UNIVERSITY OF CALIFORNIA SAN DIEGO

Multi-level Methodology for PMEM Data Consistency

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

in

Computer Science

by

Yi Xu

Committee in charge:

Professor Steven Swanson, Chair
Professor Joseph Izraelevitz
Professor Paul H. Siegel
Professor Geoffrey M. Voelker
Professor Jishen Zhao

2023

The Dissertation of Yi Xu is approved, and it is acceptable in quality and form for publication on microfilm and electronically.

University of California San Diego

2023

DEDICATION

To my family, and to those who have shared this joyful and enriching journey with me.

EPIGRAPH

Do not go where the path may lead, go instead where there is no path and leave a trail.

Ralph Waldo Emerson

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I feel very fortunate and grateful for having the support from many wonderful people. This dissertation would not be possible without them.

First of all, I want to express my gratitude to my advisor, Professor Steven Swanson, for always supporting and believing in me. Throughout this incredible journey, he has been a constant source of inspiration and encouragement. I want to thank him for his support during my moments of doubt, for providing me with the space and freedom to explore my own path, for his patience and being a good listener, and for always being there to guide me when I'm uncertain. I have truly enjoyed and benefited from our discussions on research and life. These conversations have been incredibly inspiring, providing me with invaluable insights and helping me become a better person. Thanks to his mentorship, pursuing a Ph.D. has been an unforgettable and joyful experience. I couldn't have imagined a better Ph.D. journey. I feel truly fortunate to have him as my advisor, and I am deeply grateful for the profound impact he has had on my academic and personal growth.

I want to thank my commitee memebers Professor Geoffrey M. Voelker, Professor Jishen Zhao, Professor Paul H. Siegel, and Professor Joseph Izraelevitz for their valuable comments and feedbacks. I want to express my gratitude to Professor Geoffrey M. Voelker. Every interaction with him, be it during class, committee meetings, or non-academic occasions like the wonderful hiking trips he organized, has been an absolute pleasure. I am truly thankful for the enriching conversations we've shared. I also want to specially thank Professor Joseph Izraelevitz. From the very start of my Ph.D., he has been by my side, providing unwavering support and guidance. As a wonderful mentor and teacher, he has played a crucial role in shaping my journey and growth. I am truly grateful for his presence during the initial phase of my Ph.D. journey.

I want to thanks my collaborator George Hodgkins. I also want to thank my colleagues and friends from NVSL and UCSD. I want to thank Amirsaman Memaripour, Jian Yang, Lu Zhang, Morteza Hoseinzadeh, Juno Kim for their support and guidance. I want to thank Zixuan Wang, Suyash Mahar, Narangerelt Batsoyol, Ziheng Liu, Mingyao Shen, Yun Joon Soh, Chen

Chen, Yaxuan Wu, Ke Sun, Renjie Zhao, Yizhou Shan, Haolan Liu, and Zhi Wang. This journey would not be the same without you. I will always miss the time we spend together.

I want to thank my mentors, collaborators and friends from VMware Research and UIUC: Ramnatthan Alagappan, Alex Conway, Aishwarya Ganesan, Rob Johnson, Prashant Pandey, Xundong Sun, Henry Zhu. I appreciate their mentorship and guidance. I truly enjoyed the time spent at VMware Research.

Lastly, I want to express my deepest gratitude to my parents. They have been an unwavering source of support and courage, always encouraging me to pursue my dreams and standing by me in every decision I make. Their love and guidance have shaped me into the person I am today. I cannot adequately put into words how much I love and appreciate them.

Chapter 1, Chapter 2, Chapter 3, and Chapter 6 contain material from "Clobber-NVM: Log Less, Re-execute More", by Yi Xu, Joseph Izraelevitz, and Steven Swanson, which appeared in the Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2021. The dissertation author is the primary investigator and first author of this paper.

Chapter 1, Chapter 2, Chapter 4, and Chapter 6 contain material from "Zhuque: Failure is Not an Option, it's an Exception", by George Hodgkins*, Yi Xu* (*co-first author), Steven Swanson, and Joseph Izraelevitz, which appeared in 2023 USENIX Annual Technical Conference (USENIX ATC 23). The dissertation author is the primary investigator and co-first author of this paper.

Chapter 1, Chapter 2, Chapter 5, and Chapter 6 contain material from "Persistron: Leveraging Persistent Memory in Key-value Stores via a Crash-safe and Durable Cache", by Yi Xu, Ramnatthan Alagappan, Aishwarya Ganesan, Rob Johnson, and Alex Conway, which was submitted for publication at 2024 International Conference on Management of Data. The dissertation author is the primary investigator and first author of this paper.

or commercial advantage.

VITA

2018	Bachelor of Science in Computer Science & Technology, Southeast University
2019	Internship, Oracle
2021	Internship, VMware Research Group
2022	Internship, VMware Research Group
2018–2023	Graduate Student Researcher, University of California San Diego
2022	Candidate of Philosophy, University of California San Diego
2023	Doctor of Philosophy in Computer Science, University of California San Diego

PUBLICATIONS

Yi Xu, Ramnatthan Alagappan, Aishwarya Ganesan, Rob Johnson, Alex Conway. "Persistron: Leveraging Persistent Memory in Key-Value Stores via a Crash-safe and Durable Cache", *submitted to ACM SIGMOD International Conference on Management of Data (SIGMOD 2024)*

George Hodgkins*, **Yi Xu***, Steven Swanson, Joseph Izraelevitz. "Zhuque: Failure is Not an Option, it's an Exception", *2023 USENIX Annual Technical Conference (USENIX ATC 2023)*, Boston, MA, July 2023 (* marks equal contribution).

Yi Xu, Joseph Izraelevitz, Steven Swanson. "Clobber-NVM: Log Less, Re-execute More", *26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2021)*, Virtual, April 2021.

Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour,Yun Joon Soh, Zixuan Wang, **Yi Xu**, Subramanya R. Dulloor, Jishen Zhao, Steven Swanson. "Basic Performance Measurements of the Intel Optane DC Persistent Memory Module", *Arxiv*, March 2019.

ABSTRACT OF THE DISSERTATION

Multi-level Methodology for PMEM Data Consistency

by

Yi Xu

Doctor of Philosophy in Computer Science

University of California San Diego, 2023

Professor Steven Swanson, Chair

Persistent memory (PMEM) allows direct access to persistent storage via a load/store interface. It promises to realize a vision of high performance, data persistence, a simple programming interface, low cost with minimal storage overhead. Previously, processor caches backed by PMEM were volatile, complicating the design of persistent applications and reducing their performance. The new generation of systems with flush-on-fail semantics provides persistent caches, offering the potential for much simpler, faster PMEM programming and execution models.

PMEM programming systems provide the means to apply sets of writes to persistent states atomically. Unfortunately, most of these systems impose significant overhead and are not

easy to use. Moreover, most existing approaches to incorporating PMEM realize only a subset of the benefits of PMEM.

This dissertation first presents Clobber-NVM, a failure-atomicity library that ensures data consistency by reexecution. Clobber-NVM's novel logging strategy, *clobber logging*, records only those transaction inputs that are overwritten during transaction execution. Then, after a failure, it recovers to a consistent state by restoring overwritten inputs and reexecuting any interrupted transactions. Clobber-NVM utilizes a clobber logging compiler pass for identifying the minimal set of writes that need to be logged. Based on our experiments, Clobber-NVM provides up to 2.5× performance improvement over existing solutions.

Second, it introduces Whole Process Persistence (WPP), a new programming model for systems with persistent caches. In the WPP model, all process state is made persistent. On restart after a power failure, this state is reloaded and execution resumes in an application-defined interrupt handler. We describe the Zhuque runtime, which transparently provides WPP by interposing on the C bindings for system calls in userspace. It requires little or no programmer effort to run applications on Zhuque. Our measurements show that Zhuque significantly outperforms state-of-the-art PMEM libraries. More important, unlike existing systems, Zhuque places no restrictions on how applications implement concurrency, allowing us to run a newer version of Memcached on Zhuque and gain more than 7.5× throughput over the fastest existing persistent implementations.

Finally, it presents PERSISTRON, a key-value store that exploits PMEM effectively with almost no change to the storage layer. Our main observation is that most key-value stores employ a cache to avoid expensive access to storage. By moving the cache layer to PMEM and using regular cache-management functions to manage PMEM, most code is untouched, and the cache is potentially reusable for other applications. We have implemented PERSISTRON by modifying SplinterDB. PERSISTRON can improve query performance by up to 46% over a cost-equivalent all-DRAM configuration of SplinterDB.

Chapter 1

Introduction

Persistent memory (PMEM), whether it be based on Intel Optane [56] or flash-backed DRAM over CXL [100], presents new opportunites and challenges for storage systems. PMEM exposes persistent storage as fast, byte-addressable main memory, and allow the processor to access persistent data via load and store instructions directly using the memory bus. PMEM provides instantaneous durability: writes can be persisted almost immediately, allowing in-memory data to live beyond process lifetimes and even across system reboots and unexpected power failures. PMEM is also likely to be approximately an order of magnitude cheaper per gigabyte than DRAM [68].

These unique characteristics present the opportunity to build storage systems which are:

1. Fast;
2. Simple, because serialization and deserialization can be avoided between CPU and persistent domain;
3. Cost-effective, because DRAM can be replaced with cheaper PMEM;
4. Capacity-efficient, because applications no longer need to maintain redundant copies of data in both main memory and persistent storage;
5. Instantly durable, because writes can be immediately persisted in PMEM without requiring IOs to external storage devices.

Building a system that realizes the promise of persistent programming is not simple.

One of the main challenges to obtaining these potential benefits is the complexity of retrofitting existing code to use PMEM efficiently. PMEM is slower than DRAM, so simply replacing all DRAM with PMEM will result in a slower system overall. Furthermore, PMEM is not likely to be cheap enough to replace flash in large-scale storage systems, and therefore PMEM is likely to be a new component of the memory hiearchy.

Moreover, as caches are volatile, the contents of CPU caches do not survive power loss, and, since caches may delay evicting a modified cache line, writes may not reach PMEM in program execution order. These limitations mean that in the case of an unexpected power loss, a set of logically atomic updates may be torn with only a subset of them reaching PMEM, leaving persistent data in an inconsistent state. This makes reasoning about the state of memory after a crash extremely challenging.

In order to ensure persistent data consistency in the case of power failures, programmers must design applications carefully. Custom solutions use cache line flush instructions to force data from volatile caches to PMEM in a consistent order. However, building applications on the low-level, cache-line-based primitives the ISA provides is difficult.

Over the past decades, many solutions have been proposed to handle data consistency during power failures. These solutions span the spectrum, ranging from custom log-free data structures to user-level ”transactional” programming libraries, and to key-value stores. These solutions hide different complexities from programmers and offer various interfaces. However, none of these methods have been proven to fully leverage the potential benefits of PMEM.

In this dissertation, we explore three design points that could potentially expose PMEM in a more efficient and easier-to-use way for upper-level applications in future computer systems.

Systems should be capable of handling failures, but failure-atomicity costs should minimized, in terms of both programming and performance. We should minimize the runtime costs and incur a significant portion of the recovery cost only at recovery time —- when it is truly needed.

Programming systems (e.g., libraries, programming models, language support, and

compilers) aim to help address the challenges of generic persistent memory programming. They have proliferated over the last decade. Broadly, three families of systems have emerged: each takes a different approach to consistency, and each faces significant challenges which bar widespread adoption.

The first and largest family [123, 112, 97, 127] requires programmers to access persistent state only through well-defined atomic operations (often called transactions). This provides a clean notion of consistency: after recovery from crash, each atomic section has either executed entirely or not at all, meaning that all writes within a specified code region will survive a power loss and become persistent in PMEM, or none will. These transaction-like, "all-or-nothing" semantics make programming on PMEM easier and hide architectural and caching details from programmers.

However, like all transactional memory models, this approach suffers from serious weaknesses: it is fundamentally incompatible with non-transactional synchronization, and has never gained significant traction in real systems.

Moreover, although failure-atomicity libraries give programmers the ability to designate failure-atomic code regions (transactions), this support comes with high performance overhead. Most industrial failure atomicity systems [97, 11] use an undo logging approach. In undo logging systems, where incomplete transactions are undone, the logging of the old value must occur before each write. Undo-based systems have a significant advantage in that reads do not need to be redirected via an interposition layer, but at the cost of many expensive persistence ordering fences [91].

To avoid the high logging cost, JUSTDO logging [61] proposed *recovery-via-resumption*. In contrast to undo logging, JUSTDO logging tracks enough program state (including the program counter) to *resume* a failure-atomic operation at recovery, resuming execution from the interrupted instruction. Subsequent work in iDO logging [85] dramatically increased performance by exploiting regions of code that are idempotent (a segment of code that does not overwrite its inputs). However, the runtime overhead of these systems remains quite high.

The second family of PMEM programming systems [12, 54, 61, 85] uses FASEs, regions of code protected by locks, as atomic regions for PMEM updates. Legacy code can run with minimal changes, but these systems suffer from fundamental weaknesses arising from complex locking schemes and external IO. As we will show, addressing these weaknesses either cripples the system or essentially reduces it to a transaction-based system.

The final family of systems takes the more dramatic step of making *everything* in the system persistent via whole-system persistence (WSP) [92]. WSP provides the conceptually simplest programming model: Nothing much changes and, from the program's perspective, crashes never occur. WSP faces two major challenges: First, making all of memory persistent has until recently been infeasible, because regularly flushing volatile caches to PMEM creates enormous performance overheads. Second, making *everything* persistent would require a far-reaching redesign of many system components, for an unclear benefit.

The generic PMEM programming systems promises to realize a vision of high performance, data persistence, a simple programming interface, and low storage overhead at the same time. However, they are agnostic of the application characteristics. In addition, their performance and capacity are often constrained by the limitations of PMEM.

As a result, researchers are investigating designs for multi-level hierarchies that incorporate DRAM, PMEM, and SSD in key-value stores. Most current approaches either exploit PMEM's immediate persistence or large capacity, but not both. More concretely, one approach statically decides to move some structures (e.g., the memtable [67], the log [125], or the upper levels of an LSM [125]) in to PMEM. These systems mutate these structures directly on PMEM, thus achieving immediate persistence. However, this approach does not store other (potentially large and frequently accessed) portions of data in the PMEM, forgoing capacity benefits. In contrast, the second approach uses PMEM as an extended read-only cache to augment the existing DRAM block cache, neglecting PMEM's persistence. For example, recent work on RocksDB at Facebook [68] uses PMEM as an extended volatile cache; data in PMEM is discarded after a crash. HYMEM and Spitfire [130, 110] exploit PM capacity and persistency at the same time,

but both require substantial code changes.

In this dissertation, we address the failure recovery problem at the transaction level, process level, and caching system level, respectively.

In Chapter 3, we introduce Clobber-NVM, a PMEM library that ensures failure-atomicity by reexecuting interrupted transactions. Clobber-NVM relies on a new logging method — *clobber logging*, which merges undo logging with recovery-via-resumption. Clobber logging undo logs any transaction inputs overwritten during transaction execution. If a transaction is interrupted by a failure, clobber logging recovers by first restoring the transaction's overwritten inputs then, subsequently, reexecuting the transaction to completion. This strategy results in our system requiring far less logging than traditional undo-based systems since it only logs a few inputs — in our experiments we reduce the log sizes by 1.1× to 42.6×, and the log count (ordering fences) by 2.4× to 4.7×.

In Chapter 4, we propose that WSP-style persistence is due for a renaissance: The advent of PMEM devices and platforms supporting *flush-on-fail* semantics (e.g. eADR for NVDIMMs or GPF for CXL devices) allows developers to treat caches as effectively persistent [60, 27], removing the main performance argument against WSP. Further, we believe that limiting the scope of persistence to a process – yielding *Whole Process Persistence (WPP)* – and providing well-defined, application-level semantics for system failures combine to produce a programming model that is fast, flexible enough to support legacy programs and complex locking schemes, and easy for programmers to use and understand.

WPP provides a simple abstraction to the process: its entire memory is persistent and will survive a power outage. If a power outage occurs, the process receives an OS signal after restart notifying it of the crash. The process can install a normal error handler for this signal which cleans up and exits, or performs more complex application-specific recovery; by default, program execution simply continues at the point of failure.

We describe Zhuque runtime, which provides WPP. Zhuque is faster than existing PMEM programming systems. It is between 4.7× and 10.14× faster than PMDK [97], Mnemosyne [112],

Atlas [12] and Clobber-NVM [123] on STAMP applications. Zhuque is also more flexible than these systems: Since Zhuque is agnostic about the application's locking scheme, it can run the most recent version of memcached, while those systems cannot. As a result, our Zhuque-based persistent memcached is more than 7.5× faster than any similar system. We also demonstrate Zhuque's flexibility by running unmodified Python benchmarks with minimal performance loss.

In Chapter 5, we present PERSISTCACHE, a grey-box DRAM/PMEM caching system that requires only small changes to storage layer code in order to achieve the speed, cost, and durability advantages of PMEM-based systems.

PERSISTCACHE combines a straightforward copy-on-write DRAM/PMEM cache and migration policy with hints from the application that enable it to avoid copy-on-write overheads when they are not needed for crash safety. Thus almost all code changes are in the cache, with only a few hints required from the application code.

The core idea behind the design of PERSISTCACHE is that the cache layer is an effective place to introduce PMEM support into an existing storage system. Most storage systems employ a DRAM cache, so this approach should be broadly applicable without major code changes. Implementing PMEM support at the cache layer enables us to use both DRAM and PMEM to create a large cache and to decide which pages live in PMEM versus DRAM, which makes it possible to ensure durability and to avoid PMEM overheads when possible. Furthermore, by ensuring persistence at the cache layer, we can provide instant durability and avoid the overheads of logging.

To ensure that the contents of the PMEM cache always remain consistent, even after a crash, PERSISTCACHE employs a *COW-based transaction mechanism*. No or little key-value store code needs modification to use this transaction mechanism: PERSISTCACHE infers transaction boundaries from lock acquisition and release points.

PERSISTCACHE enables high performance via two techniques: *selective transactional updates* and *DRAM caching*. Naively using transactions for all modifications can be prohibitively expensive. PERSISTCACHE allows intermediate storage updates that don't require crash safety

(such as building new data structures that aren't yet linked into the main storage data structure) to be performed non-transactionally, which reduces overhead. This is implemented by having the storage system provide hints to the caching layer when transactions are not needed.

Second, while PMEM is faster than SSD, it is still considerably slower than DRAM; thus, it is hard to match the performance of a DRAM-only cache. To solve this problem, PERSISTCACHE uses a (generally smaller) DRAM cache in addition to the PMEM cache. Hot, clean pages are opportunistically moved to the DRAM cache for better read performance. Warm pages are kept around in PMEM, reducing disk accesses.

With this approach, PERSISTCACHE presents a new design point to leverage PMEM effectively. While many prior papers propose radically new ways to manage PMEM, we take a contrarian view: we realize that PMEM can be incorporated into the cache layer, and standard cache-management functions can be used to leverage this new hardware effectively. This view has great practical utility: it delivers high performance while keeping code change to other parts of the application to a minimum.

We have implemented PERSISTCACHE by modifying the cache layer of SplinterDB, a production key-value store that offers high performance. The implementation is mostly self-contained in the caching layer. SplinterDB uses a log-structured merge tree (LSM) as its internal data structure. To integrate with the rest of SplinterDB, we added application-specific hints to mark non-transaction boundaries, such as compactions, and we also reworked one function that needed to be crash safe but could not be easily expressed as a transaction (see Chapter 5.5). We call the resulting key-value store PERSISTRON.

In our experiments, we compare PERSISTRON on a hybrid PMEM/DRAM cache to SplinterDB on a dollar-equivalent all-DRAM cache. PERSISTRON insertion throughput is up to 8% faster than in SplinterDB and query throughput is up to 46% faster, demonstrating that PERSISTRON is able to exploit the increased capacity of PMEM without suffering from its lower performance relative to DRAM. Furthermore, PERSISTRON provides instant durability, whereas SplinterDB does not. We also evaluated against MatrixKV and found that PERSISTRON

insertions were 3.5× faster than in MatrixKV and queries were up to 65% faster.

Acknowledgements

This chapter contains material from ”Clobber-NVM: Log Less, Re-execute More”, by Yi Xu, Joseph Izraelevitz, and Steven Swanson, which appeared in the Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2021. The dissertation author is the primary investigator and first author of this paper.

This chapter contains material from ”Zhuque: Failure is Not an Option, it’s an Exception”, by George Hodgkins*, Yi Xu* (*co-first author), Steven Swanson, and Joseph Izraelevitz, which appeared in 2023 USENIX Annual Technical Conference (USENIX ATC 23). The dissertation author is the primary investigator and co-first author of this paper.

This chapter contains material from ”Persistron: Leveraging Persistent Memory in Key-value Stores via a Crash-safe and Durable Cache”, by Yi Xu, Ramnatthan Alagappan, Aishwarya Ganesan, Rob Johnson, and Alex Conway, which was submitted for publication at 2024 International Conference on Management of Data. The dissertation author is the primary investigator and first author of this paper.

Chapter 2

Background

PMEM has introduced new possibilities for designing storage systems: programs can have byte-addressable access to terabytes of persistent data at near-DRAM latencies. However, utilizing PMEM in a both performant and programmer-friendly manner remains a challenging problem.

This chapter provides necessary background on PMEM and clarifies assumptions about our target systems. We describe the expected machine models.

2.1 Traditional PMEM Machine Model

Clobber-NVM is designed for a modern machine equipped with some persistent memory (e.g. Intel Optane DC PMM's [59] or persistent CXL.mem devices [100]). This multicore, cache-coherent machine contains a set of processing cores, with private and shared write-back caches. The caches contain a program's current working set, loaded from the backing memory, which consists of both persistent memory and traditional DRAM. Stores issued by the cores modify cache lines in the caches; subsequent cache line evictions write the modified data back to either volatile (DRAM) or persistent memory, depending on the cache line's physical address. The hardware manages the cache line eviction policy, it may not evict cache lines in the same order they were modified. Moreover, writes are not persistent as long as they reside in the volatile cache.

The existence of volatile caches with uncontrolled eviction policies means that the programmer needs to reason about the order in which data updates reach persistent memory. For example, in a stack `push()` operation, the node must be created *and made persistent* before pointed to by the `top` pointer. Otherwise the `top` pointer could be evicted from the cache and become persistent, while its target node, still in the volatile caches, could be lost in a power outage, leaving behind a dangling, persistent, pointer.

To avoid these inconsistencies, programmers must use cache-flush and memory-barrier instructions to enforce the ordering of stores into PMEM. For example, on Intel CPUs, the `clflush` or `clwb` instructions explicitly force a dirty cache line into memory, and `sfence` ensures subsequent writes will not complete until previous flushes that issued before the `sfence` have reached memory. However, frequent ordering fences limit the overlapping of long-latency flush instructions, result in high runtime overhead [50].

We expect hybrid machines with both PMEM and DRAM in the near term. It is important for the programmer to specify whether the manipulated memory is volatile or not. Programmers can mark regions of the program's address space persistent, and these memory regions are associated each with some named file in a PMEM-aware file system [119, 115, 18]. In the event of a failure, the file system can remap the file into another process's address space for recovery and further use.

2.2 flush-on-fail PMEM Machine Model

WPP and PERSISTRON are designed for a multi-core, cache-coherent machine equipped with PMEM (e.g. Intel Optane DC PMM's [59] or persistent CXL.mem devices [100]), and supporting *flush-on-fail* semantics, meaning that they provide a hardware guarantee that all in-flight and cached writes will reach PMEM in the event of an external power failure (as opposed to a fault in the machine or its onboard power supply). Such guarantees are provided by eADR-compliant platforms and NVDIMMs, and CXL platforms and devices supporting

Global Persistent Flush (GPF). eADR and GPF are similar solutions targeting different device interfaces: the primary hardware requirement for both is that the platform must store sufficient energy to allow caches and internal device buffers to be drained to persistence after a power failure [1, 100].

On x86 systems, both eADR and GPF require system firmware to initiate and oversee the drain to persistence in response to a System Management Interrupt (SMI) [1, 27, 26]. Upon receiving this interrupt, the processor retires all in-flight instructions, drains all stores to the cache, and saves architectural state (register file etc.) to a designated per-core memory region before beginning execution of the SMI handler [29]. In both GPF and eADR, this handler first flushes the processor caches (and, for CXL, the caches of any CXL.cache device), and then proceeds to flush the buffers on the PMEM devices (NVDIMMs for eADR, CXL.mem devices for GPF) [1, 27].

Applications still need to ensure updates are crash-consistent, but do not necessarily need to evict cachelines to enforce update persistency.

Acknowledgements

This chapter contains material from ”Clobber-NVM: Log Less, Re-execute More”, by Yi Xu, Joseph Izraelevitz, and Steven Swanson, which appeared in the Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2021. The dissertation author is the primary investigator and first author of this paper.

This chapter contains material from ”Zhuque: Failure is Not an Option, it’s an Exception”, by George Hodgkins*, Yi Xu* (*co-first author), Steven Swanson, and Joseph Izraelevitz, which appeared in 2023 USENIX Annual Technical Conference (USENIX ATC 23). The dissertation author is the primary investigator and co-first author of this paper.

This chapter contains material from ”Persistron: Leveraging Persistent Memory in

Key-value Stores via a Crash-safe and Durable Cache", by Yi Xu, Ramnatthan Alagappan, Aishwarya Ganesan, Rob Johnson, and Alex Conway, which was submitted for publication at 2024 International Conference on Management of Data. The dissertation author is the primary investigator and first author of this paper.

Chapter 3

Log Less, Re-execute More

3.1 Background

This chapter provides necessary background and clarifies assumptions about our target systems. Clobber-NVM is designed for a modern machine equipped with both PMEM and DRAM, with volatile CPU cache. We begin by describing the expected programming model, then provide necessary concepts for our program analysis.

3.1.1 Programming Model

Clobber-NVM is one of many *failure-atomicity libraries* that provide a simpler programming model for PMEM: we describe this model here. With these libraries, a programmer can designate a region of code to be *failure-atomic*, that is, all of the code region's effects will survive a failure (e.g. power loss), or none will. Once the effects of the code region are guaranteed to survive a crash, the operation is *committed*. In general, in order to ensure failure-atomicity, it is necessary to log extra information during normal execution to support *recovery* after a failure. Recovery code can then use this extra, *logged* information to clean up interrupted failure-atomic operations and return back to a consistent state.

In the literature, these failure-atomic code regions (or *operations*) are often termed *failure-atomic sections (FASEs)*, or *transactions*. Generally speaking, the boundaries of the transaction are programmer defined using some interface of the underlying failure atomicity

runtime. Note that these transactions are not traditional ACID (Atomicity, Consistency, Isolation, Durability) transactions [49, 46] — depending on the system, they may or may not provide isolation. That said, many failure-atomicity libraries link failure atomicity to concurrency control, either by building full ACID semantics or marking lock-protected critical sections as failure-atomic regions.

Clobber-NVM is a recovery-via-resumption failure-atomicity system. In these systems, interrupted transactions are resumed; this strategy stands in contrast to more traditional undo [97, 11] and redo [113, 54, 83] logging based systems, where interrupted failure-atomic updates roll back. Recovery-via-resumption requires saving sufficient program state such that resumption is possible.

Clobber-NVM uses an interface of transactions conceptually compatible with Intel's PMDK [97] library. Following this interface, we expect the programmer to explicitly mark failure atomic transactions. Furthermore, following PMDK's concurrency model, we expect transactions to acquire and release locks in a conservative, strong strict two-phase locking pattern [98, 114]: that is, locks protect memory locations from data races; transactions acquire the associated lock at transaction begin and in a fixed order (to prevent deadlock); transactions hold the locks until transaction commit. Assuming the programmer follows these constraints, both PMDK and Clobber-NVM transactions provide true ACID semantics.

3.1.2 Program Analysis Definitions

Clobber-NVM's design relies on compiler dependency analysis to minimize the amount of logging needed for proper reexecution and recovery of interrupted transactions. We describe key concepts required for this analysis here.

A Clobber-NVM *transaction* is a programmer-delineated code region with a single entry and (possibly) multiple exits. Following standard terminology [32], a code region (resp. transaction) *output* is a variable value which is assigned within the region and is *live-out* from its exit. That is, a region output is a value written in the region and read after it. Similarly, we define

a code region (resp. transaction) *input* to be a variable value that is *live-in* to the region and used within the region. That is, a region input is a value assigned before the region's execution and read within it. Note that both inputs and outputs refer to values, not variables (a natural consequence of LLVM's SSA-based [41, 39] program represention).

A code region is *deterministic* if, for a given set of inputs, it always produces the same set of outputs. Clobber-NVM expects its transactions to be deterministic, and to not cause runtime errors or program exits (e.g., segmentation faults). This assumption is universal for recovery-via-resumption systems [61, 85, 3].

