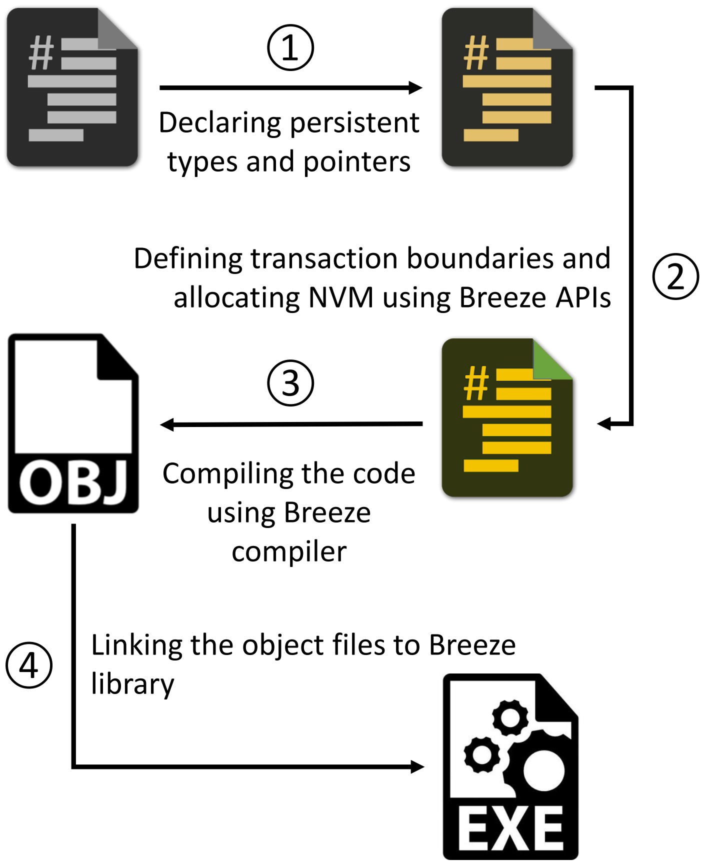

①
Declaring persistent types and pointers
②
Defining transaction boundaries and allocating NVM using Breeze APIs
③
Compiling the code using Breeze compiler
④
Linking the object files to Breeze library
OBJ
EXE

Figure 3.3: Four steps of applying Breeze to legacy software.

runtime and support failure recovery. It also exploits this information to prevent the creation of unsafe pointers from non-volatile to volatile memory and between non-volatile memory pools (i.e., contiguous regions of NVM that reside within memory-mapped files).

Figure 3.4 shows how we use these primitives for the linked-list example. The highlights show which lines needed to be changed to provide persistence.

The changes required to the code are modest. Breeze only requires annotating data structure declarations that only comprise a small portion of the source code, in order to provide pointer safety. In contrast to existing systems, Breeze does not require programmers to use special pointer types, inherit from a particular base class, or extend existing structures with new fields.

```
// declare persistent structure
typedef NVM_STRUCT Node {
 // declare persistent pointer
 NVM_PTR(struct Node) next;
 char[32] data;
} Node;
```

Figure 3.4: Declaring persistent types and pointers using Breeze primitives.

3.1.2 Atomic Sections

Specifying actions that take a persistent data structure from one consistent state to another is a central requirement for a NVM programming system. This information, allows the system to restore the data structure to a consistent state in case of a failure. Breeze facilitates implementing ACID [36] transactions by providing atomicity, consistency and durability for all transactional updates to NVM. The toolchain automates logging and allows performing updates on the main version of persistent objects. Programmers are responsible for isolation, but, in most cases, legacy software already includes concurrency control. Breeze allows it to keep using the same mechanism.

Breeze allows programmers to declare transaction boundaries, and then it automatically generates undo-log entries for resulting atomic sections. Breeze's transactional interfaces (Figure 3.5) include `nvm_tx_begin()`, `nvm_tx_commit()` and `nvm_tx_abort()`. In contrast to other systems (e.g., Intel's PMDK [44]), Breeze does not require the creation of log entries prior to updating NVM. Instead, the compiler generates necessary code to create undo-logs before every write to NVM. The design of Breeze's compiler is presented in Section 3.1.4.

3.1.3 Allocation

Breeze provides programmers with primitives to create non-volatile pools and allocate persistent objects. Programmers use `nvm_pool_create()` and `nvm_pool_open()` to create and map a file containing a newly-initialized pool or map an existing pool into applications' address

```c
void main() {
  size_t pool_size = (off_t)1 << 20;
  NVM_POOL *pop = nvm_pool_create("/nvm/pool", pool_size);
  Node *head = NULL;
  for (int i = 0; i < 10; i++) {
    nvm_tx_begin(pop); // begin transaction
    // allocate a new persistent object
    Node *t = nvm_alloc(pop, NVM_TYPE_ID(Node), sizeof(Node));
    if (t == NULL) {
      nvm_tx_abort(pop); // abort transaction
      break;
    }
    t->next = head;
    strcpy(t->data, "Hello␣world");
    head = t;
    // update the root pointer
    nvm_pool_set_root(pop, head);
    nvm_tx_commit(pop); // commit transaction
  }
  nvm_pool_close(pop);
}
```

Figure 3.5: Leveraging Breeze to perform transactional list operations.

```
POBJ_LAYOUT_BEGIN(LinkList);
POBJ_LAYOUT_ROOT(LinkList, struct Node);
POBJ_LAYOUT_TOID(LinkList, struct Root);
POBJ_LAYOUT_END(LinkList);

typedef struct Node {
  TOID(struct Node) next;
  char[32] data;
} Node;

typedef struct Root {
  TOID(struct Node) head;
} Root;

void main() {
  size_t pool_size = (off_t)1 << 20;
  PMEMobjpool *pop = pmemobj_create("/nvm/pool", POBJ_LAYOUT_NAME(LinkList),
  pool_size, 0666);
  TOID(Root) r = POBJ_ROOT(pop, Root);
  for (int i = 0; i < 10; i++) {
    TX_BEGIN(pop) { // begin transaction
      // allocate a new persistent object
      TOID(Node) t = TX_ZALLOC(Node, sizeof(Node));
      if (TOID_IS_NULL(t)) {
        pmemobj_tx_abort(1); // abort transaction
      }
      D_RW(t)->next = head;
      strcpy(D_RW(t)->data, "Hello world");
      pmemobj_persist(pop, D_RW(t), sizeof(Node));
      TX_ADD(r); // create undo-log for root
      D_RW(r)->head = t;
      pmemobj_persist(pop, D_RW(r), sizeof(Root));
    }
    TX_END { } // commit transaction
  }
  pmemobj_close(pop);
}
```

Figure 3.6: Making the linked-list persistent using PMDK.

space, respectively. In case of failures, `nvm_pool_open()` also performs necessary recovery operations on the pool to ensure its consistency. Pools are self-contained and include all the information necessary for failure recovery.

Each pool has a *root pointer* that provides access to all the live objects in the pool. Persistent objects that are not reachable from the *root pointer* are considered dead and are candidates for garbage collection. Programmers can modify the root pointer by calling `nvm_pool_set_root()`.

Programmers allocate space for persistent objects within a pool using `nvm_alloc()`. Breeze provides transactional allocation and ensures allocated objects are reclaimed if the corresponding transaction aborts or the object is not referenced by any other persistent pointer.

The programmer can explicitly free a dead object with `nvm_free()` (the call returns an error if the object is live). Calling `nvm_free()` is not necessary, but it reduces the burden on the garbage collector.

Highlighted statements of Figure 3.5 shows how to use Breeze for the linked-list example from Figure 3.2 to create persistent pools, define atomic sections and transactionally allocate persistent objects.

When a program is finished with a pool, it can close the pool with `nvm_pool_close()`. This leaves the pool in a consistent state, so no recovery is necessary the next time an application opens it. Breeze closes all pools when the program exits.

3.1.4 Breeze's Compiler

The Breeze compiler automatically inserts logging code and generates recovery code for data structures. Breeze's compiler inserts logging code before each write to NVM. The compiler inserts a function call before each NVM write to invoke the undo-logging function implemented in Breeze's user-level library.

NVM writes could also happen inside library functions. Breeze allows programmers to link the code to any pre-compiled library. The compiler inserts logging code before calling library functions to create undo-log entries for NVM regions that the library function modifies. We describe the semantics and implementation details of Breeze's compiler in Section 3.2.3.

In addition to automating the process of creating undo-logs, the compiler uses programmers annotations introduced in Section 3.1.1 to generate code in order to maintain referential integrity of persistent objects, garbage collect unreachable regions and recover from failures. Section 3.2.6 and Section 3.2.7 discuss the role of Breeze's compiler in maintaining referential integrity and performing garbage collection in more depth.

3.2 Implementation

We implemented Breeze as a C library and a C compiler under Linux. We have built Breeze's compiler by extending the LLVM compiler infrastructure [59]. Breeze does not require special hardware support. Below we describe how Breeze lays out data in persistent pools, handles atomicity and data allocation, maintains referential integrity, and recovers from failures.

3.2.1 Storage Layout

Breeze organizes persistent objects into continuous regions of NVM called memory pools. We utilize filesystem's naming mechanism to find memory pools after program restarts and `mmap()` them to the program's address space. Breeze requires the filesystem to provide direct access to NVM pages (i.e., DAX or page cache bypass) of memory-mapped files [61].

Memory pools are divided into six segments. The header segment, depicted in Figure 3.7, is the first page of a persistent memory pool and contains metadata about the pool as well as the offset of the root object.

Breeze stores programs' data in the data segment located after the header segment. Breeze's allocator is responsible for managing the data segment. The user-level library utilizes the next segments to store garbage collection data and the transaction log, discussed in Section 3.2.7

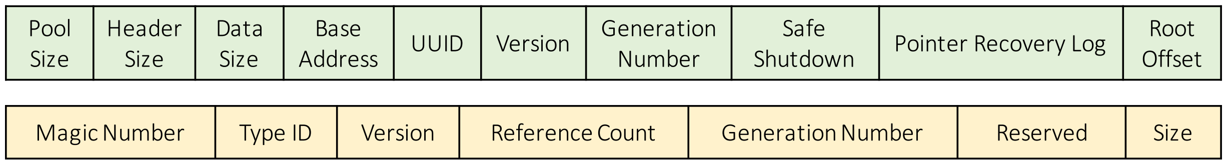


Figure 3.7: Storage layout of the pool header (top) and the object header (bottom).

and Section 3.2.3 respectively. In contrast to existing NVM programming systems such as PMDK and NVM-Direct, Breeze requires less space to store its metadata since it neither persists allocation tables nor stores additional information for persistent pointers [44, 12].

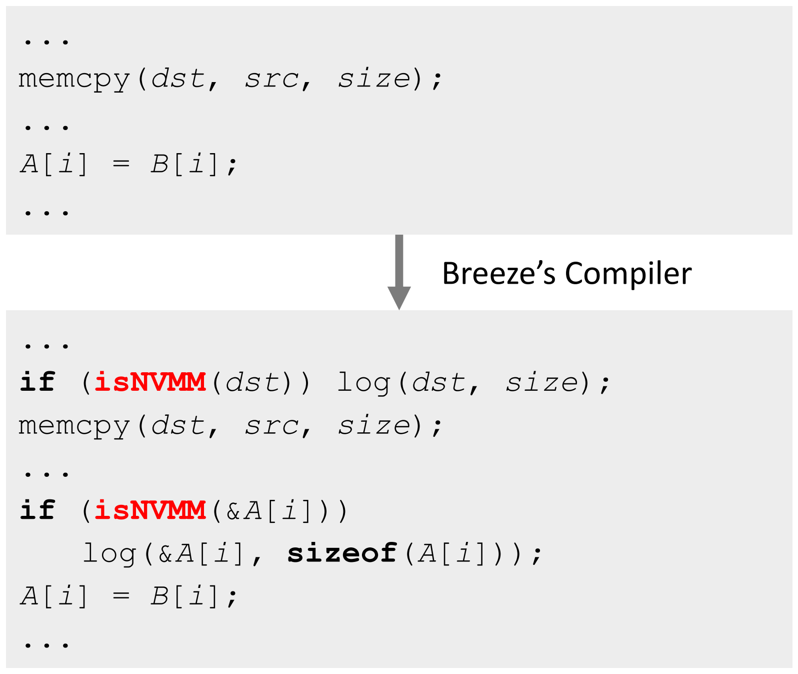
3.2.2 Memory Allocation and Management

Breeze provides transactional allocation semantics and avoids memory leaks through reference counting. Breeze's user-level library implements a two phase protocol for allocating/deallocating space for persistent objects:

- The library creates an undo-log entry in the pool's transaction log area for allocation/free requests.
- If a transaction aborts, the library uses the undo-log entries to undo allocation/deallocation requests that correspond to the aborted transaction. The library discards the undo-log entries on transaction commit.

Breeze's allocator uses per-thread allocation lists to minimize false sharing and cache contention [18]. It also organizes free persistent regions based on their sizes into different sub-lists in order to reduce allocation overhead and fragmentation [56].

Breeze does not store allocation state in persistent memory. Instead, it scans the pool during startup to find free regions and rebuild per-thread allocation lists. This enables faster allocation, and Breeze performs the recovery scan using multiple threads to minimize the cost. We also use the recovery scan to provide pointer safety and making sure persistent pointers remain valid throughout program restarts.



```
...
memcpy(dst, src, size);
...
A[i] = B[i];
...
```

```
...
if (isNVMM(dst)) log(dst, size);
memcpy(dst, src, size);
...
if (isNVMM(&A[i]))
    log(&A[i], sizeof(A[i]));
A[i] = B[i];
...
```

Figure 3.8: Breeze's compiler injects boundary check instructions before those memory writes the address of which is decided at runtime. The compiler does not add boundary checks before writes to heap-resident and stack-resident regions. During runtime, if the write operation aims to update a NVM region, Breeze's undo-log function is invoked with the target address and size of the write operation.

3.2.3 Atomicity

Breeze's compiler creates undo-log entries before allowing programs to modify persistent memory regions. In contrast to existing NVM systems that aim to provide easy to use atomicity, Breeze's approach is applicable to all types of applications and does not require any hardware support, frequent switches between the user and kernel mode, or coarse-grain logging [39, 49, 102].

First, the compiler toolchain uses LLVM's front-end (i.e., `clang`) to generate LLVM IR code. Then, it identifies every instruction that could update a region of byte-addressable memory, either directly (e.g., `store`) or indirectly (e.g., `function call`). If the compiler has enough information about the memory region that the instruction updates to decide whether it resides in volatile or persistent memory, Breeze's compiler ignores the instruction or generates undo-logging code before the instruction, respectively. The undo-logging code calls to Breeze's user-level library and provides its undo-log function with the address and size of the memory region. This function allocates space for the log entry, copies data from the memory region to the allocated space, ensures its persistence and updates the transaction metadata.

If the volatility or persistence of a memory region is unknown at compile time (e.g., function arguments), the compiler inserts a call to Breeze's user-level undo-logging function before the instruction, however, a boundary check instruction precedes the function call to avoid unnecessarily invoking the undo-log function for volatile memory regions (see Figure 3.8). Breeze's current implementation reserves a contiguous region of virtual address space for persistent memory pools at program startup and uses the lower-bound and upper-bound of this region to perform boundary checking. Figure 3.8 provides a brief explanation of how Breeze's compiler works.

As long as programmers use Breeze's compiler to compile the code, it can provide atomicity by inserting undo-logging code before the instructions that modify NVM. However, pre-compiled libraries can also modify persistent data and Breeze must log these changes as well. Section 3.2.4 explains how Breeze addresses this issue. We have also implemented a few optimization techniques to reduce the runtime cost of our atomicity technique. Section 3.2.5 briefly explains these techniques and offers insights for further optimizations.

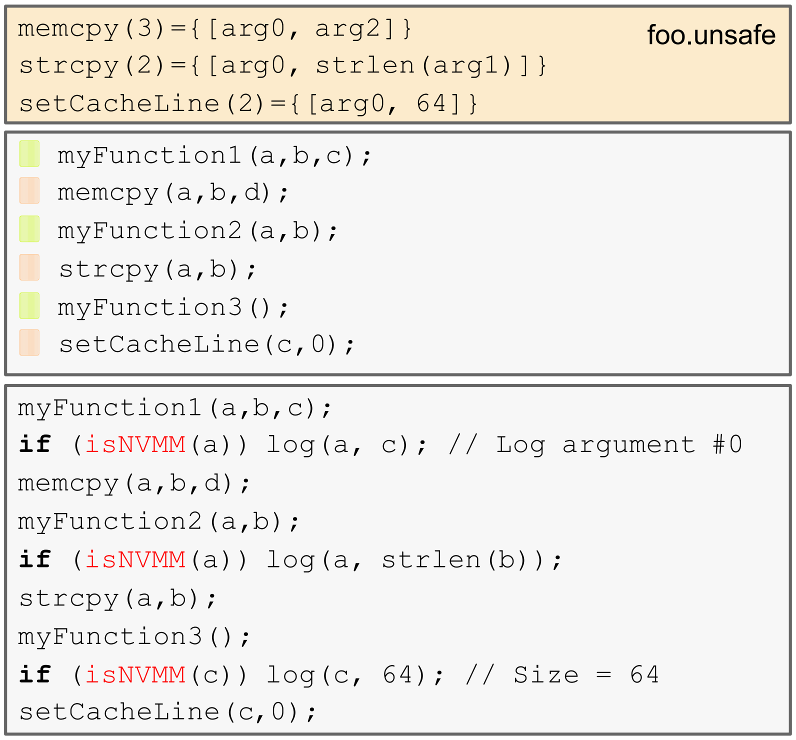
3.2.4 Pre-Compiled Libraries

Breeze allows pre-compiled libraries to safely modify NVM, since recompiling third-party libraries is not always possible or desirable, especially for those libraries the source code of which is not publicly available. As the most majority of such library functions only modify memory regions referenced by pointers passed through their arguments, the compiler can insert logging code in user's code and before the function call instruction. This technique enables Breeze to provide atomicity without requiring the programmer to recompile every piece of the program. Breeze's current implementation logs changes to persistent data passed as arguments, which is all we needed for our benchmark applications, but we can extend this to handle changes to global persistent variables by enabling programmers to pass information about such variables (address and size) to Breeze's compiler as well.

We classify functions into two groups: atomic-safe and atomic-unsafe functions. Atomic-safe functions are compiled by Breeze's compiler and contain undo-logging code for instructions in the body of the function that modify NVM. Therefore, the compiler does not need to insert any undo-logging code before calling these functions. On the other hand, atomic-unsafe functions reside in pre-compiled libraries and the Breeze's compiler has no information about presence or absence of undo-logging code inside these functions. As a result, providing these functions with pointers to NVM regions might be unsafe and the compiler needs to insert undo-logging code before calling these functions.

The programmer must create a file with annotations for unsafe functions that describes how they modify memory. This file (e.g., `foo.unsafe`) includes the list of unsafe functions and their metadata, and should be placed in a location known to the compiler. The metadata for unsafe functions provides compiler with the address and size of NVM regions that the unsafe function modifies. Figure 3.9 shows an example of `foo.unsafe`. Function names in `foo.unsafe` are followed by the list of memory regions the function modifies. These lists describe how unsafe functions modify memory based on their arguments. For example, `strcpy(2)` writes the string referenced by its second argument to a memory region referenced by its first argument. Therefore, it writes `strlen(arg1)` bytes to a memory region referenced by `arg0`. Similarly, `memcpy(3)` writes a total of `arg2` bytes (third argument) to a memory region referenced by `arg0` (first argument).

Breeze's compiler uses the information in this files to generate object files from LLVM IR codes. During this process, every function call is checked against the list of safe and unsafe functions. No extra work is required for safe functions (green bullets), however, the compiler inserts undo-logging code before unsafe function calls (orange bullets) using the metadata



```
memcpy(3)={[arg0, arg2]}
strcpy(2)={[arg0, strlen(arg1)]}
setCacheLine(2)={[arg0, 64]}
```

```
🟩 myFunction1(a,b,c);
🟧 memcpy(a,b,d);
🟩 myFunction2(a,b);
🟧 strcpy(a,b);
🟩 myFunction3();
🟧 setCacheLine(c,0);
```

```
myFunction1(a,b,c);
if (isNVMM(a)) log(a, c); // Log argument #0
memcpy(a,b,d);
myFunction2(a,b);
if (isNVMM(a)) log(a, strlen(b));
strcpy(a,b);
myFunction3();
if (isNVMM(c)) log(c, 64); // Size = 64
setCacheLine(c,0);
```

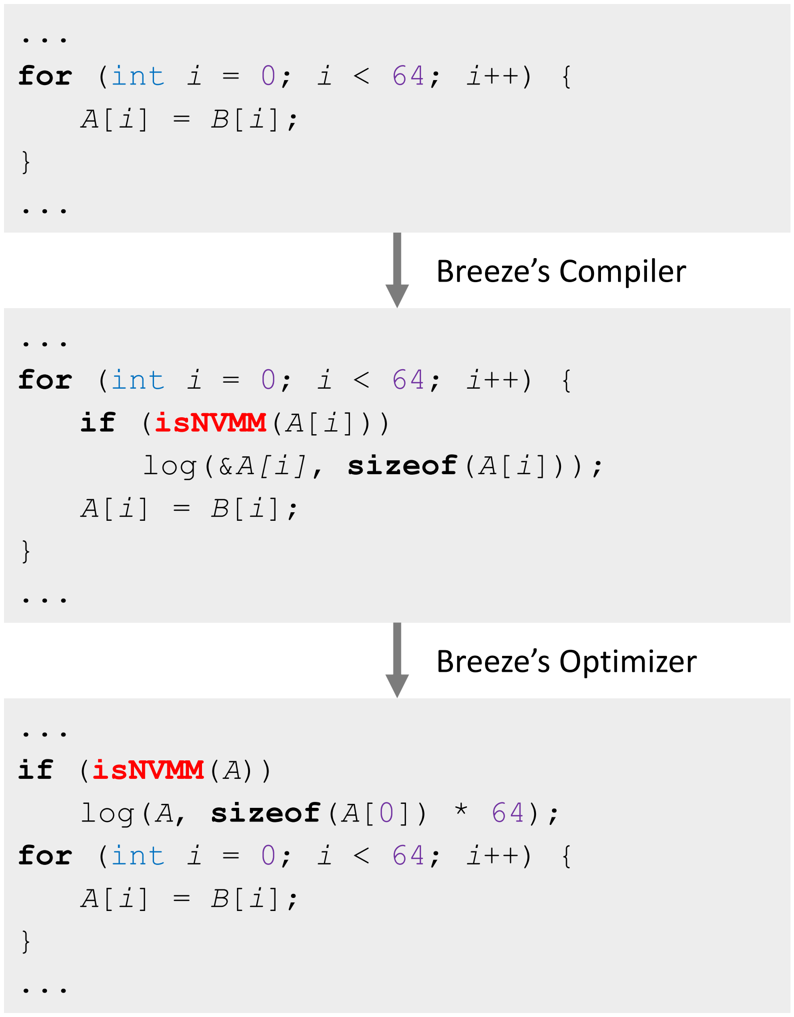
Figure 3.9: Breeze's compiler uses the information in `foo.unsafe` (top) to provide atomicity for library function calls (middle). Numbers in parenthesis for `foo.unsafe` are the number of arguments for each function. In addition to function names, `foo.unsafe` contains metadata for each function that allows the compiler to identify the address and size of memory regions that the function modifies. The programmer provides Breeze with this information.

provided by the programmer. An example of the undo-log code that Breeze's compiler generates is shown in the bottom of Figure 3.9.

