Synthesis Lectures on
Computer Architecture

Vaibhav Gogte · Aasheesh Kolli · Thomas F. Wenisch

A Primer on Memory Persistency

A Primer on Memory Persistency

Synthesis Lectures on Computer Architecture

A Primer on Memory Persistency

Vaibhav Gogte, Aasheesh Kolli, and homas F. Wenisch

ISBN: 978-3-031-79193-2 paperback
ISBN: 978-3-031-79205-2 PDF

DOI 10.1007/978-3-031-79205-2

A Publication in the Springer series
SYNTHESIS LECTURES ON COMPUTER ARCHITECTURE

Lecture #58
Series Editor: Natalie Enright Jerger, *University of Toronto*
Editor Emerita: Margaret Martonosi, *Princeton University*
Founding Editor Emeritus: Mark D. Hill, *University of Wisconsin, Madison*
Series ISSN
Print 1935-3235 Electronic 1935-3243

A Primer on Memory Persistency

Vaibhav Gogte
University of Michigan

Aasheesh Kolli
Pennsylvania State University

Thomas F. Wenisch
University of Michigan

SYNTHESIS LECTURES ON COMPUTER ARCHITECTURE #58

ABSTRACT

This book introduces readers to emerging persistent memory (PM) technologies that promise the performance of dynamic random-access memory (DRAM) with the durability of traditional storage media, such as hard disks and solid-state drives (SSDs). Persistent memories (PMs), such as Intel's Optane DC persistent memories, are commercially available today. Unlike traditional storage devices, PMs can be accessed over a byte-addressable load-store interface with access latency that is comparable to DRAM. Unfortunately, existing hardware and software systems are ill-equipped to fully avail the potential of these byte-addressable memory technologies as they have been designed to access traditional storage media over a block-based interface. Several mechanisms have been explored in the research literature over the past decade to design hardware and software systems that provide high-performance access to PMs.

Because PMs are durable, they can retain data across failures, such as power failures and program crashes. Upon a failure, recovery mechanisms may inspect PM data, reconstruct state and resume program execution. Correct recovery of data requires that operations to the PM are properly ordered during normal program execution. *Memory persistency models* define the order in which memory operations are performed at the PM. Much like memory consistency models, memory persistency models may be relaxed to improve application performance. Several proposals have emerged recently to design memory persistency models for hardware and software systems and for high-level programming languages. These proposals differ in several key aspects; they relax PM ordering constraints, introduce varying programmability burden, and introduce differing granularity of failure atomicity for PM operations.

This primer provides a detailed overview of the various classes of the memory persistency models, their implementations in hardware, programming languages and software systems proposed in the recent research literature, and the PM ordering techniques employed by modern processors.

KEYWORDS

persistent memory, non-volatile memory, storage-class memory, memory persistency models, strict persistency model, epoch persistency model, strand persistency model, failure atomicity, logging mechanisms

Contents

Preface

Upcoming persistent memory (PM) technologies aim to revolutionize the landscape of future storage systems. These memory technologies promise to deliver near-DRAM performance coupled with the non-volatility of traditional storage media, such as hard disks and solid-state drives (SSDs). PMs provide a byte-addressable load-store interface with access latency similar to DRAM, unlike hard disks and SSDs, which can only be accessed over a block-based interface. Unfortunately, existing hardware, software, and programming systems are ill-equipped to utilize the complete potential of these byte-addressable storage technologies, as they have been designed over generations to access storage over a block-based interface. Existing systems require an expensive software layer to access hard disks and SSDs. While the software layer introduces negligible overheads relative to the inherent latency of accessing a hard disk or SSD, the overheads are prohibitive relative to much faster PMs. Several mechanisms have emerged in the research literature over the past decade and are recently being employed by CPU vendors to design hardware and software systems that provide high-performance access to PMs.

PMs, such as Intel's Optane DC persistent memories, are commercially available today. Because PMs are durable, they can retain data across failures, such as power failures and program crashes. Upon a failure, recovery mechanisms can inspect the data stored in PM, use it to reconstruct the application state, and resume program execution. Correct data recovery requires that the operations to the PM are properly ordered during normal program execution. Unfortunately, ordering PM operations is complicated in modern processor systems by the volatile cache hierarchy and various buffers throughout the memory system. These hardware mechanisms reorder, coalesce, and elide memory operations, which complicate their ordering at the PM. For example, write-back caches may lazily drain updates to the PM (e.g., when cacheline conflicts occur), thereby reordering the updates relative to the original program order. On a failure, the volatile state in the caches is lost. Recovery mechanisms may then observe unintentional reordering of memory operations in the PM post-failure.

Memory persistency models guarantee ordering of PM operations. Memory persistency models are analogous to memory consistency models, which define the visibility order of memory operations to shared memory. Similarly, memory persistency models define the allowable order in which memory operations are performed at the PM. Several memory persistency models have been introduced in the literature. Some proposals have been defined as extensions to the hardware ISA and others to the semantics of high-level programming languages. The proposals differ in several key aspects: they relax PM ordering constraints in different ways, introduce varying programmability burden, and introduce differing granularity of failure atomicity for PM operations. This primer provides a detailed overview of the various classes of memory persistency models, their implementations in hardware, programming languages, and software systems proposed in the recent research literature, and the PM ordering techniques employed by modern processor systems. We organize this primer in six chapters, with each chapter detailing different aspects of PM programming.

Chapter 1 provides a brief overview of different technologies that are considered key contenders for designing PMs. It briefly covers Phase Change Memory, Spin Torque Transfer RAM, and Ferroelectric RAM that might be used to construct high-density, low-access-latency PMs. The chapter also provides a performance characterization of the commercially available Intel Optane DC Persistent Memories. PMs can be used in different configurations by storage systems. Modern file systems may access them over a block-based interface by directly replacing hard disks or SSDs with PMs, or file systems may be developed from the ground up and optimized to directly access byte-addressable storage. We cover the trade-offs for these alternatives in Chapter 1.

Chapter 2 discusses the mechanisms proposed by hardware vendors, such as Intel and ARM, to provide PM ordering guarantees. We discuss instruction set extensions introduced by the vendors to durably store and order PM accesses. As discussed earlier, correct recovery requires that memory operations are ordered to the PM. Chapter 2 provides examples to show why ordering is required for correct recovery.

Chapter 3 details different memory persistency models. Like memory consistency models, memory persistency models may be relaxed, albeit at a higher programmability burden, to improve performance. We cover strict and relaxed persistency models, which offer varying ordering constraints on the PM operations. We define these persistency models formally, and also discuss the hardware implementations proposed in the literature that build each of these models.

The persistency models discussed in Chapter 3 assume atomicity for individual updates (e.g., 8-byte updates in Intel x86). That is, in the case of a failure, either the entirety of the update is applied to the PM or the memory location retains its original value. Recovery mechanisms build from this foundation to provide failure atomicity for multiple updates. They may construct logging mechanisms to ensure that either all of the updates or none of the updates within a failure-atomic program region are durable across a failure. Chapter 4 describes these logging mechanisms. It also focuses on mechanisms proposed in the research literature to construct failure atomicity in hardware.

Chapter 5 discusses the software mechanisms designed to program persistent memory systems. It describes file systems, software transactions, and programming frameworks designed for PM programming. The testing frameworks aimed at finding memory ordering bugs in the recovery systems designed for PMs are also covered in Chapter 5. Chapter 6 concludes the primer.