3.2 Clobber Logging Design

Clobber logging is a recovery strategy that minimizes logging calls and log size by recording overwritten inputs to a transaction. If the transaction is interrupted, recovery proceeds by restoring the overwritten inputs then reeexecuting the transaction. Clobber logging's key insight is that logging only overwritten inputs is sufficient to reexecute a transaction with the exact same results, and, as a consequence, values at other addresses never need to be logged, since they will be overwritten upon reexecution.

Following the concepts introduced in Chapter 3.1.2, we deem a transaction input a *clobbered input* if it may be overwritten within this transaction, and term this write a *clobber write*. Clobbered inputs are a problem for reexecution. If an input is clobbered during transaction execution, reexecuting the code will use a new value for the input.

3.2.1 Undo-Then-Reexecute

To understand how clobber logging can ensure failure atomicity, we first describe a naive version of clobber logging that ignores dependency analysis. We term this explanatory failure-atomicity strategy *undo-then-reexecute*.

At every store, undo-then-reexecute records an undo log entry containing the old, overwritten value. On recovery, it replays the undo log backwards, leaving erasing all effects of the

transaction. Instead of stopping at this point (as conventional undo logging would), undo-then-reexecute then reexecutes the transaction from start to finish.

Undo-then-reexecute differs from classical undo logging in a few notable ways. First, it recovers the program state to a point after the interrupted transaction, as opposed to before. Second, once started, a transaction never rolls back, so a transaction can be marked as committed soon as it begins. Finally, undo-then-reexecute assumes that inputs unmodified by the transaction will be available for reexecution during recovery. In our machine model, this assumption does not hold for data in DRAM or shared with other threads. Inputs in volatile memory will disappear during a power failure, and shared inputs might change prior to reexecution.

A correct version of undo-then-reexecute must address these challenges: First, it must make a persistent copy of volatile inputs so they are available after restart. Second, it must adapt a concurrency model that prevents transaction inputs from being changed after being read and transaction outputs from being read before commit (e.g. through its locking scheme).

3.2.2 Improving Performance

Undo-then-reexecute and undo logging are both slow since they log the same information at every store. For a PMEM library to optimize undo-then-reexecute, it should attempt to remove extraneous logging. First, it should understand what logging is truly necessary in order to ensure that reexecution gives the same result. Second, it should understand what logging operations are unnecessary because it will reexecute the transaction.

To ensure reexecution of a transaction gives the same result, the transaction needs to be deterministic and have the same inputs (arguments and memory state) as the previous interrupted execution. This is why the undo-then-reexecute strategy needs the "undo" step: it needs to revert the transaction's clobbered inputs to their unmodified state.

However, undo logging records more than the clobbered transaction inputs: it also records the old values before writes to the transaction output addresses (e.g. modified memory locations). Traditional undo logging relies on these records to roll back the transaction's changes. But

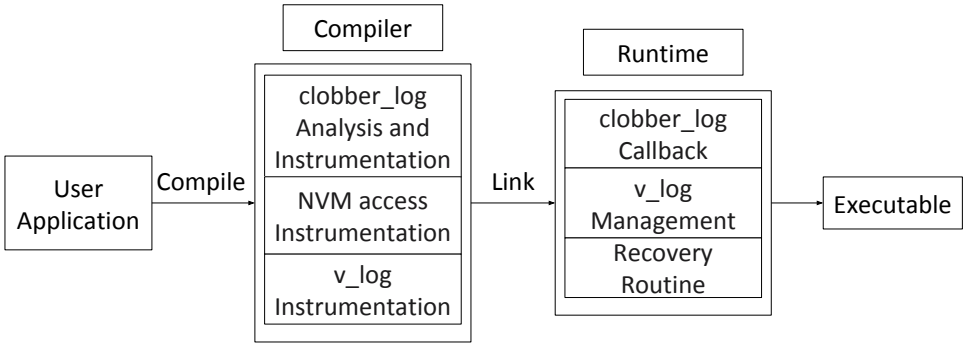


Figure 3.1. Clobber-NVM system overview

undo-then-reexecute does not need to undo its changes. Because it will reexecute the transaction, and the transaction will write the same values to the same memory locations, any written data from the previous execution will be regenerated and overwritten.

3.2.3 Clobber Logging

The clobber logging strategy is simply this: undo-then-reexecute, but *only* undo log before clobber writes. Like any correct undo-then-reexecute strategy on a persistent memory system, clobber logging needs to preserve volatile inputs and adapt an appropriate concurrency scheme. Clobber logging optimizes undo-then-reexecute by logging the clobbered inputs and ensuring all non-clobber inputs are available during recovery. Then, it reexecutes the transactions to produce the correct results. As we will show, clobber logging require less logging and incurs lower runtime overheads than conventional undo logging.

3.3 Clobber-NVM Implementation

The Clobber-NVM failure-atomicity system is a joint compiler/runtime library. Key system components are shown in Figure 3.1.

Our compiler extension, built on top of LLVM [76, 41], is principally used to identify clobber writes within transactions using dependency analysis. Due to aliasing, this identification requires significant reasoning about dependency chains. After this analysis, the compiler adds callbacks to both clobber writes and to memory accesses within transactions to automate logging

and recovery.

Our runtime library manages the program during execution and recovery. It catches callbacks inserted by the compiler and programmer and directs the accesses to the appropriate logs. The runtime manages two logs: the `clobber_log`, which logs clobbered inputs, and the `v_log`, which logs volatile transaction inputs (inputs that reside either on the stack or in the volatile heap and would not be available during recovery. Usually these are function arguments).

The runtime will also initiate recovery of interrupted transactions after a crash. Recovery, for each transaction, proceeds by first restoring volatile state from the `v_log` and restoring clobbered state from the `clobber_log`, then reexecuting the interrupted transaction using the restored inputs.

To create a Clobber-NVM program, a developer needs to write transactions in a manner that meets our programming model, compile using our compiler with its extensions, and link to our runtime library. Recovering a Clobber-NVM program is done by restarting the program.

3.3.1 Using Clobber-NVM

Clobber-NVM's C programming model is designed to be easily used — most of its annotations and requirements have equivalents in Intel's PMDK library.

Figure 3.2(a) shows an example transaction written for Clobber-NVM, which executes a persistent list insertion (the equivalent PMDK C and C++ code are also shown). First, note the Clobber-NVM transaction is isolated within a function (line 1), termed the `txfunc`. Clobber-NVM's compiler instruments the call-site to collect the function name and its arguments within the `v_log`, and the function provides a convenient handle to initiate reexecution.

Upon entering the function, the appropriate locks are acquired on both persistent data (the list) and the volatile data (the new value). As with PMDK, transactions must be synchronized using conservative, strong strict two-phase locking for proper recovery [102]. We use the `txbegin` macro to start the actual transaction (line 4). As we intend to use a volatile non-local pointer within the transaction, we must record it using the `vlog_preserve` macro (line 5). With

```
1  void plist_ins(plist* lst, char* v,
2   size_t vsz, lock* v_lk){
3    lock(lst->lk); lock(v_lk);
4    txbegin();
5    vlog_preserve(v,vsz);
6    pnode* n = pmalloc(sizeof(pnode));
7    n->val = pmalloc(strlen(v));
8    strcpy(n->val, v);
9    n->nxt = lst->hd;
10   // lst->hd is a clobbered input
11   // and will be clobber logged
12   lst->hd = n;
13   txend();
14   unlock(lst->lk); unlock(v_lk);
15 }
```

(a) Clobber-NVM

```
16 void plist_ins(TOID(plist) lst,
17  char* v, lock* v_lk){
18   lock(lst->lk); lock(v_lk);
19   TX_BEGIN(pop){
20     TOID(struct pnode) n =
21       TX_NEW(struct pnode);
22     D_RW(n)->val =
23       TX_NEW(strlen(v));
24     TX_ADD_FIELD(n, val);
25     strcpy(D_RW(n)->val, v);
26     TX_ADD_FIELD(n, nxt);
27     D_RW(n)->nxt = D_RO(lst)->hd;
28     TX_ADD_FIELD(lst, hd);
29     D_RW(lst)->hd = D_RO(n);
30   }TX_END {}
31   unlock(lst->lk); unlock(v_lk);
32 }
```

(b) PMDK C

```
33 void plist::ins(char* v,
34  lock* v_lk){
35   auto p = pool_by_vptr(this);
36   transaction::run(p,[this,v]{
37     auto n =
38       make_persistent<pnode>(v);
39     strcpy(n->val, v);
40     n->nxt = this->hd;
41     this->hd = n;
42   },this->lk,v_lk);
43 }
```

(c) PMDK C++

Figure 3.2. List insert operation using PMDK and Clobber-NVM

locks acquired and volatile inputs recorded, we can execute the transaction.

At the transaction's beginning, we allocate memory from PMEM using the `pmalloc` interface (lines 6 and 7). Note that on line 12, we will change the list head to the new node, thereby clobbering a transaction input — this clobbered input will be identified automatically by the compiler, and then recorded by the runtime. Also note that Clobber-NVM does not require special macros when accessing persistent memory — the appropriate callbacks are added by the compiler. Assuming no power loss, the transaction will terminate and commit with the `txend` macro (line 13), and release its locks (line 14).

3.3.2 Runtime and Callbacks

Clobber-NVM's runtime manages the program's persistent state during execution and recovery. This task requires it to manage a few internal structures and to catch all necessary callbacks, added by both the user and the compiler.

The user is responsible for invoking four callbacks. The `txbegin` and `txend` macros inform the runtime of a starting or committing transaction, triggering a persistent update of the transaction's status. The `pmalloc` macros informs the runtime to allocate memory from PMEM, instead of DRAM. As the application's semantics decide which sets of writes should happen failure-atomically and which memory allocation should allocate from PMEM, the compiler is unable to assist with these callbacks. The `vlog_preserve` macro informs the runtime that some volatile non-local pointer input will be used during transaction execution — this input must be preserved persistently in the `v_log`. As compiler analysis is necessarily incomplete with respect to the transaction's read set, and as the appropriate `v_log` callbacks must occur at transaction begin, the developer must provide this information.

The compiler inserts other callbacks handled by the runtime. The first type of callback occurs at possible clobber write sites. This callback triggers a logging action in the runtime to record the clobbered inputs in its `clobber_log`. The second callback occurs at every memory access, and allows the runtime to appropriately swizzle pointers to support relocatable backing

PMEM storage. The final callback occurs on the top of `txfuncs` — this callback is used to record soon-to-begin transaction, collects the function name and its arguments.

Our runtime is implemented over Intel's PMDK v1.6 libpmemobj — it replaces the transaction, logging, allocation, and recovery management but preserves the user-facing interface for PMEM region management and crash detection. In particular, Clobber-NVM's runtime manages two key logging systems: the `clobber_log`, which holds clobbered inputs, and the `v_log`, which holds both volatile inputs and tracks ongoing transaction state. Every ongoing transaction maintains one log `v_log` entry.

Our `clobber_log` is built over PMDK's undo log API. This design choice leaves Clobber-NVM's `clobber_log` very simple, and provides an additional benefit — as PMDK's performance improves, so does Clobber-NVM.

In contrast, the `v_log` is directly managed by the runtime. We manage the per-thread `v_log` using a global linked list resident in persistent memory, and allocate it on thread creation. The thread will use this log to manage its (at most one) active transaction. Using both the `vlog_preserve` macro and the compiler instrumented callback on entry into the `txfunc`, the log records the function arguments, function name and additional needed volatile data in the log at transaction begin. We use a single bit in each `v_log` to decide if re-execution is necessary on its corresponding thread. When the transaction begins, it is marked as ongoing, and the bit is cleared at commit.

3.3.3 Recovery

Clobber-NVM recovers PMEM data to a consistent state via re-execution. We here describe the Clobber-NVM's recovery process.

Figure 3.3(top row) shows a transaction progressing through normal execution. At transaction begin (top left), its inputs are already initialized and its outputs have not yet been touched. As the transaction executes (Figure 3.3, top center), it reads inputs, from both PMEM and DRAM, and writes to output addresses to both PMEM and DRAM. Note that some writes to

output addresses may be clobber writes and will overwrite some inputs. During normal execution, the transaction will progress to completion and commit (Figure 3.3, top right), completely writing all outputs.

A transaction interrupted by a crash follows a different path. As with normal execution, the transaction starts with initialized inputs and untouched outputs (Figure 3.3, top left), then progresses through execution by writing to some output addresses (Figure 3.3, top center), including clobber writes. However, a power failure during execution drops the transaction to the recovery path.

After the power loss, the transaction loses all volatile memory and some PMEM values that still resided in the machine's (volatile) caches (Figure 3.3, bottom left). At restart, Clobber-NVM first detects if there are any uncommitted, ongoing transactions using the per-thread `v_logs`. Using the transaction's logs, Clobber-NVM restores both the volatile and clobbered inputs, though the values at output addresses may still be inconsistent (Figure 3.3, bottom center).

With its inputs restored, the transaction is ready to restart (Figure 3.3, bottom right). Clobber-NVM recovers each thread independently — a valid strategy since our locking scheme ensures all ongoing transaction lock sets are disjoint. To recover a transaction, the corresponding `txfunc` is called, reexecuting the transaction from the beginning. Once the transaction executes past the point when the failure happened, the transaction has overwritten any incomplete values, erasing any inconsistencies caused by the power loss (Figure 3.3, top center). The transaction will continue to progress to completion and commit (Figure 3.3, top right).

3.3.4 Compiler

Clobber-NVM compiler identifies clobber writes and insert other utility callbacks.

The first compiler pass identifies writes within a transaction that may clobber an input, then instruments the `clobber_log` callback before the writes happen. This pass relies on classic alias analysis to identify clobber writes. However, this basic alias analysis is not necessarily precise and may over identify clobber writes, though this is a performance, not a safety issue.

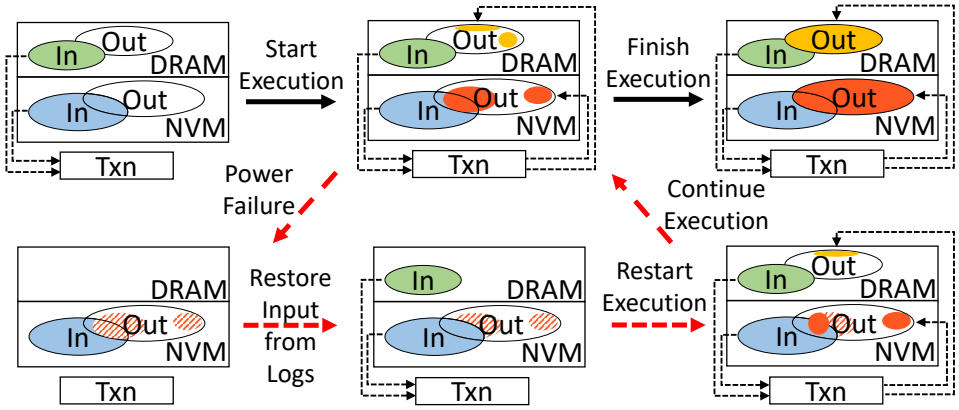


Figure 3.3. Recovery process of one transaction. The *In* and *Out* indicates input and output addresses within both DRAM and PMEM.

The pass subsequently refines the result through novel analysis propagation.

Then the second pass adds callbacks to all memory accesses: these callbacks allow the system to intercept accesses to PMEM and, as necessary, swizzle the pointers to redirect the accesses to the relocatable backing region. Finally, the third pass instruments the code for recovery: it adds `v_log` callbacks to the `txfunc` to record their names and arguments.

Clobber-NVM's compiler is built on top of LLVM [76, 41]. We use clang [37] as the frontend compiler to translates C/C++ code to LLVM IR, and introduce the three passes [40] described above to the LLVM compiler toolchain. All passes operate on LLVM IR [39].

In the remainder of this subsection, we describe our clobber write identifying pass in detail, starting with the conservative implementation, then describing our iterative refinement.

Conservative Clobber Writes Identification Clobber-NVM's conservative clobber write identification follows two steps. The first step identifies *candidate input reads*, that is, reads that could conceivably be the first operation on a value. The second step uses these reads to find *candidate clobber writes*, that is, writes that could conceivably overwrite an input. Note that both steps are conservative — all possible input reads and all possible clobber writes will be candidates.

The identification pass relies on LLVM alias analysis [38] to identify candidate input reads and clobber writes. Alias analysis produces pair-wise results that indicate two memory accesses (1) cannot, (2) may or (3) must point to the same location.

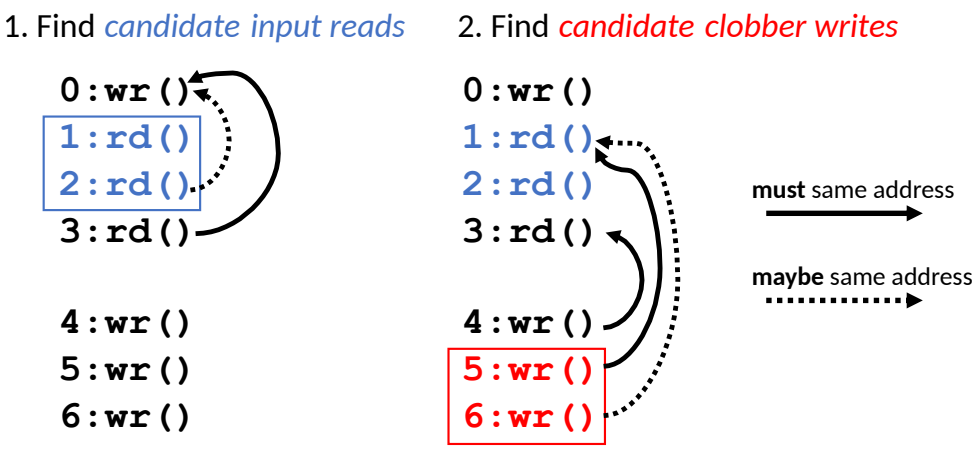


Figure 3.4. Conservative candidate clobber write identification

Using alias analysis, the first step traverses all reads in the transaction, searching for candidate input reads. Reads that are *dominated* by an earlier write (all paths to the read first execute the write), and whose dominating write must modify the same address, cannot be candidate input reads. All other reads are labeled as a candidate input read. Figure 3.4(left) shows this process.

In the second step, the compiler identifies candidate clobber writes for each candidate input read. Candidate clobber writes include all *successor* writes (writes that may be executed after the input read) that may write to the same address as the input read. Figure 3.4(right) shows this identification.

At this point, the compiler has conservatively identified all clobber writes, but some candidates may never overwrite an input. Figure 3.4(right) shows an example of such a candidate clobber write on line 6: the input will already be clobbered on line 5.

Dependency Analysis Propagation To reduce Clobber-NVM's logging cost, we perform additional dependency analysis propogation to remove candidate clobber writes that, upon further analysis, can never be true clobber writes. We target two general types of *false clobber candidates*.

The first type of false clobber candidate we term *unexposed*. This false candidate may indeed be the first write to overwrite a candidate input, but, if it does — it is provable the input candidate is a false input. The scenario is shown in Figure 3.5(left). In this scenario, the

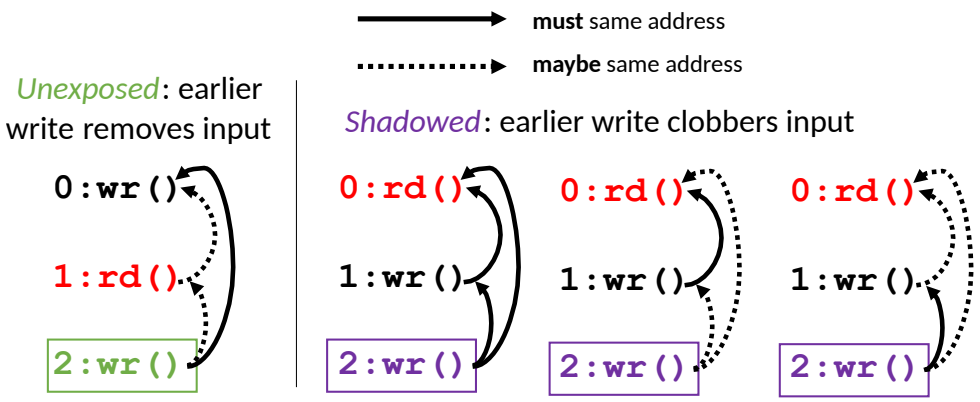


Figure 3.5. Removing false candidate clobber writes

candidate input read is dominated by some earlier write. Since the earlier write and the read may not access the same location, the read is considered a candidate input. The subsequent write may also access the read's location, and so it is considered to be a clobber candidate. However, both writes are guaranteed to access the same location through alias analysis — as a consequence, if the later write overwrites the read, the read cannot be an input (it will be dominated by the earlier write).

The second type of false clobber write candidate is a candidate which is *shadowed* by some earlier clobber write. A shadowed candidate may indeed overwrite an input, but if it does, it is guaranteed that some earlier write already clobbered it. This relationship requires two conditions. First, it requires some earlier write to dominates the shadowed write. Secondly, the alias relationship between the input read, earlier write, and shadowed write must ensure that if the shadowed write does overwrite the input, the earlier write will have first. There are three alias combinations between the three accesses that meet these criteria: they are shown in Figure 3.5(right). In practice, this kind of false candidate occurs often in loops: the first iteration clobbers the input, but subsequent iterations do not need to log.

Our analysis searches for both types of false clobber candidates by iterating over every pair of a candidate clobber write and its corresponding input read, and then looking for an additional write that would then form one of these four cases.

3.4 Evaluation

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

- How much improvement does Clobber-NVM provide for persistent data structures compared to other libraries?
- What is the reason for Clobber-NVM's high performance?
- What is Clobber-NVM's recovery overhead compared to PMDK?
- What is Clobber-NVM's performance on application-level code?
- How does the underlying data structure of an application affect its performance when build with Clobber-NVM?
- What is the effectiveness of Clobber-NVM compiler optimization?
- How much longer does it take for an application to compile with Clobber-NVM, compared to Clang?

Our experimental workloads include four data structure benchmarks and three recoverable applications.

3.4.1 Evaluation Setup

We compare against three popular PMEM libraries with Clobber-NVM.

PMDK [97] is Intel's failure atomicity library. Its later versions use hybrid undo-redo logging techniques [57]. The technique is based on a combination of undo logging for modify a group of memory atomically and redo logging for memory allocation and deallocation [103]. Based on our experiments, PMDK v1.6 generally provides the best performance among all available versions — we show this version in all experiments.

Atlas [11] is another undo-log system that allows for complicated locking schemes within failure-atomic regions. It uses lock operations to infer failure-atomic operation boundaries. Due to its weak concurrency requirements, Atlas tracks dependencies between failure atomic

operations and is prepared to rollback even completed operations — this dependency tracking incurs signficant runtime cost [61].

Mnemosyne [113] is a redo-log based system. Unlike the other systems, it uses the C++ transactional memory model to parallelize code, instead of using locks.

Prior recovery-via-resumption systems (JUSTDO [61] and iDO [85]) do not have publically available implementations. To compare Clobber-NVM's performance with their's, we implemented a compiler instrumentation pass to collect iDO's transaction information.

We run the benchmarks on a platform with two 24-core Intel Cascade Lake SP processors, running at 2.2 GHz. The platform has a total of 192 GB of DRAM and 1.5 TB (6 ×256 GB) of Intel Optane DC Persistent Memory directly attached to each processor [59]. We configured our test machine such that Optane DCPMM is in 100% App Direct mode [2]. In this mode, software has direct byte-addressable access to the Optane DCPMM. All experiments use `Ext4` to manage persistent pools and directly access PMEM pages via DAX [82].

3.4.2 Data Structure Benchmarks

In our first experiment, we compare Clobber-NVM's throughput with comparison systems on data structure benchmarks:

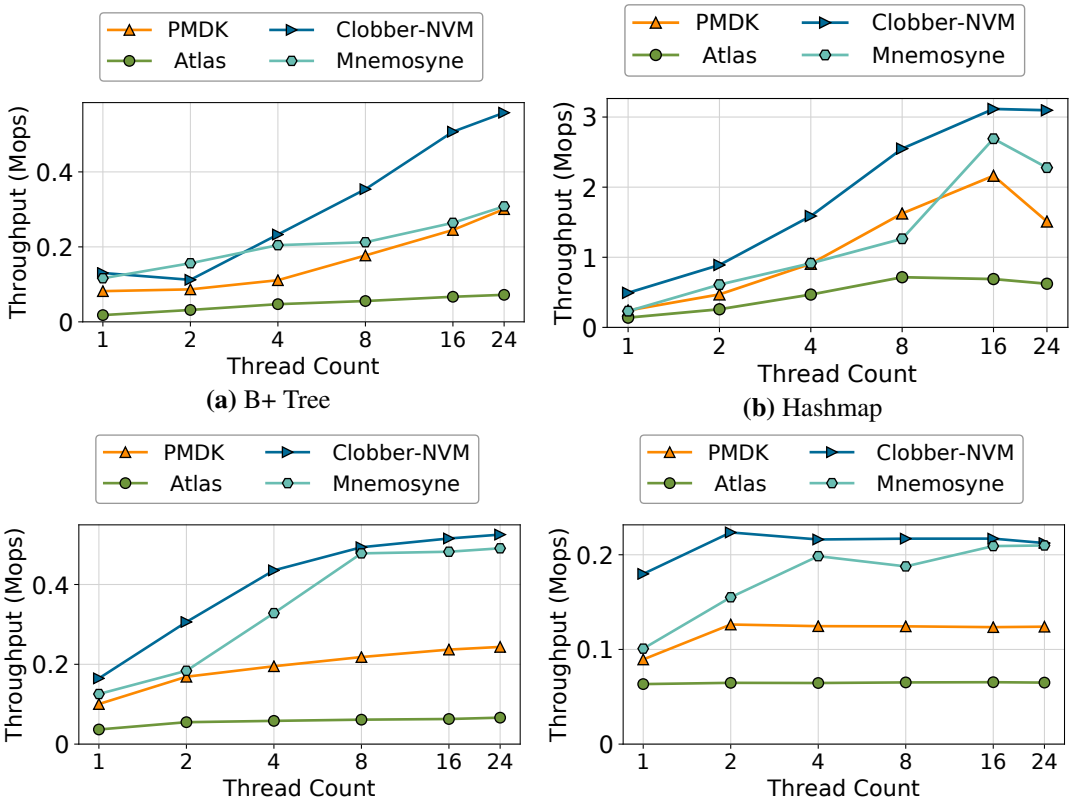
B+ Tree uses reader-writer locks at the granularity of individual nodes, stores keys in the internal nodes, and adds both the key and the value to the leaf nodes.

HashMap is adapted from the PMDK repository [58]. We create 256 instances of the HashMap, treat each one as a bucket, and protect each bucket with a reader-writer lock.

Skiplist is a skiplist with 32 levels. We use a single global lock for the entire data structure.

Red-Black Tree is implemented in accordance with the version in Linux kernel. We use a global reader-writer lock for the tree.

On all data structures except B+ Tree, we insert key-value pairs with key size 8 bytes and value size 256 bytes. On B+ Tree, the inserted key size is 32 bytes. The benchmark runs



(a) B+ Tree **(b)** Hashmap

(c) Red-black Tree **(d)** Skiplist

Figure 3.6. Measuring the throughput of different PMEM libraries: each data structure is scaled up to 24 thread. Clobber-NVM shows better performance over other libraries on all data structures

YCSB [24] workloads against different versions of the data structures. It populates each structure with 1 million entries (YCSB-Load workload).

We make the following observations. Firstly, for single thread, because undo-log entries are immediately followed by `fence` instructions, the number of log entries imposes more latency on Clobber-NVM and undo-log systems compared to redo-log systems like Mnemosyne. Hashmap insertion is relatively simple. So Clobber-NVM shows over 2.13× performance of Mnemosyne on hashmap benchmark. B+ Tree has the longest transaction. Mnemosyne can provide performance comparable to Clobber-NVM, despite it actually logs much more data.

Secondly, Clobber-NVM always outperforms undo log systems significantly on one thread. PMDK undo log shares the same logging subsystem with Clobber-NVM `clobber_log`. But it usually does significantly more undo log entries. Therefore, Clobber-NVM shows 1.82× of its performance on average, and can perform up to 2× compare to it on hashmap and skiplist. Atlas has to track dependencies between transactions, which imposes significant runtime overhead. On average, Clobber-NVM provides 4× of Atlas performance.

Thirdly, on multithread workloads, scalablility of Clobber-NVM and other lock-based system is mostly determined by the locking scheme. It is the programmer's responsibility to use finer granularity locks, in order to achieve good scalablility. For example, Clobber-NVM shows the best scalablility on B+ Tree among all data structures, since it uses per node granularity locks. Clobber-NVM provides 1.8× of Mnemosyne performance, 6.6× of Atlas performance, and 1.9× of PMDK performance. Since PMDK and Clobber-NVM rely on the same underlying lock scheme, they scale similarly across all data structures. Clobber-NVM outperforms PMDK by over 1.9× across all data structures with 24 threads. On data structures with a single global lock, Mnemosyne scales better than Clobber-NVM and PMDK, result in matching Clobber-NVM performance on rbtree and skiplist with 24 threads.

