UNIVERSITY OF CALIFORNIA SAN DIEGO

Fast, Durable, and Safe Data Management Support for Persistent Memory

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

in

Computer Science

by

Morteza Hoseinzadeh

Committee in charge:

Professor Steven Swanson, Chair
Professor Pamela Coseman
Professor Ranjit Jhala
Professor Geoffrey Voelker
Professor Jishen Zhao

2021

The dissertation of Morteza Hoseinzadeh is approved, and it is acceptable in quality and form for publication on microfilm and electronically.

University of California San Diego

2021

DEDICATION

To my wife Sahar, who has been there with me since the beginning of this journey. And to my beloved son Armin whose birth was the most fabulous graduation present from God and the most influential encouragement to me.

EPIGRAPH

Seek knowledge from the cradle to the grave.

—Prophet Mohammad (p.b.u.h.)

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

First and foremost, I would like to praise and thank God, for the abundance of his grace and blessings throughout my research work to complete my Ph.D. program successfully.

I want to express my deep and sincere gratitude to my research supervisor, Dr. Steven Swanson, for giving me the chance to work with him and supporting me throughout this research with his invaluable guidance and precious time. His dynamism, vision, sincerity, and motive have genuinely inspired me. He has shown me the paths to carry out the study and present the research works sharply. It was a tremendous privilege and honor to work and research by his side. I am incredibly grateful for what he has taught me. I would also like to thank him for his friendship, empathy, and sensibility when I needed him the most.

I am sincerely grateful to my late mother, who endured my absence while suffering from her deadly illness. I did my best to make her proud. I also would like to thank my father for his love, prayers, caring, and sacrifices to educate and prepare me for my future. I am very much thankful to my wife for her love, understanding, and continuing support to complete this research work. Also, I express my thanks to my parents-in-law for their support and valuable prayers. I am forever indebted to my family for their unconditional love and support.

Chapter 1, Chapter 2, Chapter 3, and Chapter 6 of this dissertation contain material from "Corundum: Statically-Enforced Persistent Memory Safety", by Morteza Hoseinzadeh and Steven Swanson, which appeared in the Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 21), 2021. The dissertation author is the primary investigator and first author of this paper.

Chapter 1, Chapter 2, Chapter 5, and Chapter 6 of this dissertation contain material from "Ziggurat: A Tiered File System for Non-Volatile Main Memories and Disks", by Shengan Zheng, Morteza Hoseinzadeh, and Steven Swanson, which appeared in the Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST 19), 2019. The dissertation author is the primary investigator and the second author of this paper.

VITA

2011	Bachelor of Science in Computer Engineering, Shahed University
2013	Master of Science in Computer Engineering, Sharif University of Technology
2017	Internship, Samsung Semiconductor, Inc.
2020	Internship, Oracle
2021	Doctor of Philosophy in Computer Science, University of California San Diego

PUBLICATIONS

Morteza Hoseinzadeh and Steven Swanson, "Corundum: Statically-Enforced Persistent Memory Safety", *the 26th international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021.

Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steven Swanson, "An Empirical Guide to the Behavior and Use of Scalable Persistent Memory", *the 18th USENIX Conference on File and Storage Technologies (FAST)*, 2020.

Shengan Zheng, Morteza Hoseinzadeh, and Steven Swanson, "Ziggurat: a tiered file system for non-volatile main memories and disks", *the 17st USENIX Conference on File and Storage Technologies (FAST)*, 2019.

Morteza Hoseinzadeh, "Flow-based Simulation Methodology", *IEEE Computer Architecture Letters (IEEE CAL)*, 2017.

Zhengyu Yang, Morteza Hoseinzadeh, Allen Andrews, Clay Mayers, David Thomas Evans, Rory Thomas Bolt, Janki Bhimani, Ningfang Mi, and Steven Swanson, "AutoTiering: Automatic Data Placement Manager in Mulit-Tier All-Flash Datacenter", *the 36th International Performance Computing and Communications Conference (IEEE IPCCC)*, 2017.

Zhengyu Yang, Morteza Hoseinzadeh, Ping Wong, John Artoux, Clay Mayers, David Thomas Evans, Rory Thomas Bolt, Janki Bhimani, Ningfang Mi, and Steven Swanson. "H-NVMe: a hybrid framework of NVMe-based storage system in cloud computing environment", *In 2017 IEEE 36th International Performance Computing and Communications Conference (IEEE IPCCC)*, 2017.

Morteza Hoseinzadeh, Mohammad Arjomand, and Hamid Sarbazi-Azad, “SPCM: The Striped Phase Change Memory”, *ACM Transaction on Architecture and Code Optimization (TACO).*, 2015.

Morteza Hoseinzadeh, Mohammad Arjomand, and Hamid Sarbazi-Azad, “Reducing Access Latency of MLC PCMs through Line Striping”, *the 41st International Symposium on Computer Architecture (ISCA)*, 2014.

ABSTRACT OF THE DISSERTATION

Fast, Durable, and Safe Data Management Support for Persistent Memory

by

Morteza Hoseinzadeh

Doctor of Philosophy in Computer Science

University of California San Diego, 2021

Professor Steven Swanson, Chair

Emerging fast, byte-addressable Persistent Memory (PM) considerably increases the storage performance compared to traditional disks and makes it possible to build complex data structures that can survive system failures. However, programming for PM is challenging, not least because it combines well-known programming challenges like locking, memory management, and pointer safety with novel PM-specific bug types. It also requires logging updates to PM to facilitate recovery after a crash. A misstep in any area can corrupt data, leak resources, or prevent successful recovery after a crash. Additionally, its high price limits its capacity to scale as same as traditional block devices.

This dissertation first presents Corundum, a Rust-based library with an idiomatic PM

programming interface that leverages Rust's type system to avoid the most common PM programming bugs statically. Corundum lets programmers develop persistent data structures using familiar Rust constructs and have confidence that they will be free of those bugs. We have implemented Corundum and found its performance to be as good as or better than Intel's widely-used PMDK library, HP's Atlas, Mnemosyne, and go-pmem.

Then, the dissertation presents Carbide, a robust multilingual programming framework that allows developing safe data structures in Corundum and using them in C++. Carbide strictly checks the data structure implementation for PM safety and allows unrestrictedly using them in C++ with an accurate persistent type checking process. Our experimental results show that Carbide not only does not incur a significant slowdown, but it is also faster than purely using Corundum in some of our benchmarks due to the flexibility that it provides.

Finally, it introduces Ziggurat, a tiered file system that combines PM and slow disks to create a high-performance and extensive storage system. Ziggurat steers incoming writes to PM, DRAM, or disk depending on application access patterns, write size, and the likelihood that the application will stall until the write completes. Our experimental results show that with a small amount of PM and a large SSD, Ziggurat achieves up to 38.9× and 46.5× throughput improvement compared with EXT4 and XFS running on an SSD alone, respectively.

Chapter 1

Introduction

Persistent main memory (PM) is the first new memory technology to arrive in the memory hierarchy since the appearance of DRAM in the early 1970s. PM offers numerous potential benefits, including improved memory system capacity, lower latency and higher bandwidth relative to disk-based storage, and a unified programming model for persistent and volatile program states.

PM devices are directly attached to the processors through the memory bus, which allows the processors to access data using load and store instructions. This feature exposes low-cost persistence to the user applications and eliminates the need for DRAM buffering in file systems. Since data is randomly accessible in PM, applications can skip serializing data in store operations and deserializing it in load operations. PM-based file systems also can cut buffering data in DRAM to make it available to the processors. Figure 1.1 depicts the memory technology hierarchy and shows PM sitting in the gray area, which provides the bests of both worlds: byte-addressability and non-volatility. This unique feature of PM draws researchers' attentions to use this opportunity. It proposes several fast PM-based kernel applications such as PM file systems [17, 20, 21, 83, 86], multi-tiered storage systems with PM as the performance tier [95, 53, 1, 23], and many PM programming libraries that directly expose PM to the user-level programs [75, 37, 11, 82, 28, 79].

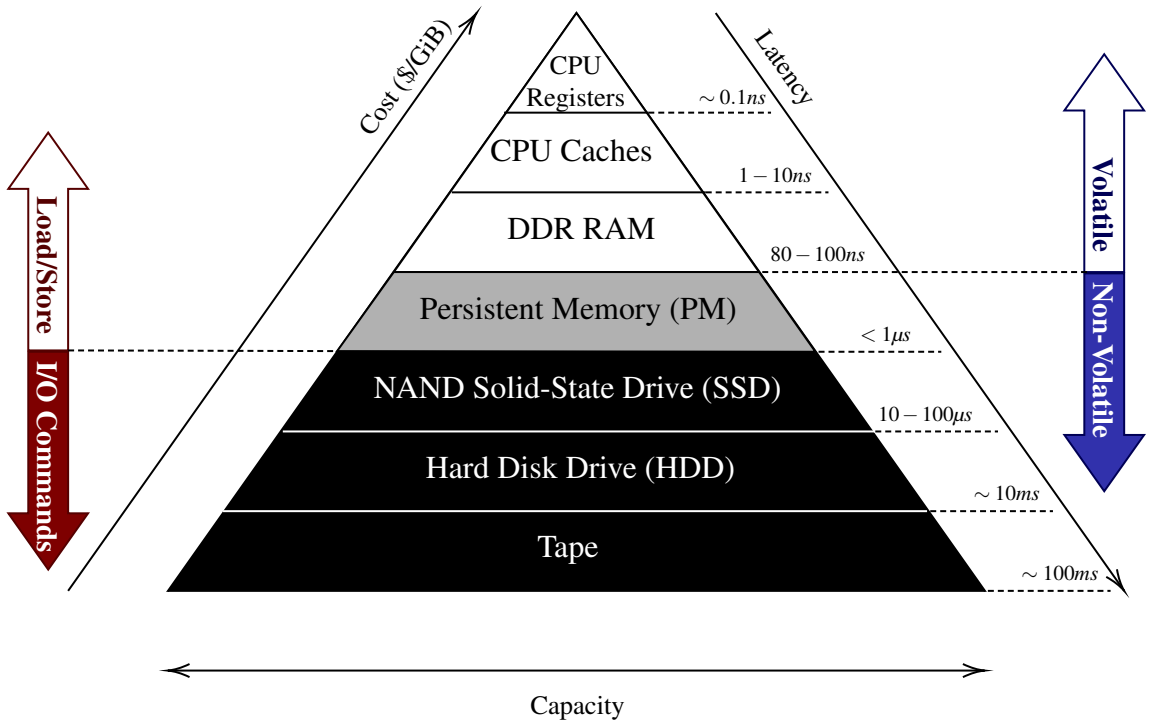


Figure 1.1: The memory hierarchy pyramid compares currently available technologies in terms of cost, performance, and capacity. Persistent memory provides byte-addressability and non-volatility at the same time.

PM high-performance makes it a desirable storage media to be used by a file system. Its direct accessibility eradicates the need for a DRAM page cache. And the file system can expose the PM to the applications without any synchronization mechanism to enable direct access memory (DAX). However, PM comes at a higher price compared with SSDs and HDDs. This fact limits the PM-based file systems' scalability regarding the storage capacity.

Additional to its high costs, PM poses a host of novel challenges. It requires memory controller and ISA support, new operating system facilities, and significant burdens on programmers. The programming challenges it poses are daunting and stem directly from its non-volatility, high-performance, and direct connection to the processor's memory bus. PM's raw performance demands the removal of system software from the common-case access path, its non-volatility requires that (when used as storage) updates must be robust in the face of system failures, and its memory-like interface forces application software to deal directly with issues like fault tolerance

and error recovery rather than relying on system software layers.

Processors use volatile caches to store data and instructions locally. Therefore, updates to a persistent region are not durable until the cache writes back the corresponding cache line to PM. Cache line write-back may happen when the cache line is outdated based on the caching policy or when a cache-flush instruction arrives. The former has a non-deterministic pattern so that we cannot rely on it to persist updates synchronously. The latter one requires the programmer to issue an expensive cache-flush operation [7, 14]. Applications often delay flushes to alleviate the performance overhead. However, holding up flushes to a later time may end up in undesirable situations. For example, a power failure can put the whole PM in an inconsistent state with several wrecked writes. There is a body of research for solving this issue either by optimizing cache-flush operations or reducing their necessity to a minimum possible amount. Presently, persistent CPU caches are yet to be manufactured. At the same time, PM-optimized cache-flush and memory fence instructions are available to the programmers, enabling them to develop crash-consistent persistent data structures [42].

Logging data to make them recoverable is a common approach to save several flushes [57, 44, 33, 38, 78]; when data is durably logged once, there is no need for synchronization in further updates. This mechanism, along with a recovery procedure, can provide crash consistency in PM applications. The programmer, however, needs to log data whenever a data modification is necessary, concisely. This requirement increases the chances of human mistakes while developing a PM application.

Furthermore, programming with PM exacerbates the impact of existing types of bugs besides introducing novel classes of programming errors. Common errors like memory leaks, dangling pointers, concurrency bugs, and data structure corruption have permanent effects (rather than dissipating on restart). New mistakes are also possible: a programmer might forget to log an update to a persistent structure or create a pointer from a persistent data structure to volatile memory. The former error may manifest during recovery while the latter is inherently unsafe

because, after the restart, the pointer to volatile memory is meaningless, and dereferencing it will result in (at best) an exception.

The challenges of programming *correctly* with PM are among the most prominent potential obstacles to the widespread adoption of PM and our ability to exploit its capabilities thoroughly. PM will be hobbled as a storage technology if programmers cannot reliably write and modify code that correctly and safely modifies persistent data structures. Some of the bugs that PM programs suffer from have been the subject of years of research and practical tool building. The solutions and approaches to these problems range from programming disciplines to improved library support to debugging tools to programming language facilities.

This dissertation focuses on providing fast, reliable, and safe approaches for managing data in persistent memory. We aim to make PM available to the user applications with high-speed, durable, and secure data management support. This dissertation first presents our approach of fully benefiting from PM's high bandwidth and low latency to design a fast storage system with a large capacity. Then we describe our PM programming library that strictly applies several PM safety rules to facilitate developing provably correct crash-consistent data structures. Finally, the dissertation introduces our unique framework for multilingual PM programming that dramatically improves flexibility in programming despite applying the same strict safety rules on persistent data type development.

Chapter 3 addresses the problem of human mistakes during the development of a PM program. Given the enhanced importance of memory and concurrency errors in PM programming, adopting the most effective and reliable methods for avoiding them makes sense. We have leveraged Rust's abilities to create Corundum, a library (or "crate" in Rust parlance) that lets programmers build persistent data structures. Corundum provides basic PM programming facilities (e.g., opening and mapping persistent memory pools) and a persistent software transactional memory interface to provide atomic updates to persistent data.

Restricting programmers to make strong safety guarantees limits the programmability,

which might not be necessary sometimes. For example, unprotected updates to reconstructible persistent objects may not satisfy all PM safety invariants, yet it is perfectly safe if enough care is taken. Chapter 4 presents a multilingual programming framework that automatically redeclares a data structure in a flexible programming language, whereas its definition is in a strictly safe language. More specifically, we develop Carbide, which works as a bridge between Corundum and C++. Carbide automatically generates secure APIs for C++ to access the data structures implemented using Corundum in Rust using a set of techniques.

Chapter 5 discusses one of the significant challenges of using PM as a large-scale storage device and our solution to address them. The average price per byte of persistent memory is significantly higher than SSD. Also, SSDs and hard drives scale to much larger capacities than PM. So, file-based workloads that are cost-sensitive or require more extensive storage capacities than PM can provide would benefit from a storage system that can leverage both technologies' strengths: PM for speed and disks for storage capacity. We develop Ziggurat, a multi-tiering file system that manages a hierarchy of heterogeneous storage media and places data in the storage device that matches the data's performance requirements and the application's future access patterns.

Finally, Chapter 6 concludes the dissertation and summarizes our contributions, including a high-performance, high-capacity multi-tiered file system, a unique PM programming library that provides the means to develop provably correct persistent data structures, and a programming framework that automatically ports strictly checked persistent data structures into a flexible programming environment.

Acknowledgments

This chapter contains material from "Ziggurat: A Tiered File System for Non-Volatile Main Memories and Disks", by Shengan Zheng, Morteza Hoseinzadeh, and Steven Swanson,

which appeared in the Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST 19), 2019. The dissertation author is the primary investigator and the second author of this paper.

This chapter contains material from "Corundum: Statically-Enforced Persistent Memory Safety", by Morteza Hoseinzadeh and Steven Swanson, which appeared in the Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 21), 2021. The dissertation author is the primary investigator and first author of this paper.

Chapter 2

Background and Motivation

Emerging non-volatile memory (NVM) technologies (e.g., PMs) and conventional block-based storage devices (e.g., SSDs or HDDs) have various performance characteristics and different interfaces. As the main target of this dissertation, PM directly connects to the processor through the memory interface, while other storage devices are accessible through the I/O bus. On one hand, it gives PM a DRAM-like performance. Many PM projects such as NOVA [87] take this opportunity and improve their performance vastly. On the other hand, persistent memory imposes novel challenges that the programmer needs to manage. Persistent memory programming libraries alleviate many of these challenges by providing a safe interface to access persistent memory. This chapter provides background on persistent memories and disks, PM programming in the user space, and PM programming in the kernel space.

2.1 Storage Technologies

Persistent memory (PM), solid-state drive (SSD), and hard disk drive (HDD) technologies have unique latency, bandwidth, capacity, and characteristics. Table 2.1 shows the performance comparison of different storage devices.

Persistent memory provides byte-addressability, persistence, and direct access via the

Table 2.1: Performance comparison among different storage media. DRAM, PM, and hard disk numbers are estimated based on [3, 47, 93]. SSD numbers are extracted from Intel's website.

Technology	Latency		Sequential Bandwidth	
	Read	Write	Read	Write
DRAM	0.1μs	0.1μs	25GB/s	25GB/s
PM	0.2μs	0.5μs	10GB/s	5GB/s
Optane SSD	10μs	10μs	2.5GB/s	2GB/s
NVMe SSD	120μs	30μs	2GB/s	500MB/s
SATA SSD	80μs	85μs	500MB/s	500MB/s
Hard disk	10ms	10ms	100MB/s	100MB/s

CPU's memory controller. Battery-backed NVDIMMs [65, 67] have been available for some time. In the event of a power failure or system crash, an onboard controller safely transfers data stored in DRAM to a non-volatile media with the power from rechargeable batteries, thereby preserving data that would otherwise be lost. Battery-free persistent memory technologies include phase change memory (PCM) [55, 77, 36], memristors [85, 89], and spin-torque transfer RAM (STT-RAM) [13, 48]. Intel's 3D-XPoint technology [64] under the name Optane DC Persistent Memory [41] is now available in the market. All of these technologies offer both longer latency and higher density than DRAM. 3D-XPoint has also appeared in Optane SSDs [40], enabling SSDs that are much faster than their flash-based counterparts.

2.2 Persistent Memory Programming in User Space

The advent of persistent main memory technologies (most notably Intel's Optane DIMMs [41]) brings byte-addressable, non-volatile memory to modern processors. The persistent memory appears in the processor's physical address space. Luckily, similar to kernel programming, user application software also can directly use the PM space.

The most popular operating system mechanism for exposing persistent memory to applications is to use a PM-aware file system to manage a large region of persistent memory. An application can use direct access (DAX) `mmap()` system call to map a file in the file system into

its virtual address space. From there, the application can access the memory directly using load and store instructions, avoiding all operating system overheads in the common case.

2.2.1 PM Programming Support

While DAX `mmap()` provides access, productively and safely using PM introduces challenges that typical PM programming libraries address. Below, we outline the most important of the PM libraries' features.

PM Pools PM libraries provide access to multiple independent *pools* of persistent memory, each with a *root* object from which other objects are reachable.

PM Allocator Each pool has a private, atomic memory allocator for allocating and reclaiming space within the pool that is robust in the face of system crashes.

PM Types Libraries usually identify which types can exist in a pool. For instance, the library may provide a base class or interface that persistent objects should inherit from or implement.

Failure-Atomic Sections Finally, PM libraries provide a means to express atomic sections, usually in the form of a persistent software transactional memory [75, 15, 8] mechanism that specifies regions of code that should be atomic with respect to both failure and concurrent transactions.

2.2.2 PM Bugs

PM programmers must reckon with a wide range of potential bugs that can cause permanent corruption, lead to unsafe behavior, or leak resources. These include common memory allocation/deallocation errors and race conditions, as well as PM-specific challenges.

The three most critical types of PM-specific bugs are logging errors, unsafe inter-pool pointers, and pointers into closed pools.

Logging errors An atomic section must log all persistent updates so that the transaction can roll back in case of a system failure [15, 22, 11, 82, 66, 12]. Failing to log an update manually (as some systems require) can let a poorly-timed system crash compromise data integrity. Updates can be hard to recognize in code, leading to unlogged updates. For instance, passing the address of a field of a persistent struct to a library function might (to the programmer's surprise) modify the data it points to.

Inter-pool pointers PM programs can simultaneously access several independent PM pools in addition to the conventional, volatile heap and stack. The PM pools need to be self-contained so that one pool does not contain a pointer into another pool or volatile memory. Dereferencing such a pointer is certain to be unsafe after a restart.

Pool Closure If a pool closes, the system must unmap the memory it contains, leaving any pointers from DRAM into the pool unsafe [15].

2.3 Persistent Memory Programming in Kernel Application

Directly accessing PM, kernel software applications can store commonly used data and programs closer to the processor so that they can randomly access data dramatically faster, as shown in Table 2.1. That motivates researchers and kernel designers to develop PM-based file systems [17, 20, 21, 83, 86].

Compared with PM, slow block-based storage devices offer larger capacity at a lower price. Tiered storage systems with SSDs and HDDs (such as [52, 92, 90]) have been proposed to bridge the gap between fast and slow block devices. However, the appearance of fast, byte-addressable PM brings even higher performance and direct access feature into tiered storage systems. Therefore, it requires a fundamentally different approach from existing tiered storage systems to fully utilize these unique characteristics of PM.

Acknowledgments

This chapter contains material from “Ziggurat: A Tiered File System for Non-Volatile Main Memories and Disks”, by Shengan Zheng, Morteza Hoseinzadeh, and Steven Swanson, which appeared in the Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST 19), 2019. The dissertation author is the primary investigator and the second author of this paper.

This chapter contains material from “Corundum: Statically-Enforced Persistent Memory Safety”, by Morteza Hoseinzadeh and Steven Swanson, which appeared in the Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 21), 2021. The dissertation author is the primary investigator and first author of this paper.

Chapter 3

Provably Safe Persistent Types using Rust

The advent of persistent memories has brought numerous potential benefits in software development. These include increased system memory capacity, lower latency, and higher bandwidth compared to disk-based storage, and a most significantly unified programming model for persistence and volatile data structure programming without serialization to make it persistent. Besides its advantages, it hosts a set of novel programming challenges. For example, it puts the burden of linearizing the updates on the programmers' shoulders.

PM's non-volatility, high performance, and direct connectivity to the processors saddle the programmer with a set of complex programming problems. Because of its raw performance, system software must be removed from the common-case access path, updates must be resilient in the presence of system failures (if it is to be used as storage), and its memory-like interface pushes application software to deal directly with issues such as fault tolerance and error recovery rather than relying on layers of system software.

Programming with PM further exacerbates the effects of existing forms of bugs while also introducing new kinds of programming errors. Memory leaks, dangling pointers, concurrency bugs, and data structure corruption are all common mistakes that have long-term consequences (rather than dissipating on restart). There is also the possibility of new errors: A programmer may

neglect to log an update or may establish a pointer from a persistent data structure to volatile memory. The former mistake may reveal itself during recovery, but the latter is intrinsically dangerous since the pointer to volatile memory becomes wild after the restart, and dereferencing it will result in an exception.

One of the most significant potential barriers to widespread adoption of PM and our ability to fully harness its possibilities is the difficulty of programming *truly crash-consistent* PM applications. PM will be hampered as a storage technique if programmers are unable to develop and change code that accurately and securely updates persistent data structures.

Years of study and practical tool development have gone into some of the bugs that plague PM programs. Programming disciplines, enhanced library support, debugging tools, and programming language capabilities are all examples of remedies and approaches to these issues.

The Rust programming language provides programming language-based mechanisms to avoid a host of common memory and concurrency errors. Its type system, standard library, and "borrow checker" allow the Rust compiler to statically prevent data races, synchronization errors, and most memory allocation errors. Further, the performance of the resulting machine code is comparable with that of compiled C or C++. In addition to these built-in static checks, Rust also provides facilities that make it easy (and idiomatic) to create new types of smart pointers that integrate cleanly with the rest of the language. Its type system also makes data modification explicit and easy to control.

We have leveraged Rust's abilities to create Corundum, a library (or "crate" in Rust parlance) that lets programmers build persistent data structures. Corundum provides basic PM programming facilities (e.g., opening and mapping persistent memory pools) and a persistent software transactional memory interface to provide atomic updates to persistent data.

Uniquely, Corundum uses static checking (in almost all cases) to enforce key PM programming invariants:

- Corundum prevents the creation of unsafe pointers between non-volatile memory regions

(or “pools”) and pointers from those pools into volatile memory.

- Corundum ensures that programs only modify persistent memory within transactions and that they log all updates to persistent state.
- Corundum prevents most persistent memory allocation errors (e.g. dangling pointers, and multiple frees) in the presence of multiple, independent pools of PM.
- Corundum hides low-level PM programming operations (e.g. preserving write ordering, creating logs, etc.) behind its high-level interface.

Our experience building Corundum demonstrates the benefits that a strong type system can bring to PM programming. We evaluate Corundum quantitatively and qualitatively by comparing it to existing PM programming libraries. We find that Corundum provides stronger guarantees than other libraries and that it is as fast or faster than most of them. We also highlight some changes to Rust that would make Corundum even more powerful and efficient.

The rest of this chapter is organized as follows. Section 3.1 gives a brief background information on Rust programming language. Section 3.2 describes Corundum. In Section 3.3 we evaluate our library. Section 3.4 places Corundum in context relative to related work. Finally, Section 3.5 concludes.

3.1 Rust

Corundum adds PM programming support to Rust programming language by providing facilities for accessing persistent memory, providing a transaction mechanism to ensure consistency in the case of failure, and statically detecting common PM programming bugs. This section provides some necessary information on the Rust programming language to understand Corundum’s capabilities and mechanisms better.

3.1.1 Overview

We use Rust to develop a PM library with solid safety guarantees. The Rust programming language [50] is intended for parallel, low-level, system programming and it uses a sophisticated type system to prevent a range of common programming errors. For instance, its type system makes data races impossible, and it provides cheap reference counting garbage collection. The result is a thoroughly modern language faster than C or C++ and with vastly stronger safety guarantees.

Rust is well-loved by the developers who use it despite having a moderately steep learning curve. It requires programmers to think differently about variables' lifetimes, mutability, and concurrency.

Since the memory allocation, synchronization, and safety invariants that PM programming provides are a strict superset of conventional volatile programming, we can leverage Rust's safety guarantees to improve the safety of PM programming.

Rust's guarantees are not builtin in the language itself but are implemented in its standard library. This means we can use the same language features to enforce new guarantees in an idiomatic and familiar (to Rust programmers) way.

Below we, describe the key features of Rust that Corundum uses to provide PM-specific safety guarantees. For a thorough introduction, consult the Rust manual [50].

3.1.2 Ownership and Borrowing

The notion of "ownership" is central to Rust's ability to guarantee memory safety. At any time, each value in Rust has a single variable that "owns" it. The value drops when the owner goes out of scope, and its resources can be reclaimed. Programs can transfer ownership through variable assignment or passing the variable to a function.

In addition to simple variables, Rust also supports "references" that refer to values without

owning them. Passing a reference to a function causes the function to "borrow" the value from the caller. While the value is borrowed, the owner cannot access it.