This primer assumes that the reader is familiar with the basics of computer architecture, with a basic understanding of hardware caching and the memory hierarchy. We cite relevant works that readers can use to obtain a complete understanding of the architecture details that are orthogonal to the topics discussed in this book. Wherever possible, we show quantitative comparison between different proposals discussed in the research literature.

Vaibhav Gogte, Aasheesh Kolli, and Thomas F. Wenisch
February 2022

Acknowledgments

We would like to thank our reviewer, Jishen Zhao, for her valuable feedback on this book. We would also like to acknowledge our colleagues at Google, notably David Culler, Rama Govindaraju, Jack Humphries, Samira Khan, and Parthasarathy Ranganathan, for their comments on the early draft of this synthesis lecture. Special thanks to Natalie Enright Jerger and Michael Morgan for their endless patience and constant encouragement that helped us finish this manuscript.

Vaibhav Gogte, Aasheesh Kolli, and Thomas F. Wenisch
February 2022

CHAPTER 1

Persistent Memories

1.1 INTRODUCTION

For decades, processor systems have benefited from Moore's Law, doubling clock frequency and transistor density every two years. Silicon performance improvement has spurred numerous innovations in designing faster processing systems, with lower data access latency, at a lower cost, every technology generation. High-performance processor systems have relied on a multi-tiered memory hierarchy for fast data access. The memory tiers closer to the processors provide faster access latency and higher throughput than the tiers farther from the processor.

Hard disks and flash memories occupy the lowest tier in the hierarchy. They provide high device density at a lower per-bit cost. Traditional storage systems are composed of hard disks or Flash, as they are non-volatile and can retain data across power failures. However, these mass storage technologies have 100x-1000x higher access latency than SRAMs and DRAMs [1–4]. The static random access memories (SRAM) and dynamic random access memories (DRAM) have been traditionally used to build the processor cache hierarchy and main memories. The SRAM-based cache hierarchy and DRAMs provide lower latency and higher throughput access to application data.

Unfortunately, the end of Moore's Law is near [5]; the technology scaling that the semiconductor industry has enjoyed for decades is coming to an end. In recent years, DRAM technology has faced major scalability challenges [6, 7] and the cost per GB of DRAM has plateaued [8]. Meanwhile, the demand for DRAM capacity continues to grow, thanks to the surge in high-performance big-data applications [8]. The two conflicting trends have resulted in the daunting memory scalability challenges in data centers.

Persistent memory technologies aim to offer a solution for DRAM scalability challenges. Persistent memories (PMs), also known as storage-class memories or non-volatile memories, are denser and consume less power than DRAM. Several technologies, such as Phase Change Memory, Resistive RAM, and Spin Torque Transfer RAM, have been recently explored for commercial production. Commercial PMs, such as Intel and Micron's 3D XPoint, are readily available today [9]. Cloud vendors have also been providing public offerings with support for Intel's Optane DC PM [10–12]. PMs have performance comparable to DRAM, and can provide a larger memory capacity with a lower cost per GB, owing to their higher density.

PMs exhibit many other useful characteristics that make them attractive to system designers. One of the most important is that PMs are non-volatile, like hard disks and SSDs. Furthermore, PMs are byte-addressable. In contrast to the block-based interface offered by traditional mass storage devices, PMs can be accessed over a load-store interface. The load-store interface allows fine-grained data manipulation and lowers access latency. As such, PMs form a tier between DRAMs and traditional storage media, which combines the performance of DRAM and density, cost, and non-volatility of hard disks.

PMs' byte-addressability has important implications for storage applications. These applications typically access storage via a block-based software layer, implemented in file systems and device drivers. Unfortunately, these software layers add hundreds of microseconds to the data access path. While the layers introduce negligible overhead for accessing HDDs/SSDs that themselves have access latencies of hundreds of microseconds to a few milliseconds (and higher), this overhead can become a primary bottleneck for PMs that can be accessed in less than a microsecond. To fully leverage the performance of PMs, it is important to re-architect storage applications for fast, byte-addressable PM-based storage.

In this chapter, we provide a brief overview of PM technologies. We detail Intel and Micron's Optane DC PM, the only high-volume commercial PM offering at the time of writing this book. This chapter further describes bottlenecks in traditional storage systems and why strategies to address them do not readily apply to PM-based storage. We discuss different organizations for integrating PMs into future software systems. Finally, this chapter discusses vendor support for programming PM systems.

1.2 PERSISTENT MEMORY TECHNOLOGIES

PMs (a.k.a. storage-class memories or non-volatile memories) aim to combine the non-volatility and density of conventional storage media like hard disks with the performance of DRAM. Emerging technologies must meet several criteria to be considered key contenders for designing PMs. They must ensure non-volatility by retaining data for a long duration without any external power. They must have high cell density and provide cost-efficiency comparable to traditional storage technologies, and be cheaper than DRAM. Finally, they must exhibit memory-like byte-addressability and offer sub-microsecond read and write access latencies.

Non-volatile DIMMs (NVDIMMs) [13–16] provide an interesting commercial alternative to conventional non-volatile memory devices, by integrating commodity components, typically, DRAM, NAND Flash, and ultracapacitors, rather than relying on an actual non-volatile memory cell. NVDIMMs use NAND Flash to back up contents of volatile DRAM. Flash is not read or written on the critical path of each access. Rather, on failure, NVDIMMs flush DRAM contents to Flash using reserve power stored in the ultracapacitors. Ultracapacitors charge via the system's power supply and hold sufficient reserve energy to flush all DRAM contents. During normal program execution, NVDIMMs complete load and store operations at DRAM speed. NVDIMM products are available commercially [13, 15] and provide a practical PM alternative today. However, since NVDIMM devices must include enough DRAM for their stated storage capacity, they suffer the same cell density and capacity constraints as DRAMs, and their cost is

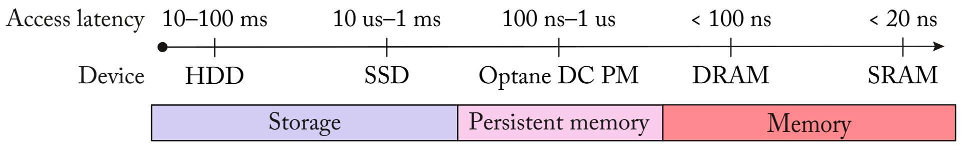


Figure 1.1: Access latency of different memory and storage technologies.

considerably higher than DRAM and much higher than Flash. As such, they are best suited for low-volume and niche applications.

There are several competing PM device technologies. PM technologies, such as Spin Torque Transfer RAM [17], Ferroelectric RAM [18], Resistive RAM (RRAM) [19, 20], and Phase Change Memory (PCM) [21, 22], are byte-addressable, achieve near-DRAM performance, and denser and cheaper than DRAM.

Spin Torque Transfer RAM (STT RAM) records a binary bit in the spin of an electron using a polarizing current in a ferromagnetic substance. Ferroelectric RAM (FeRAM) stores binary data by applying an electric field to polarize states in ferroelectric materials. FeRAM has near-infinite write endurance; it can undergo $> 10^{15}$ write cycles before a cell wears out. The main disadvantage of FeRAMs is their low cell density, which makes them cost inefficient. Resistive RAMs (RRAMs) store binary data in the form of resistive states in metal oxides. RRAMs exhibit lower read and write latencies, and have good write endurance (10^8 cycles [19]), but are yet to find a foothold in the commercial market.

