

**ARCHITECTURE AND SOFTWARE SUPPORT FOR
PERSISTENT AND VAST MEMORY**

by

Swapnil Haria

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2019

Date of final oral examination: 6/10/19

The dissertation is approved by the following members of the Final Oral Committee:

Mark D. Hill, Professor, Computer Sciences

Michael M. Swift, Professor, Computer Sciences

Andrea Arpaci-Dusseau, Professor, Computer Sciences

Kimberly Keeton, Distinguished Technologist, Hewlett Packard Labs

Mikko Lipasti, Professor, Electrical and Computer Engineering

Dedicated to my family who have always sacrificed so that I may succeed.

Acknowledgments

And what would humans be
without love?
"RARE", said Death.

TERRY PRATCHETT

I have been supported by many people to get to and through graduate school.

First, I am grateful to my advisors, Mark Hill and Mike Swift. Mark Hill taught me about asking the right questions, identifying the essence of a problem, communicating ideas clearly and being organized in work and in life. Moreover, he has always given me great advice for every situation. Mike Swift taught me to love implementation-level details, how to evaluate ideas thoroughly and to effectively critique research literature. Their complementary strengths in domains and skills resulted in a great team and provided me with two great role models for my career.

I thank my committee members for improving my research with their unique perspectives. Kimberly Keeton inspired many design decisions in our persistent memory hardware with her insightful questions. Our hardware primitives for persistent memory were directly inspired by Andrea Arpaci-Dusseau's research in filesystems. Mikko Lipasti taught me a lot about computer architecture when I was a young graduate student and I have often been inspired by his encyclopedic knowledge of architecture history.

I would like to thank my colleagues and friends who made me survive grad school. I

shared a common office with Jason Lowe-Power and Nilay Vaish and common grievances with Lena Olson. I learned a lot from all three of them about how to do research in computer architecture. I am also grateful to Jayneel Gandhi, Mark Mansi, Pratyush Mahapatra and others in the Multifacet group as well as Hongil Yoon. Finally, I enjoyed many discussions with the architecture reading and lunch groups: Gokul Ravi, Vinay Gangadhar, Vijay Thiruvengadam, Newsha Ardalani, Tony Nowatzki, David Schlais, Kyle Daruwalla, Carly Schwartz, Ravi Raju, Shyam Murthy, Suchita Pati and many others.

I thank Akshay Sood for many adventures around Madison, Arpit Agarwal and Sachin Muley for many refreshing lunches and dinners, and Yash Agarwal, Neha Govind, Ritika Oswal, Akhil Guliani, Shaleen Deep, Amrita Roy Chowdhury, Samartha Patel and many others for many fun events that kept me sane. During my early days in grad school, Urmish Thakker, Lokesh Jindal, Roney Michael, Mihir Patil and many others helped me decide to stay on for a PhD and believed in me even when I did not.

I am forever indebted to RaviGroup and PJGroup in NVIDIA Bangalore for sparking my interest in computer architecture. In particular, I thank Ravikrishnan Sree, Jayakumar Parasuraman, Sumit Shrivastava, Ajay Ganesh, Srikanth Muralidharan, Naveen Yagnamurthy and Akshay B. for patiently answering questions and teaching me a lot about computer processors.

I am also grateful to the Berta Armacanqui and her family who have provided me with a familial atmosphere and treated me and my wife as one of their own. I also thank Dawn and Randy Dorning and their family who have helped me and countless other international students feel at home in Madison.

Most importantly, I thank my family who have always supported me and deserve all the credit for my success. My partner, Sukriti Singh, is most responsible for my survival in graduate school. She has always offered a convenient escape from the rigors of work. My sister, Apexa Haria, has done more than her fair share to help me out and protect me over the years. My parents, Madhuri and Dinesh Haria, are responsible for always nurturing my

curiosity and teaching me the values I needed for success. Without my family's sacrifices over the years, I would not have been in the position I am today.

Contents

Abstract	xiii
1 Introduction	1
1.1 <i>Challenges of PM technologies</i>	3
1.2 <i>Contributions</i>	4
1.3 <i>Thesis Organization</i>	7
2 Persistent Memory Background	8
2.1 <i>System Model</i>	8
2.2 <i>Hardware Primitives for Ordering and Durability</i>	9
2.3 <i>Programming Recoverable Applications</i>	10
3 Hands-Off Persistence System	16
3.1 <i>Insights from Workload Analysis</i>	18
3.2 <i>New Hardware Primitives</i>	19
3.3 <i>Memory Persistency Model</i>	23
3.4 <i>HOPS Design</i>	28
3.5 <i>Evaluation</i>	35
3.6 <i>Comparing Related Work with HOPS</i>	38
3.7 <i>Conclusion</i>	40

4	Minimally Ordered Durable Datastructures for Persistent Memory	41
4.1	<i>Background on Functional Programming</i>	44
4.2	<i>Ordering & Flushing Overheads on Optane DCPMMs</i>	46
4.3	<i>Minimally Ordered Durable Datastructures</i>	49
4.4	<i>Implementation Details</i>	60
4.5	<i>Extensions for Concurrency</i>	62
4.6	<i>Evaluation</i>	63
4.7	<i>Comparing Related Work with MOD</i>	70
4.8	<i>Conclusion</i>	73
5	Devirtualized Memory for Heterogeneous Systems	74
5.1	<i>Chapter Background</i>	78
5.2	<i>Devirtualizing Memory</i>	79
5.3	<i>Implementing DVM for Accelerators</i>	82
5.4	<i>Discussion</i>	90
5.5	<i>Evaluation</i>	93
5.6	<i>Towards DVM across Heterogeneous Systems</i>	99
5.7	<i>Related Work in VM for Accelerators and Vast Memory</i>	102
5.8	<i>Conclusion</i>	104
6	Conclusions and Future Work	105
	Bibliography	110

List of Tables

2.1	Comparison of x86-64 primitives for PM.	10
3.1	Comparison of HOPS and x86-64 primitives for PM.	22
3.2	Ordering rules in TSO-BEP.	28
3.3	Handling of major events in HOPS.	34
3.4	Configuration of Simulated System.	36
4.1	Test Machine Configuration.	64
4.2	Benchmarks developed and used for this study. Each workload performs 1 million iterations of the operations described. We consider workloads marked with asterisk* to be write-intensive.	66
4.3	Number of update operations needed to double memory usage of datastructures with 1M entries with memory reclamation disabled. For PMDK map, set and vector, update operations happen in place without allocating memory.	70
5.1	Page Table Sizes for PageRank (first four rows) and Collaborative Filtering (last three rows) for different input graphs. PEs reduce the page table size by eliminating most L1PTEs.	86
5.2	Simulation Configuration Details	94
5.3	Graph Datasets Used for Evaluation	95
5.4	Percentage of total system memory successfully allocated with identity mapping.	99

5.5 Lines of code changed in Linux v4.10 split up by functionality. *Changes for memory-mapped segments affect heap segment, so we only count them once. . 100

List of Figures

2.1	Target System with both DRAM and PM. Structures colored orange are in the volatile domain, green in the persistent domain and yellow in the persistent domain only in presence of certain optimizations.	9
2.2	Execution Timeline for weakly ordered flushes to PM on x86-64 systems.	11
2.3	Different PM programming approaches illustrated using a sample program which updates a persistent struct pt and then set a persistent flag: (a) Low-Level Epochs, (b) High-level failure-atomic code sections, (c) Software Transactional Memory for PM.	13
2.4	Example of STM implementation for PM with undo-logging.	14
3.1	Guarantees provided by (a) ofence, (b) dfence and (c) x86-64 primitives. In (c), if the c1wb is removed, there will not be any happens-before relationship.	20
3.2	PM-STM implementation with (a) x86-64 primitives and (b) HOPS primitives.	21
3.3	Comparing HOPS and x86-64 primitives. Sample program written with (a) epochs, (b) x86-64 primitives and (c) HOPS primitives. Programmer intends for all PM stores to be durable at the end of this program.	22
3.4	Example of necessary ordering (PMO) of stores to different addresses from different threads. Using two flags, programmer desires S4 to be ordered after S1 in PMO.	26

3.5	Example of <i>reads-from</i> (rf) and <i>epoch-happens-before</i> (ehb) relations.	27
3.6	Simple example showing the use of timestamps and dependency tuples for indicating PMO happens-before relationships in HOPS. Stores shown inside a cloud are in the same epoch and thus unordered with respect to each other. . .	30
3.7	HOPS System Design, with hardware modifications marked in blue.	31
3.8	Example execution showing ofence and dfence implementation in HOPS. Persist Buffer entries in green are not yet flushed and in yellow have been flushed but flush ACKs are pending.	33
3.9	Performance of HOPS relative to x86-64 primitives, and an ideal but non crash-consistent implementation.	37
4.1	PM-STM overheads in recoverable PM workloads implemented using PMDK v1.5.	43
4.2	Implementing prepend operation for linked list (defined in (a)) as (b) impure function where original list L is modified and (c) pure function where a new updated list shadowL is created and returned; application uses the returned list as the latest updated list. (d) shadowL reuses nodes of original list L to reduce space overheads.	45
4.3	Average Latency of a PM cacheline flush on test machine with Optane DCPMM and compared to our analytical model based on Amdahl's law.	47
4.4	Functional Shadowing in action on (a) MOD vector. (b) Shadow is created on Append (i.e., push_back) operation that reuses data from the original vector. (c) Application starts using updated shadow, while old and unused data is cleaned up.	50
4.5	Failure-Atomic Code Sections (FASEs) with MOD datastructures using (a) single-versioned interface to update one datastructure and (b) multi-versioned interface to atomically update multiple datastructures, dsPtr1 and dsPtr2. Datastructures in red are temporary shadow datastructures.	54
4.6	Failure-atomic append (i.e., push_back) on single-versioned MOD vector.	54

4.7	Using multi-versioned MOD datastructures for failure-atomically (a) appending an element to a vector, (a) swapping two elements of a vector and (c) swapping two elements of two different vectors.	57
4.8	Implementation of single-versioned interface as a wrapper around the multi-versioned interface.	57
4.9	Commit implementation shown for multi-update FASEs operating on (a) single datastructure, (b) multiple datastructures pointed to by common <i>parent</i> object, and (c) (uncommon) multiple unrelated datastructures.	60
4.10	Execution Time of PM workloads, normalized to PMDK v1.4 implementation of each workload. Queue-pop* is queue-pop with the first pop operation being untimed.	65
4.11	Flushing and ordering behavior of PM workloads. queue-pop (not shown) has 1 fence and 104 flushes per operation due to pathological case described earlier.	67
4.12	L1D Cache miss ratios for our workloads.	68
5.1	Heterogeneous systems with (a) conventional VM with translation on critical path and (b) DVM with Devirtualized Access Validation alongside direct access on reads.	75
5.2	TLB miss rates for Graph Workloads with 128-entry TLB	79
5.3	Address Space with Identity Mapped and Demand Paged Allocations.	80
5.4	Memory Accesses in DVM	81
5.5	4-level Address Translation in x86-64	84
5.6	Structure of a Permission Entry. PE: Permission Entry, P15-P0: Permissions. . .	86
5.7	Pseudocode for Identity Mapping	88
5.8	Execution time for accelerator workloads, normalized to runtime of ideal implementation.	95
5.9	Dynamic energy spent in address translation/access validation, normalized to the 4KB, TLB+PWC implementation.	97

5.10 Runtime of CPU-only workloads, normalized to the ideal case. 102

Abstract

Emerging non-volatile memory technologies promise new opportunities by offering data persistence and vast memory capacity but also face substantial challenges. Data persistence enables application data to survive both planned and unplanned power outages. Unfortunately, applications must use fine-grained cacheline flush instructions to explicitly move data from volatile caches to persistent memory (PM) and rely on expensive ordering instructions to enable consistent recovery. Meanwhile, these new technologies also allow applications to store large amounts of data in memory. However, virtual memory (VM) techniques, which were designed for small, megabyte-sized memories, cause significant overheads with terabyte-scale memories.

In this dissertation, we first propose new hardware primitives to improve the performance and programmability of applications that leverage data persistence. Recently added x86-64 primitives require programmers have to track modified cachelines and individually flush them to PM. Moreover, ordering primitives that are frequently used by applications cause expensive serialization of long latency cacheline flushes. We propose new hardware, the *Hands-Off Persistence System* (HOPS), to implement a lightweight ordering fence and a separate durability fence. HOPS introduces new hardware to track stores to PM, thereby automating and lowering the cost of data movement from volatile caches to persistent memory. HOPS improves application performance by 24% on average over x86-64 systems in simulation.

Second, to improve application performance on unmodified hardware, we minimize the number of expensive ordering operations in PM applications via *Minimally Ordered Durable* (MOD) datastructures, a software-only proposal. Currently, PM applications rely on software transactional memory implementations for PM (PM-STM) to ensure atomicity of updates across power failures. On actual Intel Optane DC Persistent Memory Modules, we show that flushing and frequent ordering points in these PM-STM implementations cause overheads of up to $11\times$ over an un-recoverable baseline without flushing or ordering points. By leveraging existing implementations of immutable datastructures from functional languages, we create MOD datastructures that offer failure-atomic updates with only a single ordering operation in the common case. We develop a C++ library of MOD datastructures with vector, map, set, stack and queue implementations, and show that these datastructures improve application performance by 60% on average compared to state-of-the-art PM-STM implementations.

Finally, we address the problem of rising virtual memory overheads due to vast memories. These overheads are most acute in compute units that cannot justify large and power-hungry address translation hardware, such as emerging accelerators. However, programmers still rely on VM for memory protection and to simplify programming. To minimize VM overheads, we propose *Devirtualized Memory* (DVM), which eliminates expensive address translation on most memory accesses. We achieve this by allocating most memory so that its virtual address matches its physical address. When implemented on a graph-processing accelerator, DVM reduces VM overheads to less than 4% compared to an unsafe baseline without VM support. We also discuss how DVM can be extended for use with general-purpose CPUs.

— 1 —

Introduction

Everything starts somewhere,
though many physicists disagree.

TERRY PRATCHETT

Non-Volatile Memory (NVM) is here—NVM systems are already offered in a limited manner by Google Cloud [46] and have been announced to ship by late 2019 [66, 67]. Many different NVM technologies are being developed in industry for both general-purpose systems and specialized domains like Internet of Things (IoT) devices. These technologies include 3D XPoint™ [59], Phase Change Memory [144], Resistive RAM [25, 71], Spin-transfer Torque Magnetic RAM [57], Conductive-Bridge RAM [45] and (perhaps) Memristors [129, 140]. While these technologies differ in their physical characteristics, we can collectively abstract them as fast, high-capacity, byte-addressable and non-volatile main memory devices. When such devices are attached on the memory bus and can be accessed by applications using regular load/store instructions, we refer to them as Persistent Memory (PM).

The slowdown in capacity scaling of conventional DRAM technology [68] has paved the way for rapid development and widespread adoption of PM. While DRAM has been commercially available since 1970, 64 gigabyte (GB) DRAM devices were released for the first time in 2018 [97]. In contrast, the first generation of Intel Optane DC Persistent Memory Module (DCPMM) includes 128, 256 and 512 GB devices [67]. These memory capacities

enable systems with vast memory, i.e., memory greater than 1 terabyte (TB) per socket. Notably, all experiments in Chapter 4 of this thesis were run on a system with 3 TB of PM comprising engineering samples of 256 GB Optane DCPMMs. Furthermore, both DRAM and Optane DCPMMs are comparable in terms of cost at \$7.07 per GB [17] and \$6.57 per GB [134] respectively.

By offering low-latency access to vast capacities of memory, PM is highly attractive for today’s data-centric workloads. Data is being generated at a very prodigious pace by modern applications, e.g., Twitter (62.5 GB/hour) [125], genomics (0.99 TB/hour) [128], YouTube (7.2 TB/hour) [98] and astronomy (27 PB/hour by one project alone, the Australian Square Kilometre Array Pathfinder [96]). Keeping such huge TB-sized datasets in memory is beneficial for important workloads such as data analytics and model training for machine learning.

While memory capacity is the initial selling point for PM devices, the property of data persistence promises to have a more disruptive impact on computer systems. PM enables applications to durably store in-memory state including pointer-based datastructures. Without PM, applications must serialize such datastructures to block devices like disks or solid-state drives for durability, which is both slow and complicated. Fortunately, application state stored in PM survives both planned and unplanned power outages, a desirable quality for workloads like databases, key-value stores and long-running scientific computations [18, 82].

The emergence of Persistent Memory signals a paradigm shift, uniting volatile memory and persistent storage for the first time since the dawn of computing. By providing much lower latencies and eliminating the need for data serialization, PM can replace disks for important usecases that require temporary but persistent data storage such as checkpointing and staging areas for streaming workflows [39]. Moreover, it enables new system abstractions like virtual address spaces existing independently of process lifetimes [38].

1.1 Challenges of PM technologies

Although persistent and vast memory devices offer new opportunities, they expose substantial limitations in existing systems that are designed for small or volatile memory. In this thesis, we identify and mitigate two major challenges faced by persistent and vast memory systems that degrade both performance and programmability of such systems. We present a brief overview of these challenges in this section and defer a detailed description to later chapters.

First, even as main memory becomes persistent, many essential hardware structures such as CPU registers, caches and various buffers are likely to remain volatile and not durable in the near future. As a result, application data may either be in the volatile domain (mainly caches) or persistent domain (mainly PM). However, application data in the volatile domain is lost on a power failure and is not accessible in future executions.

Hence, computer systems must now provide new hardware and software abstractions to handle data movement from the volatile to the persistent domain. Unfortunately, current architectural support for PM pushes this burden onto the programmer by requiring explicit data movement for durability and consistency. To facilitate PM programming, we need better abstractions that are both efficient and programmer-friendly.

Second, the overheads of conventional VM techniques scale with increasing memory capacities [6], especially as relevant hardware structures such as translation lookaside buffers (TLB) fail to scale proportionally. Both Intel and Linux have recently added support for 5-level paging [124, 65], which increases the upper bound on Physical Memory (PhysM¹) from 64 TB to 4 Petabytes. However, adding another level of page tables increases the address translation latency on a TLB miss.

Even though VM overheads threaten significant performance degradation, VM offers a convenient abstraction that improves the programmability of computer systems and offers memory protection. Given the high salaries commanded by programmers, programma-

¹In this thesis, we use the term PM to refer to Persistent Memory and PhysM for Physical Memory.

bility is a first-order concern in modern computers. Hence, it is important to design new techniques that preserve the useful features of VM, while reducing its overheads in systems with vast memory.

Thus, in this dissertation, we propose architectural and software techniques to improve the performance and programmability of systems with persistent and vast memory.

1.2 Contributions

In the first two parts of this dissertation, we develop one architectural and one software technique to tackle overheads in applications relying on memory persistence. In the third part, we address virtual memory overheads arising due to vast memory capacity.

Contribution 1: Improving Performance and Programmability of PM Applications

Currently, programmers seeking to exploit the durability of PM must use cacheline flush instructions to move data from volatile caches to durable PM. Moreover, they must order these flushes carefully to ensure consistency of application data across unplanned power outages. However, currently available ordering and durability primitives are too low-level for most programmers as they operate on data at a cacheline granularity. Additionally, we show that these primitives are inefficient and contribute to significant overheads in PM applications.

To support high-level and efficient primitives for ordering and durability, we propose the *Hands-Off Persistence System* (HOPS) in Chapter 3. We use insights from an analysis of realistic PM workloads to guide the design of HOPS. Our analysis shows that ordering events occur much more frequently than durability events in PM applications. Hence, HOPS decouples ordering from durability and offers a lightweight ordering fence and a separate durability fence. Using these primitives, programmers can indicate ordering and durability guarantees at different points in the program execution instead of on individual

cachelines of data. The HOPS hardware automatically tracks PM writes to enforce these guarantees at a cacheline granularity. HOPS improves the performance of PM applications by 24% on average over existing x86-64 PM primitives in simulation.

This chapter was published as Section 6 of "An Analysis of Persistent Memory Use with WHISPER" in *The Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2017* [94]; in this dissertation, we add a formal specification of the memory persistency model in HOPS and a longer exposition of the design and implementation.

Contribution 2: Reducing Ordering Constraints in PM Applications

In the near future, applications are more likely to use PM devices for higher capacities than for persistence. As such, it may be difficult to justify intrusive hardware modifications to improve the performance of a small set of recoverable applications. However, the high overheads of data persistence may deter programmers from creating new recoverable applications. To break this catch-22 situation, we must initially rely on software-only proposals to tolerate the high overheads found in such applications.

To improve the performance of PM applications on unmodified hardware, we introduce *minimally ordered durable* (MOD) datastructures in Chapter 4. We define MOD datastructures as those that enable *failure-atomic sections* (similar to transactions) with only a single ordering point in the common case. We present a simple recipe to create MOD datastructures from existing *purely functional* datastructures, allowing us to leverage significant research efforts from the domain of functional languages as opposed to handcrafting new recoverable datastructures. We develop a C++ library of MOD datastructures with vector, map, set, stack and queue implementations. This library improves application performance by 60% on average compared to state-of-the-art software transactional memory (STM) implementations for PM, on systems with Optane DCPMMs.

This chapter is currently under review for publication.

Comparing Contributions 1 and 2. HOPS and MOD are orthogonal approaches that seek to improve the performance of recoverable PM applications by reducing the overheads of ordering points. We observed two trends in PM applications running on x86-64 systems. First, ordering events are very common in PM applications to enable consistent recovery in case of crash. Second, ordering events are very expensive on x86-64 systems, as ordering is coupled with durability which requires long-latency cacheline flushes to PM. HOPS introduces a lightweight hardware primitive for ordering two PM writes (decoupled from a separate durability primitive) to lower the cost of frequent ordering events in PM applications. However, HOPS primitives are restricted to new hardware. In contrast, MOD lowers the frequency of expensive ordering events significantly to improve performance on current, unmodified x86-64 hardware. Unfortunately, being a software-only approach, MOD cannot address the programmability issues with x86-64 primitives, which is an added benefit in HOPS.

Contribution 3: Reducing Virtual Memory Overheads

While new NVM technologies have greatly increased memory capacity, this has been accompanied by a corresponding rise in virtual memory overheads. Server-class CPUs can partly mitigate these overheads using complicated, large and power-hungry TLB designs. Unfortunately, special-purpose accelerators that have become common in such systems cannot justify such expensive hardware structures. Moreover, TLBs are not scaling in proportion to increasing memory capacity [6] and may not be sufficient for CPUs in the future. Consequently, many accelerators either do not have access to virtual memory [99, 73] or have direct access to physical memory [91], choosing performance over programmability and safety.

To combine the protection and programmability of VM with the performance of direct access to PM, we propose De-Virtualized Memory (DVM) in Chapter 5. By allocating data such that its physical and virtual addresses are usually identical ($VA==PA$), DVM

eliminates expensive page-level address translation on most memory accesses. Moreover, DVM leverages the commonly occurring contiguity of page permissions to implement efficient memory protection. Our implementation of DVM requires modest OS and IOMMU changes, and is transparent to applications. It reduces VM overheads to less than 4% on average on a graph-processing accelerator with highly irregular memory access patterns.

This chapter was published as "Devirtualizing Memory in Heterogeneous Systems" in *The Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2018 [51]. The DVM evaluation uses vast volatile DRAM, rather than PM, as vast PM was not available at the time of the original evaluation. Given the longer latencies of PM devices as compared to DRAM, it stands to reason that page tables will remain in DRAM for the near future. Thus, our results stay unaffected as the high cost of address translation that we improve will either stay constant or increase further. Also, this chapter differs from the original paper in one significant way along with other cosmetic changes. In light of recent microarchitectural side-channel attacks [83, 77], we have eliminated a Meltdown-susceptible preload optimization from our default design and made it optional for use when security is not essential.

1.3 Thesis Organization

This thesis is organized as follows. Chapter 2 provides the background on memory persistence, needed for Chapters 3 and 4. Chapter 3 presents efficient hardware primitives for specifying ordering and durability requirements in PM applications. Chapter 4 describes the design of recoverable datastructures that improve PM application performance on unmodified hardware. Then, we shift our discussion from persistence to vast memory by developing a low-overhead virtual memory technique in Chapter 5. Finally, Chapter 6 summarizes our contributions and discusses future research directions.

— 2 —

Persistent Memory Background

So much universe, and so little time.

TERRY PRATCHETT

In this thesis, we tackle the challenges caused by persistent and vast memory. This chapter provides some background details on the newfound persistence of memory, as it is a more radical development than increased memory capacity. We discuss the relevant background for vast memory systems in Chapter 5.

2.1 System Model

In our target systems, the physical address space is partitioned into volatile DRAM and durable PM, as shown in Figure 2.1. Hardware structures may be either in the volatile or the persistent domain. In case of a system failure or power down, data in the persistent domain is preserved while structures in the volatile domain such as DRAM, CPU registers, caches, etc., are wiped clean and their data is lost. This system model is similar to most prior work [20, 85, 94, 139] and representative of Optane DCPMM [67, 70].

Some systems may contain optimizations that extend the persistent domain to additional structures. For instance, Asynchronous DRAM Refresh (ADR) on Optane DCPMMs brings the memory controller for persistent memory (PM-MC) into the persistent domain by

providing sufficient battery backup to flush out all buffered data at the PM-MC to PM in case of system failure [63]. With such optimizations, applications must no longer wait for data to be written to PM (slower than DRAM) to be considered durable.

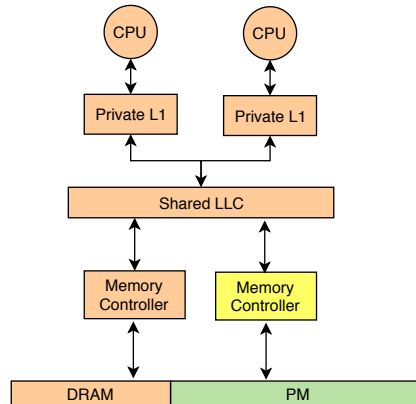


Figure 2.1: Target System with both DRAM and PM. Structures colored orange are in the volatile domain, green in the persistent domain and yellow in the persistent domain only in presence of certain optimizations.

2.2 Hardware Primitives for Ordering and Durability

Both x86-64 and ARMv8 architectures have added (or plan to add) support for PM in the form of two types of instructions for durability and ordering: flush instruction variants to explicitly writeback a cacheline from the volatile caches to the persistent domain, and fence instruction variants to order subsequent instructions after preceding flushes become durable.

In this dissertation, we focus on the x86-64 architecture, which supports both strongly ordered flushes (`clflush`) and weakly ordered flushes (`clflushopt`, `clwb`). Strongly ordered flushes stall the CPU pipeline till the flush is acknowledged as completed. In contrast, weakly ordered flushes perform flushes as a posted operation and commit instantly, as shown in Figure 2.2. However, ordering points (`sfence`) stall the CPU until inflight flushes are completed. Thus, frequent ordering points degrade performance by bringing the flush

	Ordering	Durability	Granularity	Other Effects
<code>clflush</code>	Yes	Yes	Cacheline	-
<code>clwb,clflushopt</code>		Yes (with <code>sfence</code>)	Cacheline	-
<code>sfence</code>	Yes (with <code>clwb</code>)		Control-flow	Serializing Instruction

Table 2.1: Comparison of x86-64 primitives for PM.

latency of weakly ordered flushes to the critical path. In the rest of this dissertation, we use the term flushes to refer to weakly ordered flushes.

The two weakly ordered flushes in the x86-64 architecture have different behavior. The `clflushopt` instruction evicts a cacheline from the cache to write it back to PM. On the other hand, the `clwb` instruction offers a hint to the hardware that the cacheline may be used in the future. Hence, an optimized implementation would clean the appropriate cacheline by writing back the dirty data but not evict it from the caches. In the implementations discussed in the following chapters, we use `clwb` instructions to flush out data to PM.

We discuss the drawbacks of these primitives by comparing against our proposed primitives in Section 3.2. One major issue is that the x86-64 primitives are too fine-grained and operate at a cacheline granularity, which complicates and slows the development of recoverable applications. Specifically, programmers are required to perform manual data movement at the granularity of individual cachelines. We present a comparison of these primitives in Table 2.1.

2.3 Programming Recoverable Applications

We refer to applications that leverage the persistence of PM as *recoverable applications*. Such applications store some or all application state durably in PM. When such applications start executing, they check the state of their data in PM. If they find durable state from an earlier execution, they can use it to restore progress made in previous executions instead of starting from scratch.

Two major challenges exist in programming for PM. First, data is only durable when it

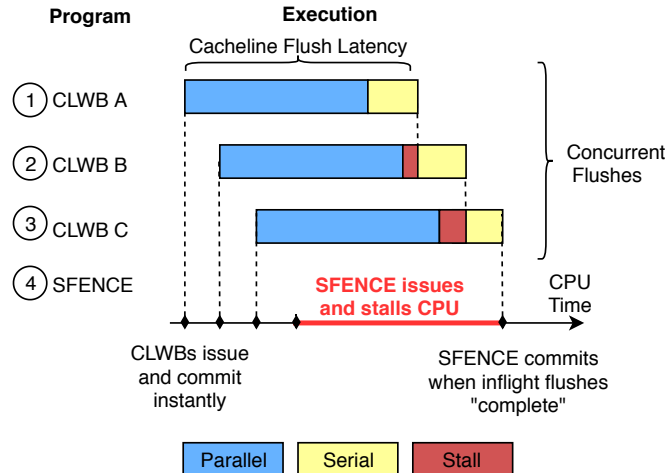


Figure 2.2: Execution Timeline for weakly ordered flushes to PM on x86-64 systems.

reaches the persistent domain; data in the volatile domain is lost on a power failure. As a result, applications that require durability must ensure that sufficient data is moved into the persistent domain to preserve application progress in case of failure. Second, system crashes at inopportune moments such as amidst datastructure modifications may result in partially updated and inconsistent datastructures. Hence, applications must explicitly order flushes to PM, to ensure that recovery code can restore any partially updated data to a consistent and usable state.