By default, variables and references are immutable, and programmers must take explicit action to create mutable variables and references. Further, only a single mutable reference can exist at any time.

3.1.3 Resource Management through RAII

Rust uses the RAII (Resource Acquisition Is Initialization) model, which invokes the object's destructor when it goes out of scope before freeing the resources. Rust makes sure that all dependent objects are dropped by recursively invoke the destructor of every field. This creates a finite chain of calls to the destructors and properly reclaiming the allocated memory.

However, data is not dependent on pointer objects. Therefore, Rust provides `Drop` trait that allows types to perform more operations while being dropped. Using this trait, the pointer types can branch the dropping chain to include their referents too.

Rust has a particular built-in trait called `Copy`. On assignments, the compiler chooses between 'move' and 'copy' operations based on the type implementing or not implementing the 'Copy' trait. Rust statically checks the types and marks them as '`Copy`' (e.g., an integer) or '`!Copy`' (e.g., a smart pointer). Borrowing rules do not apply to `Copy` types since the compiler copies them to the new owner's location on assignments. Primitive types, for instance, are `Copy`. On the contrary, the compiler uses the 'move' operation when assigning a `!Copy` type to a new owner to transfer the ownership of its resources after checking the borrowing rules for memory safety. Pointer types, as an example, are `!Copy`.

The `Copy` and `Drop` traits are exclusive because a `Copy` type is a trivial type that can be copied on assignments while the `Drop` trait is used in an RAII context. Therefore, once the programmer implements the `Drop` trait for a custom type, Rust annotates it as a `!Copy` data type and applies the borrowing rules to it and any other type containing it.

3.1.4 Mutability, Immutability, and Interior Mutability

Mutability is another central concept in Rust that governs when a value may change. The critical property that Rust maintains is that there can be one mutable reference to a piece of data or multiple immutable references, but not both. We refer to this as *the mutability invariant*. In most instances, Rust enforces this invariant statically.

In some cases, static checks are too restrictive, or the program may need to enforce further constraints on when data can be mutable. Rust provides *interior mutability* for these situations, and the wrapper type `RefCell` exemplifies this concept. Curiously, `RefCell`s are always immutable so assigning to them is impossible. However, the program can acquire mutable and immutable references by calling `RefCell::borrow_mut()` and `RefCell::borrow()`, respectively. These functions return reference objects (similar to smart pointers) that programs can use to access the data. The key is that the `RefCell` dynamically enforces the mutability invariant: Calling `borrow()` when a mutable reference object exists or `borrow_mut()` when any reference object exists will cause a `panic!()`.

3.1.5 Smart Pointers, Wrappers, and Dynamic Memory Allocation

Rust uses smart pointers and type wrappers to implement memory management, garbage collection, and concurrency control. Defining new kinds of smart pointers, type wrappers, and guard objects that integrate cleanly into the language is easy, since Rust makes dereferencing smart pointers fully transparent.

Rust's standard library provides a range of smart pointer types and type wrappers (the <> notation is Rust's syntax for generics):

1. `Box<T>` is a pointer to an object of type `T` allocated on the heap. When a `Box<T>` goes out of scope, the allocated data is freed. `Box` continuously extends dropping to its pointee as it is the only owner of its resources.

2. `Rc<T>` and `Arc<T>` are reference-counted pointers to heap-allocated objects of type `T`. `Arc<T>` is the thread-safe version of `Rc<T>`, as it uses atomic counters for reference counting. Multiple instances of these pointers can point to the same data, and when the last owner goes out of scope, the allocation is reclaimed. Since multiple `Rc/Arc` instances can refer to the same data, they do not allow the modification of data they hold. `Rc` and `Arc` extend dropping to their pointees only when they find no strong reference to the data.

3. `Cell<T>` and `RefCell<T>` (described above) are wrapper types holding an object of type `T` that is immutable except via interior mutability. While `RefCell` can contain any object, `Cell` may hold only `Copy` types.

4. `Mutex<T>` is a thread-safe version of `RefCell`. Its `lock()` method locks the Mutex and returns a reference object that is also a guard object for the Mutex. When the reference goes out of scope, the lock is released.

Each wrapper encapsulates a specific set of capabilities, and programmers can compose them to build feature-rich data types. For instance, a common idiom – `Rc<RefCell<T>>` – combines `RefCell` with `Rc` to provide shared, mutable objects.

When the last reference to an object dies, Rust "drops" it. Recursively dropping a struct drops any fields it contains. Types can implement `drop()` to implement destructor-like functionality. For instance, `drop()` for `Box` deallocates the memory it holds, and `drop()` for `Rc` decrements the reference count and frees the memory if it reaches zero.

Option Type

Rust provides optional values in form of `Option<T>` that can be either `Some(data)` (where `data` is of type `T`), or `None`. Its most common use is to allow null pointers: `Box<T>` is always a valid pointer to allocated memory, but `Option<Box<T>>` can either be `None` or a `Box<T>`.

`Option` is one instance of Rust's `enum` mechanism, and Rust provides a `match` construct that is more powerful than `switch` in C as it matches the patterns of either side of each arm, and it requires the programmer cover all possible values. For `Option<PBox<T>>`, this avoids null dereferences and forces code to account for pointers that might be null.

3.1.6 Traits and Type Bounds

Rust provides *traits* that are similar to Java or C# interfaces. Structures can implement traits. A ubiquitous Rust idiom is to "bound" or restrict the types used in a particular context based on the traits they implement.

Of particular importance are "auto traits" that all types (even primitive types) implement by default but can opt-out of it. For instance, all types implement the `Send` trait so they can be sent to another thread. Rust allows negative implementation (using `!` symbol) of auto traits for opting out of it. For instance, `Rc<T>` is labeled as `!Send` because it uses non-atomic counters and is, therefore, not safe to share among threads.

3.1.7 Panic

In response to severe errors, programs can call `panic!()` to abort the program. Rust provides an exception-like mechanism to catch panics and attempt to recover from them.

3.1.8 Unsafe Rust

Rust provides `unsafe` code blocks, which give the programmer access to a superset of regular, safe Rust. Unsafe Rust allows C-like direct manipulation of memory, unsafe casts, and other similar operations. The Rust standard library uses unsafe Rust to implement interior mutability and other critical features (e.g., system calls and low-level memory management).

Libraries like Corundum can declare types and functions to be unsafe, preventing their invocation or use in safe code.

If programmers decide to use `unsafe` constructs, they assume responsibility for enforcing the guarantees that Rust (or the library) relies upon. Typical Rust programmers do not use `unsafe` constructs.

3.2 Corundum

Corundum is a Rust library (or "crate") that uses static and dynamic checking to avoid common PM programming errors. Aside from providing these strong safety guarantees, Corundum is similar to other PM programming libraries like PMDK [75], NV Heaps [15], and Mnemosyne [82]. Corundum provides four abstractions – typed persistent memory pools, transactions, persistent smart pointers, and atomic memory allocation – that address the challenges and common bugs described in Section 2.2.

Corundum strives to provide strong safety guarantees for persistent memory programming. It achieves the following design goals:

Design Goal 1 (Only-Persistent-Objects)**:** *Pools only contain data that can be safely persistent.*

Design Goal 2 (Ptrs-Are-Safe)**:** *Pointers within a pool are always valid. Pointers between pools or from persistent memory to volatile memory are not possible. Pointers from volatile memory into a pool are safe. Closing a pool does not result in unsafe pointers.*

Design Goal 3 (Tx-Are-Atomic)**:** *Transactions are atomic with respect to both persistent and volatile data. It is not possible to modify persistent data without logging it.*

Design Goal 4 (No-Races)**:** *There are no data races or unsynchronized access to shared persistent data.*

Design Goal 5 (Tx-Are-Isolated)**:** *Transactions provide isolation so that updates are not visible until the transaction commits.*

Design Goal 6 (No-Acyclic-Leaks)**:** *Memory leaks in acyclic data structures are not possible.*

Corundum achieves these goals through a combination of several core techniques that are embodied by Corundum's abstractions. Corundum borrows and relies upon existing features of Rust to prevent data races and avoid unsafe pointer manipulation. These properties are a foundation that Corundum builds upon.

Corundum breaks a program's execution into transactions and non-transactional code, and a thread's execution alternates between the two. Transactions are atomic, isolated, and can modify persistent state but cannot modify pre-existing volatile state. The remaining, non-transactional code can modify volatile state but cannot modify persistent state.

Critically, Corundum limits the kinds of variables that can cross transaction boundaries, and this allows for careful reasoning about the invariants that hold at the boundaries. For instance, Corundum guarantees that leaks in acyclic structures are not possible by showing that they do not exist at transaction boundaries.

3.2.1 Assumptions

The correctness of Corundum rests on several assumptions. We assume that the program does not use `unsafe` constructs, since unsafe Rust can bypass almost all of the guarantees that Rust makes and relies upon.

We also assume that our implementation of the Corundum memory allocator, logging and recovery code, and reference counting system are correct. We have tested them thoroughly and techniques for building correct versions of these components are well-known [15, 82, 11, 75, 8].

Finally, students of Rust will notice that we do not discuss some common types (e.g., `Cell` or `Weak`) or their persistent analogs. Corundum's treatment of these types is analogous to the types we discuss and Corundum provides persistent counterparts. We have omitted the details for brevity.

3.2.2 Corundum Pools and Objects

Corundum provides self-contained pools of persistent memory and constrains the data types that they can hold.

Pools. Corundum pools reside in a PM-backed file. The pool contains some metadata, a root pointer, persistent memory allocation data structures, and a region of persistent memory.

Corundum identifies each memory pool with a *pool type*, and all persistent types take a pool type as a parameter. This statically binds each persistent object to its pool. Likewise, transactions are bound to a particular pool via the pool type. In our discussion, we use `P` as a representative pool type.

Programs open a pool by calling `P`'s `open()` function: `P::open<T>("foo")`. This binds `P` to the pool in file `foo`. Corundum ensures that only one open pool is bound to `P` at any time. The type parameter, `T`, determines the type of the root pointer. `P::open<T>()` returns an immutable reference to the root pointer. Pools remain open until reference to the root object drops.

Programmers can statically declare multiple pool types using the `pool!()` macro, but the number of pool types available (and, therefore, the number of simultaneously open pools) is fixed at compile time.

Persistent Objects. Objects in a pool must implement the `PSafe` auto trait. Corundum declares all primitive arithmetic types to be `PSafe`, so these types and structs composed of them are `PSafe`.

`PSafe` (like all of Corundum's auto traits) is declared `unsafe`, so programmers should not explicitly label types as `PSafe`.

Reference and raw pointers, and types that refer to state external to the program (e.g., file handles) are `!PSafe`. Having a `!PSafe` field also makes a type `!PSafe`. As a result, `Box`, `Rc`, `Arc`, `Mutex`, and the other types in Rust containing raw pointers are `!PSafe`.

```
1  #[derive(Root)]
2  #[pools(P)]
3  struct Node<T: PSafe, P: MemPool> {
4    val: T,
5    next: PRefCell<Option<Pbox<Node<T, P>, P>>,P>
6  }
7
8  impl<T: PSafe, P: MemPool> Node<T, P> {
9    fn append(n: &Node, v: T, j: &Journal<P>) {
10     let mut t = n.next.borrow_mut(j);
11     match &*t {
12       Some(succ) => append(succ, v, j);
13       None => *t = Some(Pbox::new(
14           Node {
15             val: v,
16             next: PRefCell::new(None, j)
17           }, j));
18     }
19   }
20 }
21
22 fn go(v: i32) {
23   let head = P::open::<Node<i32,P>>("list.pool", 0);
24   P::transaction(|j| {
25     append(&head, v, j);
26   });
27 }
```

Listing 1: A Corundum implementation of linked list append. Some error management code has been elided for clarity.

3.2.3 Transactions: The Basics

All modifications to a Corundum pool occur within a transaction, including memory allocations and modifications of the root pointer.

Programs create transactions by passing an anonymous function (i.e., a lambda) to `P::transaction()` (Lines 24-26 of Listing 1). The lambda takes a single argument, `j`, which is a reference to a *journal object* that holds information about the current transaction. Variable `j` is of type `Journal<P>`, so it is bound to pool `P`.

```
1  let mut done = false;
2  let p1 = P::transaction(|j|{
3      let p1 = Pbox::new(1, j);
4      let p2 = Pbox::new(2, j);
5      root.set(p2);
6      done = true;
7      ^^^^ the trait `TxInSafe` is not
8           implemented for `&mut bool`
9      p1
10     ^^ the trait bound `Pbox<i32,P>:TxOutSafe`
11        is not satisfied
12 }).unwrap();
13 ^ `p1` is dropped here.
14 ^ `p2` is alive and durable here because it
15   is reachable from the root object.
```

Listing 2: Reachability though lifetime and type bounding

The lambda can capture values from the transaction's lexical scope (e.g., `head` in Listing 1), allowing transactions to integrate smoothly into the surrounding code. `P::transaction()` returns the return value of the transaction body.

Corundum flattens nested transactions: Modifications to a pool commit when outermost transaction for that pool commits.

The programmer is responsible for acquiring locks in the correct order to avoid deadlock.

Rust's type system allows Corundum to restrict the inputs and output of the transaction. The `TxInSafe` auto trait bounds the types a transaction can capture. Corundum marks all volatile mutable references, smart pointers, wrappers, and interior mutability types as `!TxInSafe`.

This provides our first invariant:

Invariant 1 (TX-No-Volatile-Mutability)**:** *Transactions cannot modify existing volatile state.*

Details: Since all the types that provide mutable access to volatile data are `!TxInSafe`, pre-existing instances of these variables are not available inside the transaction (Line 6 of Listing 2). However, variables can be created and modified within the transaction. Preexisting volatile data can be read.

```
1 P1::transaction(|j1| {
2     let v = Box::new(10);
3     let p1 =
4       Pbox::new(v, j1);
5                 ^ Box<i32>:PSafe is not satisfied
6 }).unwrap();
```

Listing 3: Only persistent-safe objects are acceptable.

`TxOutSafe` bounds the values a transaction can return. All references and pointers are `!TxOutSafe`, so transactions can return data only by value. Line 9 of Listing 2 shows how the compiler complains when a user attempts to send out a persistent object.

Corundum also uses the concept of a *stranded type*: If a type is `!TxOutSafe`, `!Send`, and `!PSafe`, then instances of that type cannot escape a transaction. This is because 1) Since a stranded type is `!TxOutSafe`, the transaction cannot return it, 2) since it is `!PSafe`, it cannot be stored in a pool for later retrieval, 3) since it is `!Send`, it cannot be passed to another thread, and 4) it cannot be assigned to a volatile variable (TX-No-Volatile-Mutability).

Design Goal 1 (Only-Persistent-Objects) Holds: The declaration of `P::open()` includes a bound to ensure that the root is `PSafe`. If it contains references or pointers they must be one of the persistent reference/pointer types (see below), since volatile pointers and references are `!PSafe` (e.g. Listing 3).

`PSafe` is a type bound on all of the persistent smart pointer and wrapper types, so those types can only point to, refer to, or wrap `PSafe` objects.

Corundum also carefully constrains the availability of journal objects:

Invariant 2 (TX-Journal-Only)**:** *Journal objects are only available inside transactions.*

Details: The constructor for the `Journal<P>` is unsafe, so the program cannot safely create one. Therefore, the only journal objects that might be available are the ones passed to a transaction as an argument. `Journal<P>` is stranded, so it cannot escape the transaction.

3.2.4 Pointers to Persistent Data

Corundum's smart pointer, smart reference, and wrapper types play a crucial role in avoiding persistent programming errors, since they mediate access to persistent state. Their design prevents pointers from one persistent pool to another, prevents the modification of persistent state outside of transactions, and plays a role in avoiding memory leaks.

Table 3.1 summarizes Corundum's persistent smart pointer types. With the exception of `VWeak`, they mirror Rust's volatile smart pointers. The interface differs in two ways: First, each type takes a pool type as a type parameter, so pointers that reside in different pools have different types. Second, their constructors and mutating accessors take a journal object as an argument.

Three of the methods listed – `PRefCell::borrow()`, `PRefCell::borrow_mut()`, and `PMutex::lock()` – return objects that behave like references. These objects are all stranded.

`VWeak` is the only way to hold a volatile pointer to persistent data. It is "weak" in the sense that it does not affect reference counts. The `promote()` method grants access to the data a `VWeak` refers to by providing a `Parc` that refers to the same data. Promoting is only possible within a transaction. `Parc::demote()` creates `VWeak` pointers.

Design Goal 2 (Ptrs-Are-Safe) Holds: The presence of multiple pools of PM alongside the volatile heap and stack means the potential for several different kinds of pointers, complicating pointer safety.

Pointers within a pool These pointers are allowed, and Corundum relies on Rust's safety properties and mechanisms to ensure their safety.

Pointers between pools Inter-pool pointers are inherently unsafe. They are not possible in Corundum, because pointers to different pools have different types, and assignment between types is not allowed in Rust.

Pointers from a pool to volatile memory These pointers are also inherently unsafe. They are also disallowed: Pools only contain `PSafe` objects (Only-Persistent-Objects) and pointers

Table 3.1: Corundum's smart pointers and type wrappers for persistent data corresponding closely Rust's pointers and wrappers for volatile data. The key differences are that the Corundum types takes pool type as a type parameter, binding them to a particular pool.

Corundum type	API	Description
PMEM Smart Pointers for Dynamic Allocation		
`Pbox<T,P>`		Statically scoped, unshared pointer to PMEM. Deallocates when it goes out of scope.
	`new(value: T, j: &Journal<P>)`	Allocate PMEM in `P` and initialize it.
`Prc<T,P>`		Dynamic PMEM allocation with thread-unsafe reference counting.
`Parc<T,P>`		Dynamic PMEM with thread-safe reference counting.
	`new(value: T, j: &Journal<P>)`	Allocate PMEM in `P` and initialize it.
	`pclone(j: &Journal<P>)`	Create a new reference to the data.
	`downgrade()->PWeak`	Return a persistent weak (`PWeak<T,P>`).
	`demote()->VWeak`	Return a volatile weak pointer (`VWeak<T,P>`).
`PWeak<T,P>`	`upgrade(j: &Journal<P>)->Prc/Parc`	Convert to a `Option<Prc/Parc<T,P>>` if it is available
`VWeak<T,P>`	`promote(j: &Journal<P>)->Prc/Parc`	Convert to a `Option<Prc/Parc<T,P>>` if it is available
PMEM Wrappers for Interior Mutability		
`PCell<T,P>`		Interior mutability via copying data to and from PMEM using `get()` and `set()` functions.
`PRefCell<T,P>`		Interior mutability via references with dynamic borrow checking.
	`new(value: T, j: &Journal<P>)`	Create new instance on the stack and initialize it.
	`borrow()`	Return an immutable reference to value it contains.
	`burrow_mut(j: &Journal<P>)`	Return a mutable reference object (`RefMut`) for the value inside.
`PMutex<T,P>`		Thread-safe interior mutability via references.
	`new(value: T, j: &Journal<P>)`	Create new instance on the stack and initialize it.
	`lock(j: &Journal<P>)`	Lock and return a mutable reference (`PMutexGuard`).

to volatile memory are `!PSafe`. Corundum additionally provides safe generational volatile cells (`VCell`) that reset when the pool is reopened.

Pointers from volatile memory into a pool `VWeak` is the only way store a pointer to PMEM in volatile memory. The object a `VWeak` refers to can disappear if the last `Prc`/`Parc` refering to the object goes out of scope, deallocating the memory. In this case, `promote()` will return `None`, which is safe in Rust.

Pointers into closed heaps If a heap closes, dereferencing any pointers into the heap becomes unsafe. Corundum combines three approaches to prevent this.

First, accessing a pointer to a closed pool from within a transaction is not possible, since `P::transaction()` will `panic!()` if `P` is closed, and a pool will not close while a transaction is in progress.

Second, consider a reference (other than a `VMWeak`), `A`, that exists outside a transaction. Rust's borrow checker requires that a chain of in-scope references exist from the root of the pool to `A`. This includes the root pointer, whose liveness prevents `P` from closing.

Finally, `VWeak` pointers from volatile memory into the pool can exist after the pool closes. However, `VWeak::promote()` requires a journal object to retrieve a usable reference, so it can only be called from inside a transaction, which is only possible if `P` is open (see above).

3.2.5 Transactions: Mutability and Isolation

Corundum allows modification of persistent data only via interior mutability and only inside a transaction. Two wrapper types in Table 3.1 provide interior mutability for persistent data: `PMutex` and `PRefCell`.

`PMutex::lock()` returns a mutable reference to the data while acquiring a lock. The lock is automatically release at the end of the transaction.

```
1  P1::transaction(|j1| {
2      let p1 = LogCell::new(
3          Pbox::new(1, j1), j1);
4      P2::transaction(|j2| {
5          ^^^^^^^^^^^ `j1` is not `TxInSafe`
6          let p2 = Pbox::new(1, j2);
7          p1.set(p2, j1);
8                 ^^ expected P1, found P2
9      }).unwrap();
10 }).unwrap();
```

Listing 4: Cross-Pool referencing prevention via type system

`PRefCell` returns mutable and immutable references via `PRefCell::borrow_mut` and `PRefCell:borrow()`, respectively. It dynamically enforces Rust's mutability invariants for these references.

Invariant 3 (Mutable-In-Tx-Only)**:** *Mutable references to persistent data in `P` can only exist inside transactions on `P`.*

Details: The program can create a mutable reference to persistent data by calling `borrow_mut()` or `lock()` on a smart pointer. Since both of these functions require a journal object as a parameter, they can only be called inside a transaction (TX-Journal-Only).

The resulting mutable reference object (either a `PRefMut` or a `PMutexGuard`) is stranded, so it will be destroyed when it goes out of scope at the end of the transaction.

The reference that `open()` returns to the root object is immutable, so initially, there are no mutable references to pool data available outside a transaction.

Since new mutable references that a transaction creates are stranded, the number of such references outside a transaction cannot increase, so there will never be such a reference.

Design Goal 3 (Tx-Are-Atomic) Holds: For persistent data, Corundum's atomicity guarantee relies on the atomicity of the memory allocator and on all modifications to persistent data being logged.

The allocator and journal object ensure that allocations do not become persistent until the

transaction commits. Therefore, on a system failure or `panic!()`, the allocations roll back to reclaim the allocated memory.

Corundum enforces logging by requiring a journal object to make changes to data. To modify persistent data, the program needs a mutable reference object from `PRefCell::borrow_mut()` or `PMutex::lock()`. The reference object performs undo logging the first time it is dereferenced.

Atomicity should include updates to volatile state as well, since a transaction can abort if it calls `panic!()`. Corundum transactions are trivially atomic for volatile state because they cannot modify volatile state (TX-No-Volatile-Mutability).

Design Goal 4 (No-Races) Holds: Rust prevents data races and unsynchronized access using the mutability invariant, the `Mutex` type, and marker types to restrict data movement between threads. Corundum takes the same approach by providing `PMutex`, and achieves the same safety guarantees.

Design Goal 5 (Tx-Are-Isolated) Holds: Isolation requires that changes in an uncommitted transaction are not visible to concurrently executing code.

Since transactions cannot modify shared volatile data (TX-No-Volatile-Mutability), we only need to consider changes to persistent objects.

A thread must hold a lock before reading or writing shared persistent state and this can only occur inside a transaction (TX-Journal-Only). Once a thread holds the mutex, no other thread can read the data it protects until the transaction commits and the lock is released, so other threads are isolated from those changes.

3.2.6 Memory Management

Corundum constrains where programs can allocate and deallocate persistent memory and provides an allocator that can atomically commit or roll back all the allocations and deallocation that occur in a transaction.

Corundum adopts Rust's reference counting garbage collection mechanism. `Parc` and `Prc` smart pointers provide persistent reference counting. Corundum (and Rust) also support weak references to allow for cyclic data structures.

Like Rust's `Rc` and `Arc`, `Prc` and `Parc` provide a method, `pclone()`, to create a new reference to the shared data and increment the reference count. The reference counts are persistent and must be logged, so `pclone()` takes a journal object argument.

Allocation. The only way to allocate persistent memory is by creating `Pbox`, `Prc`, or `Parc` instances. Since the constructors for these types require a journal object, allocation cannot occur outside a transaction.

Deallocation. When a reference count goes to zero (for `Prc` or `Parc`) or a `Pbox` goes out of scope, the variable is "dropped" signifying that the allocator can reclaim the memory. However, instead of releasing it immediately, Corundum logs the release and performs it during transaction commit.

Logging occurs in `drop()` (i.e., the destructor) for `Prc`, `Parc` and `Pbox`. Corundum must ensure that deallocation only occurs within a transaction.

This guarantee holds since the destruction of an object only occurs in response to a change in another persistent object (e.g., the destruction of the last reference to that object). Since Corundum only allows changes to persistent memory inside transactions (Mutable-In-Tx-Only), the resulting object destruction will occur in the same transaction.

3.2.7 Memory Allocation

Corundum extends Rust's reference-counting memory management system to prevent persistent memory leaks (in the absence of cycles) and multiple-frees. Corundum provides an additional guarantee: It ensures that, by the end of the allocating transaction, the newly allocated persistent memory is reachable from some previously allocated persistent memory. This ensures that crashes (which invalidate all volatile pointers) do not leak volatile memory.

Design Goal 6 (No-Acyclic-Leaks) Holds:

Corundum adds three mechanisms extending Rust's reference counting mechanism to prevent memory leaks in acyclic persistent data structures. First, the atomicity of memory allocations within a transaction prevents the creation of orphaned data if a transaction does not commit. Second, the transaction cannot assign a newly-allocated PMEM region to a captured volatile variable (TX-No-Volatile-Mutability). This prevents persistent data from being kept alive solely by a volatile reference. Third, since the persistent pointers types are `!TxOutSafe`, PMEM data cannot escape the transaction via the transaction's return value.

As a result, the only way a new PMEM allocation can outlive the transaction is to become reachable from a region of persistent memory that was allocated in an earlier transaction.

The argument above hinges on the invariant that a reference to orphaned memory cannot exist outside a transaction. However, Rust's mechanism for spawning a new thread provides a potential escape. Consider this example:

```
1  P::transaction(|j| {
2    let a = Parc::new(j, 42);
3    thread::spawn(move |k| {
4      let b = a;
5    });
6  });
```

Rust's `thread::spawn()` function executes its argument (a lambda) in a new thread. In the code, `a` is an orphan but it moves to the thread's scope. The thread's body is outside the transaction, so our invariant does not hold. To prevent this, Corundum makes `Pbox` and `Parc` `!Send` to prevent their capture by or transmission to another thread. `VWeak` remains `Send`, so it can be used to transmit persistent state between threads.

3.2.8 Example

Lines 9-19 in Listing 1 implement `append()` for a persistent linked list in Corundum. A `Node` contains a value of type `T` and a link to the next `Node`. In Line 5, the link field is defined as a `PRefCell<Option<Pbox<Node<T,P>,P>>,P>`, which might seem daunting, but this is typical for a Rust pointer declaration. To break it down: `PBox<Node, P>` is pointer to a `Node<T,P>` in pool `P`. `Option<>` allows the pointer to be `None`. Wrapping the `Option` in `PRefCell` allows for modification via interior mutability.

The function `append()` recursively finds the end of the list `n`, and adds a `Node`. Line 10 uses `PRefCell::borrow_mut()` to get a mutable reference, `t`, to the `Option` object the `PRefCell` contains. Line 11 uses Rust's `match` construct to safely handle all possible values of `t`: `None` or `Some`. In the `Some` (i.e., non-null) case, it binds the content of the `Option` (which has type `Node`) to `succ`, and recursively calls `append()`.

If the `Option` is `None`, the code has reached the end of the list. Line 13 creates a `PBox` to allocate a new `Node` with value `v` and a `next` pointer equal to `None`. It wraps the `PBox` in a non-null value of type `Option`, and assigns it to the mutable reference.