One of the most promising technologies available today for designing commercial PM is Phase Change Memory (PCM). PCM exploits the behavior of chalcogenide glass to transition between amorphous or crystalline state. The phase change material is transformed from one state to another by passing an appropriately shaped electric pulse to modulate the temperature of the cell. The crystalline material melts and switches to its amorphous state when induced with a high-power electric pulse. Similarly, amorphous material may be crystallized by injecting a more moderate electric pulse and heating the material above its crystallizing temperature. The two states differ in their electrical resistivity, which is used to identify the bit stored in the cell.

Because the state of the chalcogenide glass is changed each time the stored bit flips, PCM has disparate read and write latencies. It exhibits read latency of less than 100 ns [21] and write latency on the order of 500 ns [21]. PCM write latency is high because the operation involves melting and recrystallizing the cell. PCM also has fairly good write endurance; it can undergo 10^6–10^8 write cycles before it wears out. PCM performance is similar to DRAM and is cost-efficient and scalable, making it one of the most promising candidates for manufacturing persistent memories commercially.

Exploring each of these technologies in detail is not the focus of this book. We refer readers to the previous synthesis lecture [23] that covers these topics in detail.

Table 1.1: Performance comparison of different memory and storage technologies

Metric	Hard Disk	SSD	DRAM	Optane DC PM
Sequential load latency	$>$ 1ms	100–500 us	$<$ 100 ns	305 ns
Random read latency	$>$ 1 ms	$<$ 1 ms	$<$ 100 ns	169 ns
Write latency	$>$ 1 ms	100–500 us	$<$ 100 ns	94 ns
Peak read bandwidth	$\approx$ 1 GB/s	1–3 GB/s	100 GB/s	6.6 GB/s
Peak write bandwidth	$\approx$ 1 GB/s	0.9–2.5 GB/s	80 GB/s	2.3 GB/s
Source	[1]	[2, 3, 4]	[24]	[24]

1.3 INTEL OPTANE DC PERSISTENT MEMORY

Intel and Micron jointly announced Optane DC PM [9], which leverages 3D XPoint memory technology. Optane DC PMs aim to revolutionize storage media by combining the byte-addressability and performance of DRAMs and non-volatility and density of hard disks and SSDs. They aim to bring storage data closer to compute, accessed over a low-latency, high-bandwidth interface. Figure 1.1 shows the access latencies of different memory and storage technologies.

Optane DC PM is byte-addressable and provides an order of magnitude faster performance than conventional storage media. It provides access latency of less than a microsecond, compared to its storage counterparts that may only be accessed in tens of microseconds or higher. It also provides higher capacity than DRAM. At the time of writing this book, commercial Optane DC PMs provide 128 GB, 256 GB, or 512 GB memory capacity per DIMM [24]. One CPU socket can be installed with up to 6 Optane DIMMs, providing up to 3 TB of memory per socket [24]. As such, Optane DC PMs can enable memory capacities substantially higher than DRAM.

1.3.1 PERFORMANCE OF OPTANE DC PERSISTENT MEMORY

Izraelevitz et al. [24] perform characterization studies to measure performance of Optane DC PM DIMMs. They measure read and write access latency and bandwidth of a 256 GB Optane DC DIMM on a Cascade Lake Intel CPU. Table 1.1 compares performance of storage devices, PM and DRAM, derived from the literature [1–4, 24]. Random-access loads to Optane DC PM incur 305 ns latency, on average, compared to 80 ns read latency to DRAM and $> 10\ \mu s$ read latency to SSDs [2–4, 24]. Spatially sequential loads incur a shorter latency of 169 ns, due to internal buffering and caching in the DIMM. Optane DC PMs provide a peak sequential read bandwidth of around 6.6 GB/s. In comparison, SSDs provide 1–3 GB/s [2–4] read bandwidth.

Measuring write latency is tricky on Optane DC PM-enabled systems. Intel requires platforms with PM to have Asynchronous DRAM Refresh (ADR) support for its memory controllers. That is, the system must have sufficient reserve power to flush write queues in the memory controllers to PM, in case of a power failure. We discuss ADR-support for PM controllers in detail in Section 2.1.2. Nevertheless, any data that reaches PM controllers is considered persistent. Izraelevitz et al. [24] measure write latency to the PM controller. Write operations to Optane DC PM take around 94 ns, comparable to 86 ns write latency in the case of DRAM. Even though PM exhibits lower write latency due to the presence of ADR-supported PM controllers, it has much lower read and write bandwidth in comparison to DRAM. The sequential write bandwidth for Optane DC PM saturates at 2.3 GB/s, higher than SSDs that peak at 0.9–2.5 GB/s [2–4], but much lower than DRAMs.

Izraelevitz et al. [24] also evaluate peak read and write bandwidth with six Optane DIMMs installed on a CPU socket. With interleaved accesses to six Optane DC DIMMs, read and write bandwidths scale to 40 GB/s and 14 GB/s, respectively. The accesses spread evenly across the devices, resulting in the higher bandwidth.

Although PM is faster than traditional storage media, it is still not as fast as DRAM. As shown in Table 1.1, PMs provide an order of magnitude lower read and write bandwidth than DRAM. The access latency, too, is higher than DRAM; load accesses to PM take 4–5× longer than that of DRAM. Due to the differences in performance characteristics of DRAM and PMs, we do not anticipate Optane DIMMs will completely replace DRAM. Instead, the most interesting system configuration integrates Optane DC PMs and DRAM together to provide higher memory capacity. Applications need to carefully design and tune data-structures in a hybrid memory system to exploit the performance and capacity characteristics of the two memory types [25].

1.3.2 MODELING PERSISTENT MEMORY SYSTEMS

Several early research works modeled PM performance characteristics using simulators and emulators due to the lack of access to real hardware. Here, we briefly discuss different simulation and emulation strategies that model PM device characteristics.

PM Emulation Platform (PMEP) [26] and Quartz [27] model PMs as slower DRAM. PMEP builds a custom BIOS that reserves a portion of the attached DRAM as emulated PM. As PM is slower than DRAM, PMEP emulates configurable latency and bandwidth to the emulated PM. It models a higher read latency to PM by periodically stalling the CPU for additional cycles, proportional to an additional access latency that the CPU would encounter on an LLC miss if the DRAM were replaced by PM. Alternatively, Quartz divides program execution into fixed-sized intervals and dynamically injects delays in software at the end of each interval.

PMs also have a significantly lower read and write bandwidth compared to DRAM. Both PMEP and Quartz throttle bandwidth to the emulated PM using a programming feature in the memory controller that limits DRAM transaction throughput on a per-DIMM basis.