Essentially, recoverable PM applications need to order some PM writes before others to enable consistent recovery, e.g., log write before corresponding data write. To achieve this, such applications need to reason about how the underlying hardware may reorder PM stores. Accordingly, these applications can indicate specific ordering constraints that must be enforced by the hardware to avoid data corruption.

In this section, we first describe *persistence models* that are specified by architectures to help programmers reason about ordering of PM stores. As these persistence models are too low-level for most application programmers, researchers have also developed high-level programming models for PM, which we describe subsequently.

Low-level Persistency Models

Persistency models are specifications of possible orders in which stores update PM [105]. Persistency models are an extension of memory consistency models which specify possible orders in which stores become visible to all cores.

With the introduction of PM, programmers must now reason about two separate memory orders in addition to program order. *Program order* (PO) refers to the total order in which instructions are dynamically executed by a software thread. *Volatile memory order* (VMO) is governed by the memory consistency model and is the order in which memory operations become visible to all cores. Additionally, we now have the *persistent memory order* (PMO) which is the order in which stores update PM. A prefix of the stores in PMO is guaranteed to have updated PM and become durable in case of a power outage.

There are two popular persistency models: Strict and Buffered Epoch persistency [105, 72]. **Strict persistency** ties the persistency model to the consistency model. At the time of failure, any updates that are visible as per VMO are also made durable. Logically, each store to PM is made durable before the next store in program order can be issued. Hence, strict persistency aims for simplicity at the cost of performance. **Buffered Epoch Persistency** (BEP) is a relaxed model that enforces ordering at an *epoch* granularity. An epoch is a group of instructions from one thread contiguous in program order. PM stores within the same epoch can be made durable in any order. A PM store from an earlier epoch is made durable before a PM store from a later epoch. Thus, programmers can order two PM stores by separating them with an epoch boundary. Note that BEP as defined above allows for PM stores to be buffered, i.e., does not require PM stores to become durable synchronously at the end of their epoch. Thus, BEP enables high-performance implementations that allow the core to execute past epoch boundaries.

For the proposals in this thesis, we limit our discussions to the x86-TSO consistency model and Buffered Epoch Persistency only.

High-level Programming Models

A popular high-level programming model that abstracts away the complexities of the underlying persistency model is *failure-atomic code sections* (FASEs) [20]. FASEs are code segments with the guarantee that all PM writes within a FASE happen atomically with respect to system failure. For example, prepending to a linked list (Figure 4.2b) in a FASE guarantees that either the linked list is successfully updated with its head pointing to the durable new node or that the original linked list can be reconstructed after a crash. FASEs are commonly implemented either as durable transactions, custom logging mechanisms in lock-based code [20] or shadow-copying techniques [136].

PM libraries [26, 61, 139] typically implement FASEs with software transactions (PM-STM) that guarantee atomicity, consistency and durability. All updates made within a transaction are durable when the transaction commits. If a transaction gets interrupted due to a crash, write-ahead logging techniques are used to allow recovery code to clean up partial updates and return persistent data to a consistent state. Hence, recoverable applications can be written by allocating datastructures in PM and only updating them within PM transactions.

<pre>Epoch 1: pt->x = 1 pt->y = 1</pre> <p>(a)</p>	<pre>// Begin-FASE pt->x = 1 pt->y = 1 flag = 1 // End-FASE</pre> <p>(b)</p>	<pre>TX_BEGIN { pt->x = 1 pt->y = 1 flag = 1 } TX_END</pre> <p>(c)</p>
--	--	--

Figure 2.3: Different PM programming approaches illustrated using a sample program which updates a persistent struct `pt` and then set a persistent `flag`: (a) Low-Level Epochs, (b) High-level failure-atomic code sections, (c) Software Transactional Memory for PM.

PM-STM implementations perform write-ahead logging with either undo-logging [26, 61], redo-logging [30, 43, 84, 139] or hybrid undo-redo logging [62, 100]. These techniques require a durable log (stored in PM). Before every store to an address in PM, we append a new log entry with the address of the store and either the new value (in redo logging) or the

old value at that address (in undo logging). Subsequently, the store is performed in-place overwriting the old value (undo logging) or buffered elsewhere (redo logging) until commit. To enforce these ordering rules, i.e., log append is ordered before data write, transactions rely on epoch boundaries. On the x86-64 hardware, these epoch boundaries guarantee durability in addition to ordering resulting in inefficient and slow transactions [94]. A transaction implemented with undo-logging and broken down into logical epochs is shown in Figure 2.4.

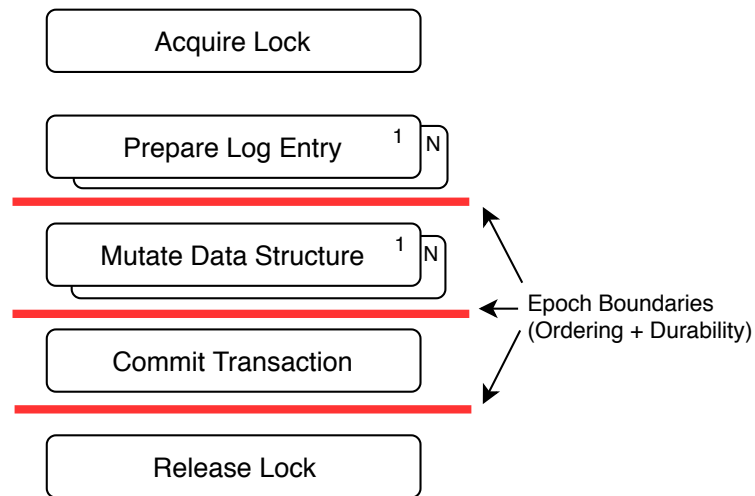


Figure 2.4: Example of STM implementation for PM with undo-logging.

Qualitative Comparison

We illustrate the difference between low-level epochs and higher-level programming models using a simple program (Figure 2.3). This program first updates `struct pt` which has two members `x` and `y` and is stored durably in PM. Then, the program uses a `flag` stored in PM to indicate that `pt` is updated and durable. The ordering required is that the `flag` update must not happen earlier in PMO than the `pt` updates. We can achieve this by first using an epoch to update `pt` and using a subsequent epoch to update `flag` (Figure 2.3a). Here, updates to `x` and `y` can be freely reordered by hardware without adversely affecting consistent recovery. Alternatively, we can perform all updates in one FASE (Figure 2.3b). If the FASE completes

successfully, all updates will happen atomically in PMO, thereby preserving the required ordering. If there is a crash within the FASE, recovery code will undo all updates from this incomplete FASE. Lastly, while epochs and FASEs are logical constructs, we show how the actual implementation of the code using a PM-STM implementation (Figure 2.3c).

This chapter has provided background on PM systems, hardware primitives and programming models, which now enables us to move on to this dissertation's contributions.

— 3 —

Hands-Off Persistence System

Memory can play tricks after the first ten thousand years.

TERRY PRATCHETT

¹ The persistence of emerging NVM devices marks a paradigm shift as it collapses the traditional storage/memory hierarchy: applications can access and persist data directly with the interface and performance of memory and the persistence of storage. Such persistent memory enables *recoverable applications* that preserve in-memory data across planned and unplanned outages. These applications can access durable data in PM at the granularity of bytes and persist complex pointer-rich datastructures without needing to serialize them to disk. Such properties are highly desirable for workloads like databases, key-value stores and long-running scientific computations [18, 82].

However, the potential of persistent memory is marred by both excessive overheads and programming challenges. Regular loads and stores can be used to access data in PM, but these memory operations can be reordered by modern CPUs and memory subsystems. Some reorderings could result in data loss or corruption in the case of an ill-timed system crash. To be able to recover after a crash without losing all prior progress, applications must carefully order and persist stores to PM. For instance, applications must update and

¹The introductions of Chapters 3 and 4 have some redundancies with Chapter 1 to facilitate readers mostly interested in reading a standalone chapter.

flush data before updating a persistent pointer that points to the data, or atomically do both.

To enable consistent and durable updates to PM, Intel has recently added new hardware primitives to the x86-64 architecture for recoverable applications. Applications can use cacheline flush instructions to move data from volatile caches to durable PM. Additionally, applications order these flushes carefully using fence instructions to ensure consistency.

These new x86-64 primitives suffer from both performance and programmability issues. First, these primitives enforce ordering between two PM stores by waiting for the first store to become durable, which is a slow operation due to the high NVM write latency. Second, programmers are required to explicitly perform data movement at a cacheline-granularity from the volatile caches to persistent memory. Such low-level primitives complicate and slow the development of PM applications. Thus, there is a need for efficient, programmer-friendly and hardware-enforced primitives for ordering and durability separately.

We propose the Hands-Off Persistence System (HOPS) to make recoverable applications both fast and easy to program. We use insights from our comprehensive analysis of realistic PM applications to guide the HOPS design. In our analysis, we found that applications require ordering guarantees much more frequently than durability. Thus, we introduce two distinct hardware primitives, a more common, lightweight, ordering fence (`ofence`) and the rarer durability fence (`dfence`). HOPS can order PM stores without making them durable to improve performance. Moreover, our proposed primitives take no parameters and operate at the control-flow level and not at a cacheline-granularity.

HOPS tracks and buffers PM stores in hardware, and automatically enforces the ordering and durability requirements indicated by the programmer. HOPS makes minimal changes to the existing cache hierarchy to avoid degrading the performance of accesses to volatile DRAM, which comprise 96% of all accesses in the PM applications we studied. Furthermore, HOPS can buffer multiple stores to the same cacheline from different epochs (a common occurrence) without requiring long-latency cacheline flushes. Beyond the improvements

in programmability, our evaluation shows that HOPS improves application performance by 24% on average compared to a baseline x86-64 system.

In this chapter, we attempt to improve the performance and programmability of recoverable PM applications, making the following contributions:

- We offer design guidelines for efficient PM hardware based on observations from realistic PM applications.
- We propose new hardware primitives, `ofence` and `dfence`, to enable programmers to reason about ordering and durability at a high-level
- We present an efficient hardware design to implement our new hardware primitives by track PM stores in hardware.

3.1 Insights from Workload Analysis

We believe that analysis precedes good design. Accordingly, we created the first benchmark suite of realistic recoverable PM applications called Wisconsin-HP Labs Suite for Persistence or WHISPER. We analyzed these applications to find characteristic trends to guide the design of HOPS. Another student (Sanketh Nalli) led the development of the benchmark suite and its analysis to which we contributed. Here, we summarize the key observations of our analysis along with the design goals inspired by each observation. We published the full analysis in Section 4-5 of our ASPLOS paper [94], where Section 6 presented HOPS, the subject of this dissertation chapter.

Observation 1: Accesses to volatile DRAM make up about 96% of all accesses.

Design Goal 1: Any PM-specific additions to caches and other structures shared between PM and DRAM should not adversely impact volatile memory accesses.

Observation 2: ACID transactions involve 5-50 ordering points.

Design Goal 2: As ordering points occur frequently, they must be fast and inexpensive.

More importantly, ordering must be decoupled from more expensive durability operations.

Observation 3: Epochs from different threads rarely conflict with each other.

Design Goal 3: Hardware can separately track PM writes from each thread, but rare inter-thread conflicts must still be handled correctly.

Observation 4: There are frequent conflicts between epochs from the same thread.

Design Goal 4: Intra-thread conflicts lead to flushing on the critical path, as dirty cachelines from older epochs must be flushed first to avoid reordering epochs. To prevent this, hardware must be capable of simultaneously tracking multiple updates to the same cacheline from different epochs.

In subsequent sections, we use these design guidelines to drive the design of efficient hardware support for PM.

3.2 New Hardware Primitives

HOPS adds two hardware primitives to allow programmers to specify ordering as well as durability requirements separately and at a high-level, e.g., without specifying individual cachelines to be persisted. These primitives are thread-local and apply to stores from the same thread, like the x86-64 primitives for PM. Here, we describe the semantics of these primitives and defer the discussion of an efficient hardware implementation until Section 3.4.

Ordering. First, we add a lightweight *ordering fence* (*ofence*) that always executes immediately and does not stall the core. In the absence of *ofence* instructions, stores can update PM in any order. The *ofence* instruction ensures that all stores from the same thread earlier than the *ofence* in program order will update PM before stores from the same thread later than the *ofence* in the program order. Figure 3.1a illustrates the behavior of *ofence*.

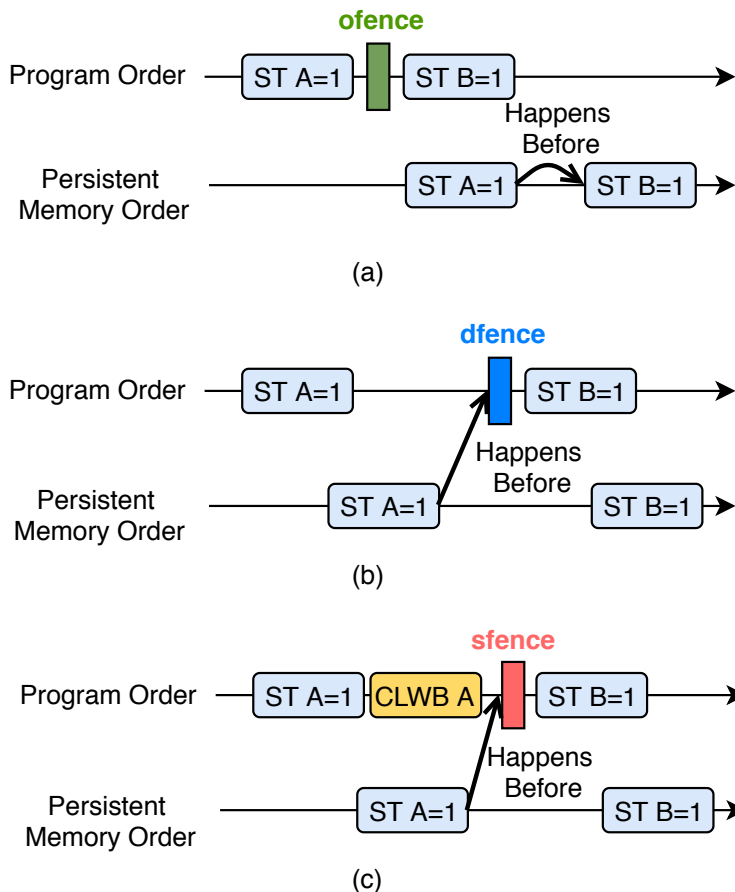


Figure 3.1: Guarantees provided by (a) `ofence`, (b) `dfence` and (c) x86-64 primitives. In (c), if the `clwb` is removed, there will not be any happens-before relationship.

Durability. Second, we propose the *durability fence* (`dfence`) that stalls the core until all outstanding PM stores from the same thread become durable. The `dfence` instruction synchronously guarantees the durability of all PM stores (from the same thread) earlier in the program order than the `dfence`. Due to the high cost of making data durable, `dfence` is more expensive than `ofence`. Figure 3.1b illustrates the behavior of `dfence`. Operating Systems can use the `dfence` instruction to ensure the durability of outstanding PM stores on context switches.

Constructing Epochs. Programmers can use either of the two primitives to demarcate an epoch boundary as two PM stores cannot be reordered if separated by either `ofence` or `dfence`. The difference between the two primitives lies in the fact that the `ofence` primitive

does not guarantee that preceding PM stores have become durable, similar to theoretical epoch boundaries [105]. Accordingly, we use the lightweight ofence primitive for all epoch boundaries except when durability of earlier stores is a requirement. For instance, programmers can use ofence for internal ordering points within a transaction and dfence only at the end of the transaction to guarantee durability or before performing irreversible and externally visible I/O operations. We show an example of an undo-logging based transaction implemented using both x86-64 and HOPS primitives in Figure 3.2. The x86-64 implementation uses heavyweight epoch boundaries that guarantee durability at all internal ordering points of the transaction, resulting in low performance.

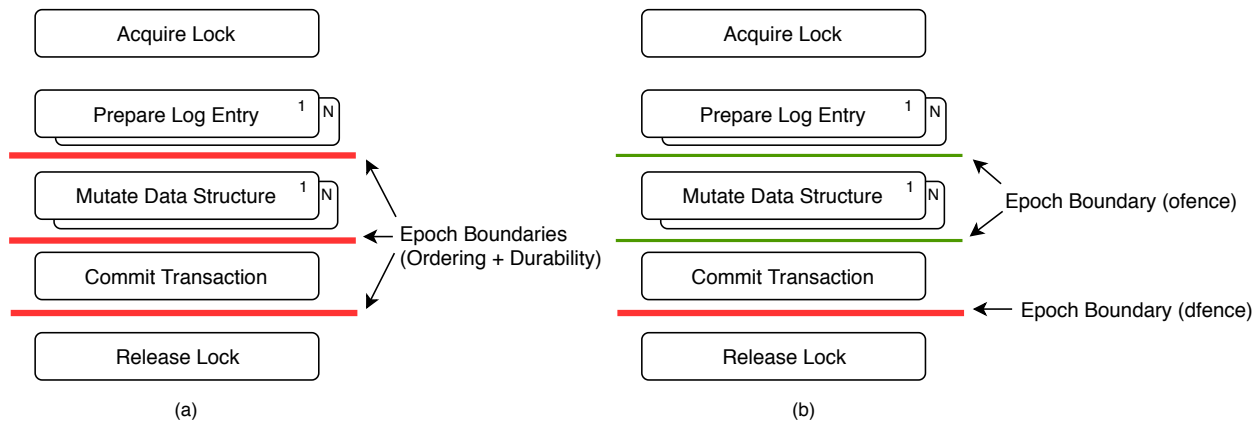


Figure 3.2: PM-STM implementation with (a) x86-64 primitives and (b) HOPS primitives.

Comparison with x86-64 primitives. The ofence and dfence primitives offer two advantages over existing primitives in the x86-64 architecture like clwb and sfence. First, x86-64 primitives do not distinguish between ordering and durability. To order two PM stores in an x86-64 system, programmers must make the first store durable (an expensive operation) before performing the second store. Effectively, the x86-64 primitives ensure durability of inflight stores at every epoch boundary, which is overkill. This behavior is illustrated in Figure 3.1c as the update to address A must be made durable before the sfence commits. This is a significant performance bottleneck as our application analysis [94] found that applications rely on ordering guarantees much more frequently than on durability

Architecture	Primitive	Ordering	Durability	Other Effects
x86-64	clflush	Yes	Yes	-
x86-64	clwb,clflushopt		Yes (with sfence)	-
x86-64	sfence	Yes (with clwb)		Serializing Instruction
HOPS	ofence	Yes		-
HOPS	dfence	Yes	Yes	Serializing Instruction

Table 3.1: Comparison of HOPS and x86-64 primitives for PM.

guarantees.

Second, HOPS primitives allow programmers to reason about ordering and durability points in an application’s control flow. In contrast, existing x86-64 primitives like `clwb` and `sfence` force programmers to manually move individual cachelines to durable PM. Missing or misplaced `clwbs` have been shown to be a major cause of correctness and performance issues in PM applications [86]. Figure 3.3 shows how a simple program with two logical epochs is represented using x86-64 and HOPS primitives. This figure also shows how PM programs written with HOPS primitives closely resemble the abstract and intuitive epochs.

We present a qualitative comparison of HOPS and x86-64 primitives in Table 3.1.

Epoch Model	x86-64	HOPS
Epoch 1: <code>pt->x = 1</code> <code>pt->y = 1</code>	<code>pt->x = 1</code> <code>pt->y = 1</code> <code>clwb (&(pt->x))</code> <code>clwb (&(pt->y))</code>	<code>pt->x = 1</code> <code>pt->y = 1</code> <code>ofence</code>
Epoch 2: <code>flag = 1</code>	<code>sfence</code> <code>flag = 1</code> <code>clwb (&flag)</code> <code>sfence</code>	<code>flag = 1</code> <code>dfence</code>
(a)	(b)	(c)

Figure 3.3: Comparing HOPS and x86-64 primitives. Sample program written with (a) epochs, (b) x86-64 primitives and (c) HOPS primitives. Programmer intends for all PM stores to be durable at the end of this program.

3.3 Memory Persistency Model

We now discuss the memory persistency model supported by HOPS. As described earlier in Chapter 2, these models specify the possible orderings in which PM stores can become durable [105]. Consequently, these models help programmers reason about how to use available hardware primitives to avoid reorderings of updates that preclude effective crash recovery.

HOPS implements the Buffered Epoch Persistency model (BEP) [72, 105] on top of the x86-TSO memory consistency model [120]. This combination (TSO-BEP) allows us to extend a simple and well-understood memory consistency model with a persistency model that allows useful performance optimizations. Specifically, BEP allows the overlapping of long-latency flushes to PM for stores belonging to the same epoch. While the BEP model has been described previously with prose [72, 105] (also in Section 2.3), we seek to precisely specify it using axiomatic notation here. An operational and axiomatic model of TSO-BEP using slightly different hardware primitives has been developed by others [111] since our original paper.

We use the following notation for memory operations (inspired by [81]) to describe these models:

- L_a^i : Load from thread i to address a .
- S_a^i : Store from thread i to address a .
- M_a^i : Load or Store from thread i to address a .
- F^i : mfence (from x86-64 ISA) on thread i .
- X^i : ofence or dfence (from HOPS) on thread i .

To express the ordering of memory operations, we define three partial orders that each order a subset of memory references:

- $M_a^i <_{po} M_b^i$: M_a^i precedes M_b^i in program order (PO).
- $M_a^i <_{vmo} M_a^j$: M_a^i precedes M_a^j in volatile memory order (VMO).
- $M_a^i <_{pmo} M_a^j$: M_a^i precedes M_a^j in persistent memory order (PMO).

Below, we describe the specific memory references that are ordered in the x86-TSO consistency model and BEP persistency model in HOPS. For instance, PMO only orders some stores and never loads, ofences or dfences. This is because PMO defines the state of PM at the time of failure, and PM is only updated by stores. PO is the order in which instructions from the same thread are dynamically executed by the processor.

Background on x86-TSO

x86-TSO refers to the Total Store Order memory consistency model that appears to match the memory consistency model implemented in the x86-64 architecture. It preserves the relative order of most pairs of memory accesses (Equations 3.1-3.3) except for the store-to-load ordering. Particularly, a store followed by a load in PO are not guaranteed to be ordered equivalently in the VMO. To enforce such an ordering, the two operations must be separated by an `mfence` operation (Equation 3.4).

$$L_a^i <_{po} L_b^i \Rightarrow L_a^i <_{vmo} L_b^i \quad (3.1)$$

$$L_a^i <_{po} S_b^i \Rightarrow L_a^i <_{vmo} S_b^i \quad (3.2)$$

$$S_a^i <_{po} S_b^i \Rightarrow S_a^i <_{vmo} S_b^i \quad (3.3)$$

$$S_a^i <_{po} F^i <_{po} L_b^i \Rightarrow S_a^i <_{vmo} L_b^i \quad (3.4)$$

HOPS Persistency Model

We now discuss the ordering of stores in PMO as per the TSO-BEP model. We first present the simpler case of ordering two stores to the same address, followed by a longer discussion on ordering two stores to different addresses.

Stores to the Same Address

TSO-BEP always orders two stores to the same address as per VMO (Equation 3.5). This property is referred to as *strong persist atomicity* [105]. One special case is for two stores to the same address from the same thread (i.e., S_a^i and \hat{S}_a^i), whose relative order in VMO (and thus PMO) is determined by PO (Equation 3.6). Across threads, the relative order in VMO depends on how memory accesses from different threads get interleaved at the caches. Programmers can influence this order by using appropriate synchronization techniques.

$$S_a^i <_{\text{vmo}} S_a^j \Rightarrow S_a^i <_{\text{pmo}} S_a^j \text{ /* Same address */} \quad (3.5)$$

$$S_a^i <_{\text{po}} \hat{S}_a^i \Rightarrow S_a^i <_{\text{pmo}} \hat{S}_a^i \text{ /* Same address, same thread */} \quad (3.6)$$

Stores to Different Addresses

To describe the ordering of stores to different addresses, we introduce the relation *epoch-happens-before* ($\xrightarrow{\text{ehb}}$). Before defining this relation, we assert that TSO-BEP only orders two stores to different addresses in case of *epoch-happens-before*:

$$S_a^i \xrightarrow{\text{ehb}} S_b^j \Rightarrow S_a^i <_{\text{pmo}} S_b^j \text{ /* Different addresses */} \quad (3.7)$$

With three equations, we define *epoch-happens-before* using a simple base case (Equation 3.8) and extend it recursively in two ways (Equations 3.9 and 3.10). In the base case, store $S1 \xrightarrow{\text{ehb}} S2$ if they are from the same thread and separated by an epoch boundary, i.e., ofence or dfence. In other words, store $S1$ belongs to an earlier epoch and store $S2$

belongs to a later epoch.

$$S_a^i <_{po} X^i <_{po} S_c^i \Rightarrow S_a^i \xrightarrow{ehb} S_c^i \text{ /* Same Thread */} \quad (3.8)$$

The ordering of PM stores to different addresses from different threads is more challenging. We want to order such stores only in case of explicit dependencies and not because they happen to be arbitrarily ordered in VMO in an execution. For example, data updates on one thread must be ordered before another thread updates a pointer to point to the data. We illustrate this case using Figure 3.4 where the programmer's intent is to order S4 after S1 in PMO. We revisit this example at the end of this section to describe how TSO-BEP preserves the necessary ordering. However, two unrelated stores from different programs must not be ordered.

Thread 1	Thread 2	Thread 3
S1: ST data = 1	L1: LD flag1, r1	L2: LD flag2, r2
<i>ofence</i>	if (r1 != 1) goto L1	if (r2 != 1) goto L2
S2: ST flag1 = 1	S3: ST flag2 = 1	S4: ST ptr = &data

Figure 3.4: Example of necessary ordering (PMO) of stores to different addresses from different threads. Using two flags, programmer desires S4 to be ordered after S1 in PMO.

Accordingly, we seek to order stores in presence of *read-after-write* (RAW) dependencies in VMO to enable many forms of cross-thread communication and synchronization. For this, we use the *reads-from* (rf) relation $S_a^i \xrightarrow{rf} L_a^j$ [118], that indicates load L_a^j returns the value written previously by store S_a^i during an execution, i.e., a RAW dependency from S_a^i to L_a^j in VMO.

We now extend the definition of *epoch-happens-before* to include *reads-from* dependencies:

$$M_a^i \xrightarrow{ehb} S_c^j \xrightarrow{rf} L_c^k \Rightarrow M_a^i \xrightarrow{ehb} L_c^k \text{ /* reads-from across threads */} \quad (3.9)$$

The above equation is not useful on its own as the order of loads is not reflected in PMO. However, our real intention is to correctly order any stores following these ordered loads in program order. Hence, we add the final extension to the definition of *epoch-happens-before*:

$$M_a^i \xrightarrow{ehb} M_b^j <_{po} S_c^j \Rightarrow M_a^i \xrightarrow{ehb} S_c^j \text{ /* program order within thread */} \quad (3.10)$$

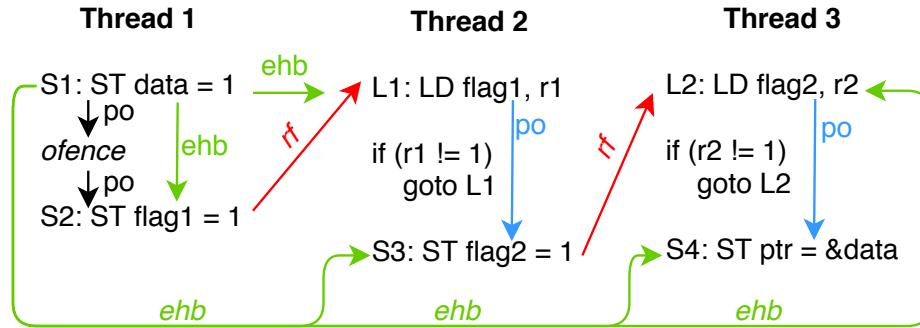


Figure 3.5: Example of *reads-from* (rf) and *epoch-happens-before* (ehb) relations.