Function `go()` opens "list.pool" and binds it to pool type `P`. The root pointer will hold a `Node` struct. Line 24 starts a transaction, which provides a journal object, which Line 25 passes to `append()`.

Several aspects of the code are notable. First, `head` and `n` are both immutable, so changes are not possible until `borrow_mut()` uses interior mutability to return a mutable reference object. Second, we must pass `j` into `append()` to ensure it executes in a transaction thereby allowing the call to `borrow_mut()` and the memory allocation (Line 13). Third, although we create call `borrow_mut()` for every link in the list, Corundum only logs the last one, since logging only happens when `*t` dereferences the reference object (Line 13). Forth, as written, `Node` and `append` only work on pool type `P`. A more complete implementation would make `P` a generic type parameter, so they could work on any pool type.

Table 3.2: Corundum more static checks than other PMEM libraries, using them to meet most of its design goals. ('S'=Static, 'D'=Dynamic, 'M'=Manual, 'GC'=Garbage Collection, 'RC'=Reference Counting)

System	Only-P-Object	Ptrs-Are-Safe			No-Races	Tx-Are-Atomic		No-Leaks
		Interpool	NV-to-V	V-to-NV		Atomicity	Isolation	
NV-Heaps [15]	M	D	S	M	S	S	M	RC
Mnemosyne [82]	M	D	S	M	S	S	M	M
libpmemobj [75]	M	D	M	M	M	M	M	M
libpmemobj++ [75]	M	D	M	M	M	S	M	M
NVM Direct [8]	D	D	S	D	M	S/M	S/M	M
Atlas [11]	M	M	M	M	M	S	M	GC
go-pmem [11]	M	M	M	M	M	S	M	GC
Corundum	S	S/D	S	D	S	S	S	RC

3.2.9 Limitations and Potential Improvements to Rust

Corundum's design statically prevents many but not all bugs that might occur in persistent programs. The design decisions of Rust also impact Corundum's design and place some limits on what it can achieve.

Uncaught Bugs. Corundum aims to protect the programmer from errors that violate the basic rules of persistent memory programming as enshrined in its design goals. However, Corundum does not attempt to protect against higher level errors (e.g., whether the algorithm in our example correctly appends to a linked list).

Dynamic Checks. In some cases, Corundum provides dynamic, rather than static, checks. In these cases we could not find a way to enforce them with Rust's type system.

Corundum performs dynamic checks to protect against unsafe dereferencing of `VWeak` pointers from DRAM into PM that can arise when a pool closes. Dereferencing these pointers is common – imagine a volatile index that stores pointers to persistent objects – so static checks would be preferable. An enhanced version Rust's lifetime mechanism might be of use in this case, since it might be able to keep the pool open until all pointers into it went out of scope.

Threads in Transaction. Corundum goes through some contortions to make it safe to spawn a thread inside a transaction, since there is no way to restrict where thread spawning is possible. The challenge is that `thread::spawn()` may leak an unreachable `Parc` when called in a transaction. To prevent this, `Parc` is `!Send`, which requires using `Parc::VWeak` to pass persistent pointers to child threads, which is cumbersome.

A solution would be to generalize Rust's ability to bound the variables that a transaction can capture to include functions. We could then make `thread::spawn() !TxInSafe` to prevent the transaction from calling it.

Deadlock. Corundum does not prevent deadlock despite several demonstrations that this is possible in a TM system [91, 6, 31, 74]. We omitted deadlock detection and recovery for simplicity and to align `PMutex`'s behavior with `Mutex`'s.

Log-Free Programming. Corundum is more restrictive than most existing PM libraries. For instance, many high-performance PM data structures are log-free and use carefully-ordered updates to ensure crash consistency. More permissive libraries allow this, but such code would not compile under Corundum. Ideally, Corundum could grow to include `unsafe` facilities that allow for log-free programming without completely sacrificing its safety properties.

Cyclic References. Like all reference-counting memory management systems, Rust (and Corundum) can leak memory in cyclic data structures. The consequences are more severe for Corundum, since the memory is persistent. Several solutions are possible (e.g., a more general garbage collection mechanism), but they would increase complexity, reduce performance, and create a mismatch between Corundum's behavior and Rust's.

Other Languages. We chose Rust as the basis for Corundum after considering several alternatives. C is notoriously unsafe. Well-behaved C++ code is an improvement and NV-Heaps and the `libpmemobj`'s C++ bindings demonstrate that C++ smart pointers and lambdas can provide some of Corundum's checks, but there are several significant gaps. For instance, there is no way to limit the types a lambda can capture.

Go [29] emphasizes simplicity and go-pmem [28] provides basic PMEM programming facilities. However, Go does not allow smart pointers or provide a sufficiently expressive type system to statically enforce the invariants that Corundum provides.

Pony [76] is a new language with similar design goals to Rust. For instance, it has a sophisticated notion of mutability and statically prevents data races, just as Rust does. However, Pony is less mature and more complex than Rust.

3.2.10 What is Essential and What Is Rust?

The biggest open question for Corundum is what aspects of its design (and our correctness argument) are byproducts of choosing Rust and which represent something more fundamental about persistent memory safety.

For instance, Corundum is unique among PMEM libraries in separating code that modifies persistent state from code that modifies volatile state. Our correctness argument relies on this property in several places, but it is not clear whether this is a fundamentally good idea for PMEM programming or simply something that was helpful in ensuring safety and was implementable in Rust.

It would be instructive to try to replicate Corundum's functionality in another language. C++ seems like the most likely mainstream candidate, since it has a flexible type system. Recreating Corundum's approach in C++ would require first recreating the Rust memory safety guarantees that Corundum relies on. This seems challenging. More illuminating would be to try to achieve the same goals using a more idiomatic C++ approach.

3.3 Evaluation

We evaluate Corundum along three axes: its success in statically enforcing PM safety properties, its ease of use, and its performance.

3.3.1 Static Checking

Corundum aims to make the programmer's life easier by statically enforcing PM safety at compile time rather than relying on dynamic checks and testing to identify bugs. To measure its success in this regard, we compare it with other PMEM programming systems.

Table 3.2 summarizes how Corundum and other PM libraries detect violations of Corundum's six design goals. In the table, "S" (for "Static") means that the compiler either enforces the invariant automatically (e.g., by generating safe code) or detects any violations and reports them, "D" ("Dynamic") means that the system will identify the problem at runtime and exit appropriately, and "M" ("Manual") means that the system does not detect violations, so they will manifest as a crash, data corruption, or other error. For No-Leaks, "GC" means the system

Table 3.3: Adding persistence to data structures with Corundum requires fewer changes (measured in lines of code) than PMDK.

App	Rust	Corundum	C++	PMDK
Linked List	192	+19 (9.9%)	146	+45 (30.8%)
Binary tree	256	+12 (4.7%)	208	+41 (19.7%)
HashMap	165	+10 (6.1%)	137	+42 (30.7%)

provides garbage collection, and "RC" means that reference counting is used.

The table shows that Corundum enforces almost all its invariants at compile time, compared to the relatively few compile-time checks other systems provide.

In some cases, this difference represents a design trade-off. For instance, NVM Direct explicitly supports unlogged stores as performance optimization. Likewise, four of the systems allow unsynchronized access which is faster but less safe. A Corundum programmer could use similar techniques to improve performance with `unsafe` blocks.

3.3.2 Ease of Use

A key goal of Corundum is to make writing safe persistent memory programs easier. Qualitatively, we would expect that the stronger static guarantees that Corundum provides should lead to less debugging. This is especially valuable since many of the bugs that Corundum protects against would manifest during a failure, making them more difficult to test. Our experience using Corundum bears this out: once code compiles, it works reliably. Getting the code to compile can take a while.

Quantitatively, we can measure programing effort by lines of code needed to add persistence to a conventional program. We implemented three data structures in C++ and Rust and then added persistence using PMDK and Corundum. Table 3.3 shows that Corundum required adding fewer lines in both relative and absolute terms.

3.3.3 Evaluation Platform

Our test platform has dual 24-core Cascade Lake processors. The CPUs are engineering samples with specs similar to the Xeon Platinum 8160. In total, the system has 384 GB (2 socket $\times$ 6 channel $\times$ 32 GB/DIMM) of DRAM, and 3 TB (2 socket $\times$ 6 channel $\times$ 256 GB/DIMM) of Intel Optane DC DIMMs. Our machine runs Fedora 27 with Linux kernel version 4.13.0.

Corundum uses some unstable features of Rust, so we use Rust Nightly version 1.52.0 built with 'release' profile. We compared Corundum with PMDK 1.8, Atlas, Mnemosyne built with '-O2'. All of them use '`clflushopt`' for durability without using non-temporal store. The go compiler applies optimizations to go-pmem by default. We use `Ext4-DAX` to mount the persistent memory and create the pool files.

3.3.4 Performance

We compared our library with PMDK's libpmemobj and libpmemobj++ by porting some PMDK data structures to Atlas, Mnemosyne, go-pmem, and Corundum.

Basic Operation Performance

Table 3.4 reports the latency of basic operations measured on the platform described in Section 3.3.3. To evaluate the impact of storage technology, we measure these operations on both Optane DC persistent memory and DRAM.

To measure dereferencing operations, we use a `Pbox<i32>`. Dereferencing a persistent pointer involves address translation and memory indirection. The Rust compiler tends to keep the base and offsets for address translation in registers. As a result, the dereferencing operation performs less than 1 ns, for both read and write. However, writing for the first time requires logging which takes around 470 ns (`DerefMut`, the 1st time).

For memory allocation, Corundum uses a buddy system [51] which performs small

Table 3.4: Corundum's basic operation latency for durability and safety support measured on Intel's Optane DC and Battery-Backed DRAM, with 50K operations per test.

Operation	Optane DC Avg (ns)	DRAM Avg (ns)
`Deref`	0.9	1.0
`DerefMut` (the 1st time)	467	235
`DerefMut` (not the 1st time)	0.4	0.4
`Alloc` (8 B)	734	241
`Alloc` (256 B)	706	264
`Alloc` (4 kB)	1685	1907
`Dealloc` (8 B)	632	384
`Dealloc` (256 B)	598	249
`Dealloc` (4 kB)	675	248
`Pbox:AtomicInit` (8 B)	822	416
`Prc:AtomicInit` (8 B)	2089	1210
`ParcAtomicInit` (8 B)	3508	2304
`TxNop`	198	198
`DataLog` (8 B)	574	253
`DataLog` (1 kB)	1070	500
`DataLog` (4 kB)	2463	1510
`DropLog` (8 B)	29	28
`DropLog` (32 kB)	28	28
`Pbox::pclone` (8 B)	698	314
`Prc::pclone`	13	8
`Parc::pclone`	333	170
`Prc::downgrade`	6	6
`Parc::downgrade`	335	174
`Prc::PWeak:upgrade`	9	9
`Parc::PWeak::upgrade`	321	175
`Prc::demote`	42	43
`Parc::demote`	75	75
`Prc::VWeak::promote`	12	13
`Parc::VWeak::promote`	352	175

allocations by splitting large free blocks, and coalescing small, free adjacent blocks on deallocation to yield larger free blocks. Therefore, small free blocks are more available than large free blocks. For example, a free block of size 8192 B can be split into 1024 small free blocks of 8 B, or only 2 large block of size 4 kB. Table 3.4 confirms this fact. In contrast to allocation, freeing memory takes almost constant time, because merging is rare.

The failure-atomic instantiation operation (AtomicInit) allocates new memory and fills it with a given value atomically using low-level redo logging in the allocator. This operation is as fast as the allocation because the allocation is the only major part of it.

Corundum preallocates a per-thread journal object, so running an empty transaction (TxNop) does not write to the PM.

DataLog shows the latency of taking an undo log for a data with 8 B, 2 kB, and 32 kB sizes. It requires allocating memory and copying data to the log location. Therefore, the larger the data, the slower the operation, due to the allocation process. However, creating a DropLog which only keeps the information of the allocation, and takes constant time.

`Pbox::pclone()` creates a new instance of `Pbox` by allocating and copying data to a new location. Therefore, the latency of `Pbox::pclone` combines PM allocation and `memcpy()`. However, `Prc` and `Parc` do not allocate memory. They only update their reference counters transactionally. `Parc` takes a log every time it increments or decrements to provide crash-consistent atomic counters which explains its longer latency compared with `Prc`. The `downgrade()` and `upgrade()` functions also transactionally update the counters and we can use the same explanation as for `pclone()`. Although the `demote()` function use similar mechanism as `downgrade()` to create volatile weak pointer, they additionally update a reference list in `Prc`/`Parc` which makes them slightly slower. However, the latency of `promote()` is similar to `upgrade()` because they perform the same operation.

Table 3.5: Microbenchmarks. The first three are used to compare the performance of Corundum with PMDK. Wordcount measures Corundum's scalability with thread count.

BST	A transaction-free (in PMDK and Corundum) and failure-atomic implementation of a Binary Search Tree
KVStore	A simple Key-Value store data structure using hash map
B+Tree	An optimized, balanced B+Tree with 8-way fanout.
wordcount	Counts the occurrences of each word in a corpus of text using a hashmap and producer/consumer threads

Workloads

Table 3.5 summarizes the workloads we used to evaluate the performance of Corundum and its scalability. The first three applications are used to compare performance with PMDK, Atlas, Mnemosyne, and go-pmem. The PMDK version of BST, KVStore, and B+Tree are available in PMDK repository. We reimplemented them in Corundum and the other libraries using the same algorithms.

Results

Figure 3.1 shows the results of our experiments comparing Corundum's performance with PMDK, Atlas, Mnemosyne, and go-pmem. Corundum's performance is almost as fast as other libraries, and sometimes significantly faster.

The Wordcount benchmark measures scalability. It uses a single IO thread to pull text from input files and distributes it to a pool of worker threads that count words. It is embarrassingly parallel. Corundum provides a separate allocator and journal object for every thread to allow concurrency. Figure 3.2 shows that the performance scales with thread count, demonstrating that Corundum does not limit scalability.

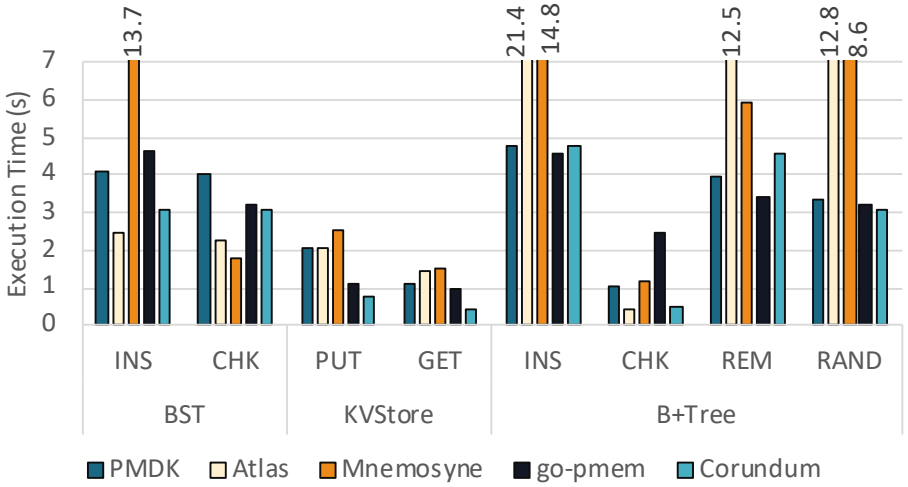


Figure 3.1: Performance comparison between Corundum, PMDK, Atlas, Mnemosyne, and go-pmem

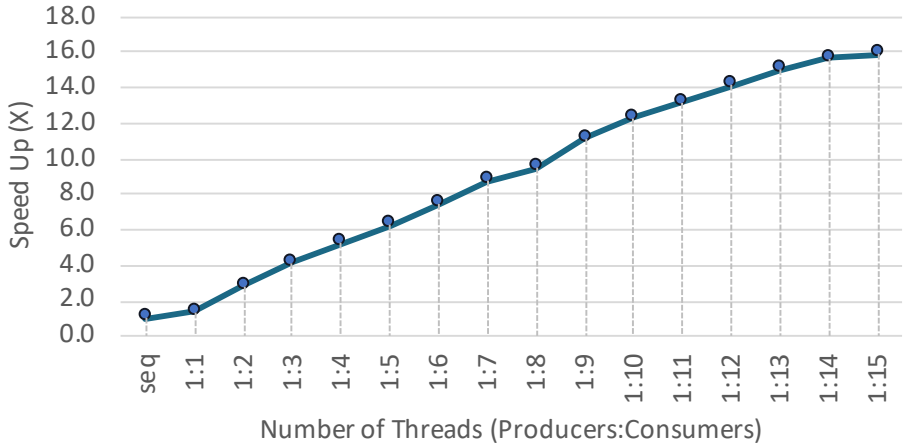


Figure 3.2: Corundum's scalable performance with regard to the number of threads. The baseline is running one producer and one consumer object sequentially (seq).

3.4 Related Work

Many projects have addressed the challenges of PM programming with software [75, 63, 82, 15, 11, 8, 58, 49, 16, 79, 59, 62, 2, 32] and/or hardware [70, 94, 45, 88]. Table 3.2 highlights the checks some of these systems provide. The systems not listed in table provide fewer checks (in some cases this is by design).

In response, several projects provide testing [61, 73, 60] and debugging tools [43, 71, 54] targeting PM systems. While useful, these tools cannot provide safety guarantees and they rely on the programmer using them reliably and providing tests with good coverage.

3.4.1 PM Programing Libraries

Among PM libraries, PMDK [75] is the most widely used. It provides dynamic checks to prevent inter-pool pointers in C and the C++ version provides static enforcement for atomicity, but otherwise the programmer is responsible for enforcing safety.

NV-Heaps [15] and Mnemosyne [82] rely on the C++ type system and a custom compiler, respectively, to avoid data races, atomicity violations, and pointers from PM into volatile memory. Both systems go to some pains help avoid bugs, but the weakness of the C/C++ type systems make the guarantees they provide easy to circumvent. NV-heaps addresses the problem of closed pools – by not allowing pools to close.

NVM Direct [8] adds extensions to C/C++ via a custom compiler that gives the programmer detailed control over logging. It is, by design, “dangerously flexible - just like C” [9] to enable as many manual optimizations as possible, so it relies heavily on the programmer to enforce safety properties. Corundum opts for safety over flexibility and does not require specific compiler support.

3.4.2 Orthogonal Persistence

The fundamentals of persistent programming have been studied for many years, and the notion of orthogonal persistence has been proposed as a guiding principle in PMEM software design [4]. The principle holds that the persistence abstraction should allow the creation and manipulation of data in an identical manner, regardless of its (non)persistence – that is that persistence should be *orthogonal* to other aspects of the language.

Despite the attractivensess of orthogonal persistence, Corundum and other recently-proposed systems are not orthogonally persistent. We made this design decision in Corundum because all three principles of orthogonal persistence face practical problems, especially with respect to performance, system complexity, and consistency.

First, "persistence independence" (i.e., using the same code for transient and persistent data) is slow and/or complicated, especially in low-level language like Rust. Persistent operations require different (and slower) instructions than operations on transient data, and an orthogonal system would require a runtime mechanism to choose which version to run or would need to always run the slow persistent code. Either choice will hurt performance.

Second, "data type orthogonality" (i.e., any data type can be persistent) leads to consistency problems since some types (e.g., network sockets or file handles) are inherently transient

The final principle of "persistence identification" (i.e., not expressing persistence in the type system) leads to complications in systems with multiple pools of persistent memory, since the question is not "transient or persistent" but "transient and, if not, in which pool". Without type information, it seems very challenging to statically prevent the creation of inter-pool pointers as Corundum does.

Furthermore, when orthogonal persistence was first formulated, persistence required a disk, so the performance cost of orthogonality was not an issue. For persistent memory, those costs would be prohibitive.

3.5 Summary

Corundum enforces PM safety invariants mostly using static checks. It, therefore, eliminates memory management, pointers safety, and logging bugs and avoids the attendant costs of testing, debugging, fixing, and recovering from them. It accomplishes this using Rust's type system to carefully control when the program can modify persistent and volatile state and when and where mutable references to persistent state can exist. Our experience shows that Corundum is relatively easy to use and our measurements show that Corundum's performance is comparable with (or better than) existing PM libraries.

Acknowledgements

This chapter contains material from "Corundum: Statically-Enforced Persistent Memory Safety", by Morteza Hoseinzadeh and Steven Swanson, which appeared in the Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 21), 2021. The dissertation author is the primary investigator and first author of this paper. Corundum was supported in part by Semiconductor Research Corporation (SRC) and by NFS award 1629395.

Chapter 4

Using Corundum Data Structures in C++

Persistent memory (PM) technology has brought the opportunity of accessing persistent data directly by using load and store instructions, improved memory system capacity, and a unified programming model for persistent and volatile programs. Many PM libraries have been introduced to operate on PM devices safely. Although they provide safe interfaces to interact with PM, not all of them verifiably address novel PM programming challenges. Of a particular class of PM libraries, some of them provide strong guarantees to prevent PM-related bugs. We presented Corundum in Chapter 3, a Rust-based PM library that falls into this class.

Corundum addresses the PM programming challenges by enforcing several PM safety rules using Rust's robust type system. Coding with Corundum restricts the programmer from doing certain operations that are against these rules. Corundum statically satisfies persistent pointer integration, data consistency, memory leak prevention, and thread-safety invariants through this restriction.

Although Corundum makes the whole persistent memory consistent considering the system failures, its rules are so restrictive that the programmer cannot apply many performance optimizations. This may reduce the programming flexibility and increase coding time compared with C++-based PM libraries.

This chapter presents Carbide, a multilingual PM framework that allows separately developing PM data structures in Corundum and using them in C++. This improves the flexibility in programming while maintaining strong PM safety guarantees. Multilingual programming essentially requires the programmer to consider strict directives to follow. Carbide's automatic code generation and static type checkers ensure that the safety invariants are satisfied in both languages. Our contributions in developing the Carbide framework is:

- Carbide preserves almost all Corundum's safety guarantees in C++.
- Carbide introduces a notion of the expanded lifetime of persistent objects with lifespans stretching between Rust's and C++'s scopes.
- Carbide statically checks the exporting procedure to prevent unsafe access to persistent data in Rust.
- Carbide performs static type checking while externally using the persistent objects from C++.
- Carbide transfers polymorphic types from a compiled library through a type parameter reduction and reparameterization technique.
- Carbide provides an option for automatically converting volatile data structures, including the C++ standard template library's container types, into persistence under specific criteria.

Our experiments show that Carbide vastly improves programmability at the cost of zero or small performance overhead when the same design is used. It improves performance by up to 60% when performance optimizations apply.

The rest of this chapter is organized as follows. Section 4.1 briefly discusses the background of PM libraries. Section 4.2 gives an overview of Carbide's design and our unique techniques to make the Corundum data structures externally available. Section 4.3 evaluates

Carbide in terms of performance, and Section 4.4 concludes this chapter. We additionally explain more details by getting through a complete example in Appendix A.

4.1 Background

Carbide strives to make safely implemented persistent types in Rust available in the flexible programming environment of C++ through an Application Programming Interface (API). To achieve this, Carbide faces a set of challenges in both PM programming and API designing. Carbide addresses these challenges using its unique multilingual design. This section provides some background information on PM programming libraries and common challenges in API programming for PM applications.

4.1.1 Persistent Memory Programming Libraries

A PM programming library/framework can be an application programming interface (API) to operate on persistent memory or a compiler pass transforming the code into a persistent version. Either of them follows a PM programming model to avoid intruding PM-related bugs that the type system cannot catch. Current software tools can be categorized as follows.

Basic PM Programming Frameworks offer a safe interface to access the PM, but they do not make any safety guarantees on the usage. Many frameworks such as PMDK [75], Atlas [11], go-pmem [28], Mnemosyne [82], and NV-Heaps [15], are examples of basic PM programming frameworks.

Code Transformation Frameworks usually analyze existing volatile program source files for adding PM compatibility to the program. Although they can theoretically transform any implementation into a PM program, practically, they need to limit the programming flexibility to verify the transformation. They include, but not limited to, AutoPersist [79], NVTraverse [26], Mirror [27], and Hippocrates [68].

Debugging/Bug-Fixing Tools check the post-production source code statically to find PM-related bugs. They usually use symbolic execution or other static analyzing techniques. However, in the case of large applications of enterprise products, they cannot test every execution path due to path explosion. NVL-C [19], Jaaru [30], and Agamotto [69], are some examples of this kind.

Testing Frameworks check the implementation dynamically after compilation and run several tests or injecting failures to find PM-related bugs. Since they do not provide completeness proof, they cannot find all dysfunctional operations in the code. PMTest [61] and XFDetector [60] are two well-known testing frameworks to evaluate PM programs.

Pre-Compilation Debugging Frameworks usually follow a formally verifiable model to evaluate the source code. These frameworks impose a lot of restrictive rules that limit programmability. For example, Corundum [37] takes benefits of Rust's type system to prevent almost all PM-related bugs statically. The result is a safe and efficient PM program that is verifiably free of those programming errors.

There is no doubt that each PM framework has some strengths and weaknesses. For example, PMDK offers tremendous flexibility as it does not impose any strict limitations on programming. At the same time, NVTraverse requires the data structure to provide a set of specific properties to perform the transformation correctly [26]. And, Corundum applies strict rules to restrict the programmer from unsafely accessing the PM.

4.1.2 API Programming for Persistent Memory

The diversity in the offerings and limitations of the current PM programming frameworks motivates us to look for a hybrid PM programming model that can be both flexible and verifiably safe. We suggest that even though the persistent data types should be implemented strictly safely, the other parts can be developed more freely. Hence, we propose that using Corundum [37] to implement persistent types and port them into C++ in the form of a compiled library can provide

safety and flexibility at the same time. However, implementing such a system may jeopardize the safety guarantees that Corundum provides because many of the techniques used in Corundum are based on Rust's features, and they are not available in C++. Below we list the main safety concerns of developing such a system:

Type Interoperability The types implemented in Rust have different memory layouts from C++; therfore, they are not fully operable. Rust allows the programmer to specify a "C" representation of the type explicitly, but this is only available for trivial types such as a struct with primitive fields. The persistent pointers and type wrappers are not specified so.

Polymorphism Persistent types are usually defined as container classes that take on many different forms. However, a polymorphic type is not portable through a compiled library as the type parameters cannot be derived in post-compilation time.

Memory Leaks C++ is not a memory-managed programming language; dynamic allocations may remain unreachable and unclaimed, leading to memory leaks.

Conflicted Lifetimes Although both C++ and Rust use an RAII model, the two programming languages have different scopes, and one object cannot be accurately managed if it travels between the two scopes.

4.2 Carbide

Carbide works as a bridge between Rust and C++ to add safe PM programming support into a flexible programming environment. To attain that, Carbide faces several challenges; some are common among PM programming systems, and the rest regards linking the two programming languages. Carbide addresses these challenges using code automation, unique language features, and static analyzers.

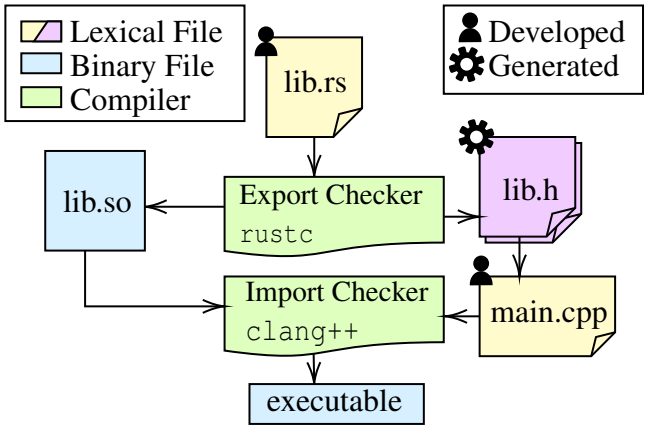


Figure 4.1: Carbide's System Description. The programmer defines persistent data types in *lib.rs* using Rust; Carbide generates the library and the header files; The programmer uses them in *main.cpp* using C++.