Wang et al. [28] improves upon the simulation and emulation strategies proposed by PMEP and Quartz and provides profiling tools and microbenchmarks that may be used to per-

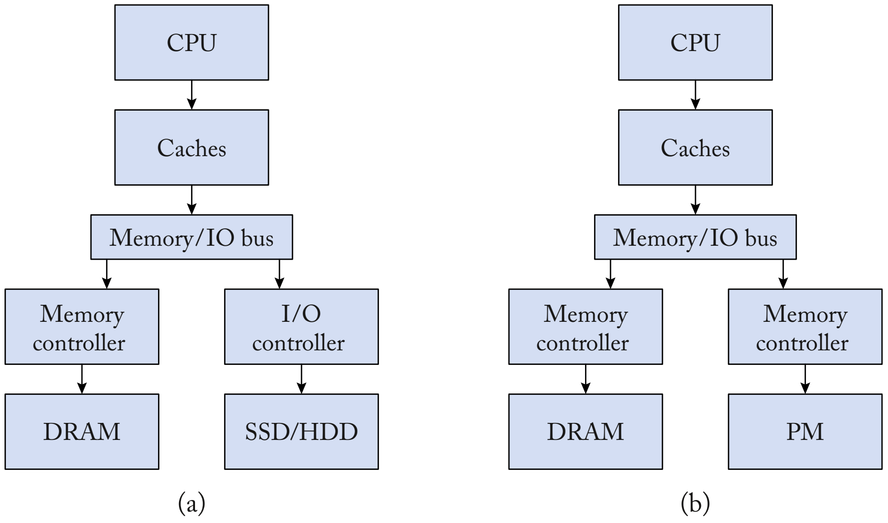


Figure 1.2: (a) Traditional storage system and (b) PM-based storage system.

form a detailed performance characterization of a PM system. They propose LENS, a low-level PM profiler, that can perform a detailed architectural analysis of Optane-based servers and expose architectural characteristics of the underlying memory system (e.g., buffer sizes and control policies in the PM controller). Based on this characterization, LENS builds a Validated cycle-accurate NVRAM Simulator (VANS) that may be used to accurately model a PM system in the simulation frameworks.

1.4 SYSTEM CONFIGURATIONS

PMs enable fast, byte-addressable access to storage. For decades, storage systems relied on file systems and device drivers to read and manipulate persistent data at a block granularity in storage media. This state of affairs is bound to change with the emergence of PMs. This section next outlines how traditional storage systems interface with block-based storage devices, such as hard-disks and SSDs. Then, it outlines different configurations to integrate persistent memories in current and future storage systems.

1.4.1 TRADITIONAL STORAGE SYSTEMS

Figure 1.2a shows a processor system with volatile shared caches and main memory, and block-based storage media, such as hard disk or SSD. In such a system, hardware components, such as processor registers, on-chip buffers, shared caches, and main memory, are volatile. In case of a power failure, these components lose their contents. On the contrary, SSDs and hard disks,

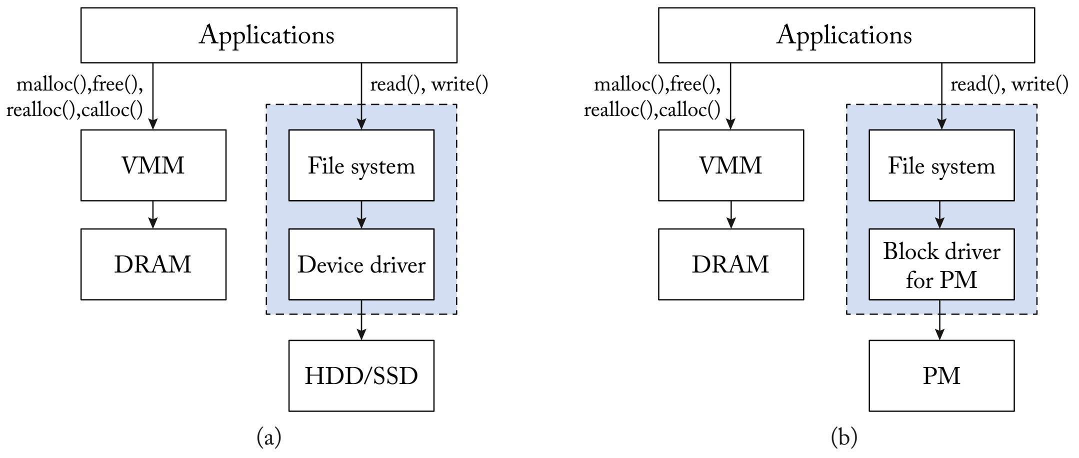


Figure 1.3: (a) File system accessing a traditional storage media and (b) file system accessing PM-based storage.

being persistent, can retain data across failure. The data stored in the storage media are used to recover the system post-failure.

Figure 1.3a outlines a file-system layer for accessing traditional storage. Existing operating systems demarcate accesses to the volatile DRAM and persistent storage. Whereas DRAM can be accessed through processor load and store instructions, operating systems manage accesses to storage through their device drivers. Applications rely on file systems and device drivers to store and retrieve data from hard disks or SSDs. The file systems provide logical abstractions (e.g., files and directories), expose APIs for applications to access file data, and enable safety features to prevent data corruption. On the other hand, device drivers, in the operating system, provide device-specific functionalities, expose a block layer to the file system, and manage I/O to the device. Applications access file data through standard file-system APIs such as open(), close(), read(), and write().

Over the past decades, file systems have been optimized for accessing slower block-based storage media. They build several mechanisms that optimize accesses to traditional storage media (like page caching, serializing accesses, etc.), but these mechanisms may not work well with faster, byte-addressable PM-based storage. We list two key bottlenecks in traditional file systems.

High software overheads. Traditional file systems and device drivers incur high overheads to access storage blocks. In addition to the device access latency, the software stack introduces a few microsecond latency overhead per access to the storage block. As per Table 1.1, SSDs and hard disks exhibit access latencies of 10 μs or higher. For SSDs and hard-disks, this additional software latency is negligible compared to the device latency itself. However, with faster storage media, e.g., persistent memories that exhibit sub-microsecond scale access latency, these software layers become critical to overall data access latency.

Large DRAM footprint. SSDs and hard disks exhibit long access latencies, in the range of a few tens of microseconds to a few milliseconds. Operating systems accelerate access to the storage by caching recently accessed pages in a *page cache* in DRAM. When an application performs a read or write operation, the operating system fetches these pages from storage to the page cache for faster access. Any subsequent reads and writes are performed from and to the page cache. Dirty pages, resulting from application writes, are flushed lazily to storage by the operating system.

Alternatively, operating systems also provide mechanisms to *mmap* file pages directly into applications' virtual address space. The operating system maps applications' virtual address space to file blocks in DRAM. The file pages are fetched into DRAM on subsequent accesses, but applications can directly modify these memory-mapped pages using CPU load and store operations. For instance, applications can issue store operations to update the pages fetched in the volatile DRAM. A subsequent *msync* operation flushes modified pages residing in the DRAM to the storage media.

In both the above scenarios, the operating system expends substantial memory footprint in DRAM to allow faster access to file data in the storage media. The duplication of the data results in the wasted memory resources.

1.4.2 PERSISTENT MEMORY SYSTEM

PMs dramatically change the way systems access storage. Figure 1.2b shows a high-level overview of how future PM-enabled processor systems might look. As PMs are byte-addressable, they can be accessed via a load-store interface similar to DRAM. Unlike conventional storage media that may be accessed only via the OS-managed block interface, CPUs can directly access PMs by issuing load and store operations. This interface avoids expensive software layers required for accessing storage, and allows for the fine-grained manipulation of data.