We demonstrate *epoch-happens-before* and *reads-from* relations in our earlier example (Figure 3.4) using Figure 3.5. Under all executions, $S1 \xrightarrow{ehb} S2$ due to the intervening of *ence* (per Equation 3.8). When the if-statement on Thread 2 is evaluated true, we know that L1 read the value written by S2, i.e. $S2 \xrightarrow{rf} L1$. According to Equation 3.9, $S1 \xrightarrow{ehb} S2 \xrightarrow{rf} L1$ results in $S1 \xrightarrow{ehb} L1$. According to Equation 3.10, $S1 \xrightarrow{ehb} L1 <_{po} S3$ results in $S1 \xrightarrow{ehb} S3$. By extending this similarly to Thread 3, we can also show that: $S1 \xrightarrow{ehb} S4$. Accordingly, S1 precedes S4 in PMO (per Equation 3.7), correctly preserving programmer intent. In this example, note that there is no *epoch-happens-before* relation between S2 and S4.

One might think that our three-equation (Equations 3.8-3.10) definition of *epoch-happens-before* is unduly complex. Alternatively, it might seem simpler to order persistent stores with the transitive closure of program order and reads-from. While simpler, this alternative formulation orders stores too much and would be detrimental to performance. In particular, it would order in PMO, two stores by a thread to different addresses in the same epoch (no intervening of *ence*), making it difficult to concurrently handle the stores, e.g., overlap

	Ordering Rules
Stores to Same Address	$S1 <_{vmo} S2 \Rightarrow S1 <_{pmo} S2$
Stores to Different Addresses	$S1 \xrightarrow{ehb} S2 \Rightarrow S1 <_{pmo} S2$

Table 3.2: Ordering rules in TSO-BEP.

flushes to separate persistency memory controllers. Our definition leaves these stores unordered, facilitating concurrent handling and good performance, as we will see.

We summarize all store-to-store orderings in Table 3.2. Programmers can use these rules to ensure that any required ordering in PMO between stores is enforced by HOPS.

3.4 HOPS Design

We now describe the HOPS design that efficiently implements our new hardware primitives. First, we present the timestamp mechanism that enforces our persistency model. Then, we describe the hardware structures introduced in HOPS to implement this mechanism. Lastly, we evaluate the HOPS design qualitatively against our design goals from Section 3.1.

Timestamp-based Ordering

HOPS uses a mechanism inspired by the vector clocks algorithm [40, 89] for determining the order in which stores can update PM (i.e., PMO). All hardware threads get a local timestamp (LocalTS) that indicates the timestamp of the current inflight epoch. Every PM store is associated with a timestamp that matches the LocalTS when the store exits the CPU write buffer. Epoch boundaries (i.e., ofence or dfence) increment the LocalTS.

Ordering stores from one thread

For single-threaded applications, the timestamps associated with PM stores are sufficient to determine relative positions in the PMO. Specifically, a PM store with timestamp LTS_i is

ordered after other stores from the same thread with timestamp $< LTS_i$ in PMO i.e., stores from earlier epochs are ordered before stores from later epochs. There are no ordering constraints among stores with equal timestamps, i.e., they belong to the same epoch.

Ordering stores from different threads

For multi-threaded applications, HOPS must enforce ordering between stores from two different threads, even as our analysis showed these cases to be rare (Table 3.2). To handle such instances, each store is also associated with a *dependency tuple* $(T_i:LTS_i)$ that consists of a thread id (T_i) and LocalTS (LTS_i) on that thread. HOPS identifies such dependencies by monitoring coherence activity as described later in Section 3.4. In the absence of any such dependencies (common case), the dependency tuple is empty. In PMO, a store with dependency tuple $(T_i:LTS_i)$ is ordered after stores from thread T_i with timestamps $< LTS_i$. Additionally, this store must also follow ordering rules described in the previous paragraph.

A cross-thread dependency is also treated as an epoch boundary, leading to the creation of a new epoch to avoid any circular dependencies. This approach is borrowed from the epoch deadlock avoidance mechanism proposed by others [72].

Figure 3.6 illustrates the use of timestamp-based ordering in HOPS. Most stores in the figure are only associated with a timestamp, which orders them with other stores from the same thread. The store to address G from thread 0 additionally has a dependency tuple that indicates it should be ordered after all stores from thread 1 with timestamp ≤ 3 . In this particular example, the cross-thread dependency occurs due to strong persist atomicity (Section 3.3, Equation 3.5).

HOPS Hardware Modifications

In this section, we describe the new hardware introduced in the HOPS implementation, namely timestamp registers and persist buffers. The HOPS system design is illustrated in Figure 3.7.

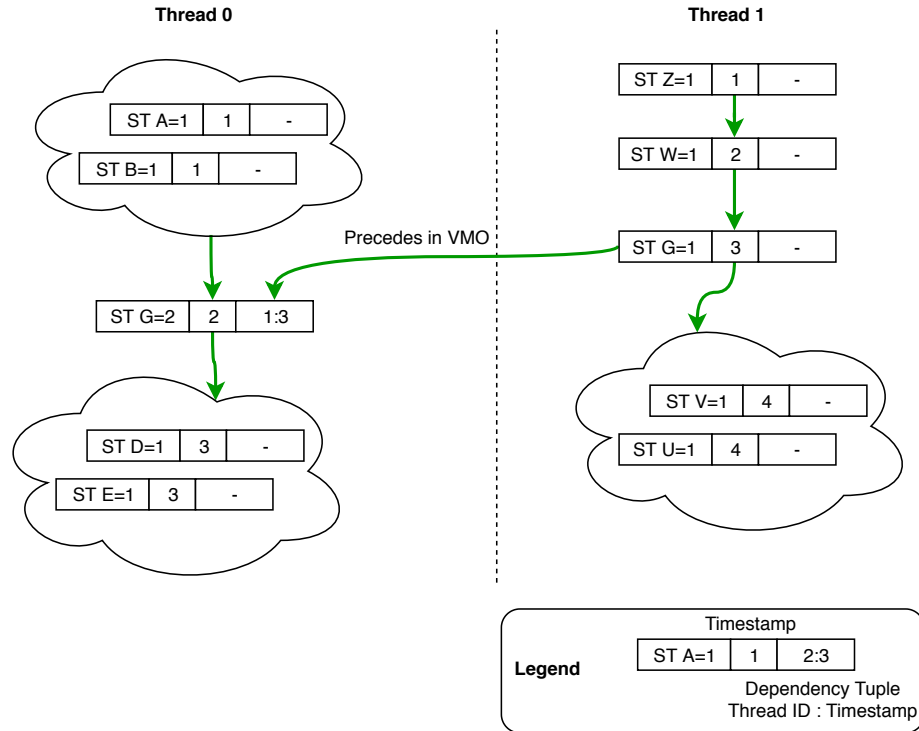


Figure 3.6: Simple example showing the use of timestamps and dependency tuples for indicating PMO happens-before relationships in HOPS. Stores shown inside a cloud are in the same epoch and thus unordered with respect to each other.

Timestamp Registers

HOPS uses timestamps for ordering PM updates. We add a local timestamp (LocalTS) register for each hardware thread, which indicates the epoch number of the inflight epoch. The LocalTS is incremented on an of fence or dfence. A global timestamp (GlobalTS) register is maintained at the LLC, which is a vector of timestamps (T_0, T_1, \dots, T_n) where T_i indicates that all stores from thread i with timestamp $< T_i$ have been made durable. The GlobalTS is updated based on messages received from the Persist Buffers (described next). Note that the LocalTS and GlobalTS values are unaffected by a thread context switch.

Persist Buffers

Persist Buffers (PBs) are per-thread first-in first-out (FIFO) structures that buffer inflight PM stores before they become durable. When a PM store exits the write buffer, the store

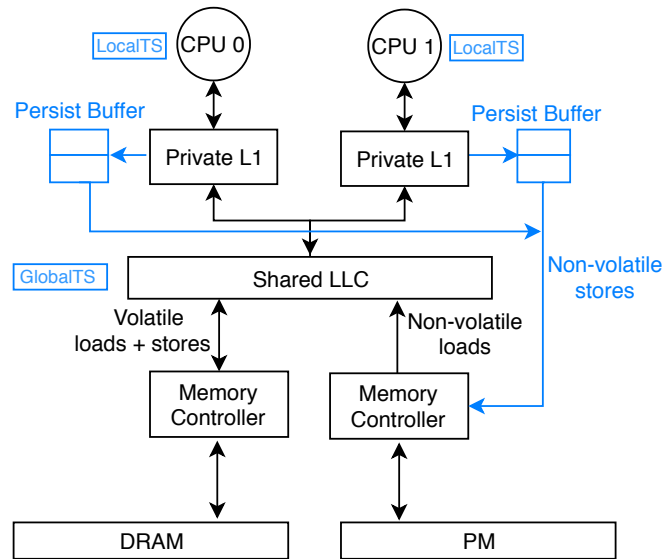


Figure 3.7: HOPS System Design, with hardware modifications marked in blue.

gets buffered at the tail of its thread’s Persist Buffer and also updates the appropriate cacheline in the caches. The cached copy is used for exploiting temporal or spatial locality but never written back to PM. Instead, the cacheline is simply discarded when it gets evicted. Note that the PBs are multi-versioned which allows multiple updates to the same address to be buffered separately as part of the same or different epochs. This allows PBs to handle self-dependencies without stalling. On a context switch, the core’s Persist Buffer is completely flushed using a `dfence` instruction.

PBs implement the timestamp mechanism (Section 3.4) to write back the buffered stores to PM in a legal order under TSO-BEP. For this purpose, each PB entry contains the address and data of a PM store along with the store’s timestamp and dependency tuple. Starting from the head, a PB writes back buffered stores following the ordering rules specified in the previous section. If the buffered entries have empty dependency tuples, a PB concurrently writes back all entries with the same timestamp by sending write requests to the appropriate memory controllers. Before proceeding with stores with later timestamps, the PB waits for memory controllers to acknowledge the durability of previous write requests.

Finally, PBs rely on the GlobalTS register to enforce rare cross-thread dependencies. At

regular intervals, a PB updates the GlobalTS register with the timestamp of the latest store written back to PM. Then for any buffered store with a dependency tuple $(T_i:LTS_i)$, the PB polls the GlobalTS register to ensure that the timestamp corresponding to thread T_i is $> LTS_i$. Once this condition is met, the buffered store is guaranteed to be ordered correctly in the PMO as per our TSO-BEP ordering rules. At this point, the buffered store can be written back to PM.

Figure 3.8 demonstrates the role of PBs for implementing ofence and dfence instructions in a sample code sequence.

Extensions to Cache Coherence Mechanism

HOPS monitors cache coherence activity to identify rare cross-thread RAW dependencies. Consider the case when a (*source*) thread writes to an address resulting in a dirty cacheline in the thread's L1 cache. Subsequently, if another (*dependent*) thread issues a read request for the same address, its L1 cache requests a shared copy of the cacheline. This creates a cross-thread RAW dependency that can be identified by an incoming coherence request from another cache for a dirty cacheline in an L1 cache. This is a conservative indication due to the possibility of false-sharing.

This RAW dependency is communicated to the dependent thread's PB by sending a dependency tuple along with the coherence response. The dependency tuple is created using the source thread's ID as well as its LocalTS. The timestamp in the dependency tuple should ideally be the TS associated with the original write to the cacheline. However, this requires a fully associative lookup in the PB for the relevant address. To avoid this expensive operation, we use the LocalTS which is acceptable as it is more recent than the timestamp of the original write.

When the dependent thread's L1 cache receives a coherence response with an additional dependency tuple, it forwards the tuple to the thread's PB. The PB appends the dependency tuple to its tail entry. Thus, the next PM store performed by the thread will get a PB entry

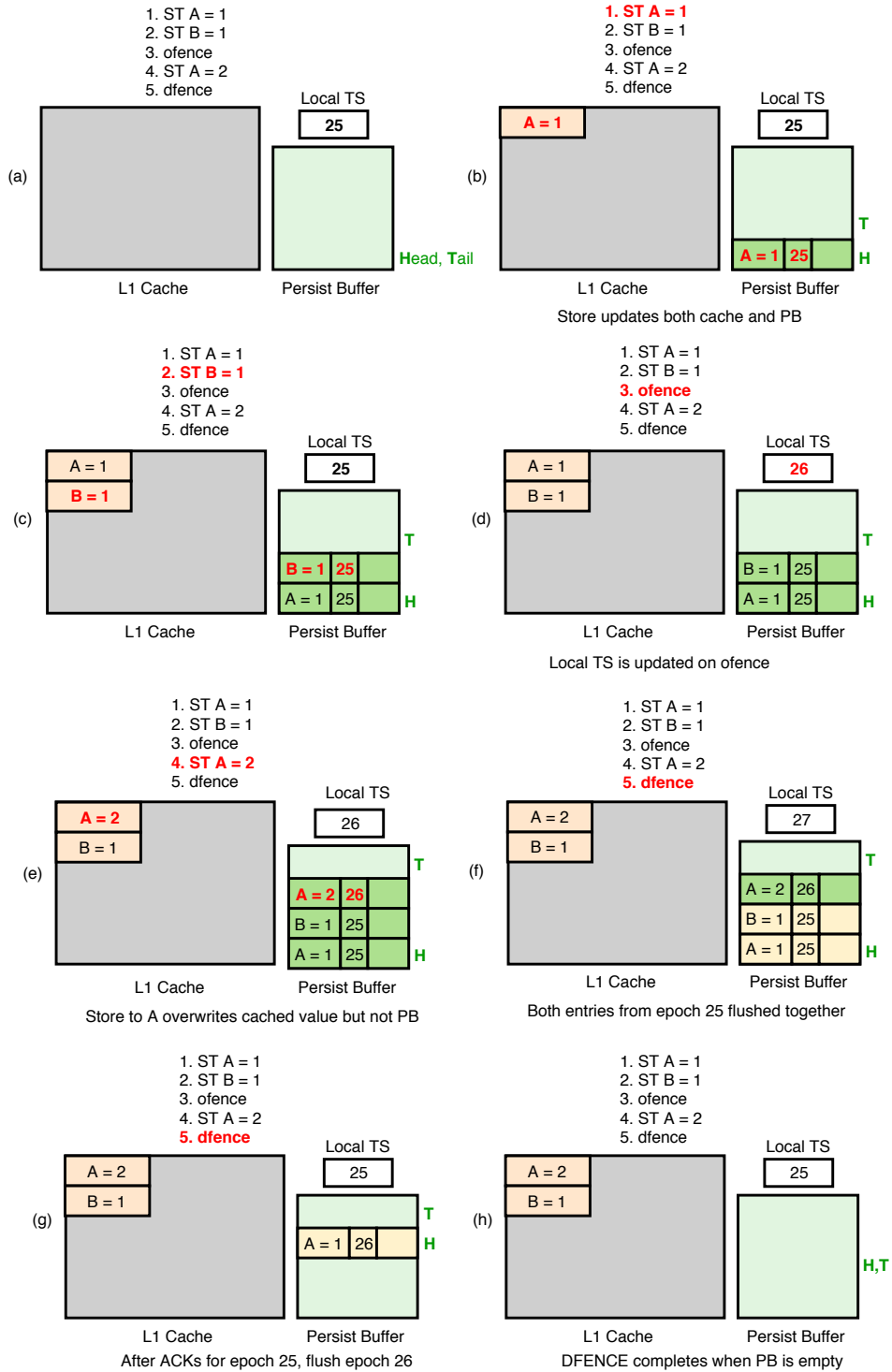


Figure 3.8: Example execution showing ofence and dfence implementation in HOPS. Persist Buffer entries in green are not yet flushed and in yellow have been flushed but flush ACKs are pending.

Events	Action
ofence	Increment Thread TS to end current epoch.
dfence	Increment Thread TS to end current epoch, and stall thread till local PB is flushed clean.
L1 read hit, miss	No change.
L1 write hit, miss	Get exclusive permissions (miss), update cache line and mark clean. Create PB entry with epoch TS = thread TS and dependency pointer (if any).
Forwarded GET	Respond with data and (if line cached exclusively) dependency pointer (thread ID, TS) to requestor.
LLC hit	No change.
LLC miss	Send request to MC, which stalls request if address present in any PB.

Table 3.3: Handling of major events in HOPS.

with a dependency tuple, marking the cross-thread dependency.

To handle the case when a L1 cache P writes back a dirty cacheline, we rely on the *sticky-M* state (similar to LogTM [148]). Accordingly, the lower-level cache receiving the dirty cacheline transitions into a new state "sticky-M@P". When another L1 cache Q requests this cacheline non-exclusively, the directory forwards the request to P in addition to responding with the cacheline. When P receives the forwarded request for an uncached cacheline, it responds to Q with an empty coherence message along with the dependency tuple. If an L1 cache R requests this cacheline exclusively, the same steps are followed as described above, but the directory transitions out of the sticky-M state into the conventional Modified state.

In Table 3.3, we summarize the actions taken in HOPS design for executing major events i.e., new primitives as well as coherence messages.

Qualitative Evaluation

We now evaluate the HOPS design in terms of the design goals laid out in Section 3.1.

Design Goal 1. HOPS tracks PM stores outside of the cache hierarchy in separate PBs. Cache modifications are restricted to a single bit per cacheline for identifying PM cachelines and sticky information in the directory, which does not affect cache access latencies. Thus,

HOPS preserves the performance of DRAM accesses, which comprise 96% of accesses in applications.

Design Goal 2. HOPS adds support for lightweight ordering points (i.e., `ofence`). The `ofence` instruction is implemented by incrementing the `LocalTS` register and always executes immediately. HOPS also introduces the separate durability fence (`dfence`) for use when durability guarantees are required.

Design Goal 3. HOPS tracks ordering information in per-thread PBs as cross-thread dependencies are rare. Cross-dependencies are handled correctly using global ordering information from the `GlobalTS` register.

Design Goal 4. Multiple stores from a thread to the same address can be buffered in a PB and are written back in TSO-BEP order. Self-dependencies do not incur any stalls or long-latency flushes, except in the rare instances when the PB is full.

3.5 Evaluation

Methodology.

For our evaluation, we used full-system mode in the `gem5` microarchitectural simulator [15]. The simulated system is a four-core (one hardware thread per core) 8-way out-of-order x86 processor with a two-level cache hierarchy and two memory controllers. Table 3.4 shows the relevant configuration parameters of the simulated system.

We extended the MOESI-hammer cache coherence protocol in `gem5` to add support for the `clwb` operation. We also modified the `sfence` implementation to stall the core until all outstanding `clwbs` are completed. We added support for optional persistent write-pending queues (PWQ) in the memory controller [63]. With PWQs, all memory requests buffered at the persistent memory controller are guaranteed to be durable in case of failure. Thus, flushes can be acknowledged early, when they reach the memory controller, instead of waiting for the long-latency write to an NVM device.

Finally, we implemented 32-entry PBs in gem5. To evaluate our implementation, we used a subset of applications from the WHISPER benchmark suite [94] and run them to completion. These applications were run on top of Linux v3.10 running in the simulator.

CPU Cores	4 cores, 8-way OOO, 2Ghz
CPU L1 Caches	private, 64 KB, Split I/D
CPU L2 Caches	private, 2 MB
Cache Policy	writeback, exclusive
Coherence	MOESI hammer protocol
DRAM	4GB, 40 cycles read/write latency
PM	4GB, 160 cycles read/write latency

Table 3.4: Configuration of Simulated System.

Compared Implementations.

We evaluated five distinct implementations:

- x86-64 (NVM): x86-64 implementation with `c1wb` and `sfence` primitives, with each `c1wb` acknowledged when the flush updates the NVM device. This implementation is our baseline.
- x86-64 (PWQ): x86-64 implementation, with each `c1wb` acknowledged when the flush is queued in the PWQ.
- HOPS (NVM): HOPS implementation with `ofence` and `dfence` primitives, with PB flushes acknowledged when the NVM device is updated.
- HOPS (PWQ): HOPS implementation, with PB flushes acknowledged when they are queued in the PWQ.
- Ideal (Non-CC): x86-64 implementation, but with applications never performing any flushes or fences. This implementation provides the upper bound on performance but is not crash-consistent.

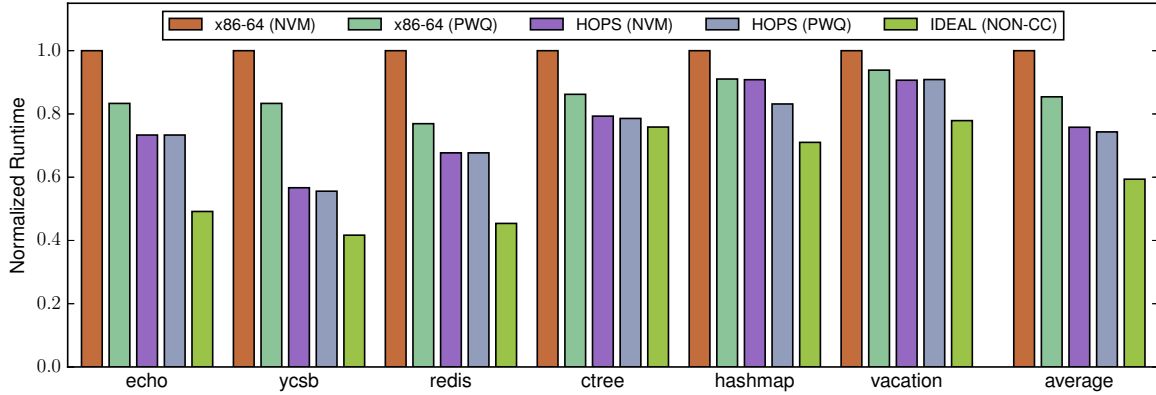


Figure 3.9: Performance of HOPS relative to x86-64 primitives, and an ideal but non crash-consistent implementation.

Performance Comparison

Figure 3.9 shows application runtimes (so smaller is better) normalized to the x86-64 (NVM) implementation.

We make the following observations:

- x86-64 (PWQ) lowers average runtimes by 15.5% compared to x86-64 (NVM), by reducing the cost of each ordering point due to lower flush latencies.
- HOPS (NVM) outperforms x86-64 (NVM) by 24.3% on average, by commonly using lightweight ofence instructions and only guaranteeing durability when needed (e.g., transaction commit).
- For the same reasons, HOPS (NVM) outperforms x86-64 (PWQ) by 10% on average.
- HOPS (PWQ) improves performance by 1.4% over HOPS (NVM) as the PWQ only lowers the cost of rare ofence operations.
- The Ideal (but unsafe) implementation outperforms the baseline (x86-64 NVM) by 40.7% and HOPS (NVM) by 19.7%.

Thus, we see that introducing the HOPS primitives offers a greater performance improvement than a PWQ. However, there is still an opportunity for new optimizations to bridge the performance gap with the ideal implementation.

3.6 Comparing Related Work with HOPS

In this section, we first discuss prior software and hardware proposals for PM. Then, we look at a proposal to decouple ordering and durability, similar to our approach, but aimed at filesystems. We discuss these here—rather than in Chapter 2—so that we can compare them with HOPS.

Software/Hardware support for PM.

There have been many hardware proposals facilitating fast PM accesses. BPFS [27] proposed augmenting caches with epoch ordering hardware to allow software control of the order of writebacks. Efficient Persist Barriers (EPB) [72] builds on this idea to provide lightweight epoch ordering and efficiently support inter-thread dependencies. Both proposals lack the durability needed for ACID transactions. Kiln [149] supports hardware transactions but without isolation guarantees. These proposals add substantial state to volatile caches proportional to the log of number of inflight epochs as well as number of cores, which can adversely affect the latency of all cache tag accesses, including those to volatile DRAM. In contrast, HOPS requires the addition of one bit per cacheline to identify cachelines corresponding to addresses in PM and sticky information at the directory controller. Moreover, Kiln requires a non-volatile last-level cache, which does not seem practical in the near future.

Most closely related is Delegated Persist Ordering (DPO), a concurrent proposal that shares with HOPS the development of persist buffers, similar in name but implemented differently [80]. Like HOPS, DPO optimizes for fast epoch boundaries by ordering epochs

without making their updates durable, handles inter-thread and intra-thread conflicts without explicit flushes and provides an express lane for persists. However, DPO does not make clear how applications ensure data is durable, e.g., for implementing ACID transactions. Additionally, DPO enforces Buffered Strict Persistency (BSP) with a relaxed memory consistency model (ARMv7). While DPO allows concurrent flushing of updates from the same epoch in systems with a single MC, it is unclear if BSP permits such concurrency in systems with multiple MCs. DPO's precise cross-dependency tracking mechanism requires that all incoming snoop requests, including common volatile accesses, snoop fully associative PBs. HOPS's epoch-granular dependency tracking eliminates this overhead at the cost of false positives. Unlike HOPS, DPO also requires a global broadcast on every flush of a buffered update from the PBs.

Techniques like *deferred commit* and *execute in log* have been proposed to optimize persistent transactions [79, 87]. Although these techniques consider an idealistic view of persistent transactions (e.g., all modified cachelines are known at the start of the transaction and four ordering points per transaction) that differs from our observations of real-world workloads, the proposed techniques can be used even for transactions implemented with *ofence* and *dfence*.

Loose-Ordering Consistency (LOC) [88] also proposes relaxing the ordering constraints of transactions. LOC introduces *eager commit* and *speculative persistence* to reduce intra-transaction and inter-transaction dependencies. In contrast, HOPS seeks to handle such dependencies efficiently.

ThyNVM [115] proposes hardware checkpointing for crash-consistency. Although transparent checkpointing removes the burden of modifying code to support persistent memory, it precludes the use of heterogeneous memory systems that include both volatile and persistent memory.

Ordering and durability. An analogous problem of conflated ordering and durability in file systems was solved by Optimistic Crash Consistency [24]. OCC introduces two

new primitives—osync and dsync—to improve file system performance while satisfying application-level consistency requirements. We follow a similar approach in this work.

3.7 Conclusion

Programming persistent memory (PM) applications is challenging as programmers must reason about crash consistency and use low-level programming models. To remove these constraints, we proposed the Hands-off Persistence System to efficiently order PM updates in hardware using persist buffers. HOPS provides high-level ISA primitives for applications to express durability and ordering constraints separately, to enable efficient transactions.

— 4 —

Minimally Ordered Durable Datastructures for Persistent Memory

The trouble about obeying orders is, it becomes a habit. And then everything depends on who's giving the orders.

TERRY PRATCHETT

Recoverable applications rely on consistent and durable updates to PM to prevent data loss or corruption in case of an ill-timed crash. For this purpose, programmers commonly rely on *failure-atomic sections* (FASEs) [20]. FASEs are code segments with the property that all PM updates within a FASE are guaranteed to happen durably and atomically with respect to failure i.e., either all updates or none survive in durable PM. PM libraries [26, 61, 139] typically implement FASEs via software transactional memory (PM-STM), offering failure-atomicity, consistency and durability.

Unfortunately, PM-STM implementations such as Intel's PMDK library [61] have frequent ordering points (5-50 per transaction [94]), which are extremely expensive on current hardware. Such frequent ordering often results in a single strongly ordered cacheline flush between ordering points, degrading application performance, particularly in small transactions. We measured the latency of a strongly ordered cacheline flush to be $4.2\times$ the

latency of DRAM accesses on real hardware—Intel Optane DCPMM. Consequently, we observed that PM workloads implemented with PMDK suffer from flushing overheads of up to $11\times$ over a baseline implementation without crash-consistency (Figure 4.1). In comparison, logging overheads from additional instructions and PM writes are much lower at about $0.6\times$ on average. In such situations, we can either seek to improve the performance of each ordering point or reduce the number of ordering points in PM applications.

In Chapter 3, we proposed the Hands-Off Persistence System to add architectural support for lightweight ordering points (*ofences*). HOPS decouples ordering from durability and eliminates long-latency cacheline flushes on ordering points. Unfortunately, the benefits of HOPS require hardware changes and thus HOPS is not beneficial in the short term.

On current x86-64 hardware, we can improve performance by reducing ordering points in an application and thus overlapping long-latency flushes to PM. x86-64 cacheline flushes are typically implemented as *posted* operations that do not wait for a response. However, ordering points (i.e., *sfence*) stall the CPU until all earlier flushes are acknowledged as completed. Our experiments on Optane DCPMM show that flushes (i.e., *clwb*) slow execution more when they are more frequently ordered. When ordered by a single *sfence*, 2 overlapped *clwbs* can be performed in $1.2 \times$ the latency of a single *clwb*, and 8 *clwbs* in $2 \times$ the latency. However, as overlapped flushes can update PM in any order, ordering points between flushes can only be eliminated if the flushes can be reordered safely i.e., without violating crash-consistency.

This thesis chapter proposes *minimally ordered durable* (MOD) datastructures that allow failure-atomic updates to be performed with **only one ordering point** in the common case on current, unmodified hardware. We design these datastructures using our proposed Functional Shadowing (FS) technique that uses side-effect free pure functions to implement non-destructive updates. On such updates, we create a new and updated copy (*shadow*) of the datastructure while preserving the original. We do not overwrite any PM data

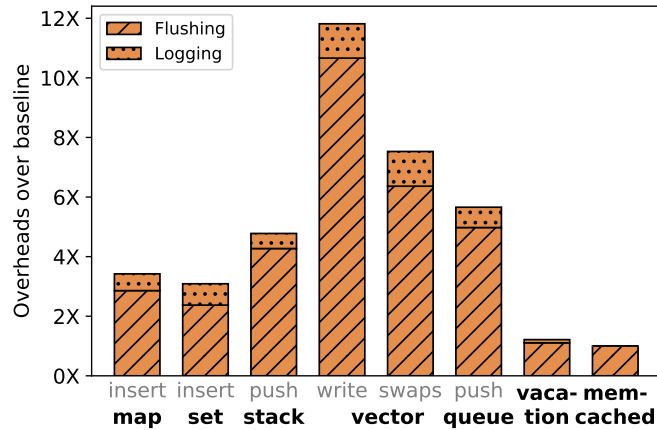


Figure 4.1: PM-STM overheads in recoverable PM workloads implemented using PMDK v1.5.

but instead perform out-of-place writes to the shadow datastructure that are not ordered with each other or logged. Thus, we can eliminate all ordering points between these non-destructive updates to minimize flushing overheads.