4.2.1 Design

Carbide generates a C++ Application Programming Interface (API) to allow safe interaction with the PM data types written in Corundum and packed in a dynamic library. Figure 4.1 depicts an overview of Carbide's tool flow. The programmer declares the persistent data types in Corundum (e.g., *lib.rs*) and runs the toolchain to generate a dynamic library and a set of C++ header files containing a safe interface to interact with the data types (e.g., *lib.h*). Then, the programmer writes the rest of the program in C++ and finally uses the C++ compiler to generate the executable. Aiming PM safety, Carbide applies rules while generating the interface that are checked by an Export Rule Checker in Rust and an Import Rule Checker in C++.

4.2.2 Design Goals and Challenges

In designing Carbide, we wish to preserve as many as possible Corundum's six design goals (see Section 3.2 in Chapter 3). Then we aim for solving four new problems that specifically arise when you try to use Corundum data types from C++:

Type Interoperability We want the types defined in Rust to be operable in C++ in the same way.

However, types in Rust have different memory layouts from C++ and cannot be ported as-is.

Polymorphism We want to support persistent polymorphic types. However, a polymorphic type is not portable as the type parameters cannot be derived post-compilation time.

Memory Leaks We want to prevent PM leaks in both languages. Since programming languages manage memory differently, memory leaks in one language become untraceable in the other language.

Lifetime Conflict We want to extend the Resource Acquisition is Initialization (RAII) model to allow objects to manage their resources across both languages by redefining the lifetime scopes.

To be successful, Carbide achieves Corundum's design goals in addition to addressing the new four API-design challenges by approaching the following goals:

- Designing a cross-language PM management system and pool type counterparts in C++.
- Automatically generating an interface in C++ to safely interact with data in Rust.
- Providing data consistency and pointer safety across both programming languages.
- Providing an export rules checker in Rust to enforce safety while designing the persistent data types.
- Providing an import rules checker in C++ to check the imported types during template type specializations.
- Defining a set of properties to make every C++ template type convertible into persistence.

We first discuss the four challenges and our solutions to address each. Then we describe Carbide's design goals in detail.

Type Interoperability

Carbide makes it possible to define a type in Rust and use it in C++ as if it was defined in C++. Since Rust and C++ do not layout memory similarly, Carbide cannot directly port every Corundum type to C++. Rather than that, Carbide exports the type's functionality through a Foreign Function Interface (FFI) and generates a corresponding type in C++ with the same functionality interface. Not to mention that Carbide can run this procedure only for portable persistent types. We define a portable type as a type with the following properties:

1. It applies Corundum's safety rules.
2. It is generic with respect to only one pool type.
3. Its public member functions are FFI-compatible.
4. It has a transactional constructor function.

For example, given a portable type `List` in Figure 4.2, step ①, Carbide generates an FFI associated with the Rust type in step ② to allow working with data externally. Then, it automatically generates a template class for it in step ③ that uses a `self` pointer as the receiver of external functions, just like the way that `this` pointer is used in a regular C++ class. We call it a *vessel* class because it is declared in C++ and is connected the Rust type.

Polymorphism

Polymorphic types, also known as Generics, are essential in object-oriented programming languages (OOP) like C++ and Rust. It is ubiquitous to define a generic container type that can take on various forms depending on the contained data type. Primarily in PM programming, persistent data structures are usually defined as a generic container type managing access to arbitrarily shaped, persistent data. While in Java-like languages, generic types have a fixed memory layout due to using references to inner data, C++ and many other system-level languages reshape template types into various memory layouts during compilation. Therefore, polymorphic

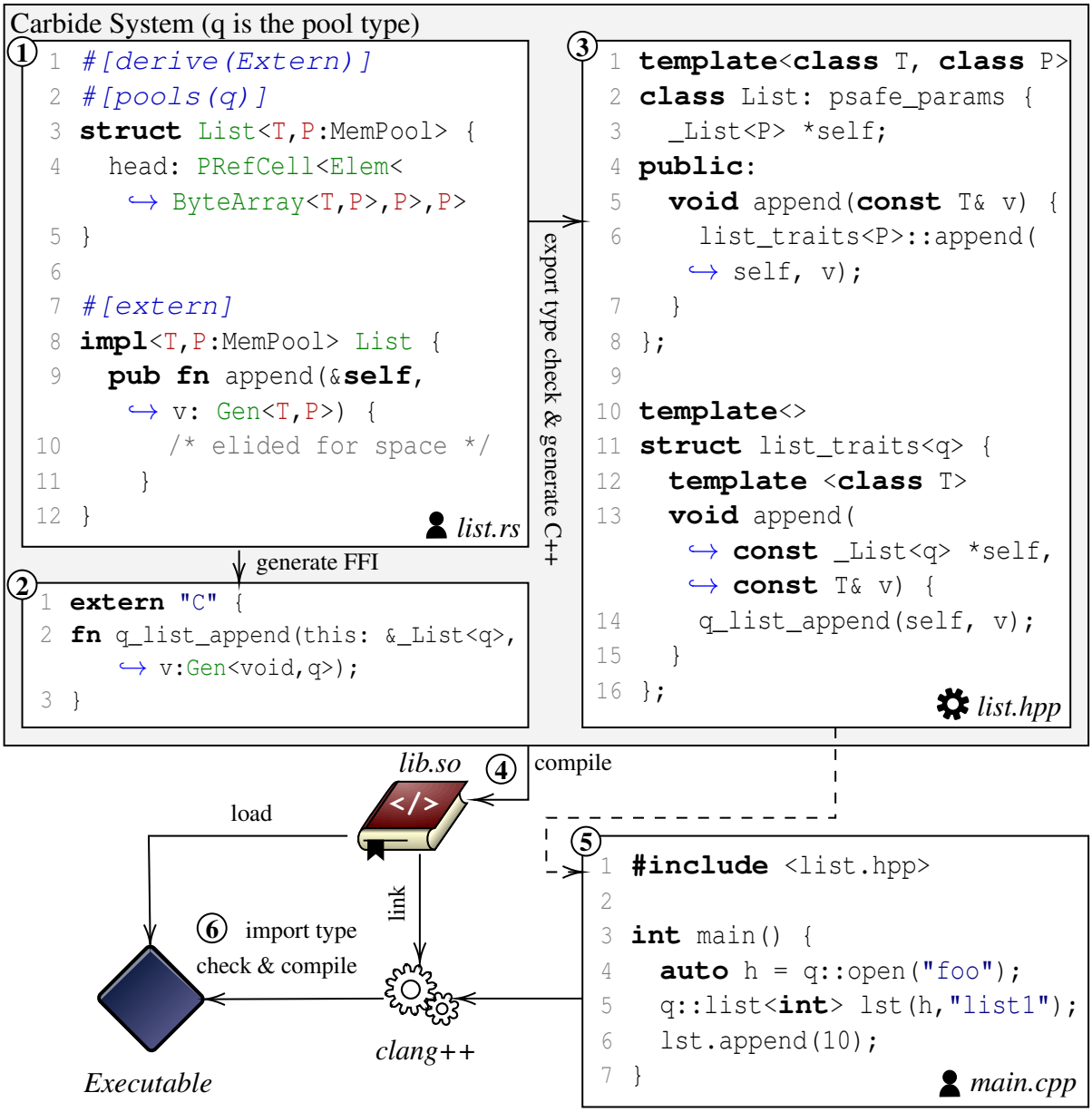


Figure 4.2: An example of using Carbide to define a `List` data structure in Rust and use it in C++ through 6 steps: (1) the programmer implements `List` using Corundum in Rust; (2) Carbide generates external interface to export `List`'s functionality; (3) Carbide reimplement the template class in C++; (4) the Rust compiler generates a dynamic library; (5) the programmer uses `List` in C++; and finally (6) *clang++* compiles *main.cpp* and links the library to the executable.

types are not portable via compiled binary libraries in those languages. To provide polymorphism through APIs, Carbide leverages uninterpreted *indirection*: a reference to a byte array that can be interpreted as any type, as shown in Figure 4.2: at Line 4 in step ①. However, some concerns arise with uninterpreted indirection:

Unaligned reinterpretation: the casting type has a different size from the byte array.

Unmatched reinterpretation: the source and target types of casting operation are different.

Access Violation: the casting is unaligned, and the object reaches beyond the allocation.

To address these problems, Carbide adds a phantom type to the byte array to involve Rust's type system so that it statically prevents incorrect reinterpretations of the type parameters: only a reference of the phantom type can be used for reinterpreting the byte array into an object, and vice versa.

Another issue is that Rust's type system cannot observe the type parameters specified in C++, so that it cannot apply the type checking rules. In order to limit the byte array reinterpretation into only one type in both C++ and Rust, we define a generic reference type (`Gen<T,P>`) implemented in both languages to engage both type systems (as shown in Figure 4.2: at Line 9 in Step 1). The byte array type and the reference type can be reciprocally interpreted as one another.

Memory Leaks

Carbide extends Corundum's memory management system, which is based on Rust's ownership mechanism; it also prevents the resources from leaking in C++ to prevent memory leaks. Memory leaks happen when a dynamically allocated piece of memory remains unclaimed and unreachable after use. When the unreachable allocations or garbage memory pile up, there will be no space left for new allocations eventually.

Many languages deal with memory leaks in different ways [34, 96], while some of them, like C and C++, relinquish garbage collection. Rust, thus Corundum, implements a reference

counting mechanism to prevent acyclic memory leaks. Considering the memory management systems in C++ and Rust, Carbide aims to manage persistent data in Rust while allowing C++ to reference them. As a result, it uses Corundum's memory management system and provides the same guarantees. To this end, Carbide enforces every persistent object, including those created in C++, to have an owner in Rust to manage the resources.

Lifetime Conflict

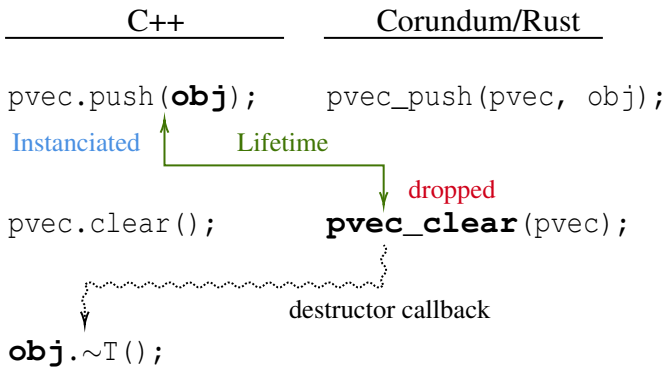
Carbide manages PM as a resource across Rust and C++ based on an extended RAII model that enforces every object to have an owner in Rust, even if created in C++. In the RAII model, the resource can be passed on to a new owner through a 'move' operation in contrast with a 'copy' operation, which creates a new allocation and copies the contents. However, the 'move' operation is unavailable in multilingual programming because the compiler cannot recognize the lifetime scopes in a pre-compiled library. This fact causes a conflict in RAII when an object is passed through an FFI: the scope of the object ends once it is passed, while it should be in scope on the other side, so if the object owns a resource, it may drop it when it goes through FFI. We divide this problem into two directions: *Rust to C++*, and *C++ to Rust*.

Carbide prevents moving an object to C++ by statically checking the input and output types of each FFI. Only objects that are both FFI-compatible and implement `Copy` trait can be passed by value. No smart pointer is copyable, but volatile raw pointers to persistent data are. Therefore, while the owner resides in Rust and manages the resource, a volatile pointer to the underlying data can access the resource from C++.

To ensure that a dynamic allocation in C++ has an owner in Rust, Carbide needs to evaluate three situations: when the allocation contains a root object; when the object is defined in C++; and when it is a persistent type defined in Rust but instantiated in C++. For the first situation, the root objects should never release the allocation as they are PM entry points.

In the second situation, the persistent object has a C++ type used to specialize a persistent

type in Rust. For example, pushing an arbitrary object into a persistent vector. Carbide limits moving objects from C++ to Rust into only one special template type which represents a generic cross-language reference (i.e., `Gen<T,P>`, where `T` is the C++ type, and `P` is the pool type). During instantiation, a `Gen` object first allocates PM and then constructs the object at the given location. For destruction, Carbide keeps destructor callback functions (in the form of a relative function pointer) to call them from Rust when its owner goes out of scope. The following graph shows the overlapped lifespan across both languages through this mechanism.



Finally, the third situation happens when a Rust type is instantiated in C++ and specializes in another type in Rust (e.g., a persistent vector of persistent vectors). In this case, Carbide calls the Rust type's destructor from C++ using an auto-generated FFI when the extended RAII determines that the object is no longer in scope.

4.2.3 PM Management and Pool Types

Carbide, similar to many other PM systems [75, 37, 15], offers multiple persistent regions, also known as *pool*s. Being based on Corundum, it uses pool types and relies on Rust's type system to avoid inter-pool pointers. Memory allocation and release operations are defined as a set of APIs for every pool type used for resource management by smart pointers through extended RAII. However, it is necessary to have the same design in C++. Carbide creates a counterpart type

for each pool in C++ and a persistent, intelligent pointer type that parameterizes the underlying data and the pool types. As a result, the C++ type system also can prevent inter-pool pointers.

As described in Section 4.2.2, a persistent object can be instantiated in C++ and owned by an object in Rust. Therefore, the pool types in C++ also provide APIs for memory allocation to allow object instantiation in C++. These APIs should be transactional to preserve memory consistency. To this end, pool types also provide Corundum's transactional interface. Then Carbide dynamically asserts a precondition that requires the existence of a running transaction at the beginning of the memory management routines.

4.2.4 PM Safety through Code Automation

Conductive to improve PM safety, Carbide generates source code to allow secure interaction between C++ and Corundum. It provides a set of procedural macros (e.g., Line 1 and 7 in step ①, Figure 4.2) to automatically generate source code for the following parts:

1. Pool type external interface
2. C++ types for interaction with pool types
3. Persistent type external interface
4. C++ types for interaction with persistent types

In addition to Corundum's pool type procedural macro, Carbide adds an external interface in the form of FFIs for a set of the routines in every pool type, including allocation, deallocation, and transaction.

Pool Type C++ Counterpart

Since the generated interface accesses low-level memory operations in Corundum, it is not safe enough to expose them to the user. Therefore, Carbide also provides a C++ class that

carefully imports the external interface into C++ for every pool type. For example, the external interface provides `txn_begin()`, `txn_commit()`, and `txn_abort()` which are not safe to use individually, while the pool type provides a higher-order function `txn()` that takes a lambda and internally calls each transaction subroutine accordingly.

Vessel Template Class

Carbide also generates FFIs for every specified member function of persistent data types. Since polymorphism is not available through ABI programming, Carbide enforces the user to wrap the template parameters in `ByteArray`, then specializes it for every pool type and generates an FFI specific to every pool. To resemble the data structure in C++, Carbide automatically generates a template class with the same interface, which internally calls the APIs. We call this type a *vessel* class because it connects C++ and Rust using the API. Since the type's routines are specialized for every pool, Carbide defines trait objects in C++ and uses it to generalize the pool type. Figure 4.2 shows an example of a generated vessel class in step ③ for a persistent object `List` defined in Rust, in step ①.

Root Object

Carbide allows multiple root objects each associated with a name (e.g., object `lst` at Line 5 in step ⑤, Figure 4.2, which is named "list1"). Since Corundum allows a single root object for every pool type – the root of the reachability tree – Carbide defines a persistent map from names to objects as a pool's root object. Each element in the map represents a named object. The type of a named object is a composition of all types defined in an algebraic data type. Listing 5 shows an example of a named root object type that has three variants: two user-defined types (i.e. `Stack` and `List`), and a custom named object for the root types defined in C++. In order to unify the generic types, Carbide specializes the variants with the owner pool type and `void` in Rust, and re-parametrizes them in C++ through vessel types, as shown in Figure 4.2,

```
pub enum NamedObject {
    Stack(Stack<void, Pool>),
    List(List<void, Pool>),
    // maybe more variants ...
    Custom(Named)
}
```

Listing 5: An example of generated named object type. The type is expanded.

step ③. Note that since the types are enforced to use byte arrays, the memory layout remains the same when the type is specialized by `void`. Therefore, it is possible to make them generic again on the other side.

4.2.5 Data Consistency

PM systems usually use a logging mechanism to protect data against system failures [75, 37, 11, 28, 82, 15], and Carbide is no exception. Carbide uses the Corundum logging mechanism and Rust's strong type system to avoid data inconsistency. However, this type system cannot apply safety rules on external usage of the exposed functionality. Additionally, C++ does not have an equally safe type system to protect data against system crashes, undefined pointer behavior, and data races in concurrent programming. We discuss these challenges in the rest of this section.

Crash Safety

Crash consistency is essential to PM programming. Carbide makes Corundum transactions available in C++ in the form of lambdas. Corundum defines a transaction as a code block with strictly protected boundaries that prevent sending unprotected types. Although it is possible in C++ to define a transaction as a lambda passing to a higher-order function, C++'s type system does not check the boundaries for unsafe crossings (i.e., entering a mutable volatile reference or exiting an orphaned persistent object).

Moreover, Corundum prevents unlogged updates to an object by limiting the smart pointers

to point to const data merely. Updates can be applied through interior mutability, meaning that a mutable reference is obtainable through a logging function of an immutable owner. The logging function is limited to transactions so that all updates are transactional.

Carbide makes it up by a combination of dynamic and static checks. We use a similar technique in C++ to provide interior mutability for const pointers. We also restrict the Rust APIs not to expose mutable references so that we can control updates through interior mutability (i.e., using `mut()` as the logging function).

To prevent orphaned objects from escaping the transaction's boundaries, Carbide explicitly deletes the move assignment and the `new` operator for the persistent types to make sure that the objects are neither dynamically instantiable nor movable. Since the memory management APIs are transactional, Carbide also dynamically prevents persistent objects from initializing outside a transaction.

Pointer Safety

One of the most prominent guarantees to make is pointer safety. Corundum statically checks the pointers to prevent the following unsafe situations [37]:

- Uncontrolled persistent-to-volatile pointers,
- Pointers between pools,
- Pointers to closed pools,
- Pointers to naturally volatile objects.

To apply the same rules in C++, Carbide introduces a new pointer type that is generic concerning both the data type and the pool type. This guarantees that pointers in one pool cannot be assigned to data in another pool. However, it is still possible to have a record type with multiple pointer fields pointing to different pools. In Corundum, this issue is handled by preventing instantiation of such types by applying strict rules on the transaction boundaries so that defining them is

practically useless. Since limiting the transaction boundaries is impossible in C++, Carbide statically analyzes the program to prevent defining such types.

Thread Safety

Concurrent programming requires extra care to prevent data races. Rust's borrowing mechanism prevents the coexistence of multiple mutable references to an object and mutable and immutable references simultaneously; thus, it prevents data races. Additionally, its ownership system guarantees that a single thread can own an object unless the object's type provides a safe shared interface (as in `Arc` pointers).

C++ does not make such guarantees, and objects can be accessed from multiple threads. This unsafe data sharing endangers data consistency. For example, multiple threads can write to the same object in multiple transactions.

Transactions should provide isolation to prevent data inconsistency in concurrent applications. One solution is to protect persistent types with a lock to ensure that there is only one thread borrowing the object at a time. Although there is a body of research suggesting different solutions to deal with data races in concurrent applications, Carbide does not enforce any thread-safety rules because they might be performance inefficient. That is because Corundum offers a robust multi-threading approach, and we can rely on the programmers to implement an optimized thread-safe data structure in Corundum as needed. Optionally, Carbide adds a reentrant mutex to every function in the generated vessel classes, if requested.

4.2.6 Exporting Rules

Besides the type system, we define new rules for safely exporting data types from Rust to C++. These rules are checked at compile time during code generation.

Generic Pools Types should be generic for the pool type. No other generalization is allowed in the definition of the external types. Alternatively, use `ByteArray` to store generic data.

Carbide implements the constructor and destructor C++ functions for every given pool type.

No Immediate Use of Generics Functions cannot forward generic types as they are not FFI-compatible due to unknown memory layouts during the compilation of the library.

No Generic References/Pointers Although the references and pointers have a fixed size, they cannot be correctly interpreted because they might not have the same memory layout in both languages. In exchange, Carbide allows using `Gen` which contains the layout information, so it is safe to transfer.

FFI-compatible Signature The function signatures should be FFI friendly to use as an API.

4.2.7 Importing Rules

Carbide maintains data consistency in C++. Some of the consistency rules can be applied through C++ type system. Others are checked by Carbide's *Import Checker*. First, we discuss the safety rules that the type system can provide, and then we list the rules checked by the import checker:

Avoid Non-Const Pointers/References No mutable pointer to a persistent object is allowed to avoid unrecoverable updates. The C++ implementation of `Gen` does not expose the mutable pointer to the object. For modifications, it returns a special `reference` type through `mut()` function, that can be treated like a non-const reference.

Transactional Updates Every update to PM should be transactional. Therefore, Carbide uses the same design as in Corundum to bound `mut()` functions inside a transaction. A pool type provides a `txn()` function that takes a lambda closure as the transactional operation. It provides a const pointer to the journal object that can be passed to `mut()` methods.

In addition to the C++ type system, we need to run a couple of more static type analyses to ensure that the data stored in PM is safe. We develop a compiler pass, the *Import Checker*, using LLVM to provide the missing type safety rules in C++:

No Native Pointers The data types should not have a pointer to the heap. The imported container types in C++ are marked as `psafe_params` (e.g., Line 2 in Step 3, Figure 4.2). If a type is marked so, the Import Checker looks into the specialization of the template type and checks the type parameters to be free of native pointers.

No Inter-Pool Objects Every data type field should reside in a single pool. This is to prevent inter-pool pointers as they are known to be unsafe [37]. The imported pool types are marked as `pool_type`. The analyzer visits the type, and when it encounters the first field or type parameter marked so, it remembers it. When it encounters another field of type parameter that is also marked as `pool_type` but points to a different pool, it stops and emits an error message.

Since C++ is too flexible, there is always a way to work around many of the rules. We consider the program is well-behaved if the following rules are abided by:

- Avoid PM pointer reinterpretation, albeit being possible due to some internal usage.
- Avoid const cast on PM pointers.
- Avoid data races as they are not guaranteed in C++.

In addition to these criteria, Carbide guarantees PM safety, given that the program only uses the provided interface to operate on PM. Directly using the exposed functions may result in an unrecoverable invalid state.

4.2.8 Persistence Specialization in C++

Although re-implementation of a small fraction of legacy code is inevitable, Carbide aims to limit the changes as much as possible. To this end, Carbide can make any C++ data type persistent if it is convertible to persistence. We say a C++ polymorphic type is *Carbide Compatible* if it is parametric with respect to the following items, and does not use any alternative in its member functions: the *allocator*, *pointer*, *reference*, and *iterator* types. Specializing such type with Carbide's version makes the type persistence safe based on Corundum's safety criteria. Therefore, we can store it in a persistent pool.

C++'s Standard Template Library (STL) provides many useful Carbide-compatible template container types and functions. We take advantage of this opportunity to make STL containers safe for PM. However, this specialization needs a minor consideration: Carbide's persistent pointer type does not expose mutability. We slightly modify the `std::pointer_traits` to include a new static function `get_mut()` which overwrites the const qualifier of the given pointer. Then, we specialize it for our persistent pointer type to create a log before returning the non-const pointer. To simplify this process, Carbide provides a type refactor class called `make_persistent` that converts the STL collections into their persistent versions. For example, `make_persistent<std::vector<T>, P>::type` reforms a `std::vector<T>` into `std::vector<T, carbide::allocator<T, P>>`.

4.3 Evaluation

We evaluate the performance of Carbide in both single-threaded and multi-threaded applications to measure the overhead of transporting safe data structures into a flexible programming environment.

Table 4.1: Microbenchmarks. The first three are used to compare the performance of Carbide with other PM systems. Wordcount measures Carbide's scalability with thread count.

BST	An implementation of a Binary Search Tree using failure-atomic updates
KVStore	A hashmap implementation of a Key-Value store data structure
B+Tree	A highly optimized, balanced B+Tree with 8-way fan-out.
wordcount	A multi-thread program that counts the occurrences of each word in a corpus of text using a hashmap

4.3.1 Evaluation Platform

We run our experiments on a platform equipped with a 40-core Intel® Xeon® Gold 6230 CPU running at 2.10 GHz. `clflushopt` and `clwb` instructions are available in our platform. In total, the system has 93 GB of DRAM, and 768 GB (single socket $\times$ 6 channel $\times$ 128 GB/DIMM) of Intel Optane DC DIMMs. Our machine runs Ubuntu 20.04 with Linux kernel version 5.4.0.

Carbide and Corundum use several unstable features of Rust, so we use Rust 1.57.0-nightly built with a 'release' profile. We compared Carbide with level '-O2' optimized releases of PMDK 1.8, Atlas, Mnemosyne, as well as optimized releases of go-pmem and Corundum. The import static checker is based on LLVM 12.0.1. The persistent memory is mounted using `Ext4-DAX`, and we use the local NUMA node in our experiments.

4.3.2 Workloads

We use the same set of microbenchmarks as in Corundum [37], summarized in Table 4.1 to compare Carbide with other PM libraries. BST, KVStore, and B+Tree are used to compare Carbide's performance with PMDK, Atlas, Mnemosyne, go-pmem, and Corundum. The PMDK versions of BST, KVStore, and B+Tree are publicly available in the PMDK repository. Using the same algorithms, we reimplemented these workloads in Carbide and the other libraries. WordCount is used to measure the scalability of Carbide and also its performance, considering

the flexibility that it provides to implement lock-free data structures.

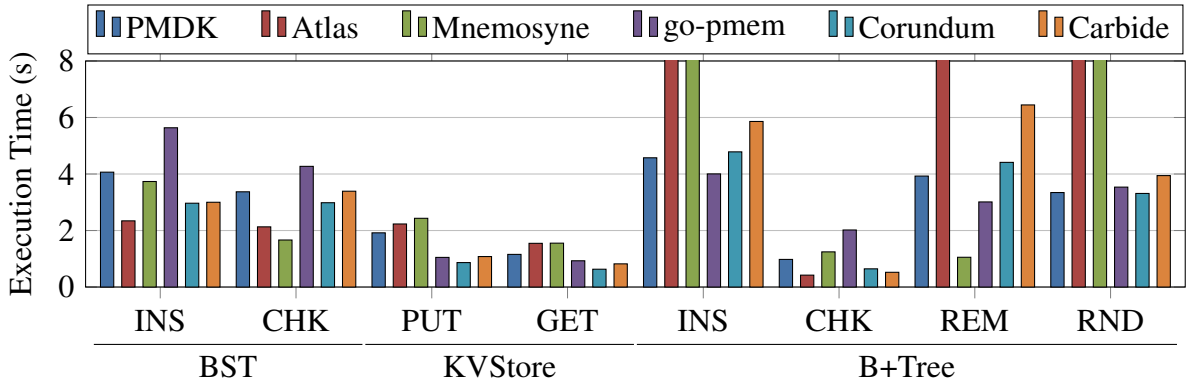
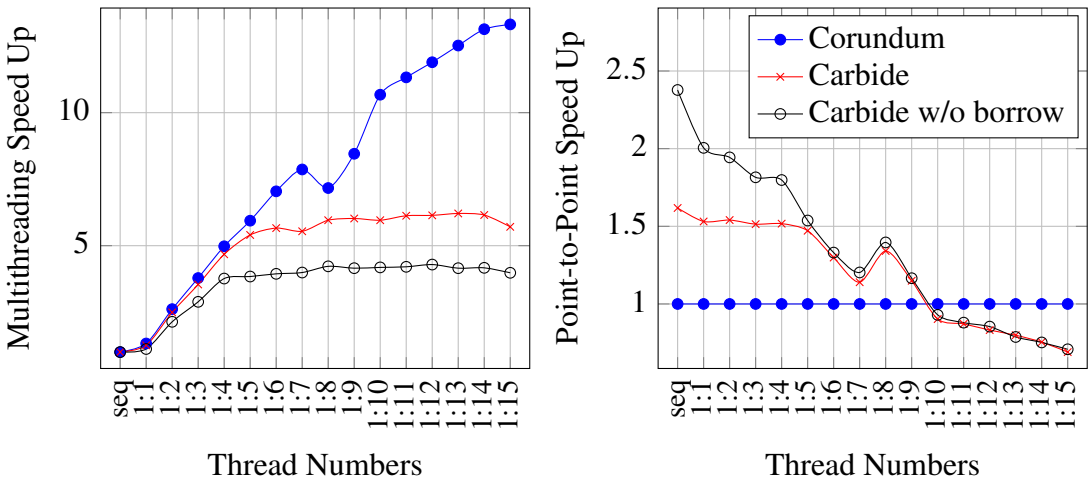


Figure 4.3: The performance results of comparing Carbide with other PM systems in terms of execution time.