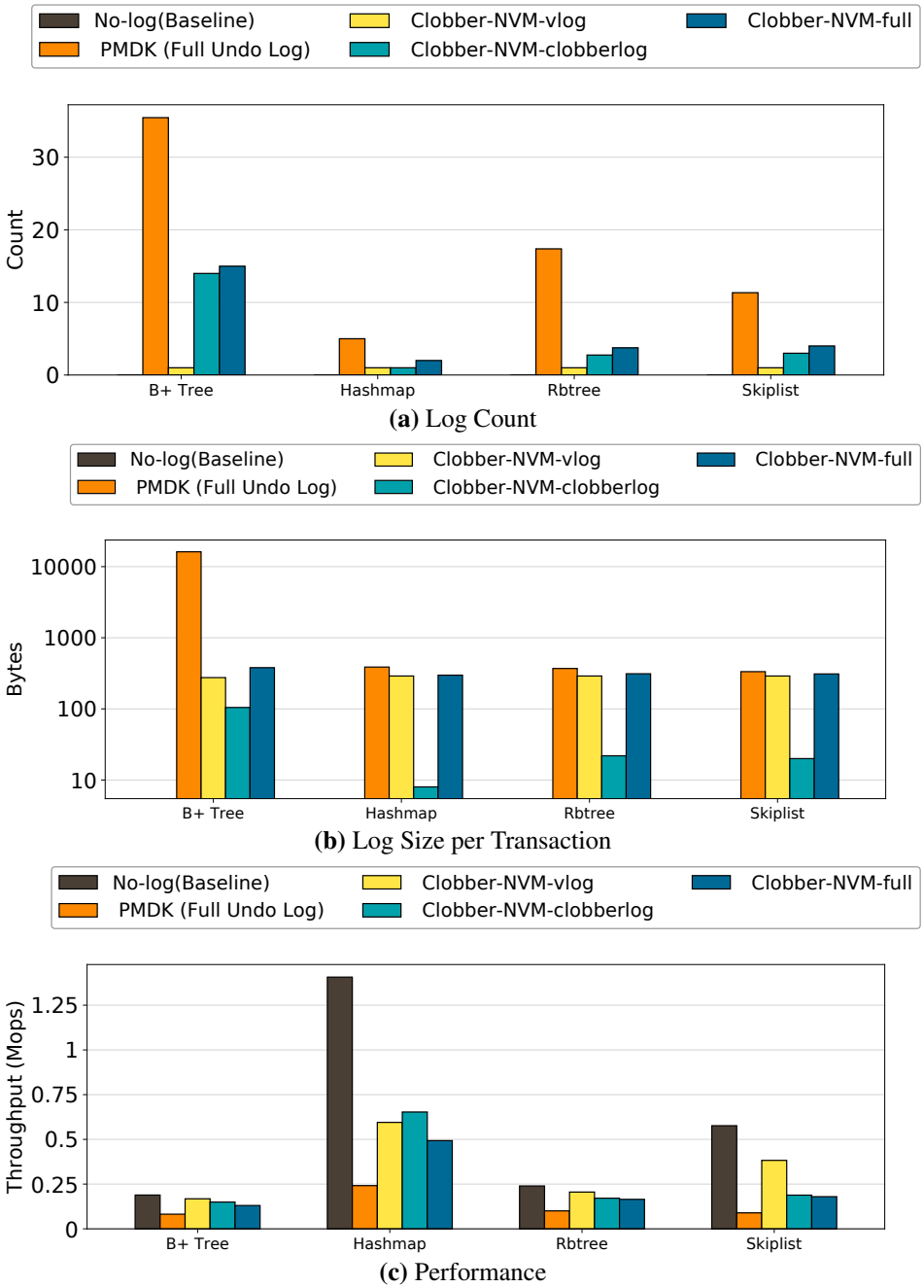


Figure 3.7. Measuring the overhead of different logging strategies: different log count and log size result in different performance. The operated key-value pair has key size 8 bytes (32 bytes for B+ Tree), and value size 256 bytes.

3.4.3 Performance Breakdown

In this experiment, using the same data structure benchmarks, we incrementally enable different logs in Clobber-NVM and PMDK (full undo log) to understand where the performance costs of Clobber-NVM reside.

No-log is the baseline performance, where the data structures do not do any logging. **Clobber-NVM-vlog** only does `v_log` in Clobber-NVM. **Clobber-NVM-clobberlog** is Clobber-NVM that only enables `clobber_log`. These three systems are not failure-atomic. **Clobber-NVM-full** represents the full version of Clobber-NVM. It does both `v_log` and `clobber_log`. **PMDK** shows PMDK's performance. It does full undo log for each transaction.

Here, we use one thread to insert key-value pairs from YCSB benchmark. Figure 3.7 shows the measured logging overhead on different data structure. In our evaluated workloads, Clobber-NVM-vlog always have one log entry per transaction. Clobber-NVM-clobberlog typically use 15.8% to 39.5% as many log entries as PMDK per transaction, Clobber-NVM, in total, uses 21.5% to 42.3% as many log entries as PMDK. Regarding the log size, PMDK requires 16.7× to 154.5× more bytes compared to Clobber-NVM-clobberlog. PMDK's log size is 1.2× to 58× of Clobber-NVM-vlog's log size, and 1.1× to 42.6× of Clobber-NVM's log size, depending on the data structure.

The comparison between Clobber-NVM-vlog and PMDK shows that `v_log` is very cheap due to its implementation. On all data structures, a great portion of log bytes are used in `v_log` (more than 70%), but Clobber-NVM-vlog's performance is comparable to No-log on most data structures. The high performance of `v_log` comes from two perspective — The `v_log` entry count is always one for the whole transaction, result in only two necessary `fences`. And the pre-allocated `v_log` buffer made it much faster compared to traditional undo log entry.

Most of Clobber-NVM's overhead is imposed by `clobber_log`. Generally speaking, fewer log entries and smaller log size result in better performance. And log entry count usually matters more than log size, which is consistent with the fact that a `fence` is usually more

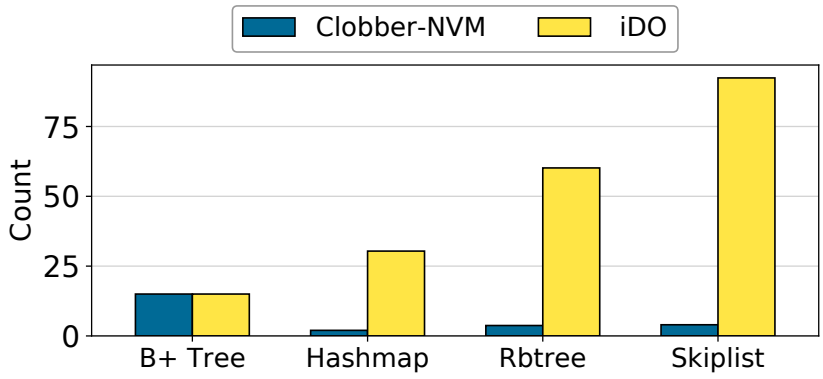
expensive than a `flush`.

The only exception is the hashmap. Its Clobber-NVM-vlog version performs 10% slower than its Clobber-NVM-clobberlog version, indicating that most of Clobber-NVM's overhead is caused by `v_log` on the hashmap insertion benchmark. On this benchmark, its `clobber_log` log count is one, and its log size is 8 bytes. Because the `v_log` log size is much larger than the `clobber_log` log size, `v_log` component causes more latency than `clobber_log` component on the hashmap insertion.

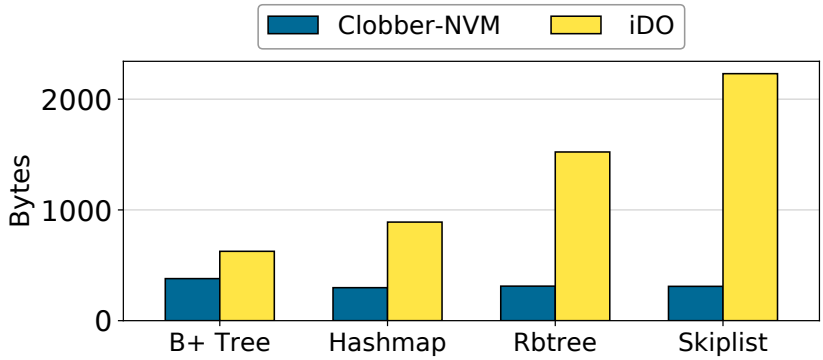
3.4.4 Comparison to iDO

iDO is a state-of-the-art recovery-via-resumption library. iDO's compiler pass breaks transactions into a series of idempotent regions, logging at the boundary between idempotent regions — failure during an iDO transaction triggers the reexecution of the idempotent code region, followed by the resumption of the remainder of the interrupted transaction. Because its code is not publically available, we reimplemented a compiler instrumentation pass to instrument the code, and collect transaction information as iDO would have.

iDO will always have the at least as many bytes persisted per transaction as Clobber-NVM. iDO's strategy of logging at the boundary of idempotent code regions requires larger log entries than Clobber-NVM — the log entries at each boundary require a snapshot of most registers, plus a flush and fence for any modified memory location. The strategy generally also incurs more logging calls than Clobber-NVM's logging of clobber writes. Clobber-NVM only needs to log when a write clobbers a transaction input, whereas iDO logging needs to log whenever a write clobbers an idempotent region's input (all transaction inputs are included in some region's input, but not all region inputs are transaction inputs as they may be intermediate state local to the transaction). Additionally, iDO places the program stack in persistent memory to avoid the need of manually copying stack variables into the persistent heap on FASE initialization [85] (the purpose of our `v_log`), but consequently needs to track accesses to stack variables during transaction execution and log them if necessary.



(a) Log Count



(b) Additional Bytes Persisted

Figure 3.8. Clobber-NVM and iDO log size per transaction

As shown in Figure 3.8, iDO not only requires more logging points, it also persists significantly more data. It logs 1× to 23× more frequently compared to Clobber-NVM, depending on the specific data structures and workload. On average, iDO logs 4.2× more bytes than Clobber-NVM, and it logs up to 7.2× more bytes on skiplist benchmark.

3.4.5 Recovery Overhead

In this experiment, we compare Clobber-NVM's recovery overhead with PMDK's recovery overhead on the same four data structures. Clobber-NVM's recovery process is composed by three steps. Firstly, it opens the persistent pool. Secondly, it applies the `clobber_log` entries to their corresponding addresses. Lastly, it reads `v_log` and re-executes the interrupted transactions based on valid `v_log` entries. PMDK share the first two steps with Clobber-NVM, but instead of applying `clobber_log` on selected addresses, it reads undo log entries of uncommitted transactions, and rolls back the transaction by rewriting the values in undo log to all pmem variables

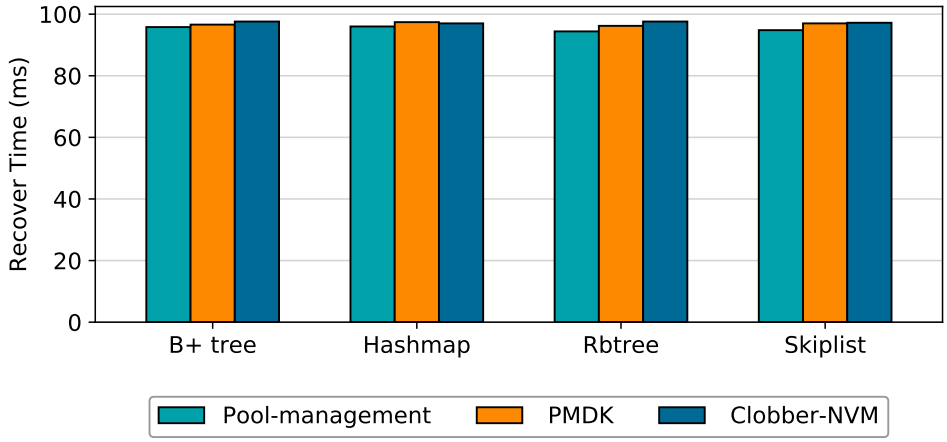


Figure 3.9. Recovery overhead on different data structure

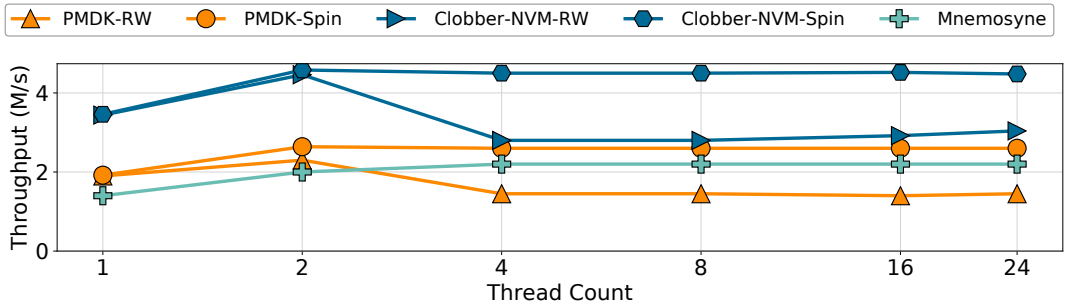
that were updated in the transactions.

We randomly crash the data structure benchmarks, and measure the average recovery overhead. As shown in Figure 3.9, the recovery latency of Clobber-NVM and PMDK are similar. Most of their recovery latency is spent on pool managements. PMDK shows marginally lower recovery overhead on bptree, rbtree and skiplist, but Clobber-NVM recovers slightly faster on hashmap. When most of writes in a interrupted transaction are not clobber writes, Clobber-NVM would have less and shorter `clobber_log` entries recorded. And if the latency of re-execution on the interrupted transaction is small, Clobber-NVM is likely to have lower recovery overhead.

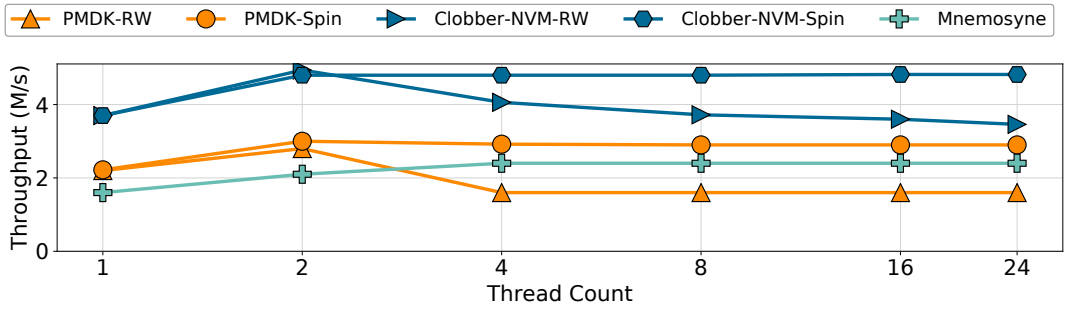
3.4.6 Memcached

Memcached [87] is a production-quality key-value store. It is already integrated with Mnemosyne [90], and we modified its volatile version to use both PMDK and Clobber-NVM. Memcached is consists of a server side and a client side. We used the tool memslap [79] as the client to generate a stream of Memcached requests according to a desired distribution. We used 4 client threads, which generated requests with uniformly distributed 16-byte keys and 64-byte values.

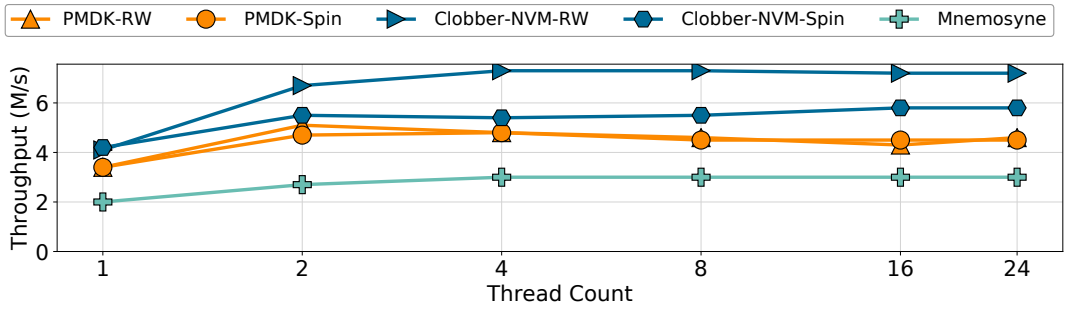
We experimented with 4 types of workloads: insertion-intensive (95% insertion / 5% search), insertion-most (75% insertion / 25% search), search-most (25% insertion / 75% search), and search-intensive (5% insertion / 95% search). As shown in Figure 3.10, Clobber-NVM



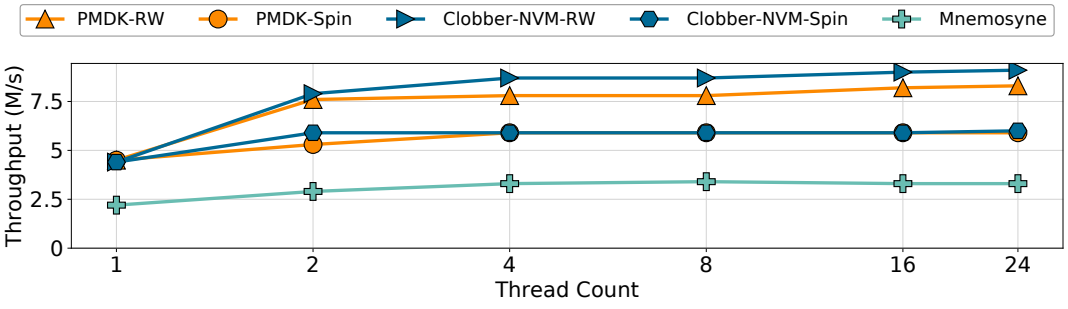
(a) 95% Insert, 5% Search



(b) 75% Insert, 25% Search



(c) 25% Insert, 75% Search



(d) 5% Insert, 95% Search

Figure 3.10. Memcached Performance on Different Workloads and Threads

outperforms PMDK and Menmosyne on all workloads. Clobber-NVM's `clobber_log` entry count is always smaller than redo/undo log entry count. And the per transaction `v_log` is less expensive. It outperforms PMDK and Menmosyne more on more insert intensive workloads, because PMDK and Menmosyne require more log entries. On more search intensive workloads, more operations do not involve logging mechanisms. Clobber-NVM's performance gain is smaller. But the longer read path of redo-log based system result in lower performance of Menmosyne compared to both Clobber-NVM and PMDK.

Clobber-NVM provides up to 2.5× of Menmosyne's throughput and 1.8× of PMDK's throughput on single thread workloads. However, older versions of memcached were notorious for exhibiting poor scaling due to coarse-grain locking [34, 85]. Therefore, we replace the exclusive lock in its original code with spinlock and reader-writer lock. As expected, spinlock works better for insert-intensive workloads, and reader-writer lock provides better scalablility for search intensive workloads.

3.4.7 Vacation

We also evaluated the STAMP benchmark suite's vacation application [17] performance with Clobber-NVM, PMDK and Mnemosyne. Vacation simulates the transactions of a travel agency, and transactions span several tables simulating travel booking reservations.

Vacation is consists of four tables. Similar to prior implementations [50] [113], we persist the tables in persistent memory, and leave the client threads in volatile memory. The tables are originally implemented on red-black trees. Here, we also replace it with an AVLtree implemented in the STAMP suite [17] to show the applications performance on a different underlying data structure. The database has 100000 records of each reservation item. The workload is consisted by 99% of item reservation of cancellation, and the rest create or destroy items. We adjust the number of queries per task to create different workloads. Again, we use **No-log** as the baseline.

Figure 3.11 shows that No-log, PMDK, and Clobber-NVM performs 17%, 9% and 7% better on avltree version compared to the red-black tree version, indicating that the undo log

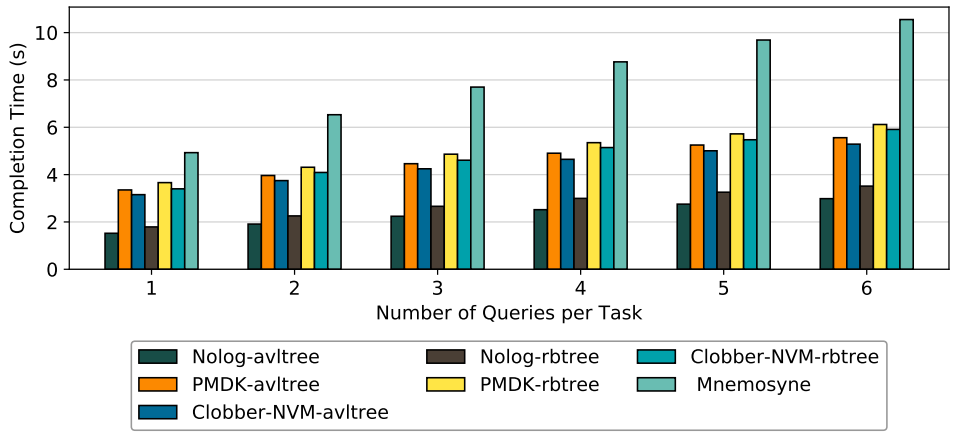


Figure 3.11. Vacation performance on different data structure

entries (and `clobber_log` entries) are data structure dependent, but the `v_log` entries in vacation are the same across different underlying data structures.

Because the number of queries per task indicate the propotion of read in one transaction, the logging overhead of PMDK and Clobber-NVM decreases when the number of queries per task increases. When the query number is six, PMDK and Clobber-NVM's overhead is 74% and 68%, respectively. We also find that, `v_log` size increases as the read propotion increases. Therefore, Clobber-NVM outperforms PMDK more with less number of queries per task. Since Mnemosyne is a redo-log based system, its logging overhead increases as the number of queries per task increases. Its overhead compared to the No-log baseline increases from 176% to 200%.

3.4.8 Yada

Yada, also from the STAMP suite [17], is a volatile mesh refinement application. It implements Ruppert's algorithm for Delaunay mesh refinement [101]. The input mesh is refined so that it has a certain minimum angle. We use the data files *ttimeu10000.2* provided in the STAMP suite as input, which is consisted of 19998 elements. Here, we compare Clobber-NVM performance with PMDK performance.

Again, **No-log** is the baseline performance in which the application is running without any logging mechanisms. We persist the graph that stores all the mesh triangles, the set that contains the mesh boundary segments, and the task queue that holds the triangles that need to be

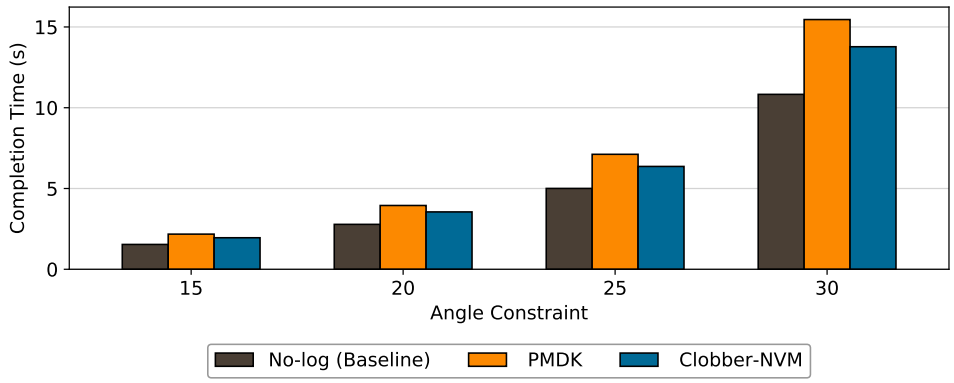


Figure 3.12. Yada Performance on Different Angle Constraining

refined. We set the angle constraining from 15 degrees to 30 degrees, and show each version of Yada performances.

On all angle constraints, PMDK imposes about 42% overhead compared to No-log version. Clobber-NVM reduces the overhead to about 27%. Because Yada is more compute intensive compared to key-value stores. The logging overhead on Yada is low. Therefore, the potential optimization space of Clobber-NVM is small.

3.4.9 Optimization Effectiveness

Clobber-NVM compiler passes identify potential clobber writes. In order to reduce the overhead introduced by conservative identification, Clobber-NVM performs additional dependency analysis propagation to remove false clobber candidates, as introduced in Chapter 3.3.4. We show performance improvement by avoiding redundant `clobber_log` on the four data structures and three applications in Figure 3.13.

On the four data structures, skiplist shows the most performance improvement of up to 15%. We find that the compiler pass removes two clobber candidates out of five, end up requiring only three `clobber_log` entries per transactions. On memcached workloads, the one consisted by 95% write and 5% read requests improves the most, to 14%. As can be expected, the avoided `clobber_log` entries are on the write request path. The unoptimized version incurs up to 32% more `clobber_log` entries and 47% more bytes. Among the three STAMP applications, Yada shows the most performance improvement. It improves its performance by 2.4%, and reduces its

`clobber_log` frequency by 36%.

The effectiveness of compiler optimization pass is application and workload dependent, we expect one application to benefit from the optimizations when:

- Its transaction is long, and is consisted mostly by writes.
- Its memory accesses follows certain patterns. For example, it updates and reads an address at each iteration of a loop.
- Its latency is mainly imposed by the logging.

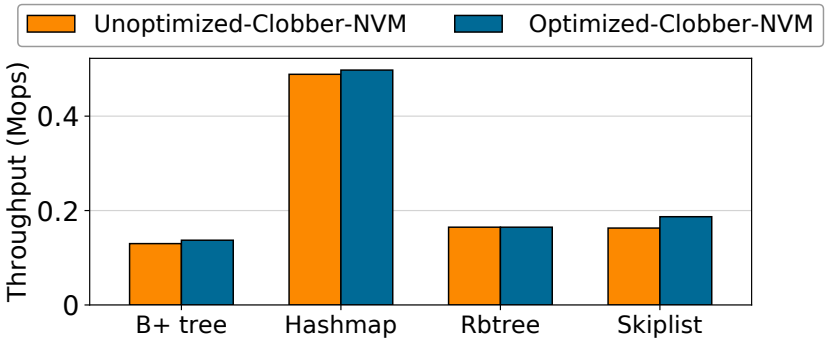
3.4.10 Compile Time Overhead

Clobber-NVM relies on compiler analysis and instrumentation. In this experiment, we show the compile time overhead of Clobber-NVM. We compare the compilation latency of Clobber-NVM with Clang-7.0.0. Figure 3.14 shows the compile latency of four data structures and three applications.

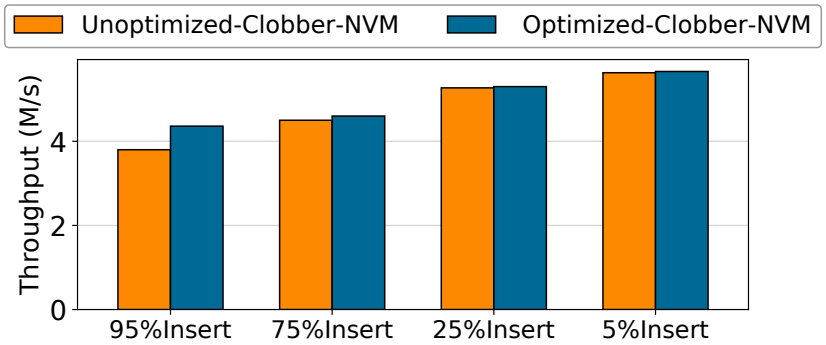
The compile overhead is similar among four data structures, Clobber-NVM adds 29% latency on average. Clobber-NVM takes 55% longer than Clang on memcached. The higher compilation latency is because we compile all files of memcached projects with Clobber-NVM compiler, while only compile the files that has pmem accesses in data structure benchmarks. With more accurate identifications of memcached pmem accesses, the compilation latency is expected to be lower. The STAMP applications also show higher compilation overhead. Since in these applications, pmem accesses are spread across relatively more files, the code analysis and instrumentation also takes longer.