We present two programming interfaces for using MOD datastructures to build FASEs. First, we provide a simple, single-versioned interface that hides the complexities of functional shadowing and is sufficient for most programmers. This interface optimizes for the common case when each FASE contains a single update operation (e.g., set, push, insert) on one datastructure. Second, we offer a multi-versioned interface that makes it easy to program complex FASEs with multiple updates to multiple datastructures. This interface exposes multiple versions of datastructures to the programmers while still abstracting away other implementation details. Thus, we allow programmers to use MOD datastructures like conventional mutable datastructures while allowing expert programmers to peel back some of the layers of the implementation in exchange for advanced functionality.

To make it straightforward for programmers to create MOD datastructures, we provide a recipe to generate these datastructures from existing implementations of purely functional datastructures [101] (confusingly called persistent datastructures for reasons not related to durability or PM). Such *functional* datastructures inherently perform updates

non-destructively. Moreover, we leverage substantial research efforts towards performance and space optimizations in these datastructures [37, 110, 126, 130]. Particularly, these datastructures prioritize *structural sharing* to ensure that the updated shadow is mainly composed of the unmodified data of the original datastructure plus modest updated state. Consequently, the new version only incurs additional space overheads of less than 0.01% over the original datastructure.

We develop a C++ library of MOD datastructures with map, set, vector, queue and stack implementations. On systems with real PM—Intel Optane DCPMM, our MOD datastructures improve the performance of recoverable PM workloads by 60% as compared to Intel PMDK v1.5.

We make the following contributions in this chapter:

- We propose MOD datastructures to lower flushing overheads in PM applications.
- We present two interfaces for programming FASEs with MOD datastructures, both requiring a single ordering point per FASE in the common case.
- We provide a recipe to create MOD datastructures from existing functional datastructures.
- We develop an analytical model for estimating the latency of concurrent cacheline flushes based on observations from Optane DCPMM.
- We release a C++ library of MOD datastructures.

4.1 Background on Functional Programming

In this section, we provide basic knowledge of functional programming as required for this chapter. In this work, we leverage two basic concepts in functional programming languages: pure functions and purely functional datastructures. These ideas are briefly described below and illustrated in Figure 4.2. We provided the background information about PM in Chapter 2.

Pure Functions.

A pure function is one whose outputs are determined solely based on the input arguments and are returned explicitly. Pure functions have no externally visible effects (i.e., side effects) such as updates to any non-local variables or I/O activity. Hence, only data that is newly allocated within the pure function can be updated. Figure 4.2 shows how a pure and an impure function differ in performing a prepend operation to a list. The impure function overwrites the head pointer in the original list L, which is a non-local variable and thus results in a side effect. In contrast, the pure function allocates a new list shadowL to mimic the effect of the prepend operation on the original list and explicitly returns the new list. Note that the pure function does not copy the original list to create the new list. Instead, it reuses the nodes of the original list without modifying them in any manner.

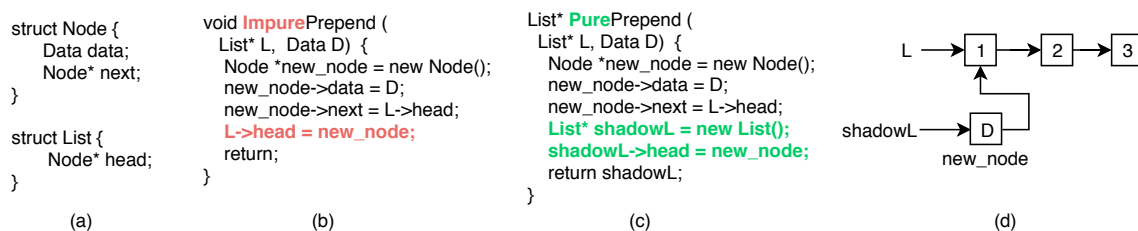


Figure 4.2: Implementing prepend operation for linked list (defined in (a)) as (b) impure function where original list L is modified and (c) pure function where a new updated list shadowL is created and returned; application uses the returned list as the latest updated list. (d) shadowL reuses nodes of original list L to reduce space overheads.

Functional Datastructures.

Commonly used in functional languages, purely functional or *persistent datastructures* are those that preserve previous versions of themselves when modified [37]. We refer to these as purely functional datastructures in this chapter to avoid confusion with persistent (i.e., durable) datastructures for PM.

Purely functional datastructures are never modified in-place. Instead, every update of a purely functional datastructure creates a logically new version while preserving the old

version. Thus, these datastructures are inherently multi-versioned.

To reduce space overheads and improve performance, most functional datastructures (even arrays and vectors) are often implemented as trees [101, 110]. Tree-based implementations allow different versions of a datastructure to appear logically different while sharing most of the internal nodes of the tree. For example, Figure 4.2 shows a simple example where the original list L and the updated list $shadowL$ share nodes labeled 1, 2 and 3. Such optimizations are called *structural sharing*.

4.2 Ordering & Flushing Overheads on Optane DCPMMs

Cacheline Flushes to PM—essential for durability in PM applications—are expensive due to the high write latency of non-volatile memory technologies. In this section, we first present the behavior of cacheline flushes on the Intel Optane DCPMM. Next, we empirically describe the benefits of reducing ordering points on flush latency and develop an analytical model to reason about flush concurrency. Lastly, we discuss the high flushing overheads in PM-STM implementations to motivate MOD datastructures.

Effects of Ordering Points

We can tolerate cacheline flush latencies by reducing ordering points and overlapping multiple weakly ordered flushes. We evaluated the efficacy of this approach on Optane DCPMMs via a simple microbenchmark. Our microbenchmark first allocates an array backed by PM. It issues writes to 320 random cachelines ($= 20\text{KB} < 32\text{KB}$ L1D cache) within the array to fault in physical pages and fetch these cachelines into the private L1D cache. Next, it measures the time taken to issue `clwb` instructions to each of these cachelines. Fence instructions are performed at regular intervals e.g., one `sfence` after every N `clwb` instructions. The total time (for 320 `clwb` + variable `sfence` instructions) is divided by 320 to get the average latency of a single cacheline flush. Figure 4.3 reports the average latency

per PM flush, for different amounts of flush concurrency.

The blue line in Figure 4.3 shows that the average flush latency can be effectively reduced by overlapping flushes, up to a limit. Compared to a single un-overlapped flush (`clwb +sfence`), performing 16 flushes concurrently reduces average flush latency by 75%. However, performing 32 flushes concurrently only reduces average flush latency by 3% compared to the case with 16 concurrent flushes. Beyond 32, there is no noticeable improvement in flush latency. Moreover, four flushes ordered by a single fence are 22% faster than two flushes ordered by one fence each.

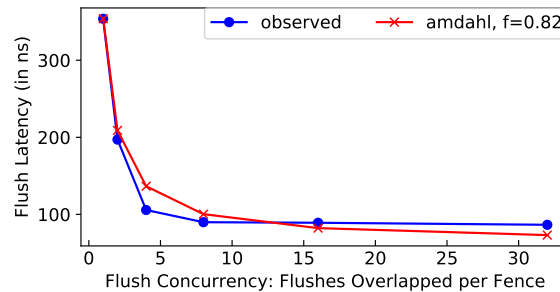


Figure 4.3: Average Latency of a PM cacheline flush on test machine with Optane DCPMM and compared to our analytical model based on Amdahl's law.

Using Amdahl's Law to Model Flush Latencies

With Optane DCPMMs, we empirically find that flush latency can be closely modeled using Amdahl's law [3] and that flushes seem to be 82% parallel and 18% serial.

We model weakly ordered cacheline flushes with a parallel fraction of work (f) as well as a serial fraction ($1 - f$). When multiple flushes are performed without ordering points, their parallel components are overlapped thereby reducing overall latency. However, their serial components must be performed sequentially i.e., one at a time. Accordingly, we can use Amdahl's law to calculate the reduction in flush latency by overlapping multiple flushes.

$$L_n = L_1 \cdot \left[(1 - f) + \frac{f}{n} \right]$$

With this equation, we can use non-overlapped flush latency (L_1) and the parallel fraction f to estimate L_n , the average latency per flush given n concurrent flushes.

The red line (model prediction) in Figure 4.3 shows a good fit with empirical blue line, with the fraction of parallel work estimated to be 0.82 using the Karp-Flatt metric [76]. This explains why increasing flush concurrency to 8 greatly improves performance, while further increases see diminishing returns. This use of Amdahl’s law motivates why our proposal seeks to reduce ordering points to enable more flushes in parallel.

We hypothesize that Amdahl’s law closely models flush concurrency due to parallelism arising from multiple banks at the memory controller and multiple virtual channels in the on-chip interconnect, and serialization from the FIFO queues in caches and memory controller. We expect that our model extends to other architectures that introduce weakly ordered flushes, albeit with different parallel fractions.

PM-STM Flush Overheads

We measured flush overheads in write-intensive recoverable PM workloads (Table 4.2) using Intel PMDK v1.5, a state-of-the-art PM-STM implementation. As shown in Figure 4.1 in the introduction, these applications suffer from flushing overheads of 1-11 \times over an implementation without crash-consistency (i.e., no logging or flushing). Note that an overhead of 1 \times doubles the execution runtime compared to the baseline. In fact, flushing overheads are the biggest performance bottlenecks in these applications as logging overheads are comparatively much smaller.

These high flushing overheads occur for two reasons. First, PM-STM implementations flush both log entries and data updates to persistent memory for durability. Second, these implementations offer limited potential for overlapping these flush instructions. Undo-logging techniques typically require 5-50 fences [94]. These fences mainly occur as log updates are ordered before the corresponding data updates. In some implementations [69, 61], the number of fences per transaction scale with the number of cachelines written. For

our workloads, we observed 3-21 flushes per transaction and 5-11 fences per transaction (Figure 4.11). Consequently, the median number of flushes overlapped per fence is 1-2, resulting in high flushing overheads.

4.3 Minimally Ordered Durable Datastructures

Minimally ordered durable (MOD) datastructures allow failure-atomic and durable updates to be performed with *one ordering point* in the common case. These datastructures significantly reduce flushing overheads that are the main bottleneck in recoverable PM applications. We have five goals for these datastructures:

1. *Failure-atomic updates* to support development of recoverable applications.
2. *Minimal ordering constraints* to tolerate flush latency and thus improve performance.
3. *Simple programming interface* that hides implementation details, while exposing additional functionalities for expert programmers.
4. *Simple recipe for creation* that can be extended to additional datastructures beyond the ones discussed in this chapter.
5. *Support for common datastructures* for application programmers such as maps and vectors from the C++ Standard Template Library [33].
6. *No hardware modifications* needed to enable high-performance applications on currently available systems.

We first introduce the *Functional Shadowing* technique used to design MOD datastructures. Then, we present a recipe for creating MOD datastructures out of existing purely functional datastructures. Then, we describe the programming interface of MOD datastructures and discuss their implementation.

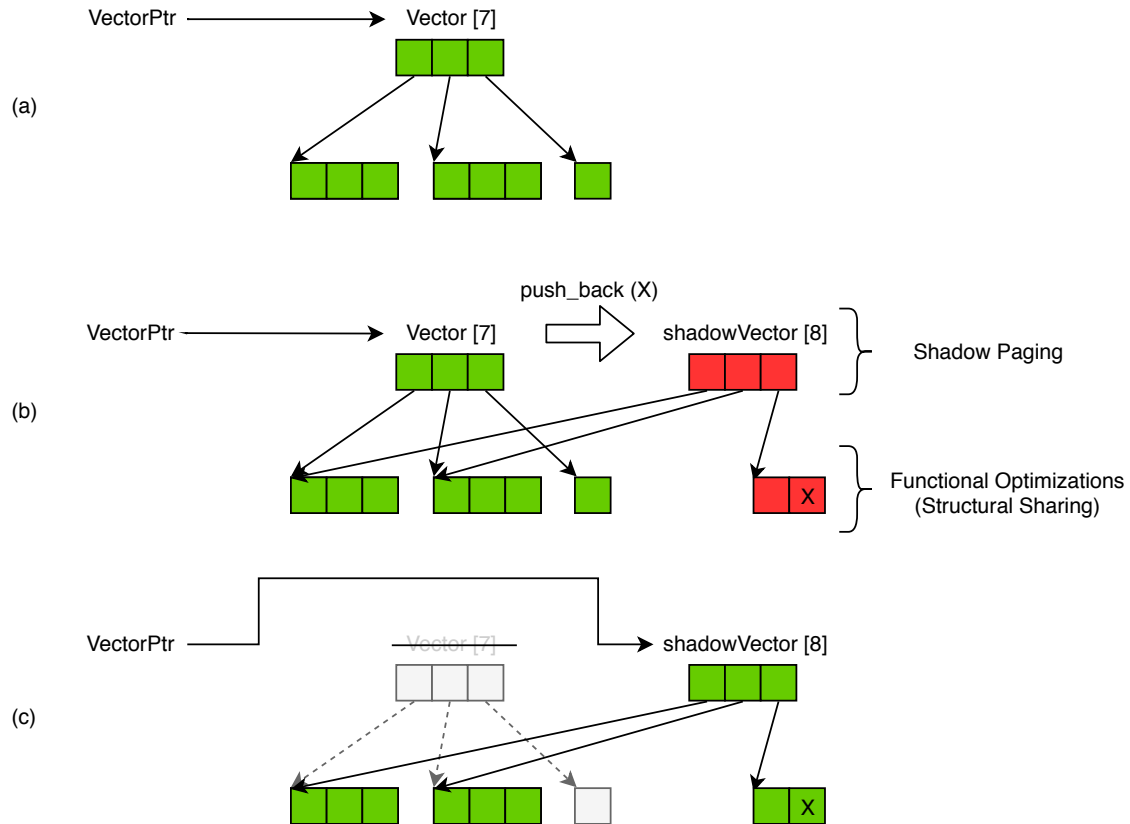


Figure 4.4: Functional Shadowing in action on (a) MOD vector. (b) Shadow is created on Append (i.e., `push_back`) operation that reuses data from the original vector. (c) Application starts using updated shadow, while old and unused data is cleaned up.

Functional Shadowing

Functional Shadowing leverages shadow paging techniques to minimize ordering constraints in updates to PM datastructures and uses optimizations from functional datastructures to reduce the overheads of shadow paging. As per shadow paging techniques, we implement non-destructive and out-of-place update operations for all MOD datastructures. Accordingly, updates of MOD datastructures logically return a new version of the datastructure without any modifications to the original data. As shown in Figure 4.4, a `push_back` operation in a vector of size 7 would result in a new version of size 8 while the original vector of size 7 remains untouched. We refer to the updated version of the datastructure as a *shadow* in accordance with conventional shadow paging techniques.

There are no ordering constraints in creating the updated shadow as it is not considered a necessary part of application state yet. We do not log these writes as they do not overwrite any data. In case of a crash at this point, recovery code will reclaim and free memory corresponding to any partially or complete shadow in PM as discussed in the next section. Due to the absence of ordering constraints, we can overlap flushes to all dirty cachelines comprising the updated shadow to minimize flushing overheads. A single ordering point is sufficient to ensure the completion of all the outstanding flushes and guarantee the durability of the shadow. Subsequently, the application must atomically replace the original datastructure with the updated shadow. For this purpose, we offer multiple efficient Commit functions described in the next subsection. In contrast, PM-STM implementations perform in-place modifications which overwrite existing data and need logging to revert partial updates in case of crashes. In-place updates also introduce ordering constraints as log writes must be ordered before the corresponding data update.

We reduce shadow paging overheads using optimizations commonly found in functional datastructures. Conventional shadow paging techniques incur high overheads as the original data must be copied completely to create the shadow. Instead, we use *structural sharing* optimizations to maximize data reuse between the original datastructure and its shadow copy. We illustrate this in Figure 4.4, where shadowVector reuses 6/8 internal nodes from the original Vector and only adds 2 internal and 3 top-level nodes. In the next subsection, we discuss a method to convert existing implementations of functional datastructures to MOD datastructures.

Generating MOD Datastructures from Functional Datastructures

While functional datastructures do not support durability by default, they have some desirable properties that make them an ideal starting point from which to generate MOD datastructures. They support non-destructive update operations which are typically implemented through pure functions. Thus, every update returns a new updated version

(i.e., shadow) of the functional datastructure without modifying the original. They export simple interfaces such as `map`, `vector`, etc. that are implemented internally as highly optimized trees such as Compressed Hash-Array Mapped Prefix-trees [127] (for `map`, `set`) or Relaxed Radix Balanced Trees [130] (for `vector`). These implementations are designed to amortize the overheads of data copying as needed to create new versions on updates.

Optimized functional implementations also have low space overheads via structural sharing, i.e., maximizing data reuse between the original data and the shadow. Tree-based implementations are particularly amenable to structural sharing. On an update, the new version creates new nodes at the upper levels of the tree, but these nodes can point to (and thus reuse) large sub-trees of unmodified nodes from the original datastructure. The number of new nodes created grows extremely slowly with the size of the datastructures, resulting in low overheads for large datastructures. As we show in our evaluation section, the additional memory required on average for an updated shadow is less than 0.01% of the memory of the original datastructure of size 1 million elements.

Moreover, the trees are broad but not deep to avoid the problem of ‘bubbling-up of writes’ [27] that plagues conventional shadow paging techniques. This problem arises as the update of an internal node in the tree requires an update of its parent and so on all the way to the root. We find that existing implementations of such low-overhead functional datastructures are commonly available in several languages, including C++ and Java.

We provide a simple recipe for creating MOD datastructures out of existing implementations of purely functional datastructures:

1. First, we use an off-the-shelf persistent memory allocator `nvm_malloc` [9] to allocate datastructure state in PM.
2. Next, we ensure that internal state of the datastructure is allocated on the persistent heap as opposed to the volatile stack.
3. Finally, we extend all update operations to flush all modified PM cachelines with

`clwb` instructions and no ordering points. These cachelines correspond to persistent memory allocated within the update function. These flushes will be ordered by an ordering point in a Commit step described later in this section.

We believe that the ability to create MOD datastructures from existing functional datastructures is important for three reasons. First, we benefit from significant research efforts towards lowering space overheads and improving performance of these datastructures [37, 101, 110, 126, 130]. Secondly, programmers can easily create MOD implementations of additional datastructures beyond those in this chapter by using our recipe to port other functional datastructures. Finally, we forecast that this approach can help extend PM software beyond C and C++ to Python, JavaScript and Rust, which have implementations of functional datastructures.

Programming Interface for MOD Datastructures

To abstract away the details of Functional Shadowing from application programmers, we provide two alternative interfaces for MOD datastructures:

- A simple, **single-versioned** interface that abstracts away the internal versioning and is sufficient for most application programmers.
- A **multi-versioned** interface that exposes multiple versions of datastructures to enable complex usecases, while still hiding the complexities of the efficient implementation.

Single-versioned Interface

The single-versioned interface to MOD datastructures (Figure 4.5a) allows programmers to perform individual failure-atomic update operations to a single datastructure. With this interface, MOD datastructures appear as mutable datastructures with logically in-place updates. Programmers use pointers to datastructures (e.g., `ds1` in Figure 4.5a), as is common

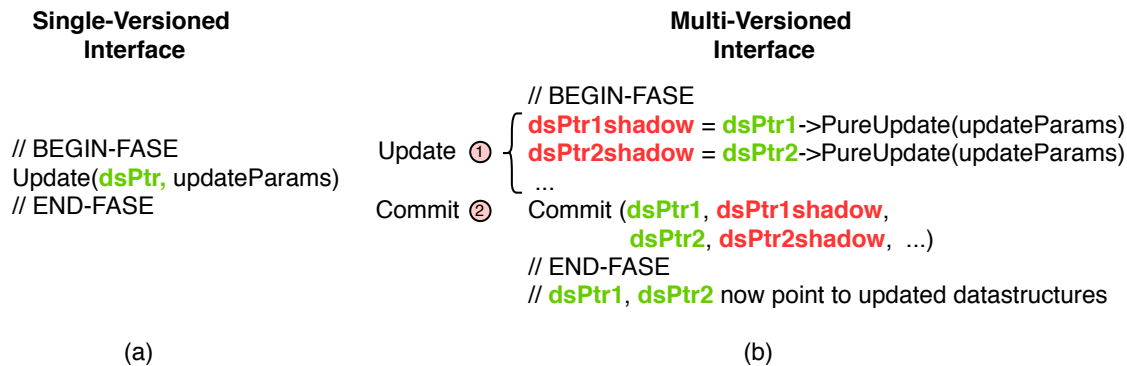


Figure 4.5: Failure-Atomic Code Sections (FASEs) with MOD datastructures using (a) single-versioned interface to update one datastructure and (b) multi-versioned interface to atomically update multiple datastructures, `dsPtr1` and `dsPtr2`. Datastructures in red are temporary shadow datastructures.

in PM programming. Each update operation is implemented as a self-contained FASE with one ordering point, as described later in this section. If the update completes successfully, the datastructure pointer points to an updated and durable datastructure. In case of crash before the update completes, the datastructure pointer points to the original durable and uncorrupted datastructure. We expose common update operations for datastructures such as `push_back`, `update` for vectors, `set` for sets/maps, `push`, `pop` for stacks and `enqueue`, `dequeue` for queues, as in the C++ Standard Template Library [33].

We provide an example of a single-versioned MOD vector performing a failure-atomic append operation in Figure 4.6. From the programmer’s point of view, when the FASE executes, an `X` is appended to the recoverable vector pointed to by `VectorPtr`, increasing its size from 7 to 8.

```
// VectorPtr points to Vector[7]
// BEGIN-FASE

push_back(VectorPtr, X)

// END-FASE
// VectorPtr points to Vector[8]
```

Figure 4.6: Failure-atomic append (i.e., `push_back`) on single-versioned MOD vector.

The single-versioned interface targets the common case when a FASE contains only one

update operation on one datastructure. This common case applies to all our workloads except `vacation` and `vector-swaps`. For instance, `memcached` relies on a single recoverable map to implement its cache and FASEs involve a single set operation.

Multi-versioned Interface

The multi-versioned interface to MOD datastructures (Figure 4.5b) is a general-purpose transaction-like programming interface. It allows programmers to failure-atomically perform updates on multiple datastructures or perform multiple updates to the same datastructure or any combination thereof. For instance, moving an element from one queue to another requires a pop operation on the first queue and a push operation on the second queue, both performed failure-atomically in one FASE. Complex operations such as swapping two elements in a vector also require two update operations on the same vector to be performed failure-atomically. In such cases, the multi-versioned interface allows programmers to perform individual non-destructive update operations on multiple datastructures to get new versions, and then atomically replace all the updated datastructures with their updated versions in a single *Commit* operation.

With this interface, programmers can build complex FASEs, each with multiple update operations on multiple datastructures. Each FASE must consist of two parts: Update and Commit. During Update, programmers perform updates on one or more MOD datastructures. On an update operation, the original datastructure is preserved and a new updated version is returned that is guaranteed to be durable only after Commit. Thus, programmers are temporarily exposed to multiple versions of datastructures. Programmers use the Commit function to atomically replace all the original datastructures with their latest updated and durable versions. Our Commit implementation (described later in this section) contains a single ordering point in the common case. We use this interface in two workloads: `vector-swaps` and `vacation`.

Figure 4.7 demonstrates different usecases of multi-versioned MOD datastructures:

- **Single Update of Single Datastructure:** While this case is best handled by the single-versioned interface, we repeat it here to show how this can be achieved with the multi-versioned interface. In Figure 4.7a, appending an element to `VectorPtr` results in an updated version (`VectorPtrShadow`). The Commit step atomically modifies `VectorPtr` to point to `VectorPtrShadow`. As a result of this FASE, a new element is failure-atomically appended to `VectorPtr`.
- **Multiple Update of Single Datastructure:** We show a FASE that swaps two elements of a vector in Figure 4.7b. The Update step involves two vector lookups and two vector updates. The first vector update results in a new version `VectorPtrShadow`. The second vector update is performed on the new version to get another version (`VectorPtrShadowShadow`) that reflects the effects of both updates. Finally, Commit makes `VectorPtr` point to the latest version with the swapped values.
- **Single Updates of Multiple Datastructures:** Figure 4.7c shows how we swap elements from two different vectors in one FASE. For each vector, we perform the update operation to get a new version. In Commit, both vector pointers are atomically updated to point to the respective new versions.
- **Multiple Updates of Multiple Datastructures:** The general case is achieved by a combination of the previous two usecases.

Implementing Interfaces to MOD Datastructures

We now discuss the efficient implementation of the two interfaces to MOD datastructures to enable FASEs with **one ordering point** in the common case.

Single-Versioned Interface

As shown in Figure 4.8, the single-versioned interface is a wrapper around the multi-versioned interface to create the illusion of a mutable datastructure. The programmer

```

// BEGIN-FASE (Vector-Append)
Update ① VectorPtrShadow = VectorPtr->push_back(X)
Commit ② CommitSingle (VectorPtr, VectorPtrShadow)
// END-FASE
// VectorPtr now points to updated vector
(a)

// BEGIN-FASE (Vector-Swap)
val1 = (*VectorPtr)[index1]
val2 = (*VectorPtr)[index2]
① { VectorPtrShadow = VectorPtr->update(index1, val2)
    VectorPtrShadowShadow =
      VectorPtrShadow->update(index2, val1)
    }
② CommitSingle (VectorPtr, VectorPtrShadow,
               VectorPtrShadowShadow)
// END-FASE
// VectorPtr now points to doubly-updated vector
(b)

// BEGIN-FASE (Multi-Vector-Swap)
① { val1 = (*VectorPtr1)[index1]
    val2 = (*VectorPtr2)[index2]
    VectorPtr1Shadow = VectorPtr1->update(index1, val2)
    VectorPtr2Shadow = VectorPtr2->update(index2, val1)
    }
② CommitUnrelated (VectorPtr1, VectorPtr1Shadow,
                  VectorPtr2, VectorPtr2Shadow)
// END-FASE
// VectorPtr1, VectorPtr2 now point to updated vectors
(c)

```

Figure 4.7: Using multi-versioned MOD datastructures for failure-atomically (a) appending an element to a vector, (a) swapping two elements of a vector and (c) swapping two elements of two different vectors.

accesses the MOD datastructure indirectly, via a pointer. On a failure-atomic update, we use the Update step to internally create an updated shadow of the datastructure by performing the non-destructive update. Then, using Commit, we ensure the durability of the shadow and atomically update the datastructure pointer to point to the updated and durable shadow. Thus, the implementation hides all details of functional shadowing from the programmer.

```

Update(dsPtr, updateParams) {
  // BEGIN-FASE
Update ① dsPtr = dsPtr->PureUpdate(updateParams)
Commit ② Commit (dsPtr, dsPtrshadow)
// END-FASE
// dsPtr now point to updated datastructure
}

```

Figure 4.8: Implementation of single-versioned interface as a wrapper around the multi-versioned interface.

Multi-Versioned Interface

Multi-versioned MOD datastructures allow programmers to build complex multi-update FASEs, each with one ordering point in the common case. However, this interface exposes the multiple versions of the datastructure to the programmer while still hiding the details of structural sharing. As described earlier, FASEs for multi-versioned MOD datastructures consist of Update and Commit.

In the Update step, the application programmer uses different update operations as required by the logic of the application. Each MOD datastructure supports non-destructive update operations. Within these update operations, all modified cachelines are flushed using (weakly ordered) `clwb` instructions and there are no ordering points or fences. However, this step results in multiple versions of the updated MOD datastructures.

The Commit step ensures the durability of the updated versions and failure-atomically updates all relevant datastructure pointers to point to the latest version of each datastructure. We provide optimized implementations of `Commit` for two common cases as well as the general implementation, as shown in Figure 4.9. We discuss the memory reclamation needed to free up unused memory in Section 4.4.

The first common case (`CommitSingle` in Figure 4.9a) occurs when only one datastructure is updated one or multiple times in one FASE (e.g., Figure 4.7a,b). To `Commit`, we update the original datastructure pointer to point to latest version after all updates, with a single 8B (i.e., size of a pointer) atomic write. We need to reclaim the memory of the old datastructure and intermediate shadow versions i.e., all but the latest shadow version.

The second common case (`CommitSiblings` in Figure 4.9b) occurs when the application updates two or more MOD datastructures that are pointed to by a common persistent object (*parent*) in one FASE. In this case, we create a new instance of the parent (`parentShadow`) that points to the updated shadows of the MOD datastructures. Then, we use a single pointer write to replace the old parent itself with its updated version. We used this approach in porting `vacation`, wherein a `manager` object has three separate recoverable maps as its

member variables. A commonly occurring parent object in PM applications is the root pointer, one for each persistent heap, that points to all recoverable datastructures in the heap. Such root pointers allow PM applications to locate recoverable datastructures in persistent heaps across process lifetimes.