4.3.3 Results

Figure 4.3 shows our experimental results comparing Carbide's performance with PMDK, Atlas, Mnemosyne, go-pmem, and Corundum. Carbide performance is comparable to other PM systems according. In BST and KVStore workloads that do not use unsafe coding and objects do not move very often, Carbide performs almost similar to Corundum. In the B+Tree workloads, multiple factors adversely affect Carbide's performance: 1) data copies around several times and each copy needs double allocations for the reference and the byte array, 2) similarly, dropping objects also calls deallocation twice, and 3) the byte array type has a drop function even if the underlying data type does not have any destructor (e.g., primitive types).

Figure 4.4a compares the scalability of Carbide and Corundum using the Wordcount benchmark. It uses a single IO thread to pull text from input files and distributes it to a pool of worker threads that count words. Since Corundum provides a separate allocator and journal object for every thread, it does not limit the scalability (i.e., it is embarrassingly parallel). However, we leverage Carbide's flexibility (e.g., we use unprotected native pointers for the local data in each tread) and implement a faster version of Wordcount.



(a) Scalability with respect to the number of threads. The baseline in each experiment is the sequential execution time.

(b) The point-to-point comparison between Corundum and Carbide on the execution time speedup.

Figure 4.4: The performance results of our experiments running Carbide in multi-thread modes, and comparing it with Corundum. The x axes show the number of producer and consumer threads separated by a colon.

Figure 4.4b compares the actual execution times when optimizations are used. It shows almost 1.6× faster in sequential execution and achieves the higher performance faster than Corundum (4 threads in Carbide has a performance equal to 10 threads in Corundum). As a result, the performance saturates sooner, and Carbide does not scale after 10 consumer threads. At this point, the C++ implementation of word counting process becomes the performance bottleneck. Notice that Corundum still scales after 10 threads because the word counting process is faster in Rust.

We also measured Carbide's performance in Wordcount without dynamic borrow checking [37] as it is not available in C++ and does not accurately evaluate the borrowers in practice. The results show that it adds 25% speedup when disabled.

4.4 Summary

Programming for PM under strict safety rules reduces the flexibility, although it does not limit the programmability. Some parts of the program that the memory safety rules are not required should provide more flexibility to the programmer. Carbide provides this opportunity to separate the persistent data structure development from the rest of the program. It provides strong safety guarantees for the persistent data structures while the rest of the program can be flexibly optimized for volatile access to the persistent data. Our experimental results show that Carbide may incur a small, or none, performance overhead when using the same design as in the other workloads, but perform faster when optimization techniques are used with flexible coding.

Chapter 5

Extensive PM Storage System with Multi-Tiering

Emerging fast, byte-addressable persistent memories (PM), such as battery-backed NVDIMMs [65] and 3D-XPoint [64] promise to increase the performance of storage systems dramatically. These non-volatile memory technologies offer vastly higher throughput and lower latency compared with traditional block-based storage devices. Therefore, PM has become a suitable technology to engineer super-fast storage systems.

Researchers have proposed several file systems [17, 20, 21, 83, 86] on PM. These file systems leverage the direct access (DAX) feature of persistent memory to bypass the page cache layer and provide user applications direct access to file data.

The high performance of persistent memory comes at a high cost. The average price per byte of persistent memory is higher than SSD. Also, PM cannot scale as large capacities as SSDs and HDDs can. So, cost-sensitive applications or workloads requiring larger capacities than what PM can provide would benefit from a storage system that can leverage the strengths of both technologies: PM for speed and disks for capacity.

Tiering is a solution to this dilemma. Tiered file systems manage a hierarchy of heteroge-

neous storage devices and place data in the storage device that matches the data's performance requirements and the application's future access patterns.

Using PM poses new challenges to the data placement policy of tiered file systems. Existing tiered storage systems (such as FlashStore [18] and Nitro [56]) are based on disks (SSDs or HDDs) that provide the same block-based interface, and while SSDs are faster than hard disks, both achieve better performance with larger, sequential writes, and neither can approach the latency of DRAM for reads nor writes.

PM supports small (e.g., 8-byte) writes and offers DRAM-like latency for reads and write latency within a small factor of DRAM. Ziggurat decides where to place data and metadata more complex: The system must decide where to write data (DRAM or PM) initially, how to divide PM between metadata, freshly written data, and data that the application is likely to read.

The first challenge is how to exploit the high bandwidth and low latency of PM fully. Using PM introduces a much more efficient way to persist data than disk-based storage systems. File systems can persist synchronous writes simply by writing them to PM, which bypasses the page cache layer and removes the high latency of disk accesses from the critical path. Nevertheless, a DRAM page cache still has higher throughput and lower latency than PM, making it competitive to perform asynchronous writes to the disk tiers.

The second challenge is reconciling PM's random access performance with the sequential accesses that disks and SSDs favor. Bandwidth and latency are no longer the only differences between different storage tiers in a tiered file system with PM and disks. Compared with disks, the gap between the sequential and random performance of PM is much smaller, making it capable of absorbing random writes. Simultaneously, the file system should leverage PM to maximize the sequentiality of writes and reads to and from the disk.

We propose Ziggurat, a tiered file system that spans PM and disks. Ziggurat exploits the benefits of PM through intelligent data placement during file writes and data migration. Ziggurat includes two placement predictors that analyze the file write sequences and predict

whether the incoming writes are both large and stable and whether updates to the file are likely to be synchronous. Ziggurat then steers the incoming writes to the most suitable tier based on the prediction: writes to synchronously-updated files go to the PM tier to minimize the synchronization overhead. Small, random writes also go to the PM tier to entirely avoid random writes to disk. The remaining large sequential writes to asynchronously-updated files go to disk.

We implement an efficient migration mechanism in Ziggurat to make room in PM for incoming file writes and accelerate reads to frequently accessed data. We first profile the temperature of file data and select the coldest file data blocks to migrate. During migration, Ziggurat coalesces adjacent data blocks and migrates them in large chunks to disk. Ziggurat also adjusts the migration policy according to the application access patterns.

The contributions presented in this chapter include:

- We describe a synchronicity predictor to efficiently predict whether an application is likely to block waiting a write to complete.
- We describe a write size predictor to predict whether the writes to a file are both large and stable.
- We describe a migration mechanism that utilizes the characteristics of different storage devices to perform efficient migrations.
- We design an adaptive migration policy that can fit different access patterns of user applications.
- We implement and evaluate Ziggurat to demonstrate the effectiveness of the predictors and the migration mechanism.

We evaluate Ziggurat using a collection of micro and macro-benchmarks. We find that Ziggurat can obtain near-PM performance on many workloads even with little PM. With a small amount of PM and a large SSD, Ziggurat achieves up to 38.9× and 46.5× throughput

improvement compared with EXT4 and XFS running on SSD alone, respectively. As the amount of PM grows, Ziggurat's performance improves until it nearly matches the performance of a PM-only file system.

The remainder of this chapter is organized as follows. Section 5.1 gives a quick backgroud on PM-based storage systems. Section 5.2 presents a design overview of the Ziggurat file system. We discuss the placement policy and the migration mechanism of Ziggurat in Section 5.4 and Section 5.3, respectively. Section 5.5 evaluates Ziggurat, and Section 5.6 shows some related work. Finally, we present our conclusions in Section 5.7.

5.1 NOVA

Our tiering file system, Ziggurat, targets emerging non-volatile memory technologies and conventional block-based storage devices (e.g., SSDs or HDDs). Ziggurat is based on the NOVA file system. This section provides a brief background on the NOVA file system.

NOVA [86] is a PM file system designed to maximize performance on hybrid memory systems while providing strong consistency guarantees. Below, we discuss the file structure and scalability aspects of NOVA's most relevant design to Ziggurat.

NOVA maintains a separate log for each inode to fully exploit PM's high performance and massive concurrency. NOVA also maintains radix trees in DRAM that map file offsets to PM locations. Figure 5.1 illustrates the relationship between the inode, its log, and its data pages. For file writes, NOVA creates *write entries* (the log entries for data updates) in the inode log. Each write entry holds a pointer to the newly written data pages, as well as its modification time (`mtime`). After NOVA creates a write entry, it updates the tail of the inode log in PM, along with the in-DRAM radix tree.

NOVA uses per-CPU allocators for PM space and per-CPU journals for managing complex metadata updates. This enables parallel block allocation and avoids contention in journaling. In

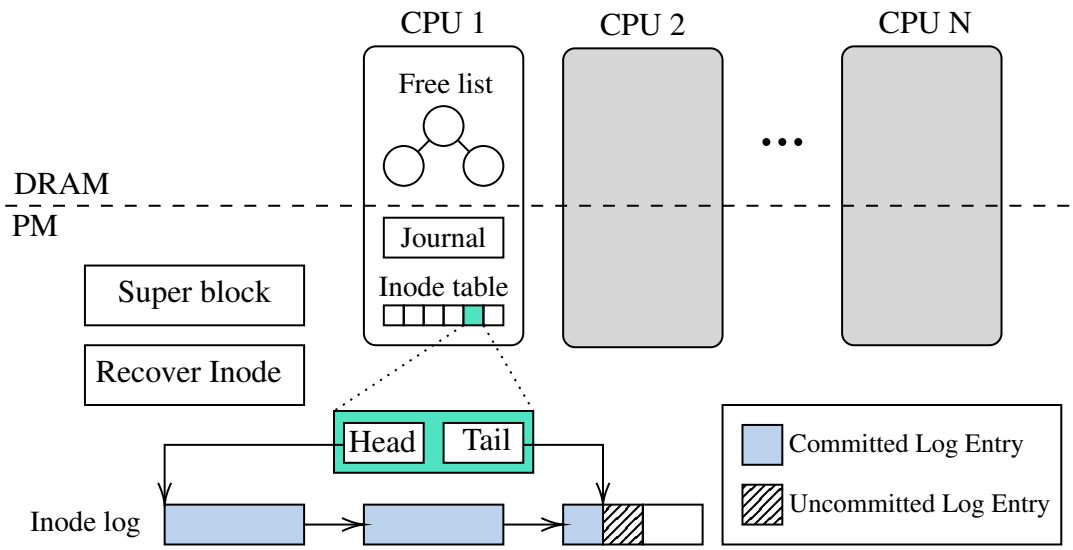


Figure 5.1: NOVA data structure. Each processor has a free list, a journal, and an inode table to guarantee scalability. An inode consists of a linked list of 4 KB log pages; the tail pointer points to the last committed log entry.

addition, NOVA has per-CPU inode tables to ensure good scalability.

5.2 Ziggurat

Ziggurat is a tiered file system that spans across PM and disks (hard or solid-state). We design Ziggurat to fully utilize the strengths of PM and disks and to offer high file performance for a wide range of access patterns.

Three design principles drive the decisions we made in designing Ziggurat. First, Ziggurat should be *fast-first*. It should use disks to expand the capacity of PM rather than using PM to improve the performance of disks as some previous systems [25, 35] have done. Second, Ziggurat strives to be *frugal* by placing and moving data to avoid wasting scarce resources (e.g., PM capacity or disk bandwidth). Third, Ziggurat should be *predictive* by dynamically learning the access patterns of a given workload and adapting its data placement decisions to match.

These principles influence all aspects of Ziggurat's design. For instance, being fast-first means, in the typical case, file writes go to PM. However, Ziggurat makes an exception if it

predicts that steering a particular write in PM would not help application performance (e.g., if the write is large and asynchronous).

Alternatively, if the writes are small and synchronous (e.g., to a log file), Ziggurat sends them to PM initially, detect when the log entries have "cooled", and then aggregate those many small writes into larger, sequential writes to disk.

Ziggurat uses two mechanisms to implement these design principles. The first is a placement policy driven by a pair of predictors that measure and analyze past file access behavior to predict future behavior. The second is an efficient migration mechanism that moves data between tiers to optimize PM performance and disk bandwidth. The migration system relies on a simple but effective mechanism to identify cold data to move from PM to disk.

We describe Ziggurat in the context of a simple two-tiered system comprising PM and an SSD, but Ziggurat can use any block device as the "lower" tier. Ziggurat can also handle more than one block device tier by migrating data blocks across different tiers.

5.2.1 Design Decisions

We made the following design decisions in Ziggurat to achieve our goals.

Send writes to the most suitable tier. Although PM is the fastest tier in Ziggurat, file writes should not always go to PM. PM is best-suited for minor updates (since small writes to disk are slow) and synchronous writes (PM has higher bandwidth and lower latency). However, targeting disk is faster for larger asynchronous writes since Ziggurat can buffer the data in DRAM more quickly than it can write to PM, and the write to disk can occur in the background. Ziggurat uses its *synchronicity predictor* to analyze the sequence of writes to each file and predict whether future accesses are likely to be synchronous (i.e., whether the application calls `fsync` shortly).

Only migrate cold data in cold files. During migration, Ziggurat targets the cold portions of cold files. Hot files and hot data in unevenly-accessed files remain in the faster tier. When the usage of the fast tier is above a threshold, Ziggurat selects files with the earliest average

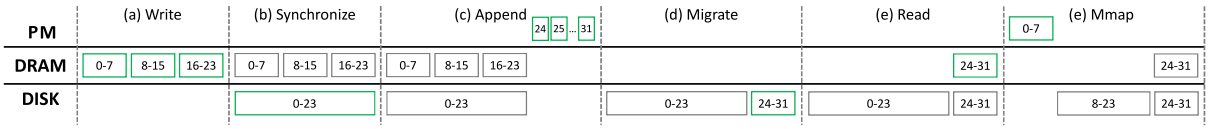


Figure 5.2: File operations. Ziggurat utilizes different storage tiers to efficiently handle I/O requests such as write, synchronize, append, migrate, read, and mmap.

modification time to migrate (Section 5.3.1). Within each file, Ziggurat migrates blocks that are older than average. Unless the whole file is cold (i.e., its modification time is not recent), in which case we migrate the whole file.

High PM space utilization. Ziggurat fully utilizes PM space to improve performance. Ziggurat uses PM to absorb synchronous writes. Ziggurat uses a dynamic migration threshold for PM based on the read-write pattern of applications, so it makes the most of PM to handle file reads and writes efficiently. We also implement reverse migration (Section 5.3.2) to migrate data from disk to PM when running read-dominated workloads.

Migrate file data in groups. To maximize the write bandwidth of disks, Ziggurat performs migration to disks as sequentially as possible. The placement policy ensures that most small, random writes go to PM. However, migrating these small write entries to disks suffers from disks' poor random access performance. To make migration efficient, Ziggurat coalesces adjacent file data into large chunks for migration to exploit sequential disk bandwidth (Section 5.3.3).

High scalability. Ziggurat extends NOVA's per-CPU storage space allocators to include all the storage tiers. It also uses per-CPU migration and page cache writeback threads to improve scalability.

5.2.2 File Operations

Figure 5.2 illustrates how Ziggurat handles operations on files (write, synchronize, append, migrate, read, and mmap) that span multiple tiers.

Write The application initializes the first 24 blocks of the file with three sequential writes in

(a). Ziggurat first checks the results from the synchronicity predictor and the write size predictor (Section 5.4) to decide which tier should receive the new data. The three writes are significant in the example, and Ziggurat predicts that the accesses are asynchronous, so Ziggurat steers these writes to disk. It writes the data to the page cache in DRAM and then asynchronously writes them to disk.

Synchronize The application calls `fsync` in (b). Ziggurat traverses the write log entries of the file and writes back the dirty data pages in the DRAM page cache. The writeback threads merge all adjacent dirty data pages to perform large sequential writes to the disk. If the file data were in PM, `fsync` would be a no-op.

Append After the `fsync`, the application performs eight synchronous writes to add eight blocks to the end of the file in (c). The placement predictor recognizes the pattern of small synchronous writes, and Ziggurat steers the writes to PM.

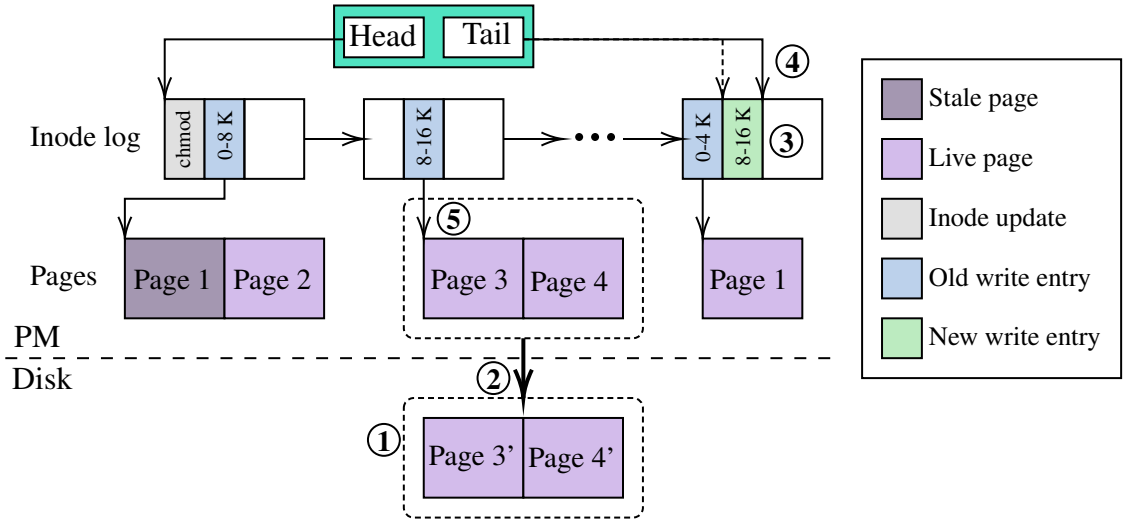
Migrate When the file becomes cold in (d), Ziggurat evicts the first 24 data pages from DRAM and migrates the last eight data blocks from PM to disk using group migration (Section 5.3.3).

Read The user application reads the last eight data blocks in (e). Ziggurat fetches them from disk to DRAM page cache.

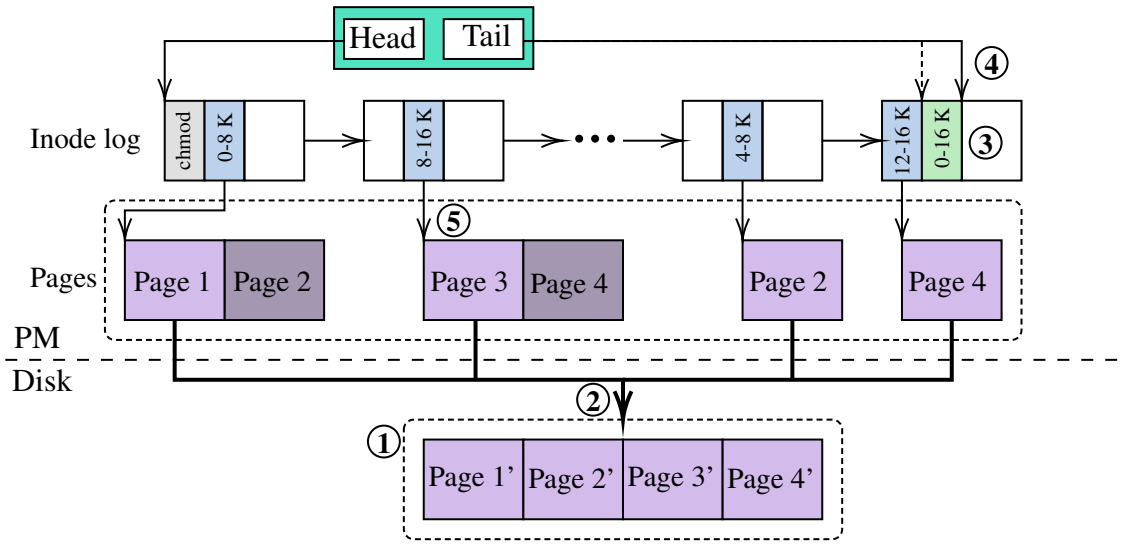
Memory map The user application finally issues a `mmap` request to the head of the file in (f). Ziggurat uses reverse migration to bring the data into PM and then maps the pages into the application's address space.

5.3 Migration Mechanism

The purpose of migration is to make room in PM for incoming file writes and speed up reads to frequently accessed data. We use *basic migration* to migrate data from disk to PM



(a) The file structure of Ziggurat and basic migration



(b) Group migration

Figure 5.3: Migration mechanism of Ziggurat. Ziggurat migrates file data between tiers using its basic migration and group migration mechanisms. The blue arrows indicate data movement, while the black ones indicate pointers.

to utilize PM space when running read-dominated workloads fully. We use *group migration* to migrate data from PM to disk by coalescing adjacent data blocks to achieve high migration efficiency and free up space for future writes. Ziggurat can achieve near-PM performance for most accesses as long as the migration mechanism is efficient enough.

In this section, we first describe how Ziggurat identifies good targets for migration. Then, we illustrate how it migrates data efficiently to maximize the disk bandwidth with basic migration and group migration. Finally, we show how to migrate file logs efficiently.

5.3.1 Migration Profiler

Ziggurat uses a migration profiler to identify cold data to migrate from PM to disk.

Implementation. Ziggurat first identifies the cold files to migrate. Ziggurat profiles the temperature of each file by maintaining *cold lists*, the per-CPU lists of files on each storage tier, sorted by the average modification time(`amtime`) computed across all the blocks in the file. The per-CPU cold lists correspond to per-CPU migration threads, which migrate files from one tier to another. Ziggurat updates the cold list whenever it modifies a file. To identify the coldest blocks within a cold file, Ziggurat tracks the `mtime` for each block in the file.

To migrate data, Ziggurat pops the coldest file from a cold list. If the `mtime` of the popped file is not recent (more than 30 seconds ago), then Ziggurat treats the whole file as cold and migrates all of it. Otherwise, the modification time of the file's block varies, and Ziggurat migrates the write entries with `mtime` earlier than the `amtime` of the file. Hence, the cold part of the file migrates to a lower tier, and the hot part of the file stays in the original tier.

Deciding when to migrate. Most existing tiered storage systems (such as [10, 53]) use a fixed utilization threshold to decide when to migrate data to lower tiers. However, a higher threshold is not suitable for write-dominated workloads since intensive file writes deviate from the available space in persistent memory. In this case, the file writes must either stall before the migration threads clean up enough space in PM or write to disk. On the other hand, a lower threshold is not desirable for read-dominated workloads since reads have to load more blocks from disks instead of PM. We implement a dynamic threshold for PM in Ziggurat based on the overall read-write ratio of the file system. The migration threshold rises from 50% to 90% as the read-write ratio of the system changes.

5.3.2 Basic Migration

The goal of basic migration is to migrate the coldest data in Ziggurat to disk. When the usage of the upper tier is above the threshold, a per-CPU migration thread migrates the coldest data in a cold file to disk. The migration process repeats until the usage of the upper tier is below the threshold again.

The granularity of migration is a write entry. During migration, we traverse the in-DRAM radix tree to locate every valid write entry in the file and migrate the write entries with `mtime` earlier than the `amtime` of the file.

Figure 5.3a illustrates the basic procedures of how Ziggurat migrates a write entry from PM to disk. The first step is to allocate contiguous space on the disk to hold the migrated data. Ziggurat copies the data from PM to disk. Then, it appends a new write entry to the inode log with the new location of the migrated data blocks. After that, it updates the log tail in PM and the radix tree in DRAM. Finally, Ziggurat frees the old blocks of PM.

Ziggurat uses locks in the granularity of a write entry instead of an entire file to improve scalability. Ziggurat locks write entries during migration, but other parts of the file remain available for reading. The migration does not block file writes. If any foreground file I/O request tries to acquire the inode lock, the migration thread stops migrating the current file and releases the lock.

If a write entry migrates to a disk when the DRAM page cache usage is low (i.e., below 50%), Ziggurat copies the pages in the DRAM page cache to accelerate future reads. Writes also benefit from this since unaligned writes have to read the partial blocks from their neighbor write entries to fill the data blocks.

Ziggurat implements reverse migration, which migrates file data from disks to PM using basic migration. Write entries are migrated successively without grouping since PM can handle sequential and random writes efficiently. File `mmap` uses reverse migration to enable direct access to persistent data. Reverse migration also optimizes the performance of read-dominated workloads

when PM usage is low since the performance depends on memory size. If Ziggurat can only migrate data from a faster tier to a slower one, then the precious available space of PM stays idle when running read-dominated workloads. Meanwhile, the data on disks contend for a limited DRAM. Reverse migration makes full use of PM in such a scenario.

5.3.3 Group Migration

Group migration avoids fine-grain migration to improve efficiency and maximize sequential bandwidth to disks. Ziggurat tends to fill PM with small writes due to its data placement policy. Migrating them from PM to disk with basic migration is inefficient because it incurs disk high random access latency.

Group migration coalesces small write entries in PM into large sequential ones to disk. There are four benefits: (1) It merges small random writes into large sequential writes, which improves migration efficiency. (2) If the migrated data is reread, loading continuous blocks is much faster than loading scattered blocks around the disk. (3) By merging write entries, the log itself becomes smaller, reducing metadata access overheads. (4) It moderates disk fragmentation caused by log-structured writes by mimicking garbage collection.

As illustrated in Figure 5.3b, the steps of group migration are similar to migrating a write entry. In step 1, we allocate large chunks of data blocks in the lower tier. In step 2, we copy multiple pages to the lower tier with a single sequential write. After that, we append the log entry and update the inode log tail, which commits the group migration. The stale pages and logs are freed afterward. Ideally, the *group migration size* (the granularity of group migration) should be set close to the future I/O size so that applications can fetch file data with one sequential read from disk. In addition, it should not exceed the CPU cache size to maximize the performance of loading the write entries from disks.

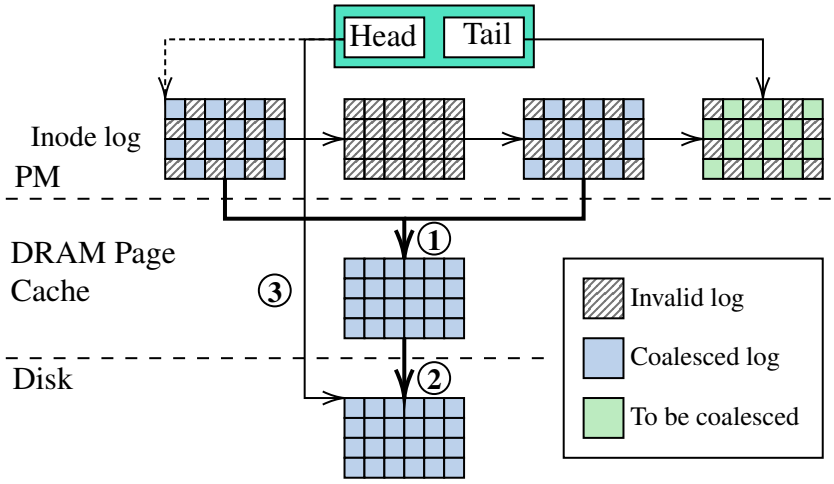


Figure 5.4: Log migration in Ziggurat. Ziggurat compacts logs as it moves them from PM to disk.