3.5 Related Work

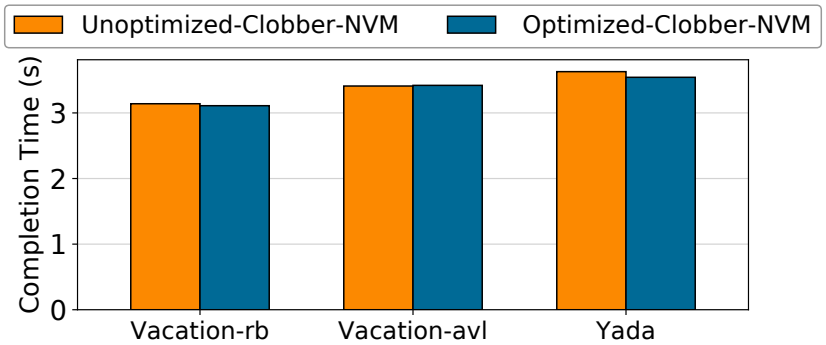
Researchers have proposed many PMEM-optimized data structures [21, 94, 42, 124, 14, 111], and the architecture community has been working on better hardware support for failure-atomicity [70, 69, 7, 44, 31, 107]. Recently, the research community has also focused on using persistent memory for specific applications [81, 15]. For example, researchers [81]



(a) Data structures



(b) Memcached



(c) STAMP applications

Figure 3.13. Optimization effectiveness on data structures and applications

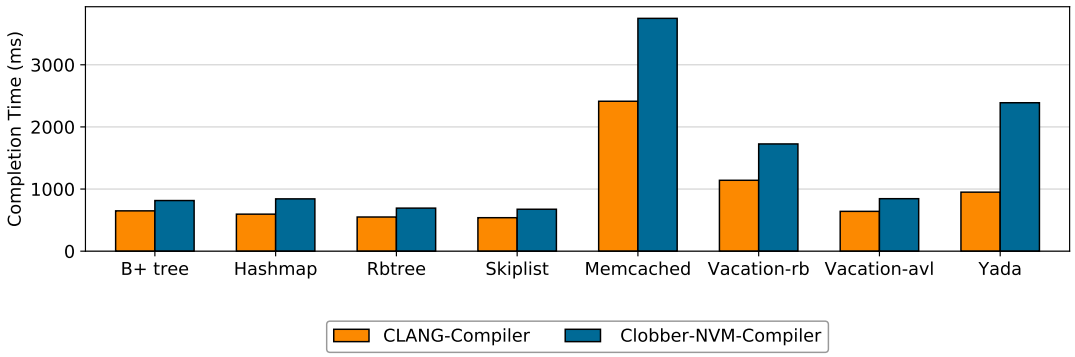


Figure 3.14. Compile latency on data structures and applications

proposed creating failure atomic GPU kernels by leveraging idempotence to reduce logging cost, taking advantage of the necessary copies used for GPU kernel execution. However, high efficiency general purpose libraries based on commodity hardware is still an open research area. Undo log approach includes NV-heaps [19], Atlas [11], and PMDK-v1.4 [97]. Mnemosyne [113] SoftWrAP [43] and NVthread [19] relies on redo logs. Undo-logging systems usually requires expensive ordering fences at least proportional to the number of contiguous data ranges modified in each transaction. In contrast, redo-logging implementations generally require fewer ordering fences regardless of the transaction size. However, their load interposition and load redirection to updated PMEM addresses slows down read speed and increases system complexity.

A number of systems were proposed to optimize redo/undo systems by maintaining a shadow copy of working set during runtime. Kamino-Tx [86] relies on dual PMEM copies to achieve memory persistence. Romulus [30] uses a volatile redo-log with a shadow copy in PMEM. Both DudeTM [83] and NV-HTM [10] uses a persistent redo-log with shadow copy stored in DRAM. PMThread [117] maintains a DRAM copy and an additional PMEM copy. All of these approaches at least double the memory consumption of the application as at least two copies of the data are maintained. Clobber-NVM only have log entries during an update operation, the additional space overhead will usually be even much smaller than conventional redo and undo log systems.

Currently, PMEM failure-atomicity systems have operation semantics that rely on either lock-inferred failure atomic sections (FASEs) [11, 85, 61, 54], classical transactions [113, 83, 10, 86] or programmer delineated transaction boundary with a proper lock scheme [97, 43]. Several recent works optimize scalability by relaxing the traditional ACID semantics. Pisces [47] exploits snapshot isolation on persistent memory, and TimeStone [72] provides three isolation levels to user and achieves higher concurrency on more relaxed isolation model. Compared to these libraries, we target applications that require full ACID semantics.

A recent line of works provide better performance by only ensure periodic persistence [94] [21] [117]. By periodic persistence, their consistency guarantees is made at per-epoch

granularity, as oppose to per failure-atomic operation in Clobber-NVM and most other systems [97, 11, 85, 113]. In these periodic persistence systems, after failure, persistent data will be recovered to the state of the last completed epoch.

In JUSTDO [61] and iDO [85] logging, the system recovers by resuming execution of the interrupted failure-atomic section. JUSTDO logs and persists the program counter, the to-be-updated address, and the value to be written before each `store` happens. Because of the high cost to operate on conventional machines, JUSTDO assumes it will work on a machine with persistent cache. Its successor, iDO logging, avoids logging before individual stores by using compiler-support to identify idempotent regions and instruments adds logging at their boundaries (almost all idempotent regions contain fewer than 4 writes). iDO logs and persists program state - registers, live stack variables, and the program counter. Clobber-NVM is different from JUSTDO and iDO primarily in its use of clobber logging, which restarts the entire transaction from the beginning, instead of at an intermediate logging point. Therefore, Clobber-NVM logging overhead is, in general, significantly lower than both these systems, as their logged state at each logging point is much larger than Clobber-NVM's, and they always require more logging points [85]. Clobber-NVM also supports volatile data usage in transaction, which is generally not supported by these systems. In JUSTDO, use of volatile data and cache values in registers are forbidden during transaction execution, and iDO does not allow volatile heap usage during a failure atomic section while maintaining the stack in PMEM to reduce logging cost.

3.6 Conclusion

This chapter describes Clobber-NVM, a system that recovers by re-executing interrupted transactions. Clobber-NVM leverages compiler analysis to identify necessary log entries, and automatically adds logging for selected variables — clobber input addresses and non-local volatile variables — at compile time. At recovery, interrupted transactions roll back to the prior-execution state by applying the `clobber_log` and `v_log`, then roll forward to a consistent after-

execution state. Compared with existing PMEM user-level libraries, Clobber-NVM simplifies the logging system and reduces runtime overhead. Our evaluation shows that Clobber-NVM significantly improves performance compared to state-of-the-art failure-atomic libraries.

Acknowledgements

This chapter contains material from ”Clobber-NVM: Log Less, Re-execute More”, by Yi Xu, Joseph Izraelevitz, and Steven Swanson, which appeared in the Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2021. The dissertation author is the primary investigator and first author of this paper.

The author would like to thank the ASPLOS anonymous reviewers and the shepherd for their helpful comments. The author is also thankful to co-authors for their help and support. Finally, the author would like to thank the members of the NVSL research group for their insightful comments.

Chapter 4

Failure is Not an Option, it's an Exception

4.1 Motivation

Machines equipped with PMEM now support *flush-on-fail* semantics, meaning that they provide a hardware guarantee that all in-flight and cached writes will reach PMEM in the event of an external power failure (as opposed to a fault in the machine or its onboard power supply). However, utilizing PMEM in a both performant and programmer-friendly manner remains a challenging problem.

This chapter reviews existing general-purpose persistent memory programming models and their limitations to motivate WPP.

Most existing persistent programming libraries rely on marking regions as *failure-atomic*, that is, all of the code region's effects will survive a failure or none will. Models differ in whether regions are explicitly marked (*transactional*) or inferred from locks (*FASE-based*). In addition, one work has proposed making the whole system persistent.

Transactional Libraries

Transactional PMEM libraries expect the programmer to explicitly mark failure atomic sections. For concurrency, these libraries either rely on off-the-shelf transactional memory systems or require the use of their own locks. For example, NV-Heaps [20], Mnemosyne [112], and DudeTM [84] are built on existing transactional memory (TM) systems, and implement their

failure-atomicity techniques (e.g. redo or undo logging) on top of those systems.

Meanwhile, transactional libraries that rely on locks generally expect transactions to acquire and release locks in a conservative, strong strict two-phase locking pattern [98, 114], that is: transactions acquire all locks at transaction begin, transactions release all locks at transaction commit, and locks are released in the order they are acquired. For example, PMDK [97], Pangolin [127] and Clobber-NVM [123] require applications to follow this lock pattern.

FASE-based libraries

Atlas [12] proposed the concept of failure-atomic sections (FASEs) as an alternative to transactions. A FASE is a failure-atomic operation which begins when a thread acquires its first lock and ends when it holds none — importantly, the final lock held may be different from the first lock. Because this locking scheme allows updates to be visible to other FASEs before a FASE commits, FASE-based libraries are required to track dependencies between threads, and roll back dependent FASEs in case of failure. Because FASEs are dynamically formed at runtime, user annotation is not required for existing lock-based code. NVThreads [54], JUSTDO [61], and iDO [85] follow this model.

Whole System Persistence

Instead of basing persistence on bounded sections of code, whole-system persistence (WSP) [92] focuses on the persistence of the entire system. WSP describes a system substantially similar to eADR and GPF, where an interrupt at power failure triggers the draining of volatile caches/buffers to persistence. This model requires no annotation and avoids the extra work done by transactional or FASE-based systems, but requires that large amounts of state be made persistent at the instant of failure, which until the advent of flush-on-fail systems was not possible.

4.2 Limitations of Prior Art

In this section, we argue that the existing programming models for persistent memory, namely transactional or FASE-based, necessitate an alternative path, especially when working with legacy code.

Fortunately, the emergence of persistent caches has enabled our efforts to develop a revitalize a model that does not fit either of these directions, namely, whole process persistence, in which all process state is preserved at a power failure.

4.2.1 Limitations of Transactions

The fact that many failure atomicity libraries leverage transactional memory is not surprising — transactions are commonly leveraged for durability within databases and file systems. When applied to (volatile) multi-threaded code, the transactional memory programming model simplifies concurrency by exporting to the programmer "single global lock" semantics, that is, the programmer should simply protect groups of accesses to shared data as "transactions," each of which are mutually exclusive. The transactional programming model is in theory appealing as programmers need not worry about data races on shared data, multiple locks, or parallel performance. To this transactional programming model, many failure atomicity libraries add persistence: transactions become both visible to other threads, and persistent, upon transaction completion.

In practice, however, despite decades of research and dedicated hardware support, (volatile) transactional memory has failed to become a common programming paradigm for general purpose multi-threaded code. Transactions generally mix poorly with both other synchronization methods (locks, barriers, condition variables, etc.) [9, 126] and IO [88, 99], tend to incur significant performance overhead when compared to fine grained locking [33, 9], and are incompatible with legacy multi-threaded code [99], whose locking discipline is rarely compatible without significant rewriting. Support for transactional memory in C++, for example, remains

experimental [88]. There is no indication that persistent transactional memory systems will solve these problems, indeed, they appear to perpetuate them.

Generally, the transactional programming model is exported to the programmer using a scoped transaction, (e.g. `transaction{}`) and the library guarantees transactions will execute mutually exclusively (e.g. PMDK's C++ interface). However, for PMEM, transactional libraries may syntactically decouple mutual exclusion from failure atomicity due to language limitations (e.g. PMDK's C interface). In such an API, the library expects the programmer to first explicitly acquire the necessary locks to gain mutual exclusion before, subsequently, executing the transaction's failure atomic contents.

Despite this apparent separation, a transactional PMEM library's programming model imposes hard limits on the locking discipline - it expects that all transactional updates are mutually exclusive and isolated by the locking discipline. This restriction effectively forces the application to use a limited locking scheme such as single-global-lock or strong strict conservative two phase locking to protect any failure-atomic update. The programming model *explicitly disallows* releasing or acquiring a lock while executing a failure-atomic update.

For more complex locking schemes in which failure-atomic writes are visible to other threads before they are committed, the use of a FASE-based programming model is required, and is often necessary for legacy programs as, in general, their existing synchronization fails to follow the restrictive transactional requirements.

4.2.2 Limitations of FASEs

Despite being, at first appearances, more compatible with legacy code, we argue the FASE-based model is also fundamentally flawed, or, at the very least, excessively permissive. The FASE model defines a failure-atomic code region as a "contiguous critical section," that is, it defines a failure-atomic code region as stretching from a thread's first lock acquire until the point where it holds no locks. While flexible with respect to locking scheme, this model requires tracking runtime dependencies between concurrently running failure-atomic code regions, which

may not be isolated from each other. This permissiveness results in complicated and degenerate scenarios for recovery.

As a contribution of this work, we demonstrate that, for certain adversarial application patterns, any FASE-based system will either fail to recover or collapse into a degenerate case in which literally all program state must be logged for recovery, including volatile data never accessed within failure atomic regions — effectively, the FASE programming model requires whole process persistence for correctness.

Theorem 1 (FASE Limitation)**.** *There exist applications for which, in order to consistently recover from a crash, a reasonably permissive FASE-based failure atomicity system requires all volatile program state be available at recovery.*

We prove this theorem by counterexample. This counterexample (Figure 4.1) can emerge naturally where two threads communicate via shared variables and one executes IO, a common pattern in event-based servers. In these servers, some threads handle the IO socket (thread 2 in example), some threads are application workers (thread 1), and they communicate via shared flags to manage outstanding requests. Detecting this pattern requires detailed reasoning about synchronization, and therefore prevents the blind use of FASEs on applications.

In the remainder of this section, we describe the counterexample and a brief sketch of our proof's reasoning. A full proof by contradiction, formal definitions, and additional discussion incorporating related work can be found in Chapter 4.3.

Figure 4.1 gives our adversarial application that breaks FASE-based systems. In this example, two threads compute a fixed series of four values for nonvolatile variable `x`. Thread 1 computes the first value, Thread 2 the second and third, and Thread 1 the final, fourth value.

The two "tricks" of the code are that (1) the long FASE executed by thread 1 (lines 49 through 65) spans the entire example and (2) the third value of `x`, computed, but not assigned, outside of a FASE (line 82), is dependent on an access to a large volatile array `Q`.

Recovery of this example presents an unsolvable problem. First, we note that Thread

```
44 lock_t lock0, lock1, lock2;
45 bool cond1 = false, cond2 = false;
46 int Q[] = rand(); // large random volatile array
47 nvm<int> x = 0; // x resides in nvm
```

```
48 void thread1{
49   lock0.lock();
50     x = (int s1=f1(x));
51
52     lock1.lock();
53       cond1 = true;
54     lock1.unlock();
55
56     bool w = true;
57     while(w){
58       lock2.lock();
59         if(cond2)
60           {w = false;}
61       lock2.unlock();
62     }
63
64     x = (int s4=f4(x));
65   lock0.unlock();
66 }
```

```
67 void thread2{
68   bool w = true;
69   while(w){
70     lock1.lock();
71       if(cond1){
72         w = false;
73         x =(int s2=f2(x));
74       }
75     lock1.unlock();
76   }
77
78   int in;
79   printf("x=%d", s2);
80   scanf("%d",&in);
81   /*****/
82   int s3 = f3(s2,in,Q);
83
84   lock2.lock();
85     x = s3;
86     cond2 = true;
87   lock2.unlock();
88 }
```

Figure 4.1. FASE counterexample

1's long FASE, due to failure-atomicity semantics, forces recovery to recover either to the very beginning of the program or the very end. However, both options are impossible for a crash at line 81, just before `x`'s third value is computed. At this point, thread 2 has already issued IO, so rolling back program state at recovery is inconsistent with the external world. However, rolling forward from this point requires the computation of the third value of `x`, which is dependent on an arbitrarily sized volatile array (`Q`). Since `Q` can be of any size, it can be replaced, without loss of generality, with any or all of the program's volatile state, effectively requiring whole process persistence.

Our proof requires failure atomicity systems to be "reasonably permissive," by which we mean that this counter example can be expressed as valid input for the system. Systems that restrict locking to two-phase-locking (e.g. [97, 123]) or a single, semantic, global lock (i.e. transactional memory [112]) avoid this counterexample by prohibiting the locking pattern. Of course, by the same token, this restriction hampers their utility for legacy code, which rarely follows such a strict locking discipline.

The FASE programming model may be fixable by prohibiting situations like the counter-example. Simple (but undesirable) solutions include prohibiting all volatile accesses or all IO in the program. Alternatively, we could try to prohibit the precise counter-example problem by targeting the interplay between FASE dependencies, volatile accesses, and IO. One potential approach to achieve this involves a specification that disallows volatile accesses concurrently with a FASE execution. However, formally defining this specification is tricky, and formally verifying the proper use of FASEs is almost certainly undecidable through the halting problem. Notably, the requirement of a transactional locking scheme (e.g. strict, strong conservative 2PL) would also prevent the counterexample by explicitly disallowing its locking discipline.

To our knowledge, all existing FASE-based systems (e.g. [61, 85, 12, 54]) are "reasonably permissive" and would both accept this code as valid input and fail to recover correctly on it.

4.3 Proof

In this section, we provide a proof of theorem 1:

Theorem (FASE Limitation)**.** *There exist applications for which, in order to consistently recover from a crash, a reasonably permissive FASE-based failure atomicity system requires all volatile program state be available at recovery.*

4.3.1 Definitions

We begin by defining terms. By *application* we mean a multi-threaded program, executed as a *process*. The process's *internal state* consists of all its data, including heap, globals, and stack. Some memory locations are designated *nonvolatile*, their contents (the *nonvolatile state*) survive a power outage; the remainder are *volatile*, and their contents (the *volatile state*) are lost. The process may perform IO operations — we term the set of IO operations performed by an executing process its *external state*. The process, being multi-threaded, contains code regions that execute while a lock is held, these are termed *critical sections*.

If power is lost during process execution, its volatile state is lost. The purpose of a *failure atomicity system* is to provide *consistent recovery* from a power outage. For consistent recovery, the system selects a point in execution, termed the *recovery point*. The recovery point is *consistent* with the external state; process execution from initialization through the recovery point would generate the observed IO. For failure atomicity, the recovery point also lies outside all critical sections. Consistent recovery of a process consists of selecting a valid recovery point and restoring the persistent state's contents to its values as of this point. If the power failure interrupts a critical section, consistent recovery will involve, for failure atomicity, chosing a recovery point outside the critical section and undoing or redoing changes made within the section.

We assume a powerful failure atomicity system which is free, during pre-crash execution, to intercept the process at any point and log data in nonvolatile memory. After a crash, the system

has access both to these logs and the process's nonvolatile state — its task is to ensure that the nonvolatile state is restored to a recovery point; consistent with IO operations and outside any critical section. The failure atomicity system must be *reasonably permissive* with respect to its programming model — we require the system's programming model to support our adversarial example. To all our knowledge, all existing FASE-based systems are "reasonably permissive".

Figure 4.1 gives our counterexample. The "trick" is that the long FASE executed by thread 1 (lines 49 through 65) is dependent on non-FASE code executed by thread 2 that contains both IO and accesses to large volatile data (lines 79 through 82).

4.3.2 Proof Sketch

We prove Theorem 1 by contradiction. We consider a process executing the code sample in Figure 4.1 and suffering a power failure on line 81. Suppose, for contradiction, there exists a FASE system for which, given this situation, could restore the program's nonvolatile state to a recovery point consistent with the external state and outside any critical section. As thread 1, by construction, executes a critical section (FASE) for its duration, our recovery point for thread 1 must lie at line 49 or line 65 — all other points violate failure atomicity. We consider both options.

Suppose the recovery point lies at line 49 (i.e. recover `x` to 0), it is inconsistent with the external process state due to the IO executed before the failure on lines 79 and lines 80, which indicate that thread 2 (and therefore thread 1) have progressed beyond this recovery point, leading to a contradiction.

Suppose the recovery point lies at line 65 (i.e. recover `x` to `s4`). First we note that the value `s4` has a true dependence (read-after-write) on `s3`, and `s3` has a true dependence on both the inputed seed `in` and large volatile array `Q`. Since `s3` cannot be computed before `in` is known, `s3` must be computed after the `scanf` on line 80 is executed. Since the failure can interrupt the computation of `s3` after the `scanf`, all inputs to `f3` must be preserved in nonvolatile storage for recovery. However, since `Q` is an arbitrarily sized volatile array, Q can be of any size and can be

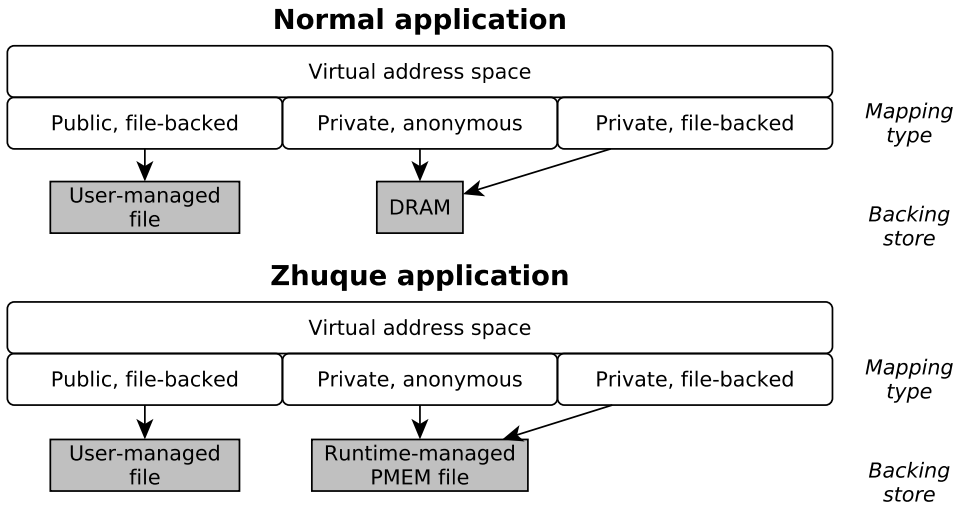


Figure 4.2. Virtual memory in Zhuque. The runtime modifies the backing store based on the mapping type, but the interface presented to the userspace application does not change.

replaced, without loss of generality, with any or all of the program's volatile state, requiring the failure atomicity system to preserve all volatile process state and leading to a contradiction.

4.4 Design

Whole process persistence (WPP) is our answer to the limitations of transaction- and FASE-based programming models. In WPP, the in-memory state of an individual process is made persistent with, in simple cases, no modification to the application, primarily by interposing on the creation of virtual memory mappings (see Figure 4.2). WPP is designed for systems with flush-on-fail support, so we expect the contents of the process's PMEM-backed cache lines to survive a power failure. When the process is restarted after a power failure, it receives an OS signal, which it can ignore or handle with a signal handler. If no signal handler is installed, or if the installed signal handler does not exit the program, each thread continues execution at the point where it was interrupted by the failure.

There are several benefits to this model over transactions and FASEs. First and most importantly, WPP solves the problem described in Chapter 4.2 by discarding the concept of a failure-atomic section. The visible effects of an instruction on process state (that is, not including effects on OS state or peripherals) are guaranteed to survive a failure at least from

the point at which they are visible to other threads. Second, restarting at the point of failure removes the need to ”redo” or ”undo” any writes at recovery, and with it the need to keep a persistent log and incur the cost of extra writes to PMEM. Third, no longer needing to define failure-atomic sections either reduces the programmer's burden directly, compared to manually-annotated failure-atomicity systems, or allows them to design concurrency schemes orthogonal to persistency without incurring overhead, unlike FASE-based systems.

There are two requirements that must be satisfied in order for an application to use WPP. The first is that its threading and virtual memory must be managed using a well-defined API for those purposes (i.e., on POSIX: `mmap()`, `pthread_create()`, etc). Any modern application targeting a POSIX system would have to go out of its way in order to violate this requirement.

The second is that applications must check error returns from system calls and other mechanisms that access non-process-private state, to detect failure-related errors beyond the process boundary, such as an application using a file on a filesystem that was not remounted after system restart. This requirement is more onerous than the first, but in our experience a wide range of applications can be correctly restarted without modification or special handling.

The principal challenge in implementing WPP is preserving process state across a power failure. Continuing execution after failure requires that the process's virtual address space, volatile architectural state, and relevant kernel-resident state (e.g., the file descriptor table) are a) persistent or b) can be resurrected along with the application.

The remainder of this section introduces Zhuque, our runtime implementing WPP, and describes how it makes process state persistent and restores that state after failure.

4.4.1 Overview

Zhuque provides WPP functionality by interposing on system calls which allocate resources (memory, file descriptors, threads), and by modifying the application startup process. In order to do this, we modified `libc`, which provides C bindings for system calls and implements the application startup process.

Interposing on system calls allows Zhuque to ensure that all application state which is normally volatile is instead stored in PMEM, as shown in Figure 4.2. It also allows Zhuque to track memory mappings and system calls so it can reconstruct the program's address space and re-create its kernel-resident state after a failure. Remaining volatile architectural state (e.g., the register file) is preserved by writing it to PMEM at failure.

When the application is resurrected after failure, Zhuque restores the application's address space, respawns its threads, and each thread reloads its architectural state. Execution resumes by calling the program's power failure signal handler, if it exists, and then resuming execution of each thread at the point interrupted by power failure.

4.4.2 Ensuring State Persistence

The first requirement that Zhuque must fulfill is ensuring that all state required for continuing correct execution of a program is preserved across power failures. This state can be divided into three categories based on its storage location: architectural state, memory state, and file state.

If a system supports flush-on-fail, it would be possible to modify its firmware to write per-thread architectural state (register file, floating-point configuration, etc.) to PMEM in response to power failure. However, we do not have the ability to modify that firmware, so we emulate it using

File state is either inherently persistent, if the file was opened read-only or if changes have been written to disk, or is buffered awaiting being written to disk, in which case it is actually memory state and is handled as described below.

Automatically ensuring memory state is persistent is more complex, and is one of the main innovations of this design. Memory state itself can be divided into dynamic and static memory.

Dynamic memory Programs conjure dynamically allocated (heap) memory and thread stacks by calling anonymous `mmap()` (often via `malloc()`). Zhuque interposes on `mmap()` so that

requests for anonymous memory return DAX-mapped persistent memory backed by a runtime-managed PMEM file, making heap and stack memory persistent.

Static memory Before an application binary is executed, the loader uses `mmap()` to create memory regions to hold code and static data (globals) from the application binary and linked dynamic libraries. Zhuque treats these regions differently based on whether they are un/zero-initialized, or initialized to non-zero values. The loader creates un/zero-initialized regions with anonymous `mmap()`, so they are treated as dynamic memory.

Initialized static memory, however, actually takes up space in the binary, and is loaded by mapping that region of the binary into memory as a private mapping. Thus, Zhuque transforms any writable, private mapping backed by a file to a writable, shared mapping that is backed by a PMEM file (see Figure 4.2), which is populated with the initialization values from the binary.

This mechanism also cleanly handles other outputs of dynamic loading, like relocations of position-independent code and cross-binary symbol resolutions, since they also are stored in writable, file-backed, private mappings.

4.4.3 Ensuring Correct Restoration

Having persisted the application's state, we also have to ensure it can be restored correctly. Recovery must restore the application address space, restore kernel-resident state, and restore architectural state.

Application address space All of the PMEM-backed memory mappings managed by Zhuque, as well as any other mappings the application created with `mmap()`. Zhuque stores the mapping table in a persistent memory file, and updates it to match any changes to the address space as they occur, so no action is required at failure to ensure this metadata is persistent.

At recovery, restoring the virtual memory map to its previous state must be done first, because all other state to be restored is stored in virtually mapped persistent memory. Restoration consists of re-mapping each virtual memory region with the correct backing store and access

permissions. This restoration also replaces dynamic loading.

Kernel-resident state Any data required for continuing execution that resides outside the address space (and architectural state) of the process. The specific data varies depending on operating system and implementation decisions: for instance, Zhuque tracks the state of open Linux file descriptors in PMEM and restores them at restart using system calls. We discuss Zhuque's handling of kernel-resident state in Chapter 4.5.

Architectural state Any state stored in the processor itself and directly accessible from software. This state is per-thread, and since it includes the program counter and stack pointer registers, restoring it is equivalent to restarting execution of the thread (so it must be done last).

Zhuque manipulates the saved PC and stack so that the thread resumes as if it had just called the application-defined failure handler (if it exists), and then that handler returns to the point interrupted by execution when it executes a `RET` instruction. To avoid references to a thread which has not yet been recreated, threads wait to restart execution after they are created until all threads have been created.

4.5 Implementation

Figure 4.3 depicts Zhuque's place in the runtime environment, and Figure 4.4 shows the changes to control flow at initialization and termination. This section describes the life cycle of a Zhuque process, describes how Zhuque handles the userspace-kernel boundary, and finally discusses some limitations of our prototype implementation.

4.5.1 Process Life Cycle

When Zhuque starts a process, it checks an environment variable for a path to a directory which holds or will hold the persistent state for that process. One file in the directory holds the process's global "process context", a memory map of a C structure. The directory also holds all other persistent memory files allocated during the process's life.