In Figure 1.2b, volatile components (e.g., CPU registers, caches, DRAM) lose their contents when power fails. PM retains its contents following power interruptions. Thus, PM contents may be used to recover the system, reconstruct the volatile program state, and resume program execution. PMs can be employed by storage systems to store persistent data structures that may be used to recover the system post failure. With the emergence of byte-addressable PMs, storage can be accessed in several ways by future software systems.

- Traditional file systems such as Linux Ext4 and Windows NTFS can build device drivers for PM and expose the standard file-system interface for applications to access storage. Such mechanisms view PMs as faster block-based storage devices, but entirely ignore their memory-like aspects.

- PM-optimized file systems can bypass the block layer completely and provide direct access to PM. Such solutions leverage the byte-addressability of PMs and expose a standard file-system interface to applications to access storage.

- A virtual memory manager can provide native support that allows applications to directly access PMs through a load-store interface. This solution enables direct access to PM and bypasses file-system overheads, but requires invasive changes to application software that manages PM data-structures.

We explore each of the three system configurations in detail below.

1.4.3 PERSISTENT MEMORY SUPPORT IN EXISTING FILE SYSTEMS

The device driver in traditional file systems like Linux Ext4 and Windows NTFS can provide support for PMs. Figure 1.3b shows a high-level overview of a file system that uses PM as a fast block-based storage device. The operating system's device drivers use PMs as a block-based storage device. Conventional file systems rely on a block layer, similar to that in SSDs and hard disks, to store file data. They expose a standard file-system interface for applications to perform reads and writes to the files. As file systems provide the same interface, legacy applications can use PMs without any changes to achieve performance improvement over traditional storage media.

Such solutions discard the memory-like behavior of PMs, hiding it behind a block-device interface. Unfortunately, conventional file systems were designed for slow storage devices that exhibit access latency of tens of microseconds or higher. Software overheads in file systems and device drivers that add a few microsecond overhead to storage access are insignificant for slower storage media. These microsecond-scale software overheads are unacceptable for PMs that can be accessed in less than a microsecond.

File systems also accelerate access to storage by caching recently accessed pages in DRAM, as we explain in Section 5.1. This caching requires frequent memory copies between DRAM and PM, and demands additional footprint for mapping a page cache in DRAM. These mechanisms are redundant for PM, that itself provides memory-like properties. Smarter solutions can provide direct access to byte-addressable PMs, bypass bulky software layers, and entirely eliminate page cache in DRAM.

1.4.4 PERSISTENT MEMORY FILE SYSTEMS

File systems like PMFS [29] and NOVA [30] are optimized from the ground up for PMs to bypass the block device driver and directly access the byte-addressable storage. As shown in Figure 1.4a, these file systems still expose the standard interface for applications to read and write file data in PM, requiring minimal changes to existing applications. They optimize file-system calls for PM, remove unnecessary software overheads, and avoid expensive memory copies be-

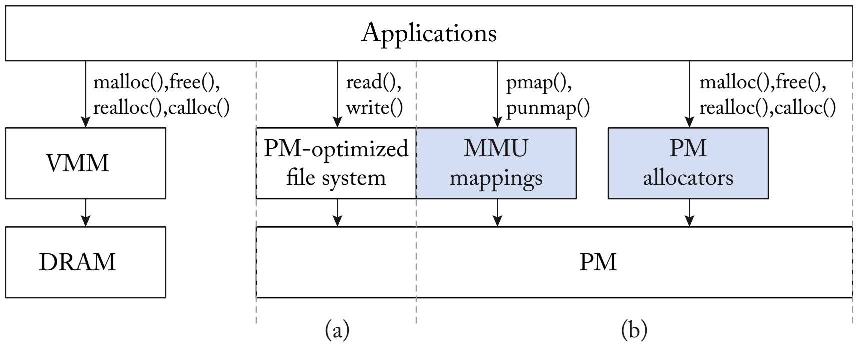


Figure 1.4: (a) PM-optimized file system and (b) mechanisms that enable direct access to PM.

tween DRAM and PM. These file systems access PMs over the fast byte-addressable load/store interface, unlike conventional block-based file systems.

Conventional file systems perform accesses at a block granularity as explained earlier in Section 5.1. For instance, for each write operation, they fetch a block from the storage device to the DRAM, perform updates, and then flush the data block back to the storage. Alternatively, PM-optimized file systems may perform in-place data manipulation without unnecessary memory copies, at a fine granularity. NOVA [30] is one such example of a file system optimized for PM. It performs up to 10× better than the Linux ext4 file system. We cover PM-optimized file systems in detail later in Chapter 5.

1.4.5 DIRECT ACCESS TO PERSISTENT MEMORIES

With PM-optimized file systems, applications still need to rely on standard file-system APIs to manipulate PM data. However, PMs present a golden opportunity for applications to build custom persistent data structures, manage them in user-space, and manipulate them through CPU loads and stores, similar to volatile data structures in DRAM. Applications can build high-performing, sophisticated data structures in PM that are persistent across power failures.

The direct access (DAX) feature in PM-optimized file systems provides this support, as shown in Figure 1.4b. Similar to Linux mmap system call [31], DAX maps file pages in PM directly into an application's virtual address space. DAX lets applications directly manipulate storage data in PM, unlike mmap that manipulates file pages in DRAM. Any load or store operations performed by an application directly affect PM data.

PM-optimized file systems such as PMFS [29] and NOVA [30] also provide the DAX feature for PMs. Applications may *pmap* file data into its virtual address space for the direct access to PM. Similarly, NV-Heaps [32], Mnemosyne [33], and Makalu [34] provide memory allocators for PMs. As shown in Figure 1.4b, these solutions allow applications to "malloc" and "free" memory in PMs, manage persistent heap, and build custom data structures.

Direct access to PM exposes raw device performance to the applications. However, this also brings forth new kinds of challenges.

As PM is directly mapped in the application's address space, data stored in PM are prone to corruption due to spurious writes and such corruption then is persistent across reboots. Software systems such as PMFS [29] defend against such failures by ensuring that only intended writes occur to PM. Other solutions such as PMTest [35], Janus [36], and Liu et al. [37] provide frameworks to debug and verify software systems and provide secure access to PM. We discuss these mechanisms in detail in Chapter 5.

Applications must manage volatile data in DRAM and persistent data in PM. Any data pointers between DRAM and PM may be tricky to manage. For instance, in case of a power failure, DRAM contents are lost. Post failure, pointers in PM may point to data (now lost) in DRAM. NV-Heaps [32] and Mnemosyne [33] provide software tools to aid programmers in avoiding such scenarios. We detail these implementations in Chapter 5.

As PM data structures are persistent, applications must ensure that they are consistent across power failures. Any intermediate power failure between a sequence of updates may corrupt the underlying data structure. A common technique that ensures data consistency atomically moves a data structure from one state to another. However, it is burdensome for programmers to reason about such failures while designing applications. A wide class of literature is devoted to providing data consistency for PMs. The hardware mechanisms [38–43], programming languages [44–46], and runtime systems [47–51] aid in ensuring that PM data structures are consistent in case of failures. These mechanisms provide abstractions that programmers can leverage to define atomic regions in the program and order memory accesses to PM. A large part of this book is dedicated to describing these mechanisms. We detail these mechanisms in Chapters 4–5.