5.3.4 File Log Migration

Ziggurat migrates file logs in addition to data when PM utilization is too high, freeing up space for hot data and metadata. Ziggurat periodically scans the cold lists and initiates log migration on cold files. Figure 5.4 illustrates how log migration is performed. Ziggurat copies live log entries from PM into the page cache. The log entries are compacted into new log pages during coping. Then, it writes the new log pages back to disk and updates the inode metadata cache in DRAM to point to the new log. After that, Ziggurat atomically replaces the old log with the new one and reclaims the old log.

5.4 Placement Policy

Ziggurat steers synchronous or small writes to PM, but it steers asynchronous, large writes to disk, because writing to the DRAM page cache is faster than writing to PM, and Ziggurat can write to the disk in the background. It uses two predictors to distinguish these two types of writes.

Synchronicity predictor The synchronicity predictor predicts whether the application is likely to call `fsync` on the file shortly. The synchronicity predictor counts the number of data

blocks written to the file between two calls to `fsync`. If the number is less than a threshold (e.g., 1024 in our experiments), the predictor classifies it as a synchronously updated file. The predictor treats file write operations opened with `O_SYNC` as synchronous as well.

Write size predictor The write size predictor ensures that a write is large enough to exploit disk bandwidth effectively and that the future writes within the same address range are also likely to be large. The second condition is critical. For example, if the application initializes a file with large I/Os, and then performs many small I/Os, these small new write entries read and invalidate discrete blocks, increasing fragmentation and leading to many random disk accesses to service future reads.

Ziggurat's write size predictor keeps a counter in each write entry to indicate whether the write size is large and stable. When Ziggurat rewrites an old write entry, it first checks whether the write size is big enough to cover at least half the area taken up by the original log entry. If so, Ziggurat transfers the counter value of the old write entry to the new one and increases it by one. Otherwise, it resets the counter to zero. If the number is larger than four (a tunable parameter), Ziggurat classifies the write as "large". Writes that are both large and asynchronous go to disk.

5.5 Evaluation

In this section, we discuss our experiments and observations of running different workloads on Ziggurat and measuring the performance gain that it offers.

5.5.1 Experimental Setup

Ziggurat is implemented on Linux 4.13. We used NOVA as the codebase to build Ziggurat and added around 8.6k lines of code. To evaluate the performance of Ziggurat, we run micro-benchmarks and macro-benchmarks on a dual-socket Intel Xeon E5 server. Each processor runs at

Table 5.1: NUMA latency and bandwidth of our platform. We use the increased latency and reduced bandwidth of the remote NUMA node to emulate the lower performance of PM compared to DRAM.

Node	0	1
0	76.6	133.7
1	134.2	75.5

(a) NUMA latency (ns)

Node	0	1
0	52213.9	25505.9
1	25487.3	52111.8

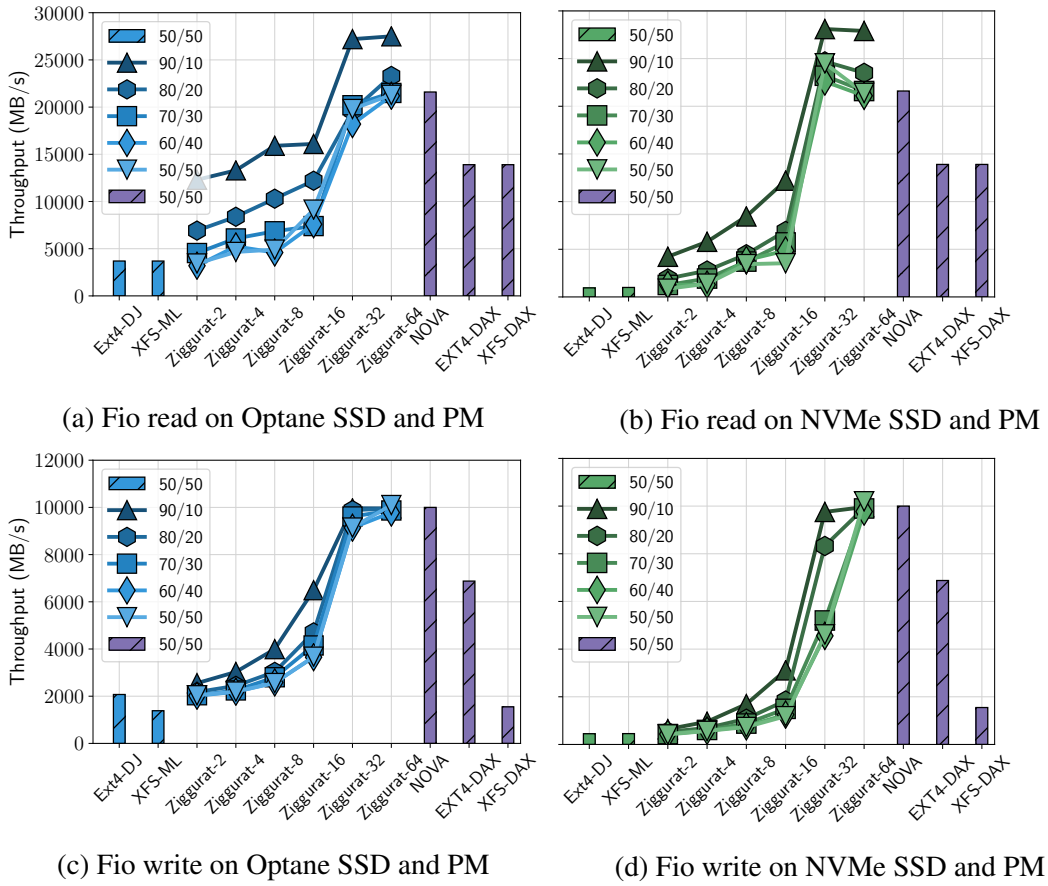
(b) NUMA bandwidth (MB/s)

2.2GHz, has 10 physical cores, and is equipped with 25 MB of L3 cache and 128 GB of DRAM. The server also has a 400 GB Intel DC P3600 NVMe SSD and a 187 GB Intel DC P4800X Optane SSD.

As persistent memory devices are not yet available, we emulate the latency and bandwidth of PM with the NUMA effect on DRAM. There are two NUMA nodes in our platform. During the experiments, the entire address space of NUMA node 1 is used for PM emulation. All applications are pinned to run on the processors and memory of NUMA node 0. Table 5.1 shows the DRAM latency of our experimental platform by Intel Memory Latency Checker [39].

We compare Ziggurat with different types of file systems. For PM-based file systems, we compare Ziggurat with NOVA [86], Strata [53] (PM only), and the DAX-based file systems on Linux: EXT4-DAX and XFS-DAX. For disk-based file systems, we compare Ziggurat with EXT4 in the data journaling mode (-DJ) and XFS in the metadata logging mode (-ML). Both EXT4-DJ and XFS-ML provide data atomicity, like Ziggurat. For EXT4-DJ, the journals are kept in a 2 GB journaling block device (JBD) on PM. For XFS-ML, the metadata logging device is 2 GB of PM. We limit the capacity of the DRAM page cache to 10 GB.

For tiered file systems, we only make the comparison among Ziggurat with different configurations. Strata is the only currently available tiered file system that spans PM and disks to the best of our knowledge. However, the publicly available version of Strata only supports a few applications and has trouble running workloads with dataset sizes larger than PM size and multi-threaded applications.



(a) Fio read on Optane SSD and PM

(b) Fio read on NVMe SSD and PM

(c) Fio write on Optane SSD and PM

(d) Fio write on NVMe SSD and PM

Figure 5.5: Fio performance. Each workload performs 4 KB reads/writes to a hybrid file system backed by PM and SSD (EXT4-DJ, XFS-ML, and Ziggurat) or a PM-only file system (NOVA, EXT4-DAX, and XFS-DAX).

We vary the PM capacity available to Ziggurat to show how performance changes with different storage configurations. The dataset size of each workload is smaller than 64 GB. The variation starts with Ziggurat-2 (i.e., Ziggurat with 2 GB of PM). In this case, most of the data must reside on the disk, forcing Ziggurat to migrate data to accommodate incoming writes frequently. Ziggurat-2 is also an interesting comparison point for EXT4-DJ and XFS-ML since those configurations take different approaches to use a small amount of PM to improve file system performance. The variation ends with Ziggurat-64 (i.e., Ziggurat with 64 GB of PM). The group migration size is set to 16 MB. We run each workload three times and report the average across these runs.

Table 5.2: Zipf Parameters. We vary the Zipf parameter, θ, to control the amount of locality in the access stream.

Locality	90/10	80/20	70/30	60/40	50/50
Parameter θ	1.04	0.88	0.71	0.44	0

5.5.2 Microbenchmarks

We demonstrate the relationship between access locality and the read/write throughput of Ziggurat with Fio [5]. Fio can issue random read/write requests according to Zipfian distribution. We vary the Zipf parameter θ to adjust the locality of random accesses. We present the results with a range of localities range from 90/10 (90% of accesses go to 10% of data) to 50/50 (Table 5.2). We initialize the files with 2 MB writes, and the total dataset is 32 GB. We use 20 threads for the experiments, each thread performs 4 KB I/Os to a private file, and all writes are synchronous.

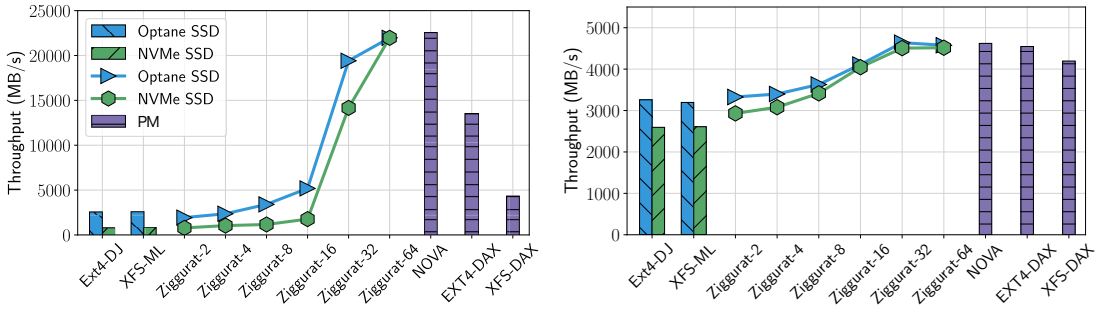
Figure 5.5 shows the results for Ziggurat, EXT4-DJ, and XFS-ML on Optane SSD and NVMe SSD, as well as NOVA, EXT4-DAX, and XFS-DAX on PM. The gaps between the throughputs from Optane SSD and NVMe SSD in both graphs are large because Optane SSD's read/write bandwidth is much higher than the NVMe SSD's. The throughput of Ziggurat-64 is close to NOVA for the 50/50 locality, the performance gap between Ziggurat-64 and NOVA is within 2%. This is because when all the data fits in PM, Ziggurat is as fast as NOVA. The throughput of Ziggurat-2 is within 5% of EXT4-DJ and XFS-ML.

In Figure 5.5a and Figure 5.5b, the random read performance of Ziggurat grows with increased locality. The major overhead of reads comes from fetching cold data blocks from disk to DRAM page cache. There is a dramatic performance increase in 90/10, due to CPU caching and the high locality of the workload.

In Figure 5.5c and Figure 5.5d, the difference between the random write performance of Ziggurat with different amounts of locality is small. Since all the writes are synchronous 4 KB aligned writes, Ziggurat steers these writes to PM. If PM is full, Ziggurat writes the new data blocks to the DRAM page cache and then flushes them to disk synchronously. Since the

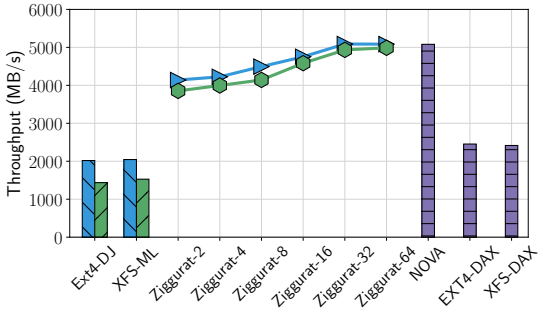
access pattern is random, the migration threads cannot easily merge the discrete data blocks to perform group migration in large sequential writes to disk. Therefore, the migration efficiency is limited by the random write bandwidth of disks, which leads to accumulated cold data blocks in PM. Increasing PM size, increasing locality, or reducing work set size can all help alleviate this problem.

We also measure the disk throughput of the random write workloads on Ziggurat-2 to show how Ziggurat fully utilizes disk bandwidth to achieve maximum performance. Although it is hard to merge the discrete data blocks to perform group migration, the per-CPU migration threads make full use of the concurrency of disks to achieve high migration efficiency. The average disk write bandwidth of Ziggurat-2 is 1917 MB/s and 438 MB/s for Optane SSD and NVMe SSD, respectively. These values are very close to the bandwidth limit numbers in Table 2.1.



(a) Fileserver

(b) Webproxy



(c) Varmail

Figure 5.6: Filebench performance (multi-threaded). Each workload runs with 20 threads so as to fully show the scalability of the file systems. The performance gaps between Optane SSD and NVMe SSD are smaller than the single-threaded ones.

Table 5.3: Filebench workload characteristics. These workloads have different read/write ratios and access patterns.

Workload	Average file size	# of files	I/O size (R/W)	Threads	R/W ratio
Fileserver	2MB	16K	16KB/16KB	20/1	1:2
Webproxy	2MB	16K	1MB/16KB	20/1	5:1
Varmail	2MB	16K	1MB/16KB	20/1	1:1

5.5.3 Macrobenchmarks

We select three Filebench workloads: fileserver, webproxy, and varmail to evaluate the overall performance of Ziggurat. Table 5.3 summarizes the characteristics of these workloads.

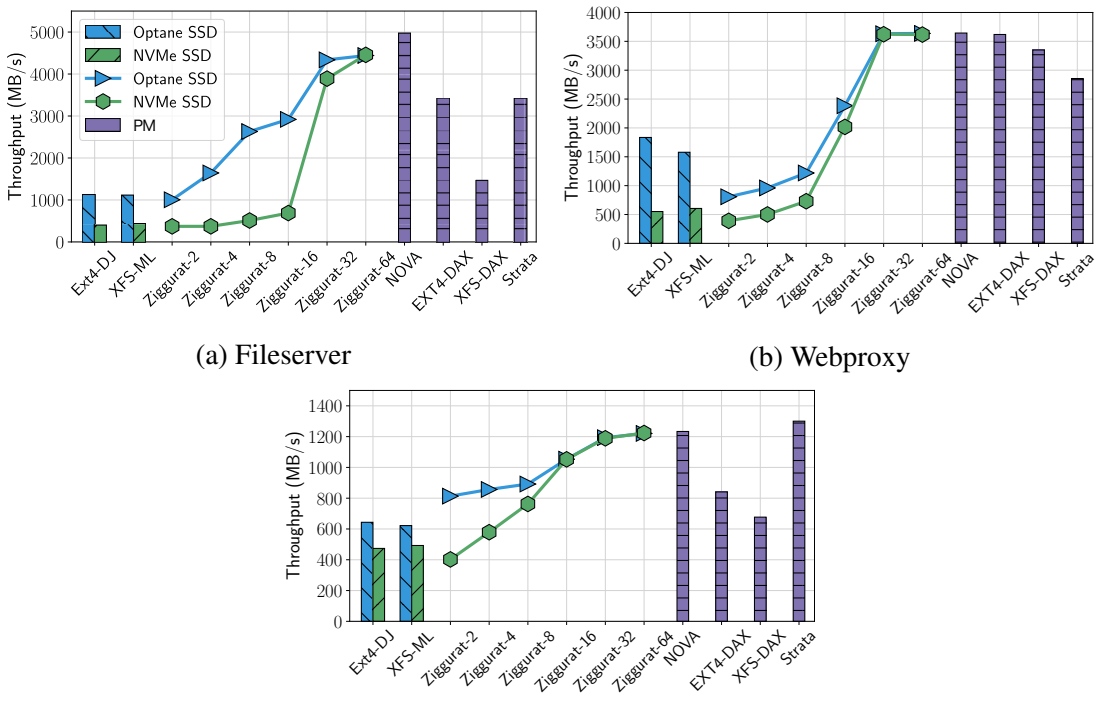
Figure 5.6 shows the multi-threaded Filebench throughput on our five comparison file systems and several Ziggurat configurations. In general, we observe that the throughput of Ziggurat-64 is close to NOVA, the performance gap between Ziggurat-64 and NOVA is within 3%. Ziggurat gradually bridges the gap between disk-based file systems and PM-based file systems by increasing the PM size.

Fileserver emulates the I/O activity of a simple file server, which consists of creates, deletes, appends, reads and writes. In the fileserver workload, Ziggurat-2 has similar throughput to EXT4-DJ and XFS-ML. The performance increases significantly when the PM size is larger than 32 GB since most of the data reside in memory. Ziggurat-64 outperforms EXT4-DAX and XFS-DAX by 2.6× and 5.1×.

Webproxy is a read-intensive workload, which involves appends and repeated reads to files. Therefore, all the file systems achieve high throughputs by utilizing the page cache.

Varmail emulates an email server with frequent synchronous writes. Ziggurat-2 outperforms EXT4-DJ and XFS-ML by 2.1× (Optane SSD) and 2.6× (NVMe SSD) on average. Varmail performs an `fsync` after every two appends. Ziggurat analyzes these synchronous appends and steers them to PM, eliminating the cost of most of the `fsync`s.

Figure 5.7 illustrates the single-threaded Filebench throughputs. Strata achieves the best



(a) Fileserver

(b) Webproxy

(c) Varmail

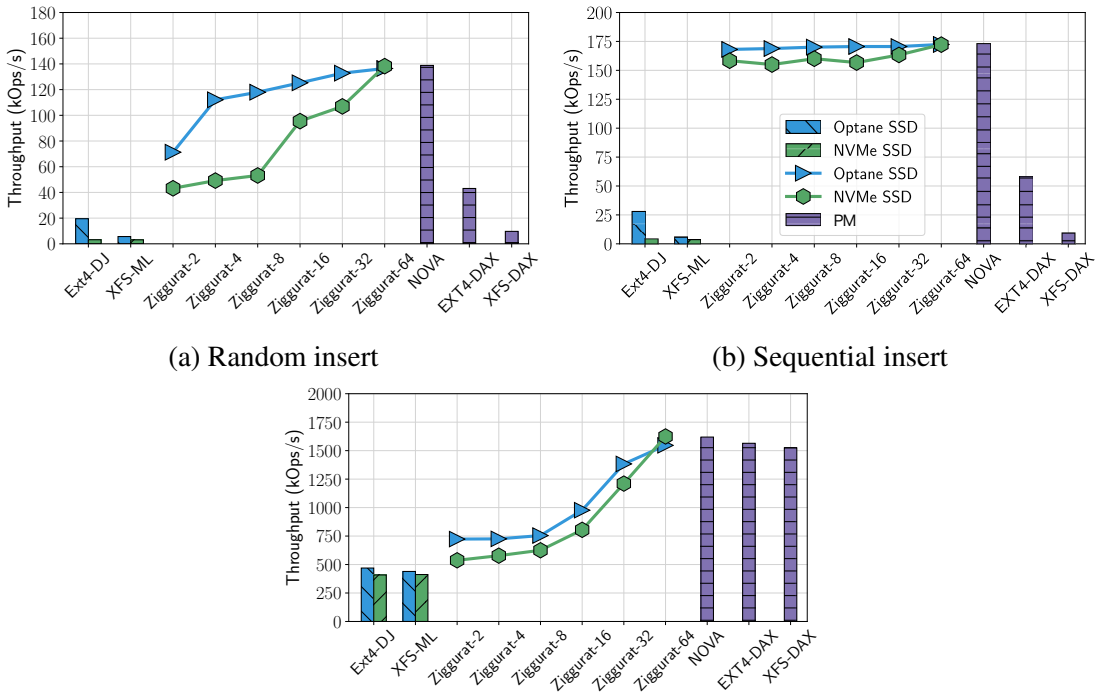
Figure 5.7: Filebench performance (single-threaded). For small amounts of PM, Ziggurat is no slower than conventional file systems running on disk. With large amount of PM, performance nearly matches that of PM-only file systems.

throughput in the varmail workload since its digestion skips many temporary durable writes which are superseded by subsequent writes. However, Ziggurat-64 outperforms Strata by 31% and 27% on the fileserver and webproxy workloads due to the inefficiency of reads in Strata.

5.5.4 Rocksdb

We illustrate the high performance of updating a key-value store with write-ahead logging (WAL) on Ziggurat with Rocksdb [24], a persistent key-value store based on log-structured merge trees (LSM-trees). Every update to RocksDB is written to two places: an in-memory data structure called memtable and a write-ahead log in the file system. When the size of the memtable reaches a threshold, RocksDB writes it back to disk and discards the log.

We select three Rocksdb workloads from db_bench: random insert (FillUniqueRandom),



(a) Random insert

(b) Sequential insert

(c) Sequential read

Figure 5.8: Rocksdb performance. Ziggurat shows good performance for inserting file data with write-ahead logging, due to the clear distinction between hot and cold files and its migration mechanism.

sequential insert (FillSeq), and sequential read (ReadSeq) to evaluate the key-value throughput and migration efficiency of Ziggurat. We set the writes to synchronous mode for a fair comparison. The database size is set to 32 GB.

Figure 5.8 measures the Rocksdb throughput. In the random insert workload, Ziggurat with 2 GB of PM achieves 8.6× and 13.2× better throughput than EXT4-DJ and XFS-ML, respectively. In the sequential insert workload, Ziggurat is able to maintain near-PM performance even when there are only 2 GB of PM. It achieves up to 38.9× and 46.5× throughput of EXT4-DJ and XFS-ML, respectively.

WAL is a good fit for Ziggurat. The reason is three-fold. First, since the workload updates WAL files much more frequently than the database files, the migration profiler can differentiate them easily. The frequently-updated WAL files remain in PM, whereas the rarely-updated

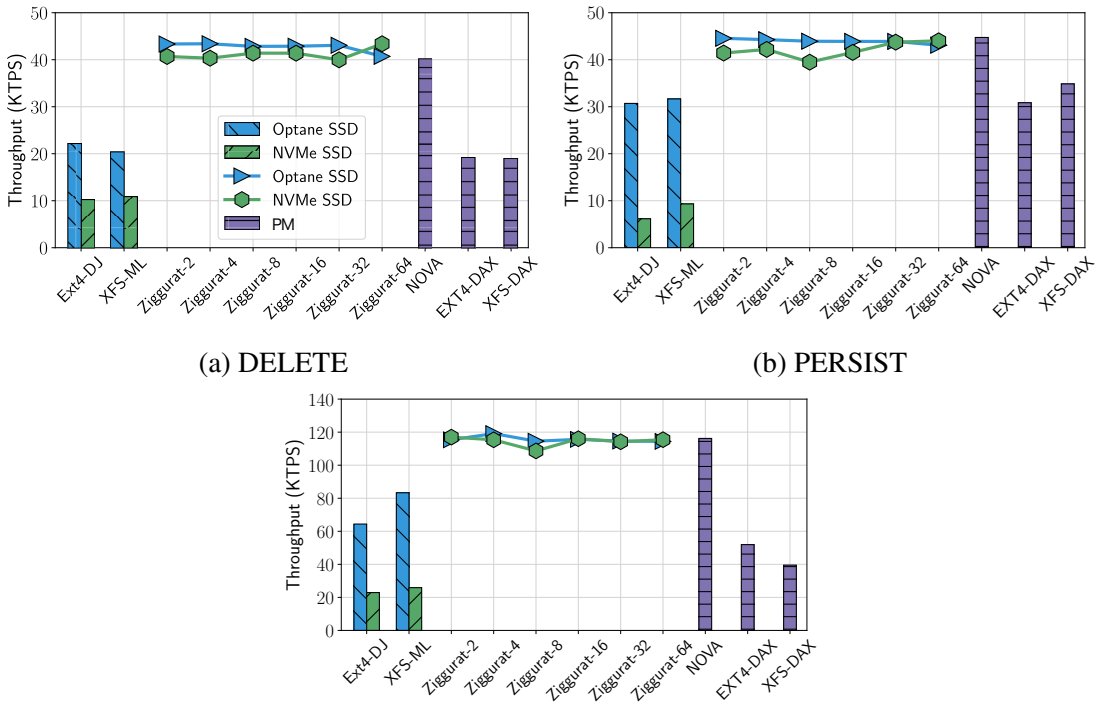
database files are subject to migration.

Second, the database files are usually larger than the group migration size. Therefore, group migration can fully-utilize the high sequential bandwidth of disks. Moreover, since Rocksdb mostly updates the journal files instead of the large database files, the migration threads can merge the data blocks from the database files and perform sequential writes to disk without interruption. The high migration efficiency helps clean up PM space more quickly so that PM can absorb more synchronous writes, which in turn boosts the performance.

Third, the WAL files are updated frequently with synchronous and small updates. The synchronicity predictor can accurately identify the synchronous write pattern from the access stream of the WAL files, and the write size predictor can easily discover that the updates to these files are too small to be steered to disk. Therefore, Ziggurat steers the updates to PM so that it can eliminate the double copy overhead caused by synchronous writes to disks. Since the entire WAL files are hot, Ziggurat is able to maintain high performance as long as the size of PM is larger than the total size of the WAL files, which is only 128 MB in our experiments.

Comparing Figure 5.8a and Figure 5.8b, the difference between the results from random and sequential insert of Ziggurat is due to read-modify-writes for unaligned writes. In the random insert workload, the old data blocks of the database files are likely to be on disk, especially when the PM size is small. Thus, loading them from disks introduces large overhead. However, in the sequential insert workload, the old data blocks come from recent writes to the files which are likely to be in PM. Hence, Ziggurat achieves near-PM performance in the sequential insert workload.

In sequential read, Ziggurat-2 outperforms EXT-DJ and XFS-ML by 42.8% and 47.5%. With increasing PM size, the performance of Ziggurat gradually increases. The read throughputs of Ziggurat-64, NOVA, EXT4-DAX, and XFS-DAX are close (within 6%).



(a) DELETE

(b) PERSIST

(c) WAL

Figure 5.9: SQLite performance. Ziggurat maintains near-PM performance because the hot journal files are either short-lived or frequently updated, so Ziggurat keeps them in PM. Migrating the cold database file with group migration in the background imposes little overhead to foreground file operations.

5.5.5 SQLite

We analyze the performance of different logging mechanisms on Ziggurat by measuring SQLite [80], a popular light-weight relational database that supports both undo and redo logging. It hosts the entire database in a single file, with other auxiliary files for logging (rollback or write-ahead log). We use Mobibench [46] to test the performance of SQLite with three journaling modes: DELETE, PERSIST and WAL. DELETE and PERSIST are rollback journaling modes. The journal files are deleted at the end of each transaction in DELETE mode. The PERSIST mode foregoes the deletion and instead overwrites the journal header with zeros. The WAL mode uses write-ahead logging for rollback journaling. The database size is set to 32 GB in the experiments. The experimental results are presented in Figure 5.9.

For DELETE and PERSIST, the journal files are either short-lived or frequently updated. Therefore, they are classified as hot files by the migration profiler of Ziggurat. Hence, Ziggurat only migrates the cold parts of the database files, leaving the journal files in PM to absorb frequent updates. The performance gain comes from accurate profiling and high migration efficiency of Ziggurat. With an efficient migration mechanism, Ziggurat can clear up space in PM fast enough for in-coming small writes. As a result, Ziggurat maintains near-PM performance in all configurations. Compared with block-based file systems running on Optane SSD, Ziggurat achieves 2.0× and 1.4× speedup for DELETE and PERSIST on average, respectively. Furthermore, Ziggurat outperforms block-based file systems running on NVMe SSD by 3.9× and 5.6× for DELETE and PERSIST on average, respectively.