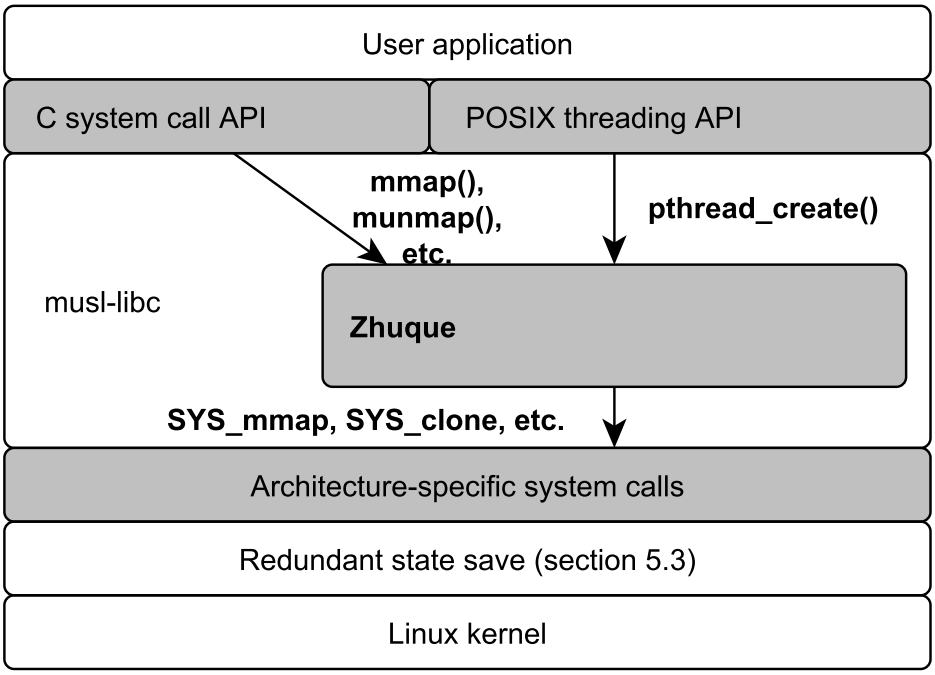


Figure 4.3. Zhuque architecture. User applications link to the C APIs provided by musl libc, and we modify the implementation of the APIs and the arguments passed to the underlying system calls. To protect against failures in kernel mode, we save userspace context to PMEM on entry to the kernel.

Zhuque takes control of the process after the dynamic loader loads its own metadata using information provided by the kernel ("loader bootstrapping"). If the context file is present, then Zhuque takes steps to *restart* the process. If the context file is missing, but the environment variable is set, then it is a newly created Zhuque process (i.e., a *clean start*).

Clean start In the clean start case, Zhuque creates and initializes the process context file. Then, it records the locations of the dynamic loader and the main binary in the mapping table and remaps their static memory sections to memory backed by a persistent file. This retroactive process is necessary because our userspace runtime cannot interpose on mappings created by the kernel.

Next, control returns to the loader and it loads the application's dynamically-linked dependencies. Our code intercepts the loader's calls to `mmap()` and `mprotect()` during this process in order to record the mapping metadata and transform any writable, private mappings into persistent memory regions.

After loading is complete, control returns to Zhuque just before `main()` executes. Zhuque

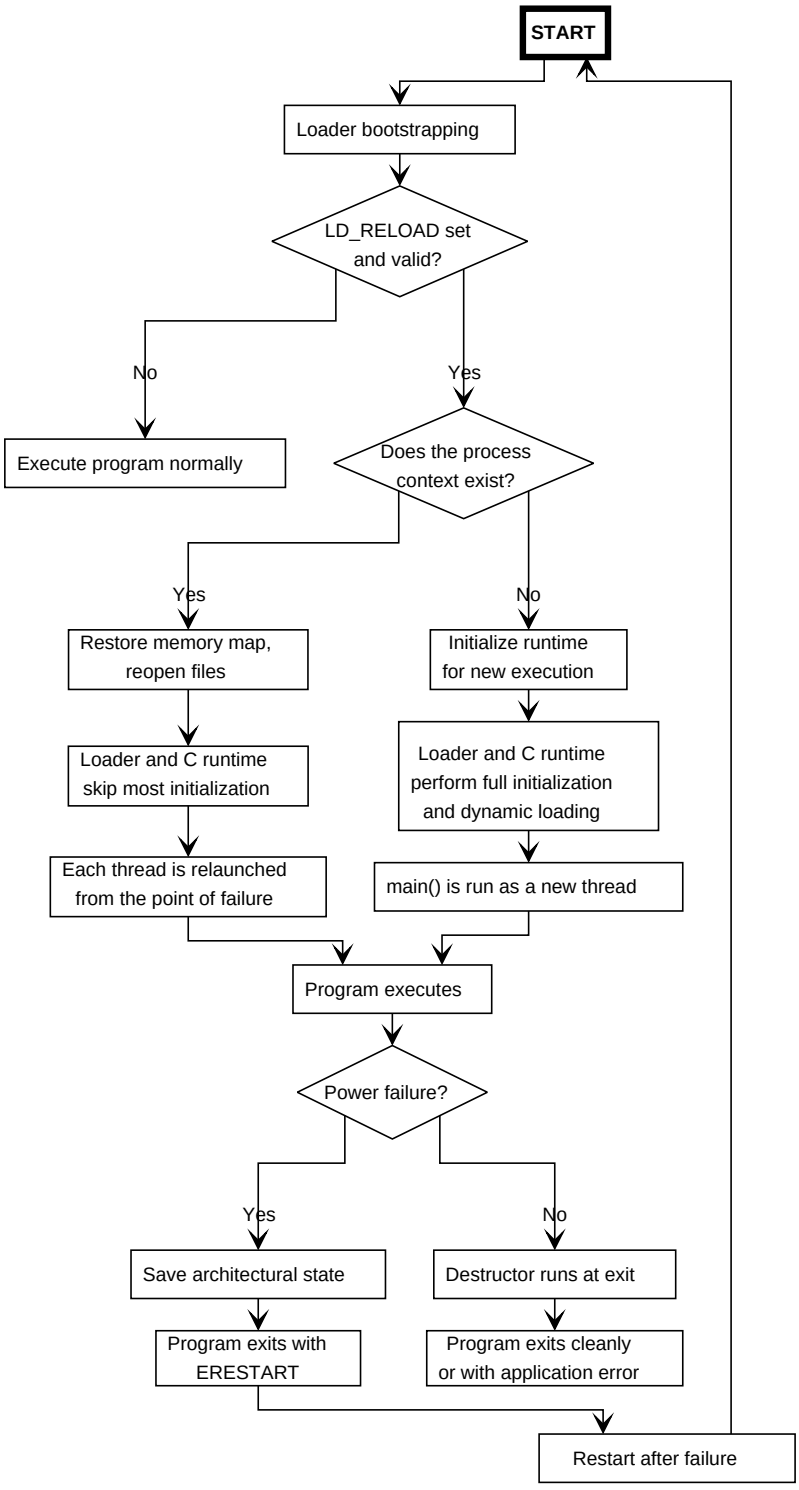


Figure 4.4. Zhuque runtime control flow. Zhuque modifies runtime startup and termination; application code is not modified.

copies `main()`'s arguments into PMEM and runs it in a new thread with a persistent stack.

Power failure To save volatile architectural state (e.g. the register file) to PMEM at failure, we propose repurposing existing functionality. NVDIMM eADR and CXL GPF both rely on a System Management Interrupt (SMI) to implement the flush-on-fail process on x86 systems (see Chapter 2.2). SMI handling saves volatile architectural state to a designated per-core region (the *SMRAM*) before beginning execution of the handler, and x86 allows the SMRAM to be PMEM-backed [29]. However, the location of the SMRAM is controlled by system firmware. Unfortunately, updates to firmware must be signed by the manufacturer — the firmware

Instead, to test Zhuque's application support, we emulate the SMI's state save using userspace signals. If SIGPWR is delivered while the process is executing, the volatile thread receives it and sends a second signal to each thread. When the kernel interrupts a thread to run the signal handler, it first pushes the register file and other state needed to resume execution onto the persistent thread stack. The handler body saves the current stack pointer and some context not saved by handler entry in PMEM, and then exits the thread directly, preserving the contents of the stack. Thus, at recovery, we have access to a persistent memory region containing a snapshot of volatile architectural state at failure, as if it had been saved by an SMI.

Restart after failure On restart, the runtime opens the context file, re-creates PMEM mappings, re-opens file descriptors, and finally re-maps file-backed memory. If a file descriptor was closed after being used to create a mapping, it is temporarily re-opened while the mapping is restored.

After the virtual memory map and file set are re-established, Zhuque restarts the execution of each thread from the point of failure. Zhuque does this by starting each thread with the same start routine, and the same initial stack pointer, so that the bottom frames of the stack are overwritten with new frames of the same size, and the contents of application frames are preserved. From this entry routine, we use assembly to restore architectural state, including setting the stack pointer and PC to the addresses saved at failure. Execution resumes within the

runtime's failure handler, which calls the user-defined failure handler if present. If there is no user-defined handler, or the handler does not exit the program, execution continues at the point interrupted by the signal at failure.

4.5.2 Kernel-resident State

In order to preserve correctness in a userspace-only implementation, our runtime tracks and restores two pieces of kernel state tied to the process: the file descriptor set and the thread set.

To track the **thread set**, our runtime interposes on calls to `pthread_create()`, wrapping the passed thread entry point and arguments in our own entry point function (which itself is wrapped in the musl entry point function). It also saves both the Linux and pthreads identifiers for the thread; the Linux ID is used at failure to signal each thread individually with `tgkill()`, while the pthreads ID is the address of the thread metadata, and is used at restoration to continue execution at the point of failure. The Linux ID of a thread changes when the process is restarted, while the pthreads ID does not, because we restart threads at recovery with a modified version of `pthread_create()` which uses an existing thread metadata object rather than creating a new one.

To track the **file descriptor set**, Zhuque interposes on calls which assign (e.g. `open()`, `socket()`), modify (e.g. `fcntl()`, `bind()`), or release (i.e. `close()`) file descriptors and replays them at restart, using `dup()` to patch any discrepancy in assigned descriptors. This approach is sufficient for sockets with stateless protocols, `epoll` file descriptors, and simple file accesses.

However, pipes and files require special handling: for regular files, we ensure that they will not be deleted between failure and restoration by creating separate hardlinks to the files and using those to reference the file, deleting them when the program exits cleanly; we also open them without kernel buffering (i.e. with O_SYNC) due to the limits of a userspace implementation. And for pipes, we use the `splice()` family of system calls to save and restore unconsumed contents at failure/restoration. Support for restoring network sockets is best-effort, and a more

systematic approach to network support under WPP is an interesting future extension of this work.

4.5.3 Failures in kernel mode

When an application makes a system call, or is suspended by an interrupt, the kernel will save the application's volatile architectural state on the suspended thread's kernel stack and restore it when application execution resumes. In our implementation the kernel stack is volatile, so we must save this state in persistent memory to allow recovery if power failure interrupts the kernel-mode operation.

To enable this, we added a `prctl()` operation to designate a page as a redundant state save area. We added code on all entries from user- to kernel-mode which checks whether such a page has been provided, and if so saves the state there as well as to the kernel stack. Accesses to the page must not fault: since a page fault is itself an interrupt, a fault in interrupt entry deadlocks the kernel. We found that there is no way (in our test kernel version) to reliably prevent access to a filesystem DAX page from faulting, so we use device DAX to provide the save memory.

4.5.4 Limitations

There are two notable limitations of our implementation, neither of which is fundamental to the design.

Multi-process applications are not supported. Zhuque currently has no support for persisting multiple processes in the same process tree; if an application under our runtime forks a new process while leaving the `LD_RELOAD` environment variable unchanged, the child process will crash when it attempts to use the same context object as its parent. By the same token, we make no attempt to preserve OS process IDs across failures, so applications that save and retrieve their PID after failure may find it invalid. Our runtime supports unrestricted concurrency schemes, so we believe it would be possible to extend it to support multi-process operation by interposing on the creation and termination of processes, similar to our approach to threads.

Some ASLR is not supported. Address space layout randomization (ASLR) is a security technique which randomizes the address of virtual memory mappings. Random addresses returned by `mmap()`to userspace are not an obstacle, since the randomization only occurs once under Zhuque. However, Zhuque cannot prevent the kernel from mapping libc and the application binary at a random location at restart, which means that their static memory cannot be recreated at the same locations when ASLR is enabled. It would be possible to move those mappings after they are created, but since it does not otherwise affect correctness or performance, and it would add significant complexity to the startup process, we chose not to implement this feature.

4.6 Evaluation

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

- How much performance improvement does Zhuque provide for persistent applications compared to existing libraries?
- How much performance overhead does Zhuque incur compared to native, volatile execution?
- What benefits does Zhuque provide by enabling zero-effort persistence?

Our experimental workloads include a set of micro-benchmarks, a set of Python benchmarks, and three recoverable applications.

4.6.1 Evaluation Setup

We compare against native application performance based on musl and the performance of four popular PMEM libraries.

Musl [89] stands for the default (volatile) implementation based on musl libc.

PMDK [97] is Intel's failure atomicity library. It uses hybrid undo-redo log for both failure-atomicity [57] and memory allocation [103]. **Atlas** [12] also uses an undo logging-based mechanism for failure-atomicity. It can automatically infer failure-atomic region boundaries

by analyzing lock behaviors in application code. **Clobber-NVM** [123] is a state-of-the-art PMEM library. It records clobber_log and v_log during runtime, and recovers an application by re-executing any interrupted transactions. **Mnemosyne** [112] is a redo-log based system. Instead of relying on locks in user applications, Mnemosyne uses the C++ transactional memory model to parallelize code.

To measure their performance with flush-on-fail semantics, we removed the flushes and fences from all three comparison libraries. These *flush-on-fail (FoF)* versions are used for all experiments described below, unless otherwise noted.

We run the benchmarks on a platform with one 20-core Intel Xeon Gold 6230 processor, running at 2.1 GHz. The platform has a total of 96 GB of DRAM and 768 GB (6 ×128 GB) of Intel Optane DC Persistent Memory directly attached to the processor [59]. We configured our test machine such that Optane DCPMM is in 100% App Direct mode [2]. In this mode, software has direct, byte-granularity access to the Optane DCPMM. Zhuque uses DAX-mapped [82] `Ext4` files for all PMEM allocations except the kernel state save area, which uses device DAX for the reasons described in Chapter 4.5.3. Unless otherwise noted, all Zhuque experiments are run on the modified kernel with the state save area activated, while all comparison software is run on an unmodified kernel of the same version (Ubuntu kernel 4.15.0-169). Each data point reported is the average over five runs.

We observed variations in memory usage across different benchmarks, ranging from 20 MB to 1 GB. For all but two of the Python benchmarks, the memory usage exceeds the Last Level Cache (LLC) capacity of 30.25 MB.

As discussed in Chapter 4.5.1, we were unable to modify platform firmware to direct the SMI state save to PMEM, because firmware updates must be signed by the manufacturer [28]. However, since it only affects events at failure, we expect that making this change would produce no measurable changes in the steady-state performance results we report below.

We verified that, with Zhuque enabled, all benchmark applications restarted and ran to completion correctly after randomly-timed simulated power failures.

4.6.2 Microbenchmarks

In our first experiment, we compare Zhuque's performance with the default (volatile) musl libc on a set of microbenchmarks from the libc-bench [73] benchmark suite which tests the operations modified by Zhuque: thread creation and virtual memory mapping. We implemented the latter within the libc-bench test harness. Descriptions of the `pthread_create()` benchmarks can be found in the libc-bench documentation [73]. The `mmap()` benchmarks are described below.

anon_slab creates a 16 MB anonymous mapping, writes to every page, and then removes the mapping. **anon_chunks** creates, writes to every page, and then removes 128 128 kB anonymous mappings. **file_slab** creates a 16 MB file-backed mapping, writes to every page, and then removes the mapping. **file_chunks** creates, writes to every page, and then removes 128 128 kB mappings backed by a 16 MB file.
files_chunks creates, writes to every page, and then removes 128 128 kB mappings each backed by a separate 128 kB file.

We ran these benchmarks on four implementations: **Zhuque** is Zhuque using DAX-mapped PMEM as the backing store, running on modified kernel. **Zhuque(original kernel)** is Zhuque without kernel modification. **Zhuque on DRAM** uses non-PMEM files (loads and stores access the DRAM page cache) as a backing store, but also saves kernel states. **Musl(original kernel)** is unmodified musl libc with unmodified kernel. We report the results in Figure 4.5.

We observe that Zhuque introduces significant overheads to modified operations, especially `mmap()` and `munmap()` – the bottleneck for all of the poorly-performing microbenchmarks is the allocation of new anonymous memory. The overhead is per-operation, and does not depend on the size of the mapping. This is not surprising: our modified versions still perform the original operations, and are also required to modify userspace data structures maintained by Zhuque in persistent memory, often including a search whose time is linear in the number of created mappings. As demonstrated by the results below, these overheads do not translate to a significant

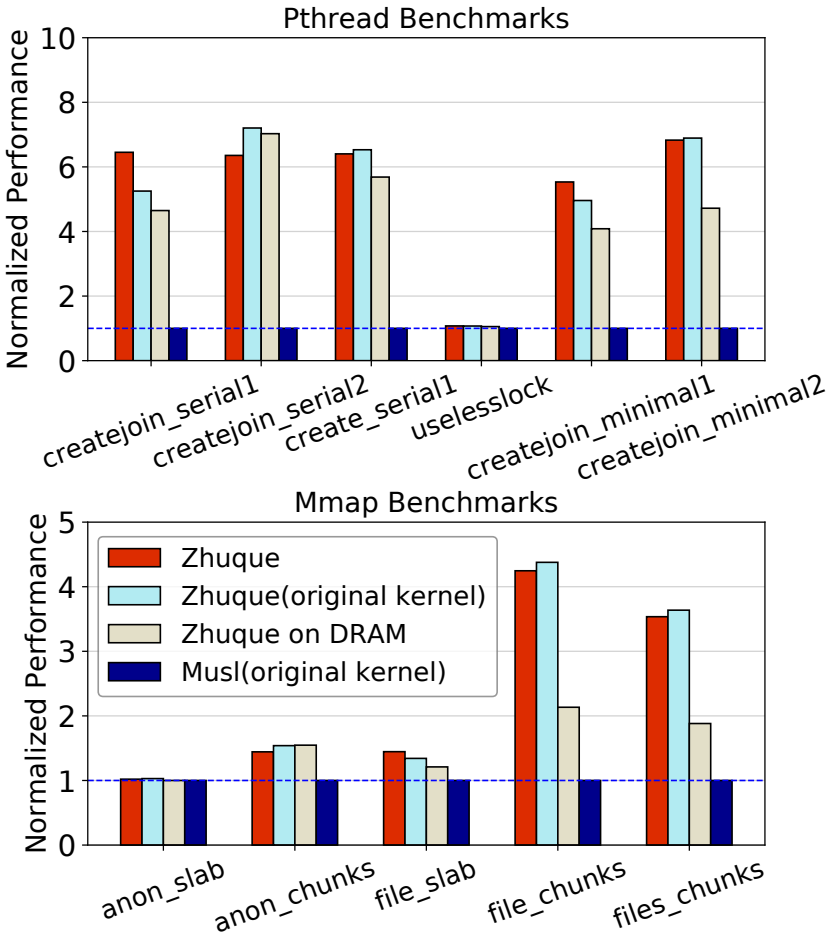


Figure 4.5. Measuring the overhead of Zhuque on basic operations. Performance values are normalized to the Musl(original kernel) value.

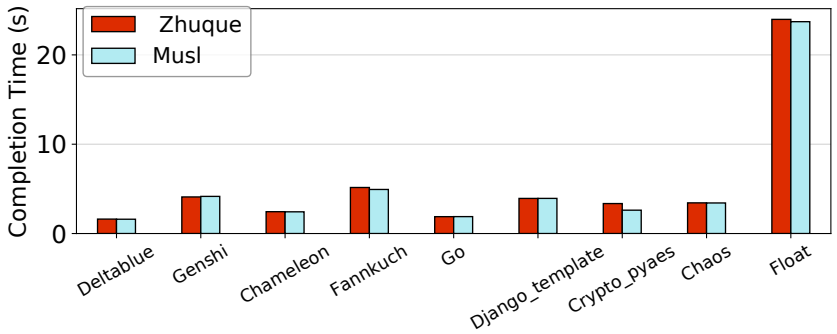


Figure 4.6. Measuring the overhead of different Python benchmarks: Zhuque matches native performance on some benchmarks

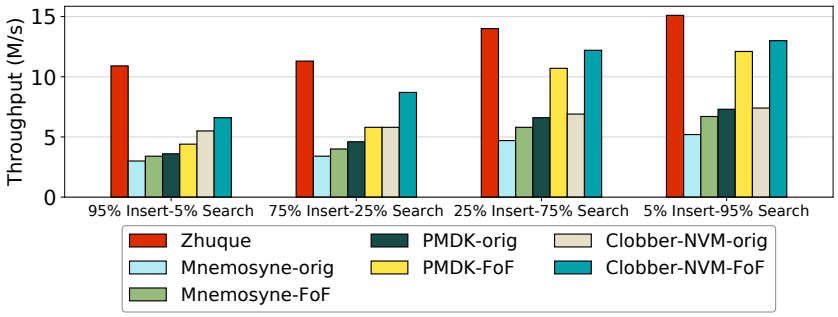


Figure 4.7. Porting Existing PMEM Libraries to Flush-on-Fail Machines Provides up to 1.76× Improvement

slowdown on macrobenchmarks, because modifications of the virtual memory map are rarely on an application's critical path. In addition, these overheads are a result of implementation choices, not the fundamental design. We anticipate they would decrease substantially in a kernel-based implementation of WPP.

4.6.3 Python Benchmarks

In this experiment, we ran nine Python benchmarks on the musl and Zhuque configurations using the CPython interpreter. It demonstrates that Zhuque can run a wide range of unmodified Python applications. We chose the first nine benchmarks (out of 42), in alphabetical order, from version 1.0.0 of the Pyperformance benchmark [36] suite. Descriptions of the benchmarks can be found in the Pyperformance documentation [36]. We ran them in our own benchmark framework, but did not modify the benchmarks themselves. We also added dynamic thread stack support, not present in vanilla musl, in order to support the large stack sizes required by the interpreter. We report the results in Figure 4.6.

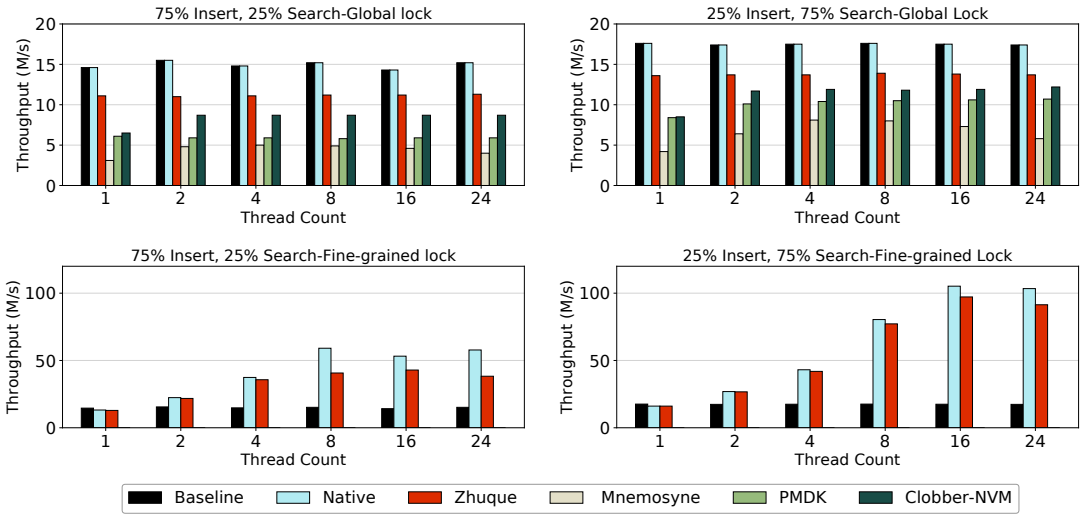


Figure 4.8. Zhuque Enables Newest Version of Memcached to Run on PMEM, Provides Significantly Better Performance

Most of the Python benchmarks perform competitively with the musl versions. Those that perform worse generally incur overhead from frequent random-access reads and writes to data structures too large to fit in the cache, causing thrashing and exposing the higher access latency of persistent memory compared to DRAM.

4.6.4 Memcached

Memcached [87] is a widely-deployed key-value store. Early versions (1.2.*) have been ported to Mnemosyne, PMDK, and Clobber-NVM. We ran memcached-1.2.5 on Zhuque unmodified.

We evaluate memcached performance with four types of workloads: insertion-intensive (95% insertion / 5% search), insertion-mostly (75% insertion / 25% search), search-mostly (25% insertion / 75% search), and search-intensive (5% insertion / 95% search). We use memslap [79] to generate a stream of uniformly distributed memcached requests with 16-byte keys and 64-byte values. As shown in Figure 4.7, Mnemosyne, PMDK and Clobber-NVM can perform up to 1.76× faster with flushes and fences removed. This result indicates that the flush-on-fail semantics can benefit existing PMEM libraries.

Figure 4.8 presents the results of these experiments, using Musl-based memcached-

1.2.5 performance on DRAM as baseline. We find Zhuque can always provide nearly 80% of musl memcached performance. Across different thread configurations, Zhuque provides up to 3.58× Mnemnosyne's throughput, 1.81× of PMDK's throughput and 1.71× of Clobber-NVM's throughput.

Poor scalability is a well-known problem with early versions of memcached [34, 85]. Memcached went through a rewrite of the synchronization framework to use fine-grained locking across seven years of development and over thirty versions [62]. Many current (transactional) PMEM libraries have strict requirements for applications' concurrency schemes. These requirements make converting recent versions of memcached to run on PMEM a complicated and difficult process. Zhuque places no restrictions on the locking scheme, so the newest version (1.6.17) can run unmodified on Zhuque. By simply running the newest version on Zhuque, we can provide more than 7.5× performance of the best performant older version of persistent memcached on the same workload, with the same thread count.

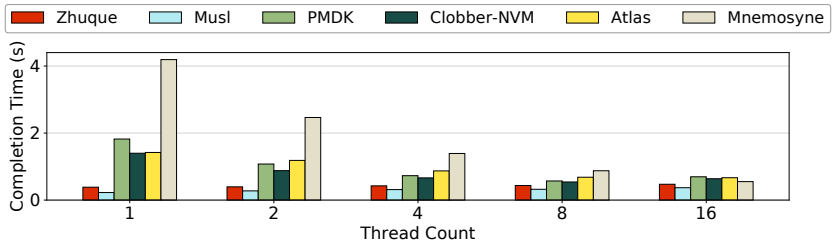
4.6.5 Vacation and Yada

Furthermore, we evaluated the performance of the Vacation and Yada applications from the STAMP benchmark suite [17], each targeting common PMEM applications such as, respectively, KV-stores and graph workloads. Prior works [50, 123, 112] provide readily available implementations of these applications built on top of existing PMEM libraries, with the exception of Yada - Mnemosyne.

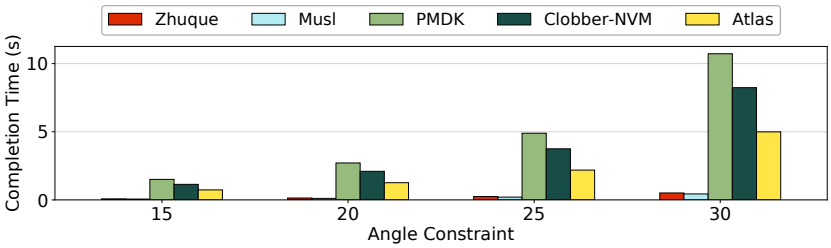
We compare Zhuque with Musl, Mnemosyne, PMDK, Atlas and Clobber-NVM Vacation performance. Vacation is a travel reservation system, consisting of tables updated concurrently using transactions that span multiple tables.

Prior implementations [50, 123, 112] persist the tables in PMEM, and leave the client side in volatile memory. Zhuque persists the entire vacation application, including both the server tables and the client threads.

Figure 4.9 shows that Zhuque performs 10.8×, 4.8×, 3.7× and 3.6× faster on Vaca-



(a) Vacation



(b) Yada

Figure 4.9. Vacation and Yada performance on different PMEM libraries and Zhuque

tion than Mnemosyne, PMDK, Atlas and Clobber-NVM, respectively. The performance gain comes from fewer logging writes and more efficient memory management: Zhuque uses vanilla `malloc()`, while the other libraries use either PMDK's transactional allocator (PMDK, Clobber-NVM) or a hand-written allocator (Mnemosyne, Atlas). The memory management efficiency problem is more prominent on Yada, which implements Ruppert's algorithm for Delaunay mesh refinement [101].

4.7 Related Work

PMEM Software & Hardware. In the past decade, researchers have designed many highly-optimized PMEM data structures [21, 94, 42, 124, 14, 111]. They are designed to reduce the cost of persistent updates while ensuring failure-atomicity. Because they are carefully designed by experts to cope with PMEM characteristics, they usually provide good performance. However, using and developing these data structures takes significant programming effort.