In these two common cases, our approach requires **only one ordering point per FASE**. The single ordering point is required in the commit operation to guarantee the durability of the shadow before we replace the original data. The entire FASE is a single epoch per the epoch persistency model [105]. Both of the common cases require an atomic write to a single pointer, which can be performed via an 8-byte atomic write. In contrast, PM-STM implementations require 5-50 ordering points per FASE [94].

For the general and uncommon case (CommitUnrelated in Figure 4.9c) where two unrelated datastructures get updated in the same FASE, we need to atomically update two or more pointers. For this purpose, we use a very short transaction (STM) to atomically update the multiple pointers, albeit with more ordering constraints. Even in this approach, the majority of the flushes are performed concurrently and efficiently as part of the non-destructive updates. Only the flushes to update the persistent pointers in the Commit transaction cannot be overlapped due to PM-STM ordering constraints.

Thus, multi-versioned datastructures enable complex FASEs with efficient implementations for the two common cases.

Correctness

We provide a simple and intuitive argument for correct failure-atomicity of MOD datastructures. The main correctness condition is that there must not be any pointer from persistent data to any unflushed or partially flushed data. MOD datastructures support non-destructive updates that involve writes only to newly allocated data and so there is no possibility of any partial writes corrupting the datastructure. All writes performed to the new version of the datastructure are flushed to PM for durability. During Commit, one

<p>(a)</p> <p>CommitSingle</p> <p>(ds, dsShadow, ..., dsShadowN)</p> <p>FENCE</p> <p>dsOld = ds</p> <p>ds = dsShadowN</p> <p>Reclaim (dsOld, dsShadow, ...)</p>	<p>(b)</p> <p>CommitSiblings</p> <p>(parent, ds1, ds1Shadow, ds2, ds2Shadow, ...)</p> <p>parentShadow = new Parent</p> <p>parentShadow->ds1 = ds1shadow</p> <p>parentShadow->ds2 = ds2shadow</p> <p>...</p> <p>FLUSH parentShadow</p> <p>FENCE</p> <p>parentOld = parent</p> <p>parent = parentShadow</p> <p>Reclaim (parentOld)</p>	<p>(c)</p> <p>CommitUnrelated</p> <p>(ds1, dsShadow, ds2, ds2Shadow, ...)</p> <p>ds1Old = ds1</p> <p>ds2Old = ds2</p> <p>...</p> <p>FENCE</p> <p>Begin-TX {</p> <p> ds1 = ds1Shadow</p> <p> ds2 = ds2Shadow</p> <p>} End-TX</p> <p>Reclaim (ds1Old, ds2Old, ..)</p>
--	--	---

Figure 4.9: Commit implementation shown for multi-update FASEs operating on (a) single datastructure, (b) multiple datastructures pointed to by common *parent* object, and (c) (uncommon) multiple unrelated datastructures.

fence orders the pointer writes after all flushes are completed i.e., all updates are made durable. Finally, the pointer writes in Commit are performed atomically. If there is a crash within a FASE before the atomic pointer writes in Commit, the persistent pointers point to the consistent and durable original version of the datastructure. If the atomic pointer writes complete successfully, the persistent pointers points to the durable and consistent new version of the datastructures. Thus, we support correct failure-atomic updates of MOD datastructures.

4.4 Implementation Details

Having described the design of MOD datastructures, we now discuss how our implementation of MOD datastructures tackles common challenges with recoverable datastructures.

Memory Reclamation

Leaks of persistent memory cannot be fixed by restarting a program and thus, are more harmful than leaks of volatile memory. Such PM leaks can occur on crashes during the execution of a FASE. Specifically, allocations from an incomplete FASE leak PM data that must be reclaimed by recovery code. Additionally, our MOD datastructures must also

reclaim data belonging to the old version of the datastructure on completion of a successful FASE.

We use reference counting for memory reclamation. Our MOD datastructures are implemented as trees. In these trees, each internal node maintains a count of other nodes that point to it i.e., parent nodes. We increment reference counts of nodes that are reused on an update operation and decrement counts for nodes whose parents are deleted on a delete operation. Finally, we deallocate a node when its reference count hits 0.

Our key optimization here is to recognize that reference counts do not need to be stored durably in PM. On a crash, all reference counts in the latest version can be scanned and set to 1 as the recovered application sees only one consistent and durable version of each datastructure.

We rely on garbage collection during recovery to clean up allocated memory from an incomplete FASE (on a crash). As all of our datastructures are implemented as trees, we can perform a reachability analysis starting from the root node of each MOD datastructure to mark all memory currently referenced by the application. Any unmarked data remaining in the persistent heap is a PM leak and can be reclaimed at this point. A conventional solution for catching memory leaks is to log memory allocator activity. However, this approach reintroduces ordering constraints and degrades the performance of all FASEs in order to prevent memory leaks in case of a rare crash.

Automated Testing

While it is tricky to test the correctness of recoverable datastructures, the relaxed ordering constraints of shadow updates allow us to build a simple and automated testing framework for our MOD datastructures. We generate a trace of all PM allocations, writes, flushes, commits, and fences during program execution. Subsequently, our testing script scans the trace to ensure that all PM writes (except those in commit) are only to newly allocated PM and that all PM writes are followed by a corresponding flush before the next fence. By

verifying these two invariants, we can test the correctness of recoverable applications as per our correctness argument in Section 4.3.

Proposed Optimizations

There are two general optimizations that can lead to further improvements in MOD datastructures that we have not implemented.

Offline Memory Reclamation. If CPU resources are available, we can move memory reclamation off the critical path and onto a separate dedicated thread. During `Commit`, the application adds a reference to the old datastructure to a per-thread reclamation queue instead of actively reclaiming the memory. This lowers latency by eliminating the cost of decrementing reference counts and freeing memory from the main thread, but at the cost of throughput due to coordination overhead. This optimization is similar to offline log truncation done in some PM-STM implementations [55, 139].

In-place Shadow Updates. We can improve performance by performing some updates in-place without increasing ordering constraints. When a MOD datastructure is modified more than once in a FASE, we can perform in-place updates for all modifications after the first one. The first update creates a shadow copy that is not linked into the application state, and thus further modifications can be done in-place without any ordering or logging. For example, in `vector-swaps` (Figure 4.7b), we can perform the second update in-place without incurring any shadow copying overheads.

4.5 Extensions for Concurrency

We currently focus on efficient persistence of MOD datastructures and not concurrency. Nonetheless, we have some ideas for enabling concurrent accesses to MOD datastructures.

A simple solution is to make use of one reader-writer lock per datastructure to allow concurrent read accesses and exclusive write access. As we use locks to ensure isolation

but not atomicity, we do not need to log the acquire and release operations of these locks as performed in Atlas [20]. Instead, it is sufficient to restore the locks to an unlocked and initialized state on recovery.

The out-of-place updates performed by MOD datastructures make them conducive for read-copy-update (RCU) based concurrency [48]. The RCU mechanism is best suited for read-mostly datastructures wherein writers perform out-of-place updates and readers can read stale versions of the datastructure. Under such conditions, RCU allows concurrent execution of readers and writers with minimal overheads for readers. However, one challenge is to identify when no readers are accessing older versions of datastructures so that the old versions can be safely reclaimed.

Finally, it might be interesting to see if MOD datastructures can be made into lock-free datastructures [41, 53, 90], which they already have similarities to. For instance, both MOD and lock-free datastructures perform out-of-place updates which are committed using a single, atomic pointer update. However, we leave such extensions for future work.

4.6 Evaluation

The goal of Functional Shadowing is to improve performance by minimizing ordering constraints. As such, we answer three questions in our evaluation:

1. **Performance:** Does FS improve the performance of our recoverable workloads compared to PM-STM?
2. **Ordering Constraints:** Do workloads with FS show fewer ordering constraints than with PM-STM?
3. **Additional Overheads:** What are the additional overheads introduced by our FS approach?

Methodology

Test System Configuration. We ran our experiments on a machine with actual Persistent Memory—Intel Optane DCPMM [67]—and upcoming second-generation Xeon Scalable processors (codenamed Cascade Lake). We configured our test machine such that Optane DCPMM is in 100% App Direct mode [56] and uses the default Directory protocol. In this mode, software has direct byte-addressable access to the Optane DCPMM. Table 4.1 reports relevant details of our test machine. We measured read latencies using Intel Memory Latency Checker v3.6 [137].

CPU	
Type	Intel Cascade Lake
Cores	96 cores across 2 sockets
Frequency	1 GHz (Turbo Boost to 3.7 GHz)
Caches	L1: 32KB Icache, 32KB Dcache L2: 1MB, L3: 33 MB (shared)
Memory System	
PM Capacity	2.9 TB (256 GB/DIMM)
PM Read Latency	302 ns (Random 8-byte read)
DRAM Capacity	376 GB
DRAM Read Latency	80 ns (Random 8-byte read)

Table 4.1: Test Machine Configuration.

Hardware Primitives. The Cascade Lake processors on our test machine support the new `c1wb` instruction for flushing cachelines. The `c1wb` instruction flushes a dirty cacheline by writing back its data but may not evict it. Our workloads use `c1wb` instructions for flushing cachelines and the `sfence` instructions to order flushes.

OS interface to PM. Our test machine runs Linux v4.15.6. The DCPMMs are exposed to user-space applications via the DAX-filesystem interface [142]. Accordingly, we created an `ext4-dax` filesystem on each PM DIMM. Our PM allocators create files in these filesystems to back persistent heaps. We map these PM-resident files into application memory with flags `MAP_SHARED_VALIDATE` and `MAP_SYNC` [29] to allow direct user-space access to PM.

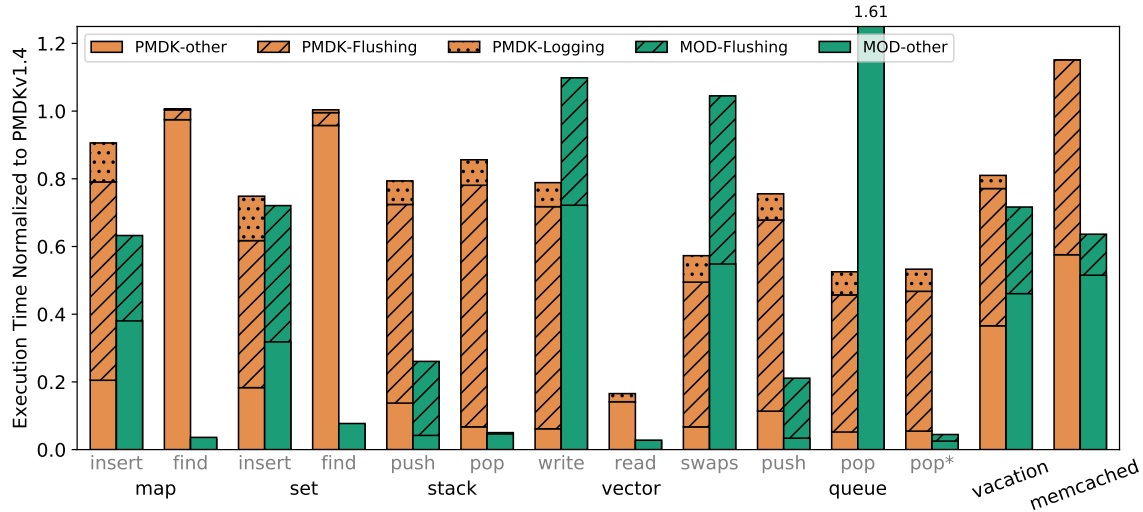


Figure 4.10: Execution Time of PM workloads, normalized to PMDK v1.4 implementation of each workload. Queue-pop* is queue-pop with the first pop operation being untimed.

PM-STM Implementation. We use the PM-STM implementation (*libpmemobj*) from Intel’s PMDK library [61] in our evaluations. We choose PMDK as it is publicly available, regularly updated, Intel-supported and hopefully optimized for Intel’s PM hardware. Moreover, PMDK (v1.4 or earlier) has been used for comparison by most recent PM studies [30, 84, 85, 122, 123]. We evaluate both PMDK v1.5 (released October 2018), which uses hybrid undo-redo logging techniques as well as PMDK v1.4, which primarily relies on undo-logging.

Workloads. Our workloads include several microbenchmarks as well as two recoverable applications. As described in Table 4.2, our microbenchmarks involve operations on commonly used datastructures:(hash)map, set, queue, list and vector. The vector-swaps workload emulates the main computation in the canneal benchmark from the PARSEC suite [14]. The baseline map datastructure can be implemented by either hashmap or ctree from the WHISPER suite [94]. Here, we compare against hashmap which outperformed the ctree implementation on Optane DCPMM. Moreover, we also measured two recoverable applications from the WHISPER suite: memcached and vacation. We modified these applications to use the PMDK and MOD map implementations. The only other PM-STM application in WHISPER is redis, but it also uses a map datastructure so we found it

Benchmark	Description	Configuration
map-insert*	Insert random keys with constant value in map	Key Size: 8 bytes, Value Size: 32 bytes
map-find	Lookup random keys in prepopulated map with 1M entries	Key Size: 8 bytes, Value Size: 32 bytes
set-insert*	Insert random keys in set	Key Size: 8 bytes, Value Size: 32 bytes
set-find	Lookup random keys in prepopulated set with 1M entries	Key Size: 8 bytes, Value Size: 32 bytes
stack-push*	Insert elements at the front of a stack	Element size: 8 bytes
stack-pop	Remove elements from front of stack with 1M elements	Element size: 8 bytes
vector-write*	Update elements at random indices in vector	Vector Size: 1M entries, Element Size: 8 bytes
vector-read	Lookup elements at random indices in vector	Vector Size: 1M entries, Element Size: 8 bytes
vector-swaps*	Swap two elements at random indices in recoverable vector	Vector Size: 1M entries, Element Size: 8 bytes
queue-push*	Insert elements at the back of a queue	Element size: 8 bytes
queue-pop	Remove elements from front of queue with 1M elements	Element size: 8 bytes
vacation*	Travel reservation system with four recoverable maps	Prepopulated entries:1M 55% user reservations, Query Range:80%
memcached*	In-memory key value store using one recoverable map	95% sets, 5% gets, key size:16 bytes, value size:512 bytes

Table 4.2: Benchmarks developed and used for this study. Each workload performs 1 million iterations of the operations described. We consider workloads marked with asterisk* to be write-intensive.

redundant for our purposes. The other WHISPER benchmarks are not applicable for our evaluation as they are either filesystem-based or do not use PM-STM. We ran all workloads to completion on our test machine with Optane DCPMMs.

Performance

Figure 4.10 shows the execution time (so smaller is better) of PM workloads with PMDK v1.5 transactions and MOD datastructures, normalized to the runtime of the same workload with PMDK v1.4 transactions (used in prior works). We make the following observations:

1. The new version (1.5) of PMDK library performs 30% better on average (geometric mean) than version 1.4, due to performance optimizations targeting transaction overheads [60].
2. MOD datastructures offer a speedup of 73% on average over PMDK v1.4 and 60% over PMDK v1.5. For write-only workloads, the performance improvement can be mainly attributed to having only one ordering point per operation. Consequently, average flushing overheads in our write-intensive workloads decrease from 65% of the overall runtime with PMDK v1.5 to 45% with MOD. For read-only workloads,

the speedup mainly comes from optimizations present in the original functional datastructures.

3. MOD workloads spent 43% of the execution time on average on *other* activity as opposed to 20% for PMDK v1.5. The differences arise due to additional overheads in the FS approach, such as greater cache pressure and memory allocation. We analyze these factors later in this section.
4. For 3 workloads, MOD datastructures perform worse than both PMDK versions. For `vector-write` and `vector-swaps`, MOD datastructures require greater flushes (12.5 and 14 vs 4 and 7.5 respectively) than PMDK v1.5. For `queue-pop`, performing the first pop operation involves an expensive recursive operation to reverse a large list. `queue-pop*` is the same workload with the first pop operation performed outside the timed loop and thus shows much lower overheads.

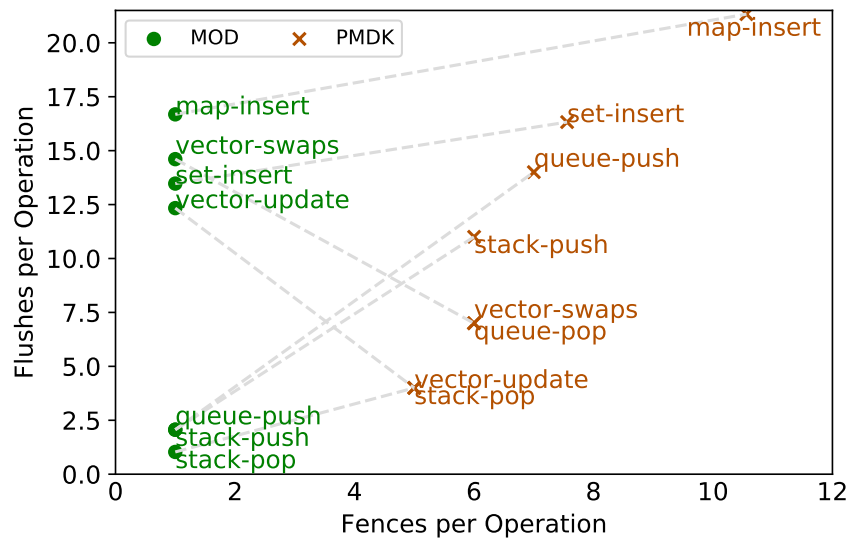


Figure 4.11: Flushing and ordering behavior of PM workloads. `queue-pop` (not shown) has 1 fence and 104 flushes per operation due to pathological case described earlier.

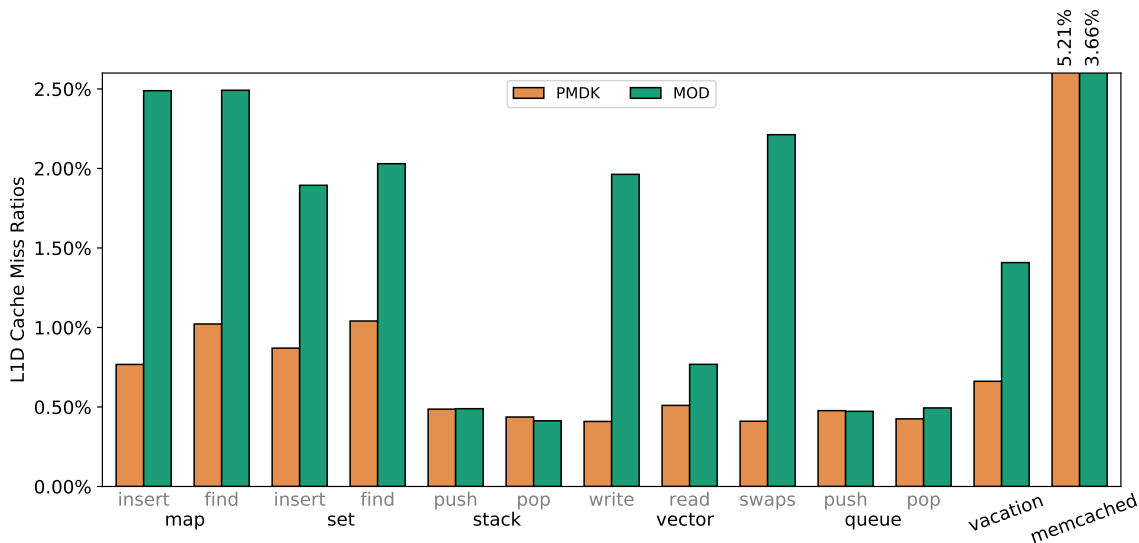


Figure 4.12: L1D Cache miss ratios for our workloads.

Flushing Concurrency

Figure 4.11 illustrates the flushing and ordering behavior of PM workloads with PMDK v1.5 and MOD datastructures. For improving performance, fewer fences per datastructure operation (i.e., push, pop, set) are most important, followed by fewer flushes per operation. Workloads using PMDK datastructures typically exhibit 5-11 fences per operation. Consequently, there is less opportunity to overlap long-latency flushes to PM. In contrast, MOD datastructures always have only one ordering point, allowing all long-latency flushes to be overlapped. Furthermore, PMDK datastructures require two flushes per PM update, one each for the data and log writes. Meanwhile, each PM update for MOD datastructures requires one flush for the data and in some cases, flushes of updated nodes pointing to the data. On average, MOD datastructures exhibit fewer flushes per operation (except in case of vector and queue-pop).

Additional Overheads

Functional Shadowing primarily incurs two overheads: increased cache pressure and space overheads. First, functional datastructures (including vectors and arrays) are implemented

as pointer-based datastructures with little spatial locality. Secondly, extra memory is allocated on every update for the shadow, resulting in additional space overheads.

Cache Pressure.

As PM is expected to have longer read and write latencies than DRAM, miss latencies for cachelines resident in PM will also be correspondingly higher. Hence, higher cache miss ratios result in greater performance degradation than in conventional DRAM-only systems. Unfortunately, it was not possible to separate cache misses to PM and DRAM in our experiments on real hardware, but we expect most of the cache accesses to be for PM cachelines in our workloads.

The pointer-based functional implementations generate more cache misses particularly in the small L1D cache, as seen in Figure 4.12. This is evident in case of map, set and vector, which show almost $4\times$ the cache misses with MOD datastructures than with PMDK. The PMDK implementations of map, set and vector are contiguously laid out in memory and thus have greater spatial locality and fewer pointer-chasing patterns. However, the pointer-based implementations of MOD datastructures are necessary to reduce the shadow copying overheads.

MOD implementations of stack and queue show low cache miss ratios, comparable to the PMDK implementations. Comparable miss rates are to be expected as both MOD and PMDK datastructures are pointer-based implementations. Moreover, push and pop operations in these datastructures only operate on the head or the tail, resulting in high temporal locality of accesses.

Space overheads.

Structural sharing optimizations in our datastructure implementations minimize the space overheads of Functional Shadowing. Naive shadowing techniques that create a complete copy of the original datastructure require $1\times$ additional space on every update. In Table 4.3,

	map	set	stack	queue	vector
MOD	120,409	93,658	149,750	79,947	32,654
PMDK	∞	∞	199,387	100,116	∞

Table 4.3: Number of update operations needed to double memory usage of datastructures with 1M entries with memory reclamation disabled. For PMDK map, set and vector, update operations happen in place without allocating memory.

we report the worst-case space overheads for our datastructures by measuring the number of updates required to double the memory usage of a datastructure of size 1M entries, with memory reclamation disabled.

On average, it takes about 95K updates for MOD datastructures with 1M to double in size, even without any memory reclamation. More importantly, each updated version only requires $0.00002-0.00004\times$ extra memory beyond the original version. In reality, we reclaim the old version of the data once the new version is guaranteed to be durable. Hence, the memory requirements are equal to the original datastructure. However, our worst-case measurement is more interesting as it reveals the additional pressure on the memory allocator to allocate memory for the shadow.

4.7 Comparing Related Work with MOD

We categorize prior work on reducing PM flushing overheads into general optimizations for PM-STM implementations and specific optimizations for durable datastructures.

PM-STM Optimizations.

There have been software and hardware proposals to improve the performance of PM-STM implementations.

Software approaches include Mnemosyne [139], NV-Heaps [26], SoftWrap [43], Intel PMDK [61], JUSTDO [69], iDO [85], Romulus [30], DudeTM [84]. Mnemosyne, SoftWrap, Romulus and DudeTM rely on redo logging, NV-Heaps employs undo logging techniques

and PMDK recently switched from undo logging (in v1.4) to hybrid undo-redo log (in v1.5) [62]. Each of these approaches requires **4+ ordering points per FASE**, although Mnemosyne has a torn-bit raw word log optimization to reduce ordering points to one per transaction. Most undo-logging implementations require ordering points proportional to the number of contiguous data ranges modified in each transaction and can have as many as 50 ordering points in a transaction [94]. In contrast, redo-logging implementations require relatively constant number of ordering points regardless of the size of the transaction and are better for large transactions. However, redo logging requires load interposition to redirect loads to updated PM addresses, resulting in slow reads and increased complexity. MOD datastructures require no load interposition and require only **one ordering point per FASE**.

Romulus and DudeTM both utilize innovative approaches based on redo-logging and shadow paging to reduce ordering constraints. Romulus uses a volatile redo-log with shadow data stored in PM while DudeTM uses a persistent redo-log with shadow data stored in DRAM. Both of these approaches double the memory consumption of the application as two copies of the data are maintained. This is a greater challenge with DudeTM as the shadow occupies DRAM capacity, which is expected to be much smaller than available PM. Our MOD datastructures only have two versions during an update operation. Moreover, there is significant space overlap between the two versions. Both DudeTM and Romulus incur logging overheads and require store interposition, unlike MOD datastructures.

The optimal ordering constraints for PM-STM implementations under idealized scenarios have been analyzed [79]. The results show that PM-STM performance can be improved using new hardware primitives that support Epoch or Strand Persistency [79], neither of which are currently supported by any architectures. Moreover, no PM-STM implementations or PM software have been designed for strand persistency yet. MOD datastructures offers an alternative approach to PM-STM to reduce ordering constraints on currently

available hardware.

Finally, better hardware primitives for ordering and durability have also been proposed [94, 123]. For instance, DPO [80] and HOPS [94] propose lightweight ordering fences that do not stall the CPU pipeline. Efficient Persist Barriers [72] move cacheline flushes out of the critical path of execution by minimizing epoch conflicts. Speculative Persist Barriers [123] allow the core to speculatively execute instructions immediately following an ordering point. Forced Write-Back [100] proposes cache modifications to perform efficient flushes with low software overheads. All these proposals reduce the performance impact of each ordering point in PM applications, whereas we reduce the number of ordering points in these applications. Moreover, these proposals require invasive hardware modifications to the CPU core and/or the cache hierarchy while MOD datastructures improve performance on unmodified hardware.

Recoverable Datastructures.

While Functional Shadowing provides a way to directly convert existing functional datastructures into recoverable ones, the following papers demonstrate the value of handcrafting recoverable datastructures. Dali [95] is a recoverable prepend-only hashmap that is updated non-destructively while preserving the old version. Updates in both Functional Shadowing and Dali are logically performed as a single epoch to minimize ordering constraints. However, our datastructures are optimized to reuse data between versions, while the Dali hashmap uses a list of historical records for each key. The CDDS B-tree [136] is a recoverable datastructure that also relies on versioning for crash-consistency. Version numbers are stored in the nodes of the B-tree to allow nondestructive updates but additional work is done on writes to order nodes by version number for faster lookup. However, it is not straightforward to extend such fine-grained versioning to other datastructures beyond B-trees. Instead, we rely on versioning at the datastructure-level.

There have also been several attempts at optimizing recoverable B+-trees, which are

commonly used in key-value stores and filesystems. NV-Tree [147] achieves significant performance improvement by storing internal tree nodes in volatile DRAM and reconstructing them on a crash. wB+-Trees uses atomic writes and bitmap-based layout to reduce the number of PM writes and flushes for higher performance. These optimizations cannot be directly extended to commonly used datastructures such as vectors and queues. Our MOD datastructures are all implemented as trees, and could allow these optimizations to apply generally to more datastructures with further research.

Lock-Free Datastructures.

MOD datastructures share some similarities with lock-free datastructures [41, 53, 90]. Both types of datastructures perform out-of-place updates which are committed by atomically updating a pointer. Accordingly, we expect that ideas from lock-free datastructures could enable concurrent accesses to MOD datastructures.

4.8 Conclusion

Persistent memory devices are close to becoming commercially available. Ensuring consistency and durability across failures introduces new requirements on programmers and new demands on hardware to efficiently move data from volatile caches into persistent memory. Minimally ordered durable datastructures provide an efficient mechanism that copes with the performance characteristics of Intel's Optane DCPMM for much higher performance. Rather than focusing on minimizing the amount of data written, MOD datastructures minimize the ordering points that impose long program delays. Furthermore, they can be created via simple extensions to a large library of existing highly optimized functional datastructures providing flexibility to programmers.

— 5 —

Devirtualized Memory for Heterogeneous Systems

If you ignore the rules people will,
half the time, quietly rewrite them
so that they don't apply to you.

TERRY PRATCHETT

In addition to data persistence, emerging PM devices also provide vast memory capacity accessible at low-latency. This makes PM devices very attractive for modern data-centric workloads like machine learning. When training deep neural networks, the memory capacity required storing feature maps (i.e., intermediate results) and gradients scales linearly with the depth of the network [22]. For instance, a Recurrent Neural Network layer with 1760 hidden units and a mini-batch size of 64 requires 1.3 GB of memory capacity per layer [35]. Thus, TB-scale main memories composed of PM devices facilitate our ability to explore deeper neural networks.

Unfortunately, virtual memory—initially designed for megabyte-sized memories—incurs significant overheads at memory capacities in the order of hundreds of gigabytes and beyond. For CPUs, address translation overheads have worsened with increasing memory capacities, reaching up to 50% of overall execution time for some big-memory workloads [6, 75]. These overheads occur despite large, complicated and power-hungry address translation hardware with massive two-level translation lookaside buffers (TLBs).