In WAL mode, there are three types of files: the main database files and two temporary files for each database: *WAL* and *SHM*. The *WAL* files are the write-ahead log files, which are hot during key-value insertions. The *SHM* files are the shared-memory files which are used as the index for the WAL files. They are accessed by SQLite via `mmap`.

Ziggurat's profiler keeps these hot files in PM. Meanwhile, the cold parts of the large database files migrate to disks in the background. This only introduces very small overhead to the foreground database operations. Therefore, Ziggurat maintains near-PM performance even when there's only about 5% of data is actually in PM (Ziggurat-2), which outperforms block-based file systems by 1.61× and 4.78×, respectively. Ziggurat also achieves 2.22× and 2.92× higher performance compared with EXT4-DAX and XFS-DAX on average.

5.5.6 MySQL

We further evaluate the throughput of databases on Ziggurat with MySQL [72], another widely-used relational database. We measure the throughput of MySQL with TPC-C [81], a representative online transaction processing (OLTP) workload. We run the experiments with a data set size of 20 GB.

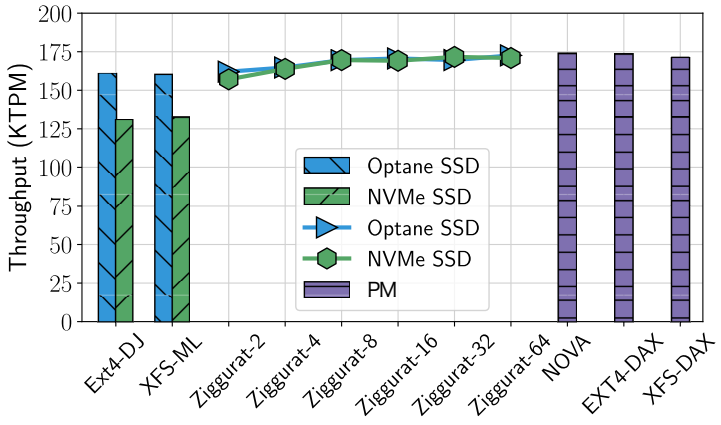


Figure 5.10: MySQL performance. Ziggurat manages to keep high throughput even with little PM.

Figure 5.10 shows the MySQL throughput. The performance of Ziggurat is always close to or better than EXT-DJ and XFS-ML. On average, Ziggurat-2 outperforms disk-based file systems by 1% (Optane SSD) and 19% (NVMe SSD). During the transactions, Ziggurat steers most of the small updates to PM. Since the transactions need to be processed in DRAM, Ziggurat is capable of migrating the data blocks to disks in time, which leaves ample PM space to receive new writes. Thus, Ziggurat maintains near-PM performance even with little PM.

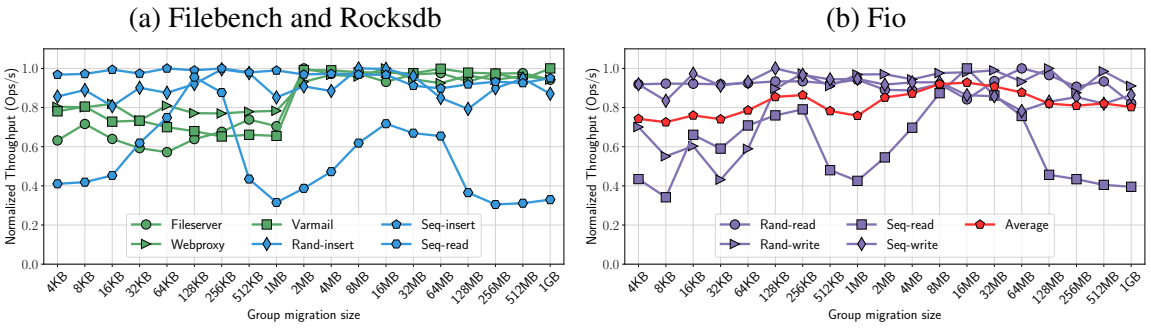


Figure 5.11: Performance impact of group migration size. We use Filebench (green), Rocksdb (blue), and Fio (purple) as our benchmarks. The average throughput (red) peaks when the group migration size is set to 16 MB.

5.5.7 Parameter Tuning

We illustrate the impact of the parameter choices on performance by measuring the throughput of workloads from Fio, Filebench and Rocksdb with a range of thresholds. We run the workloads with Ziggurat-2 on NVMe SSD.

Group migration size. We vary the group migration size from 4 KB to 1 GB. The normalized throughputs relative to the maximum performance are shown in Figure 5.11. In general, larger group migration size provides better performance for write-dominated workloads, such as Rand-write from Fio and Rand-insert from Rocksdb.

For read-dominated workloads, such as Seq-read from Fio and Seq-read from Rocksdb, the throughputs peak when the group migration size is set to 128 KB and 16 MB. This is because the maximum I/O size of our NVMe SSD is 128 KB and the CPU cache size of our experimental platform is 25 MB. Note that the group migration size is also the granularity of loading file data from disk to DRAM since we fetch file data in the granularity of write entry. On one hand, if the group migration size is too small, Ziggurat has to issue multiple requests to load the on-disk file data into DRAM, which hurts performance. On the other hand, if the group migration size is too large, then a small read request will fetch redundant data blocks from disk, which will waste I/O bandwidth and pollute CPU cache. As Figure 5.11b shows, the average throughputs of all ten workloads peak when the group migration size is set to 16 MB.

The throughputs of Filebench workloads are saturated when the group migration size reaches 2 MB because the average file size of the workloads is 2 MB. During the migration of the Filebench workloads, the data blocks of a file are coalesced into one write entry, which suits the access pattern of whole-file-reads.

Synchronous write size threshold. We vary the synchronous write size threshold from 4 KB to 1 GB. The performance results are insensitive to the synchronous write size threshold throughout the experiments. The standard deviation is less than 3% of the average throughput. We further examine the accuracy of the synchronicity predictor given different synchronous write

size thresholds. The predictor accurately predicts the presence or absence of an `fsync` in the near future 99% of the time. The lowest accuracy (97%) occurs when the synchronous write size is set between the average file size and the append size of Varmail. In this case, the first `fsync` contains the writes from file initialization and the first append, while the subsequent `fsync`s only contain one append. In general, the synchronous write size threshold should be set a little larger than the average I/O size of the synchronous write operations from the workloads. In this case, the synchronicity predictor can not only identify synchronously updated files easily, but also effectively distinguish asynchronous, large writes from rest of the access stream.

Sequential write counter threshold. We vary the sequential write counter threshold of Ziggurat from 1 to 64. We find that different sequential write counter thresholds have little impact on performance since the characteristics of our workloads are stable. Users should balance the trade-off between accuracy and prediction overhead when running workloads with unstable access patterns. A higher threshold number improves the accuracy of the sequential predictor, which can effectively avoid jitter in variable workloads. However, it also introduces additional prediction overhead for Ziggurat to produce correct prediction.

5.6 Related Work

The introduction of multiple storage technologies provides an opportunity of having a large uniform storage space over a set of different media with varied characteristics. Applications may leverage the diversity of storage choices either directly (e.g. the persistent read cache of RocksDB), or by using PM-based file systems (e.g. NOVA, EXT4-DAX or XFS-DAX). In this section, we place Ziggurat's approach to this problem in context relative to other work in this area.

PM-Based File Systems. BPFS [17] is a storage class memory (SCM) file system, which is based on shadow-paging. It proposes short-circuit shadow paging to curtail the overheads of

shadow-paging in regular cases. However, some I/O operations that involve a large portion of the file system tree (such as moving directories) still impose large overheads. Like BPFS, Ziggurat also exploits fine-grained copy-on-write in all I/O operations.

SCMFS [84] offers simplicity and performance gain by employing the virtual address space to enable continuous file addressing. SCMFS keeps the mapping information of the whole available space in a page table which may be scaled to several Gigabytes for large PM. This may result in a significant increase in the number of TLB misses. Although Ziggurat similarly maps all available storage devices into a unified virtual address space, it also performs migration from PM to block devices, and group page allocation which reduces TLB misses.

PMFS [21] is another PM-based file system which provides atomicity in metadata updates through journaling, but large size write operations are not atomic because it relies on small size in-place atomic updates. Unlike PMFS, Ziggurat's update mechanism is always through journaling with fine-grained copy-on-writes.

Dong *et al.* propose SoupFS [20], a simplified soft update implementation of an PM-based file system. They adjust the block-oriented directory organization to use hash tables to leverage the byte-addressability of PM. It also gains performance by taking out most synchronous flushes from the critical path. Ziggurat also exploits asynchronous flushes to clear the critical path for higher write throughput.

Tiering Systems. Hierarchical storage Management (HSM) systems date back decades to when disks and tapes were the only common massive storage technologies. There have been several commercial HSM solutions for block-based storage media such as disk drives. IBM Tivoli Storage Manager is one of the well-established HSM systems that transparently migrates rarely used or sufficiently aged files to a lower cost media. EMC DiskXtender is another HSM system with the ability of automatically migrating inactive data from the costly tier to a lower cost media. AutoTiering [90] is another example of a block-based storage management system. It uses a sampling mechanism to estimate the IOPS of running a virtual machine on other tiers. It

calculates their performance scores based on the IOPS measurement and the migration costs, and sorts all possible movements accordingly. Once it reaches a threshold, it initiates a live migration.

Since the invention of NVDIMMs, many fine-grained tiering solutions have been introduced. Agarwal *et al.* propose Thermostat [1], a methodology for managing huge pages in two-tiered memory which transparently migrates cold pages to PM as the slow memory, and hot pages to DRAM as the fast memory. The downside of this approach is the performance degradation for those applications with uniform temperature across a large portion of the main memory. Conversely, Ziggurat's migration granularity is variable, so it does not hurt performance due to fixed-size migration as in Thermostat. Instead of huge pages, it coalesces adjacent dirty pages into larger chunks for migration to block devices.

X-Mem [23] is a set of software techniques that relies on an off-line profiling mechanism. The X-Mem profiler keeps track of every memory access and traces them to find out the best storage match for every data structure. X-Mem requires users to make several modifications to the source code. Additionally, unlike Ziggurat, the off-line profiling run should be launched for each application before the production run.

Strata [53] is a multi-tiered user-space file system that exploits PM as the high-performance tier, and SSD/HDD as the lower tiers. It uses the byte-addressability of PM to coalesce logs and migrate them to lower tiers to minimize write amplification. File data can only be allocated in PM in Strata, and they can be migrated only from a faster tier to a slower one. The profiling granularity of Strata is a page, which increases the bookkeeping overhead and wastes the locality information of file accesses.

5.7 Summary

We have implemented and described Ziggurat, a tiered file system that spans across PM and disks. We manage data placement by accurate and lightweight predictors to steer incoming

file writes to the most suitable tier, as well as an efficient migration mechanism that utilizes the different characteristics of storage devices to achieve high migration efficiency. Ziggurat bridges the gap between disk-based storage and PM-based storage, and provides high performance and large capacity to applications.

Acknowledgments

This chapter contains material from "Ziggurat: A Tiered File System for Non-Volatile Main Memories and Disks", by Shengan Zheng, Morteza Hoseinzadeh, and Steven Swanson, which appeared in the Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST 19), 2019. The dissertation author is the primary investigator and the second author of this chapter.

Chapter 6

Conclusion

Persistent Memory (PM) technologies such as 3D XPoint and battery-backed DRAM provide non-volatile memory to the system and application software through a byte-addressable interface. Comparing to the traditional block-based storage media, PM offers massively higher bandwidth and lower latency. These benefits in performance come at financial and programming costs. Programming for PM requires extra effort to achieve data consistency. The volatile CPU caches intercept the memory access path and reorder the actual PM write operations leading to a nondeterministic synchronization behavior and eventually data inconsistency in the presence of a system failure. Additionally, PM's price per byte is multiple times higher than the traditional disks, limiting its scalability and storage capacity.

Chapter 3 addressed the programming issues by presenting Corundum, a Rust PM library that provides a safe means to access PM, in addition to enforcing several PM safety rules at compilation time. As a result, the data structures implemented in Corundum are formally verifiable to be free of any PM-related bugs. Our experimental results showed that most workloads implemented in Corundum perform faster than those written using PMDK, Atlas, Mnemosyne, and go-pmem, despite providing strong safety guarantees.

The enforced safety features of Corundum make it challenging to use risky optimization

to gain more performance. Chapter 4 presented Carbide, a programming framework that allows writing strictly safe persistent data structures in Corundum, and freely using them in C++. Carbide automatically generates the same interface for the Corundum types in C++ and allows using C++ flexibility to apply massive optimizations. We showed that a multi-threaded workload in Corundum can achieve up to 2.4× speedup compared with its pure Corundum implementation.

To address the last issue, Chapter 5 presented Ziggurat, a multi-tiered file system that provides disk capacity and PM throughput at the same time. Ziggurat achieves this goal by intelligently redirecting I/O operations to the best suitable device. In Ziggurat, data blocks can migrate between devices based on an accurate profiling mechanism. We showed that with a small amount of PM and a large SSD, the overall I/O throughput improved by up to 38.9× and 46.5× compared with EXT4 and XFS running on an SSD alone, respectively.

Acknowledgments

This chapter contains material from “Corundum: Statically-Enforced Persistent Memory Safety”, by Morteza Hoseinzadeh and Steven Swanson, which appeared in the Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 21), 2021. The dissertation author is the primary investigator and first author of this paper.

This chapter contains material from “Ziggurat: A Tiered File System for Non-Volatile Main Memories and Disks”, by Shengan Zheng, Morteza Hoseinzadeh, and Steven Swanson, which appeared in the Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST 19), 2019. The dissertation author is the primary investigator and the second author of this paper.

Appendix A

A Detailed Example of Carbide

Carbide allows the programmer to develop a safe data structure in Corundum and use it in C++. This process contains several automated steps. This appendix gives a complete example of using Carbide to develop a persistent key-value store with a command-line interface (CLI). The persistent key-value store is a hashmap implemented in Corundum, and the CLI is written in C++. We first discuss the Corundum implementation of the data structure, then we move to its usage in C++ and the CLI.

A.1 Data Type Definition

We define the data structure `KvStore<K,V,P>` in Corundum, where `K` is the key type, `V` is the value type, and `P` is the pool type. `KvStore` consists of a fixed size array of `Buckets`, which is a dynamically sized vector of key-index pair, and a vector of values. For better readability, we declare the following type aliases:

```
1 const BUCKETS_MAX: usize = 10;
2 type Bucket<K, P> = PVec<(K, usize), P>;
3 type BucketArray<K, P> = [PRefCell<Bucket<K, P>, P>; BUCKETS_MAX];
4 type ValueVector<V, P> = PVec<V, P>;
```

Since each `Bucket` in the `BucketArray` should be modifiable to allow adding or removing a key, every element in `BucketArray` is wrapped in a `PRefCell` to provide interior mutability. Then, we define the `KvStore` type as follows:

```
5  #[derive(Extern)]
6  #[mods(pool1)]
7  pub struct KvStore<K: PSafe, V: PSafe, P: MemPool> {
8    buckets: BucketArray<ByteArray<K, P>, P>,
9    values: PRefCell<ValueVector<ByteArray<V, P>, P>, P>,
10 }
```

Note that the `values` should provide interior mutability, too, so that it is possible to push and pop from the vector. In this example, the values do not change. Therefore we do not need to use interior mutability for each element.

An important point here is that the `KvStore` is not allowed to directly use the type parameters `K` and `V` because they do not have a fixed memory layout. Carbide warns if this rule is violated during the export rule checking process and suggests using `ByteArray` to make the type portable. The reason for not raising an error is that the type may consist of a sub-type that takes the type parameter and internally uses a `ByteArray` to wrap it, which is safe.

The programmer should specify the `Extern` procedural macro (Line 5) in order to generate the external interface for a specialized `KvStore` with `pool1` (as specified in Line 6). This procedural macro performs the exporting rule checks before generating the external interface.

A.1.1 Reducing Generic Type Parameters

To make the type exportable, Carbide specializes the type parameters with `void`, except for the pool type. For example, it creates `__KvStore<P>` type from `KvStore<K,V,P>` in the follow format:

```
pub type __KvStore<P: corundum::MemPool> = KvStore<void, void, P>;
```

Then, it specializes in the pool type in the named object type for every given pool module (as shown in Listing 5). Carbide reproduces the same parameters in the vessel class so that the programmer does not see the type parameter reduction in Rust and reparameterization in C++. The final result is the same parametric polymorphism in both languages.

A.1.2 Persistent Object Constructor

Carbide also requires the object to implement a constructor that takes the sizes of the type parameters (`K` and `V`, in this example), and a reference to the journal object, such as the following:

```
11 impl<K: PSafe, V: PSafe, P: MemPool> KvStore<K, V, P> {
12   pub fn new(_: usize, _: usize, _: &Journal<P>) -> Self {
13     let mut me = MaybeUninit::<BucketArray<ByteArray<K, P>, P>>::
     ↪ uninit();
14     let mut vec = Vec::<PRefCell<Bucket<ByteArray<K, P>, P>, P>>::
     ↪ with_capacity(BUCKETS_MAX);
15     for _ in 0..BUCKETS_MAX {
16       vec.push(PRefCell::new(Bucket::new()));
17     }
18     unsafe {
19       std::ptr::copy_nonoverlapping(vec.as_ptr(),
     ↪  &mut (*me.as_mut_ptr())[0], BUCKETS_MAX);
20     }
21     Self {
22       buckets: unsafe { me.assume_init() },
23       values : PRefCell::new(ValueVector::new())
24     }
25   }
26 }
```

The method `new()` at Line 12 is a constructor that Carbide uses to initialize a `KvStore` object in C++. In this example, we do not need any type parameter size since the hashmap is initially empty. However, the `buckets` should be initialized with a fixed array of `Buckets`. `ByteArray` is `!Copy`, and Rust does not allow instantiating a fixed-size array of a type which is not `Copy`, because the array uses `memcpy()` to initialize the elements instead of calling the constructors. For

that reason, we first create an uninitialized vector of `Buckets` in Line 13, initialize the elements one by one in Lines 15-17, and then copy the vector elements to the array in Line 19. The `copy_nonoverlapping()` and `MaybeUninit::init()` functions are considered unsafe as they may interfere memory safety.

A.1.3 Exporting the Functionality

The next step is to define the operations on `KvStore`. Here we have four functions of `put()`, `get()`, `foreach()`, and `clear()`. The key insight is that these functions should be FFI-compatible to be able to be called externally, and the type parameters cannot be determined as FFI-safe while compiling. Therefore, Carbide does not allow directly using the type parameters in the argument list. Alternatively, it allows using the type parameters wrapped in `Gen<T,P>`, where `T` is the type parameter, and `P` is the pool type. `Gen` is predefined in both Rust and C++ to provide an interoperable reference object. Keeping these criteria in mind, we define `KvStore`'s functionality as listed in the following `impl` section:

```
27 #[extern]
28 impl<K: PSafe + TxInSafe, V: PSafe + TxInSafe, P: MemPool>
29 KvStore<K, V, P> {
30     pub fn put(&self, key: Gen<K, P>, val: Gen<V, P>,
31         hash: extern fn(&K)->usize, eq: extern fn(&K,&K)->bool
32     ) {
33         let h = hash(&key);
34         let index = h % BUCKETS_MAX;
35 
36         for e in &*self.buckets[index].borrow() {
37             if eq(&e.0, &key) {
38                 P::transaction(|j| {
39                     let values = self.values.borrow();
40                     values[e.1].update_from_gen(val, j);
41                 }).unwrap();
42                 return;
43             }
44         }
45 
```

```
46          P::transaction(|j| {
47              let mut values = self.values.borrow_mut(j);
48              values.push(ByteArray::from(val), j);
49              let mut bucket = self.buckets[index].borrow_mut(j);
50              bucket.push((ByteArray::from(key), values.len() - 1), j);
51          }).unwrap();
52      }
53
54      pub fn get(&self, key: Gen<K,P>, hash: extern fn(&K)->usize,
55          eq: extern fn(&K,&K)->bool
56      ) -> Gen<V,P> {
57          let h = hash(&key);
58          let index = h % BUCKETS_MAX;
59
60          for e in &*self.buckets[index].borrow() {
61              if eq(&e.0, &key) {
62                  let values = self.values.borrow();
63                  return unsafe { values[e.1].get_gen() };
64              }
65          }
66          Gen::null()
67      }
68
69      pub fn foreach(&self, f: extern fn(&K,&V)->()) {
70          let values = self.values.borrow();
71          for index in 0..BUCKETS_MAX {
72              for e in &*self.buckets[index].borrow() {
73                  f(&e.0, &values[e.1]);
74              }
75          }
76      }
77
78      pub fn clear(&self, j: &Journal<P>) {
79          for key in &self.buckets {
80              key.borrow_mut(j).clear();
81          }
82          self.values.borrow_mut(j).clear();
83      }
84  }
```

A typical `KvStore` data structure uses a hash function to map keys of a hashable type into a scalar number that can be used for indexing the values. Although it is possible to generate a

hash function from the byte array, it becomes complicated when the data has a reference field (e.g., a string field). Therefore, our `KvStore` example takes the hash function pointer as an argument in both `get()` and `put()` functions (Line 31 and Line 54). The same analogy stands for `eq()` function to test the keys (Line 31 and Line 55). Notice that the function pointer also has a FFI-compatible signature. It is also possible to make a function transactional by adding a reference to a `Journal` object in the argument list, as exemplified in `clear` (Line 78).

A.1.4 Type Parameter Reproduction for Generic Functions

Since a reference in FFI is equivalent to a pointer in C++, Carbide allows having them in the function pointers. However, a pointer to a type parameter in an FFI function is prevented by the compiler. Carbide internally reduces the type parameters using the `void` type to make that possible and reparameterize the functions in the vessel classes. This type-reduction and reparameterization is automated and hidden from the programmer.

A.1.5 FFI Function Input

The type `ByteArray` itself (as in key and value types) is not FFI-compatible, so it cannot be used as an input type. It is impossible to have a generic FFI, so using the type parameters anywhere in the function declaration raises a compilation error. We develop an FFI-compatible reference type `Gen` that is convertible into a `ByteArray` of the same data and pool type. For example, the `put()` function (Lines 30-52) receives a `Gen<V, P>` as the value and converts it into a `ByteArray` in Line 48.

The `put()` function first calculates the bucket index using the hash function. Then it tries to find the old value assigned to the key. If successful, it transactionally updates the value by calling `update_from_gen()` as in Lines 38-41. Otherwise, it transactionally appends the new key value pair through Lines 46-51.

A.1.6 FFI Function Output

The `get()` function also takes the `key`, `hash()`, and `eq()`, but it returns a reference to the requested object. Like the inputs, Carbide prevents functions from returning a parametrically typed object or a reference to it. However, it allows functions to return a `Gen` object referring to a type parameter. Note that there is a significant difference between using `Gen` as an input type and using it as an output type. When `Gen` is used as an input, the object is most likely initialized in C++ and being transferred to Rust to construct a `ByteArray` object. In contrast, an output `Gen` refers to an already instantiated `ByteArray` object in Rust.

A.1.7 Anatomy of `Gen<T,P>`

Since Rust and C++ have separated scopes, neither of their type systems can manage the lifetime of the objects. Carbide uses a notion of expanded lifetime (explained in Section 4.2.2) to address this issue using the cross-language reference object `Gen<T,P>`. `Gen` is provided to bridge this gap and create a reference from C++ to a Rust object that allows expanding the lifetime between the two scopes. `Gen` and `ByteArray` are reciprocally convertible to each other. However, converting a `Gen` object into a `ByteArray` takes its ownership (i.e., it manages the resource) while converting a `ByteArray` to a `Gen` shares the ownership and does not involve the resource management. This is to guarantee that every persistent resource has an owner in Rust to manage it. However, The Rust implementation of `Gen` should cooperate with C++ to make such a guarantee.

In C++, `Gen` has two constructors: a constructor from `void*`, and a constructor from an object of the parametric type `T`. The former is used when creating a reference to a Rust-resident `ByteArray` (e.g., using `Gen` as an output type), and the latter is used to create an object in C++ and send it to Rust (e.g., using `Gen` as an input type). Considering that Carbide reduces the type parameters using the `void` type, the first constructor can be used to reparameterize the output type parameters. Carbide uses this process for generating the vessel class. `Gen<T, P>`

implements implicit cast operation to `Gen<void, P>`, so the second constructor can also be used for re-parameterizing the input types.

A.2 API Source Generation for Persistent Objects

Carbide gets every exporting type specialized with `void` type and the given pool type (in this example, `pool1`) and uses the same specialization for the FFIs. After that, it generates a C++ header file containing the demangled symbols in the library to let the program call them. In the `KvStore` example, four user-defined external functions are translated into the following C++ code snippet:

```
1  template<typename K, typename V, typename P>
2  struct KvStore;
3
4  template<typename P>
5  using __KvStore = KvStore<void, void, P>;
6
7  void __pool1_kvstore_put(const __KvStore<pool1> *__self,
       ↪                   Gen<void, pool1> key,
       ↪                   Gen<void, pool1> val,
       ↪                   size_t (*hash)(const void*),
       ↪                   bool (*eq)(const void*, const void*));
8
9  Gen<void, pool1> __pool1_kvstore_get(const __KvStore<pool1> *__self,
       ↪                         Gen<void, pool1> key,
       ↪                         size_t (*hash)(const void*),
       ↪                         bool (*eq)(const void*, const void*));
10
11 void __pool1_kvstore_foreach(const __KvStore<pool1> *__self,
       ↪                         void (*f)(const void*, const void*));
12
13 void __pool1_kvstore_clear(const __KvStore<pool1> *__self,
       ↪                       const Journal<pool1> *j);
```

The `KvStore` type has a declaration in C++ at Line 2 as a template type, but its definition is in Rust. This forward declaration is merely used to type the pointers and use the C++'s type system.

The `void` specialized version of `KvStore` at Line 5 (i.e., `__KvStore`) is for tidiness.

Carbide automatically generates three resource management functions for every persistent type: 1) a `new()` function that Carbide requires for every data type to instantiate the object externally, 2) an `open()` function that Carbide automatically generates to allow loading a named object of the given type, and 3) a `drop()` function to release the allocation. Below is an example of C++-translated resource management functions for the `KvStore` type:

```
14 const __KvStore<pool1> *__pool1_kvstore_new(size_t K_size,
     ↪                                         size_t V_size,
     ↪                                         const void *j);
15
16 const __KvStore<pool1> *__pool1_kvstore_open(const __pool1_root_t *p,
     ↪                                          size_t K_size,
     ↪                                          size_t V_size,
     ↪                                          const char *name);
17
18 void __pool1_kvstore_drop(__KvStore<pool1> *obj);
```

A.3 Vessel Class Generation for Persistent Types

Carbide heuristically reimplements the polymorphism in C++ using the generated FFI. It preserves the interface to use the type the same way it is used in Rust. Carbide reduces the type parameters of FFI and the types in Rust, compiles the result to generate a dynamic library, and then reparameterizes the type in C++. We call the C++ reparameterized type a *vessel class*.