Because PMEM's bandwidth is significantly higher than traditional secondary storage, PMEM file systems [116, 22, 64, 120] aim to expose raw PMEM performance as much as possible. It is easy for existing applications to use files on these PMEM file systems, but they are not designed to solve the same problem that Zhuque targets (failure-resilience of the application's

in-memory data).

The architecture community has also sought better hardware support for PMEM, often by allowing more permissive (and thus performant) store ordering at the memory controller [70, 69, 7, 63, 45, 31, 107, 71]. Many of these systems demonstrate dramatic performance gains in simulation, but none that we are aware of are available in production.

General-purpose PMEM libraries. General-purpose PMEM libraries aim to ensure failure-atomicity for applications which directly access PMEM, with low overhead and minimal code changes.

Traditional undo-log systems [20, 12, 97] and redo-log systems [112, 43] write to a log alongside every visible update, at least doubling each write. On non-flush-on-fail PMEM machines, undo-logging usually requires expensive memory fences at the end of each log write, while redo-logging needs to redirect loads even if the machine supports flush-on-fail.

To avoid expensive synchronization between threads, some undo/redo systems buffer writes in ”shadow” copies of PMEM data [86, 30, 84, 10, 117], at least doubling the amount of PMEM required by the application. These systems still incur the write amplification and read redirection costs of conventional logging, and with flush-on-fail semantics the synchronization they are designed to reduce is no longer required at all. Compared to these systems, Zhuque does not amplify or redirect memory accesses, nor increase the size of the working set.

All these systems rely on either lock-inferred failure atomic sections (FASEs) [12, 85, 61, 54, 117], classical transactions [112, 84, 10, 86], or programmer delineated transaction boundaries with a restricted lock scheme [97, 43, 123] to identify failure-atomic operations. In contrast, Zhuque is not concerned with synchronization: it uses the same concurrency model as the original application.

JUSTDO [61], iDO [85], and Clobber-NVM [123] recover by resuming execution of the interrupted failure-atomic section or re-executing interrupted transactions. These systems are similar to Zhuque in that they also resume execution at the point of failure, but they all restrict

concurrency and require manual annotation of atomic sections.

Single Level Stores. WPP transparently makes processes persistent by providing continuous checkpointing, durability guarantees for external observers of application IO (known as *external synchrony* [95]), and POSIX compatibility. Single level stores (SLS) [105, 104, 106, 25, 108, 74, 109] also provide persistent address spaces to applications, but they suffer from high overhead, rarely support external synchrony, and are often hard to use due to custom APIs [105, 108].

The high overhead arises from checkpointing and, for some systems, the enforcement of external synchrony. Checkpointing overhead is high because traditional storage devices are far slower than DRAM, and SLSes usually amplify writes by tracking memory modification at page granularity. To achieve external synchrony, SLS systems must delay IO until data is safely persisted. If checkpoints are too frequent, the communication delay can cause high overheads.

Because of the performance penalty of providing external synchrony, most SLSes choose not to enforce it. The state-of-the-art SLS system Aurora [109] points out the value of external synchrony, but does not support it. Zhuque shows that flush-on-fail semantics allow continuous checkpointing, making external synchrony feasible and performant.

4.8 Conclusion

This chapter has described Whole Process Persistence (WPP), a programming model that treats power failure as a recoverable exception. Zhuque, implementing WPP, transparently makes applications failure-resilient by interposing on the POSIX-specified APIs. Zhuque ensures process state survives power failures, and allows for resumption of execution.

Compared with existing solutions, Zhuque greatly simplifies the programming model. Our evaluation shows that Zhuque significantly improves performance compared to state-of-the-art, and tends to match original volatile application performance on certain workloads.

Acknowledgements

This chapter contains material from ”Zhuque: Failure is Not an Option, it’s an Exception”, by George Hodgkins*, Yi Xu* (*co-first author), Steven Swanson, and Joseph Izraelevitz, which appeared in 2023 USENIX Annual Technical Conference (USENIX ATC 23). The dissertation author is the primary investigator and co-first author of this paper.

The author would like to thank the ATC anonymous reviewers and the shepherd for their helpful comments. The author is also thankful to co-authors for their help and support. Finally, the author would like to thank the members of the NVSL research group for their insightful comments.

Chapter 5

Leveraging Persistent Memory in Key-value Stores via a Crash-safe and Durable Cache

5.1 Key-Value Stores

PERSISTRON is designed for a machine equipped with both PMEM (e.g. Intel Optane DC PMM's [56] or persistent CXL.mem devices [100]) and DRAM. It assumes the system support *flush-on-fail* semantics.

In this chapter we outline common design elements of modern key-value stores and specific details of SplinterDB that are relevant to this system and motivate the system.

Write-optimized key-value stores. Most high-performance key-value stores, including SplinterDB, use write-optimized data-structures, such as log-structured merge (LSM) trees or B^{ε}-trees.

Figure 5.1 gives a high-level view of the data structures in SplinterDB. Updates (e.g. insertions of new key-value pairs or deletions of keys) are recorded in the ***memtable***, an in-memory B-tree. Most (if not all) LSMs have a memtable, although their memtable might use a different data structure, such as a skip list or hash table, depending on their expected workload. The techniques in this paper apply to other memtable structures, as well.

The rest of the SplinterDB data structure is a tree of trees and filters. The main tree is called the ***trunk***, and there are additional B-trees attached to the trunk called ***branches***. All data

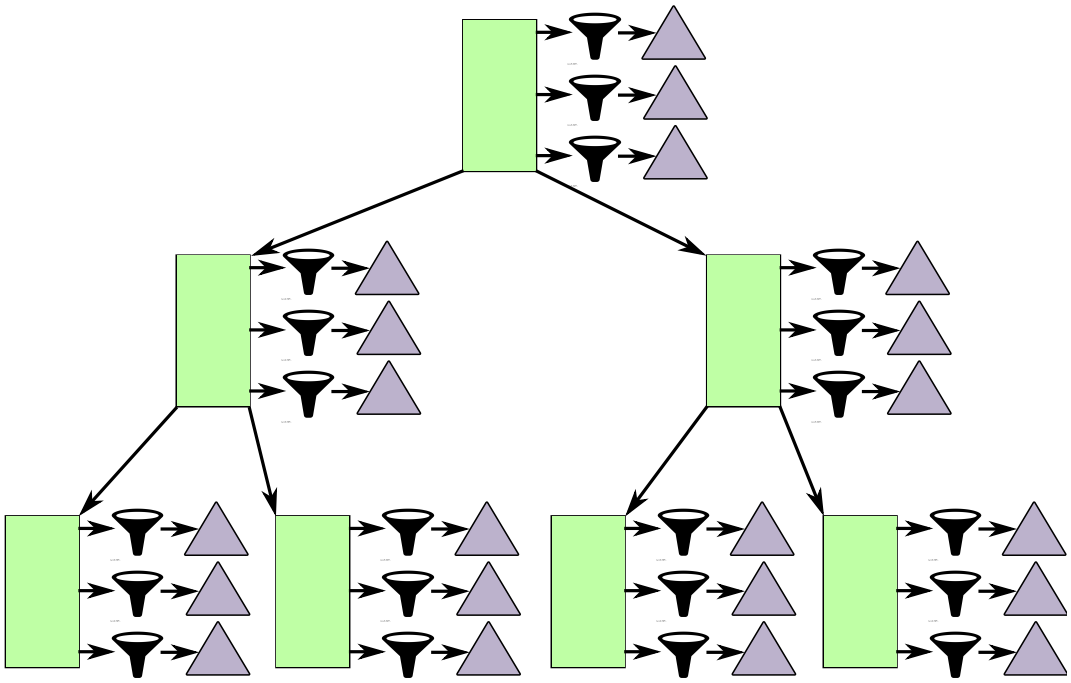


Figure 5.1. A size-tiered B^ε-tree, the data structure in SplinterDB. Trunk nodes (▭) contain pointers to children and to filters (⏷) and branches (△). In SplinterDB, branches are B-trees.

is stored in branches, and branches are immutable. Queries walk down the trunk, searching in each branch along the way. Queries use the filters to reduce the number of branches that they have to search. The branches correspond roughly to SSTables in other LSM trees, and the trunk corresponds roughly to the manifest.

When the memtable reaches a preset size threshold, it is converted into a branch and attached to the root of the trunk. The task of converting a memtable into a branch can be done in the background and without holding a lock on the trunk root. A lock on the root is required only at the end, when adding a pointer to the new branch to the root. SplinterDB supports multiple memtables so that updates can continue in a new memtable while an older one is being converted into a branch.

SplinterDB performs background actions, called ***compactions*** and ***flushes***, to reduce the total number of branches along each query path, in order to keep queries fast.

A compaction merges several branches attached to a single trunk node into a single branch, and then updates the trunk node to point to the single output branch instead of the several

input branches. Compactions consequently reduce the number of branches (and filters) that must be searched during a query. The bulk of the work of compaction—constructing the new branch—is performed "offline", i.e. without holding any locks on the trunk. Compactions lock the trunk only for the very last step—replacing the trunk's pointers to the old branches with the pointer to the new branch.

A flush copies pointers to some branches from a parent trunk node to one of its child trunk nodes. Flushes ensure that all the branches attached to a node have roughly the same size, which makes compactions efficient. SplinterDB stores several pieces of metadata in order to orchestrate flushes, queries, and compactions (see the original SplinterDB paper for details [23]). What will be important in the context of PERSISTRON is that a flush atomically moves branch pointers from a parent trunk node to one of its children and updates branch metadata in the parent and child.

Caching, logging, and checkpointing. Most storage systems maintain a page cache, either internally or by using the OS page cache (or, in some cases, both). Some key-value stores use their internal page cache only for reads [6] and disable OS-level write caching (e.g. using `O_DIRECT`), but this may increase I/O as all mutations are immediately written to disk.

Another approach, used in SplinterDB, is to use write-back caching and logical logging. High-level operations, such as inserting a key-value pair or deleting a key, are recorded in the logical log. The log may be buffered in memory and written to disk at regular intervals, upon transaction commit (possibly using group commit), or in response to an application's request for all updates to be made durable. Mutations of the key-value store's on-disk representation are written to the cache, and those dirty pages may be written to disk later. The system periodically takes a ***checkpoint***, i.e. it writes to disk a consistent snapshot of the key-value store state. This enables it to discard the log of updates that are now recorded in the checkpoint. After a crash, the system recovers by loading the last checkpoint and replaying updates from the log.

Checkpointing and logical logging allow the application to make two trade-offs. First,

the application can accelerate updates, at a cost of losing some recent data after a crash, by making the log durable less often. Second, the application can improve update performance by taking checkpoints less often. Less-frequent checkpoints increase recovery time and disk-space usage because they increase the average size of the log. But, no matter how infrequently the application performs checkpoints, it will always have to pay the overhead of logging.

The goal of this system is to use persistent memory to get instant durability with no logging overhead, no checkpointing overhead, and minimal recovery.

Locking and concurrency. SplinterDB uses page-level upgradable reader-writer locks for all of its concurrency control. Locking is integrated with the cache, i.e. a thread must lock a page in order to access it. All data structures—memtables, trunk nodes, branches, filters, and logs—are accessed through SplinterDB's internal cache.

As is common for software that has been optimized to support high degrees of concurrency, SplinterDB breaks large, complex mutations into small steps that atomically transition the data structure from one consistent state to another using only a handful of locks. For example, insertions use hand-over-hand read locks with preemptive splitting to add new items to the memtable B-tree. Insertions upgrade to write locks only when they need to perform node splits or they reach a leaf. If an upgrade fails, the insertion restarts. This supports multiple concurrent insertions while avoiding concurrency bottlenecks at the top of the memtable B-tree. Furthermore, it means that each B-tree split is an independent atomic step, as is the update of the leaf. Similarly, flushes hold locks only on the parent and child trunk nodes involved in the flush, and hold them only to update a few pointers and some metadata. Compactions hold no locks until the end, when they lock a single trunk node to update pointers.

SplinterDB performs atomic steps that involve more than one page by acquiring locks on all the pages, performing the updates, and then releasing the locks. Locked pages cannot be written to disk, so this ensures that other threads always see a consistent view and that an inconsistent combination of pages cannot be written back to disk.

Our crash-safety mechanism will exploit this structure by adding a transaction mechanism to the cache. Cache transactions will enable SplinterDB to perform its atomic steps durably in persistent memory.

5.2 PERSISTRON Overview

We now provide an overview of PERSISTRON's design. We start by laying out our objectives, and then describe the high-level design and challenges involved in realizing this design. We finally compare prior approaches for incorporating PMEM into key-value stores and explain how PERSISTRON's approach represents a new design point in this space.

5.2.1 Objectives

PERSISTRON's high-level goal is to leverage PMEM effectively in key-value stores. In particular, PERSISTRON intends to exploit PMEM's unique characteristics: 1. instantaneous durability and 2. large capacity. Additionally, PERSISTRON intends to offer (at least) the same level of performance as a purely DRAM-based configuration without increasing the system cost (while providing instantaneous durability).

5.2.2 High-level Design and Challenges

The high-level idea in PERSISTRON is to integrate PMEM into the *cache-layer* of the key-value store and reuse regular cache-management functions to effectively utilize PMEM. As we discussed earlier, most key-value stores have a DRAM cache which caches on-disk key-store pages. PERSISTRON replaces this DRAM cache with PMEM.

Figure 5.2 shows the high-level architecture of PERSISTRON. All reads and writes performed by the application go through the PERSISTRON cache layer. Because the cache is persistent, writes in PERSISTRON become immediately durable. This achieves our first goal of instantaneous durability; all updates survive once acknowledged by the key-value store. Because PERSISTRON integrates PMEM as a cache, it need not statically decide which portions of data

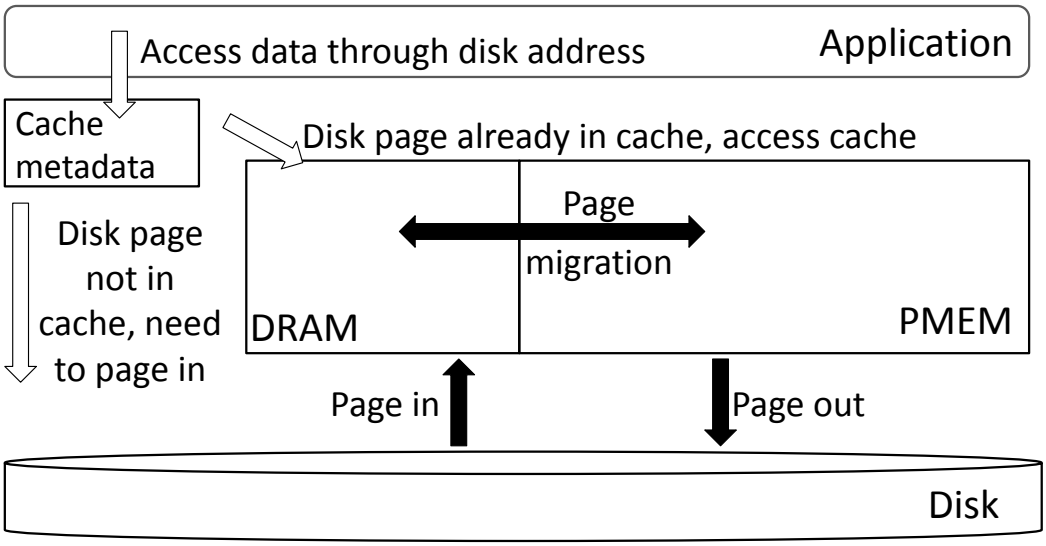


Figure 5.2. PERSISTRON architecture. The figure shows the high-level architecture of PERSISTRON. Applications access on-disk data through the caching layer, which is consisted of a DRAM portion and a PMEM portion. Pages dynamically migrate between the two portions.

reside in PMEM. Rather, cache admission and eviction dynamically move hot pages into the cache, which enables PERSISTRON to effectively utilize the large capacity of PMEM.

PERSISTRON must address two challenges to realize the above benefits. The first challenge is that of ensuring crash safety. While writes become immediately durable, they must be performed carefully such that the application is left in a consistent state after a crash. In particular, most PMEM devices can atomically write only 8 bytes of data [77]. Thus, when the application modifies a larger chunk of data, the cache layer must persist these modification as an atomic unit. PERSISTRON uses copy-on-write-based transactions to achieve this goal. Chapter 5.3 describes this mechanism.

The second challenge is that PERSISTRON must offer high performance, despite accesses to PMEM being slower than DRAM. PERSISTRON thus augments the PMEM cache with a DRAM cache, which can store frequently accessed, unmodified pages in order to expedite reads. We describe how PERSISTRON incorporates the DRAM cache in Chapter 5.4. One challenge is that an application might want to modify one or more page that are present in the DRAM cache. PERSISTRON must ensure update atomicity and durability in such cases as well. To this end, we

Table 5.1. Existing Approaches. *The table shows how different approaches exploit persistent memory in a multi-level hierarchy consisting of DRAM, PMEM, and SSD. None of the existing approaches utilize PMEM for both immediate persistence and capacity.*

Approach	Immediate durability	Exploits PMEM capacity	Grey-box	Examples
Log in PMEM	✓	×	×	HiLSM [78]
Log + parts of data structure in PMEM	✓	×*	×	MatrixKV [125], RangeKV [80], NoveLSM [67]
DRAM/PMEM volatile cache	×	✓	✓	Kassa et al. [68]
Log + mini-page DRAM/PMEM cache	✓	✓	×	Spitfire [130], HYMEM [110]
Durable, crash-safe, and large PMEM cache	✓	✓	✓	**PERSISTRON**

extend the transaction mechanisms to perform CoW between the DRAM and PMEM caches in Chapter 5.4.4.

5.2.3 Comparison to Prior Approaches

PERSISTRON's approach of using a persistent, crash-safe, and large cache layer as the foundation is a novel design point in integrating PMEM into key-value stores. We now describe prior approaches and how PERSISTRON differs from them.

One commonly used approach (see Table 5.1) moves the key-value store's log to PMEM. With this approach, all updates survive failures because the log is synchronously written to PMEM upon each modification. However, this approach does not utilize the large capacity offered by PMEM devices. Another common approach is to move parts of the key-value store data structures (e.g., the L0-level files in an LSM) to PMEM in addition to the log. This approach offers instantaneous durability and utilizes capacity better than the previous approach. However, by statically fixing what portions live in PMEM, it cannot fully exploit PMEM's capacity. Additionally, writes to other part of the data structure must still go to the underlying SSD, making background writes slower.

An alternative approach is to treat the PMEM device as a cache (in addition to the DRAM cache). This approach is able to utilize PMEM's capacity well on reads because it dynamically moves the most frequently accessed data in DRAM and PMEM. However, this approach fails to exploit the immediate persistence property of PMEM devices. All cache contents are lost upon a crash. Another approach involves restructuring pages into mini-pages, and caching them individually in DRAM. This approach utilizes write-ahead logging for immediate persistence. While it takes advantage of PMEM capacity and persistency, the system needs to be mini-page aware, which may result in a significant amount of code changes.

These prior design points indicate that a cache is the right way to exploit the capacity benefits of PMEM. One approach to achieve instantaneous durability in this cache-based approach is to move just the log into PMEM and direct log writes to PMEM. While this approach makes log writes instantly durable, background writes (e.g., compaction) must go to the underlying storage, forgoing performance.

PERSISTRON offers a new design point: by integrating PMEM into the cache layer, unlike prior approaches, PERSISTRON can effectively utilize the large capacity of PMEM with no or little code change. Compared to the read-only caching strategy, PERSISTRON makes all the writes go through the cache (in addition to reads) and makes the cache writes durable and crash-safe.

5.3 PERSISTRON PMEM Cache

In this section, we discuss how PERSISTRON manages the PMEM cache and enables atomic updates to pages stored in PMEM.

5.3.1 Cache Internals

Each slot in the cache logically consists of a data page and metadata about the slot. The metadata is stored in the entry table, and includes the disk address of the page currently stored in the slot and the status bits for the page (e.g. dirty, read-locked, etc.). The cache also contains

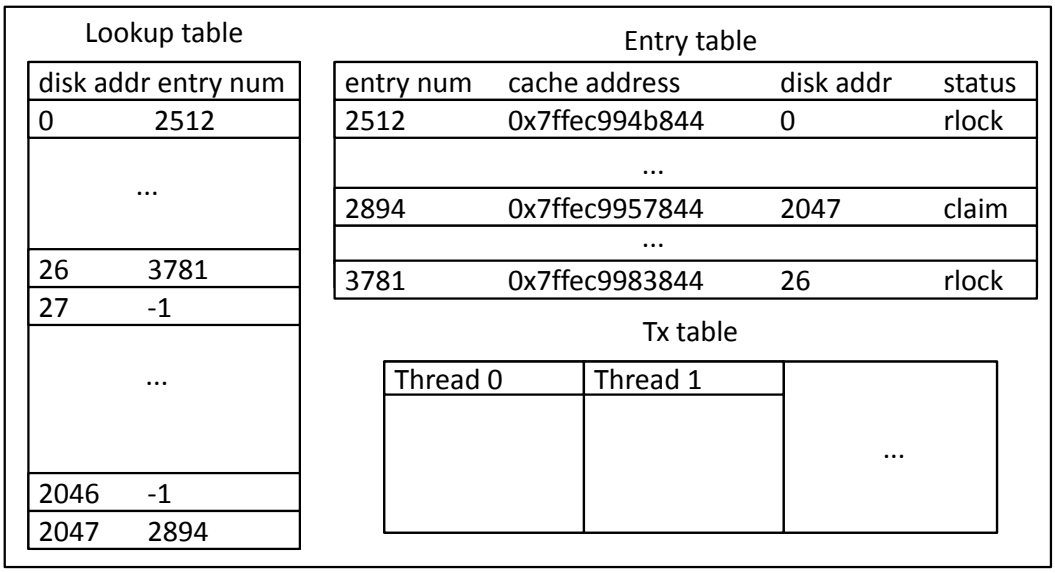


Figure 5.3. PERSISTRON Cache Metadata. The figure shows the metadata structures maintained in PERSISTRON's cache. The entry table stores entry metadata for each cache entry. The lookup table stores the mapping from disk address to cache entry number. The transaction table records per-thread transaction management metadata.

a page table mapping disk addresses to slot indexes. When the application requests a page (by specifying its disk address), the cache uses the page table to find the page in cache (or detect that it is not present).

The cache also manages locking of pages. Applications cannot access a page without read-locking it. They may then upgrade to a write-lock.

In the traditional setting, with just a DRAM-only cache, the entry table and the lookup table are volatile. However, with PMEM cache, these structures must survive crashes. PERSISTRON thus maintains the entry table in PMEM and uses it to reconstruct the page table after a crash.

5.3.2 Transaction Mechanisms

As we discussed, an application (such as a key-value store) moves its internal data structures from one consistent state to the next by modifying a few pages in the cache in an atomic fashion. With these pages now residing in PMEM, PERSISTRON must ensure these updates are failure atomic. To do so, PERSISTRON employs transactions.

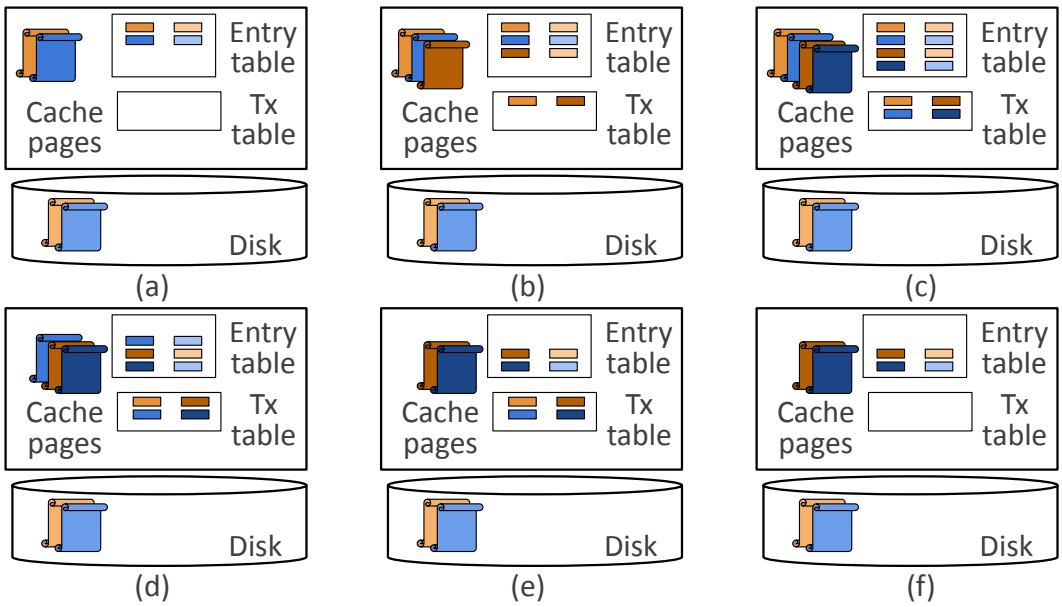


Figure 5.4. PERSISTRON Transactions. The figure shows the sequence of events that happen during normal transaction execution. If failure happens during (a)-(c), PERSISTRON rolls back the transaction. Otherwise, the transaction will be rolled forward.

PERSISTRON treats every atomic set of page mutations performed by the application as a cache transaction. To ensure failure atomicity, PERSISTRON needs to track which pages are being modified in a transaction. If a crash occurs, these pages must be brought back to their original state. To do so, PERSISTRON relies upon copy-on-write and employs a *transaction table* (see Figure 5.3) to record which pages are being modified. Since the transaction table is needed for recovery, it is stored in PMEM.

At a high level, upon a transaction start, PERSISTRON creates an entry in the transaction table. When the application acquires a write lock on a page, PERSISTRON records the entry number of this page in a transaction entry (say T). Further, PERSISTRON copies the page to a new cache slot and changes the lookup table to point to this new slot; all modifications now go to this new page copy. PERSISTRON also records the new cache entry number in T. When the transaction commits, the entries for the old pages are marked as unmapped. Finally, the entry in the transaction table is erased.

5.3.3 Crash Recovery

If a crash occurs when transactions are in progress, PERSISTRON must be able to recover to a consistent state. The information in the transaction table helps in this regard. Upon recovery, PERSISTRON scans the transaction table, one transaction entry at a time, and decides whether to rollback or rollforward the transaction. The presence of a transaction entry indicates that the system crashed when the transaction was running. Thus, PERSISTRON by default decides to rollback the transaction. If any page in the transaction cannot be rolled back to its old state, then the transaction is rolled forward. A page cannot be rolled back if the old entry for the page is marked as unmapped in the entry table; this means the transaction had committed and the system was in the process of unmapping the old entries (before the crash).

If PERSISTRON decides to rollback a transaction, it simply marks the new entries in the entry table as unmapped. On the other hand, if the transaction is rolled forward, then the all the remaining old entries that were not yet unmapped are marked unmapped.

Figure 5.4 shows the sequence of events that happen during normal operation. PERSISTRON starts execution from Figure 5.4(a). At the brown page lock acquire, it adds the old entry number to the transaction table. Then, it allocates a new page and adds it to the entry table. Finally, it copies the old page content over and add the new entry number to the transaction table. At this point, it reaches the state in Figure 5.4(b). The same process continues as PERSISTRON writes to the blue page and the transaction goes into the state in Figure 5.4(c). The execution is finished at this point. To end the transaction, PERSISTRON unmaps the old entries and removes them from the entry table. The unmap of the brown and the blue page makes the transition from Figure 5.4(c) to Figure 5.4(d) and Figure 5.4(d) to Figure 5.4(e), respectively. Finally, PERSISTRON commits the transaction by releasing all held locks and cleaning up the transaction table.

If a crash happens during a transaction in its state between Figure 5.4(a) to Figure 5.4(c), PERSISTRON rolls back the transaction. Once a transaction leaves the state in Figure 5.4(c),

PERSISTRON always rolls it forward in case of a crash.

5.3.4 Inferring Transactions

So far, we described how transactions work but not how applications start and end a transaction. Since applications use locks to access cache pages, PERSISTRON can infer the transaction boundary automatically. In particular, PERSISTRON infers the transaction boundaries from lock acquisition and release points. PERSISTRON starts a transaction for a thread when the thread acquires its first write-lock. PERSISTRON records further lock acquisitions. Every time a thread acquires a write lock on a page, PERSISTRON creates a copy and adds the old and the new page entry numbers in the transaction table as described above. PERSISTRON treats the the last lock release as the end of the transaction. At that point, PERSISTRON commits the transaction, as described above.

However, this scheme alone is not sufficient for crash safety. Consider the following scenario. Thread T_1 takes a lock on pages P_1 and P_2 and has released the lock on P_2. P_2 thus is now visible to other threads. If a crash occurs at this point, T_1's transaction must (and will) be rolled back because no old entries would be marked unmapped (as we unmap only after transaction commit). So, the system would be in a state where a transaction rolled back but its effects were visible to others.