The programmer should also replace function pointers to unsafe (pre-compiled) functions with pointers to safe (user-defined) wrapper functions. This enables Breeze's compiler to insert proper undo-logging code before calls to unsafe functions without requiring disruptive changes to the source code.

3.2.5 Optimizations

The goal here is to minimize the performance overhead of using Breeze's compiler to create undo-log records. A perfect optimization technique could enable the compiler to remove



```
...
for (int i = 0; i < 64; i++) {
    A[i] = B[i];
}
...
```

```
...
for (int i = 0; i < 64; i++) {
    if (isNVMM(A[i]))
        log(&A[i], sizeof(A[i]));
    A[i] = B[i];
}
...
```

```
...
if (isNVMM(A))
    log(A, sizeof(A[0]) * 64);
for (int i = 0; i < 64; i++) {
    A[i] = B[i];
}
...
```

Figure 3.10: An example of using the optimizer to reduce the frequency of boundary checks and calls to the log function.

all boundary checks and keep the frequency of calls to the undo-log function at the minimum. Our first step in this direction is to avoid boundary check and undo-log creation for pointers that reference stack and volatile heap resident data. Considering that most of the memory accesses for a wide range of applications are for heap and stack data [78], this significantly reduces the performance overhead of Breeze by reducing the frequency of boundary checks.

We have also implemented a simple optimization for loops to reduce the frequency of function calls and boundary checks (Figure 3.10). The aim of this technique is to replace multiple fine-grain undo-log entries with a single coarse-grain undo-log. As a result, this technique can reduce the frequency of function calls and boundary checks by $n-1$, where n is the total iterations of the loop (64 in our example).

We are working on a few other ways to reduce the performance overhead of Breeze on NVM applications. An example is to avoid repeating boundary checks and calls to the undo-log function for the same memory region. The compiler utilizes use-define chains to decide if inserting undo-logging code before an instruction is necessary. This is done by tracking all reachable definitions for a use (a pointer referencing a memory region that is being updated by the corresponding instruction) and making sure undo-logging code is present for every single reachable definition.

3.2.6 Pointer Safety

Breeze allows programmers to directly update persistent pointers. This approach has no effect on validity of persistent pointers as long as the physical to virtual mapping of non-volatile pages does not change. In contrast to Mnemosyne [95] which maintains this mapping through kernel-level support, Breeze allows this mapping to change while maintaining validity of persistent pointers. If the operating system cannot maintain the same base address for a persistent memory pool after a restart (e.g., because another pool is already mapped to that address), Breeze adjusts persistent pointers to account for the change by running Algorithm 1. This algorithm adjusts every pointer inside the persistent pool by adding the offset between the old and new base addresses.

For example, assume the operating system is mapped a persistent pool at address 0xA000 when a failure occurs. After the failure, the application calls `nvm_pool_open()` in order to run recovery and map the persistent pool to its address space. However, the operating system finds that 0xA000 is already in use, so it has to map the pool to address 0xB000. In order to maintain referential integrity, Breeze transactionally adds 0x1000 to all persistent pointers in the pool.

Since failures might occur during the execution of Algorithm 1, Breeze keeps track of

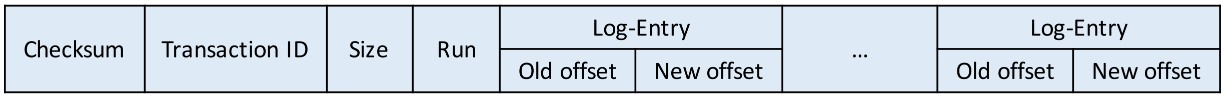


Figure 3.11: Storage layout of the reference count activation records.

the persistent pool's base address for each recovery attempt as well as generation numbers for persistent objects and the persistent pool in order to tolerate such failures. Generation numbers are integer values assigned to persistent pools and objects. Breeze increments the pool's generation number for every recovery attempt. The library updates an object's generation number with the pool's generation number once it finishes recovering the object.

Consider the previous example. A failure occurs before Algorithm 1 finishes updating the pointers when the pool resides at 0xB000. When the application opens the pool next time, the kernel happens to map it to 0xC000. The generation numbers and history of base addresses enables Breeze to add 0x1000 to pointers that were updated during the last recovery session while adding 0x2000 to pointers that were not. Breeze can run the recovery algorithm in parallel to reduce recovery time, or it could perform recovery lazily when the application accesses an object for the first time after recovery. Breeze only requires a small persistent area to run the recovery algorithm. The maximum size of NVM required to run the recovery algorithm is $|recovery-threads| \times max(object-size)$. For example, Breeze only requires 4 KB to run the recovery code for 1 KB objects using 4 threads.

3.2.7 Garbage Collection

Breeze's garbage collection algorithm leverages the persistent pointer information to track reference count of persistent data structures. Breeze implements a two-phase garbage collection algorithm similar to the scheme introduced by NV-Heaps [18] and uses weak pointers to avoid memory leaks due to cycles.

The first phase of Breeze's GC algorithm runs during transaction commit and identifies the pointers that have changed by comparing the objects in the transaction's log with the original version of those objects. It records the new value of the pointers along with their original values in a data structure called activation record (Figure 3.11). Breeze ensures persistence of these activation records before marking the transaction as committed. Algorithm 2 describes the first phase of Breeze's garbage collecting algorithm: creating activation records on transaction commit. We use a circular buffer of size `log_size` to maintain these activation records for each transaction. The circular buffer accommodates 65,538 activation records, therefore, no transaction is expected to wait for another transaction to complete. If there is no available activation records, the oldest transaction in the circular buffer is considered as permanently blocked and restarted to open up space for new transactions.

The second phase of the algorithm consumes the activation records inside the circular buffer. For each entry, it increments the reference count on the object the pointer now points to and decrements the count on the object it used to point to. A background thread iterates over the activation records and translates them into necessary changes to the reference counts. Since updating reference counts is not an idempotent operation, the background thread converts each activation record into a set of redo-log entries prior to updating reference counts.

Algorithm 1 Fixing persistent pointers on recovery

```
procedure POINTER_REWRITE($pool, region$)
    $region\_base \leftarrow pool.base\_addr$
    if $region.gen\_num \neq pool.gen\_num$ then
        $i \leftarrow region.gen\_num - pool.gen\_num - 1$;
        $region\_base \leftarrow pool.ptr\_recovery\_log[i]$
    end if
    $disp \leftarrow pool - region\_base$
    if $disp = 0$ then
        return
    end if
    tx_begin
    for all $ptr \in region.pointers$ do
        $ptr \leftarrow ptr + disp$
    end for
    $t \leftarrow pool.gen\_num + pool.recovery\_attempts$
    $region.gen\_num \leftarrow t$
    tx_end
end procedure
```

Algorithm 2 Creating a list of pointer changes

```
procedure LOG_POINTER_CHANGES($tx$)
    $log \leftarrow rc\_log[tx.id \mod log\_size]$
    $log.tx\_id \leftarrow tx.id$
    $n \leftarrow 0$
    for all $obj \in page$ do
        for all $ptr \in obj$ do
            if $ptr.old\_val \neq ptr.new\_val$ then
                $log[n].old\_val \leftarrow ptr.old\_val$
                $log[n].new\_val \leftarrow ptr.new\_val$
                $n \leftarrow n + 1$
            end if
        end for
    end for
    $log.size \leftarrow n$
    Update $log.checksum$
    Persist $log$
end procedure
```

3.3 Evaluation

This section evaluates Breeze and compares its performance and ease-of-use against NVM-Direct [12] and PMDK [44] (version 1.0). We describe our evaluation platform and then explain our experiments for measuring ease of use, performance overhead and recovery time of Breeze.

3.3.1 Applications

We use two groups of applications to benchmark Breeze. The first group includes persistent implementations of a B+Tree and a hash table using Breeze, NVM-Direct [12] and PMDK [44]. The implementation of the B+Tree is similar to that of NV-Tree [99]. We use Cuckoo hashing [82] as the mapping algorithm for our hash table implementation. We use DJB2 and Jenkins hash functions for the Cuckoo hashing implementation [97].

The second group are legacy applications that can benefit from NVM. We have incorporated Breeze and PMDK with Memcached [31] and MongoDB [77] for this purpose. Memcached is a general purpose memory caching system with no durability semantics. MongoDB is a persistent document store with support for atomic updates. For these experiments, MongoDB is configured to ensure persistence of each insert/update operation before acknowledging the client.

We exercise these applications with the YCSB [21] workloads described in Table 3.2. YCSB provides a common ground to evaluate the performance of different key-value storage systems. For our experiments, we populate the storage system with 10 million key-value pairs of size 1 kB. Then, we run each of the workloads from Table 3.2 to measure the performance of our target system. The outcome of this process is the average latency and throughput of the storage system. The latency/throughput numbers for workload C are not presented here since it only contains read operations which results in identical performance for Breeze and PMDK.

Table 3.2: The percentage of different operations in each YCSB workload.

Workload	**Read**	**Update**	**Insert**	**Read & Update**
YCSB-A	50	50	-	-
YCSB-B	95	5	-	-
YCSB-D	95	-	5	-
YCSB-F	50	-	-	50

Table 3.3: Configuration of the NVM emulation platform

Size	**Read latency**	**Write bandwidth**	***clwb* latency**
32 GB	100 ns	1/8 DRAM	40 ns

3.3.2 Test System

We utilize Intel's Persistent Memory Emulation Platform (PMEP) [27] to emulate the latency and bandwidth of NVM for our experiments. PMEP is a dual socket platform equipped with Intel Xeon processors with 8 cores running at 2.6 GHz. The platform has a total of 4 DDR3 channels, where channels 2 and 3 are marked by the BIOS for emulating NVM. Table 3.3 shows how we configured the platform for the experiments.

3.3.3 Ease of Use

To compare Breeze's ease of use with other systems' we use the number of lines of code that must be changed in order to enable an existing application to use NVM. The B+Tree and hash table implementations serve as a baseline to compare Breeze against NVM-Direct [12] and PMDK [44]. Table 3.4 shows that Breeze requires between 55% and 82% fewer modified lines than NVM-Direct or PMDK.

We have also incorporated Breeze and PMDK into MongoDB [77] and Memcached [31].

Table 3.4: Comparing Breeze against NVM-Direct [12] and PMDK [44] in terms of ease of use. The numbers measure the lines of code that needs to be changed.

	NVM-Direct	**PMDK**	**Breeze**
B+Tree	96	101	18
Hash Table	77	45	20

Table 3.5: Measuring the programming effort of incorporating Breeze and PMDK [44] with MongoDB [77] (Mongo) and Memcached [31] (Mcache) as an indicator of ease of use.

	Lines changed		**Percent changed**	
	Breeze	**PMDK**	**Breeze**	**PMDK**
Mongo	882	1273	0.05	0.07
Mcache	541	1547	0.05	0.14

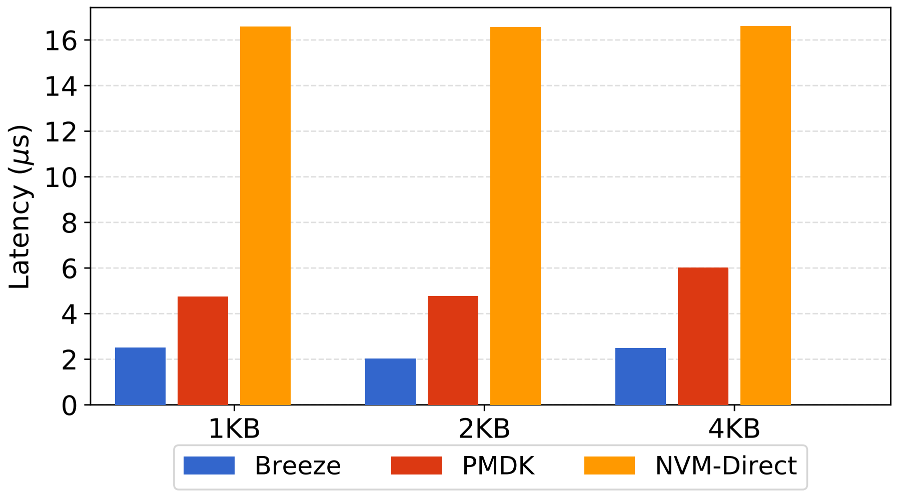


Figure 3.12: Measuring the average allocation latency for objects of size 1, 2 and 4 kilobytes.

Table 3.5 shows the ratio of the source code for each application that must be modified as a result of incorporating Breeze and PMDK. In contrast to PMDK, Breeze requires between 1.2× to 2.8× fewer changes to the source code. We did not replicate these experiments for NVM-Direct, but we expect similar numbers for PMDK and NVM-Direct.

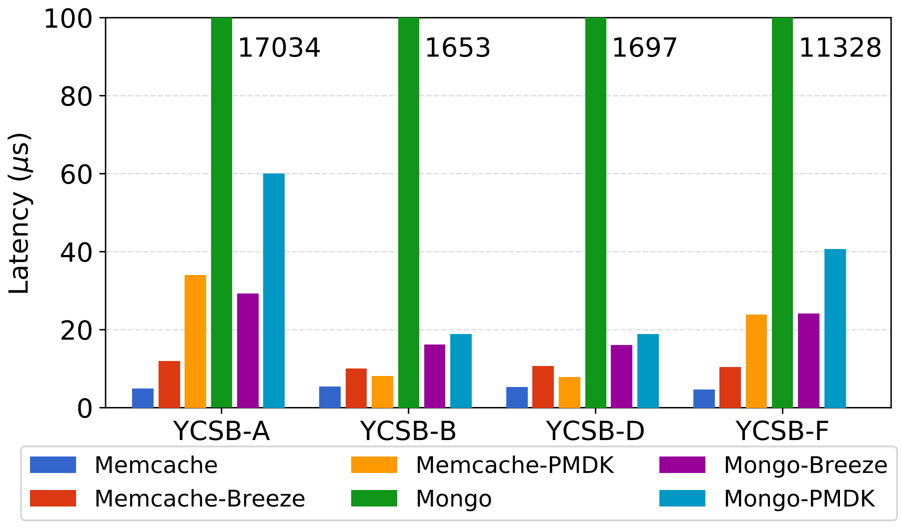


Figure 3.13: Average latency of unmodified and NVM-enabled versions of MongoDB and Memcached using YCSB workloads.

3.3.4 Performance

Figure 3.12 compares the allocation overhead of Breeze against NVM-Direct [12] and PMDK [44]. For these experiments, we report the average latency of allocating one million objects of size 1, 2, and 4 kB. In contrast to PMDK and NVM-Direct, Breeze requires 1.35× and 7.2× less time for allocation, respectively.

We also report performance measurements of Breeze using our benchmark applications and YCSB workloads. We compare latency and throughput of Memcached [31] and MongoDB [77] against the Breeze-enabled (Memcache-Breeze and Mongo-Breeze) and PMDK-enabled (Memcache-PMDK and Mongo-PMDK) versions. We only use Breeze to compile MongoDB's storage engine to avoid unnecessary overhead of boundary checks. Figure 3.13 and Figure 3.14 summarize the results. Both NVM-enabled versions of MongoDB outperform its original version by avoiding system calls (e.g., `msync()`) to persist data. NVM-enabled versions of Memcached show higher latency and lower throughput compared to the unmodified version due to the cost of providing persistence. Breeze provides superior performance for such applications compared to the PMDK version because of its optimized undo-logging scheme and

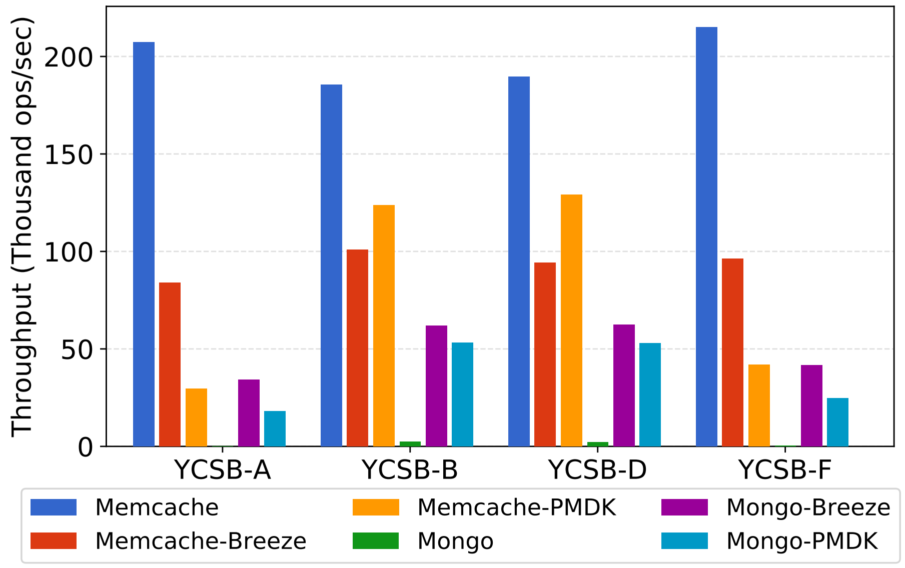


Figure 3.14: Average throughput of unmodified and NVM-enabled versions of MongoDB and Memcached using YCSB workloads.

not persisting allocation metadata. The only exception is running YCSB-B and YCSB-D against Memcache-Breeze where the overhead of boundary checks cancels out the performance benefits of Breeze's optimized transaction implementation.

Finally, we have incorporated Breeze, PMDK and NVM-Direct with the B+Tree and hash table implementations to compare the performance of Breeze with NVM-Direct and PMDK. According to the results from Figure 3.15 and Figure 3.16, NVM-Direct shows higher latency and lower throughput compared to Breeze and PMDK due to its inefficient implementation of logging and allocation. Breeze outperforms PMDK for YCSB-A and YCSB-F (write-heavy) because of its optimized undo-logging scheme. Since only the last level of the B+Tree is persistent, the performance difference between Breeze and PMDK is minimal for the B+Tree benchmarks. Furthermore, Breeze and PMDK show similar performance for read-heavy workloads.

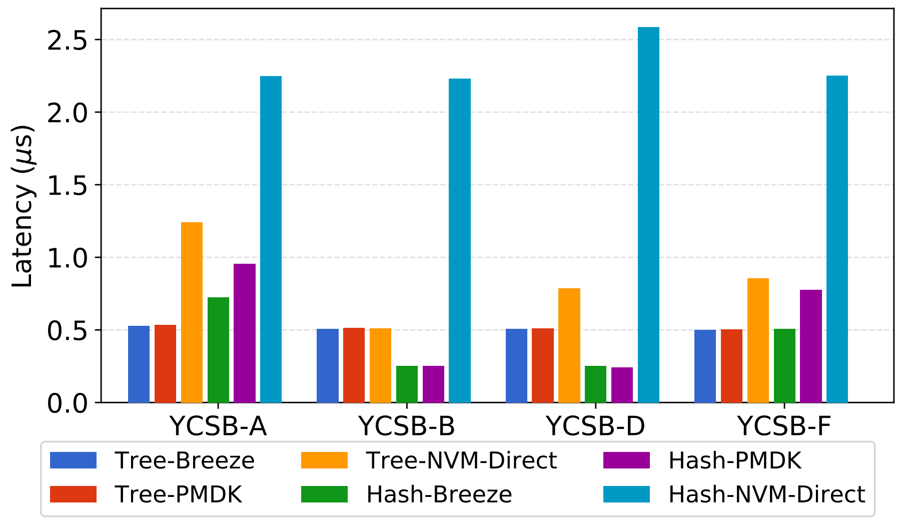


Figure 3.15: Average latency of Breeze, NVM-Direct and PMDK using YCSB workloads.