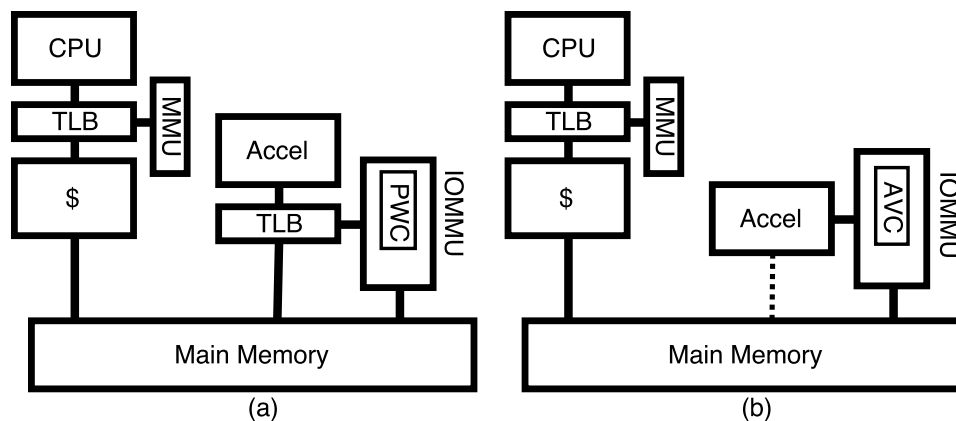


Figure 5.1: Heterogeneous systems with (a) conventional VM with translation on critical path and (b) DVM with Devirtualized Access Validation alongside direct access on reads.

As main memory capacity rises to terabytes, VM overheads will further worsen, especially as TLB sizes have stagnated [75].

Even today, VM overheads are more acute in compute units like accelerators that cannot justify allocating substantial area and power resources to address translation hardware. The end of Dennard Scaling and slowing of Moore’s law has intensified research and academic focus on heterogeneous systems having special-purpose accelerators alongside conventional CPUs. In such systems, computations are offloaded from general-purpose cores to these accelerators. Specialized accelerators are an attractive alternative to general-purpose CPUs as they offer massive performance and/or power improvements in domains such as graph processing [1, 49], data analytics [145, 146], and neural computing [23, 50]. Unfortunately, expensive address translation on every memory access degrades the performance of accelerators. Moreover, supporting VM on accelerators requires memory management hardware like TLBs and page-table walkers, which erodes the benefits in power efficiency. Thus, supporting conventional VM on accelerators is expensive.

Ideally, accelerators want direct access to host physical memory to avoid address translation overheads, eliminate expensive data copying and facilitate fine-grained data sharing. This approach is simple to implement as it does not need large, power-hungry structures such as translation lookaside buffers (TLBs). Moreover, the low power and area consump-

tion are extremely desirable for small accelerators.

However, allowing direct access to physical memory (PhysM) is not generally acceptable. Applications rely on the memory protection and isolation of virtual memory (VM) to prevent malicious or erroneous accesses to their data [102]. Similar protection guarantees are needed when accelerators are multiplexed among multiple processes. Additionally, a virtual address space shared with the CPUs is needed to support “pointer-is-a-pointer” semantics [117]. This allows pointers to be dereferenced on both the CPU and the accelerator which increases the programmability of heterogeneous systems.

To preserve the useful features of VM with minimal overheads, we propose a radical idea to *de-virtualize* virtual memory by eliminating address translation on most memory accesses (Figure 5.1). We achieve this by allocating most memory such that its virtual address (VA) is the same as its physical address (PA). We refer to such allocations as Identity Mapping (VA==PA). As the PA for most accesses is identical to the VA, DVM replaces slow page-level address translation with faster region-level Devirtualized Access Validation (DAV). For DAV, the IO memory management unit (IOMMU) verifies that the process holds valid permissions for the access and that the access is to an identity-mapped page. Conventional address translation is still needed for accesses to non identity-mapped pages. Thus, DVM provides a virtual address space, useful for programmers, while avoiding expensive address translation in the common case.

DAV can be optimized by exploiting the underlying contiguity of permissions. Permissions are typically granted and enforced at coarser granularities than a 4KB page and are uniform across regions of virtually contiguous pages, unlike translations. While DAV is still performed via hardware page walks, we introduce Permission Entries (PEs), which are a new page table entry format for storing coarse-grained permissions. PEs reduce DAV overheads in two ways. First, depending on the available contiguity, page walks can be shorter. Second, PEs significantly reduce the size of the overall page table thus improving the performance of page walk caches. DVM for accelerators is completely transparent to

applications, and requires small OS changes to identity map memory allocations on the heap and construct PEs.

Furthermore, devirtualized memory can optionally be used to reduce VM overheads for CPUs by identity mapping most segments in a process's address space. This requires additional OS and hardware changes.

This chapter describes a memory management approach for heterogeneous systems and makes these contributions:

- We propose DVM to minimize VM overheads, and implement OS support in Linux 4.10.
- We develop a compact page table representation by exploiting the contiguity of permissions through a new page table entry format called the Permission Entry.
- We design the Access Validation Cache (AVC) to replace both TLBs and Page Walk Caches (PWC). For a graph processing accelerator, DVM with an AVC is 2X faster while consuming 3.9X less dynamic energy for memory management than a highly optimized VM implementation with 2M pages.
- We extend DVM to support CPUs (cDVM), thereby enabling unified memory management throughout the heterogeneous system. cDVM lowers the overheads of VM in big-memory workloads to 5% for CPUs.

However, DVM does have some limitations. Identity Mapping allocates memory eagerly and contiguously (Section 5.3) which aggravates the problem of memory fragmentation, although we do not study this effect in this chapter. Additionally, while copy-on-write (COW) and fork are supported by DVM, on the first write to a page, a copy is created which cannot be identity mapped, eschewing the benefits of DVM for that mapping. Thus, DVM is not as flexible as VM but avoids most of the VM overheads.

5.1 Chapter Background

Our work focuses on accelerators running big-memory workloads with irregular access patterns such as graph-processing, machine learning and data analytics. As motivating examples, we use graph-processing applications like Breadth-First Search, PageRank, Single-Source Shortest Path and Collaborative Filtering as described in Section 5.5. First, we discuss why existing approaches for memory management are not a good fit for these workloads.

Accelerator programming models employ one of two approaches for memory management (in addition to unsafe direct access to PhysM). Some accelerators use separate address spaces [73, 99]. This necessitates explicit copies when sharing data between the accelerator and the host processor. Such approaches are similar to discrete GPGPU programming models. As such, they are plagued by the same problems: (1) the high overheads of data copying require larger offloads to be economical; and (2) this approach makes it difficult to support pointer-is-a-pointer semantics, which reduces programmability and complicates the use of pointer-based data structures such as graphs.

To facilitate data sharing, accelerators (mainly GPUs) have started supporting unified virtual memory, in which accelerators can access PhysM shared with the CPU using virtual addresses. This approach typically relies on an IOMMU to service address translation requests from accelerators [2, 64], as illustrated in Figure 5.1. We focus on these systems, as address translation overheads severely degrade the performance of these accelerators [28].

For our graph workloads, we observe high TLB miss rates of 21% on average with a 128-entry TLB (Figure 5.2). There is little spatial locality and hence using larger 2MB pages improves the TLB miss rates only by 1% on average. TLB miss rates of about 30% have also been observed for GPU applications [107, 109]. While optimizations specific to GPU microarchitecture for TLB-awareness (e.g., cache-conscious warp scheduling) have been proposed to mitigate these overheads, these optimizations are not general enough to support efficient memory management in heterogeneous systems with multiple types of

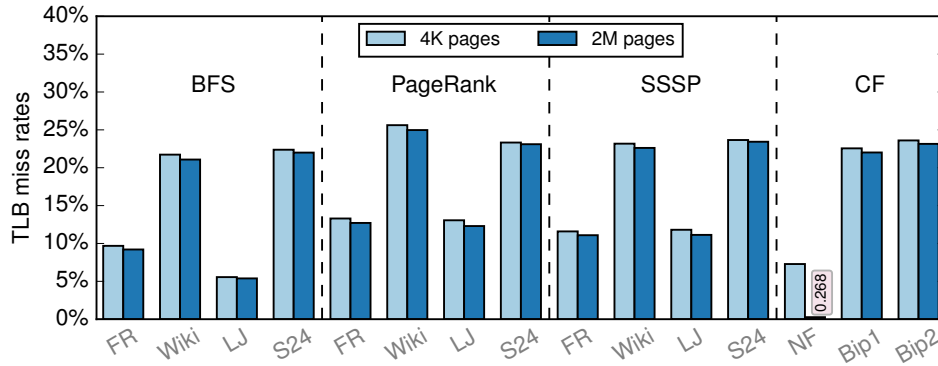


Figure 5.2: TLB miss rates for Graph Workloads with 128-entry TLB

accelerators.

Some accelerators (e.g., Tesseract [1]) support simple address translation using a base-plus-offset scheme such as Direct Segments [6]. With this scheme, only memory within a single contiguous PhysM region can be shared, limiting its flexibility. Complicated address translation schemes such as range translations [75] are more flexible as they support multiple address ranges. However, they require large and power-hungry Range TLBs, which may be prohibitive given the area and power budgets of accelerators.

As a result, we see that there is a clear need for a simple, efficient, general and performant memory management approach for accelerators.

5.2 Devirtualizing Memory

In this section, we present the high-level design of our Devirtualized Memory (DVM) approach. Before discussing DVM, we enumerate the goals for a memory management approach suitable for accelerators (as well as CPUs).

List of Goals

Programmability. Simple programming models are important for increased adoption of accelerators. Data sharing between CPUs and accelerators must be supported, as accelera-

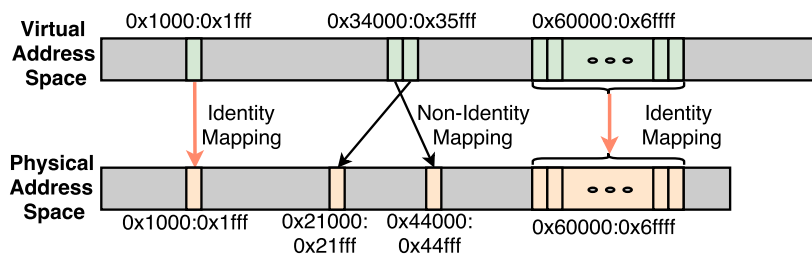


Figure 5.3: Address Space with Identity Mapped and Demand Paged Allocations.

tors are typically used for executing parts of an application. Towards this end, solutions should preserve pointer-is-a-pointer semantics. This improves the programmability of accelerators by allowing the use of pointer-based data structures without data copying or marshalling [117].

Power/Performance. An ideal memory management scheme should have near zero overheads even for irregular access patterns in big-memory systems. Additionally, MMU hardware must consume little area and power. Accelerators are particularly attractive when they offer large speedups under small resource budgets.

Flexibility. Memory management schemes must be flexible enough to support dynamic memory allocations of varying sizes and with different permissions. This precludes approaches whose benefits are limited to a single range of contiguous virtual memory.

Safety. No accelerator should be able to reference a physical address without the right authorization for that address. This is necessary for guaranteeing the memory protection offered by virtual memory. This protection attains greater importance in heterogeneous systems to safeguard against buggy or malicious third-party accelerators [103].

Devirtualized Memory

To minimize VM overheads, DVM introduces *Identity Mapping* and leverages permission validation [78, 143] in the form of *Devirtualized Access Validation*. Identity mapping allocates memory such that all VAs in the allocated region are identical to the backing PAs. DVM uses identity mapping for all heap allocations. Identity mapping can fail if no suitable

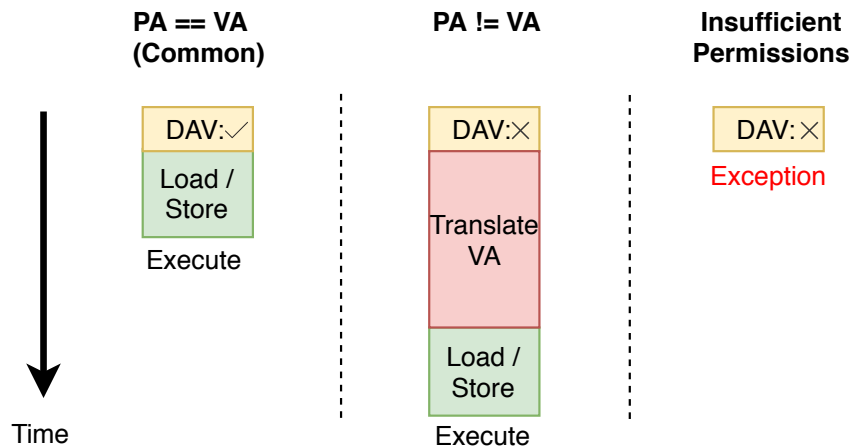


Figure 5.4: Memory Accesses in DVM

address range is available in both the virtual and physical address spaces. In this case, DVM falls back to demand paging. Figure 5.3 illustrates an address space with identity mapping.

As $PA=VA$ for most data on the heap, DVM can avoid address translation on most memory accesses. Instead, it is sufficient to verify that the accessed VA is identity mapped and that the application holds sufficient permissions for the access. We refer to these checks as Devirtualized Access Validation. In rare cases when $PA \neq VA$, DAV fails and DVM resorts to address translation as in conventional VM. Memory accesses in DVM are illustrated in Figure 5.4.

DVM is designed to satisfy the goals listed earlier:

Programmability. DVM enables shared address space in heterogeneous systems at minimal cost, thereby improving the programmability of such systems.

Power/Performance. DVM optimizes for performance and power-efficiency by performing DAV much faster than full address translation. DAV latency is minimized by exploiting the contiguity of permissions for compact storage and efficient caching performance (Section 5.3). Even in the rare case of an access to a non-identity mapped page, performance is no worse than conventional VM as DAV reduces the address translation latency, as explained in Section 5.3.

Flexibility. DVM facilitates page-level sharing between the accelerator and the host CPU since regions as small as a single page can be identity mapped independently, as shown in Figure 5.3. However, regions of size 128KB or greater are needed for improving performance by shortening the page table walk required for DAV (described in Section 5.3). This allows DVM to benefit a variety of applications, including those that do not have a single contiguous heap. Furthermore, DVM is transparent to most applications.

Safety. DVM completely preserves conventional virtual memory protection as all accesses are still checked for valid permissions. If appropriate permissions are not present for an access, an exception is raised on the host CPU.

5.3 Implementing DVM for Accelerators

Having established the high-level model of DVM, we now dive into the implementation of identity mapping and devirtualized access validation. We add support for DVM in accelerators with modest changes to the OS and IOMMU and without any CPU hardware modifications.

First, we describe two alternative mechanisms for fast DAV. Next, we discuss OS modifications to support identity mapping. Here, we use the term *memory region* to mean a collection of virtually contiguous pages with the same permissions. Also, we use page table entries (PTE) to mean entries at any level of the page table.

Devirtualized Access Validation

We implement DAV efficiently with one of two mechanisms:

- Conventional page table walk accelerated by the *permission bitmap* and TLB.
- *Compact page table* walk accelerated by the *Access Validation Cache*.

Note that the IOMMU uses a separate copy of page tables to avoid affecting CPU hardware. We use the following 2-bit encoding for permissions—00:No Permission, 01:Read-Only, 10:Read-Write and 11:Read-Execute.

Permission Bitmap

The Permission Bitmap (BM) stores page permissions for identity-mapped pages. For such pages, virtual-to-physical address mappings are not needed. Instead, we apply the invariant that any page with valid permissions ($\neq 00$) in the BM is identity mapped. For each memory access from an accelerator, the IOMMU consults the BM to check permissions. For pages with 00 permissions, the IOMMU relies on regular address translation with a full page table walk. If the page is identity mapped, the IOMMU updates the BM with its permissions.

The Bitmap is a flat bitmap stored in physical memory, maintaining 2-bit permissions for each physical page. As with page tables, there is a separate bitmap for each process. It is physically indexed, and lookups assume $PA=VA$. For a page size of 4KB, this incurs storage overheads of approximately 0.006% of the physical memory capacity for each active accelerator. Finding a page's permissions simply involves calculating the offset into the bitmap and adding it to the bitmap's base address. We cache BM entries along with L2-L4 PTEs in a simple data cache to expedite access validation.

The hardware design of the Bitmap is like the Protection Table used in Border Control (BC) [102], although the storage and lookup policies differ. The Protection Table in BC stores permissions for all physical pages and is looked up after address translation. BM in DVM stores permissions for only identity-mapped pages, and is looked up to confirm identity mapping and avoid full address translation.

The simple BM has the benefit of leaving host page tables unchanged. However, it stores permissions individually for every physical page, which is inefficient for big-memory systems, especially with sparse memory usage. We address this drawback with our alternative

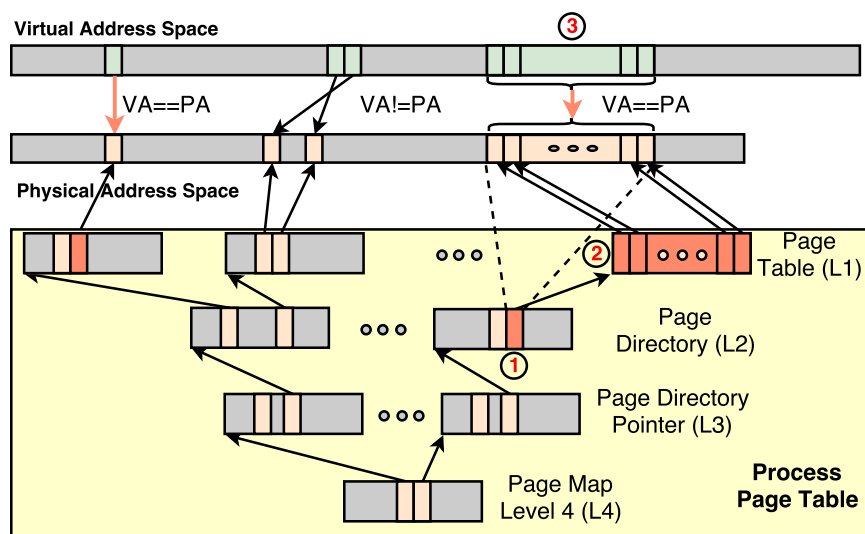


Figure 5.5: 4-level Address Translation in x86-64

mechanism described next.

Compact Page Tables

We leverage available contiguity in permissions to store them at a coarse granularity resulting in a compact page table structure. Figure 5.5 shows an x86-64 page table. An L2 Page Directory entry (L2PDE) ① maps a contiguous 2MB VA range ③. Physical Page Numbers are stored for each 4K page in this range, needing 512 L1 page table entries (PTEs) ② and 4KB of memory. However, if pages are identity mapped, PAs are already known and only permissions need to be stored. If permissions are the same for the entire 2MB region (or an aligned sub-region), these could be stored at the L2 level. For larger regions, permissions can be stored at the L3 and L4 levels. For new 5-level page tables, permissions can also be stored at the L5 levels.

We introduce a new type of leaf PTE called the Permissions Entry (PE), shown in Figure 5.6. PEs are direct replacements for regular PTEs at any level, with the same size (8 bytes) and mapping the same VA range as the replaced PTE. PEs contain sixteen permission fields, currently 2-bit each. A permission entry bit is added to all PTEs, and is 1 for PEs

and 0 for other regular PTEs.

Each PE records separate permissions for sixteen aligned regions comprising the VA range mapped by the PE. Each constituent region is 1/16th the size of the range mapped by the PE and is aligned on an appropriate power-of-two granularity. For instance, an L2PE maps a 2MB VA range of sixteen 128KB (=2MB/16) regions aligned on 128KB address boundaries. An L3PE maps a 1GB VA range of sixteen 64MB regions aligned on 64MB address boundaries. Other intermediate sizes can be handled simply by replicating permissions. Thus, a 1MB region is mapped by storing permissions across 8 permission fields in an L2PE. Region ③ in Figure 5.5 can be mapped by an L2PE with uniform permissions stored in all 16 fields.

PEs implicitly guarantee that any allocated memory in the mapped VA range is identity-mapped. Unallocated memory i.e., gaps in the mapped VA range can also be handled gracefully, if aligned suitably, by treating them as regions with no permissions (00). This frees the underlying PAs to be re-used for non-identity mapping in the same or other applications or for identity mappings in other applications. If region 3 is replaced by two adjacent 128 KB regions at the start of the mapped VA range with the rest unmapped, we could still use an L2PE to map this range, with relevant permissions for the first two regions, and 00 permissions for the rest of the memory in this range.

On an accelerator memory request, the IOMMU performs DAV by walking the page table. A page walk ends on encountering a PE, as PEs store information about identity mapping and permissions. If insufficient permissions are found, the IOMMU may raise an exception on the host CPU.

If a page walk encounters a leaf PTE, the accessed VA may not be identity mapped. In this case, the leaf PTE is used to perform address translation i.e., use the page frame number recorded in the PTE to generate the actual PA. This avoids a separate walk of the page table to translate the address. More importantly, this ensures that even in the fallback case (PA!=VA), the overhead (i.e., full page walk) is no worse than conventional VM.

Input Graph	Page Tables (in KB)	% occupied by L1PTEs	Page Tables with PEs (in KB)
FR	616	0.948	48
Wiki	2520	0.987	48
LJ	4280	0.992	48
S24	13340	0.996	60
NF	4736	0.992	52
BIP1	2648	0.989	48
BIP2	11164	0.996	68

Table 5.1: Page Table Sizes for PageRank (first four rows) and Collaborative Filtering (last three rows) for different input graphs. PEs reduce the page table size by eliminating most L1PTEs.

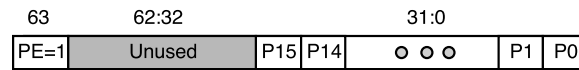


Figure 5.6: Structure of a Permission Entry. PE: Permission Entry, P15-P0: Permissions.

Incorporating PEs significantly reduces the size of page tables (Table 5.1) as each higher-level (L2-L4) PE directly replaces an entire sub-tree of the page table. For instance, replacing an L3PTE with a PE eliminates 512 L2PDEs and up to 512×512 L1PTEs, saving as much as 2.05 MB. Most of the benefits come from eliminating L1PTEs as these leaf PTEs comprise about 98% of the size of the page tables. Thus, PEs make page tables more compact.

Access Validation Cache

The major value of smaller page tables is improved efficacy of caching PTEs. In addition to TLBs which cache PTEs, modern IOMMUs also include page walk caches (PWC) to store L2-L4 PTEs [5]. During a page walk, the page table walker first looks up internal PTEs in the PWC before accessing main memory. In existing systems, L1PTEs are not cached to avoid polluting the PWC [11]. Hence, page table walks on TLB misses incur at least one memory access, for obtaining the L1PTE.

We propose the Access Validation Cache (AVC), which caches all intermediate and leaf entries of the page table, to replace both TLBs and PWCs for accelerators. The AVC is a

standard 4-way set-associative cache with 64B blocks. The AVC caches 128 distinct PTEs, resulting in a total capacity of 1 KB. It is physically-indexed and physically tagged cache, as page table walks use physical addresses. For PEs, this provides 128 sets of permissions.

On every memory reference by an accelerator, the IOMMU walks the page table using the AVC. In the best case, page walks require 2-4 AVC accesses and no main memory access. Caching L1PTEs allows AVC to exploit their temporal locality, as done traditionally by TLBs. But, L1PTEs do not pollute the AVC as the introduction of PEs greatly reduces the number of L1PTEs. Thus, the AVC can perform the role of both a TLB and a traditional PWC.

Due to the smaller page tables, even a small 128-entry (1KB) AVC has very high hit rates, resulting in fast access validation. As the hardware design is similar to conventional PWCs, the AVC is just as energy-efficient. Moreover, the AVC is more energy-efficient than a comparably sized, fully associative (FA) TLB due to a less associative lookup.

Optional Meltdown-susceptible Preload on Loads

The original paper [51] implemented an optional preload optimization which we describe below. However, we have eliminated this optimization from our default implementation in this chapter to reduce vulnerability to side-channel attacks such as Meltdown [83] and Spectre [77].

The eliminated pre-load optimization is as follows. If an accelerator supports the ability to squash and retry an in-flight load, DVM can allow a preload to occur in parallel with DAV. As a result, the validation latency for loads can be overlapped with the memory access latency. If the access is validated successfully, the preload is treated as the actual memory access. Otherwise, it is discarded, and the access is retried to the correct, translated PA. For stores, this optimization is not possible because the physical address must be validated before the store updates memory.

```

Memory-Allocation (Size S)
PA ← contiguous-PhysM-allocation(S)
if PA ≠ NULL then
  VA ← VM-allocation(S)
  Move region to new VA2 equal to PA
  if Move succeeds then
    | return VA2 // Identity-Mapped
  end
  else
    | Free-PhysM(PA, S)
    | return VA // Fallback to Demand-Paging
  end
end
else
  | VA ← VM-allocation(S)
  | return VA // Fallback to Demand-Paging
end

```

Figure 5.7: Pseudocode for Identity Mapping

Identity Mapping

As accelerators typically only access shared data on the heap, we implement identity mapping only for heap allocations, requiring minor OS changes. The application's heap is actually composed of the heap segment (for smaller allocations) as well as memory-mapped segments (for larger allocations).

To ensure $VA==PA$ for most addresses in memory, firstly, physical frames (and thus PAs) need to be reserved at the time of memory allocation. For this, we use *eager paging* [75]. Next, the allocation is mapped into the virtual address space at VAs equal to the backing PAs. This may result in heap allocations being mapped anywhere in the process address space as opposed to a hardcoded location. To handle this, we add support for a flexible address space. Below, we describe our implementation in Linux 4.10. Figure 5.7 shows the pseudocode for identity mapping.

Eager Contiguous Allocations

Identity Mapping in DVM is enabled by eager contiguous allocations of memory. On memory allocations, the OS allocates physical memory then sets the VA equal to the PA. This is unlike demand paging used by most OSes, which allocates physical frames lazily at the time of first access to a virtual page. For allocations larger than a single page, contiguous allocation of physical memory is needed to guarantee $VA==PA$ for all the constituent pages. We use the eager paging modifications to Linux's default buddy allocator developed by others [75] to allocate contiguous powers-of-two pages. Once contiguous pages are obtained, extra pages obtained due to rounding up are freed to the allocator immediately. Eager allocation may increase physical memory use if programs allocate much more memory than they actually use.

Flexible Address Space

Operating systems historically dictated the layout of usermode address spaces, specifying where code, data, heap, and stack reside. For identity mapping, our modified OS assigns VAs equal to the backing PAs. Unfortunately, there is little control over the allocated PAs without major changes to the default buddy allocator in Linux. As a result, we could have a non-standard address space layout, for instance with the heap below the code segment in the address space. To allow such cases, the OS needs to support a flexible address space with no hard constraints on the location of the heap and memory-mapped segments.

Heap. We modify the default behavior of glibc `malloc` to always use the `mmap` system call instead of `brk`. This is because identity mapped regions cannot be grown easily, and `brk` requires dynamically growing a region. We initially allocate a memory pool to handle small allocations. Another pool is allocated when the first is full. Thus, we turn the heap into noncontiguous memory-mapped segments, which we discuss next.

Memory-mapped segments. We modify the kernel to accommodate memory-mapped segments anywhere in the address space. Address Space Layout Randomization (ASLR)

already allows randomizing the base positions of the stack, heap as well as memory-mapped regions (libraries) [135]. Our implementation further extends this to allow any possible positions of the heap and memory-mapped segments.

Low-memory situations. While most high-performance systems are configured with enough memory capacity, contiguous allocations can result in fragmentation over time and preclude further contiguous allocations.

In low memory situations, DVM reverts to standard paging. Furthermore, to reclaim memory, the OS could convert permission entries to standard PTEs and swap out memory (not implemented). We expect such situations to be rare in big-memory systems, which are our main target. Also, once there is sufficient free memory, the OS can reorganize memory to reestablish identity mappings.

5.4 Discussion

Here we address potential concerns regarding DVM.

Security Implications. Side-channel attacks have become important architectural considerations recently, especially in light of Meltdown [83] and Spectre variants [54, 77]. These attacks use speculative instruction execution to cross software protection boundaries that are known (Meltdown/Foreshadow [16]) or unknown (most Spectre variants) to hardware. Of these attacks, Spectre variants are orthogonal to virtual memory—conventional address translation or DVM—because all virtual addresses they generate in their instruction-level speculation have appropriate virtual address permissions. We believe that DVM can eliminate Meltdown/Foreshadow by always verifying memory protection before a data fetch, thus eliminating speculative preload at the cost of some performance loss. Without preload, we see no evidence that DVM would be vulnerable to Meltdown or Foreshadow as DVM would no longer do any speculation. Moreover, we know of no non-speculative timing side-channel that DVM introduces, but the possibility has not been formally ruled out. We

leave a formal proof of the security properties of DVM for future work.

Prior to Meltdown and Spectre, there have been other side-channel attacks specifically targeting the memory management unit (MMU) [44, 47, 121]. Such attacks seek to derandomize virtual addresses corresponding to the code and data segments of a victim process. A proposed defense against such attacks is to increase the entropy of address-space layout randomization (ASLR). For instance, Linux provides 28 bits of ASLR entropy while Windows 10 offers 24 bits for the heap [141]. DVM randomizes physical addresses, which may have fewer bits than virtual addresses, but still provides 20 bits for 2MB-aligned allocation in 2TBs of physical address space. However, even the stronger Linux randomization has been derandomized by existing attacks [44, 47, 121]. Moreover, earlier works [47] have concluded that address randomization is not an effective defense mechanism against side-channel attacks.