CHAPTER 2
Data Persistence

In a system with volatile hardware components (like CPU registers, processor caches, etc.), simply issuing a store operation does not guarantee that the update is durably written to PM. For instance, when a store operation is issued by the CPU, modern processors perform the update to a write-back cache. The write-back cache then lazily drains the update to PM, for example, when a cache line conflict occurs. Any subsequent failure erases the volatile write-back cache (or generally, any volatile component) and the update is lost. To prevent data loss on failure, hardware systems must guarantee that the updates reach PM.

In this chapter, we discuss mechanisms recently added by hardware vendors such as Intel and ARM to guarantee data durability in case of a failure.

2.1 ENSURING DATA DURABILITY

Hardware vendors such as Intel and ARM have proposed ISA extensions to facilitate PM programming. ISA vendors provide instructions required to ensure that the data is durably written to PM. The ISA extensions are used to flush data from volatile hardware structures (e.g., write-back caches) and guarantee that the update reaches the *persistence domain*. Persistence domain is the region in the memory hierarchy where store operations may be considered to be persistent. Once a store operation reaches the persistence domain, it is guaranteed to be visible to recovery in case of a power failure.

The PM-enabled hardware systems we have discussed to this point (earlier in Chapter 1) assume that PM alone lies in the persistence domain. That is, any data that resides in components other than PM is lost on failure.

Alternately, systems may detect an imminent power failure and leverage an external power backup to flush data in some or all volatile hardware structures to persistence. The additional backup in the form of an external battery or super capacitors can provide enough power to flush in-flight data. Effectively, backup power can bring the persistence domain closer to the CPU cores. Note that the external source must have enough reserve power to guarantee that it can flush all in-flight data from the hardware components that lie in the persistence domain to PM. Any data that reaches the persistence domain is then considered durable. Of course, any in-flight data that cannot be flushed to PM on failure is lost.

While having an external power backup makes PM programming easier and more performant, there is a clear trade-off between which hardware components may be flushed to PM on failure and the amount of reserve power required to ensure their persistence. Bringing the

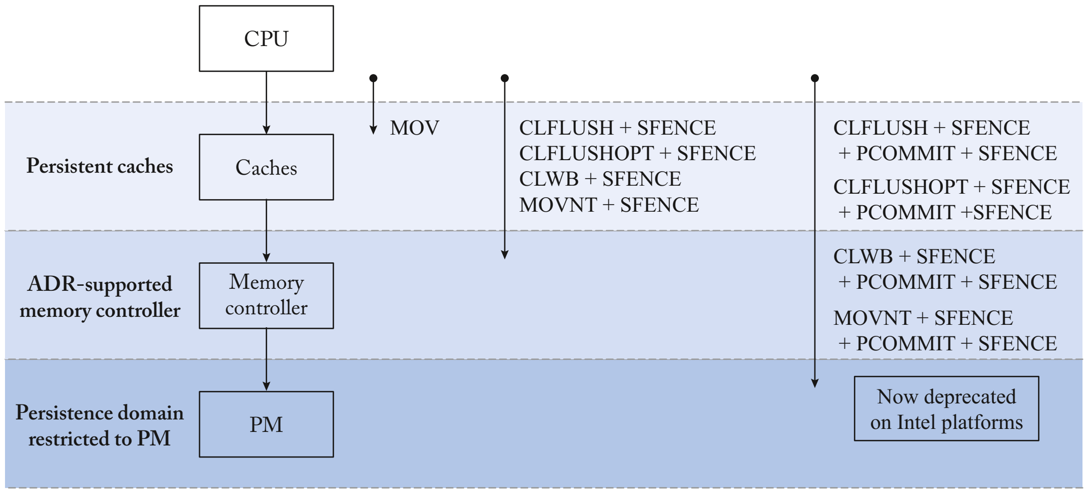

CPU
Persistent caches
Caches
ADR-supported memory controller
Memory controller
Persistence domain restricted to PM
PM
MOV
CLFLUSH + SFENCE
CLFLUSHOPT + SFENCE
CLWB + SFENCE
MOVNT + SFENCE
CLFLUSH + SFENCE + PCOMMIT + SFENCE
CLFLUSHOPT + SFENCE + PCOMMIT +SFENCE
CLWB + SFENCE + PCOMMIT + SFENCE
MOVNT + SFENCE + PCOMMIT + SFENCE
Now deprecated on Intel platforms

Figure 2.1: Persistence domains that guarantee persistence of data in case of a failure. It also outlines Intel x86 instructions required to ensure data persistence.

persistence domain closer to the CPU cores is desirable, but systems require added resources (e.g., bigger batteries) to provide backup for those hardware components, which adds to the total cost of ownership for the system. The following sections show different ways in which the persistence domain may be modeled and how it can affect the PM programming. Figure 2.1 shows different persistence domains for PM systems. It also outlines instructions on Intel x86 platform required to flush data from the volatile domain to each of the persistence domains.

2.1.1 PERSISTENCE AT CPU CACHES

CPU caches, on-chip buffers, and PM controllers may all lie in the persistence domain, provided the system ensures sufficient power backup to flush their contents to PM on failure. On failure, a non-maskable interrupt routine is invoked to flush CPU caches to PM [52]. As CPU caches lie in the persistence domain, a store operation is persistent as soon as the store operation completes in the L1 cache. As shown in Figure 2.1, a `MOV` operation on Intel x86 platforms completes when its update is written to the L1 cache and guarantees that the data is persistent.

Extending the persistence domain to CPU caches assures a simple programming model as completion of the store operation guarantees its durability. Unfortunately, such a system requires a significant amount of backup power to flush large CPU caches, on-chip buffers and the memory controller to PM upon power loss. Such backup power may be in the form of distributed uninterruptable power supply (UPS) [53] or rechargeable lithium-ion batteries in each server rack. The battery source requires additional space in server racks, increases total cost of ownership, and is not environmentally friendly [54]. The rechargeable batteries also demand substantial maintenance as they may sustain only a few hundred charging cycles. Nevertheless, given external power backup, if installation and maintenance costs are justifiable, servers may extend the persistence domain to include hardware caches, on-chip buffers, and PM controllers.

2.1.2 PERSISTENCE AT PM CONTROLLER

The Asynchronous DRAM Refresh (ADR) [55] feature on Intel platforms assures that write-pending queues in the memory controllers are flushed to PM on failure. As shown in Figure 2.1, ADR extends the persistence domain to the PM controllers. With ADR-supported write-pending queues, CPUs need only flush an update to the PM controller to assure its durability; the CPU need not wait for the write to be drained to the PM device to ensure the data will be preserved across failures. Write-back caches and on-chip buffers are volatile; ADR does not flush those components to PM on failure.

Intel x86 provides several instructions that can be used together to flush updates to the PM controller. On recent Intel x86 platforms, CPUs may flush updates from volatile write-back caches to the PM controller using `CLWB`, `CLFLUSHOPT`, or `CLFLUSH` instructions. A cache-line write-back (`CLWB`) instruction flushes an update while retaining a clean copy of the cache line in the write-back cache. An optimized cache flush (`CLFLUSHOPT`) instruction is similar to `CLWB`, except that it invalidates the cache line as it flushes it to the PM controller. Although both `CLWB` and `CLFLUSHOPT` may be used for data persistence, the `CLWB` instruction enables better performance by retaining a copy of data in the caches. As such, the `CLWB` instruction retains temporal locality of application accesses, as any subsequent memory operations to the same location result in a cache hit.