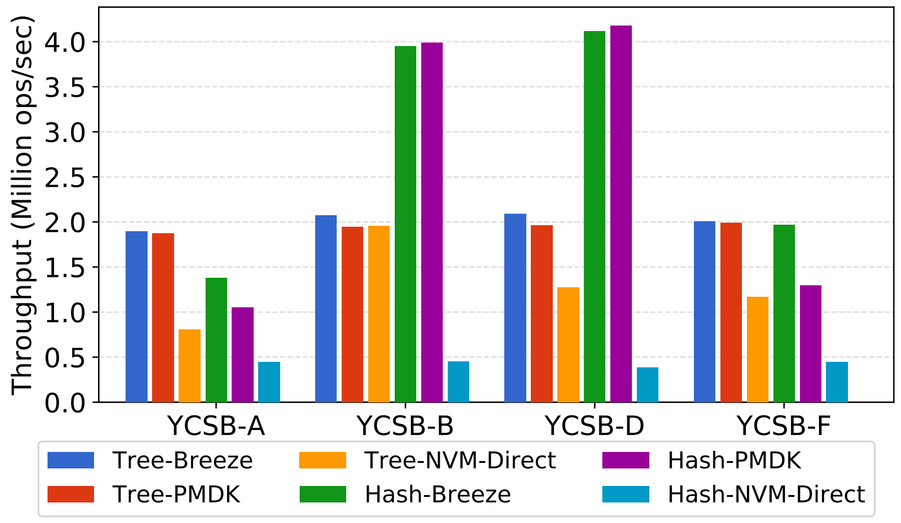


Figure 3.16: Average throughput of Breeze, NVM-Direct and PMDK using YCSB workloads.

3.3.5 Recovery Time

This section measures the overhead of reconstructing allocation lists and rewriting persistent pointers during startup. For these experiments, we use objects of size 1 kB and persistent pools with a total capacity of 4, 8 and 16 gigabytes. Also, we varied the number of recovery threads to measure the scalability of our scheme.

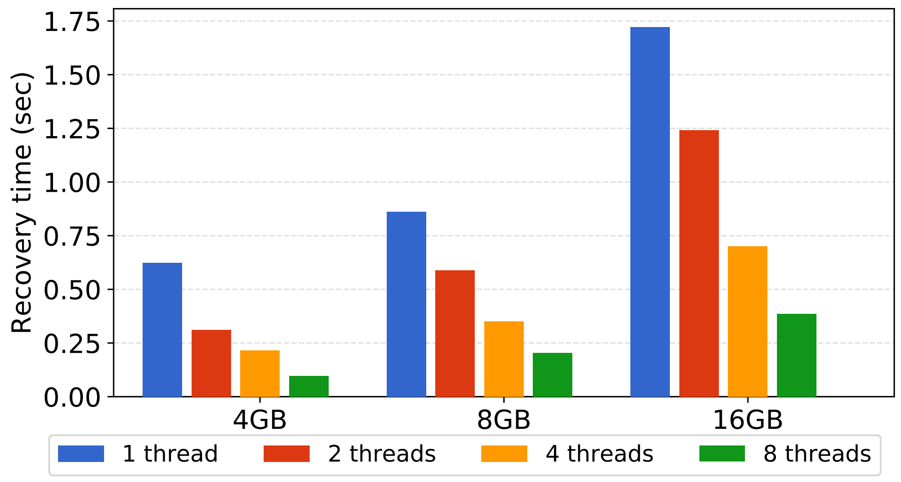


Figure 3.17: Measuring the recovery time of Breeze when the persistent pool can be mapped to the same virtual address space after recovering from failures.

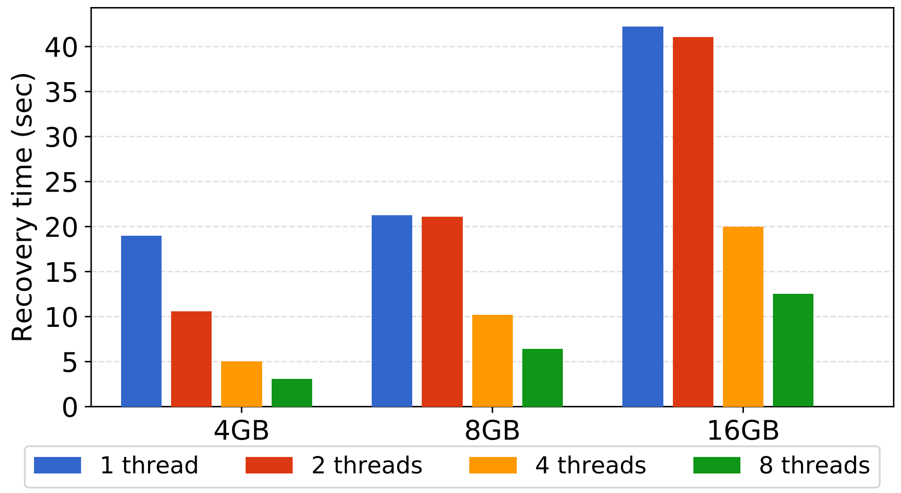


Figure 3.18: Measuring the overhead of rewriting persistent pointers when the persistent pool is mapped to a different virtual address space.

Figure 3.17 shows the average recovery time of Breeze when the virtual address of persistent pools does not change. The numbers directly show the overhead of recreating allocation lists.

We also measured the overhead of rewriting persistent pointers by mapping pools into different virtual addresses during startup. Figure 3.18 shows the average recovery time for these experiments that indicate utilizing more threads can significantly reduce the recovery time.

3.4 Related Work

Designing NVM-optimized systems is one way to address the consistency and safety challenges. Since logging is a major bottleneck in transactional systems, many have proposed logging schemes that offer better performance by considering NVM characteristics [40, 96, 55]. These schemes also serve as building blocks for larger systems like TANGO [7] that uses shared logs to build distributed data structures. Moreover, there are versions of popular data structures such as B-Trees that are redesigned with respect to NVM properties, such as byte-addressability [99]. Database researchers are also actively looking for ways to optimize transactional protocols, e.g. OLTP, in order to exploit NVM potentials and improve transaction latency and throughput [85, 55, 93]. In contrast to our solution, these systems do not provide a general approach for optimizing existing software and assume NVM-optimized applications are created from scratch.

Another direction to approach failure consistency is adding persistence to the memory controller and processor caches. Whole system persistence [79] and JUSTDO Logging [49] are examples of exploiting persistent caches to recover from failures by retaining the programs' execution state throughout failures. Additional support from the software is still necessary to prevent potential unrecoverable conditions such as deadlocks [6]. Klin [102], WrAP [26], NVM Duet [64] and ThyNVM [88] combine hardware and software techniques to offer atomic writes to NVM. In contrast, Breeze only relies on existing hardware support in commodity processors.

Other efforts in this area are either focused on providing transactional semantics and safety features through programming interfaces and language support [18, 95, 12, 44, 62, 14, 39] or improving the performance of such transactional systems [57, 69]. NV-Heaps [18] and Mnemosyne [95] are among the first systems to offer such semantics at user-level. In contrast to Breeze, Mnemosyne does not offer garbage collection and sacrifices flexibility to provide pointer safety by not allowing persistent memory regions to be mapped into a different virtual address space after creation. NV-Heaps provides atomicity and safety guarantees through C++ classes. However, applying NV-Heaps to existing code requires disruptive changes.

NVM-Direct [12] provides similar semantics as NV-Heaps by leveraging compiler support and introducing new programming keywords. In contrast to Breeze, NVM-Direct introduces new syntax and requires programmers to use specific keywords to define persistent pointers and perform atomic updates. Thus, applying NVM-Direct to legacy code leads to far more changes to the source code.

Intel has published the PMDK [44] library that provides a framework for building NVM-optimized applications and data structures. The library provides a set of primitives to create transactions and manage persistent regions. However, it does not guarantee referential integrity of persistent data. Also, it requires disruptive changes to existing programs as programmers are required to use special pointer types and create log entries.

DUDETM [62] provides direct NVM access through a crash-consistent transactional system and aims to reduce the overhead of redo and undo logging by maintaining a shadow, volatile copy of persistent data in DRAM. By reducing the number of memory fences and cache flushes in the critical path, DUDETM improves transaction latency and throughput. However, it requires programmers to replace all memory accesses (loads and stores) in a transaction with DUDETM APIs (`dtmRead` and `dtmWrite`).

In addition to the recent proposals, there are others such as Rio Vista [67] and RVM [91] that offer transactional semantics and failure recovery for byte-addressable storage. However, their support is limited to durability semantics and they do not offer features such as referential integrity and garbage collection.

There are other proposals that adopt existing programming constructs to help support NVM programming. Atlas [14] and NVthreads [39] use lock-based critical sections to infer which operations on a data structure should be atomic. Atomic `msync()` [83] extends conventional `msync()` to provide stronger atomicity guarantees. Both methods have limitations since the former is only applicable to a particular class of applications and the latter requires frequent system calls. NVthreads, in particular, is not applicable to lock-free data structures and limits the performance benefits of NVMs due to the frequent context switches caused by using the page-table protection mechanism to detect writes to NVM pages.

3.5 Summary

Breeze provides direct access to non-volatile memories without requiring disruptive changes to legacy software. Breeze works with commodity hardware and offers transactional semantics, referential integrity and garbage collection. The toolchain lifts the burden of logging from programmers, automatically generates recovery code, allows programmers to use normal pointers and legacy libraries to manipulate persistent data, and provides garbage collection. Our measurements show that Breeze significantly simplifies the task of modifying existing code to use NVM while still providing better performance than other proposed systems.

Acknowledgments

This chapter contains material from “Breeze: User-Level Access to Non-Volatile Main Memories for Legacy Software”, by Amirsaman Memaripour and Steven Swanson, which appeared in the 36th International Conference on Computer Design (ICCD 2018). The dissertation author is the primary investigator and first author of this paper.

The author would like to thank the ICCD anonymous reviewers and the shepherd for their helpful comments. The author is also thankful to Subramanya R. Dulloor from Intel for his support and hardware access. Finally, the author would like to thank the members of the NVSL research group for their insightful comments.

Chapter 4

A Flexible, Optimizing Compiler for Non-Volatile Memory Programming

Non-volatile memories (NVMs) reside on the memory bus and allow the processor to access persistent data via load and store instructions. They promise to fill the gap between volatile memory and persistent storage while offering higher density and lower cost per GB relative to DRAM. The recent announcement of Intel DC Persistent Memory shows that existing NVMs can deliver 76% of DRAM performance for micro-benchmarks and storage applications [48, 51].

NVMs allow applications to bypass the filesystem and directly access persistent data using load and store instructions, thus exploiting the maximum performance these devices have to offer. However, direct access to NVM creates new challenges. Existing hardware does not support multipart and arbitrary sized atomic writes to NVM — a failure, for instance, a power outage, can tear large writes and leave persistent data corrupted (large writes are not *failure-atomic*). Programmers must implement logging algorithms (e.g., undo-logging) using the low-level, cache-line-based primitives the ISA provides to ensure large or multipart writes are failure-atomic [3].

Failure-atomicity libraries for NVM aim to facilitate NVM programming by giving programmers the ability to designate failure-atomic code regions whose writes all become persistent at once [86, 18, 95, 14]. These libraries represent a path to incremental adoption of NVM programming. However, they introduce new programming requirements that make NVM programming more complex and error-prone.

Failure-atomicity libraries generally require annotating every access to persistent data [62, 37, 57]. These annotations allow the libraries to augment NVM accesses with additional instructions and ensure failure-atomicity. For example, annotating a write to an NVM object lets the library create a copy of its current value before performing the update, and use the log to roll-back changes if a crash occurs. Programmers must also mark the beginning and end of failure-atomic regions.

These annotations are error-prone and require extensive (e.g., line-by-line) changes to existing programs. Furthermore, the annotations vary between failure-atomicity libraries, since each library provides its own APIs for defining failure-atomic operations, managing persistent memory (e.g., allocate a new persistent object), annotating NVM accesses, and restoring access to persistent objects after failures. These library-specific APIs require programmers to essentially rewrite the program if they ever decide to use an alternative library. They make it impractical (or at least very time consuming) to retarget code from one library to another.

Prior work uses compiler and hardware support to mitigate the complexities of these annotations. However, these solutions introduce significant performance overhead. Atlas [14] uses a compiler pass to automate these annotations, but it naively annotates all heap accesses and imposes significant cost at runtime. NVthreads [39] and Failure-Atomic Msync [83] avoid annotations by using page table protection bits and relying on page faults to intercept writes to NVM. The required page faults hurt performance and only allow tracking NVM accesses at 4 kB granularity.

These annotations, whether inserted manually by the programmer or automatically by a compiler pass, are opaque to the compiler. By hiding the meaning of these annotations from the compiler, failure-atomicity libraries effectively disable compiler optimizations across the annotations [49].

In this chapter, we propose *NVHooks* to reduce or eliminate the need for manual annotation of NVM programs. NVHooks automatically annotates NVM accesses and enables libraries to intercept those accesses without involving the programmer and facilitates retargeting NVM programs to new failure-atomicity libraries. NVHooks also provides a series of NVM-specific optimizations that leverage the semantics of NVM annotations to reduce their performance cost.

NVHooks makes the following contributions:

- It automates annotating NVM accesses and avoids disruptive changes to programs.
- It provides the first NVM-specific optimization passes that reduce the performance overhead of automatically annotating programs.
- It reduces the programming cost of retargeting programs towards using a new failure-atomicity library.
- It introduces NVHooks-aware micro-benchmarks to allow comparing the performance of different failure-atomicity libraries.

The rest of this chapter is organized as follows. We discuss the design and implementation of NVHooks in Section 4.1 and Section 4.2, respectively. Section 4.3 showcases the performance of NVHooks annotations and evaluates the effectiveness of its optimization passes. We discuss related work in Section 4.4 and summarize the chapter in Section 4.5.

4.1 System Overview

NVHooks is a compiler extension that facilitates the adoption of failure-atomicity runtimes for NVM by reducing the cost of annotating code. NVHooks uses the compiler to make the

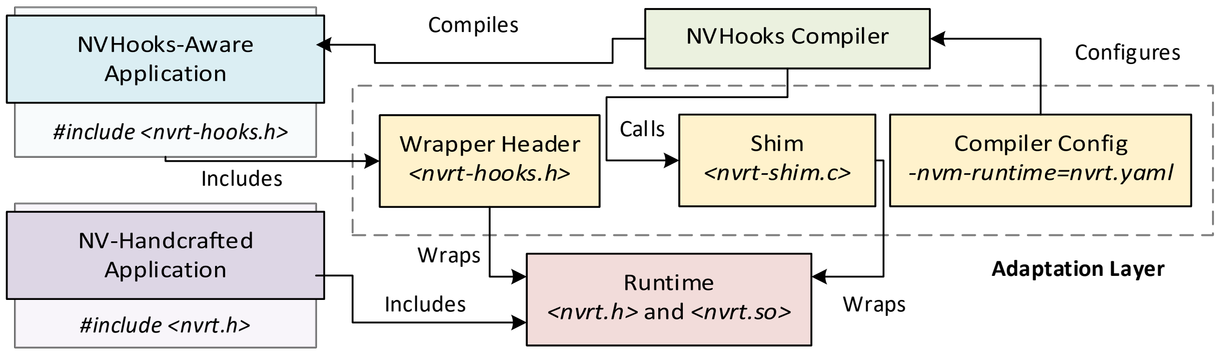

NVHooks-Aware Application
#include <nvrt-hooks.h>
Compiles
NVHooks Compiler
Configures
Wrapper Header <nvrt-hooks.h>
Calls
Shim <nvrt-shim.c>
Compiler Config -nvm-runtime=nvrt.yaml
Includes
NV-Handcrafted Application
#include <nvrt.h>
Wraps
Runtime <nvrt.h> and <nvrt.so>
Adaptation Layer
Includes
Wraps

Figure 4.1: **NVHooks system overview:** the adaptation layer uses NVHooks to hide the complexity of runtimes from applications.

instrumentation automatic and transparent to programs. It allows programmers to avoid adopting a new syntax or annotation scheme. NVHooks minimizes the programming effort of using persistent programming libraries that require annotating NVM accesses and prevents common mistakes that manual annotation introduces. Furthermore, it adds new compile-time optimizations that are aware of the semantics of persistent programming primitives and reduce the performance overhead of persistence.

NVHooks provides a set of *hooks* (i.e., callbacks) to the failure-atomicity library (or *runtime*) that intercept accesses to persistent memory. NVHooks applies these callbacks without requiring programmers' intervention. These hooks enable the runtime to add additional code at every access, allowing the runtime to, for example, log writes for failure-atomic updates, dereference a relative pointer, or force writes from caches into persistent memory. Developers of a failure-atomicity runtime can provide NVHooks compiler support by writing a small adaptation layer that links the NVHooks callbacks to the runtime's API and specifies what optimizations are valid for the runtime's semantic model.

Figure 4.1 shows the relationship between the NVHooks compiler, a failure-atomicity runtime (i.e., *nvrt*) and its adaptation layer for NVHooks, as well as persistent applications. The rest of this section describes each component in more depth, and finishes by describing the process of using NVHooks to build executables. We leave the specification and implementation details of

Table 4.1: **NVHooks callback functions.** The runtimes provide the implementation for each function (e.g., logging for `pre_nvm_write()` and relative to absolute pointer translation for `to_absolute_ptr()`).

`bool is_nvm(void *ptr);` Returns true if `ptr` references persistent memory and false otherwise. NVHooks places the rest of the callbacks as the body of an if statement, controlled by the output of `is_nvm()`.
`void *to_absolute_ptr(uintptr_t ptr);` Accepts a relative pointer (`ptr`) to an NVM region as input and returns its corresponding virtual (absolute) address at execution.

`void pre_nvm_read(void *ptr, size_t size);`
`void pre_nvm_write(void *ptr, size_t size);`
Based on the kind of NVM access (read or write), NVHooks precedes the NVM operation with a callback to either of these functions and provides the callback with the virtual address and the number of bytes the operation would access.

`void post_nvm_read(void *ptr, size_t size);`
`void post_nvm_write(void *ptr, size_t size);`
Similar to `pre_nvm_read()` and `pre_nvm_write()`, NVHooks injects these callbacks to track read/write accesses to NVM, however, they are invoked after completion of the NVM access.

the NVHooks optimizations for Section 4.2.

4.1.1 NVHooks Compiler

The NVHooks compiler instruments every instruction that could either read from or write to persistent memory with callbacks. The runtime can use these callbacks to intercept all accesses to NVM. Intercepting persistent memory accesses is a core requirement for NVM runtimes, and runtimes use these callbacks for a variety of purposes (e.g., to log persistent updates, dereference a relative pointer, or read a copy of the location in DRAM).

NVHooks instruments the NVM accesses and exposes a set of generic callbacks to the runtimes. We have built these callbacks to be general enough to handle a wide range of runtimes. Table 4.1 provides the list of these callbacks and specifies where NVHooks places each callback

```
void store_32bit(uint32_t *ptr, uint32_t value) {
  if (is_nvm(ptr)) {
    uint32_t *absPtr = to_absolute_ptr(ptr);
    pre_nvm_write(absPtr, 4);
    *absPtr = value;
    post_nvm_write(absPtr, 4);
  } else *ptr = value;
}
```

Figure 4.2: An example of how NVHooks instruments code to enable tracking persistent writes. The highlighted code shows the compiler insertions.

relative to its corresponding memory access. Note that depending on the runtime, its associated adaptation layer can specify which specific callbacks are required, thereby obviating extraneous annotations.

In its initial pass, NVHooks instruments every instruction that could reference NVM at execution. We refer to this pass of instrumentation as the *initial pass*, where the compiler instruments all memory accesses that could potentially access NVM. The initial pass introduces non-trivial performance overheads. Section 4.2 introduces new optimizations to reduce the overheads.

At each memory access, NVHooks inserts the same series of callbacks. For instance, Figure 4.2 illustrates how NVHooks attaches callbacks to a simple 32-bit write. The compiler first inserts a branch that uses the `is_nvm()` callback to avoid the additional overhead for non-NVM accesses.

For every NVM access, the inserted code uses `to_absolute_ptr()` to translate the access's address into a native pointer. Finally, NVHooks wraps the actual memory access with two callbacks: one preceding and the other succeeding the access (e.g., `pre_nvm_write()` and `post_nvm_write()` for a write to NVM).

Annotations, either automatically inserted by the compiler or manually added by programmers, cannot intercept NVM accesses in pre-compiled functions, which impedes using

```
void *memset(void *ptr, int value, size_t num)
    __attribute__((annotate("nvm_write(ptr,num)")));
void *memcpy(void *dst, const void *src, size_t num)
    __attribute__((annotate("nvm_write(dst, num);
    nvm_read(src, num)")));
```

Figure 4.3: The highlighted code informs NVHooks that `memset()` writes `num` bytes to `ptr`, and `memcpy()` reads and writes `num` bytes from `src` and to `dst`, respectively.

pre-compiled code (e.g., library functions). Since NVHooks does not have the source for pre-compiled code and library functions, it cannot automatically annotate these functions. It can, however, let the programmer describe how the method accesses NVM with respect to its arguments using C/C++ function attributes. Function attributes are a language feature that allows programmers to pass information about a function to the compiler (e.g., `always_inline`). If the programmer can describe the addresses of all persistent accesses in a function using its arguments, NVHooks can add the appropriate annotations at the call site. Programmers can use `nvm_read()` and `nvm_write()` to annotate pre-compiled functions, as we show for `memset()` and `memcpy()` in Figure 4.3. Otherwise, NVHooks does not annotate calls to the function, which could result in permanent data corruption or runtime errors if the function accesses NVM. This property is common to all runtimes that require annotations.

4.1.2 Failure-Atomicity Runtime

As discussed in Section 2.3, a failure-atomicity runtime provides a number of systems to make programming for NVM easier. These runtimes all support failure-atomic updates to NVM, and generally also provide several other capabilities. In general, they provide a method for allocating and freeing persistent memory in a persistent pool, a means of naming objects in the pool, and support for relocating the pool to a different virtual address. Adapting a runtime to use NVHooks requires the runtime developer to write three (relatively simple) files, which we call the *adaptation layer*.