To avoid this problem, PERSISTRON ensures that transaction effects are visible to other threads only if the transaction can be rolled forward after a crash. To achieve this end, PERSISTRON intercepts all lock releases and delays the actual release until the end of the transaction (i.e., when the last lock is released). In effect, PERSISTRON changes the locking scheme from 2-phase locking to strict 2-phase locking. With this new scheme, in the above example, P_2's lock would not be released (and would be invisible to other threads) and thus is safe to rollback the transaction upon recovery.

Inferring transaction boundaries works well in practice because most applications use two-phase locking for isolation guarantees. Nonetheless, our current implementation cannot

work well with other locking schemes; we leave this extension as an avenue for future work. One concern with our technique of delaying the actual lock release is that of reduced concurrency compared to a early release scheme. However, as we show in our evaluation, adding this delay has little or no overhead.

5.3.5 Non-Transactions

Sometimes, applications might not require atomicity of updates. A typical intended use case is building new data structures that will be hooked into the main data structure later. PERSISTRON allows applications to avoid transactional overheads for these cases. The application can declare "non-transactions", which disable PERSISTRON's copy-on-write and other transaction mechanisms, reducing overheads.

With non-transactional updates, it is possible to see a partial set of updates after a crash. However, when used to build a new data structure that is not connected to the main data structure, these partial updates do not cause any unsafety. However, these new pages may need to be deallocated; otherwise space can be leaked. Therefore, applications must perform such deallocations for non-transactional updates. We discuss how PERSISTRON handles this in (Chapter 5.5).

5.4 DRAM Caching

Because even a large PMEM cache will be slower for in-cache operations than a DRAM cache, PERSISTRON uses a DRAM cache in addition to its PMEM cache. This cache is designed to hold frequently accessed clean pages, while letting the PMEM cache hold both pages that are dirty or just warm. In this section, we describe how pages move between the caches and how PERSISTRON maintains consistent persistent state across them.

5.4.1 Clean Hot Page Caching

PERSISTRON's DRAM cache is designed to hold clean frequently accessed pages. This is because clean pages are recoverable from disk in the case of a system failure, and frequently accessed pages benefit the most from DRAM's performance.

When a clean page is accessed in PMEM, PERSISTRON determines whether it should be moved into DRAM. PERSISTRON's cache layer already uses a clock algorithm to determine hotness for eviction, and it reuses this hotness heuristic to determine whether a clean page should be migrated to DRAM. Specifically, each cache page entry (in the entry table) has an *accessed* bit that is set when the page is read-locked. Periodically, PERSISTRON walks through these entries and clears the bits. When a page is accessed, if the *accessed* bit is already set, then it indicates that the page is hot. To determine if a page is synced to disk, PERSISTRON maintains a dirty bit that is set whenever the page is write-locked and cleared when it is written back to disk.

If both the above conditions are met, PERSISTRON migrates the page from PMEM to DRAM. First, PERSISTRON acquires a write lock on the PMEM page to prevent other threads from accessing the page. Then, it allocates a new locked page in DRAM and copies the page contents. After this, it frees the old page in PMEM. Finally, the lock on the DRAM page is released, making the page accessible in DRAM.

5.4.2 Warm Page Eviction

Because the DRAM cache is limited in size, pages must be evicted. Since the DRAM cache consists of pages that have been frequently accessed, these evicted pages are likely still warm, so PERSISTRON simply moves pages evicted from the DRAM cache into the PMEM cache. This offers two advantages: since the PMEM cache is large, this can avoid unnecessary disk accesses, and upon a crash recovery, PERSISTRON can start with a warm cache.

The procedure to move warm pages to PMEM is similar to that of moving hot pages into DRAM. First, PERSISTRON acquires a write lock on the DRAM page, preventing concurrent

accesses. Next, it allocates the page in PMEM and copies the DRAM page contents to the new PMEM page. Lastly, it frees the DRAM page and unlocks the PMEM page.

5.4.3 Hybrid Cache Control

Pages move from disk to PMEM, i.e. they go from cold to warm.

The two caches operate independently, which introduces a tricky race condition on page-in, which can occur as follows. Cache lookups first check th DRAM cache, then the PMEM cache, and then bring the page from disk into PMEM if it is not in either cache. Since the two caches operate independently, the two lookups are not performed atomically. Thus cache lookups may race with page migrations from one cache to another, causing a thread to conclude that a page is in neither cache when in fact it simply got migrated from one to another.

To avoid this race condition, threads set up the slot and page table for a page-in and then check the other cache one more time before issuing the I/O. By locking the entry for a page in one cache, the thread precludes any further migrations during this last check.

5.4.4 Transactions Involving DRAM Pages

We need to extend PERSISTRON's transaction mechanism to correctly handle write-locks on pages that are currently in the DRAM cache.

We augment the transaction mechanisms described earlier (Chapter 5.3.2) to support transactions involving pages in DRAM as follows. When a thread acquires a write lock on a DRAM page, PERSISTRON records the disk address of the page in the transaction table, and marks that transaction entry as involving a DRAM page.

Upon a transaction commit, PERSISTRON copies all involved pages from DRAM to PMEM. PERSISTRON first allocates a locked page in PMEM and copies the content. Then it records the PMEM entry number in the transaction table and removes the page from DRAM.

Note that a single transaction might modify pages in PMEM in addition to modifying DRAM pages. In this case, when a PMEM page is modified, PERSISTRON uses the copy-on-

write and transaction mechanisms described earlier. During the commit phase, the DRAM pages must be copied to PMEM before PERSISTRON can unmap any old PMEM page (created by copy-on-write). Once, PERSISTRON has copied all the pages from DRAM to PMEM, it proceeds to clean the old PMEM pages as described earlier (Chapter 5.3.2). Then, PERSISTRON releases all the locks that have been held and commits the transaction.

By persisting all the DRAM pages before beginning to erase the old PMEM pages, PERSISTRON guarantees that, once it starts to erase PMEM pages, it will be able to roll forward if the system crashes while commiting the transaction.

If, after a crash, PERSISTRON decides to roll back a partially committed transaction, then it erases any DRAM pages that were copied to PMEM, effectively rolling them back to the on-disk version. It rolls back PMEM-pages as described above. If it decides to roll forward, then at least one old PMEM page must have been freed before the system crashed, which means that all the DRAM pages were copied to PMEM before the crash. So PERSISTRON needs only to release all the old PMEM pages, as described above.

5.4.5 Non-Transactions Involving DRAM Pages

As described earlier, applications might sometimes not require atomicity of updates. PERSISTRON reduces the overheads on these updates by allowing applications to mark regions of code as non-transactions. When DRAM pages are involved in a non-transactional update, PERSISTRON needs to ensure the update is persisted once it is finished. During non-transaction execution, PERSISTRON migrates a DRAM page to PMEM at its write lock release.

Similar to PMEM pages during non-transactional updates, the DRAM pages may also be partially persisted in PMEM before non-transaction commit. These partial updates will be cleaned up the same way as discussed in Chapter 5.3.5.

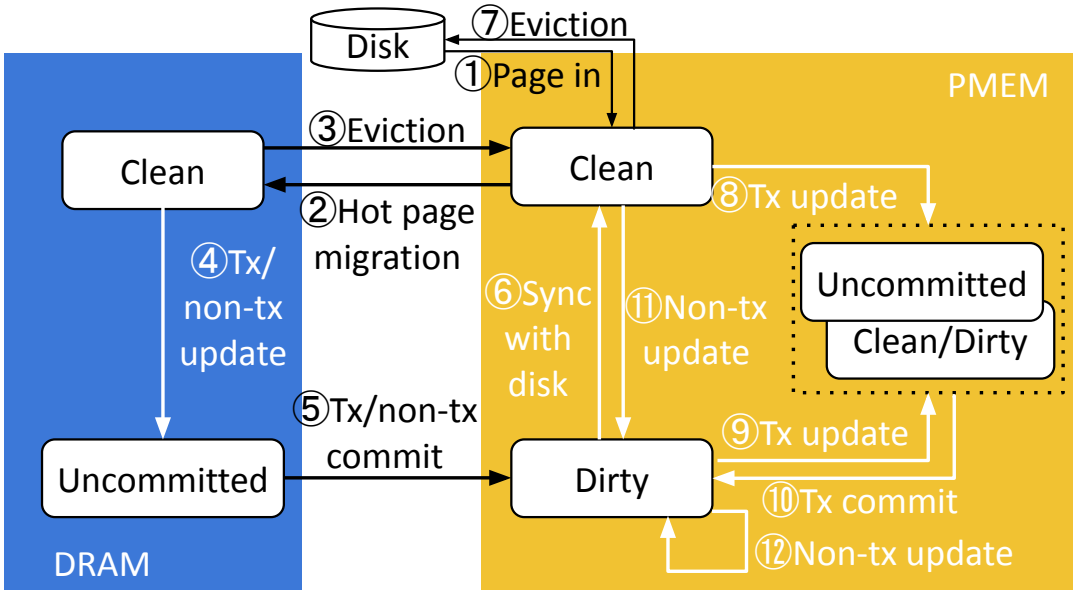


Figure 5.5. PERSISTRON page life cycle. The figure shows the change of a disk page's in-cache state. The state changes through page read, update, transaction commit, synchronization with the disk, page in, and eviction.

5.4.6 Life Cycle of a Page

Figure 5.5 shows the entire life cycle of a page. When the page is first paged-in from the disk, it is admitted into the PMEM cache (①). If these clean pages are accessed frequently, they are migrated to DRAM as clean pages (②). They can be read or written. When a clean DRAM page is not accessed very frequently, it will be evicted into PMEM (③). DRAM pages can be modified in a transaction. Before the transaction commits, they are uncommitted and stay within DRAM (④). At transaction commit, the uncommitted pages are written to PMEM (⑤). We call these modified pages within PMEM dirty because they are persisted not yet synced with the disk copy and so cannot be evicted from the cache. A dirty page when it is synced with the disk becomes clean and then can be evicted (⑥, ⑦). A clean or a dirty page in PMEM can be modified. When the transaction is in progress, an additional uncommitted copy is created by the transaction mechanism (⑧, ⑨). When the transaction commits, the uncommitted page becomes a dirty page and the old clean or dirty page is unmapped (⑩). Non-transactional updates do not create copies; clean and dirty pages are both modified in-place and thus become (stay) dirty (⑪, ⑫).

5.5 PERSISTRON Integration with SplinterDB

This section describes how we used the persistent cache in SplinterDB.

SplinterDB contains four types of data structure: the trunk, memtables, branches, and filters. Branches and filters are built offline and linked into the trunk only once they are complete. Once they are linked into the trunk, they remain static until they are no longer referenced and can be deallocated. Thus we use non-transactions to build these data structures. We then use a transaction to link them into the trunk. Updates to the memtables and trunk are performed transactionally.

The main challenge that we need to address is recovering the allocation state of the disk after a crash. In particular, we need to ensure that any pages allocated for an in-progress branch or filter get freed after the crash. SplinterDB maintains two types of allocation data structure: a global in-memory reference-count table and, for each memtable, branch, filter, and the trunk, a per-structure allocation list. The reference-count table is used to satisfy allocations. The per-structure allocation lists are used for deallocations.

After a crash, PERSISTRON reconstructs the reference-count table by walking the trunk and all the per-structure allocation lists. Note that this does not require walking the branches, filters, or memtables themselves. Thus any in-progress branches or filters automatically get discarded and freed after a crash.

The allocation lists for memtables and the trunk are updated as part of the same transaction that updates their corresponding memtable or trunk, and hence they are always consistent.

The second challenge is to verify SplinterDB satisfies PERSISTRON locking requirements. As described in Chapter 5.1, we find SplinterDB decomposes its updates into small steps that use only a handful of locks. The main violation we found was in the flushing code, which moves branches down the trunk by acquiring write-locks hand-over-hand as it goes. We modified the code to downgrade to read locks at each step, which enables the transaction inferrence algorithm to infer separate transactions for each step.

The last type of change we made to SplinterDB is non-transaction boundaries annotation, which includes compaction and flushing operations. These boundaries take less than 20 LOC across the whole codebase.

5.6 Evaluation

We evaluate PERSISTRON to answer the following questions:

- Can PERSISTRON offer instant durability while matching or exceeding the performance of an all-DRAM cache by using a hybrid DRAM/PMEM cache of equivalent cost?
- How does PERSISTRON scale with more threads?
- How do the different optimizations (and optimization hints, such as non-transactions), affect performance?
- How does PERSISTRON performance compare to other key-value stores targeting a three-level cache hierarchy of DRAM, PMEM, and SSD?

5.6.1 Experimental Setup

All results are collected on a PowerEdge R640 with 2 16-core 2.30 Ghz Intel Xeon(R) Gold 5218 CPUs, 187 GiB RAM and a 1 TiB Dell Express P4510 Flash NVMe.

We downloaded SplinterDB and MatrixKV from Github. We use the default configuration for MatrixKV, except that we enable direct I/O for reads and writes, and enable block alignment. We use the default configuration for SplinterDB.

PERSISTRON's current implementation is built for commercially available platforms with eADR [53] and thus it does not issue cache flushes, such as CLWB instructions. For a fair comparison, we have disabled the cache-flush instructions in MatrixKV.

For standardization, our benchmarks consist of runs of the Yahoo Cloud Serving Benchmark (YCSB) [24]. YCSB has a collection of standard workloads, Load (uniformly random insertions) and Runs A (50% Zipfian updates and 50% Zipfian point queries), B (5% Zipfian updates and 95% Zipfian point queries), C (Zipfian point queries), D (95% recent point queries,

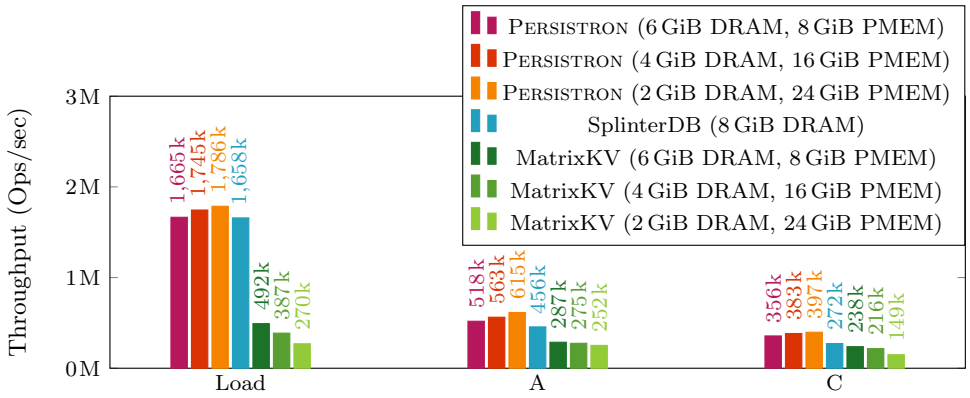


Figure 5.6. Throughput of cost-equivalent configurations of PERSISTRON on YCSB Load, A and C. Higher is better.

5% insertions), F (50% Zipfian point queries, 50% Zipfian read-modify-writes) and E (95% scans of length 1–100, 5% insertions).

Unless mentioned otherwise, all experiments use 24B keys and 100B values, as these are representative of common real workloads [4, 8]. All workloads use a 60 GiB dataset (520 M items) and perform about 60 GiB of logical reads and writes: 520 M operations, except YCSB E, which performs 14M operations, 95% of which are scans of average length 50. For our DRAM-only experiments, we use a 8GiB cache, which is about 12% as large as the database, which a typical ratio of DRAM to database size, as reported in real workloads [4, 8]. For our DRAM/PMEM caches, we generally keep the total dollar cost the same, assuming PMEM is a fourth the cost of DRAM.

5.6.2 Cost-Equivalent Hardware Configurations

The goal of this experiment is to determine whether PERSISTRON can offer instant durability and equivalent or better throughput to an all-DRAM configuration of SplinterDB. We do this by benchmarking PERSISTRON on several different "cost-equivalent" hardware configurations. Specifically, we assume that PMEM costs 4× less than DRAM per gigabyte, and we consider several different configurations where we successively replace 2 GiB of DRAM with 8 GiB of PMEM. We also benchmark SplinterDB on an all-DRAM configuration, as a baseline. All systems were configured to use 8 threads.

For each hardware configuration, we run the core YCSB Load, A, and C workloads. We focus on these workloads because they cover a pure insertion, a pure query, and a common mixed workload. (We present full YCSB benchmark results in Sec. 5.6.3.)

The results are shown in Figure 5.6. PERSISTRON's load performance at least matches SplinterDB on all configurations and generally improves as much PMEM is added, up to an 8% more with 24 GiB. Note that PERSISTRON is providing instant durability for each operation, whereas SplinterDB may lose recent updates after a crash.

For queries, PERSISTRON outperforms SplinterDB across all hardware configurations by 30–46%, with performance improving as more DRAM is traded for PMEM.

While adding more PMEM at the expense of DRAM generally improves performance, we expect that deployments are likely to have lower PMEM-to-DRAM ratios available. Therefore, the rest of the evaluation will focus on the 4 GiB DRAM/16 GiB PMEM configuration, because we anticipate this configuration to be a commonly used one in practical deployments.

In the 4 GiB DRAM/16 GiB PMEM configuration, PERSISTRON insertions are about 5% faster than SplinterDB and queries are about 41% faster. Thus, we conclude that PERSISTRON does achieve its primary design objective of matching or exceeding the performance of a state-of-the-art all-DRAM KV store while providing instant durability and without increasing hardware costs.

We also benchmark MatrixKV, based on RocksDB, across the same hardware configurations. MatrixKV load performance is about 3.5× slower than PERSISTRON's, and queries are about 40% slower than PERSISTRON's. Since MatrixKV is based on RocksDB, it is expected to be much slower than SplinterDB. As for its declining performance with increasing PMEM, we believe this follows from MatrixKV's static allocation of different data structures to DRAM and PMEM. PERSISTRON can move hot pages to DRAM and store (potentially more) warm pages in PMEM to accelerate queries of popular items, but MatrixKV cannot.

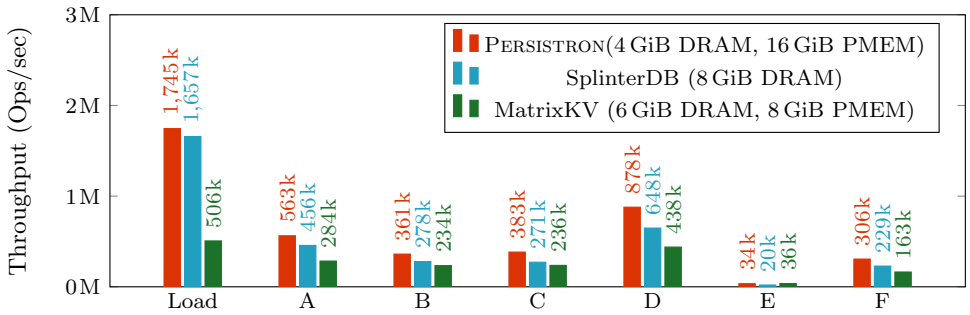


Figure 5.7. Throughput of PERSISTRON on YCSB. Higher is better.

5.6.3 YCSB

In these experiments, we run the full YCSB benchmark suite using each system's best cost-equivalent hardware configuration, as found in Figure 5.6. All systems were given 8 foreground threads. SplinterDB and PERSISTRON were configured to perform all work (e.g. compactions) on foreground threads. MatrixKV was given an additional 7 background flushing threads and 16 background compaction threads.

The results are shown in Figure 5.7.

Despite providing instantaneous durability, PERSISTRON has 5% higher throughput of SplinterDB on YCSB Load, and more than 3× that of MatrixKV.

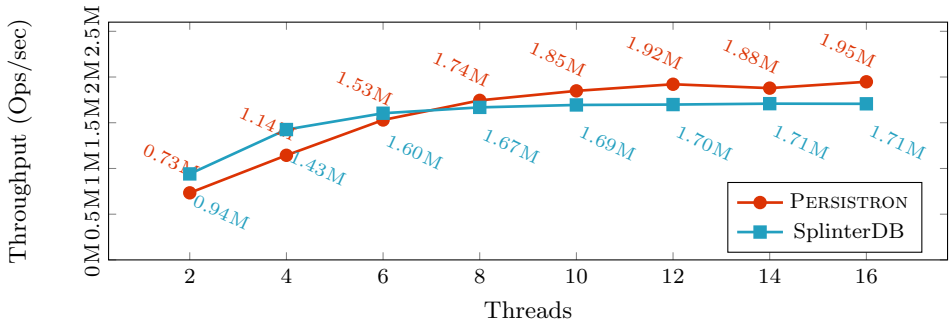
On the Run workloads, PERSISTRON has up to 40% higher throughput than SplinterDB, demonstrating that its larger (cost-equivalent) PMEM cache can deliver substantial performance improvement.

MatrixKV slightly outperforms PERSISTRON on workload E, which consists of 95% small scans and 5% inserts. We believe this is due to the relatively slow SSD used in these experiments. SplinterDB (and hence PERSISTRON) are tuned for the throughput and latency of Optane-based NVMe drives.

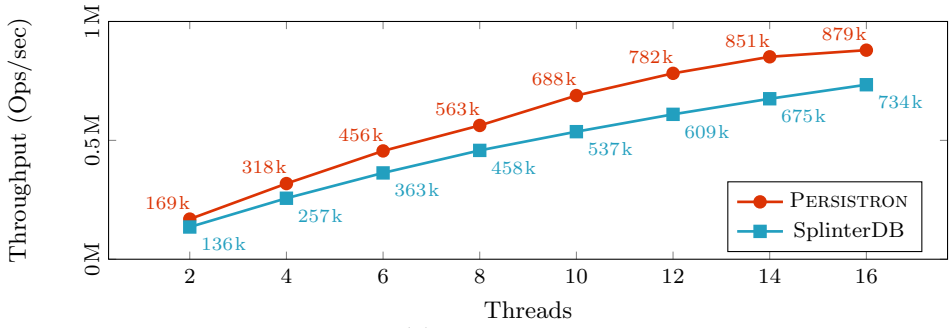
5.6.4 Scaling

Figures 5.8a to 5.8c show how SplinterDB and PERSISTRON throughput scales with increasing threads on YCSB Load, A, and C.

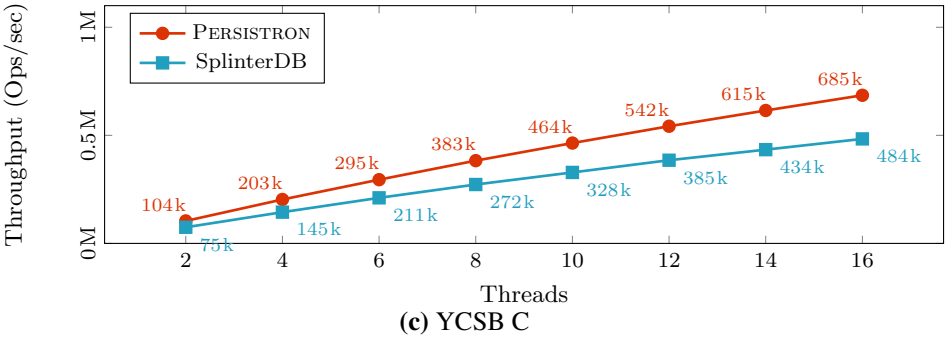
On the Load workload, PERSISTRON is slower than SplinterDB at low thread counts, but



(a) YCSB Load



(b) YCSB A



(c) YCSB C

Figure 5.8. Scaling throughput of PERSISTRON on YCSB Load, A and C. Higher is better.

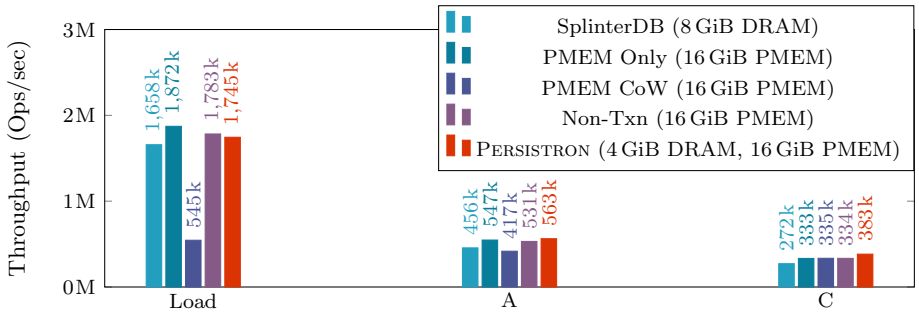


Figure 5.9. Throughput of individual optimizations on YCSB Load, A and C. Higher is better.

faster at high thread counts. This makes sense because, at low thread counts, SplinterDB and PERSISTRON are both CPU (and memory throughput) bound. Furthermore, since the workload is entirely insertions, it primarily uses PERSISTRON's PMEM cache, not the DRAM cache. Thus, the reduced performance primarily reflects the lower performance of PMEM versus DRAM. With higher thread counts, however, SplinterDB becomes I/O bound before PERSISTRON because PERSISTRON is able to exploit PMEM's capacity to avoid I/O to disk. As a result, PERSISTRON outperforms SplinterDB.

For lookups, PERSISTRON outperforms SplinterDB across the board by roughly 40% due to its larger (PMEM+DRAM) cache. Furthermore, PERSISTRON's mechanisms for caching hot pages in DRAM effectively mitigates the lower performance of PMEM relative to DRAM. Overall, PERSISTRON is able to exploit the capacity of PMEM while minimizing its performance downsides.

For Run A, a mixed workload, PERSISTRON is 20-24% faster than SplinterDB across the board.

5.6.5 Impact of Individual Optimizations

In this subsection, we examine the performance impact of different features and optimizations of PERSISTRON.

This progression starts with SplinterDB on DRAM. The first modification is to simply run the system on a larger PMEM cache (PMEM Only). This improves performance somewhat by increasing the cache size, but it does not provide consistent instant durability. This is

achieved by performing copy-on-write to PMEM pages, together with transactions (PMEM CoW). Unsurprisingly, this add substantial overhead to update workloads. However, providing non-transaction boundary hints recovers almost all of that performance (Non-Tx). Adding the DRAM cache provides a 14% improvement on a pure read workload (C) at a 2% cost to insertions (PERSISTRON).

5.7 Related Work

Key-value Stores for Limited Hierarchies. Many key-value stores have been built to take advantage of PM. However, most prior systems target a two-level hierarchy with DRAM and PM. Earlier systems such as Echo [5] and more recent systems such as HiKV [118], Bullet [55], FPTree [96], HybridKV [35], ChameleonDB [128], and FlatStore [16] fall into this category. SLM-DB [66] targets a two-level hierarchy with PM and SSD. SLM-DB organizes on-disk data in a single level (i.e., only L0 level in the LSM). SpanDB is optimized for a two-level hierarchy with a fast NVMe SSD and a slow SSD [13]. Recent research [75] has explored a cost model that can be used for targeted data placement. While this work shares cost-driven motivation similar to that of PERSISTRON, it limited the placement configuration between PM and DRAM. Unlike all these prior systems, PERSISTRON targets a more realistic use case where the database cannot fit just in DRAM and PM but needs a SSD layer too. By building a PM cache layer, PERSISTRON provides a way to cache frequently accessed portions of a large database in PM.

Exploiting PM in a Multi-level Hierarchy A few recent key-value stores target a memory-storage hierarchy with DRAM, PM, and SSD. However, these prior systems either utilize PM for immediate persistence or capacity but not both. For example, NoveLSM [67] proposes to keep memtables in PM (in addition to the DRAM memtables). These PM memtables are mutated in place, obviating the need for logging. However, logging is enabled for DRAM-backed memtables. Having many memtables avoids write stalls. PERSISTRON, in contrast, mutates all memtable contents in PM directly, completely removing the need for logging. MatrixKV [125],

stores L0 tables in an LSM in PM. It also maintains the log in PM for immediate persistence. Similar to MatrixKV, RangeKV [80] also maintains the log and L0 tables in PM. HiLSM keeps the log in PM and builds a DRAM index for the PM log [78]. All these systems only exploit PM for immediate persistence but not its large capacity. These approaches also statically fix what portions of data reside in PM. PERSISTRON, in contrast, exploits both immediate persistence. Also, through its persistent cache approach, PERSISTRON dynamically decides which portions reside in PM. Another approach exploits the large capacity of PM. However, these approaches do not exploit PM's persistence. For example, recent work at Facebook [68] uses PM as an extended volatile cache in RocksDB. Similarly, PolyEMT [93] exploits PMEM capacity by using it as a volatile cache. Additionally, it also uses a portion of PMEM as a separate write-cache to the SSD to speed up writes. However, PolyEMT does not offer instantaneous durability and crash safety. Only forced pages are made durable and crash consistency is left to applications.