`CLWB` and `CLFLUSHOPT` alone do not ensure data durability. For instance, any ongoing `CLWB` operation may be lost on power failure, if the flush operation does not reach the ADR-supported PM controller. A subsequent `SFENCE` instruction is required to guarantee completion of preceding `CLWB` operations, as shown in Figure 2.1. Thus, execution of a `CLWB+SFENCE` (or a `CLFLUSHOPT+SFENCE`) sequence assures that an update is durably written to the PM controller.

Similar to `CLFLUSHOPT`, `CLFLUSH` flushes dirty cache line to the PM controller and invalidates the cache line. A subsequent access to the same memory location results in a compulsory cache miss. However, unlike `CLFLUSHOPT`, `CLFLUSH` implements an implicit fence operation that guarantees the flush operation completes before any subsequent memory operations are issued.

Alternatively, non-temporal stores (`MOVNT`) may also be employed to durably perform updates. Non-temporal stores bypass volatile caches completely and directly perform updates to the PM controller. As updates are not cached, non-temporal stores fail to exploit temporal locality of the memory accesses. Subsequent memory accesses miss in cache and need to fetch data from PM.

Figure 2.2 shows a performance comparison of the four operations on Intel Optane DC PM, derived from prior work [24]. `CLFLUSH` encodes an implicit fence operation. Thus, it disallows any subsequent `CLFLUSH` operations from being issued and performed in parallel. On the

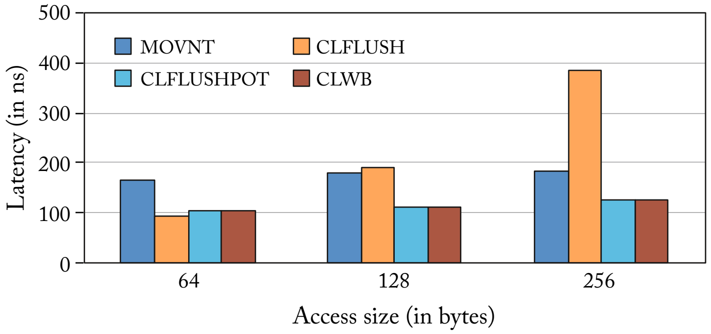


Figure 2.2: Latency of flush operations (derived from [24]) to the ADR-supported PM controller.

contrary, multiple MOVNT, CLWB, and CLFLUSHOPT instructions may be issued in parallel, followed by a single SFENCE that guarantees their persistence. As shown in Figure 2.2, CLFLUSH latency is 3.1× higher than CLWB for larger accesses that involve higher concurrency. While this figure only shows the latency of the flush operation itself, if a workload has a flush followed by a subsequent read to the same cacheline, CLWB considerably outperforms CLFLUSHOPT for such a sequence. Given these observations, PM programs commonly employ CLWB operations to flush multiple cache lines to PM, followed by a single SFENCE for data persistence.

2.1.3 PERSISTENCE AT PERSISTENT MEMORY DEVICE

Without any external power available to flush hardware components, PM alone lies in the persistence domain. On failure, recovery uses data retained by PM to restore the state of the system. The CPU needs to flush data all the way to PM to guarantee that it is persistent.

The Intel x86 ISA had originally planned a PCOMMIT instruction to guarantee that the data is flushed from the memory controller to PM. As shown in Figure 2.1, PCOMMIT could be used along with cache flush instructions (e.g., CLWB) to flush data from volatile caches and memory controllers to the PM. A cache flush + SFENCE operation sequence flushes data from volatile caches to the controller. A subsequent PCOMMIT instruction makes sure that the updates drain from write-pending queues in the PM controller to PM. The second SFENCE operation guarantees completion of the PCOMMIT operation. With only PM in the persistence domain, PCOMMIT or CLWB alone are insufficient to flush data to persistence. The instruction sequence shown in Figure 2.1 is required to flush data from write-back caches to PM.

Unfortunately, flushing data all the way to PM has a high performance penalty. The PCOMMIT+SFENCE instruction sequence stalls the CPU until memory controller drains entirely to PM. As the PM device latency is high, the operation may take a few hundreds of CPU cycles to complete.

The PCOMMIT instruction has since been deprecated [55]. Intel platforms supporting PMs are now required to have ADR-supported memory controllers. Thus, the cache flush instructions are sufficient to flush data to the persistence domain that includes the ADR-supported memory controller and PM. A PCOMMIT instruction is no longer required and is not supported by the x86 ISA.

The Optane DC PMs attach to the same DIMM slots as the DRAMs. However, as Intel requires ADR-supported memory controllers, PM DIMMs are not supported by legacy Intel platforms. They are supported on newer Intel server platforms, starting from Cascade Lake, that provide ADR support for the memory controller.

2.1.4 ARMv8.2 ISA EXTENSIONS

So far, this chapter described Intel x86 ISA extensions required for PM programming. Specifically, it covered extensions to explicitly force data eviction from the volatile to persistence domain.

The ARMv8.2 ISA also provides support for emerging byte-addressable persistent memories [56]. It introduces a DC CVAP instruction that performs a cache flush operation to the *point of persistence*. Unlike Intel x86, the ARM ISA does not assume ADR-like support for its memory controllers. The point of persistence is implementation-defined. It may be restricted to PMs alone, or may comprise PM controllers with reserve power backup. If the persistence domain includes PM alone, the DC CVAP instruction writes back updates to PM, eliminating the need for an additional PCOMMIT-like instruction.

2.2 ORDERING MEMORY OPERATIONS TO PM

The persistent data structures stored in PM may be used to recover the system on failure. Recovery requires that memory operations to PM are ordered. Without proper memory ordering, underlying data structures may be non-recoverable in case of a failure.

2.2.1 WHY IS ORDERING REQUIRED?

Figure 2.3a shows an example that appends a new node to a linked list data structure residing in PM. Assume the linked list comprises two nodes and, initially, the tail pointer points to node 2. An append to the tail pointer involves two operations. First, a new node is created and filled with a desired value. Second, the tail pointer is updated to point to the newly created node.

In a fault-free program execution, the two operations may occur in any order. That is, as shown in Figure 2.3b, the tail node may update to point to a "garbage" node first, followed by filling the node with a valid value. Absent any failure, the final state of the linked list is intact and it comprises the three valid nodes.