Shim <pmdk-shim.c>

```c
void *to_absolute_ptr(void *p) {
  PMEMoid oid = getPMEMoid(p);
  return pmemobj_direct(oid);
}

void pre_nvm_write(void *p, size_t sz) {
  if (pmemobj_tx_stage() != TX_STAGE_NONE)
  {
    pmemobj_tx_add_range_direct(p, sz);
  }
}
```

Wrapper Header <pmdk-hooks.h>

```c
#include <libpmemobj.h>
void pmemobj_tx_commit(void);
void pmemobj_tx_abort(int err);
int pmemobj_tx_begin(PMEMobjpool *pop,
        ...);
```

Compiler Config <pmdk.yaml>

```yaml
swizzle-pointers: yes
track-nvm-writes: yes
track-nvm-reads: no
```

Figure 4.4: **Sample adaptation layer for PMDK.** The shim calls into the runtime to implement NVHooks callbacks (e.g., `pmemobj_tx_add_range_direct()` to undo-log).

The first piece of runtime integration with NVHooks is the *compiler configuration file*. This file describes to the NVHooks compiler which method annotations and optimizations are valid for the particular runtime. The programmer passes the configuration file to the NVHooks compiler via a command-line option (e.g., `-nvm-runtime=pmdk.yaml`).

The *shim* is a C file that provides an NVHooks-compliant interface to the runtime by implementing NVHooks callbacks. The runtime developer writes the shim to attach the automated NVHooks callbacks to the runtime API. The shim, for instance, implements `is_nvm()` to enable NVHooks to identify NVM accesses at execution time.

The shim connects part of the runtime's API to the NVHooks compiler, but runtimes have other methods that NVHooks does not automate. For instance, runtimes generally require programmers to use specific API calls to manage persistent memory pools, allocate persistent memory, and annotate transaction boundaries. NVHooks does not automate the use of these API calls as they are runtime or application specific, and generally, require small amounts of boiler-plate code. These runtime specific API calls are passed through to the application by an NVHooks-specific *wrapper header file* that hides API calls implemented by the shim, but exposes the remainder needed by NVHooks-aware applications.

Figure 4.4 shows a portion of the adaptation layer for PMDK. The shim implements

```
#include <pmdk-hooks.h>
typedef struct Node {
  char key[32];
  struct Node *next;
} Node;
PMEMobjpool *pop; // PMDK persistent pool
void insert(Node *prev, Node *node) {
  pmemobj_tx_begin(pop); // start a transaction
  if (is_nvm(prev)) {
    Node *pp = to_absolute_ptr(prev);
    pre_nvm_write(&pp->next, 8);
    pp->next = node;
    post_nvm_write(&pp->next, 8);
  } else prev->next = node;
  pmemobj_tx_commit(); // commit the transaction
}
```

Figure 4.5: **NVHooks obviates annotations for access tracking and pointer manipulation.** The compiler automates annotations by inserting the highlighted lines.

`pre_nvm_write()` and `to_absolute_ptr()` to call into PMDK's undo-logging and pointer manipulation functions, respectively. The wrapper header file exposes transaction management APIs such as `pmemobj_tx_begin()` to NVHooks-aware applications, and the configuration file notifies NVHooks to instrument writes and to translate NVM pointers.

4.1.3 NVHooks-Aware Application

NVHooks compiler automates annotating NVM accesses and applications only need to communicate with the runtime (via the API header file) to open persistent pools, allocate NVM regions, and specify the boundaries of failure-atomic operations (if required by the runtime). Figure 4.5 shows how an application interacts with NVHooks and the adaptation layer. It also highlights the annotations the runtime would have needed without NVHooks. In this example, we use PMDK as the runtime and study a snippet from the transactional insert function of a persistent linked-list.

If an application meets the concurrency requirements of two runtimes (see Table 2.1 as a reference), NVHooks reduces the programming effort to switch between the two runtimes. For example, consider a B+Tree that works with Atlas, uses two-phase locking, and does not modify persistent data outside critical sections. NVHooks allows switching from Atlas to PMDK by replacing the adapter and minor changes to the recovery code to use PMDK (e.g., replacing pool management, allocation, and transaction management APIs of Atlas with those of PMDK). This feature is especially useful in comparing the performance of runtimes, as we showcase in Section 4.3.

4.1.4 Building Programs

Once the adaptation layer is implemented, programmers can add persistence to an application using the target runtime. Programmers include the NVHooks-specific wrapper header file, and use it to access the runtime. Next, programmers use NVHooks to compile the code. To benefit from the NVM-specific optimization passes that work for the target runtime, programmers use a new compile option (`-nvm-runtime`) to let NVHooks know about the target runtime (e.g., `-nvm-runtime=pmdk.yaml`). Finally, programmers link the compiled binaries with the shim and the runtime to make the executable.

4.2 Compilation Passes

NVHooks introduces five passes to the LLVM compiler toolchain [59, 35]. Its first pass, the initial pass, instruments all non-stack accesses with the NVHooks callbacks (Table 4.1). The other four passes are NVM-specific optimization passes.

The initial pass operates on the compiled code before any other optimization pass (including LLVM optimization passes). The runtime's configuration file provides the initial pass with the list of callbacks it should add to the code (e.g., `pre_nvm_write()` and `to_absolute_ptr()`

in Figure 4.4).

The NVM-specific optimization passes reduce the overhead of NVHooks callbacks. They operate on the instrumented code (output of the initial pass) and run after any other conventional optimizations the compiler performs (e.g., loop unrolling). Runtimes use the compiler configuration file to specify which NVHooks optimization passes to run. For example, adding `static-check:yes` to the configuration file instructs NVHooks to apply the static boundary check optimization.

All passes operate on LLVM assembly [33], a language-independent representation that is used throughout all LLVM compilation passes. We use clang [32] as the frontend compiler that translates C/C++ code to LLVM assembly code.

NVHooks goes through the optimization passes in the same order as we introduce them in this section. The rest of this section provides the specification and implementation details of these optimization passes.

4.2.1 Static Boundary Checks

Our simplest optimization reduces the cost of `is_nvm()` callbacks by effectively inlining them. To implement it, we take advantage of the fact that the virtual memory system only uses the lower 48 bits of virtual addresses to access memory. It discards the upper 16 bits and allows addressing 256 TB of both volatile and persistent memory [66]. So long as the application does not utilize the upper 16 bits of pointers (e.g., to store metadata for a multi-version data-structure), NVHooks can use these bits to reduce the overhead of `is_nvm()` callbacks.

The static boundary check (i.e., *static-check*) optimization uses the 48^{th} bit of pointers to distinguish between NVM and volatile memory (setting it to 1 and 0, respectively). This property enables NVHooks to identify NVM pointers without calling back to the runtime and only execute swizzling and tracking callbacks for pointers with a value higher than 2^{48}, effectively inlining the `is_nvm()` callback.

The static-check has no impact on the total addressable memory and applications can still address 256 TB of memory. NVHooks uses only 1 bit and leaves the remainder 15 bits to the application and the runtime and its shim. Our shim for PMDK, for instance, uses these bits to allow applications to access up to 2^{15} different persistent pools.

4.2.2 Merging Callbacks for Field Accesses

Our next optimization pass increases the granularity of tracking NVM accesses. Tracking accesses at fine granularities is expensive as every access incurs the cost of several callbacks to the runtime. Fine granularity accesses are also inefficient due to the nature of NVM devices. For example, the access granularity of Intel DC Persistent Memory is 256 bytes — the programmer effectively uses the same bandwidth to write 64 or 256 bytes to persistent memory [51].

The *coalesce-fields* optimization pass focuses on accesses to fields of NVM data structures. Instead of tracking individual field accesses, this pass merges all callbacks into one that covers them all. Figure 4.6 illustrates the application of the coalesce-fields pass to a function that initiates the `Node` structure from Figure 4.5. The figure represents the output of the initial and coalesce-fields pass with `Initial` and `Optimized` prefixes, respectively. The optimized version calls into the runtime before the first and after the last access, thereby eliding callbacks for individual fields and allowing the NVM device to merge the accesses.

This pass expands on the LLVM's MemcpyOptimizer that merges consecutive memory accesses into single `memset()` or `memcpy()` calls [34]. It assumes programs are data-race free and relies on LLVM's alias-analysis and control flow graph infrastructure to identify accesses to data structure fields.

```
void initNewNode_Initial(Node *node) {
  if (is_nvm(node)) {
    Node *p = to_absolute_ptr(node);
    pre_nvm_write(p->key, 32);
    memset(p->key, 0, 32);
    post_nvm_write(p->key, 32);
  } else memset(node->key, 0, 32);
  if (is_nvm(node)) {
    Node *p = to_absolute_ptr(node);
    pre_nvm_write(&p->next, 8);
    p->next = NULL;
    post_nvm_write(&p->next, 8);
  } else node->next = NULL;
}
void initNewNode_Optimized(Node *node) {
  if (is_nvm(node)) {
    Node *p = to_absolute_ptr(node);
    pre_nvm_write(p, 40);
    memset(p->key, 0, 32);
    p->next = NULL;
    post_nvm_write(p, 40);
  } else {
    memset(node->key, 0, 32);
    node->next = NULL;
  }
}
```

Figure 4.6: **Coalescing access tracking callbacks** for writes to various fields of a data structure enables reducing the overhead of tracking accesses to NVM.

```
void updateKey_Initial(Node *node, char *key) {
  size_t sz = strlen(key);
  for (size_t i = 0; i <= sz; i++) {
    if (is_nvm(node)) {
      Node *p = to_absolute_ptr(node);
      pre_nvm_write(&p->key[i], 1);
      p->key[i] = key[i];
      post_nvm_write(&p->key[i], 1);
    } else node->key[i] = key[i];
  }
}
void updateKey_Optimized(Node *node, char *key) {
  size_t sz = strlen(key);
  Node *p = node;
  if (is_nvm(node)) {
    p = to_absolute_ptr(node);
    pre_nvm_write(p->key, sz + 1);
  }
  for (size_t i = 0; i <= sz; i++) {
    p->key[i] = key[i];
  }
  if (is_nvm(node))
    post_nvm_write(p->key, sz + 1);
}
```

Figure 4.7: **The coalesce-loops optimization pass** reduces the overhead of NVHooks annotations for loops.

4.2.3 Merging Callbacks for Loops

Our next optimization reduces the cost of callbacks for loop-based accesses to arrays. Loops that traverse arrays are a useful target for optimization — they are a common code structure and result in callbacks at many indices. We show an example of this situation in Figure 4.7, in the top method (`updateKey_Initial()`), where we highlight the annotations added by the initial pass; there are four callbacks per iteration.

Our *coalesce-loops* optimization makes the assumption that arrays are located in contiguous regions of either NVM or volatile memory — a single array cannot span the two memory types. Furthermore, it assumes that the virtual address of the array does not change during the execution of the loop. This optimization uses these assumptions to eliminate annotations inside loops by merging them into pre and post loop callbacks. The coalesce-loops pass reduces the number of callbacks from four callbacks per iteration to four callbacks per loop.

The coalesce-loops pass identifies array accesses inside loops where the reference to the array as well as the lower and upper bounds of the loop are loop-invariant. Since the pointer to the array is loop-invariant, the pass can use loop-invariant-code-motion [5, 1] to move `to_absolute_ptr()` and `is_nvm()` outside the loop. Next, the pass uses loop-distribution [58] to break the loop into three loops: one containing only pre-access callbacks (L_P), another with post-access callbacks (L_S), and the third loop with the rest of the instructions in the original loop (L_M). L_P and L_S come before and after L_M, respectively. The semantics of the tracking callbacks allow the pass to apply loop-unrolling [25] to L_P and L_S; it then reduces each loop into a single callback that covers all the NVM accesses inside L_M. The bottom function in Figure 4.7 (`updateKey_Optimized()`) shows the result of the coalesce-loops pass. The optimized loop only calls into the runtime from outside the loop.

```
void safeUpdate_Initial(Node *node, char *key) {
  if (is_nvm(node)) {
    Node *p = to_absolute_ptr(node);
    pre_nvm_write(p->key, 32);
    memset(p->key, 0, 32);
    post_nvm_write(p->key, 32);
  } else memset(node->key, 0, 32);
  size_t sz = strlen(key);
  if (is_nvm(node)) {
    Node *p = to_absolute_ptr(node);
    pre_nvm_write(p->key, sz);
    memcpy(p->key, key, sz);
    post_nvm_write(p->key, sz);
  } else memcpy(node->key, key, sz);
}
void safeUpdate_Optimized(Node *node, char *key) {
  Node *p;
  if (is_nvm(node)) {
    p = to_absolute_ptr(node);
    pre_nvm_write(p->key, 32);
    memset(p->key, 0, 32);
  } else memset(node->key, 0, 32);
  size_t sz = strlen(key);
  if (is_nvm(node)) {
    memcpy(p->key, key, sz);
    post_nvm_write(p->key, 0, 32);
  } else memcpy(node->key, key, sz);
}
```

Figure 4.8: **Removing redundant access tracking callbacks** allows reducing the cost of annotations without altering the persistence semantics of runtimes.

4.2.4 Removing Redundant Access Tracking

This optimization pass identifies and removes unnecessary access tracking callbacks. The initial pass can generate redundant annotations for functions that access the same NVM address multiple times. We show an example of these annotations in the top function in Figure 4.8 (`safeUpdate_Initial()`) where we zero a buffer then copy a string into it. This code results in redundant `pre_nvm_write()` and `post_nvm_write()` on lines 11 and 6, respectively. If the runtime uses UNDO logging to enforce failure-atomicity, this duplication is unnecessary. In an UNDO logging runtime, the runtime logs the original value of the modified address and, in the event of a failure, the runtime will *undo* any modifications made in the failure-atomic code region. Thus, after the first modification to an address in a failure-atomic region, the runtime does not need to log subsequent modifications to that address.

The *no-extra-tracking* pass uses a modified version of common-subexpression-elimination [19] to remove these duplicated callbacks. Consider the instruction sequence (A) in Figure 4.9 (the left box), where `pre_callback` and `post_callback` are pre and post access callbacks, respectively, and `nvm_access` represents an NVM read or write. The optimization pass can reduce (A) to (B) so long as at least one of the accesses ($\texttt{nvm\_access}_S$) is the superset of all other accesses in the instruction sequence. The bottom function in Figure 4.8 (`safeUpdate_Optimized()`) shows the result of applying no-extra-tracking to the output of the initial pass.

This optimization also has a special case surrounding allocation for UNDO logging runtimes. Some runtimes (e.g., PMDK and Atlas) do not require tracking writes to newly allocated NVM blocks — the writes will be ignored if the failure-atomic code region is interrupted by failure. As such, these writes need no call to the `pre_nvm_write()` callback, and we can coalesce the `post_nvm_write()` callback to cover the entire allocated block. Runtimes can support this optimization by adding `track-newly-allocated:no` and the name of their NVM allocation functions (e.g., `pmalloc:pmemobj_alloc` for PMDK) to their compiler configuration files (Figure 4.4).

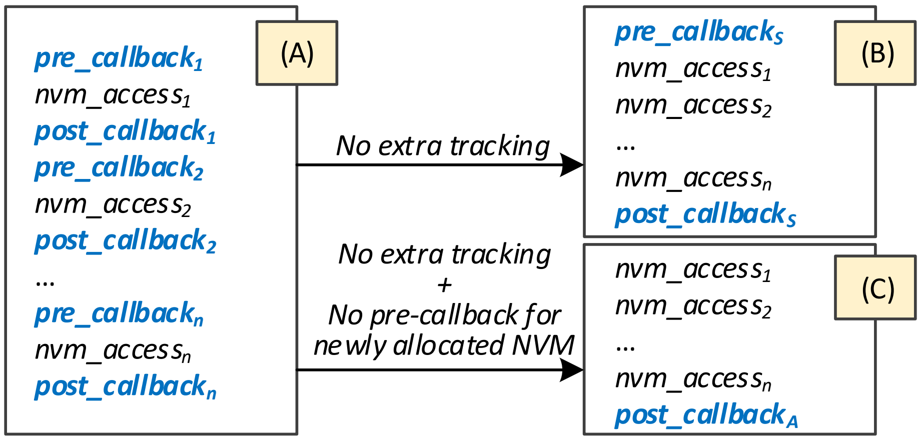


Figure 4.9: **The no-extra-tracking pass** reduces the cost of access tracking callbacks by removing redundant callbacks.

The no-extra-tracking pass then reduces (A) to (C) in Figure 4.9 so long as all accesses are to the newly allocated block, where `post_callback`$_A$ provides the address and the size of the entire allocated NVM block to the runtime.

4.3 Evaluation

In this section, we measure the performance overhead of NVHooks instrumentation and the effectiveness of its optimization passes to answer the following questions:

- Can we match the performance of handcrafted, failure-atomic applications by applying NVHooks optimizations on the initial instrumentation?
- What are the performance implications of NVHooks optimizations on various NVM runtimes?
- Which NVM runtime provides the best performance?
- What is the programming cost of retargeting benchmark applications to various runtimes?

4.3.1 Evaluation Setup

We use a HashMap and a B+Tree to measure the performance implications of NVHooks annotations and its optimization passes. For each data structure, there are two versions of the code: a volatile and a handcrafted, failure-atomic version. The volatile version is not failure-atomic; we use NVHooks (the initial pass) and each of the runtimes from Table 4.2 to add failure-atomicity to the volatile versions. The handcrafted versions contain hand-optimized annotations, use PMDK as the runtime, and provide the reference for measuring the effectiveness of NVHooks optimizations.

We use the transactional (handcrafted) HashMap from the PMDK repository [42] and make it thread-safe by creating 64 instances of the HashMap and protecting each one with a reader-writer lock. We create the volatile version of the HashMap by removing the failure-atomicity code and PMDK annotations from its original version.

The B+Tree is an in-house implementation of a thread-safe, volatile tree. It uses reader-writer locks at the granularity of individual nodes, stores keys in the internal nodes, and adds both the key and the value to the leaf nodes. We created the handcrafted, failure-atomic version of the tree by carefully annotating it with PMDK API.

Benchmark

The benchmark uses eight threads to run YCSB [21] workloads against different versions of the data structures and reports average throughput for ten runs. It first populates each structure with 1 million entries (YCSB-Load) and then runs two workloads with a combination of read and update operations: a write-dominant (YCSB-A with 50% read and 50% update) and a read-dominant (YCSB-B with 95% read and 5% update) workload. We use 8 and 32-byte keys for the HashMap and the B+Tree, respectively. The value size is 1024 bytes for both structures.

Table 4.2: **The list of failure-atomicity runtimes** used to evaluate NVHooks.

Runtime	Summary
PMDK	Intel's persistent memory development kit that provides a collection of tools and libraries for constructing persistent programs. PMDK uses both redo- and undo-logging internally.
Kamino-Tx	It provides lightweight transaction support for persistent memory programming. Kamino-Tx offers atomic in-place updates and reduces the overhead of creating copies of persistent objects in the critical path while ensuring crash consistency.
Atlas	It provides failure-atomicity for lock-based programs and adopts compiler support to reduce the cost of annotating NVM accesses. Atlas uses the semantics of critical sections to create globally consistent snapshots.

Runtimes

We integrated three runtimes (see Table 4.2) with NVHooks: PMDK [86], Kamino-Tx [69], and Atlas [14]. Their corresponding adaptation layers provide the implementation of NVHooks callbacks and expose the runtime-specific persistent pool management, allocation, and transaction management APIs to the application.

We use these APIs to write a few lines of code to add the runtimes to each data structure. For instance, we wrote 32, 15, and 36 lines of code to integrate PMDK, Atlas, and Kamino-Tx with the volatile HashMap, respectively. By integrating with NVHooks, all adapted runtimes support the relocation of persistent pools, which the vanilla Atlas does not provide.

Testbed

We run the benchmarks on a platform with two 24-core Intel Cascade Lake SP processors, running at 2.2 GHz. The platform has a total of 192 GB of DRAM and 1.5 TB (6 $\times$256 GB) of Intel Optane DC Persistent Memory directly attached to each processor [48]. We pin all benchmarks to one of the processors to avoid accesses to remote DRAM and NVM. All experiments use ext4 to manage persistent pools and directly access NVM pages via DAX [61]. The benchmarks run in a Docker [71, 10] container on Fedora 27 (Linux kernel version 4.19) and use PMDK 1.6, LLVM 7.0, and Clang 7.0.

4.3.2 Overhead of Automating Annotations

We used hand-annotated (handcrafted) versions of the B+Tree and HashMap for comparison against their NVHooks-aware counterparts. We wrote and modified 720 lines of code to manually adapt the 629-line volatile B+Tree implementation for PMDK. For the handcrafted HashMap, we used the original PMDK HashMap, which 42% of its code is logging and swizzling annotations that NVHooks can automate.

NVHooks optimizations reduce the cost of initial instrumentation by up to 27% and provide 104% and 97% of the throughput of the handcrafted B+Tree and HashMap data structures, respectively. Figure 4.10 shows the evaluation results. *Initial* and *Optimized* refer to the configurations with all NVHooks optimizations disabled and enabled, respectively. All benchmarks use PMDK as the runtime.

4.3.3 Benefits of the Optimization Passes

We use the volatile data structures from Section 4.3.1 to evaluate the effect of NVHooks optimizations on the cost of automatic instrumentation. These benchmarks also allow comparing the performance of various runtimes in the presence of write-only, write-dominant, and read-dominant workloads. We incrementally apply NVHooks passes to the data structures in the following order and report the performance in Figure 4.11.