A.3.1 Type Traits

The vessel class parameterizes the contained types and the pool type. However, the pool types are not practically generic, as there is a finite number of defined pool types. In Carbide, every FFI has a unique implementation for each pool type. For instance, in Section A.2, the `put()` function is translated into `__pool1_kvstore_put()` that specializes every argument type with

either `void` or `pool1`. But, we want to make the `KvStore` type generic for the pool type too. We use the *traits* technique to generalize the pool type. The following piece of code represents the trait class of `KvStore` that parameterizes the pool type in the generated APIs listed in Section A.2.

```
1  template < class _P >
2  struct kvstore_traits {
3      static const __KvStore<_P>* __create(size_t K_size, size_t V_size,
       ↪  const journal *j);
4
5      static void drop(__KvStore<_P> *obj);
6
7      static const __KvStore<_P>* open(const void *p, size_t K_size,
       ↪ size_t V_size, const char *name);
8
9      template < class K, class V >
10     static void clear(const __KvStore<_P> *__self, const Journal<
       ↪ pool1> *j);
11
12     template < class K, class V >
13     static void foreach(const __KvStore<_P> *__self, void (*f)(const
       ↪  K*, const V*));
14
15     template < class K, class V >
16     static void put(const __KvStore<_P> *__self, Gen<K, _P> key, Gen<
       ↪ V, _P> val, uintptr_t (*hash)(const K*), bool (*eq)(const K*,
       ↪ const K*));
17
18     template < class K, class V >
19     static Gen<V, _P> get(const __KvStore<_P> *__self, Gen<K, _P> key,
       ↪  uintptr_t (*hash)(const K*), bool (*eq)(const K*, const K*));
20 };
```

Since we chose `pool1` at Line 6 in Section A.1, Carbide implements this trait class for `pool1`, in the follow way:

```
21 template<>
22 struct kvstore_traits<pool1> {
23     typedef typename pool_traits<pool1>::journal journal;
24
25     static const __KvStore<pool1>* __create(size_t K_size, size_t
       ↪ V_size, const journal *j) {
```

```
26          return __pool1_kvstore_new(K_size, V_size, j);
27      }
28
29      static void drop(__KvStore<pool1> *obj) {
30          __pool1_kvstore_drop(obj);
31      }
32
33      static const __KvStore<pool1>* open(const __pool1_root_t *p,
    ↪ size_t K_size, size_t V_size, const char *name) {
34          return __pool1_kvstore_open(p, K_size, V_size, name);
35      }
36
37      template < class K, class V >
38      static void clear(const __KvStore<pool1> *__self, const Journal<
    ↪ pool1> *j) {
39          __pool1_kvstore_clear(__self, j);
40      }
41
42      template < class K, class V >
43      static void foreach(const __KvStore<pool1> *__self, void (*f)(
    ↪ const K*, const V*)) {
44          __pool1_kvstore_foreach(__self, (void(*)(const void*, const
    ↪ void*))f);
45      }
46
47      template < class K, class V >
48      static void put(const __KvStore<pool1> *__self, Gen<K, pool1> key,
    ↪  Gen<V, pool1> val, uintptr_t(*hash)(const K*), bool(*eq)(const
    ↪  K*, const K*)) {
49          __pool1_kvstore_put(__self, key, val, (uintptr_t(*)(const void
    ↪ *))hash, (bool(*)(const void*,const void*))eq);
50      }
51
52      template < class K, class V >
53      static Gen<V, pool1> get(const __KvStore<pool1> *__self, Gen<K,
    ↪ pool1> key, uintptr_t (*hash)(const K*), bool(*eq)(const K*,
    ↪ const K*)) {
54          return __pool1_kvstore_get(__self, key, (uintptr_t (*)(const
    ↪ void*))hash, (bool(*)(const void*, const void*))eq);
55      }
56  };
```

The trait class creates a type alias for the pool-specific journal object in Line 23. It gets the three

resource management methods, and the four user-defined functions specialized with `pool1`.

A.3.2 Vessel Class Implementation

Using this trait class, we can define the vessel class as follows:

```
57 template<class K, class V, class _P>
58 class pkvstore_t : public carbide::psafe_type_parameters {
59
60     typedef pool_traits<_P>  pool_traits;
61     typedef typename pool_traits::handle  handle;
62     typedef typename pool_traits::journal  journal;
63     typedef carbide::pointer_t<__KvStore<_P>, _P>  pointer;
64     using Self = pkvstore_t;
65
66     pointer inner;
67     typename carbide::make_persistent<std::string, _P>::type name;
68     bool moved;
69     bool is_root;
70
71     static std::unordered_set<std::string> objs;
72
73 private:
74     inline __KvStore<_P>* self() const {
75         return const_cast<__KvStore<_P>*>(inner.operator->());
76     }
77
78 public:
79     explicit pkvstore_t(const journal *j,
    ↪ const std::string &name = "(anonymous)") {
80         inner = pointer::from(kvstore_traits<_P>::__create(sizeof(K),
    ↪           sizeof(V), j));
81         this->name = name.c_str();
82         is_root = false;
83         moved = false;
84     }
85
86     explicit pkvstore_t(const pkvstore_t &other) {
87         if (pool_traits::valid(this)) {
88             assert(!other.moved, "object was already moved");
89             const_cast<Self&>(other).moved = true;
```

```
90          }
91          inner = other.inner;
92          name = other.name;
93          is_root = false;
94          moved = false;
95      }
96
97      explicit pkvstore_t(const handle *pool, const std::string &name)
    ↪ {
98          _P::txn([this,&pool,&name](auto j) {
99              assert(objs.find(name) == objs.end(),
    ↪                 "'%s' was already open", name.c_str());
100             this->name = name.c_str();
101             objs.insert(name);
102             pointer obj = pointer::from_unsafe(kvstore_traits<_P>::open(
    ↪                         pool, sizeof(K), sizeof(V), name.c_str()));
103             memcpy((void*)&inner, (void*)&obj, sizeof(pointer));
104             is_root = true;
105         } );
106     }
107
108     pkvstore_t() = delete;
109     pkvstore_t &operator=(pkvstore_t &&) = delete;
110     void *operator new (size_t) = delete;
111     void *operator new[] (size_t) = delete;
112     void  operator delete[] (void*) = delete;
113     void  operator delete   (void*) = delete;
114
115     ~pkvstore_t() noexcept(false) {
116         if (!moved && pool_traits::valid(this)) {
117             if (is_root) {
118                 auto n = name.c_str();
119                 assert(objs.find(n)!=objs.end(), "'%s' is not open", n);
120                 objs.erase(n);
121             } else {
122                 kvstore_traits<_P>::drop(
123                     static_cast<__KvStore<_P>*>(
124                         static_cast<void*>(inner)
125                     )
126                 );
127             }
128         }
129     }
```

```
130
131     // user-defined methods
132     void clear(const Journal<pool1> *j) const {
133         kvstore_traits<_P>::template clear<K,V>(self(), j);
134     }
135
136     void foreach(void (*f)(const K*, const V*)) const {
137         kvstore_traits<_P>::template foreach<K,V>(self(), f);
138     }
139
140     void put(Gen<K, _P> key, Gen<V, _P> val,
    ↪ uintptr_t(*hash)(const K*),
    ↪ bool(*eq)(const K*, const K*)) const {
141         kvstore_traits<_P>::template
142             put<K,V>(self(), key, val, hash, eq);
143     }
144
145     Gen<V, _P> get(Gen<K, _P> key,
    ↪ uintptr_t(*hash)(const K*),
    ↪ bool(*eq)(const K*, const K*)) const {
146         return kvstore_traits<_P>::template
147             get<K,V>(self(), key, hash, eq);
148     }
149 };
```

Line 58 defines the vessel class as a descendent of the `psafe_type_parameters` to signal the import analyzer to check the type parameters during instantiation. The vessel class `pkvstore_t` generalizes the key (`K`), the value (`V`), and the pool (`_P`) types. It contains four member variables:

1. `inner` is a persistent pointer to the original object created in Rust. In other words, it is the `self` object (a.k.a. the receiver) of `KvStore` type.
2. `name` is a persistent string value to address this object when it is instantiated as a named root object. In other cases, this field is unused.
3. `moved` is a boolean value to help resource management by dynamically indicating what variable is the owner of this object.
4. `is_root` is also a boolean value to indicate that whether this object is a root to skip

deallocation on destruction.

The vessel class has three explicit constructors for different situations. The first one is a transactional constructor defined in Lines 79-84, and it is used to create a non-root `pkvstore_t` object.

The second one is a copy constructor defined in Lines 86-95. We use the copy constructor to move the original `KvStore` reference to a new owner. We first check if the new owner is a persistent object, and if that is true, we reset the `moved` flag for the old owner. Using this mechanism, and knowing that if the owner is a persistent object, we can decide whether to reclaim the allocation in the destructor function (Line 116).

The last automatically generated constructor (Lines 97-106) is used when the object is instantiated as a root. We call this method an open constructor because it uses the open function to access data or create a permanent instance of the type.

The vessel type maintains the list of loaded objects' names in a static set, `objs` (Line 71). This set is used to keep track of root objects of this type. If the object with a specific name was already open, the open constructor throws an exception to prevent multiple references to the same persistent object.

The destructor checks if this object is the persistent owner of the original object. If so, it checks if the object is a root. If it is not a root, it calls the drop function from the `kvstore_traits`. There is no need to reclaim the allocation if it is a root, as removing the name from the list of loaded objects would suffice.

A.4 Using the Data Structure in C++

Carbide implements the vessel template classes for every Corundum type with the same interface in C++. The programmer may work with the data structure using the same interface as defined in Corundum. The types are defined in separate header files with uncapitalized

names. In this example, the Corundum type `KvStore<K,V,P>` is transformed into a vessel class in '*kvstore.hpp*' with a new name of '`pkvstore_t<K,V,P>`'. Since it is specialized only with `pool1`, the value of `P` cannot be something else. To make them tidier, Carbide creates an alias for every type that is specialized with a pool. For example, `pool1::kvstore<K,V>` is equivalent to `pkvstore_t<K,V,pool1>`.

Given the Carbide's type definitions, we develop a CLI for our `KvStore` in C++. The code snippet below shows how we use Carbide's automatically generated source code to utilize Rust-implemented `KvStore`.

```
1  #include "lib.h"
2  #include "pool1.hpp"
3  #include "kvstore.hpp"
4
5  #include <pstdlib>
6  #include <functional>
7
8  void h(char *argv[]) {
9    std::cerr << "usage: " << argv[0] << " <pool file path>\n";
10 }
11
12 typedef pool1::make_persistent<std::string>::type pstr;
13
14 size_t hash(const pstr *key) {
15   return std::hash<pstr>{}(*key);
16 }
17
18 bool eq(const pstr *a, const pstr *b) {
19   return std::string(a->c_str()) == b->c_str();
20 }
21
22 int main(int argc, char *argv[]) {
23   if (argc < 1) {
24     show_usage(argv);
25     return 1;
26   }
27
28   std::string cmd, key;
29   int val;
```

```
30  const char *path = argv[1];
31
32  using namespace open_flags;
33
34  pool1 p1(path, O_CFNE | O_1GB);
35  auto hdl = p1.handle();
36
37  pool1::kvstore<pstr, int> kv(hdl, "KVStore");
38
39  std::cout << "$ ";
40  while (std::cin >> cmd) {
41    if (cmd == "exit" || cmd == "q") {
42      return 0;
43    } else if (cmd == "get") {
44      std::cin >> key;
45      auto res = kv.get(pstr(key.c_str()), hash, eq);
46      if (res)
47        std::cout << *res << std::endl;
48      else
49        std::cout << "none" << std::endl;
50    } else if (cmd == "put") {
51      std::cin >> key >> val;
52      kv.put(pstr(key.c_str()), val, hash, eq);
53    } else if (cmd == "burst") {
54      std::cin >> cmd >> val;
55      if (cmd == "get") {
56        for(int i=0; i < val; i++) {
57          char key[32] = {0};
58          sprintf(key, "key%d", i);
59          kv.get(pstr(key), hash, eq);
60        }
61      } else if (cmd == "put") {
62        for(int i=0; i < val; i++) {
63          char key[32] = {0};
64          sprintf(key, "key%d", i);
65          kv.put(pstr(key), i, hash, eq);
66        }
67      }
68    } else if (cmd == "print" || cmd == "p") {
69      kv.foreach([](auto k, auto v) {
70        printf("%5s => %-5d ", k->c_str(), *v);
71      });
72      printf("\n");
```

```
73      } else if (cmd == "clear" || cmd == "c") {
74        pool1::txn([&](auto j){
75          kv.clear(j);
76        })
77      } else {
78        std::cout << "bad command '" << cmd << "'\n";
79      }
80      std::cout << "$ ";
81    }
82    return 0;
83  }
```

In this example we have defined `pool1` as the persistent memory pool and `kvstore` as the persistent type. We open the pool in Line 34 and obtain its handle at Line 35 in order to use it for loading other root objects. In Line 37, we instantiate the kvstore. Carbide allows making STL types persistent using `make_persistent`; we make the `string` type a persistent in Line 12, and use it to get `kvstore` specialized. The value time is integer. Note that we use the open constructor to access the persistent object named "KVStore".

The rest of the code is pretty straightforward. To use any persistent operation, we just need to call the function as in Line 45. However, if the operation is transactional, such as `clear()` at Line 75, we need to open a transaction to access a journal object.

Bibliography

[1] Neha Agarwal and Thomas F Wenisch. Thermostat: Application-transparent page management for two-tiered main memory. *ACM SIGARCH Computer Architecture News*, 45(1):631–644, 2017.

[2] Mohammad Alshboul, James Tuck, and Yan 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. IEEE, 2018.

[3] Joy Arulraj, Andrew Pavlo, and Subramanya R Dulloor. Let's talk about storage & recovery methods for non-volatile memory database systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 707–722. ACM, 2015.

[4] Malcolm Atkinson and Ronald Morrison. Orthogonally persistent object systems. *The VLDB Journal*, 4(3):319–401, 1995.

[5] Jens Axboe. Flexible I/O Tester, 2017. `https://github.com/axboe/fio`.

[6] Emery D Berger, Ting Yang, Tongping Liu, and Gene Novark. Grace: Safe multithreaded programming for c/c++. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 81–96, 2009.

[7] 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.

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

[9] Bill Bridge. personal communication, 2020.

[10] Ignacio Cano, Srinivas Aiyar, Varun Arora, Manosiz Bhattacharyya, Akhilesh Chaganti, Chern Cheah, Brent N Chun, Karan Gupta, Vinayak Khot, and Arvind Krishnamurthy. Curator: Self-managing storage for enterprise clusters. In *NSDI*, pages 51–66, 2017.

[11] 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.

[12] Andreas Chatzistergiou, Marcelo Cintra, and Stratis D Viglas. Rewind: Recovery write-ahead system for in-memory non-volatile data-structures. *Proceedings of the VLDB Endowment*, 8(5):497–508, 2015.

[13] E Chen, D Apalkov, Z Diao, A Driskill-Smith, D Druist, D Lottis, V Nikitin, X Tang, S Watts, and S Wang. Advances and future prospects of spin-transfer torque random access memory. *IEEE Transactions on Magnetics*, 46(6):1873–1878, 2010.

[14] 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*, pages 228–243, 2013.

[15] 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.

[16] Nachshon Cohen, David T Aksun, and James R Larus. Object-oriented recovery for non-volatile memory. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–22, 2018.

[17] 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*, pages 133–146. ACM, 2009.

[18] Biplob Debnath, Sudipta Sengupta, and Jin Li. Flashstore: high throughput persistent key-value store. *Proceedings of the VLDB Endowment*, 3(1-2):1414–1425, 2010.

[19] 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*, pages 125–136, 2016.

[20] Mingkai Dong and Haibo Chen. Soft updates made simple and fast on non-volatile memory. In *2017 USENIX Annual Technical Conference (USENIX ATC 17). USENIX Association, Santa Clara, CA*, pages 719–731, 2017.

[21] 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*, page 15. ACM, 2014.

[22] 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.

[23] Subramanya R Dulloor, Amitabha Roy, Zheguang Zhao, Narayanan Sundaram, Nadathur Satish, Rajesh Sankaran, Jeff Jackson, and Karsten Schwan. Data tiering in heterogeneous memory systems. In *Proceedings of the Eleventh European Conference on Computer Systems*, page 15. ACM, 2016.

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

[25] Ru Fang, Hui-I Hsiao, Bin He, C Mohan, and Yun Wang. High performance database logging using storage class memory. In *2011 IEEE 27th International Conference on Data Engineering*, pages 1221–1231. IEEE, 2011.

[26] Michal Friedman, Naama Ben-David, Yuanhao Wei, Guy E Blelloch, and Erez Petrank. Nvtraverse: in nvram data structures, the destination is more important than the journey. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 377–392, 2020.

[27] Michal Friedman, Erez Petrank, and Pedro Ramalhete. Mirror: making lock-free data structures persistent. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 1218–1232, 2021.

[28] Jerrin Shaji George, Mohit Verma, Rajesh Venkatasubramanian, and Pratap Subrahmanyam. go-pmem: Native support for programming persistent memory in go. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 859–872, 2020.

[29] The Go programming language. `https://golang.org/`.

[30] Hamed Gorjiara, Guoqing Harry Xu, and Brian Demsky. Jaaru: Efficiently model checking persistent memory programs. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 415–428, 2021.

[31] Vincent Gramoli, Rachid Guerraoui, and Vasileios Trigonakis. Tm2c: a software transactional memory for many-cores. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 351–364, 2012.

[32] 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.

[33] R HAGMANN. Reimplementing the cedar file system using logging and group commit. In *Proc. 11th ACM Symposium on Operating System Principles Austin, TX*, pages 155–162, 1987.

[34] Dongsu Han, Aditiya Agarwala, David G Andersen, Michael Kaminsky, Konstantina Papagiannaki, and Srinivasan Seshan. Mark-and-sweep: getting the" inside" scoop on neighborhood networks. In *Proceedings of the 8th ACM SIGCOMM conference on Internet measurement*, pages 99–104, 2008.

[35] Dave Hitz, James Lau, and Michael A Malcolm. File system design for an nfs file server appliance. In *USENIX winter*, volume 94, 1994.

[36] Morteza Hoseinzadeh, Mohammad Arjomand, and Hamid Sarbazi-Azad. Reducing access latency of mlc pcms through line striping. In *Proceeding of the 41st annual international symposium on Computer architecuture (ISCA)*, pages 277–288, 2014.

[37] Morteza Hoseinzadeh and Steven Swanson. Corundum: Statically-enforced persistent memory safety. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2021, page 429–442, New York, NY, USA, 2021. Association for Computing Machinery.

[38] Qingda Hu, Jinglei Ren, Anirudh Badam, Jiwu Shu, and Thomas Moscibroda. Log-structured non-volatile main memory. In *2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17)*, pages 703–717, 2017.

[39] Intel. Intel memory latency checker. `https://software.intel.com/en-us/articles/intelr-memory-latency-checker`.

[40] Intel. Intel Optane Memory, 2017. `http://www.intel.com/content/www/us/en/architecture-and-technology/optane-memory.html`.

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

[42] Intel. Intel Architecture Instruction Set Extensions Programming Reference, 2021. `https://software.intel.com/content/dam/develop/external/us/en/documents-tps/architecture-instruction-set-extensions-programming-reference.pdf`.

[43] An introduction to pmemcheck, 2015. `https://pmem.io/2015/07/17/pmemcheck-basic.html`.

[44] 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.

[45] Jungi Jeong, Chang Hyun Park, Jaehyuk Huh, and Seungryoul Maeng. Efficient hardware-assisted logging with asynchronous and direct-update for persistent memory. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 520–532. IEEE, 2018.

[46] Sooman Jeong, Kisung Lee, Jungwoo Hwang, Seongjin Lee, and Youjip Won. Androstep: Android storage performance analysis tool. In *Software Engineering (Workshops)*, volume 13, pages 327–340, 2013.

[47] Sudarsun Kannan, Andrea C Arpaci-Dusseau, Remzi H Arpaci-Dusseau, Yuangang Wang, Jun Xu, and Gopinath Palani. Designing a true direct-access file system with devfs. In *16th USENIX Conference on File and Storage Technologies*, page 241, 2018.

[48] Takayuki Kawahara. Scalable spin-transfer torque ram technology for normally-off computing. *IEEE Design & Test of Computers*, 28:52–63, 2010.

[49] J. Hyun Kim, Young Je Moon, Hyunsub Song, Jay H. Park, and Sam H. Noh. On providing os support to allow transparent use of traditional programming models for persistent memory. *J. Emerg. Technol. Comput. Syst.*, 16(3), June 2020.

[50] Steve Klabnik and Carol Nichols. The rust programming language. `https://doc.rust-lang.org/book/`.

[51] Kenneth C. Knowlton. A fast storage allocator. *Commun. ACM*, 8(10):623–624, October 1965.

[52] KR Krish, Ali Anwar, and Ali R Butt. hats: A heterogeneity-aware tiered storage for hadoop. In *Cluster, Cloud and Grid Computing (CCGrid), 2014 14th IEEE/ACM International Symposium on*, pages 502–511. IEEE, 2014.

[53] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. Strata: A cross media file system. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 460–477. ACM, 2017.

[54] Philip Lantz, Subramanya Dulloor, Sanjay Kumar, Rajesh Sankaran, and Jeff Jackson. Yat: A Validation Framework for Persistent Memory Software. In *USENIX Annual Technical Conference (ATC)*, 2014.

[55] Benjamin C Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Architecting phase change memory as a scalable dram alternative. In *ACM SIGARCH Computer Architecture News*, volume 37, pages 2–13. ACM, 2009.

[56] Cheng Li, Philip Shilane, Fred Douglis, Hyong Shim, Stephen Smaldone, and Grant Wallace. Nitro: A capacity-optimized ssd cache for primary storage. In *USENIX Annual Technical Conference*, pages 501–512, 2014.

[57] Qingrui Liu, Joseph Izraelevitz, Se Kwon Lee, Michael L Scott, Sam H Noh, and Changhee Jung. ido: Compiler-directed failure atomicity for nonvolatile memory. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 258–270. IEEE, 2018.

[58] Qingrui Liu, Joseph Izraelevitz, Se Kwon Lee, Michael L Scott, Sam H Noh, and Changhee Jung. ido: Compiler-directed failure atomicity for nonvolatile memory. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 258–270. IEEE, 2018.

[59] Sihang Liu, Korakit Seemakhupt, Gennady Pekhimenko, Aasheesh Kolli, and Samira Khan. Janus: Optimizing memory and storage support for non-volatile memory systems. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, page 143–156, New York, NY, USA, 2019. Association for Computing Machinery.

[60] Sihang Liu, Korakit Seemakhupt, Yizhou Wei, Thomas Wenisch, Aasheesh Kolli, and Samira Khan. Cross-failure bug detection in persistent memory programs. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1187–1202, 2020.

[61] 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, 2019.

[62] Leonardo Marmol, Mohammad Chowdhury, and Raju Rangaswami. PM: Simplifying Application Usage of Persistent Memory. *ACM Trans. Storage*, 14(4), December 2018.

[63] Amirsaman Memaripour, Joseph Izraelevitz, and Steven Swanson. Pronto: Easy and fast persistence for volatile data structures. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 789–806, 2020.

[64] Micron. 3D XPoint Technology, 2017. `http://www.micron.com/products/advanced-solutions/3d-xpoint-technology`.

[65] Micron. Battery-backed nvdimms, 2017. `https://www.micron.com/products/dram-modules/nvdimm#/`.

[66] 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, 1992.

[67] Dushyanth Narayanan and Orion Hodson. Whole-system persistence. *ACM SIGARCH Computer Architecture News*, 40(1):401–410, 2012.

[68] Ian Neal, Andrew Quinn, and Baris Kasikci. Hippocrates: healing persistent memory bugs without doing any harm. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 401–414, 2021.

[69] Ian Neal, Ben Reeves, Ben Stoler, Andrew Quinn, Youngjin Kwon, Simon Peter, and Baris Kasikci. {AGAMOTTO}: How persistent is your persistent memory application? In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, pages 1047–1064, 2020.

[70] Matheus Almeida Ogleari, Ethan L Miller, and Jishen 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. IEEE, 2018.

[71] Kevin Oleary. How to detect persistent memory programming errors using intel inspector - persistence inspector, 2018. `https://software.intel.com/`.

[72] Oracle Corporation. MySQL. `https://www.mysql.com/`.

[73] Ismail Oukid, Daniel Booss, Adrien Lespinasse, and Wolfgang Lehner. On testing persistent-memory-based software. In *Proceedings of the 12th International Workshop on Data Management on New Hardware*, pages 1–7, 2016.

[74] Victor Pankratius and Ali-Reza Adl-Tabatabai. Software engineering with transactional memory versus locks in practice. *Theory of Computing Systems*, 55(3):555–590, 2014.

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

[76] Pony programming language. `https://www.ponylang.io/`.

[77] Moinuddin K Qureshi, Vijayalakshmi Srinivasan, and Jude A Rivers. Scalable high performance main memory system using phase-change memory technology. *ACM SIGARCH Computer Architecture News*, 37(3):24–33, 2009.

[78] Mendel Rosenblum and John K Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems (TOCS)*, 10(1):26–52, 1992.

[79] Thomas Shull, Jian Huang, and Josep Torrellas. Autopersist: An easy-to-use java nvm framework based on reachability. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, page 316–332, New York, NY, USA, 2019. Association for Computing Machinery.

[80] SQLite. SQLite, 2017. `https://www.sqlite.org`.

[81] Tpc benchmark c. `http://www.tpc.org/tpcc/`.

[82] Haris Volos, Andres Jaan Tack, and Michael M. Swift. Mnemosyne: Lightweight persistent memory. In *ASPLOS '11: Proceeding of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, New York, NY, USA, 2011. ACM.

[83] M Wilcox. Add support for nv-dimms to ext4, 2017.

[84] Xiaojian Wu and AL Reddy. Scmfs: a file system for storage class memory. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, page 39. ACM, 2011.

[85] Cong Xu, Xiangyu Dong, Norman P Jouppi, and Yuan Xie. Design implications of memristor-based rram cross-point structures. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011*, pages 1–6. IEEE, 2011.

[86] Jian Xu and Steven Swanson. Nova: A log-structured file system for hybrid volatile/non-volatile main memories. In *FAST*, pages 323–338, 2016.

[87] 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, February 2016. USENIX Association.

[88] Yuanchao Xu, ChenCheng Ye, Yan Solihin, and Xipeng Shen. Hardware-based domain virtualization for intra-process isolation of persistent memory objects. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 680–692. IEEE, 2020.

[89] J Joshua Yang, Dmitri B Strukov, and Duncan R Stewart. Memristive devices for computing. *Nature nanotechnology*, 8(1):13, 2013.

[90] Zhengyu Yang, Morteza Hoseinzadeh, Allen Andrews, Clay Mayers, David Thomas Evans, Rory Thomas Bolt, Janki Bhimani, Ningfang Mi, and Steven Swanson. Autotiering: automatic data placement manager in multi-tier all-flash datacenter. In *Performance Computing and Communications Conference (IPCCC), 2017 IEEE 36th International*, pages 1–8. IEEE, 2017.

[91] Zhen Yu, Yu Zuo, and Yong Zhao. Convoider: A concurrency bug avoider based on transparent software transactional memory. *International Journal of Parallel Programming*, 48(1):32–60, 2020.

[92] Gong Zhang, Lawrence Chiu, and Ling Liu. Adaptive data migration in multi-tiered storage based cloud environment. In *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on*, pages 148–155. IEEE, 2010.

[93] Yiying Zhang, Jian Yang, Amirsaman Memaripour, and Steven Swanson. Mojim: A reliable and highly-available non-volatile memory system. In *ACM SIGARCH Computer Architecture News*, volume 43, pages 3–18. ACM, 2015.

[94] 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.

[95] Shengan Zheng, Morteza Hoseinzadeh, and Steven Swanson. Ziggurat: a tiered file system for non-volatile main memories and disks. In *17th {USENIX} Conference on File and Storage Technologies ({FAST} 19)*, pages 207–219, 2019.

[96] Benjamin Zorn. Comparing mark-and sweep and stop-and-copy garbage collection. In *Proceedings of the 1990 ACM conference on LISP and functional programming*, pages 87–98, 1990.