Effect on Memory Isolation. While DVM sets $PA=VA$ in the common case, this does not weaken isolation. Just because applications can address all of PhysM does not give them permissions to access it [21]. This is commonly exploited by OSes. For instance, in Linux, all physical memory is mapped into the kernel address space, which is part of every process. Although this memory is addressable by an application, any user-level access to this region will be blocked by hardware due to lack of permissions in the page table.

Copy-on-Write (CoW). CoW is an optimization for minimizing the overheads of copying data, by deferring the copy operation till the first write. Before the first write, both the source and destination get read-only permissions to the original data. It is most commonly used by the `fork` system call to create new processes.

CoW can be performed with DVM without any correctness issues. Before any writes occur, there is harmless read-only aliasing. The first write in either process allocates a new page for a private copy, which cannot be identity-mapped, as its VA range is already visible to the application, and the corresponding PA range is allocated for the original data. Thus, the OS reverts to standard paging for the address. Thus, we recommend against using

CoW for data structures allocated using identity mapping.

Unix-style Fork. The fork operation in Unix creates a child process, and copies a parent's private address space into the child process. Commonly, CoW is used to defer the actual copy operation. As explained in the previous section, CoW works correctly, but can break identity mapping.

Hence, we recommend calling `fork` before allocating structures shared with accelerators. If processes must be created later, then the `posix_spawn` call (combined `fork` and `exec`) should be used when possible to create new processes without copying. Alternatively, `vfork`, which shares the address space without copying, can be used, although it is typically considered less safe than `fork`. Others have also recommended deprecating `fork` due to several issues, including lack of compatibility with new mechanisms [7].

Virtual Machines. DVM can be extended for virtualized environments as well. The overheads of conventional virtual memory are exacerbated in such environments [10] as memory accesses need two levels of address translation (1) guest virtual address (gVA) to guest physical address (gPA) and (2) guest physical address to system physical address (sPA).

To reduce these costs, DVM can be extended in three ways. With guest OS support for multiple non-contiguous physical memory regions, DVM can be used to map the gPA to the sPA directly in the hypervisor, or in the guest OS to map gVA to gPA. These approaches convert the two-dimensional page walk to a one-dimensional walk. Thus, DVM brings down the translation costs to unvirtualized levels. Finally, there is scope for broader impact by using DVM for directly mapping gVA to sPA, eliminating the need for address translation on most accesses.

Comparison with Huge Pages. Here we offer a qualitative comparison, backed up by a quantitative comparison in Section 5.5. DVM breaks the serialization of translation and data fetch, unlike huge pages. Also, DVM exploits finer granularities of contiguity by having 16 permission fields in each PE. Specifically, 128KB (=2MB /16) of contiguity is

sufficient for leveraging 2MB L2PEs, and 64MB (=1GB/16) contiguity is sufficient for 1GB L3PEs.

Moreover, supporting multiple page sizes is difficult [31, 131], particularly with set associative TLBs which are commonly used due to their power-efficiency. On the other hand, PEs at higher levels of the page table allow DVM to gracefully scale with increasing memory sizes.

Finally, huge page TLB performance still depends on the locality of memory references. TLB performance can be an issue for big-memory workloads with irregular or streaming accesses [104, 113], as shown in Figure 5.2. In comparison, DVM exploits the locality in permissions which is found in most applications due to how memory is typically allocated.

5.5 Evaluation

Methodology

We quantitatively evaluate DVM using a heterogeneous system containing an out-of-order core and the Graphicionado graph-processing accelerator [49]. Graphicionado is optimized for the low computation-to-communication ratio of graph applications. In contrast to software frameworks, where 94% of the executed instructions are for data movement, Graphicionado uses an application-specific pipeline and memory system design to avoid such inefficiencies. Its execution pipeline and datapaths are geared towards graph primitives—edges and vertices. Also, by allowing concurrent execution of multiple execution pipelines, the accelerator can exploit the available parallelism and memory bandwidth.

To match the flexibility of software frameworks, Graphicionado uses reconfigurable blocks to support the vertex programming abstraction. Thus, a graph algorithm is expressed as operations on a single vertex and its edges. Most graph algorithms can be specified and executed on Graphicionado with three custom functions, namely `processEdge`, `reduce`

and apply. The graph is stored as a list of edges, each in the form of a 3-tuple (srcid, dstid, weight). A list of vertices is maintained where each vertex is associated with a vertex property (i.e., distance from root in BFS or rank in PageRank). The vertex properties are updated during execution. Graphicionado also maintains ancillary arrays for efficient indexing into the vertex and the edge lists.

We simulate a heterogeneous system with one CPU and the Graphicionado accelerator with the open-source, cycle-level gem5 simulator [15]. We implement Graphicionado with 8 processing engines and no scratchpad memory as an IO device with its own timing model in gem5. The computation performed in each stage of a processing engine is executed in one cycle, and memory accesses are made to the shared memory. We use gem5’s full-system mode to run workloads on our modified Linux operating system. The configuration details of the simulation are shown in Table 5.2. For energy results, we use access energy numbers from Cacti 6.5 [92] and access counts from our gem5 simulation.

CPU	
Cores	1
Caches	64KB L1, 2MB L2
Frequency	3 GHz
Accelerator	
Processing Engines	8
TLB Size	128-entry FA
TLB Latency	1 cycle
PWC/AVC Size	128-entry, 4-way SA
PWC/AVC Latency	1 cycle
Frequency	1 GHz
Memory System	
Memory Size	32 GB
Memory B/W	4 channels of DDR4 (51.2 GB/s)

Table 5.2: Simulation Configuration Details

Workloads

We run four common graph algorithms on the accelerator—PageRank, Breadth-First Search, Single-Source Shortest Path and Collaborative Filtering. We run each of these workloads

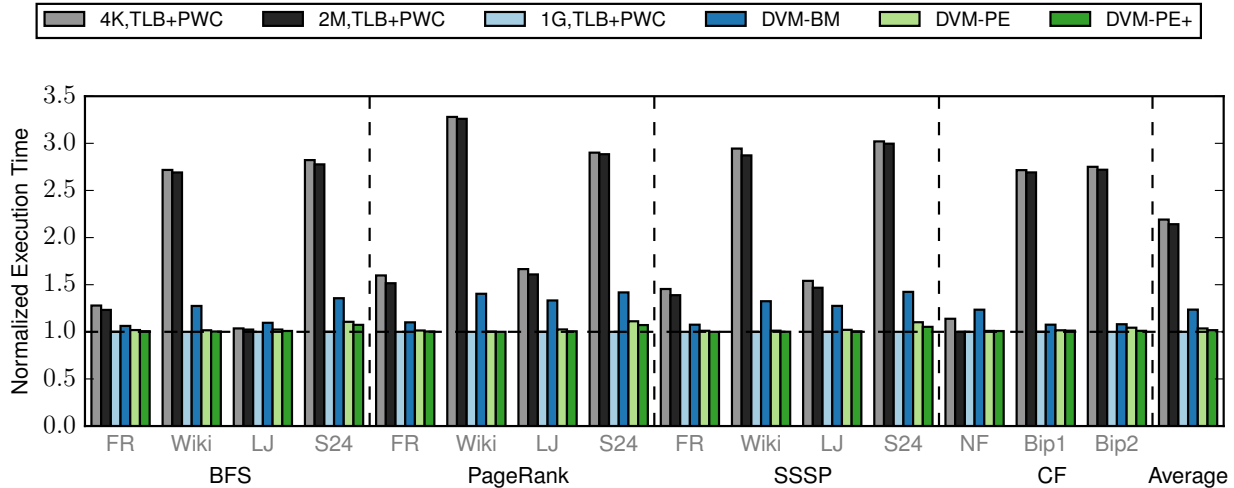


Figure 5.8: Execution time for accelerator workloads, normalized to runtime of ideal implementation.

with multiple real-world as well as synthetic graphs. The details of the input graphs can be found in Table 5.3. The synthetic graphs are generated using the graph500 RMAT data generator [19, 93]. To generate synthetic bipartite graphs, we convert the synthetic RMAT graphs following the methodology described by Satish et al [119].

Graph	# Vertices	# Edges	Heap Size
Flickr (FR) [34]	0.82M	9.84M	288 MB
Wikipedia (Wiki) [34]	3.56M	84.75M	1.26 GB
LiveJournal (LJ) [34]	4.84M	68.99M	2.15 GB
RMAT Scale 24 (RMAT)			6.79 GB
Netflix (NF) [8]	480K users, 18K movies	99.07M	2.39 GB
Synthetic Bipartite 1 (SB1)	969K users, 100K movies	53.82M	1.33 GB
Synthetic Bipartite 2 (SB2)	2.90M users, 100K movies	232.7M	5.66 GB

Table 5.3: Graph Datasets Used for Evaluation

Results

We evaluate seven separate implementations. We evaluate conventional VM implementations using an IOMMU with 128-entry fully associative (FA) TLB and 1KB PWC. We show

the performance with three page sizes—4KB, 2MB and 1GB. Next, we evaluate three DVM implementations with different DAV hardware. First, we store permissions for all VAs in a flat 2MB bitmap in memory for 1-step DAV, with a separate 128-entry cache for caching bitmap entries (*DVM-BM*). 2-bit permissions are stored for all identity-mapped pages in the application’s heap for fast access validation. If no permissions (i.e., 00) are found, full address translation is performed, expedited by a 128-entry FA TLB. Second, we implement DAV using page tables modified to use PEs and a 128-entry (1KB) AVC (*DVM-PE*). Third, we extend *DVM-PE* by allowing preload on reads (*DVM-PE+*) to both learn the impact of Meltdown and give performance for any system unconcerned about Meltdown. Finally, we evaluate an ideal implementation in which the accelerator directly accesses physical memory without any translation or protection checks.

Performance

Figure 5.8 shows the execution time of our graph workloads for different input graphs for the above systems, normalized to the ideal implementation.

DVM-PE outperforms most other VM implementations with only 3.5% overheads. Meltdown-susceptible Preload support in *DVM-PE+* further reduces DVM overheads to only 1.7%. The performance improvements come from being able to complete most page walks entirely from the AVC without any memory references. Conventional PWCs typically avoid caching L1PTEs to prevent cache pollution, so page walks for 4K pages require at least one memory reference.

DVM-BM incurs 23% DVM overheads, much lower than most other VM implementations but greater than the other DVM variants. Unfortunately, the hit rate of the BM cache is not as high as the AVC, due to the much larger size of standard page tables and use of 4KB pages instead of 128KB or larger regions.

4K,TLB+PWC and *2M,TLB+PWC* have high VM overheads, on average 119% and 114% respectively. As seen in Figure 5.2, the irregular access patterns of our workloads result in

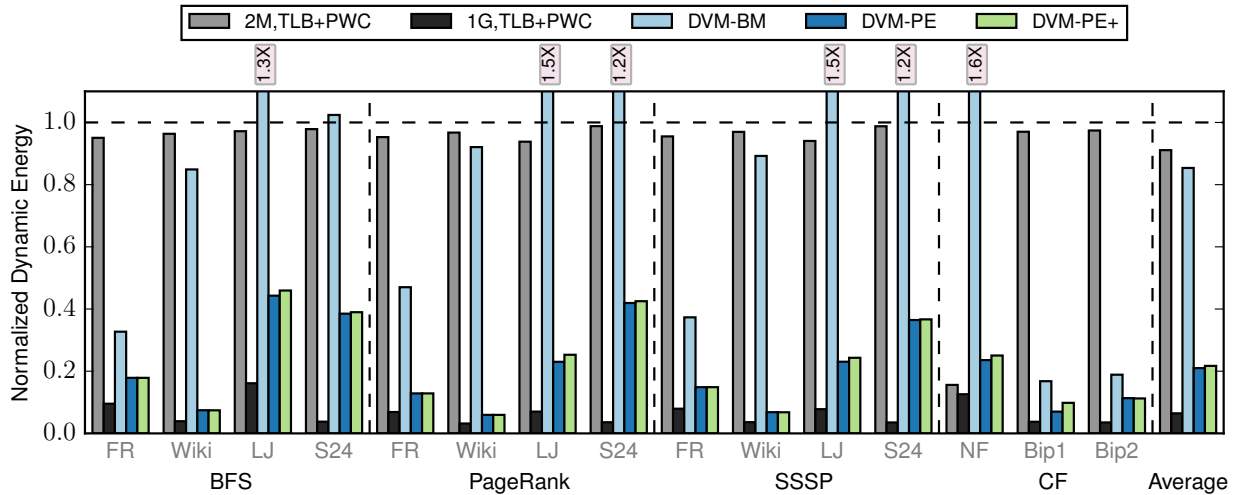


Figure 5.9: Dynamic energy spent in address translation/access validation, normalized to the 4KB, TLB+PWC implementation.

high TLB miss rates. Using 2MB pages does not help much, as the TLB reach is still limited to 256 MB ($128 \times 2\text{MB}$), which is smaller than the working sets of most of our workloads. NF has high TLB hit rates due to higher temporal locality of accesses. Being a bipartite graph, all its edges are directed from 480K users to only 18K movies. The small number of destination nodes results in high temporal locality. As a result, moving to 2MB pages exploits this locality showing near-ideal performance.

1G, TLB+PWC also performs well—virtually no VM overhead—for the workloads and system that we evaluate, but with three issues. First, <10 1GB pages are sufficient to map these workloads, but not necessarily for future workloads. Second, there are known OS challenges for managing 1GB pages. For instance, huge page allocation is not performed dynamically in Linux. Instead, huge pages need to be explicitly pre-allocated through `procf`s and requested via `libhugetlbfs`. Third, the 128-entry fully associative TLB we assume is power-hungry and often avoided in industrial designs (e.g., Intel currently uses four-way set associative).

Energy

Energy is a first-order concern in modern systems, particularly for small accelerators. Here, we consider the impact of DVM on reducing the dynamic energy spent in MMU functions, like address translation for conventional VM and access validation for DVM. We calculate this dynamic energy by adding the energy of all TLB accesses, PWC accesses, and memory accesses by the page table walker [74]. We show the dynamic energy consumption of VM implementations, normalized to $4K,TLB+PWC$ in Figure 5.9.

DVM-PE offers 76% reduction in dynamic translation energy over the baseline. This mainly comes from removing the FA TLB. Also, page walks can be entirely serviced with AVC cache accesses without any main memory accesses, significantly decreasing the energy consumption. Memory accesses for discarded preloads for non-identity mapped pages increase dynamic energy slightly in *DVM-PE+*.

DVM-BM shows a 15% energy reduction on average over the baseline. However, the energy consumption is higher than other DVM variants due to memory references on bitmap cache misses.

1G,TLB+PWC shows low energy consumption due to lack of TLB misses.

Identity Mapping

To evaluate the risk of fragmentation with eager paging and identity mapping, we use the *shbench* benchmark from MicroQuill, Inc [58]. We configure this benchmark to continuously allocate memory of variable sizes until identity mapping fails to hold for an allocation ($VA \neq PA$). Experiment 1 allocated small chunks of memory, sized between 100 and 10,000 bytes. Experiment 2 allocated larger chunks, sized between 100,000 and 10,000,000 bytes. Finally, we ran four concurrent instances of *shbench*, all allocating large chunks as in experiment 2. For each of these, we report the percentage of memory that could be allocated before identity mapping failed for systems with 16 GB, 32 GB and 64 GB of total memory capacity. We observe that 95 to 97% of memory can be allocated with identity mapping,

System Memory	% Memory Allocated (PA==VA)		
	Expt 1	Expt 2	Expt 3
16 GB	96%	95%	96%
32 GB	97%	97%	96%
64 GB	97%	97%	97%

Table 5.4: Percentage of total system memory successfully allocated with identity mapping.

even in memory-constrained systems with 16 GBs of memory. Our complete results are shown in Table 5.4.

5.6 Towards DVM across Heterogeneous Systems

DVM can provide similar benefits for CPUs (cDVM). With the end of Dennard Scaling and the slowing of Moore’s Law, one avenue for future performance growth is to reduce waste everywhere. Towards this end, we discuss the use of DVM for CPU cores to reduce waste due to VM. This opportunity comes with MMU hardware and OS changes that are real, but more modest than we initially expected.

Hardware Changes

To avoid intrusive changes to the CPU pipeline, we retain conventional TLBs and perform DAV on TLB misses. Thus, cDVM expedites the slow page table walk on a TLB miss using the AVC and compact page tables (as described in Section 5.3).

OS Support for cDVM

The simplest way to extend to CPUs is to enable the OS VM and CPU page table walkers to handle the new compact page tables with PEs. Next, we can optionally extend to code and/or stack, but typically the heap is much larger than other segments. We have

Affected Feature	LOC changed
Code Segment	39
Heap Segment	1*
Memory-mapped Segments	56
Stack Segment	63
Page Tables	78
Miscellaneous	15

Table 5.5: Lines of code changed in Linux v4.10 split up by functionality. *Changes for memory-mapped segments affect heap segment, so we only count them once.

implemented a prototype providing this flexibility in Linux v4.10. The lines of code changed is shown in Table 5.5.

Stack. The base addresses for stacks are already randomized by ASLR. The stack of the main thread is allocated by the kernel, and is used to setup initial arguments to launch the application. To minimize OS changes, we do not identity map this stack initially. Once the arguments are setup, but before control passes to the application, we move the stack to the VA matching its PA.

Dynamically growing a region is difficult with identity mapping, as adjacent physical pages may not be available. Instead, we eagerly allocate an 8MB stack for all threads. This wastes some memory, but this can be adjusted. Stacks can be grown above this size using gcc's Split Stacks [132].

The stacks of other threads beside the main thread in a multi-threaded process are allocated as memory-mapped segments and can be handled as discussed previously.

Code and globals. In unmodified Linux, the text segment (i.e., code) is located at a fixed offset near the bottom of the process address space, followed immediately by the data (initialized global variables) and the bss (uninitialized global variables) segments. To protect against return-oriented programming (ROP) attacks [116], OSes have begun to support position independent executables (PIE) which allow binaries to be loaded at random offsets from the base of the address space [114]. PIE incurs a small cost on function calls due to an added level of indirection.

PIE randomizes the base position of the text segment and keeps data and bss segments adjacent. We consider these segments as one logical entity in our prototype and allocate an identity-mapped segment equal to the combined size of these three segments. The permissions for the code region are then set to be Read-Execute, while the other two segments are to Read-Write.

Performance Evaluation

We evaluate the performance benefits of cDVM using memory intensive CPU-only applications like mcf from SPEC CPU 2006 [52], BT, CG from NAS Parallel Benchmarks [4], canneal from PARSEC [14] and xsbench [133].

Using hardware performance counters, we measure L2 TLB misses, page walk cycles and total execution cycles of these applications on an Intel Xeon E5-2430 machine with 96 GB memory, 64-entry L1 DTLB and 512-entry DTLB. Then, we use BadgerTrap [42] to instrument TLB misses and estimate the hit rate of the AVC. Finally, we use a simple analytical model to conservatively estimate the VM overheads under cDVM, like past work [6, 11, 12, 32, 75, 106]. For the ideal case, we estimate running time by subtracting page walk cycles for 2MB pages from total execution cycles.

We compare cDVM with conventional VM using 4KB pages and 2MB pages with Transparent Huge Paging (THP). From our results in Figure 5.10, we see that conventional VM adds about 29% overheads on average with 4KB pages and 13% with THP, even with a two-level TLB hierarchy. THP improves performance by expanding TLB reach and shortening page walks. Due to the limits of our evaluation methodology, we can only estimate performance benefits of the AVC. We find that cDVM reduces VM overheads from 13% with 2MB pages to within 5% of the ideal implementation without address translation. The performance benefits come from shorter page walks with fewer memory accesses. Thus, we believe that cDVM merits more investigation to optimize systems with high VM overheads.

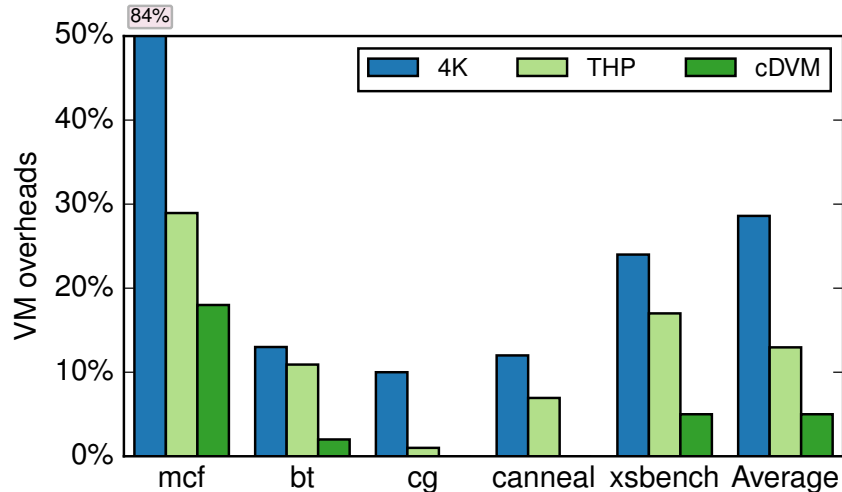


Figure 5.10: Runtime of CPU-only workloads, normalized to the ideal case.

5.7 Related Work in VM for Accelerators and Vast Memory

Overheads of VM. The increasing VM overheads have been studied for CPU workloads (e.g., Direct Segments [6]), and recently for accelerators (e.g., Cong et al. [28], Picorel et al. [108]).

VM for Accelerators. Border Control (BC) [102] recognized the need for enforcing memory security in heterogeneous systems. BC provides mechanisms to checking permissions on physical addresses of requests leaving the accelerator. BC does not affect address translation overheads within the accelerator, which can be reduced significantly through DVM.

Most prior proposals have lowered virtual memory overheads for accelerators using changes in TLB location or hierarchy [28, 138]. For instance, two-level TLB structures in the IOMMU with page walks on the host CPU have been shown to reduce VM overheads to within 6.4% of ideal [28]. This design is similar to our 2M,TLB+PWC implementation which uses large pages to improve TLB reach instead of a level 2 TLB as in the original proposal, and uses the IOMMU PWC. We see that TLBs are not very effective for workloads with irregular access patterns. Moreover, using TLBs greatly increases the energy use of

accelerators.

Particularly for GPGPUs, microarchitecture-specific optimizations such as coalescers have been effective in reducing the address translation overheads [107, 109]. However, these techniques cannot be easily extended for other accelerators. Finally, constraining the virtual-to-physical mapping has been shown to lower address translation overheads, specifically for memory-side processing units [108]. As in DVM, such mappings help break the translate-then-fetch serialization on most memory accesses.

Address Translation for CPUs. Several address translation mechanisms have been proposed for CPUs, which could be extended to accelerators. Coalesced Large-Reach TLBs (CoLT) [106] use eager paging to increase contiguity of memory allocations, and coalesces translation of adjacent pages into each TLB entries. However, address translation remains on the critical path of memory accesses. CoLT can be optimized further with identity mapping and DVM. Cooperative TLB prefetching [13] has been proposed to exploit correlations in translations across multicores. The AVC exploits any correlations among the processing lanes of the accelerator.

Coalescing can also be performed for PTEs to increase PWC reach [11]. This can be applied directly to our proposed AVC design. However, due to our compact page table structure, benefits will only be seen for workloads with much higher memory footprints. Furthermore, page table walks can be expedited by skipping one or more levels of the page table [5]. Translation skipping does not increase the reach of the page table, and is less effective with DVM, as page table walks are not on the critical path for most accesses.

Direct Segments (DS) [6] are efficient but inflexible. Using DS requires a monolithic, eagerly-mapped heap with uniform permissions, whose size is known at startup. On the other hand, DVM individually identity-maps heap allocations as they occur, helping mitigate fragmentation. RMM [75] are more flexible than DS, supporting heaps composed of multiple memory ranges. However, it requires power-hungry hardware (range-TLBs, range-table walkers in addition to TLBs and page-table walkers) thus being infeasible for

accelerators, but could also be optimized with DVM.

5.8 Conclusion

Shared memory is important for increasing the programmability of accelerators. We propose Devirtualized Memory (DVM) to minimize the performance and energy overheads of VM for accelerators. DVM enables almost direct access to PhysM while enforcing memory protection. DVM requires modest OS and IOMMU changes, and is transparent to applications. We also discuss ways to extend DVM throughout a heterogeneous system, to support both CPUs and accelerators with a single approach.

— 6 —

Conclusions and Future Work

It's still magic even if you know how
it's done.

TERRY PRATCHETT

Persistent Memory devices are finally here, and they are highly attractive as they offer low-latency access to a vast capacity of persistent data. In this thesis, we explore the performance and programmability issues introduced by these devices on account of being persistent and vast. To improve performance, we isolate the bottleneck in each case and propose solutions to mitigate or eliminate it. For making such systems easier to program, we provide useful features such as memory protection and virtual address spaces and introduce better hardware primitives to lessen the programmers' burden.

In recoverable applications that leverage the persistence of PM, we find that frequently occurring and expensive ordering operations degrade application performance. To minimize the overheads of ordering operations, we propose a lightweight ordering primitive (*ofence*) along with a separate durability primitive (*dfence*) in Chapter 3. Furthermore, we use insights from a study of PM applications to design efficient hardware to implement these new primitives. Beyond improving performance, our *Hands-Off Persistence System* (HOPS) also simplifies the programming of PM applications. Our new primitives allow programmers to reason about ordering and durability at a high-level and not at the granularity of individual cachelines. Moreover, the HOPS hardware automatically moves data

from the volatile caches to the persistent domain.

In Chapter 4, we propose *Minimally Ordered Durable* (MOD) datastructures that reduce the number of expensive ordering operations in PM applications running on current, unmodified x86-64 hardware. We present a simple recipe to create these MOD datastructures from existing purely functional datastructures. This allows PM application developers to leverage existing research efforts from functional programming domain instead of hand-crafting new recoverable PM datastructures. We implement datastructures commonly used by application programmers such as vector, map, set, stack and queue. Finally, we evaluate these datastructures on real PM—Intel Optane DCPMM—against a state-of-the-art PM-STM implementation and achieve average performance improvements of 60%.

Lastly, we address the rising overheads of virtual memory (VM) in systems with vast memories in Chapter 5. As memory management hardware like TLBs cannot scale proportionally with increasing memory capacities, VM overheads have steadily increased to as much as 50% of application runtime. We study these overheads in accelerators, which are especially vulnerable as they cannot justify allocating resources towards large and power-hungry TLBs. Even so, virtual memory is important for improving the programmability of accelerators and offering memory protection. We propose *De-Virtualized Memory* (DVM) to lower VM overheads by eliminating expensive address translation on most accesses. By allocating memory such that a page’s virtual address matches its physical address, DVM offers almost direct access to PM while providing applications with a virtual address space and memory protection.

Future Work

In this section, we provide some useful directions for future work based on the three contributions of this thesis.

Hands-Off Persistent System

In Chapter 3, we introduced new hardware primitives and the HOPS design. Real PM was not available at the time of this study and so we used simulation to evaluate our idea. One useful starting point for future work would be to repeat the software analysis using newer PM applications on Intel Optane DCPMMs (publicly available from late 2019). Moreover, our simulation infrastructure should also be improved to more-closely model DCPMMs in two ways. First, the simulated PM devices should match DCPMMs in terms of bandwidth and latency. Second, the latency of x86-64 primitives in the simulator should be made comparable to those on real hardware. Of course, further changes may be needed as DCPMMs or other PM technologies move from first to subsequent generations.

The performance of HOPS in a system with multiple memory controllers for PM should be evaluated. PM systems are expected to have multiple memory controllers. Simulating such systems might reveal some issues in the HOPS design, providing a starting point for further optimizations.

Finally, while we focused on applications using PM-STM techniques, HOPS can also support other programming models such as lock-based approaches. While the HOPS primitives should be directly applicable to other models, the hardware design may have to be extended or modified, for example, to track lock-acquire and release operations.

Minimally-Ordered Durable Datastructures

Regarding Chapter 4, one limitation of MOD datastructures is that incomplete FASEs may leak some memory in case of crash. We hypothesize that the memory leak can be fixed by extending functional techniques to the memory allocator as well. If the memory allocation itself used functional shadowing, it should be possible to identify and reclaim the leaked memory.

It would be instructive to look at the performance of more complicated MOD datastructures such as priority queues. Unfortunately, the absence of realistic PM applications

that rely on these datastructures is an impediment to near-term research. Even in our evaluation, we only found applications that used maps and had to use microbenchmarks to evaluate the others.

Programming models for multi-threaded PM applications have not been looked at deeply by us or others in this area. While conventional STM implementations have offered isolation, many PM-STM implementations such as Intel's PMDK [61], JustDo [69], iDO [85] and Romulus [30] do not currently offer isolation and advocate the use of locks. Thus, a challenging next step for MOD datastructures is the development of an effective strategy for supporting concurrent accesses to these datastructures. The Romulus design has shown optimizations such as flat combining [36] and left-right synchronization [112] perform better than simple mutexes in presence of shadow copying techniques for PM applications, and could be a useful starting point.