The Initial Pass

NVHooks initial pass instruments all NVM writes for PMDK, Kamino-Tx, and Atlas. Both PMDK and Kamino-Tx use undo-logging and persist updates to NVM (i.e., flush the cache-lines and issue memory fences) at the commit point. Thus, they only require NVHooks to track writes and do not need the post-access callbacks. NVHooks follows NVM writes with `post_nvm_write()` for Atlas as it immediately persists every NVM write after it is issued. For all benchmarks, NVHooks instruments reads and writes with `is_nvm()` and `to_absolute_ptr()` callbacks.

Both benchmarks provide better performance when using PMDK as the runtime, especially in the presence of write-only and write-dominant workloads. Atlas requires more instrumentation and persists individual writes to NVM, which significantly impact the performance of write operations. Kamino-Tx aims to reduce the cost of undo-logging on the critical path by using a background thread to complete logging. This approach is not effective for write-dominant workloads and data structures with coarse-grain locks. The background thread falls behind the foreground threads for such workloads, which exposes the cost of logging and the overhead of synchronizing the background and foreground threads to the critical path.

PMDK outperforms Atlas by $2\times$ and Kamino-Tx by $3\times$ for the B+Tree benchmark and across all workloads. Benefits of using PMDK are most significant for the HashMap benchmark, as it uses coarse-grain locks and performs small, random NVM writes. In comparison to Atlas and Kamino-Tx, PMDK provides $2\times$ and $8\times$ higher throughput for the HashMap benchmark (across all workloads).

Static-Check

The first optimization that NVHooks applies is static-check, which replaces `is_nvm()` callbacks with comparisons. Static-check is more effective for read-dominant workloads and reduces the cost of the initial pass by up to 5%.

Coalesce-Fields

The second optimization is coalesce-fields that merges callbacks for consecutive accesses to the fields of NVM structures. The HashMap writes to fields of each entry while serving insert and update operations, and coalesce-fields can improve the performance of these operations by up to 17%. As for the B+Tree, the proportion of writes to the fields of structures is insignificant in comparison to total writes to NVM. Thus, coalesce-fields only marginally improves the performance of the B+Tree benchmarks.

Coalesce-Loops

Next, NVHooks applies the coalesce-loops optimization to merge the callbacks for loops. Both data structures only write to NVM from inside a loop to serve insert operations (YCSB-Load) – the B+Tree and the HashMap use loops to grow in response to the growing number of key-value pairs. The coalesce-loops pass increases the throughput of insert operations for PMDK, Atlas, and Kamino-Tx by up to 8%, 32%, and 39%, respectively.

No-Extra-Tracking

This optimization avoids unnecessary `pre_nvm_write()` callbacks (i.e., undo-logging) and is the last optimization that we enable. It provides the highest performance improvement for the write-only workload, as both data structures only allocate NVM while running YCSB-Load. The no-extra-tracking optimization improves the throughput of insert operations by up to 119% across all benchmarks.

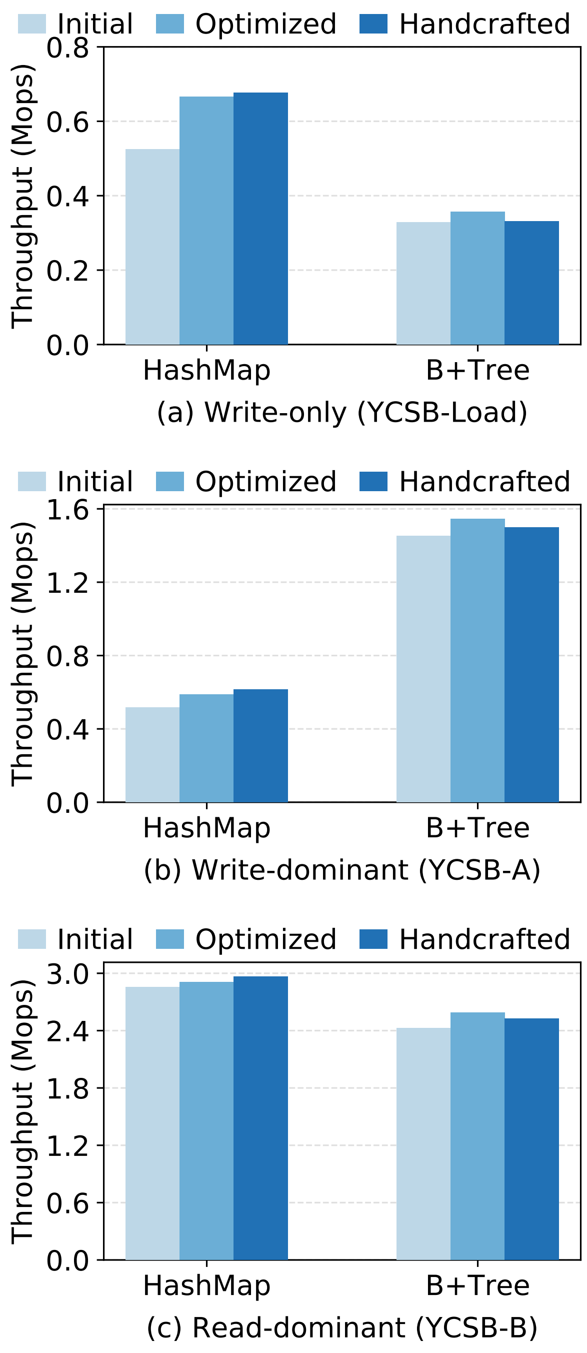


Figure 4.10: **Measuring the overhead of automating annotations:** we compare the performance of NVHooks-aware data structures (initial and optimized) against their optimally annotated versions (handcrafted).

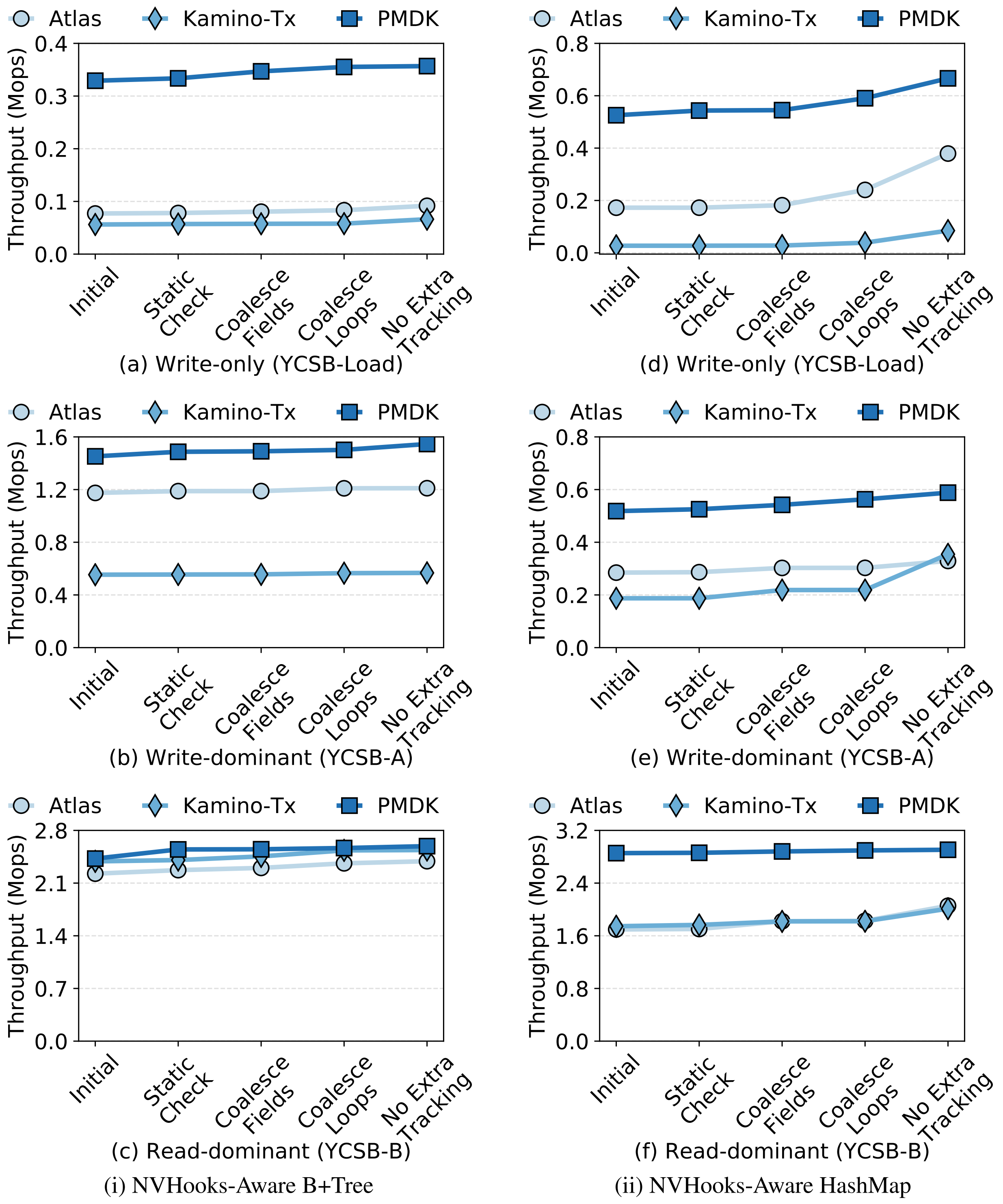


Figure 4.11: **Evaluating NVHooks optimizations**: we incrementally enable optimizations – the horizontal axes show the order in which we apply the optimizations. No-extra-tracking shows the performance after applying all optimizations.

4.4 Related Work

Previous work has provided compiler support to facilitate NVM programming, but NVHooks, to our knowledge, is the first system that provides a generic mechanism to instrument programs for various runtimes and NVM-specific optimizations to reduce the cost of those runtimes.

NVM-Direct [12] and NVM-C [24] introduce new constructs to the C programming language (e.g., non-volatile type qualifiers) to differentiate between volatile and non-volatile memory accesses. NVHooks relies on the standard C/C++ and does not impose changes to the language. As a result, it avoids radical changes to existing programs and frees programmers from the burden of learning a new syntax.

Atlas [14] and Breeze [70] rely on the compiler to instrument memory accesses and enable the runtime to intercept persistent reads/writes or other instructions of interest (e.g., `lock()` and `unlock()` for Atlas). NVHooks improves on this approach by providing a series of optimizations to reduce the performance impact of the runtime interception as well as generic APIs that allows switching between different runtimes (e.g., PMDK and Atlas) with minor changes to the program.

iDo [63] uses compiler-support to identify idempotent instruction sequences – it instruments idempotent regions instead of individual stores to persistent data. NVHooks provides generic instrumentation and does not extract runtime-specific semantics (e.g., idempotent regions).

NVM runtimes facilitate persistent programming by providing the means to add failure-atomicity to programs [86, 18, 69, 100, 23]. They stifle persistent programming by introducing new syntax or requiring runtime-specific annotations. NVHooks provides generic instrumentation and optimizations that reduce the programming overhead of C runtimes without impacting their performance. Its optimizations are also applicable to other languages (e.g., C++).

4.5 Summary

This chapter describes NVHooks, a system that uses the compiler to automate annotating NVM accesses and avoid disruptive changes to programs. NVHooks introduces a series of NVM-specific optimization passes that allow automatically instrumented programs to match the performance of their handcrafted, optimally annotated counterparts. It provides support for runtime-specific adaptation layers to mitigate the programming effort of retargeting applications towards using a different NVM runtime. The chapter also offers micro-benchmarks to evaluate and compare the performance of different NVM runtimes. Our evaluation shows that applying NVHooks to popular data structures creates failure-atomic structures that match the performance of their optimally annotated versions.

Acknowledgments

This chapter contains material from “NVHooks: A Flexible, Optimizing Compiler for Non-Volatile Memory Programming”, by Amirsaman Memaripour, Yi Xu, Steven Swanson, and Joseph Izraelevitz which is submitted to the 25th Architectural Support for Programming Languages and Operating Systems (ASPLOS 2020). The dissertation author is the primary investigator and first author of this paper.

The author would like to thank Intel Corporation for providing access to the evaluation testbed as well as the members of the NVSL research group for their insightful comments.

Chapter 5

Easy and Fast Persistence for Volatile Data Structures

Emerging non-volatile main memory (NVM) technologies such as 3D XPoint [22, 48] offer higher density than DRAM with comparable latency and bandwidth, allowing computer architects to attach them to processors via the memory bus. Programs can then use load and store instructions to access persistent data directly. Bypassing the storage stack and directly accessing NVM is essential for unleashing the performance benefits that NVMs offer [90]. However, this strategy requires careful reasoning to ensure a consistent-state in NVM in the wake of a crash — data in the caches will not survive [69, 50].

NVMs appear to be an exceptional opportunity for building fast, persistent, data structures, and researchers have approached this problem in two ways. NVM failure-atomicity libraries (e.g., [18, 95]) allow programmers to delineate *failure-atomic* updates to persistent data - writes within the update become persistent all at once. By identifying failure-atomic code regions and persistent writes, programmers can adapt an existing data structure to NVM using these libraries [9, 20]. Alternatively, researchers have built custom data structures from scratch for NVM (e.g., [94, 80]). Unfortunately, both of these design options are labor-intensive, require detailed program knowledge, and are a fertile source of subtle errors [100]. Furthermore, these options effectively ignore the wide range of useful, volatile data structures currently available (e.g., the C++ Standard Template Library or the Java Collection data structures).

In this chapter, we introduce *Pronto*, a library that reduces the programming effort required to add persistence to off-the-shelf, volatile data structures, preserving the original operation of the data structure and, for concurrent data structures, their concurrency scheme. Furthermore, Pronto minimizes the performance overhead of this transformation by moving almost all durability-related code off the critical path.

Pronto transforms the volatile data structure by changing every operation on the original data structure into a failure-atomic operation. Adding Pronto to an existing volatile data structure is simple. For sequential data structures, adding Pronto requires only adding a thin wrapper class around the data structure's API and using the Pronto allocator. For concurrent data structures, adding Pronto also requires one additional line of code per API method.

Pronto uses a novel mechanism called *Asynchronous Semantic Logging* (ASL) to convert each operation on a volatile data structure into a failure-atomic operation. ASL records the arguments and execution order of each update operation performed on the data structure rather than recording the details (e.g., pointer updates) of how the data structure changed. For instance, ASL would record the insertion of an item into a binary tree rather than recording how the tree's internal structure changed. ASL is analogous to operation logging in database systems [76], but addresses the specific needs of logging for persistent, in-memory data structures. To recover from program or system failures, Pronto plays back semantic logs for a structure to reconstruct its most recent consistent state (read-only operations need not be logged at all in Pronto). To limit the cost of replaying semantic logs, Pronto creates periodic snapshots.

In this chapter, we describe the Pronto system and demonstrate that many common, non-persistent data structure implementations (e.g., RocksDB's MemTable and containers from the GNU C++ Standard Template Library) are readily amenable to a Pronto adaptation with minimal programming effort, and, furthermore, these new Pronto adaptations perform better than other failure-atomic variants.

This chapter of the dissertation makes the following contributions:

- It introduces ASL, a new software mechanism that reduces the programming effort and performance overhead of adding failure-atomicity to volatile data structures.
- It explores the design decisions and correctness constraints of ASL in the context of NVMs.
- It provides an implementation for Pronto and evaluates its performance.
- It demonstrates how to use Pronto to convert both sequential and concurrent volatile data structures into persistent data structures with only a few lines of code.

The rest of this chapter is organized as follows. We discuss the design and implementation of Pronto in Section 5.1 and Section 5.2, respectively. Section 5.3 presents the evaluation results and puts Pronto's performance in perspective. We discuss related work in Section 5.4 and summarize in Section 5.5.

5.1 Design Overview

Pronto adds persistence to volatile data structures with minimal code changes and moves the cost of durability off the critical execution path. It accomplishes this by creating asynchronous semantic logs (ASLs) that allow for the reconstruction of the latest consistent state of the data structures during recovery from a failure. The semantic logs record every operation invoked on the object, and its arguments, and this logging is done *asynchronously*, that is, in parallel, with the actual operation.

In terms of the programming cost, our ASLs are useful in that they avoid the need to log (and, consequently, annotate) fine-grained changes to the underlying data structure. With semantic logging, we need only log the method call and its arguments — replaying operations after a failure is sufficient to recover the data structure's state. Code changes, as a consequence, are minimal — for sequential data structures we need only intercept the public methods of the data structure and its allocator, and concurrent data structures also require one more line of code.

Our ASLs also reduce the performance cost of persistence by logging asynchronously, especially for slower NVMs. By decoupling log creation from operation execution and performing logging in parallel, ASL can drastically reduce the performance cost of persistence. In fact, if the logging is quick enough, Pronto can almost completely hide the overhead of logging. By moving such operations off the critical path, programmers can hide the cost of such operations.

Pronto is broadly applicable to most data structures, so long as they meet certain criteria (all criteria are generally common to standard data structures). First, the data structure and its interface must be properly *encapsulated* and *deterministic* so that modifications only occur through public methods and the effect of those methods is only a function of the current state of the data structure and the arguments to the method. In effect, this means that the methods cannot read or write global variables. Second, if the data structure is thread-safe (i.e., supports concurrent accesses), it must be *linearizable* [38, 72].

An update to the data structure is linearizable if the data structure's synchronization mechanisms (e.g., locks) ensure that the effect of multiple (potentially parallel) updates is the same as those updates being applied one at a time in some order [38]. Linearizability is the common correctness condition for concurrent data structures, and most practical data structures meet this condition (e.g. [92, 30, 60]). For any linearizable data structure that uses locks to order updates that do not commute, Pronto provides failure-atomicity with no loss of concurrency.

These requirements are not onerous in practice, since they closely correspond to common data structure design practices. Most container libraries (e.g., the C++ STL) and many custom data structures (e.g., the core data structures of RocksDB [29] and Memcached [31]) meet them.

This section describes the design of Pronto. We begin with a description of the Pronto system and runtime. Next, we describe Pronto's programming interface and elaborate on the durability and concurrency semantics that Pronto offers. Finally, we give examples for using Pronto for both sequential and concurrent data structures.

5.1.1 Pronto System Overview

The Pronto runtime maintains three entities for each persistent data structure it manages. An *asynchronous semantic log*, a volatile *online image* of the data structure in volatile memory, and a persistent *snapshot* of the data structure. This subsection describes Pronto's runtime in terms of its ASL, memory management and snapshot mechanisms.

Asynchronous Semantic Logging

Pronto's semantic logs record the high-level updates that the data structure undergoes rather than the fine-grain changes to the memory that holds it. For example, Pronto only creates a single log record for inserting a new key-value pair to a B-Tree, unlike undo-logging that requires recording the fine-grain changes to the B-Tree's structure that happen as part of the insert. Since recording the high-level operations is usually fast, ASL is more efficient than normal write-ahead logging.

For clarity, we describe ASL in terms of method invocations (or "updates," read-only operations need not be logged) on container-style objects (e.g., linked lists, hash maps, and vectors), but ASL will work for any deterministic, linearizable (or sequential) data structure with a well-defined set of operations that Pronto's ASL can record.

For every operation that modifies the data structure, Pronto creates a *semantic log entry*, a persistent record that records the method invoked (e.g., an insert) and a copy of its arguments.

Besides an ASL and a persistent snapshot, Pronto maintains a volatile online image for each data structure. The online image reflects the current state of the data structure. In addition to logging operations, Pronto applies each operation to the volatile version and read-only operations

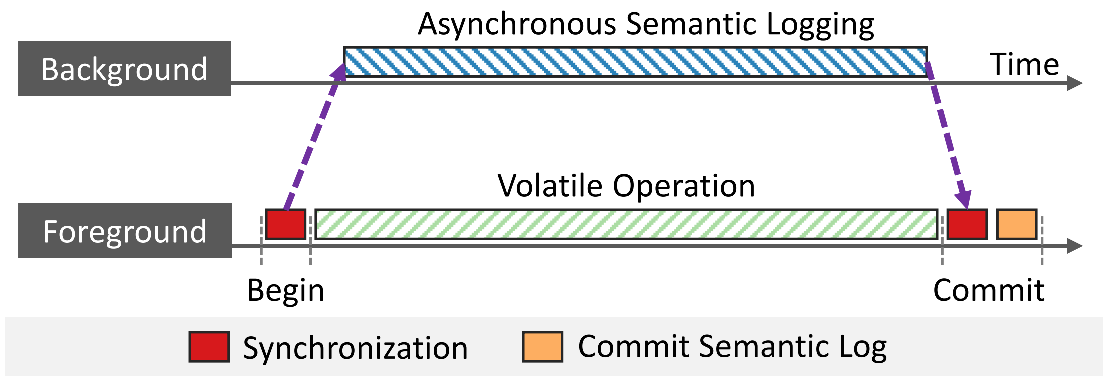


Figure 5.1: Communications between the foreground and background execution paths to guarantee every committed semantic log represents a completed update operation.

run against it.

After a crash and upon restart, Pronto can recreate the volatile online image (i.e., recover the last consistent state of the data structure) by replaying the ASL. A snapshot mechanism described below keeps the cost of recovering the volatile online image manageable.

The key optimization that Pronto makes is to perform logging in an *ASL thread* that runs in parallel with the foreground update to the online image. If applying an update to the online image is slower than logging its arguments, Pronto can hide the ASL's latency.

Under ASL, an operation is not complete until both the update to the volatile online image is finished, and the semantic log entry is persistent. To enforce this requirement, the foreground thread must wait for the ASL thread to finish logging before any of the update's effect becomes visible to other threads. In practice, this means synchronizing with the ASL thread before releasing any lock that protects the operation's effects (changes) from being visible to other concurrent operations. This guarantees that the commit order of ASLs agrees with the execution order of updates to the data structure that do not commute (e.g., `insert(`K_1`, `V_1`)` and `erase(`K_1`)`).