HYMEM and Spitfire [130, 110] exploit PM capacity and persistency at the same time, but require substantial code changes. Both Spitfire [130] and HYMEM [110] change the data layout by restructuring pages into mini-pages, which can be cached individually in DRAM. This requires the rest of the storage system to be modified to be mini-page aware. Moreover, because of the data layout complexity, they are either single-threaded, or require complicated additional concurrency control. On the contrary, PERSISTCACHE requires no data layout changes, poses no limitation to existing optimizations, and requires no additional concurrency control.

General approaches to exploit PMEM persistency, performance, and usability In the past decades, researchers have proposed PMEM-specific file systems [116, 121, 65, 129], user-level libraries [20, 12, 97, 112, 43, 86, 30, 84, 10, 117, 48, 72, 123, 52], hardware support [70, 69, 7, 63, 45, 31, 107], and data structures [21, 94, 42, 124, 14, 111]. However, most systems are single-level PMEM systems. So they are bounded by memory capacity, and limited by PMEM media performance. A few proposed to use DRAM to accelerate PMEM accesses [84, 30, 117]. But they generally rely on using an additional copy of the in-memory data structure to maintain crash safety and achieve high performance, which at least doubles the memory consumption.

5.8 Conclusion

This paper presents PERSISTRON, a key-value store that incorporate PMEM via a crash-safe, durable cache. PERSISTRON, unlike prior approaches, presents a way to fully exploit PMEM characteristics, and does so with no to little changes to applications. PERSISTRON's new techniques enable it to achieve instant durability, crash-safety, and high performance. More broadly, our design makes the case for re-using well-known systems abstractions (such as the cache) to incorporating new storage hardware. The challenges in managing the PMEM hardware (e.g., crash safety and slow accesses) can be handled within this abstraction. Consequently, this view has great practical utility because it does not require intrusive changes to applications.

Acknowledgements

This chapter contains material from "Persistron: Leveraging Persistent Memory in Key-value Stores via a Crash-safe and Durable Cache", by Yi Xu, Ramnatthan Alagappan, Aishwarya Ganesan, Rob Johnson, and Alex Conway, which was submitted for publication at 2024 International Conference on Management of Data. The dissertation author is the primary investigator and first author of this paper.

The author would like to thank the co-authors for their help and support. The author would also like to thank the members of VMware Research Group and the NVSL research group for their insightful comments.

Chapter 6

Conclusion

Persistent memory (PMEM), like Intel Optane or persistent CXL.mem devices, presents new opportunities and challenges for storage systems. PMEM is directly accessible by applications through load and store instructions, and its durability allows data to persist beyond process lifetimes, system reboots, and unexpected power failures. However, effectively leveraging this capability poses significant challenges. Applications must ensure data consistency (failure-atomicity) during power failures. Implementing applications that meet these requirements is both challenging and prone to errors, and achieving high performance during regular operations adds further complexity.

My dissertation focuses on persistent memory (PMEM) systems, particularly on designing PMEM systems that guarantee failure-atomicity with minimal runtime overhead and user programming efforts [123, 51, 122].

In this dissertation, we proposed easier-to-use and more efficient PMEM systems. We designed three systems that ensure PMEM data consistence from different levels.

In Chapter 3, we introduced Clobber-NVM, which reduces logging cost in traditional PMEM transaction system. In a traditional PMEM transaction system, data consistency is usually ensured by maintaining a copy of the transaction's write set during normal execution (commonly done through undo/redo logs) and then recovering using these logs. However, this logging process often imposes significant performance overhead.

Clobber-NVM identifies *clobber writes* in the application code. Re-executing a transaction with a clobber write produces different results. Clobber-NVM significantly reduces the logging cost by creates an undo log only before each clobber write occurs. In the event of a failure, the system can recover to a consistent state by restoring the undo logs of the clobber writes and reexecuting interrupted transactions from the beginning. Thereby, it overwrites PMEM addresses that may have become inconsistent (These addresses were written to when the transaction was interrupted).

We adapted Memcached and several other benchmarks to run on top of the system. The evaluated benchmark results demonstrate that, compared to our system, classical undo logging logs up to 42.6× more bytes and requires 2.4× to 4.7× more expensive ordering instructions. It improves end-to-end performance by up to 2.6×.

In Chapter 4, we presented zero effort PMEM programming. It provides PMEM data consistency from process level. By using Clobber-NVM, compiler assistance avoids most programming efforts required for application programmers. However, it is still based on the traditional PMEM transaction programming model. In a traditional PMEM transaction system, strict concurrency restrictions are imposed to the application code: applications need to make sure locks are acquired and released in a conservative, strong strict two-phase locking pattern: that is, locks protect memory locations from data races; transactions acquire the associated lock at the transaction begin and in a fixed order (to prevent deadlock); transactions hold the locks until transaction commit. They only provide true data consistency under the assumption the programmer follows these constraints.

A significant portion of the existing codebases do not follow the constraints. Zhuque [51] is the first zero effort PMEM programming system. Because of its support for porting existing code, We were able to run a newer version of Memcached (which violates the concurrency requirement of the transaction system) directly on Zhuque, and found it to be more than 7.5× performance of the best-performant older version of persistent Memcached.

By making all process states persistent, Zhuque treats power failure as a recoverable

exception. At recovery, the process can install a normal error handler for this signal which cleans up and exits, or performs more complex application-specific recovery; by default, program execution simply continues at the point of failure.

Due to the advancement of PMEM devices and platforms, they now support flush-on-fail semantics (e.g. eADR for NVDIMMs or GPF for CXL devices). These semantics effectively allow the CPU cache to be treated as persistent. By leveraging the features of the new hardware, Zhuque has shown comparable performance to its volatile counterpart, on most evaluated benchmarks.

In Chapter 5, we presented a grey-box DRAM/PMEM caching system. It can be easily integrated into existing key value stores, while ensuring that the contents of the PMEM cache always remain consistent. The grey-box solution requires only minor adjustments to the application (storage system) code. This enables the system to achieve speed, cost, capacity, ease of programming, and durability at the same time.

The system combines a straightforward copy-on-write DRAM/PMEM cache and migration policy with hints from the application that enable it to avoid copy-on-write overheads when they are not needed for crash safety. Thus almost all code changes are in the cache. But by taking only a few hints from the application code, the copy-on-write mechanisms are efficiently bypassed.

From the performance perspective, Persistron is able to match or slightly improve on the update performance of an all-DRAM configuration of SplinterDB while providing instant durability. Further, Persistron, via its larger DRAM + PMEM cache, improves query performance by up to 46% over a cost-equivalent all-DRAM configuration of SplinterDB.

Acknowledgements

This chapter contains material from ”Clobber-NVM: Log Less, Re-execute More”, by Yi Xu, Joseph Izraelevitz, and Steven Swanson, which appeared in the Proceedings of International

Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2021. The dissertation author is the primary investigator and first author of this paper.

This chapter contains material from ”Zhuque: Failure is Not an Option, it’s an Exception”, by George Hodgkins*, Yi Xu* (*co-first author), Steven Swanson, and Joseph Izraelevitz, which appeared in 2023 USENIX Annual Technical Conference (USENIX ATC 23). The dissertation author is the primary investigator and co-first author of this paper.

This chapter contains material from ”Persistron: Leveraging Persistent Memory in Key-value Stores via a Crash-safe and Durable Cache”, by Yi Xu, Ramnatthan Alagappan, Aishwarya Ganesan, Rob Johnson, and Alex Conway, which was submitted for publication at 2024 International Conference on Management of Data. The dissertation author is the primary investigator and first author of this paper.

Bibliography

[1] eadr characteristics at failure. https://groups.google.com/g/pmem/c/K35X70fzAMw/m/5qEhhzb8AAAJ, 2021.

[2] Alper Ilkbahar. Intel Optane DC Persistent Memory Operating Modes Explained, 2018.

[3] Mohammad Alshboul, Hussein Elnawawy, Reem Elkhouly, Keiji Kimura, James Tuck, and Yan Solihin. Efficient checkpointing with recompute scheme for non-volatile main memory. *ACM Trans. Archit. Code Optim.*, 16(2), May 2019.

[4] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *SIGMETRICS*, 2012.

[5] Katelin A Bailey, Peter Hornyack, Luis Ceze, Steven D Gribble, and Henry M Levy. Exploring storage class memory with key value stores. In *Proceedings of the 1st Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads*, pages 1–8, 2013.

[6] Dhruba Borthakur. Rocksdb github wiki – performance benchmarks, 2013.

[7] Miao Cai, Chance C Coats, and Jian Huang. Hoop: Efficient hardware-assisted out-of-place update for non-volatile memory. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 584–596. IEEE, 2020.

[8] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H.C. Du. Characterizing, modeling, and benchmarking RocksDB Key-Value workloads at facebook. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 209–223, Santa Clara, CA, February 2020. USENIX Association.

[9] Calin Cascaval, Colin Blundell, Maged Michael, Harold W. Cain, Peng Wu, Stefanie Chiras, and Siddhartha Chatterjee. Software transactional memory: Why is it only a research toy? *Commun. ACM*, 51(11):40–46, November 2008.

[10] Daniel Castro, Paolo Romano, and João Barreto. Hardware transactional memory meets memory persistency. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 368–377, 2018.

[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, page 433–452, New York, NY, USA, 2014. Association for Computing Machinery.

[12] 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. ACM, 2014.

[13] Hao Chen, Chaoyi Ruan, Cheng Li, Xiaosong Ma, and Yinlong Xu. Spandb: A fast, cost-effective lsm-tree based {KV} store on hybrid storage. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 17–32, 2021.

[14] Shimin Chen and Qin Jin. Persistent b¡sup¿+¡/sup¿-trees in non-volatile main memory. *Proc. VLDB Endow.*, 8(7):786–797, February 2015.

[15] Sui Chen, Faen Zhang, Lei Liu, and Lu Peng. Efficient gpu nvram persistence with helper warps. In *2019 56th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2019.

[16] Youmin Chen, Youyou Lu, Fan Yang, Qing Wang, Yang Wang, and Jiwu Shu. Flatstore: An efficient log-structured key-value storage engine for persistent memory. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1077–1091, 2020.

[17] Chi Cao Minh, JaeWoong Chung, C. Kozyrakis, and K. Olukotun. Stamp: Stanford transactional applications for multi-processing. In *2008 IEEE International Symposium on Workload Characterization*, pages 35–46, 2008.

[18] Dave Chinner. xfs: updates for 4.2-rc1, 2015. http://oss.sgi.com/archives/xfs/2015-06/msg00478.html.

[19] 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 XVI, page 105–118, New York, NY, USA, 2011. Association for Computing Machinery.

[20] 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.

[21] Nachshon Cohen, David T. Aksun, Hillel Avni, and James R. Larus. Fine-grain checkpointing with in-cache-line logging. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 441–454, New York, NY, USA, 2019. Association for Computing Machinery.

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

[23] Alexander Conway, Abhishek Gupta, Vijay Chidambaram, Martin Farach-Colton, Richard Spillane, Amy Tai, and Rob Johnson. Splinterdb: Closing the bandwidth gap for nvme key-value stores. In *2020 {USENIX} Annual Technical Conference ({USENIX}{ATC} 20)*, pages 49–63, 2020.

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

[25] George Copeland, Michael Franklin, and Gerhard Weikum. Uniform object management. In *International Conference on Extending Database Technology*, pages 253–268. Springer, 1990.

[26] Intel Corporation. Asynchronous event handling. In *CXL Type 3 Memory Device Software Guide*, page 65. June 2021. Revision 1.0.

[27] Intel Corporation. Gpf sequence. In *CXL Type 3 Memory Device Software Guide*, page 121. June 2021. Revision 1.0.

[28] Intel Corporation. Microcode update facilities: Update signature and verification. In *Intel 64 and IA-32 Architectures Software Developer's Manual*, volume 3, chapter 10.11, pages 10–36–10–37. March 2023. Order No. 325462-079US.

[29] Intel Corporation. System management mode: Smram. In *Intel 64 and IA-32 Architectures Software Developer's Manual*, volume 3, chapter 32.4, pages 32–4–32–9. March 2023. Order No. 325462-079US.

[30] Andreia Correia, Pascal Felber, and Pedro Ramalhete. Romulus: Efficient algorithms for persistent transactional memory. In *Proceedings of the 30th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '18, page 271–282, New York, NY, USA, 2018. Association for Computing Machinery.

[31] Mahesh Dananjaya, Vasilis Gavrielatos, Arpit Joshi, and Vijay Nagarajan. Lazy release persistency. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 1173–1186, New York, NY, USA, 2020. Association for Computing Machinery.

[32] Marc A. de Kruijf, Karthikeyan Sankaralingam, and Somesh Jha. Static analysis and compiler design for idempotent processing. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, page 475–486, New York, NY, USA, 2012. Association for Computing Machinery.

[33] Dave Dice, Alex Kogan, Yossi Lev, Timothy Merrifield, and Mark Moir. Adaptive integration of hardware and software lock elision techniques. In *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '14, pages 188–197. Association for Computing Machinery, 2014.

[34] David Dice, Virendra J. Marathe, and Nir Shavit. Lock cohorting: A general technique for designing numa locks. *ACM Trans. Parallel Comput.*, 1(2), February 2015.

[35] Chen Ding, Jiguang Wan, and Rui Yan. Hybridkv: An efficient key-value store with hybridtree index structure based on non-volatile memory. In *Journal of Physics: Conference Series*, volume 2025, page 012093. IOP Publishing, 2021.

[36] Python Software Foundation. The python performacne benchmark suite, 2021.

[37] The LLVM Foundation. Clang: A C Language Family Frontend for LLVM, 2019.

[38] The LLVM Foundation. LLVM: Basic Alias Analysis, 2019.

[39] The LLVM Foundation. LLVM Language Reference Manual, 2019.

[40] The LLVM Foundation. LLVM's Analysis and Transform Passes, 2019.

[41] The LLVM Foundation. The LLVM Compiler Infrastructure, 2019.

[42] Michal Friedman, Maurice Herlihy, Virendra Marathe, and Erez Petrank. A persistent lock-free queue for non-volatile memory. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '18, page 28–40, New York, NY, USA, 2018. Association for Computing Machinery.

[43] Ellis R Giles, Kshitij Doshi, and Peter Varman. Softwrap: A lightweight framework for transactional support of storage class memory. In *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–14. IEEE, 2015.

[44] Vaibhav Gogte, William Wang, Stephan Diestelhorst, Peter M Chen, Satish Narayanasamy, and Thomas F Wenisch. Relaxed persist ordering using strand persistency. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 652–665. IEEE, 2020.

[45] Vaibhav Gogte, William Wang, Stephan Diestelhorst, Peter M Chen, Satish Narayanasamy, and Thomas F Wenisch. Relaxed persist ordering using strand persistency. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 652–665. IEEE, 2020.

[46] Jim Gray and Andreas Reuter. *Transaction processing: concepts and techniques*. Elsevier, 1992.

[47] Jinyu Gu, Qianqian Yu, Xiayang Wang, Zhaoguo Wang, Binyu Zang, Haibing Guan, and Haibo Chen. Pisces: A scalable and efficient persistent transactional memory. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '19, page 913–928, USA, 2019. USENIX Association.

[48] Jinyu Gu, Qianqian Yu, Xiayang Wang, Zhaoguo Wang, Binyu Zang, Haibing Guan, and Haibo Chen. Pisces: A scalable and efficient persistent transactional memory. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '19, pages 913–928. USENIX Association, 2019.

[49] Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM computing surveys (CSUR)*, 15(4):287–317, 1983.

[50] Swapnil Haria, Mark D. Hill, and Michael M. Swift. Mod: Minimally ordered durable datastructures for persistent memory. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 775–788, New York, NY, USA, 2020. Association for Computing Machinery.

[51] George Hodgkins, Yi Xu, Steven Swanson, and Joseph Izraelevitz. Zhuque: Failure is not an option, it's an exception. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 833–849. USENIX Association, 2023.

[52] 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*, pages 429–442, 2021.

[53] HPE Superdome Flex 280 Server, 2021. https://buy.hpe.com/us/en/servers/mission-critical-x86-servers/superdome-flex-servers/superdome-flex-server/hpe-superdome-flex-280-server/p/1012865453.

[54] Terry Ching-Hsiang Hsu, Helge Brügner, Indrajit Roy, Kimberly Keeton, and Patrick Eugster. Nvthreads: Practical persistence for multi-threaded applications. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, page 468–482, New York, NY, USA, 2017. Association for Computing Machinery.

[55] Yihe Huang, Matej Pavlovic, Virendra Marathe, Margo Seltzer, Tim Harris, and Steve Byan. Closing the performance gap between volatile and persistent key-value stores using cross-referencing logs. In *2018 USENIX Annual Technical Conference (USENIX ATC*

18), pages 967–979, 2018.

[56] Intel. Intel optane dc persistent memory. https://newsroom.intel.com/news-releases/intel-data-centric-launch/#gs.gylgr6, 2019.

[57] Intel Corporation. Pmdk issues: introduce hybrid transactions, 2017.

[58] Intel Corporation. Examples for libpmemobj: A Transactional HashMap, 2019.

[59] Intel Corporation. Intel Optane DC Persistent Memory, 2019.

[60] Intel Corporation. eADR: New Opportunities for Persistent Memory Applications, 2021.

[61] 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.

[62] Joseph Izraelevitz, Lingxiang Xiang, and Michael L Scott. Performance improvement via always-abort htm. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 79–90. IEEE, 2017.

[63] Jungi Jeong and Changhee Jung. Pmem-spec: persistent memory speculation (strict persistency can trump relaxed persistency). In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 517–529, 2021.

[64] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. Splitfs: Reducing software overhead in file systems for persistent memory. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 494–508, 2019.

[65] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. Splitfs: Reducing software overhead in file systems for persistent memory. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 494–508, 2019.

[66] Olzhas Kaiyrakhmet, Songyi Lee, Beomseok Nam, Sam H. Noh, and Young ri Choi. Slm-db: Single-level key-value store with persistent memory. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 191–205, Boston, MA, 2019. USENIX Association.

[67] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Redesigning lsms for nonvolatile memory with novelsm. In *Proceedings of the 2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 993–1005, 2018.

[68] Hiwot Tadese Kassa, Jason Akers, Mrinmoy Ghosh, Zhichao Cao, Vaibhav Gogte, and Ronald Dreslinski. Improving performance of flash based key-value stores using storage class memory as a volatile memory extension. In *Proceedings of the 2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 821–837, 2021.

[69] Aasheesh Kolli, Steven Pelley, Ali Saidi, Peter M. Chen, and Thomas F. Wenisch. High-performance transactions for persistent memories. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, page 399–411, New York, NY, USA, 2016. Association for Computing Machinery.

[70] Aasheesh Kolli, Jeff Rosen, Stephan Diestelhorst, Ali Saidi, Steven Pelley, Sihang Liu,

Peter M. Chen, and Thomas F. Wenisch. Delegated persist ordering. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-49. IEEE Press, 2016.

[71] Kunal Korgaonkar, Joseph Izraelevitz, Jishen Zhao, and Steven Swanson. Vorpal: Vector clock ordering for large persistent memory systems. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, PODC '19, page 435–444. Association for Computing Machinery, 2019.

[72] R. Madhava Krishnan, Jaeho Kim, Ajit Mathew, Xinwei Fu, Anthony Demeri, Changwoo Min, and Sudarsun Kannan. Durable transactional memory can scale with timestone. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 335–349, New York, NY, USA, 2020. Association for Computing Machinery.

[73] Eta Labs. libc-bench, 2021.

[74] Charles R Landau. The checkpoint mechanism in keykos. In *[1992] Proceedings of the Second International Workshop on Object Orientation in Operating Systems*, pages 86–91. IEEE, 1992.

[75] Robert Lasch, Thomas Legler, Norman May, Bernhard Scheirle, and Kai-Uwe Sattler. Cost modelling for optimal data placement in heterogeneous main memory. *Proceedings of the VLDB Endowment*, 15(11):2867–2880, 2022.

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

[77] Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. Recipe: Converting concurrent dram indexes to persistent-memory indexes. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 462–477, 2019.

[78] Wenjie Li, Dejun Jiang, Jin Xiong, and Yungang Bao. Hilsm: An lsm-based key-value store for hybrid nvm-ssd storage systems. In *Proceedings of the 17th ACM International Conference on Computing Frontiers*, pages 208–216, 2020.

[79] libMemcached.org. libMemcached, 2011.

[80] Zhan Lin, Lu Kai, Zhilong Cheng, and Jiguang Wan. Rangekv: An efficient key-value store based on hybrid dram-nvm-ssd storage structure. *IEEE Access*, 8:154518–154529, 2020.

[81] Zhen Lin, Mohammad Alshboul, Yan Solihin, and Huiyang Zhou. Exploring memory persistency models for gpus. In *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 311–323. IEEE, 2019.

[82] Linux Kernel Organization. Direct Access for Files, 2020.

[83] Mengxing Liu, Mingxing Zhang, Kang Chen, Xuehai Qian, Yongwei Wu, Weimin Zheng, and Jinglei Ren. Dudetm: Building durable transactions with decoupling for persistent memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, page 329–343, New York, NY, USA, 2017. Association for Computing Machinery.

[84] Mengxing Liu, Mingxing Zhang, Kang Chen, Xuehai Qian, Yongwei Wu, Weimin Zheng,

and Jinglei Ren. Dudetm: Building durable transactions with decoupling for persistent memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 329–343. Association for Computing Machinery, 2017.

[85] 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.

[86] Amirsaman Memaripour, Anirudh Badam, Amar Phanishayee, Yanqi Zhou, Ramnatthan Alagappan, Karin Strauss, and Steven Swanson. Atomic in-place updates for non-volatile main memories with kamino-tx. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, page 499–512, New York, NY, USA, 2017. Association for Computing Machinery.

[87] Memcached. http://memcached.org/.

[88] Transactional memory study group (SG5). Technical specification for c++ extensions for transactional memory iso/iec ts 19841:2015, 2015.

[89] musl libc, 2021. https://musl.libc.org/.

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

[91] Sanketh Nalli, Swapnil Haria, Mark D. Hill, Michael M. Swift, Haris Volos, and Kimberly Keeton. An analysis of persistent memory use with whisper. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, page 135–148, New York, NY, USA, 2017. Association for Computing Machinery.

[92] Dushyanth Narayanan and Orion Hodson. Whole-system persistence with non-volatile memories. In *Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2012)*. ACM, March 2012.

[93] Iyswarya Narayanan, Aishwarya Ganesan, Anirudh Badam, Sriram Govindan, Bikash Sharma, and Anand Sivasubramaniam. Getting more performance with polymorphism from emerging memory technologies. In *12th ACM International Conference on Systems and Storage (SYSTOR 19)*, Haifa, Israel, April 2019. ACM.

[94] Faisal Nawab, Joseph Izraelevitz, Terence Kelly, Charles B Morrey III, Dhruva R Chakrabarti, and Michael L Scott. Dalí: A periodically persistent hash map. In *31st International Symposium on Distributed Computing (DISC 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[95] Edmund B Nightingale, Kaushik Veeraraghavan, Peter M Chen, and Jason Flinn. Rethink the sync. *ACM Transactions on Computer Systems (TOCS)*, 26(3):1–26, 2008.

[96] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. Fptree: A hybrid scm-dram persistent and concurrent b-tree for storage class memory. In *Proceedings of the 2016 International Conference on Management of Data*, pages 371–386, 2016.

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

[98] Yoav Raz. The principle of commitment ordering, or guaranteeing serializability in a heterogeneous environment of multiple autonomous resource mangers using atomic commitment. In *Proceedings of the 18th International Conference on Very Large Data Bases*, VLDB '92, page 292–312, San Francisco, CA, USA, 1992. Morgan Kaufmann Publishers Inc.

[99] Wenjia Ruan, Trilok Vyas, Yujie Liu, and Michael Spear. Transactionalizing legacy code: An experience report using gcc and memcached. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 399–412. Association for Computing Machinery, 2014.

[100] Andy Rudoff, Chet Douglas, and Tiffany Kasanicky. Persistent memory in cxl. In *Proceedings of the 2021 SNIA Persistent Memory + Computational Storage Summit*, April 2021.

[101] J. Ruppert. A delaunay refinement algorithm for quality 2-dimensional mesh generation. *Journal of Algorithms*, 1995.

[102] Steve Scargall. *Concurrency and Persistent Memory*, pages 277–294. Apress, Berkeley, CA, 2020.

[103] Steve Scargall. *PMDK Internals: Important Algorithms and Data Structures*, pages 313–331. Apress, Berkeley, CA, 2020.

[104] Jonathan S Shapiro and Jonathan Adams. Design evolution of the eros single-level store. In *USENIX Annual Technical Conference, General Track*, pages 59–72, 2002.

[105] Jonathan S Shapiro, Jonathan M Smith, and David J Farber. Eros: a fast capability system. In *Proceedings of the seventeenth ACM symposium on Operating systems principles*, pages 170–185, 1999.

[106] Eugene Shekita and Michael Zwilling. Cricket: A mapped, persistent object store. Technical report, University of Wisconsin-Madison Department of Computer Sciences, 1990.

[107] Seunghee Shin, James Tuck, and Yan Solihin. Hiding the long latency of persist barriers using speculative execution. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, page 175–186, New York, NY, USA, 2017. Association for Computing Machinery.

[108] Frank G Soltis. *Fortress Rochester: The Inside Story of the IBM iSeries*. System iNetwork, 2001.

[109] Emil Tsalapatis, Ryan Hancock, Tavian Barnes, and Ali José Mashtizadeh. The aurora operating system: revisiting the single level store. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 136–143, 2021.

[110] Alexander van Renen, Viktor Leis, Alfons Kemper, Thomas Neumann, Takushi Hashida, Kazuichi Oe, Yoshiyasu Doi, Lilian Harada, and Mitsuru Sato. Managing non-volatile memory in database systems. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1541–1555, 2018.

[111] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy H. Campbell. Consistent and durable data structures for non-volatile byte-addressable memory. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies*, FAST'11, page 5, USA, 2011. USENIX Association.

[112] 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.

[113] Haris Volos, Andres Jaan Tack, and Michael M. Swift. Mnemosyne: Lightweight persistent memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, page 91–104, New York, NY, USA, 2011. Association for Computing Machinery.

[114] Gerhard Weikum and Gottfried Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[115] Matthew Wilcox. Add support for NV-DIMMs to ext4, 2014. https://lwn.net/Articles/613384/.

[116] Xiaojian Wu and A. L. Narasimha Reddy. SCMFS: A file system for storage class memory. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 39:1–39:11, New York, NY, USA, 2011. ACM.

[117] Zhenwei Wu, Kai Lu, Andrew Nisbet, Wenzhe Zhang, and Mikel Luján. Pmthreads: Persistent memory threads harnessing versioned shadow copies. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 623–637, New York, NY, USA, 2020. Association for Computing Machinery.

[118] Fei Xia, Dejun Jiang, Jin Xiong, and Ninghui Sun. Hikv: A hybrid index key-value store for dram-nvm memory systems. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 349–362, 2017.

[119] 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.

[120] 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, 2016.

[121] 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, 2016.

[122] Yi Xu, Ramnatthan Alagappan, Aishwarya Ganesan, Rob Johnson, and Alex Conway. Persistron: Leveraging persistent memory in key-value stores via a crash-safe and durable cache. In *Submitted to Proceedings of the 2024 International Conference on Management of Data*, 2024.

[123] Yi Xu, Joseph Izraelevitz, and Steven Swanson. Clobber-nvm: Log less, re-execute more. In *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021.

[124] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. Nv-tree: Reducing consistency cost for nvm-based single level systems. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, FAST'15,

page 167–181, USA, 2015. USENIX Association.

[125] Ting Yao, Yiwen Zhang, Jiguang Wan, Qiu Cui, Liu Tang, Hong Jiang, Changsheng Xie, and Xubin He. Matrixkv: Reducing write stalls and write amplification in lsm-tree based kv stores with matrix container in nvm. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 17–31, 2020.

[126] Richard M. Yoo, Yang Ni, Adam Welc, Bratin Saha, Ali-Reza Adl-Tabatabai, and Hsien-Hsin S. Lee. Kicking the tires of software transactional memory: Why the going gets tough. In *Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '08, pages 265–274. Association for Computing Machinery, 2008.

[127] Lu Zhang and Steven Swanson. Pangolin: A fault-tolerant persistent memory programming library. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '19, page 897–911, USA, 2019. USENIX Association.

[128] Wenhui Zhang, Xingsheng Zhao, Song Jiang, and Hong Jiang. Chameleondb: a key-value store for optane persistent memory. In *Proceedings of the Sixteenth European Conference on Computer Systems*, pages 194–209, 2021.

[129] 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.

[130] Xinjing Zhou, Joy Arulraj, Andrew Pavlo, and David Cohen. Spitfire: A three-tier buffer manager for volatile and non-volatile memory. In *Proceedings of the 2021 International Conference on Management of Data*, pages 2195–2207, 2021.