Finally, we believe that MOD datastructures represent a way to extend PM programming to languages beyond C and C++. Existing functional implementations of datastructures in python and rust could be used to rapidly develop recoverable datastructures in these languages. Such efforts might be useful to jump-start the development of a software ecosystem for PM applications.

Devirtualized Memory

In Chapter 5, we concentrated our efforts on DVM on accelerators and presented an evaluation on a graph-processing accelerator. The DVM design does not exploit any particular trait of the target accelerator and thus can be directly implemented on other accelerators such as machine learning accelerators. Other accelerators may have lesser VM overheads than graph-processing accelerators and require different trade-offs in the DVM design. At the other end of the spectrum, some latency-sensitive accelerators may require beefier DVM hardware to keep DVM overheads low.

We also presented a basic extension of DVM for CPUs along with a preliminary exten-

sion. Having $PA \Rightarrow VA$ may enable other optimizations in the hardware, especially in case of virtual caches. A comprehensive study of DVM-supported hardware optimizations for CPUs would be useful.

Finally, security has recently become a first-class architectural consideration with the emergence of the side-channel attacks such as Meltdown [83], Foreshadow [16] and Spectre variants [77, 54]. Of these, Spectre variants are orthogonal to DVM as the virtual addresses generated during instruction-level speculation have appropriate virtual address permissions. As such, it would be worthwhile to formally prove the security properties of DVM to answer two major questions- whether DVM is vulnerable to Meltdown or Foreshadow and if DVM provides the opportunity for new side-channel attacks.

As persistent and vast memory devices become more widely available and interesting applications are developed, we hope that the techniques and ideas presented in this dissertation provide a firm starting point for future developments.

Bibliography

- [1] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A Scalable Processing-in-memory Accelerator for Parallel Graph Processing," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA)*, 2015.
- [2] AMD, "AMD I/O Virtualization Technology (IOMMU) Specification, Revision 3.00," http://support.amd.com/TechDocs/48882_IOMMU.pdf, Dec. 2016.
- [3] G. M. Amdahl, "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities," in *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference (AFIPS)*, 1967.
- [4] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatarishnan, and S. K. Weeratunga, "The NAS Parallel Benchmarks - Summary and Preliminary Results," in *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing (SC)*, 1991.
- [5] T. W. Barr, A. L. Cox, and S. Rixner, "Translation Caching: Skip, Don't Walk (the Page Table)," in *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA)*, 2010.

- [6] A. Basu, J. Gandhi, J. Chang, M. D. Hill, and M. M. Swift, "Efficient Virtual Memory for Big Memory Servers," in *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA)*, 2013.
- [7] A. Baumann, J. Appavoo, O. Krieger, and T. Roscoe, "A Fork() in the Road," in *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS)*, 2019.
- [8] J. Bennett and S. Lanning, "The Netflix Prize," in *KDD Cup and Workshop in conjunction with KDD, CA*, 2017.
- [9] T. Berning, "NVM Malloc: Memory Allocation for NVRAM," https://github.com/hyrise/nvm_malloc, 2017.
- [10] R. Bhargava, B. Serebrin, F. Spadini, and S. Manne, "Accelerating Two-dimensional Page Walks for Virtualized Systems," in *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2008.
- [11] A. Bhattacharjee, "Large-reach Memory Management Unit Caches," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2013.
- [12] A. Bhattacharjee, D. Lustig, and M. Martonosi, "Shared Last-level TLBs for Chip Multiprocessors," in *Proceedings of the IEEE 17th International Symposium on High Performance Computer Architecture (HPCA)*, 2011.
- [13] A. Bhattacharjee and M. Martonosi, "Inter-core Cooperative TLB for Chip Multiprocessors," in *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2010.
- [14] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC Benchmark Suite: Characterization and Architectural Implications," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2008.

- [15] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The Gem5 Simulator," *SIGARCH Computer Architecture News*, vol. 39, no. 2, Aug. 2011.
- [16] J. V. Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution," in *27th USENIX Security Symposium (USENIX Security)*, 2018.
- [17] J. Carbone, "DRAM prices set to fall," <https://www.sourcetoday.com/supply-chain/dram-prices-set-fall>, 2018.
- [18] A. M. Caulfield, J. Coburn, T. Mollov, A. De, A. Akel, J. He, A. Jagatheesan, R. K. Gupta, A. Snavely, and S. Swanson, "Understanding the Impact of Emerging Non-Volatile Memories on High-Performance, IO-Intensive Computing," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2010.
- [19] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-MAT: A Recursive Model for Graph Mining," in *SIAM International Conference on Data Mining*, 2004. [Online]. Available: <http://www.cs.cmu.edu/~christos/PUBLICATIONS/siam04.pdf>
- [20] D. R. Chakrabarti, H.-J. Boehm, and K. 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)*, 2014.
- [21] J. S. Chase, H. M. Levy, M. J. Feeley, and E. D. Lazowska, "Sharing and Protection in a Single-address-space Operating System," *ACM Transactions on Computer Systems*, vol. 12, no. 4, Nov. 1994.

- [22] T. Chen, B. Xu, C. Zhang, and C. Guestrin, "Training Deep Nets with Sublinear Memory Cost," *arXiv preprint*, 2016. [Online]. Available: <http://arxiv.org/abs/1603.05027>
- [23] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A Spatial Architecture for Energy-efficient Dataflow for Convolutional Neural Networks," in *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA)*, 2016.
- [24] V. Chidambaram, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Optimistic Crash Consistency," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP)*. New York, NY, USA: ACM, 2013. [Online]. Available: <http://doi.acm.org/10.1145/2517349.2522726>
- [25] P. Clarke, "Intel, Micron Launch Bulk-Switching ReRAM," https://www.eetimes.com/document.asp?doc_id=1327289, 2015.
- [26] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson, "NV-Heaps: Making Persistent Objects Fast and Safe with Next-generation, Non-volatile Memories," in *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.
- [27] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee, "Better I/O Through Byte-addressable, Persistent Memory," in *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles (SOSP)*, 2009.
- [28] J. Cong, Z. Fang, Y. Hao, and G. Reinman, "Supporting Address Translation for Accelerator-Centric Architectures," in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2017.
- [29] J. Corbet, "Two more approaches to persistent-memory writes," <https://lwn.net/Articles/731706/>, 2017.

- [30] A. Correia, P. Felber, and P. Ramalhete, "Romulus: Efficient Algorithms for Persistent Transactional Memory," in *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2018.
- [31] G. Cox and A. Bhattacharjee, "Efficient Address Translation for Architectures with Multiple Page Sizes," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.
- [32] G. Cox and A. Bhattacharjee, "Efficient Address Translation for Architectures with Multiple Page Sizes," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.
- [33] cppreference, "Containers Library," <https://en.cppreference.com/w/cpp/container>, 2018.
- [34] T. Davis, "The university of florida sparse matrix collection," <http://www.cise.ufl.edu/research/sparse/matrices>.
- [35] G. Diamos, S. Sengupta, B. Catanzaro, M. Chrzanowski, A. Coates, E. Elsen, J. Engel, A. Hannun, and S. Satheesh, "Persistent RNNs: Stashing Recurrent Weights On-chip," in *Proceedings of the 33rd International Conference on International Conference on Machine Learning (ICML)*, 2016.
- [36] D. Dice, V. J. Marathe, and N. Shavit, "Flat-combining NUMA Locks," in *Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2011.
- [37] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan, "Making Data Structures Persistent," *Journal of Computer and System Sciences*, vol. 38, 1989.
- [38] I. El Hajj, A. Merritt, G. Zellweger, D. Milojevic, R. Achermann, P. Faraboschi, W.-m. Hwu, T. Roscoe, and K. Schwan, "SpaceJMP: Programming with

- Multiple Virtual Address Spaces,” in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. New York, NY, USA: ACM, 2016. [Online]. Available: <http://doi.acm.org/10.1145/2872362.2872366>
- [39] P. Fernando, A. Gavrilovska, S. Kannan, and G. Eisenhauer, “NVStream: Accelerating HPC Workflows with NVRAM-based Transport for Streaming Objects,” in *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, 2018.
- [40] C. J. Fidge, “Timestamps in message-passing systems that preserve the partial ordering,” in *Proceedings of the 11th Australian Computer Science Conference (ACSC)*, 1988.
- [41] K. Fraser and T. Harris, “Concurrent programming without locks,” *ACM Transactions on Computer Systems (TOCS)*, vol. 25, no. 2, May 2007.
- [42] J. Gandhi, A. Basu, M. D. Hill, and M. M. Swift, “BadgerTrap: A Tool to Instrument x86-64 TLB Misses,” *SIGARCH Computer Architecture News*, vol. 42, no. 2, Sep. 2014.
- [43] E. R. Giles, K. Doshi, and P. Varman, “SoftWrAP: A lightweight framework for transactional support of storage class memory,” in *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*, 2015.
- [44] E. Goktas, R. Gawlik, B. Kollenda, E. Athanasopoulos, G. Portokalidis, C. Giuffrida, and H. Bos, “Undermining Information Hiding (and What to Do about It),” in *USENIX Security Symposium*, 2016.
- [45] N. Gonzales, J. Dinh, D. Lewis, N. Gilbert, B. Pedersen, D. Kamalanathan, J. R. Jameson, and S. Hollmer, “An Ultra Low-Power Non-Volatile Memory Design Enabled by Subquantum Conductive-Bridge RAM,” in *2016 IEEE 8th International Memory Workshop (IMW)*, 2016.

- [46] Google Cloud, "Available first on Google Cloud: Intel Optane DC Persistent Memory," <https://cloud.google.com/blog/topics/partners/available-first-on-google-cloud-intel-optane-dc-persistent-memory>, 2019.
- [47] B. Gras, K. Razavi, E. Bosman, H. Bos, and C. Giuffrida, "ASLR on the line: Practical cache attacks on the MMU," *Network and Distributed System Security Symposium (NDSS)*, 2017.
- [48] D. Guniguntala, P. E. McKenney, J. Triplett, and J. Walpole, "The read-copy-update mechanism for supporting real-time applications on shared-memory multiprocessor systems with linux," *IBM Systems Journal*, vol. 47, no. 2, Apr. 2008.
- [49] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi, "Graphicionado: A high-performance and energy-efficient accelerator for graph analytics," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Oct 2016.
- [50] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "EIE: Efficient Inference Engine on Compressed Deep Neural Network," in *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA)*, 2016.
- [51] S. Haria, M. D. Hill, and M. M. Swift, "Devirtualizing Memory in Heterogeneous Systems," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2018.
- [52] J. L. Henning, "SPEC CPU2006 Benchmark Descriptions," *SIGARCH Computer Architecture News*, vol. 34, no. 4, Sep. 2006.
- [53] M. Herlihy, "A methodology for implementing highly concurrent data objects," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 15, no. 5, 1993.
- [54] M. D. Hill, J. Masters, P. Ranganathan, P. Turner, and J. L. Hennessy, "On the Spectre and Meltdown Processor Security Vulnerabilities," *IEEE Micro*, vol. 39, no. 2, March 2019.

- [55] J. Huang, K. Schwan, and M. K. Qureshi, "NVRAM-aware Logging in Transaction Systems," *Proceedings of the VLDB Endowment*, vol. 8, December 2014.
- [56] A. Ilkbahar, "Intel Optane DC Persistent Memory Operating Modes Explained," <https://itpeernetwork.intel.com/intel-optane-dc-persistent-memory-operating-modes/>, 2018.
- [57] E. T. Inc., "Spin-transfer Torque MRAM Technology," <https://www.everspin.com/spin-transfer-torque-mram-technology>.
- [58] M. Inc., "SmartHeap and SmartHeap MC," <http://microquill.com/smartheap/>, 2011.
- [59] Intel, "Intel Optane technology," <https://www.intel.com/content/www/us/en/architecture-and-technology/intel-optane-technology.html>.
- [60] Intel, "New release of PMDK," <https://pmem.io/2018/10/22/release-1-5.html>.
- [61] Intel, "Persistent Memory Development Kit," <http://pmem.io/pmdk>.
- [62] Intel, "PMDK issues: introduce hybrid transactions," <https://github.com/pmem/pmdk/pull/2716>.
- [63] Intel, "Deprecating the PCOMMIT instruction," <https://software.intel.com/en-us/blogs/2016/09/12/deprecate-pcommit-instruction>, 2016.
- [64] intel, "Intel® Virtualization Technology for Directed I/O, Revision 2.4," <https://www.intel.com/content/dam/www/public/us/en/documents/product-specifications/vt-directed-io-spec.pdf>, 2016.
- [65] Intel, "5-Level paging and 5-Level EPT," https://software.intel.com/sites/default/files/managed/2b/80/5-level_paging_white_paper.pdf, May 2017.

- [66] Intel, "Intel Optane DC Persistent Memory Readies for Widespread Deployment," <https://newsroom.intel.com/news/intel-optane-dc-persistent-memory-readies-widespread-deployment>, 2018.
- [67] Intel, "Intel Optane DC Persistent Memory," <https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html>, 2019.
- [68] International Roadmap for Devices and Systems, "More Moore," 2017.
- [69] J. Izraelevitz, T. Kelly, and A. 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)*, 2016.
- [70] J. Izraelevitz, J. Yang, L. Zhang, J. Kim, X. Liu, A. Memaripour, Y. J. Soh, Z. Wang, Y. Xu, S. R. Dullloor, J. Zhao, and S. Swanson, "Basic Performance Measurements of the Intel Optane DC Persistent Memory Module," *arXiv preprint*, 2019. [Online]. Available: <http://arxiv.org/abs/1903.05714>
- [71] S. H. Jo, T. Kumar, S. Narayanan, W. D. Lu, and H. Nazarian, "3D-stackable crossbar resistive memory based on Field Assisted Superlinear Threshold (FAST) selector," in *2014 IEEE International Electron Devices Meeting (IEDM)*, 2014.
- [72] A. Joshi, V. Nagarajan, M. Cintra, and S. Viglas, "Efficient Persist Barriers for Multi-cores," in *Proceedings of the 48th International Symposium on Microarchitecture (MICRO)*, 2015.
- [73] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-I. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony,

- K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omer-nick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-Datacenter Performance Analysis of a Tensor Processing Unit," in *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA)*, 2017.
- [74] V. Karakostas, J. Gandhi, A. Cristal, M. D. Hill, K. S. McKinley, M. Nemirovsky, M. M. Swift, and O. S. Unsal, "Energy-efficient address translation," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, March 2016.
- [75] V. Karakostas, J. Gandhi, F. Ayar, A. Cristal, M. D. Hill, K. S. McKinley, M. Nemirovsky, M. M. Swift, and O. Ünsal, "Redundant Memory Mappings for Fast Access to Large Memories," in *Proceedings of the 42Nd Annual International Symposium on Computer Architecture (ISCA)*, 2015.
- [76] A. H. Karp and H. P. Flatt, "Measuring Parallel Processor Performance," *Communications of the ACM (CACM)*, vol. 33, May 1990.
- [77] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre Attacks: Exploiting Speculative Execution," <https://spectreattack.com/spectre.pdf>, 2017.
- [78] E. J. Koldinger, J. S. Chase, and S. J. Eggers, "Architecture Support for Single Address Space Operating Systems," in *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, New York, NY, USA, 1992. [Online]. Available: <http://doi.acm.org/10.1145/143365.143508>

- [79] A. Kolli, S. Pelley, A. Saidi, P. M. Chen, and T. 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)*, 2016.
- [80] A. Kolli, J. Rosen, S. Diestelhorst, A. Saidi, S. Pelley, S. Liu, P. M. Chen, and T. F. Wenisch, "Delegated Persist Ordering," in *The 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016.
- [81] A. Kolli, S. Pelley, A. Saidi, P. M. Chen, and T. F. Wenisch, "Persistence programming 101," in *Non-volatile Memory Workshop (NVMW)*, 2015.
- [82] D. Li, J. S. Vetter, G. Marin, C. McCurdy, C. Cira, Z. Liu, and W. Yu, "Identifying Opportunities for Byte-Addressable Non-Volatile Memory in Extreme-Scale Scientific Applications," in *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium (IPDPS)*, 2012.
- [83] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown," <https://meltdownattack.com/meltdown.pdf>, 2017.
- [84] M. Liu, M. Zhang, K. Chen, X. Qian, Y. Wu, W. Zheng, and J. 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)*, 2017.
- [85] Q. Liu, J. Izraelevitz, S. K. Lee, M. L. Scott, S. H. Noh, and C. Jung, "iDO: Compiler-Directed Failure Atomicity for Nonvolatile Memory," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018.
- [86] S. Liu, Y. Wei, J. Zhao, A. Kolli, and S. Khan, "PMTTest: A Fast and Flexible Testing Framework for Persistent Memory Programs," in *Proceedings of the Twenty-Fourth*

International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2019.

- [87] Y. Lu, J. Shu, and L. Sun, "Blurred Persistence: Efficient Transactions in Persistent Memory," *Trans. Storage*, vol. 12, no. 1, Jan. 2016. [Online]. Available: <http://doi.acm.org/10.1145/2851504>
- [88] Y. Lu, J. Shu, L. Sun, and O. Mutlu, "Loose-Ordering Consistency for Persistent Memory," in *IEEE International Conference on Computer Design (ICCD)*, 2014.
- [89] F. Mattern, "Virtual Time and Global States of Distributed Systems," in *Parallel and Distributed Algorithms*, 1988.
- [90] M. M. Michael and M. L. Scott, "Simple, fast, and practical non-blocking and blocking concurrent queue algorithms," in *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing (PODC)*, 1996.
- [91] H. A. Moghaddam, Y. H. Son, J. H. Ahn, and N. S. Kim, "Chameleon: Versatile and practical near-DRAM acceleration architecture for large memory systems," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Oct 2016.
- [92] N. Muralimanohar, R. Balasubramonian, and N. Jouppi, "Optimizing NUCA Organizations and Wiring Alternatives for Large Caches with CACTI 6.0," in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2007.
- [93] R. C. Murphy, K. B. Wheeler, B. W. Barret, and J. A. Ang, "Introducing the Graph 500," in *Cray User's Group (CUG)*, 2010.
- [94] S. Nalli, S. Haria, M. D. Hill, M. M. Swift, H. Volos, and K. Keeton, "An Analysis of Persistent Memory Use with WHISPER," in *Proceedings of the Twenty-Second Inter-*

- national Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.
- [95] F. Nawab, J. Izraelevitz, T. Kelly, C. B. M. III, D. R. Chakrabarti, and M. L. Scott, "Dalí: A Periodically Persistent Hash Map," in *31st International Symposium on Distributed Computing (DISC)*, 2017.
- [96] R. Newman and J. Tseng, "Cloud Computing and the Square Kilometre Array," http://www.skatelescope.org/uploaded/8762_134_Memo_Newman.pdf, 2011.
- [97] S. Newsroom, "Samsung Begins Mass Producing Industry's First 16Gb, 64GB DDR4 RDIMM," <https://news.samsung.com/us/samsung-begins-mass-producing-industrys-first-16gb-64gb-ddr4-rdimm/>, 2018.
- [98] J. Nicas, "YouTube Tops 1 Billion Hours of Video a Day, on Pace to Eclipse TV," <https://www.wsj.com/articles/youtube-tops-1-billion-hours-of-video-a-day-on-pace-to-eclipse-tv-1488220851>, 2011.
- [99] T. Nowatzki, V. Gangadhar, N. Ardalani, and K. Sankaralingam, "Stream-Dataflow Acceleration," in *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA)*, 2017.
- [100] M. A. Ogleari, E. L. Miller, and J. Zhao, "Steal but No Force: Efficient Hardware Undo+Redo Logging for Persistent Memory Systems," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018.
- [101] C. Okasaki, "Purely Functional Data Structures," Ph.D. dissertation, Carnegie Mellon University, 1998.

- [102] L. E. Olson, J. Power, M. D. Hill, and D. A. Wood, "Border control: Sandboxing accelerators," in *48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Dec 2015.
- [103] L. E. Olson, S. Sethumadhavan, and M. D. Hill, "Security Implications of Third-Party Accelerators," *IEEE Computer Architecture Letters*, vol. 15, no. 1, Jan. 2016.
- [104] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, G. Parulkar, M. Rosenblum, S. M. Rumble, E. Stratmann, and R. Stutsman, "The Case for RAMClouds: Scalable High-performance Storage Entirely in DRAM," *SIGOPS Oper. Syst. Rev.*, vol. 43, no. 4, Jan. 2010.
- [105] S. Pelley, P. M. Chen, and T. F. Wenisch, "Memory Persistency," in *Proceeding of the 41st Annual International Symposium on Computer Architecture (ISCA)*, 2014.
- [106] B. Pham, V. Vaidyanathan, A. Jaleel, and A. Bhattacharjee, "CoLT: Coalesced Large-Reach TLBs," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2012.
- [107] B. Pichai, L. Hsu, and A. Bhattacharjee, "Architectural Support for Address Translation on GPUs: Designing Memory Management Units for CPU/GPUs with Unified Address Spaces," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.
- [108] J. Picorel, D. Jevdjic, and B. Falsafi, "Near-Memory Address Translation," in *26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2017.
- [109] J. Power, M. D. Hill, and D. A. Wood, "Supporting x86-64 address translation for 100s of GPU lanes," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2014.
- [110] J. P. B. Puente, "Persistence for the Masses: RRB-vectors in a Systems Language," *Proceedings of the ACM on Programming Languages*, vol. 1, September 2017.

- [111] A. Raad and V. Vafeiadis, "Persistence Semantics for Weak Memory: Integrating Epoch Persistency with the TSO Memory Model," *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, 2018.
- [112] P. Ramalhete and A. Correia, "Brief Announcement: Left-Right - A Concurrency Control Technique with Wait-Free Population Oblivious Reads," 2015.
- [113] P. Ranganathan, "From Microprocessors to Nanostores: Rethinking Data-Centric Systems," *Computer*, Jan 2011.
- [114] RedHat, "Position Independent Executables (PIE)," <https://access.redhat.com/blogs/766093/posts/1975793>, 2012.
- [115] J. Ren, J. Zhao, S. Khan, J. Choi, Y. Wu, and O. Mutlu, "ThyNVM: Enabling Software-transparent Crash Consistency in Persistent Memory Systems," in *Proceedings of the 48th International Symposium on Microarchitecture (MICRO)*. New York, NY, USA: ACM, 2015. [Online]. Available: <http://doi.acm.org/10.1145/2830772.2830802>
- [116] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, "Return-Oriented Programming: Systems, Languages, and Applications," *ACM Trans. Inf. Syst. Secur.*, Mar. 2012.
- [117] P. Rogers, "The programmer's guide to the apu galaxy," 2011.
- [118] S. Sarkar, P. Sewell, J. Alglave, L. Maranget, and D. Williams, "Understanding POWER Multiprocessors," in *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2011.
- [119] N. Satish, N. Sundaram, M. M. A. Patwary, J. Seo, J. Park, M. A. Hassaan, S. Sengupta, Z. Yin, and P. Dubey, "Navigating the Maze of Graph Analytics Frameworks Using Massive Graph Datasets," in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2014.

- [120] P. Sewell, S. Sarkar, S. Owens, F. Z. Nardelli, and M. O. Myreen, "X86-TSO: A Rigorous and Usable Programmer's Model for x86 Multiprocessors," *Communications of the ACM (CACM)*, vol. 53, no. 7, 2010.
- [121] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh, "On the Effectiveness of Address-space Randomization," in *Proceedings of the 11th ACM Conference on Computer and Communications Security (CCS)*, 2004.
- [122] S. Shin, S. K. Tirukkovalluri, J. Tuck, and Y. Solihin, "Proteus: A Flexible and Fast Software Supported Hardware Logging Approach for NVM," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2017.
- [123] S. Shin, J. Tuck, and Y. Solihin, "Hiding the Long Latency of Persist Barriers Using Speculative Execution," in *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA)*, 2017.
- [124] K. A. Shutemov, "5-level paging," <https://lwn.net/Articles/708526/>, Jan. 2005.
- [125] I. L. Stats. Twitter Usage Statistics. <http://www.internetlivestats.com/twitter-statistics/> (Visited on 2018-4-29).
- [126] M. Steindorfer, "Efficient Immutable Collections," Ph.D. dissertation, University of Amsterdam, 2017.
- [127] M. J. Steindorfer and J. J. Vinju, "Optimizing Hash-array Mapped Tries for Fast and Lean Immutable JVM Collections," in *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2015.
- [128] Z. D. Stephens, S. Y. Lee, F. Faghri, R. H. Campbell, C. Zhai, M. J. Efron, R. Iyer, M. C. Schatz, S. Sinha, and G. E. Robinson, "Big Data: Astronomical or Genomical?" *PLOS Biology*, vol. 13, no. 7, pp. 1–11, 07 2015. [Online]. Available: <https://doi.org/10.1371/journal.pbio.1002195>

- [129] D. B. Strukov, Snider, G. S., D. R. Stewart, and R. S. Williams, "The missing memristor found," *Nature*, vol. 453, 2008.
- [130] N. Stucki, T. Rompf, V. Ureche, and P. Bagwell, "RRB Vector: A Practical General Purpose Immutable Sequence," in *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2015.
- [131] M. Talluri, S. Kong, M. D. Hill, and D. A. Patterson, "Tradeoffs in Supporting Two Page Sizes," in *Proceedings of the 19th Annual International Symposium on Computer Architecture (ISCA)*, 1992.
- [132] I. L. Taylor, "Split Stacks in GCC," <https://gcc.gnu.org/wiki/SplitStacks>, Feb. 2011.
- [133] J. R. Tramm, A. R. Siegel, T. Islam, and M. Schulz, "XSbench - The Development and Verification of a Performance Abstraction for Monte Carlo Reactor Analysis," in *The Role of Reactor Physics toward a Sustainable Future (PHYSOR)*, Kyoto, 2014.
- [134] M. Tyson, "Intel Optane DC persistent memory starts at 6.57 per GB," <https://hexus.net/tech/news/storage/129284-intel-optane-dc-persistent-memory-starts-657-per-gb/>, 2019.
- [135] A. van de Ven, "Linux patch for virtual address space randomization," <https://lwn.net/Articles/120966/>, Jan. 2005.
- [136] S. Venkataraman, N. Tolia, P. Ranganathan, and R. H. Campbell, "Consistent and Durable Data Structures for Non-volatile Byte-addressable Memory," in *Proceedings of the 9th USENIX Conference on File and Storage Technologies (FASE)*, 2011.
- [137] V. Viswanathan, "Intel Memory Latency Checker v3.6," <https://software.intel.com/en-us/articles/intelr-memory-latency-checker>, December 2018.
- [138] P. Vogel, A. Marongiu, and L. Benini, "Lightweight Virtual Memory Support for Many-core Accelerators in Heterogeneous Embedded SoCs," in *Proceedings of the 10th*

- International Conference on Hardware/Software Codesign and System Synthesis (CODES)*, 2015.
- [139] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: Lightweight Persistent Memory," in *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.
- [140] S. Vongehr and X. Meng, "The missing memristor has not been found," *Scientific Reports*, vol. 5, 2015.
- [141] D. Weston and M. Miller, "Windows 10 Mitigation Improvements," <https://www.blackhat.com/docs/us-16/materials/us-16-Weston-Windows-10-Mitigation-Improvements.pdf>, 2016.
- [142] M. Wilcox, "DAX: Page cache bypass for filesystems on memory storage," <https://lwn.net/Articles/618064/>, 2014.
- [143] E. Witchel, J. Cates, and K. Asanović, "Mondrian Memory Protection," in *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2002.
- [144] H.-S. P. Wong, S. Raoux, S. Kim, J. Liang, J. P. Reifenberg, B. Rajendra, M. Asheghi, and K. E. Goodson, "Phase Change Memory," *Proceedings of the IEEE*, vol. 98, no. 12, Dec 2010.
- [145] L. Wu, A. Lottarini, T. K. Paine, M. A. Kim, and K. A. Ross, "Q100: The Architecture and Design of a Database Processing Unit," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.
- [146] S. L. Xi, O. Babarinsa, M. Athanassoulis, and S. Idreos, "Beyond the Wall: Near-Data Processing for Databases," in *Proceedings of the 11th International Workshop on Data Management on New Hardware (DAMON)*, 2015.

- [147] J. Yang, Q. Wei, C. Chen, C. Wang, K. L. Yong, and B. 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)*, 2015.
- [148] L. Yen, J. Bobba, M. R. Marty, K. E. Moore, H. Volos, M. D. Hill, M. M. Swift, and D. A. Wood, "LogTM-SE: Decoupling Hardware Transactional Memory from Caches," in *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture (HPCA)*. Washington, DC, USA: IEEE Computer Society, 2007. [Online]. Available: <http://dx.doi.org/10.1109/HPCA.2007.346204>
- [149] J. Zhao, S. Li, D. H. Yoon, Y. Xie, and N. P. Jouppi, "Kiln: Closing the Performance Gap Between Systems with and Without Persistence Support," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. New York, NY, USA: ACM, 2013. [Online]. Available: <http://doi.acm.org/10.1145/2540708.2540744>