Figure 5.1 illustrates the parallel execution of the foreground thread (bottom) and ASL thread (top). ASL operations are blue, DRAM updates are green, and synchronization is red. *Begin* marks the beginning of both logging and update execution. *Commit* marks completion of

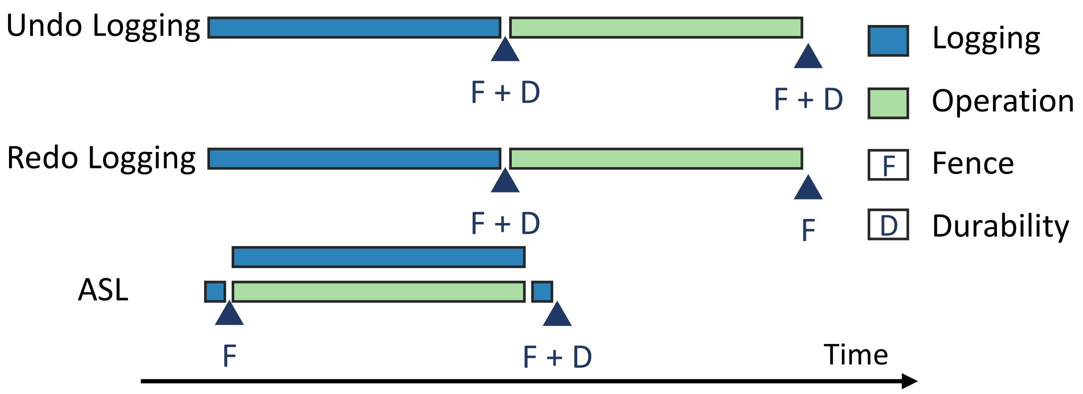


Figure 5.2: Comparing the execution path of ASL against undo-logging and redo-logging. The operation represents a deterministic update, such as inserting a new node to a tree.

the operation. The small orange box in the foreground thread is the commit point for the ASL log entry when the entry becomes persistent.

Figure 5.2 compares ASL with undo-logging and redo-logging [69]. ASL allows executing the *Logging* code in parallel with the *Operation* and decreases the execution complexity of memory-barriers and cache-line flushes in the critical path, thereby reducing the total overhead of adding persistence to volatile data structures.

Memory Management and Addressing

Pronto provides a volatile memory allocator that manages a contiguous region of memory to hold the online, volatile image. Data structures must use the allocator for any internal objects (e.g., links in a linked list) and applications must use the allocator for objects they pass to data structure methods via a pointer. This requirement ensures that the data structure and all memory reachable from it are fully contained within the memory region the allocator manages.

The online image of a data structure uses native pointers for addressing, so it is not relocatable (i.e., it must always reside at the same virtual address). This is not a fundamental limitation of Pronto or ASL, but it is necessary to support the easy conversion of volatile data structures into persistent data structures without compiler support. Previous work has shown how to ensure relocatability with a compiler [70]. Those techniques would apply to Pronto. We describe the allocator in detail in Section 5.2.2.

Pronto also manages NVM space for semantic logs and snapshots. It allocates space by mapping NVM files into the program's address space. ASL uses the mapped NVM space as a circular buffer and writes over old semantic log entries that precede the latest snapshot. Section 5.2.1 provides additional details.

Snapshots

Pronto provides a snapshot mechanism that works closely with its volatile memory allocator to take periodic snapshots of online images. Snapshots, which are durably stored on NVM, reduce the ASL storage requirements and improve recovery time since Pronto only needs to store ASL entries since the last snapshot and replay those entries after a crash.

Snapshots contain a persistent copy of the (volatile) memory pages used by the the volatile online images of the data structures along with a description of currently allocated memory (provided by Pronto's allocator). Pronto always keeps the latest snapshot on NVM to ensure fast recovery.

The application can change the frequency of snapshots to trade off between snapshot overhead and recovery time. We describe the mechanics of taking a snapshot in Section 5.2.3 and measure its performance impact in Section 5.3.7.

5.1.2 Using Pronto

Pronto offers a simple C++ interface for creating persistent data structures with ASL support. The interface provides access to Pronto's volatile memory allocator, a mechanism to specify the boundaries of operations that the ASL will record, and a directory that allows accessing persistent data structures across restarts. Table 5.1 summarizes the interface.

Programmers can use Pronto to add persistence to both sequential (single-threaded) and concurrent (thread-safe) volatile data structures. This section provides an example of using Pronto

Table 5.1: Pronto's programming interface

`PersistentObject(name)`	Every persistent object must inherit from this class. Pronto identifies objects by their unique name (provided to the constructor) and maintains a persistent directory for mapping names to references to objects.
`get_object<T>(name)`	Uses the persistent directory to return a reference to the persistent object of type `<T>` identified by `name`.
`op_begin(args)`	Marks the beginning of a failure-atomic operation, which accepts `args` as input, and initiates ASL.
`op_commit()`	Waits for the operation's ASL to complete and then marks the semantic log entry as committed.

`palloc(size)`
`prealloc(ptr, size)`
`pfree(ptr)`
	Programmers must replace `malloc()`, `realloc()` and `free()` with `palloc()`, `prealloc()` and `pfree()` for managing memory for their data structures (e.g., using GCC's `--wrap` flag) to allow Pronto create periodic asynchronous snapshots.

for each case and elaborates on the requirements for using Pronto with concurrent data structures.

Adding Pronto to Sequential Data Structures

Adding Pronto to a volatile single-threaded data structure is straight-forward.

The programmer adds Pronto by creating a wrapper object for the volatile data structure, and the wrapper object inherits from `PersistentObject`. Extending the `PersistentObject` superclass provides a naming mechanism to enable programmers to access instances of the class across restarts using a unique name. Any instance of this new class is a persistent object, where the latest consistent state of its internal data structure survives failures and each public method executes as a failure-atomic operation.

The wrapper object contains a member copy of the original data structure and wrapper methods for every function in the data structure's API. For any method that updates the wrapped data structure, the programmer inserts a special `op_begin()` at the top of the corresponding wrapper method and `op_commit()` at the end. The `op_begin()` method triggers semantic log

```
template <class T>
class PVector : PersistentObject {
  // Alloc conforms with STL allocator
  // Alloc.allocate() calls palloc()
  // Alloc.deallocate() calls pfree()
  vector< T, Alloc<T> > *vVector;
public:
  PVector(string name):PersistentObject(name) {
    // alloc is an instance of Alloc<T>
    // *new* uses palloc() for allocation
    vVector = new vector< T, Alloc<T> >(alloc);
  }
  void push_back(T value) {
    op_begin(value);
    vVector->push_back(value);
    op_commit();
  }
  void pop_back() {
    op_begin();
    vVector->pop_back();
    op_commit();
  }
  size_t size() {
    // no logging needed for read-only ops
    return vVector->size();
  }
};
```

Figure 5.3: Creating a template persistent vector using the STL's vector container and Pronto.

entry creation and takes a copy of the input arguments, while the `op_commit()` method commits the operation. Note that Pronto only requires instrumenting public update methods, while existing NVM libraries (e.g., PMDK [44]) require tracking all writes to NVM. Pronto uses a simple source preprocessor to provide every `op_begin()` with a pointer to the public method that calls into it, which enables mapping semantic logs to their matching public methods during recovery. This preprocessor also generates code to convert each semantic log entry to a corresponding method call and automate replaying semantic logs at recovery. Pronto assumes that the implementation of the data structure does not change before recovery.

Finally, the programmer must use Pronto's memory allocator to manage memory for the wrapped data structure.

Figure 5.3 is an example of using Pronto's APIs from Table 5.1 to create a persistent version of the `vector` container from the GNU C++ Standard Template Library (STL). We create a wrapper class (`PVector`) for the `stl::vector` that extends `PersistentObject`. Since STL containers support user-specified allocators, we pass a reference to Pronto's allocator to the constructor of the `stl::vector`. Update methods of the STL vector are wrapped and surrounded by `op_begin()` and `op_commit()`. For the sake of simplicity, we only illustrate the implementation of the constructor, `push_back()` and `pop_back()` methods.

Adding Pronto to Concurrent Data Structures

Pronto supports a wide class of concurrent data structures that synchronize internally using locks. So long as they meet the standard correctness condition of *linearizability*, Pronto can make them resilient to power outages with simple code changes. In a linearizable (concurrent) data structure, each method appears to occur at some atomic instant in time between its invocation and return; putting the operations in this order gives us a *linearization order*, and the concurrent data structure must behave exactly like a sequential data structure executing the operations in this order [38, 72].

Converting a thread-safe data structure in Pronto follows the exact same requirements as a sequential data structure, save for the call to `op_commit()`, which, instead of being called in the wrapper object, is called within the wrapped data structure at a programmer identified point. For proper integration with Pronto, the order in which operations call `op_commit()` must be a valid linearization order. Put more simply, if two data structure operations cannot (semantically) commute (e.g., performing insert(k_1,v_1) and erase(k_1) against a hash-map), then their calls to `op_commit()` must occur in program order.

In practice, this requirement can be trivially met by ensuring that the lock that protects the operation's data structure modifications also protects the call to `op_commit()`. As a consequence, programmers can preserve their existing isolation for operations and avoid disruptive changes to the program to use a new synchronization interface.

If Pronto is properly integrated into a linearizable data structure according to the above requirements, it generates a *durably linearizable* data structure [50], in which the data structure's operations not only appear to atomically occur in between their invocation and response, but also become persistent at the same instant. For blocking data structures that use locks to enforce linearizability, Pronto provides failure-atomicity with no loss of concurrency.

Figure 5.4 shows an example of using Pronto with a thread-safe, concurrent hash-map. Since STL containers are not thread-safe, we use locks to serialize accesses to each bucket of the hash-map. By committing semantic logs before releasing the per-bucket locks, we force semantic logs to commit in the order that the program performs non-commutable operations (e.g., insert(K_1, V_1) and insert(K_1, V_2)), but in either order for operations that commute (e.g., insert(K_1, V_1) and insert(K_2, V_2) when $K_1 \neq K_2$).

Requirements for Concurrent Data Structures

The following equation formalizes the requirement for committing ASL entries for concurrent updates to a linearizable data structure. H_S and H_P denote sequential and parallel

```
template <class T>
class HashMap : PersistentObject {
  const unsigned Buckets = 32;
  unordered_map<T, T, hash<T>, equal_to<T>, Alloc<T>> *vMaps[Buckets];
  mutex locks[Buckets];
public:
  HashMap(string name):PersistentObject(name) {
    // initialize vMaps and per-bucket locks
  }
  void insert(T key, T value) {
    op_begin(key, value);
    unsigned b = hash<T>{}(key) % Buckets;
    locks[b].lock();
    vMaps[b]->insert(make_pair(key, value));
    op_commit();
    locks[b].unlock();
  }
};
```

Figure 5.4: Creating a persistent, concurrent hash-map using Pronto and C++ STL's unordered_map container.

execution histories, respectively, and $H_S \approx H_P$ denotes that H_S is a valid linearization order of H_P. op_1 and op_2 represent two atomic operations that occur in both H_S and H_P. The relations $<_{H_S}$ and $<_{commit}$ refer to the H_S order and the Pronto commit order respectively.

$$if \quad \forall_{H_S \approx H_P} \quad op_1 <_{H_S} op_2 \quad then \quad op_1 <_{commit} op_2 \tag{5.1}$$

This requirement allows Pronto to reconstruct persistent objects after failures by replaying semantic logs sequentially according to their commit order – as the commit order of semantic logs represents a valid sequential execution order of their corresponding failure-atomic operations.

5.2 Implementation

This section elaborates on the implementation of Pronto and revisits the most interesting technical challenges we addressed in building it by answering the following questions:

- How to minimize the programming effort of building persistent objects from volatile ones?
- How to implement ASL with minimum overhead on the critical execution path?
- How to identify modified memory pages to efficiently create periodic, asynchronous snapshots?
- How to store asynchronous, consistent snapshots of off-the-shelf volatile data structures with minor changes to the source code?
- How to use semantic logs and snapshots to reconstruct persistent objects after failures?

Pronto comprises a user-level C++ library and a simple source preprocessor. Below we describe how the library manages logs, allocates memory, takes snapshots, and recovers from failures. Then we describe the preprocessor.

5.2.1 Asynchronous Semantic Logging

To reduce the overhead of semantic logging on the critical path, Pronto creates a dedicated background ASL thread for every foreground thread. Foreground threads notify ASL threads upon starting a new failure-atomic operation by calling `op_begin()` and sync up with them to ensure the persistence of semantic logs before committing the log entry.

Pronto uses `pthread_create()` to create an ASL thread for every foreground thread, evenly distributes foreground threads over available physical cores, and co-locates foreground threads with their ASL threads. Sharing physical cores (i.e., running as hyperthreads) enable foreground and ASL threads to share L1 cache-lines and synchronize at low cost. Figure 5.5 shows the assignment of foreground and ASL threads to CPU cores and demonstrates the synchronization points between the two threads.

Pronto's implementation aims to minimize the overhead of ASL on the critical path and trades CPU and recovery time for faster execution of update operations. However, multiple user

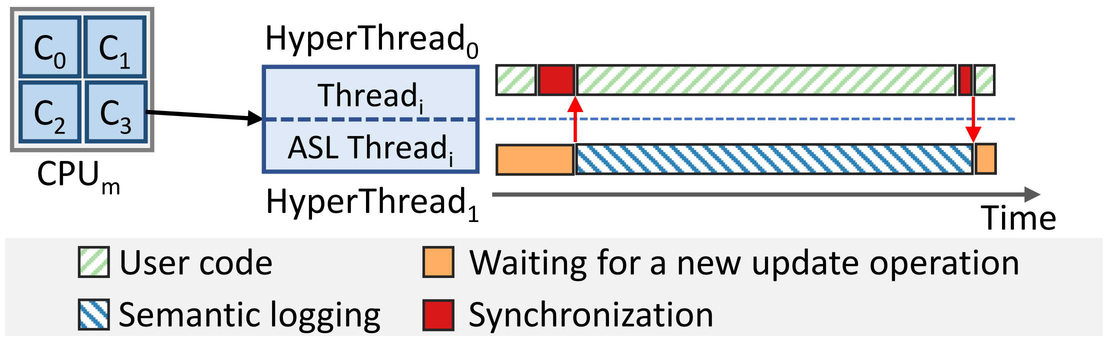


Figure 5.5: Pronto evenly distributes user threads over physical CPU cores and co-locates each one with its ASL thread.

threads can share a single ASL thread for programs that are read-dominated or less sensitive to ASL overhead.

Pronto stores semantic logs in NVM-resident files and creates a separate file for each persistent object. These files comprise a header and a body. The header includes the commit number of the last committed semantic log and relative pointers to the head and tail of the file's body. Having a separate file for each object reduces the contention on the log's header. The body stores semantic logs in a circular buffer.

Semantic log entries contain a pointer to the method they must replay during recovery, as well as a shallow copy of its input arguments. Making a copy is necessary. Otherwise, the application might change a value after the log entry is created, leading to a different result during recovery.

Pronto uses DAX `mmap()` to directly map the file to the program's virtual address space, bypass the storage stack, and access the NVM pages via load/store [61]. ASL threads use non-temporal store instructions followed by memory-barriers to avoid cache pollution while appending semantic logs to the mapped pages, which also improves the performance of creating large semantic logs. Support for DAX `mmap()` is currently available through ext-4, XFS, and NOVA [87, 98].

5.2.2 Memory Allocator

Pronto uses a custom memory allocator for the volatile online image of persistent objects to facilitate creating asynchronous snapshots. The allocator serves allocations from a contiguous volatile memory pool, which could reside on NVM if the DRAM capacity is not sufficient, and maintains a bitmap for the pool to differentiate between used and unused regions. The bitmap granularity is 4 KB.

Pronto serves allocations by regions from an extensible volatile memory pool, which can expand by mapping huge-pages into the program's address space. Pronto uses huge-pages to reduce the number of page-table entries and thus the overhead of creating asynchronous snapshots. The allocator always maps the volatile memory pool at the same virtual address to keep pointers valid throughout restarts and allow recovering objects from snapshots. Pronto maintains per-object allocators that serve allocation and free operations through per-core free-lists to reduce contention, allocation latency and synchronization overhead. Free-lists sort memory regions based on their size and assign them into buckets to reduce lookup time. Each bucket holds a pointer to a doubly-linked list of unused memory regions [8, 28].

5.2.3 Periodic Snapshots

To create a persistent snapshot, Pronto must freeze the execution at a point of time where all persistent objects are in a consistent state (i.e., before or after running an update operation), and then copy the entire online image to NVM. The process of creating snapshots comprises a synchronous and an asynchronous phase.

During the synchronous phase, Pronto freezes persistent objects in a consistent state by blocking new update operations and awaiting completion of those that are yet to be committed. It then streams the state of allocation tables, including the bitmap and free-lists, to NVM and simultaneously marks the allocated volatile pages as read-only.

Next, Pronto unblocks new update operations and starts the asynchronous phase, where it saves the read-only volatile pages to NVM. Pronto uses multiple threads to expedite the copying. The threads examine the allocated 2 MB volatile pages, identify its used 4 KB regions using the bitmap, stream the used regions to NVM, and make each page writable as soon as the NVM copy is durable. An update operation that attempts to write to a read-only volatile page will trigger a page-fault handler, which takes over copying the target page to NVM before marking it writable and returning to the operation that caused the page-fault.

Pronto creates full snapshots for the sake of simplicity. To support incremental snapshots, it can keep volatile pages read-only until modified by an update operation, and only include writable (i.e., modified) pages in new snapshots.

For every persistent object, Pronto also records the identifier of its last committed operation and the tail offset of its semantic log at the time of creating the snapshot. It then recycles any log entry that precedes this tail offset for creating new semantic logs.

5.2.4 Recovery Management

After a crash, Pronto uses a combination of ASL and durable snapshots to restore persistent objects to their state before the failure. It uses the most recent snapshot to restore the latest durable state of its memory pool.

Next, it replays semantic logs against their corresponding persistent objects in commit order. For every persistent object, Pronto only replays semantic log entries recorded after the latest snapshot. Once it replays all log entries, it passes control to user code.

Pronto uses multiple threads to recover persistent objects and assigns a subset of the persistent objects to each recovery thread. Pronto uses a valid linearization order, which is dictated by the commit order of update operations, to replay the semantic logs. Since the original execution of the program is deadlock-free and Pronto replays update operations in a valid linearization order, Pronto's recovery is deadlock-free.

5.2.5 Preprocessor

Pronto's preprocessor reduces the programming effort of using Pronto by automatically generating the code for translating method calls into matching semantic logs during execution and decoding semantic logs to matching method calls during recovery.

For every public method that updates the data structure, the preprocessor passes a pointer to the method as an extra argument to `op_begin()`. It then extends these data structures with a new function that creates semantic logs. These functions, which ASL uses at runtime, store all the input arguments provided to `op_begin()` as well as the pointer to the caller public method in a semantic log entry.

The preprocessor creates a member function for each persistent data structure to enable replaying semantic logs during recovery. This function translates semantic log entries of its data structure to the corresponding public method calls.

The preprocessor also overloads the `new` operator of persistent data structures (i.e., every class that extends `PersistentObject`) to allocate all memory the data structure uses with Pronto's allocator.

5.3 Evaluation

In this section, we evaluate Pronto's performance to provide answers to the following questions:

- What is the performance overhead of using Pronto to add persistence to volatile data structures?
- Can programmers use Pronto to build persistent data structures that outperform highly-optimized NVM data structures?

- What is the performance benefit of using Pronto as the failure-atomicity mechanism for existing applications?
- How much is the speedup of replacing existing NVM libraries with Pronto for persistent data structures?
- When does ASL perform best in hiding the persistence cost?
- What is the cost of creating asynchronous snapshots for data structures with either sequential or random memory access patterns?
- How does the size of data structures, the frequency of snapshots, and the number of threads impact the recovery time?

5.3.1 Testbed Setup

The evaluation platform has two Intel Cascade Lake-SP (engineering sample) processors with 12 physical cores and hyper-threading enabled that run at 2.2 GHz. The platform has 192 GB of DRAM and 1.5 TB (6 $\times$256 GB) of NVM (Intel Optane DC 2666 MHz QS [48, 51]) on each socket. All benchmarks run on one processor and do not go over NUMA to access memory or NVM. We use ext4 to provide direct-access (DAX) to NVM pages [61].

5.3.2 Persistence for Volatile Data Structures

We measure the overhead of using Pronto to add persistence to both sequential (single-threaded) and concurrent (thread-safe) volatile data structures.

Overhead for Sequential Data Structures

Our first experiment uses four containers from the GNU C++ Standard Template Library (STL) to evaluate the overhead of integrating Pronto with volatile data structures. These containers

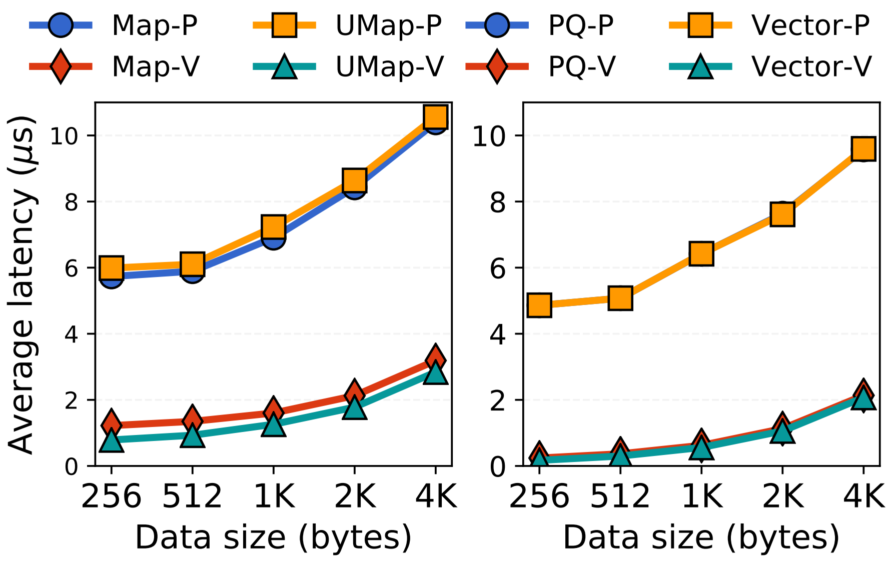


Figure 5.6: Measuring the overhead of using Pronto to add failure-atomicity to the volatile benchmarks. The horizontal axis is the data size of insert operations (excluding the key for Map and Unordered Map benchmarks) in bytes and the vertical axis is the average latency in microseconds. V and P stand for Volatile and Persistent, respectively. UMap and PQ represent the Unordered Map and Priority Queue data structures, respectively.

are:

- *map*: a sorted map that stores key-value pairs in a red-black tree.
- *unordered_map*: an unordered hash-table that stores key-value pairs.
- *vector*: a resizable array data structure.
- *priority_queue*: an adapter for the vector container that creates a max-heap from the inserted elements.

Since STL containers provide deterministic update operations and support using user-defined allocators, we create persistent versions of each container by creating a wrapper class that extends Pronto and wraps calls to the container's public methods, similar to the wrapper for STL's vector in Figure 5.3. To measure the performance of vector and priority_queue, we insert 5 million elements to both versions of each container. We use traces from YCSB [21] to evaluate

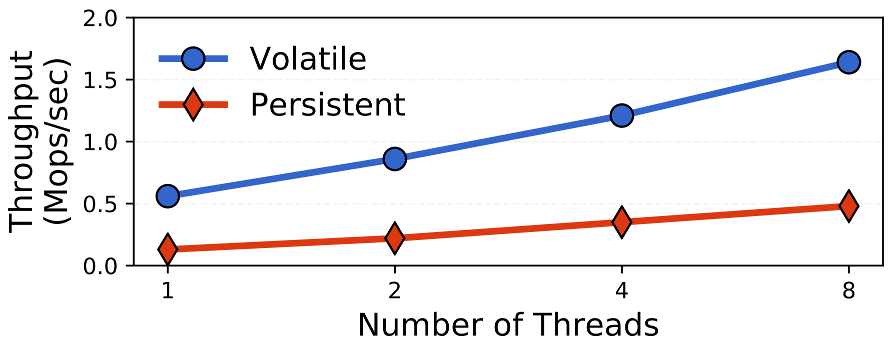


Figure 5.7: Measuring the throughput of the volatile and persistent (Pronto) versions for the concurrent hash-map. Numbers show throughput in millions of 1 KB inserts per second.

map and unordered_map containers. The traces comprise 5 million insert operations with 32-byte keys.

We measure the average latency of both volatile and persistent versions of the benchmarks to quantify the performance overhead of Pronto. Figure 5.6 shows how the average latency for the benchmarks change as we increase the size of data inserted into the STL containers. We create a snapshot for persistent benchmarks at least once every 15 seconds.

For small operations, such as inserting small values into the vector, Pronto imposes more overhead (up to 28×) as the synchronization between the user and the ASL thread is relatively more expensive, and the latency of the operation is significantly smaller than persisting the semantic log. The synchronization overhead is minimal for programs with more complex logic like the priority queue and the map. Moreover, ASL threads use non-temporal stores followed by memory fences to create semantic logs (i.e., copying pointers to operations and their input data to NVM), which perform poorly for small writes and increase the relative overhead of ASL for small operations.

Therefore, the overhead of Pronto is significant for small operations (e.g., 28× for inserting 256-byte values into STL's vector) and lowest for programs with compute-intensive operations and large memory footprints (e.g., 3.2× for adding key-value pairs with 4 KB values to STL's Map).

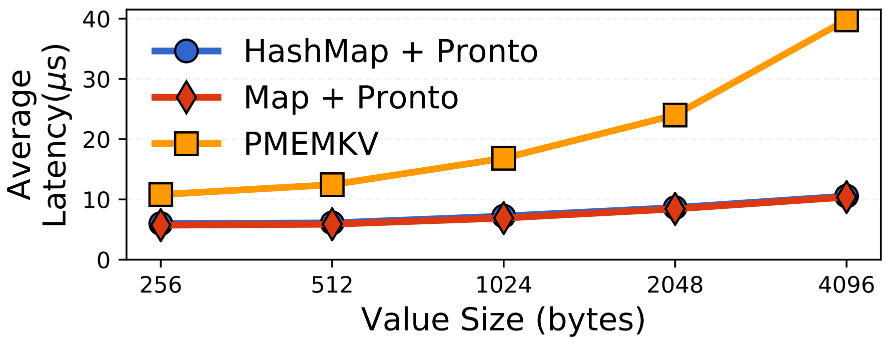


Figure 5.8: Comparing the performance of PMEMKV against the persistent versions of STL's map (Map + Pronto) and unordered_map (HashMap + Pronto) containers.

Concurrent Data Structures

Our next experiment uses the persistent hash-map implementation from Figure 5.4, which adds locking to 32 instances of STL's unordered_map container to support concurrent operations, and compare its throughput against the volatile version of the hash-map to measure Pronto's scalability. We use jemalloc [28] as the allocator for the volatile hash-map since thread-safe malloc uses an internal lock and serializes concurrent accesses. For the persistent hash-map, we create a snapshot at least once every 10 seconds. Figure 5.7 shows the average throughput for inserting 5 million key-value pairs with 1 KB values to the hash-map implementations – as we increase the number of threads (from 1 to 8), both volatile and Pronto versions of the concurrent hash-map show similar scalability.

5.3.3 NVM-Optimized Data Structures

Our next experiment compares the performance of Pronto against NVM-optimized data structures. We use the YCSB traces from Section 5.3.2 to compare the performance of the failure-atomic versions of STL's map and unordered_map containers against PMEMKV [45], which is an NVM-optimized key-value store. We configure PMEMKV v0.3x to use its *kvtree2* storage engine, which adopts undo-logging to implement failure-atomic updates. The persistent

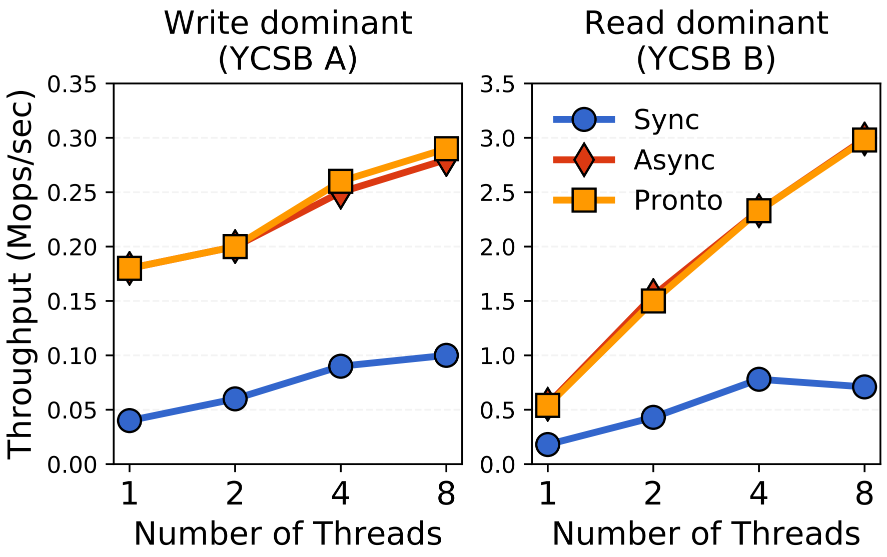


Figure 5.9: Comparing the performance of NVM-optimized version of RocksDB (i.e., Pronto) against its original version with synchronous and asynchronous writes using read-dominant and write-dominant workloads from YCSB.

map and unordered_map containers outperform PMEMKV and provide up to 3.83× and 3.77× lower latency, respectively. Figure 5.8 summarizes the results and reports the average latency of inserting key-value pairs in microseconds.

5.3.4 Optimizing Persistent Data Structures

To demonstrate the performance benefit of using Pronto to optimize existing persistent data structures, we modify RocksDB 5.17 [29], a persistent key-value store library, and replace its default failure-atomicity mechanism (redo-logging) with ASL. Using write-dominant (YCSB A with 50% reads and 50% writes) and read-dominant (YCSB B with 95% reads and 5% writes) traces from YCSB, we compare the performance of the modified version of RocksDB against its original version with *synchronous* and *asynchronous* writes. A synchronous-write does not return unless its redo-log is durable, while an asynchronous-write immediately returns once its redo-log reaches the filesystem's page-cache. As a consequence, a failure may cause the last few asynchronous writes to be lost.

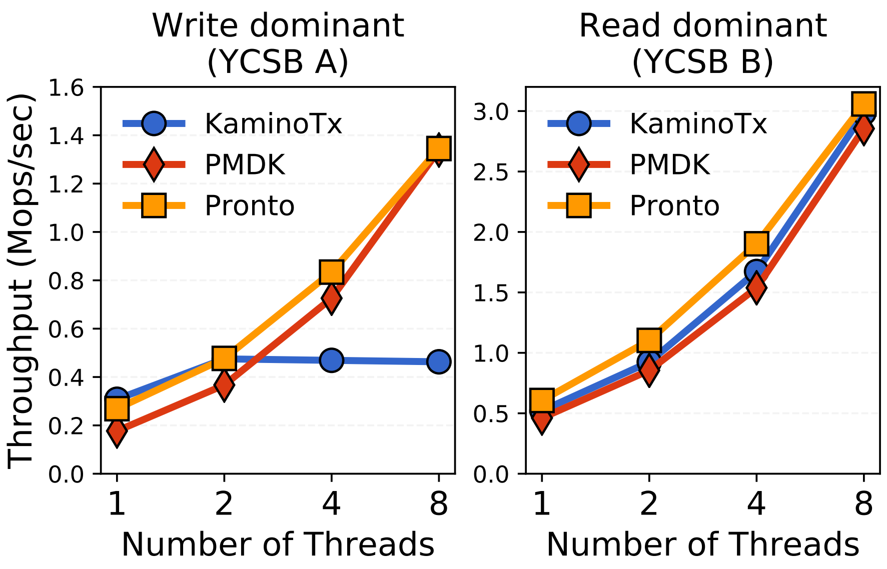


Figure 5.10: Comparing the performance of Pronto against PMDK [44], and KaminoTx using the B+Tree benchmark from KaminoTx [69]. We report the throughput for (read and write) operations with 1 KB values.

We warm-up the key-value stores by inserting 5 million key-value pairs (i.e., YCSB load phase) and then perform 5 million put/get operations based on the workload characteristics (YCSB A and YCSB B). Figure 5.9 shows that the Pronto version outperforms RocksDB with synchronous writes with a wide margin and matches the performance of asynchronous writes for both read-dominant and write-dominant workloads, despite giving stronger guarantees on failure.

5.3.5 Comparing ASL against Undo-Logging

We use the concurrent, persistent B+Tree implementation from Kamino-Tx [69] to compare the performance of Kamino-Tx and PMDK 1.5 [44], existing NVM libraries that accomplish failure-atomic updates using undo-logging, against Pronto. We create a new version of the B+Tree by removing its failure-atomicity code and wrapping it by a Pronto object, thereby making it failure-atomic through Pronto. For the Pronto version of the B+Tree, we create a snapshot after performing 50% of the insert operations (around once every 5 seconds). The Kamino-Tx and PMDK versions only persist the last level of the B+Tree and reconstruct the internal nodes after

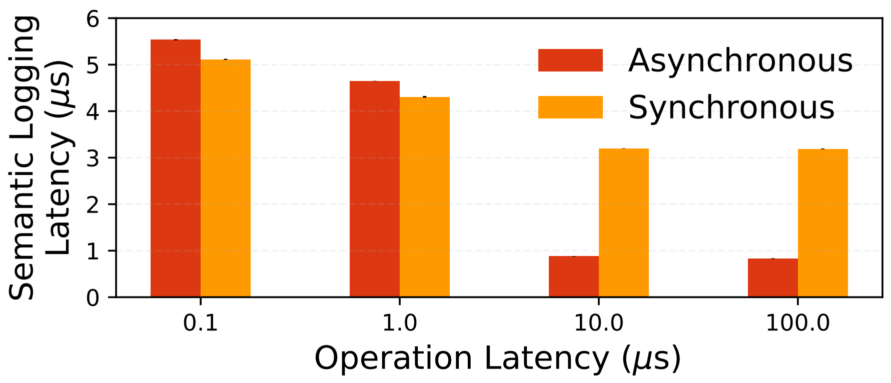


Figure 5.11: Comparing the latency of creating asynchronous to synchronous semantic logs on the critical path. The latency of volatile operations varies from 100 ns to 100 μs, and the size of semantic log entries is 1 KB.

restarts.

Figure 5.10 shows the average throughput of running the YCSB workloads from Section 5.3.4 against the Kamino-Tx, PMDK, and Pronto versions of the B+Tree. In comparison to PMDK and Kamino-Tx, Pronto provides higher performance for the write-dominant workload (YCSB A). Kamino-Tx does not scale when running YCSB A as it uses a single persister thread. Pronto offers slightly higher throughput for the read-dominant workload (YCSB B).

5.3.6 Sensitivity Analysis

We use a microbenchmark to measure the sensitivity of ASL to the latency of the volatile operations. We vary the operation latency from 100 ns to 100 μs and report the overhead of creating 1 KB asynchronous semantic logs on the critical path.

Figure 5.11 shows the results and compares the cost of ASL to synchronous semantic logging, where Pronto creates the 1 KB semantic logs on the critical path and before performing the volatile operations. We report average latencies of 5 million operations across five runs, and show the standard deviation atop each bar (the small, horizontal bars in black).

The experiments show that for sub-microsecond operations, ASL falls short in hiding the persistence overhead as the operation latency is a fraction of the cost of ASL. For other operations,

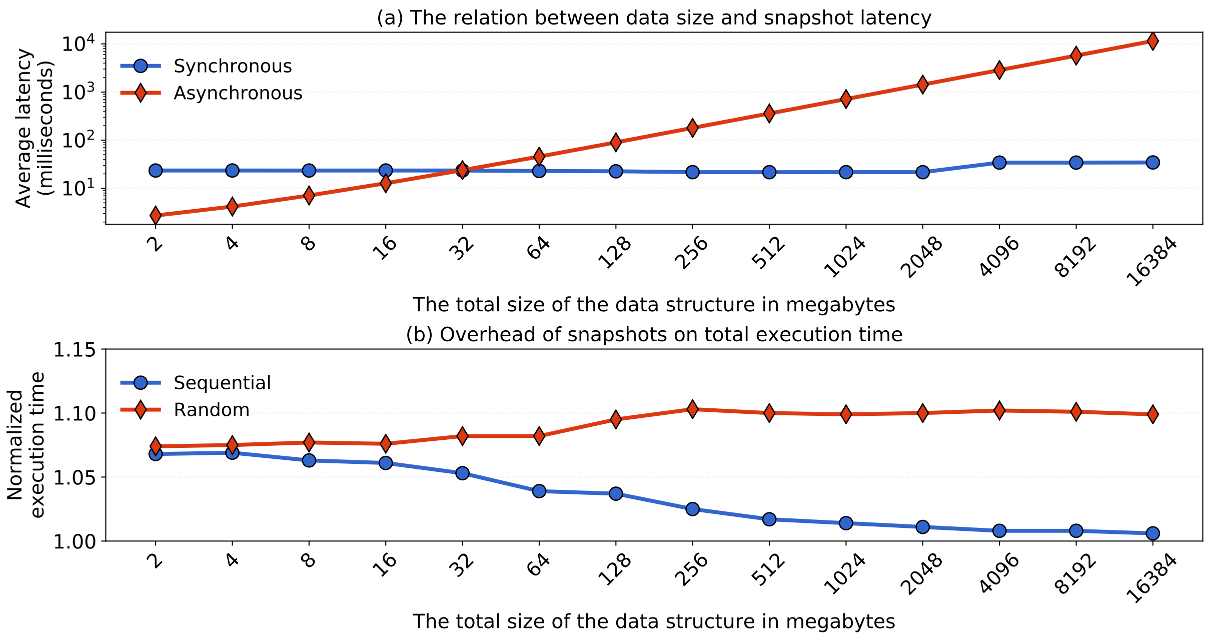


Figure 5.12: Measuring the impact of data size (i.e., total memory allocated by persistent objects) on the overhead of Pronto's periodic snapshots.

ASL moves the entire cost of creating semantic logs to the background and only exposes a small fraction of semantic logging (i.e., committing entries and transferring the operation arguments to the ASL thread) to the critical path.

Note that the cost of persisting semantic logs and committing them decreases as we increase the latency of the volatile operations (i.e., the gap between consecutive writes to the same NVM address). This behavior is due to how Intel Optane DC persistent memory handles back-to-back writes to the same address [51].

5.3.7 Overhead of Snapshots

Snapshot performance is critical for Pronto because it dictates the frequency at which programmers can create snapshots, and thus the trade-off between execution and recovery time. Here we use two micro-benchmarks to quantify the impact of Pronto's snapshot mechanism on the average latency and the total execution time of programs.

The first benchmark studies how the latency of the synchronous and asynchronous steps of creating snapshots change in response to increasing the workload size. Figure 5.12 (a) presents the outcome of this benchmark that varies the workload size (i.e., size of the persistent objects) from 2 MB to 16 GB and measures the latency of both synchronous and asynchronous paths of creating snapshots. The latency of the asynchronous path grows linearly with the workload size, as the size of memory regions that Pronto must persist on NVM increases. However, the latency of the synchronous path only changes from 22 to 34 milliseconds. Thus, Pronto only stalls those update operations that run during the first few milliseconds of creating a new snapshot.

The other benchmark evaluates the impact of snapshots on the total execution time of programs that perform sequential or random 64-bit memory accesses (50% read and 50% write). We vary the workload size and run the benchmark with and without creating a snapshot to calculate normalized execution times. We vary the frequency of creating snapshots between 2 ms and 16 seconds based on the size of the data structure. Figure 5.12 (b) shows the normalized execution time for this benchmark. As the workload size increases, the impact of creating snapshots on the execution time converges to a constant: for programs with random memory access, the constant overhead is about 10%, while programs with sequential memory access only suffer from a 0.8% increase of the execution time. The overhead of Pronto's snapshots is higher on the random-access benchmark because randomly accessing memory while creating an asynchronous snapshot escalates the chance of writing to read-only memory pages, which increases synchronous writes to NVM as well as the impact of Pronto's snapshots on the total execution time.

5.3.8 Recovery Time

We use a new benchmark, which uses Pronto to implement failure-atomic quick-sort, to measure the impact of data-structure size (i.e., size of the online image), number of threads, and snapshot frequency on the recovery time. The benchmark uses quick-sort to sort a large string array, comprising 1 KB strings. We vary the number of elements in the array from 2^{20} (1 GB) to 2^{25} (32 GB), the number of sort threads from 1 to 8, and the snapshot frequency from 2 to 32 seconds. Pronto uses 16 threads to load the snapshot and a single thread to replay semantic logs during recovery.

These experiments show that the primary determinant of recovery time for the failure-atomic quick-sort is the object size, as the snapshot frequency and the number of sort threads has no significant impact on the recovery time. Pronto recovers the 1 GB and 32 GB objects in less than 400 milliseconds and 7 seconds, respectively.

5.4 Related Work

A large body of research with a focus on NVM implications on computer architecture [102, 81], system software [98, 101, 54], and programming support [18, 95] exists that address different challenges of integrating NVMs with existing computer hardware and software. This work, in particular, focuses on reducing the overhead of adding failure-atomicity to volatile data structures in systems equipped with both volatile and non-volatile memories.

Researchers have built several persistent object libraries for NVMs. NV-Heaps [18], Mnemosyne [95], and PMDK [44] provide libraries that allow programs directly and transactionally access NVM. NVM Direct [12] achieves similar goals and adds compiler support. In contrast to Pronto, these systems require disruptive changes to existing programs and impose the overhead of transactional persistence on the critical path of execution.

Kamino-Tx removes the overhead of logging from the critical path and provides atomic in-place updates by maintaining two copies of persistent data [69]. It provides the same set of programming interfaces as PMDK and supports building highly available and reliable persistent data structures via replication. Compared to Pronto, it demands significant changes to existing programs; it also requires persisting transaction and allocation metadata in the critical path.

Atlas [14] automates enforcing failure-atomicity so long as persistent data is only modified inside critical sections, which are surrounded by acquisition and release of locks. NVthreads [39] provides similar failure-atomicity guarantees by using the page table protection bits to automatically track data modifications at the granularity of virtual memory pages and implement copy-on-write. JUSTDO [49] extends on the idea of failure-atomic critical sections and utilizes persistent CPU caches to reduce the memory footprint of logs. In contrast, Pronto provides failure-atomic updates to data structures at the granularity of method calls, uses its allocator to track modified regions that it must persist on NVM, and moves logging off the critical path without requiring hardware support.

Other work has focused on automatically creating persistent versions of volatile data structures. In [50], the authors explore a transform that takes a nonblocking, volatile data structure and creates a persistent version by transforming memory fences into cache-line flushes into NVM. In contrast to this work, Pronto supports blocking data structures and also avoids extraneous cache-line flushes by moving most of the persistence instructions off the critical path.

Periodic checkpoints [2] and persistent virtual memory (pVM [52]) are other means of providing failure-atomicity to programs. However, they both require rigorous changes to the source code and enforce persistence synchronously.

5.5 Summary

We have described Pronto, a system that adds persistence to both sequential and concurrent volatile data structures and reduces the overhead of durability on the critical path of execution through asynchronous semantic logging. Pronto shrinks the performance gap between volatile and persistent data structures by trading recovery time for faster execution. It allows programmers to add failure-atomicity to existing code (e.g., GNU C++ STL containers) without requiring disruptive changes, while the resulting persistent containers provide comparable performance to the volatile versions. Furthermore, our persistent version of the STL's map container outperforms PMEMKV, a persistent key-value store highly optimized for NVM, by up to 3.8×.

Acknowledgments

This chapter contains material from "Pronto: Easy and Fast Persistence for Volatile Data Structures", by Amirsaman Memaripour, Joseph Izraelevitz, and Steven Swanson which is submitted to the 25th Architectural Support for Programming Languages and Operating Systems (ASPLOS 2020). The dissertation author is the primary investigator and first author of this paper.

The author would like to thank Intel Corporation for providing access to the evaluation testbed as well as the members of the NVSL research group for their insightful comments.

Chapter 6

Conclusion

Non-Volatile Memory (NVM) technologies (e.g., 3D XPoint and battery-backed DRAM) expose persistent storage via a byte-addressable load/store interface. However, because caches do not retain their data after a power outage, programmers must be careful to ensure that the state of NVM is useful after a crash. Failure-atomicity libraries provide the means to apply sets of writes to persistent state atomically and avoid inconsistency in the wake of a failure. Unfortunately, these libraries require extensive and error-prone annotations in code. Generally speaking, the programmer needs to annotate all persistent memory accesses, and the annotations vary between libraries. Prior attempts to remove these annotations or add compiler support has incurred unacceptable overheads to runtime performance.

Throughout this dissertation, we have presented a collection of libraries and systems that provide easy to use and fast programming support for persistent memory. This collection comprises a failure-atomicity library that facilitates changing existing software to use NVM, a series of NVM-specific compilation and optimization passes that minimize the programming effort of using failure-atomicity libraries, and a new NVM library that simplifies adding persistence to volatile data structures.

In Chapter 3, we introduced Breeze, an NVM toolchain that provides direct access to persistent memory without requiring disruptive changes to legacy software. Breeze guarantees data consistency and validity of persistent pointers regardless of failures. Porting Memcached and MongoDB to use Breeze only requires changes to 5% of the source code compared to 7-14% for PMDK and NVM-Direct. Breeze also provides equal or superior performance compared to PMDK and NVM-Direct, outperforming them by up to 10×.

In Chapter 4, we presented NVHooks, a compiler that automatically instruments accesses to persistent memory with callbacks to failure-atomicity libraries. NVHooks reduces the programming effort of adopting failure-atomicity libraries and facilitates retargeting programs to a new library. It also offers a series of NVM-specific optimization passes to reduce the cost of these annotations. Our evaluation shows that NVHooks not only minimizes the programming effort of constructing persistent programs, but it also matches the performance of hand-annotated code.

Finally, we described Pronto in Chapter 5. Pronto uses a combination of asynchronous semantic logging (ASL) and persistent, asynchronous snapshots to reduce the programming effort required to make volatile data structures persistent. ASL is generic enough to allow programmers to add persistence to the existing volatile data structure, such as C++ Standard Template Library containers, with very little programming effort. Moreover, ASL mitigates the overhead of durability code (e.g., logging) on the critical path. In contrast to highly-optimized NVM data structures written with other libraries, Pronto data structures are easier to build and offer equal or superior performance.

Acknowledgments

This chapter contains material from “Breeze: User-Level Access to Non-Volatile Main Memories for Legacy Software”, by Amirsaman Memaripour and Steven Swanson, which appeared in the 36th International Conference on Computer Design (ICCD 2018). The dissertation author is the primary investigator and first author of this paper.

This chapter contains material from “NVHooks: A Flexible, Optimizing Compiler for Non-Volatile Memory Programming”, by Amirsaman Memaripour, Yi Xu, Steven Swanson, and Joseph Izraelevitz which is submitted to the 25th Architectural Support for Programming Languages and Operating Systems (ASPLOS 2020). The dissertation author is the primary investigator and first author of this paper.

This chapter contains material from “Pronto: Easy and Fast Persistence for Volatile Data Structures”, by Amirsaman Memaripour, Joseph Izraelevitz, and Steven Swanson which is submitted to the 25th Architectural Support for Programming Languages and Operating Systems (ASPLOS 2020). The dissertation author is the primary investigator and first author of this paper.

Bibliography

[1] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. Compilers, Principles, Techniques. *Addison wesley*, 7(8):9, 1986.

[2] M. Alshboul, J. Tuck, and Y. Solihin. Lazy Persistency: A High-Performing and Write-Efficient Software Persistency Technique. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 439–451, June 2018.

[3] Andy Rudoff. Deprecating the PCOMMIT Instruction, 2016. Available at `https://software.intel.com/en-us/blogs/2016/09/12/deprecate-pcommit-instruction`.

[4] Andy Rudoff. Persistent Memory Programming: The Current State and Future Direction, 2018. Available at `https://www.snia.org/sites/default/files/PM-Summit/2018/presentations/03_PMSummit_18_Rudoff_Final_Post.pdf`.

[5] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler Transformations for High-performance Computing. *ACM Comput. Surv.*, 26(4):345–420, December 1994.

[6] Anirudh Badam. How Persistent Memory will Change Software Systems. *Computer*, 46(8):45–51, August 2013.

[7] Mahesh Balakrishnan, Dahlia Malkhi, Ted Wobber, Ming Wu, Vijayan Prabhakaran, Michael Wei, John D. Davis, Sriram Rao, Tao Zou, and Aviad Zuck. Tango: Distributed Data Structures over a Shared Log. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13', pages 325–340, New York, NY, USA, 2013. ACM.

[8] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. Hoard: A Scalable Memory Allocator for Multithreaded Applications. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IX, pages 117–128, New York, NY, USA, 2000. ACM.

[9] Kumud Bhandari, Dhruva R Chakrabarti, and Hans-J Boehm. Implications of CPU Caching on Byte-Addressable Non-Volatile Memory Programming. *Hewlett-Packard, Tech. Rep. HPL-2012-236*, 2012.

[10] Carl Boettiger. An Introduction to Docker for Reproducible Research. *SIGOPS Oper. Syst. Rev.*, 49(1):71–79, January 2015.

[11] Y. Breitbart, D. Georgakopoulos, M. Rusinkiewicz, and A. Silberschatz. On Rigorous Transaction Scheduling. *IEEE Transactions on Software Engineering*, 17(9):954–960, Sept 1991.

[12] Bill Bridge. NVM Support for C Applications, 2015. Available at `http://www.snia.org/sites/default/files/BillBridgeNVMSummit2015Slides.pdf`.

[13] Chad Thibodeau, Arthur Sainio, Mark Carlson and Alex McDonald. Containers and Persistent Memory, 2017. Available at `https://www.snia.org/sites/default/files/CSI/Containers-and-Persistent-Memory-FInal.pdf`.

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

[15] Shimin Chen and Qin Jin. Persistent B+-trees in Non-volatile Main Memory. *Proc. VLDB Endow.*, 8(7):786–797, February 2015.

[16] Vijay Chidambaram, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Optimistic Crash Consistency. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 228–243, New York, NY, USA, 2013. ACM.

[17] Chih Chieh Chou, Jaemin Jung, AL Narasimha Reddy, Paul V Gratz, and Doug Voigt. vNVML: An Efficient User Space Library for Virtualizing and Sharing Non-volatile Memories. 2019.

[18] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. NV-Heaps: Making Persistent Objects Fast and Safe with Next-generation, Non-volatile Memories. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '11, pages 105–118, New York, NY, USA, 2011. ACM.

[19] John Cocke. Global Common Subexpression Elimination. *ACM Sigplan Notices*, 5(7):20–24, 1970.

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

[21] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM.

[22] Intel Corporation. Intel/Micron 3D-Xpoint Non-Volatile Main Memory, 2015. Available at `https://www.intel.com/content/www/us/en/architecture-and-technology/intel-micron-3d-xpoint-webcast.html`.

[23] Andreia Correia, Pascal Felber, and Pedro Ramalhete. Romulus: Efficient Algorithms for Persistent Transactional Memory. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, SPAA '18, pages 271–282, New York, NY, USA, 2018. ACM.

[24] Joel E. Denny, Seyong Lee, and Jeffrey S. Vetter. NVL-C: Static Analysis Techniques for Efficient, Correct Programming of Non-Volatile Main Memory Systems. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '16, pages 125–136, New York, NY, USA, 2016. ACM.

[25] Jack J Dongarra and A_R Hinds. Unrolling Loops in Fortran. *Software: Practice and Experience*, 9(3):219–226, 1979.

[26] Kshitij Doshi and Peter Varman. WrAP: Managing Byte-Addressable Persistent Memory, 2012. Available at `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.303.5364&rep=rep1&type=pdf`.

[27] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. System Software for Persistent Memory. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 15:1–15:15, New York, NY, USA, 2014. ACM.

[28] J Evans. Scalable Memory Allocation using jemalloc, 2011. *URL https://www.facebook. com/notes/facebook-engineering/scalable-memory-allocation-using-jemalloc/480222803919*, 2016.

[29] Facebook. RocksDB, 2017. `http://rocksdb.org`.

[30] Bin Fan, David G. Andersen, and Michael Kaminsky. MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 371–384, Lombard, IL, 2013. USENIX.

[31] Brad Fitzpatrick. Distributed Caching with Memcached. *Linux Journal*, 2004(124):5–, August 2004.

[32] The LLVM Foundation. Clang: A C Language Family Frontend for LLVM, 2019. Available at `https://clang.llvm.org/`.

[33] The LLVM Foundation. LLVM Language Reference Manual, 2019. Available at `https://llvm.org/docs/LangRef.html`.

[34] The LLVM Foundation. LLVM's Analysis and Transform Passes, 2019. Available at `https://llvm.org/docs/Passes.html#memcpyopt-memcpy-optimization`.

[35] The LLVM Foundation. The LLVM Compiler Infrastructure, 2019. Available at `https://llvm.org/`.

[36] Jim Gray. The Transaction Concept: Virtues and Limitations (Invited Paper). In *Proceedings of the Seventh International Conference on Very Large Data Bases - Volume 7*, VLDB '81', pages 144–154. VLDB Endowment, 1981.

[37] Jinyu Gu, Qianqian Yu, Xiayang Wang, Zhaoguo Wang, Binyu Zang, Haibing Guan, and Haibo Chen. Pisces: A Scalable and Efficient Persistent Transactional Memory. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 913–928, Renton, WA, July 2019. USENIX Association.

[38] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990.

[39] Terry Ching-Hsiang Hsu, Helge Brügner, Indrajit Roy, Kimberly Keeton, and Patrick Eugster. NVthreads: Practical Persistence for Multi-threaded Applications. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 468–482, New York, NY, USA, 2017. ACM.

[40] Jian Huang, Karsten Schwan, and Moinuddin K. Qureshi. NVRAM-aware Logging in Transaction Systems. *Proc. VLDB Endow.*, 8(4):389–400, December 2014.

[41] Intel. An Introduction to pmemcheck, 2015. Available at `http://pmem.io/2015/07/17/pmemcheck-basic.html`.

[42] Intel Corporation. Examples for libpmemobj: A Transactional HashMap. Available at `https://github.com/pmem/pmdk/tree/master/src/examples/libpmemobj/hashmap`.

[43] Intel Corporation. Non-Volatile Memory. Available at `http://www.intel.com/content/www/us/en/architecture-and-technology/non-volatile-memory.html`.

[44] Intel Corporation. Persistent Memory Development Kit. Available at `http://pmem.io/pmdk/`.

[45] Intel Corporation. PMemKV. Available at `https://github.com/pmem/pmemkv`.

[46] Intel Corporation. Enterprise and Cloud Storage Processor for the Digital Era, 2016. Available at `https://www.intel.sg/content/www/xa/en/storage/enterprise-cloud-storage-processor.html`.

[47] Intel Corporation. Intel Architecture Instruction Set Extensions Programming Reference, 2019. Available at `https://software.intel.com/sites/default/files/managed/0d/53/319433-022.pdf`.

[48] Intel Corporation. Intel Optane DC Persistent Memory, 2019. `https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html`.

[49] Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. Failure-Atomic Persistent Memory Updates via JUSTDO Logging. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16', pages 427–442, New York, NY, USA, 2016. ACM.

[50] Joseph Izraelevitz, Hammurabi Mendes, and Michael L. Scott. Linearizability of Persistent Memory Objects Under a Full-System-Crash Failure Model. In Cyril Gavoille and David Ilcinkas, editors, *Distributed Computing*, pages 313–327, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[51] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module. *CoRR*, abs/1903.05714, 2019.

[52] Sudarsun Kannan, Ada Gavrilovska, and Karsten Schwan. pVM: Persistent Virtual Memory for Efficient Capacity Scaling and Object Storage. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16', pages 13:1–13:16, New York, NY, USA, 2016. ACM.

[53] Kevin Oleary. How to Detect Persistent Memory Programming Errors using Intel Inspector, 2018. Available at `https://software.intel.com/en-us/articles/detect-persistent-memory-programming-errors-with-intel-inspector-persistence-inspector`.

[54] Daehyeok Kim, Amirsaman Memaripour, Anirudh Badam, Yibo Zhu, Hongqiang Harry Liu, Jitu Padhye, Shachar Raindel, Steven Swanson, Vyas Sekar, and Srinivasan Seshan. Hyperloop: Group-based NIC-offloading to Accelerate Replicated Transactions in Multitenant Storage Systems. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, pages 297–312, New York, NY, USA, 2018. ACM.

[55] Wook-Hee Kim, Jinwoong Kim, Woongki Baek, Beomseok Nam, and Youjip Won. NVWAL: Exploiting NVRAM in Write-Ahead Logging. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16', pages 385–398, New York, NY, USA, 2016. ACM.

[56] Kenneth C. Knowlton. A Fast Storage Allocator. *Commun. ACM*, 8(10):623–624, October 1965.

[57] Aasheesh Kolli, Steven Pelley, Ali Saidi, Peter M. Chen, and Thomas F. Wenisch. High-Performance Transactions for Persistent Memories. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 399–411, New York, NY, USA, 2016. ACM.

[58] David J. Kuck. A Survey of Parallel Machine Organization and Programming. *ACM Comput. Surv.*, 9(1):29–59, March 1977.

[59] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.

[60] Xiaozhou Li, David G. Andersen, Michael Kaminsky, and Michael J. Freedman. Algorithmic Improvements for Fast Concurrent Cuckoo Hashing. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14', pages 27:1–27:14, New York, NY, USA, 2014. ACM.

[61] Linux Kernel Organization. Direct Access for Files. Available at `https://www.kernel.org/doc/Documentation/filesystems/dax.txt`.

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

[63] Q. Liu, J. Izraelevitz, S. K. Lee, M. L. Scott, S. H. Noh, and C. Jung. iDO: Compiler-Directed Failure Atomicity for Nonvolatile Memory. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 258–270, Oct 2018.

[64] Ren-Shuo Liu, De-Yu Shen, Chia-Lin Yang, Shun-Chih Yu, and Cheng-Yuan Michael Wang. NVM Duet: Unified Working Memory and Persistent Store Architecture. *SIGPLAN Not.*, 49(4):455–470, February 2014.

[65] Sihang Liu, Yizhou Wei, Jishen Zhao, Aasheesh Kolli, and Samira Khan. PMTest: A Fast and Flexible Testing Framework for Persistent Memory Programs. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 411–425. ACM, 2019.

[66] Chris Lomont. Introduction to x64 Assembly, 2012. Available at `https://software.intel.com/en-us/articles/introduction-to-x64-assembly`.

[67] David E. Lowell and Peter M. Chen. Free transactions with rio vista. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, SOSP '97', pages 92–101, New York, NY, USA, 1997. ACM.

[68] Mark Carlson. Persistent Memory: What Developers Need to Know. Available at `https://www.snia.org/sites/default/files/SDCEMEA/2018/Presentations/Persistent-Memory-for-Developers-SNIA-SDC-EMEA-2018.pdf`.

[69] Amirsaman Memaripour, Anirudh Badam, Amar Phanishayee, Yanqi Zhou, Ramnatthan Alagappan, Karin Strauss, and Steven Swanson. Atomic In-place Updates for Non-volatile Main Memories with Kamino-Tx. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 499–512, New York, NY, USA, 2017. ACM.

[70] Amirsaman Memaripour and Steven Swanson. Breeze: User-Level Access to Non-Volatile Main Memories for Legacy Software. In *Proceedings of the 36th IEEE International Conference on Computer Design*, pages 413–422, 2018.

[71] Dirk Merkel. Docker: Lightweight Linux Containers for Consistent Development and Deployment. *Linux J.*, 2014(239), March 2014.

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

[73] Micron Technology. Breakthrough Non-Volatile Memory Technology. Available at `https://www.micron.com/about/emerging-technologies/3d-xpoint-technology`.

[74] Micron Technology. NVDIMM. Available at `https://www.micron.com/products/dram-modules/nvdimm/`.

[75] Mike Ferron-Jones. A New Breakthrough in Persistent Memory Gets Its First Public Demo. Available at `https://itpeernetwork.intel.com/new-breakthrough-persistent-memory-first-public-demo/`.

[76] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. ARIES: A Transaction Recovery Method Supporting Fine-granularity Locking and Partial Rollbacks Using Write-ahead Logging. *ACM Trans. Database Syst.*, 17(1):94–162, March 1992.

[77] MongoDB, Inc. Introduction to MongoDB. Available at `http://docs.mongodb.org/manual/core/introduction/`.

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

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

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

[81] M. A. Ogleari, E. L. Miller, and J. Zhao. Steal but No Force: Efficient Hardware Undo+Redo Logging for Persistent Memory Systems. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 336–349, Feb 2018.

[82] Rasmus Pagh and Flemming Friche Rodler. Cuckoo Hashing. *J. Algorithms*, 51(2):122–144, May 2004.

[83] Stan Park, Terence Kelly, and Kai Shen. Failure-atomic Msync(): A Simple and Efficient Mechanism for Preserving the Integrity of Durable Data. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 225–238, New York, NY, USA, 2013. ACM.

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

[85] Steven Pelley, Thomas F. Wenisch, Brian T. Gold, and Bill Bridge. Storage Management in the NVRAM Era. *Proc. VLDB Endow.*, 7(2):121–132, October 2013.

[86] pmem.io. Persistent Memory Development Kit, 2017. `http://pmem.io/pmdk`.

[87] pmem.io. Using Persistent Memory Devices with the Linux Device Mapper, 2018. Available at `https://pmem.io/2018/05/15/using_persistent_memory_devices_with_the_linux_device_mapper.html`.

[88] Jinglei Ren, Jishen Zhao, Samira Khan, Jongmoo Choi, Yongwei Wu, and Onur Mutlu. ThyNVM: Enabling Software-Transparent Crash Consistency in Persistent Memory Systems. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, pages 672–685, New York, NY, USA, 2015. ACM.

[89] Andy Rudoff. Persistent Memory Programming. *USENIX; login*, 42, 2017.

[90] Andy Rudoff. Persistent Memory: The Value to HPC and the Challenges. In *Proceedings of the Workshop on Memory Centric Programming for HPC*, MCHPC'17, pages 7–10, New York, NY, USA, 2017. ACM.

[91] M. Satyanarayanan, Henry H. Mashburn, Puneet Kumar, David C. Steere, and James J. Kistler. Lightweight Recoverable Virtual Memory. *ACM Trans. Comput. Syst.*, 12(1):33–57, February 1994.

[92] David Schwalb, Markus Dreseler, Matthias Uflacker, and Hasso Plattner. NVC-Hashmap: A Persistent and Concurrent Hashmap For Non-Volatile Memories. In *Proceedings of the 3rd VLDB Workshop on In-Memory Data Mangement and Analytics*, IMDM '15', pages 4:1–4:8, New York, NY, USA, 2015. ACM.

[93] Jihye Seo, Wook-Hee Kim, Woongki Baek, Beomseok Nam, and Sam H. Noh. Failure-Atomic Slotted Paging for Persistent Memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17', pages 91–104, New York, NY, USA, 2017. ACM.

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

[95] Haris Volos, Andres Jaan Tack, and Michael M. Swift. Mnemosyne: Lightweight Persistent Memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 91–104, New York, NY, USA, 2011. ACM.

[96] Tianzheng Wang and Ryan Johnson. Scalable Logging Through Emerging Non-volatile Memory. *Proceedings of the VLDB Endowment*, 7(10):865–876, June 2014.

[97] Wikipedia. Non-Cryptographic Hash Functions. Available at `https://en.wikipedia.org/wiki/List_of_hash_functions#Non-cryptographic_hash_functions`.

[98] Jian Xu and Steven Swanson. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 323–338, Santa Clara, CA, 2016. USENIX Association.

[99] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. NV-Tree: Reducing Consistency Cost for NVM-based Single Level Systems. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, FAST'15, pages 167–181, Berkeley, CA, USA, 2015. USENIX Association.

[100] Lu Zhang and Steven Swanson. Pangolin: A Fault-Tolerant Persistent Memory Programming Library. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 897–912, Renton, WA, 2019. USENIX Association.

[101] Yiying Zhang, Jian Yang, Amirsaman Memaripour, and Steven Swanson. Mojim: A Reliable and Highly-Available Non-Volatile Memory System. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15', pages 3–18, New York, NY, USA, 2015. ACM.

[102] Jishen Zhao, Sheng Li, Doe Hyun Yoon, Yuan Xie, and Norman P. Jouppi. Kiln: Closing the Performance Gap Between Systems with and Without Persistence Support. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, pages 421–432, New York, NY, USA, 2013. ACM.