However, the ordering of the two operations is crucial when failure is taken into consideration. In Figure 2.3b, in case of an intermediate failure between the two operations, the tail

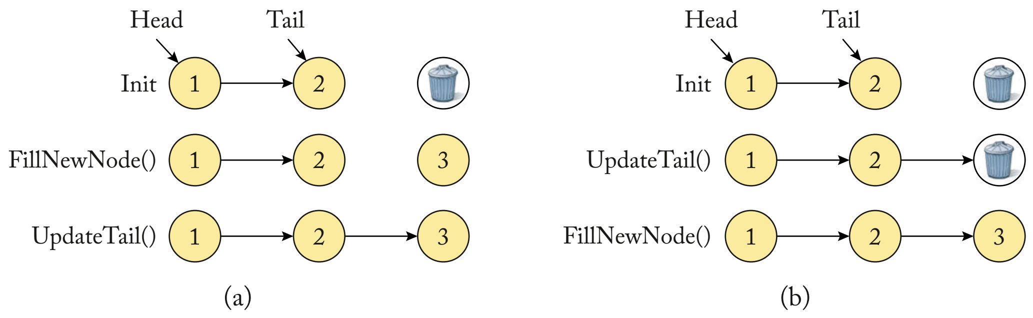


Figure 2.3: Example of an append operation to the linked list that shows why memory operations need to be ordered to PM for correct recovery. (a) An append to the linked list that involves creating a new node followed by an update to the tail pointer that points to the newly created node and (b) a scenario where the two operations reorder; an update to the tail pointer occurs prior to filling the newly created node with a valid value.

points to a garbage node. Due to reordering between the two operations, a failure corrupts the linked list data structure in PM. Post failure, the linked list is unrecoverable.

One important distinction to note here is that, such an ordering constraint is unnecessary when a linked list resides in volatile DRAM. On failure, the entirety of DRAM loses its contents. Thus, any memory reordering that might have happened prior to the failure is lost, and its side-effects are not visible to recovery. With PM, partial updates may be durable, revealing any memory access reordering that might have occurred.

2.2.2 SOURCES OF PM OPERATION REORDERING

There are numerous sources of memory reorderings in modern processor systems, which may be transparent to programmers. Figure 2.4 shows one such potential reordering of operations in a processor system with a volatile core and write-back caches, and durable PM controller (not shown in Figure 2.4 for simplicity) and PM.

Assume the linked list operations in Figure 2.3 map down to two store operations to PM locations. For correct recovery, the store to memory location A must happen before the store to memory location B. Suppose, memory locations A and B lie on different cachelines in the write-back caches. ❶, ❷ The CPU performs the two operations to the write back caches. The two operations complete when cache lines are updated. The dirty cache lines may lazily write back their contents to PM sometime in the future, potentially due to cache line conflicts. ❸ Due to a conflict, the cache writes back update to location B to PM. An update to location A still resides in volatile write-back cache. ❹ Failure occurs, wiping the volatile contents in the cache, including the update to location A. After failure, the update to location B is in PM, while the

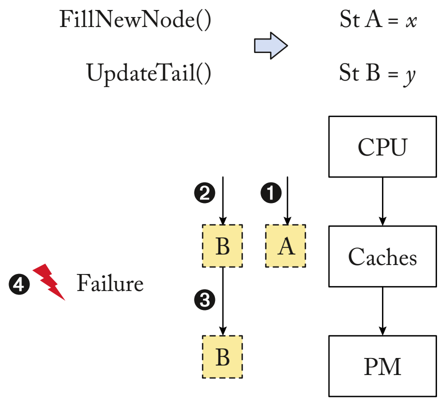


Figure 2.4: The updates to memory locations A and B may reorder to PM, as they drain from write-back caches out of order.

update to A is lost, creating the scenario we previously explained in Figure 2.3b. Thus, memory reordering introduced by write-back caches corrupts the linked list.

Unfortunately, modern processor, compiler and software systems introduce numerous sources of such memory reorderings that are transparent to programmers. Out-of-order cores, buffers and write-back caches may reorder PM operations. Any in-flight operations in volatile on-chip buffers may also reorder memory operations to PM. Compilers include numerous optimizations to boost performance of generated code that reorder instructions, including by reordering memory operations. Several commonly used compiler optimizations, such as constant propagation, loop invariant code motion, and common sub-expression elimination reorder memory operations, transparent to programmers.

One may assume that memory consistency models already define the order on memory accesses, and that programmers may rely on memory ordering guarantees provided by hardware or high-level programming languages to design data-structures for PM. Unfortunately, memory consistency models define only the visibility of memory order in the shared memory system. Ordering constraints that arise from the memory consistency models are enforced by the system to ensure writes are observed in an order consistent with the memory model, and they do not govern the order those writes are applied to the memory device itself. As shown in Figure 2.4, a shared memory system with write-back caches may reorder memory accesses to PM while still providing a strictly ordered consistency model. That is, even if a strict consistency model serializes store operations to the write-back caches, the updates may still drain out-of-order to PM.

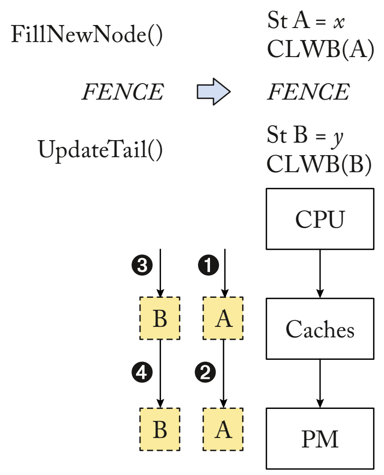


Figure 2.5: A sequence of cache-line write back and fence instructions guarantee that the updates to memory locations A and B reach PM in a correct order.

2.2.3 APPLYING CORRECT MEMORY ORDER TO PM

To design persistent data structures, hardware, compiler, and/or software systems need to apply a correct order on memory accesses to PM. In the example shown in Figure 2.4, the memory reordering between the operations to PM may be prevented by flushing the updates from write-back caches to PM, and ordering the operations using a fence. Figure 2.5 shows the modified code that prevents memory reordering. ❶ The CPU performs a store to the memory location A in the write-back cache. ❷ The CPU issues a cache flush operation (e.g., `CLWB`) to write back the update to location A to PM. A subsequent fence operation ensures that the `CLWB` operation completes. ❸ The CPU performs a store to memory location B after the update to location A is performed in PM. ❹ The CPU flushes update to location B using a `CLWB` operation.

Programmers or compiler mechanisms need to annotate programs with the cache flush and fence operations to prevent such memory reorderings. Correctness of recoverable persistent data structures relies on ensuring correct order on memory operations to PM. The hardware ISA, compiler mechanisms and high-level languages must define and implement mechanisms to enforce ordering constraints. Programmers can rely on the guarantees provided by these implementations to ensure correct ordering on PM operations.

Pelley [57] proposes *memory persistency models*, analogous to memory consistency models, to define the order in which memory operations occur to PM. They provide an interface for programmers to reason about memory ordering to PM. We discuss these models and their implementation in detail in Chapter 3.

CHAPTER 3
Memory Persistency Models

PM data can survive failures due to power interruptions or program crashes. In the case of a failure, recovery code inspects the PM data, reconstructs volatile state, and resumes program execution. Recovery requires that memory operations to the PM are properly ordered. Memory consistency models [58, 59] enforce the order in which memory operations are *visible* in shared memory systems. They specify the allowed shared memory state and observable load values when load and store operations are performed by multiple cores to overlapping memory locations. Unfortunately, these visibility constraints do not apply to the order operations are drained to PM; consistency models do not specify constraints on when an update is sent to the memory device, only on the value that subsequent loads may return (i.e., an update may linger in a write-back cache without draining to PM). As discussed earlier in Chapter 2, write-back caches, on-chip buffers, and processor interconnects may all reorder updates as they drain to PM. However, the order that writes drain to PM determines the memory image that is available to recovery code; hence, correct recovery depends critically on reasoning about the order of the writes to PM.

Memory persistency models define the order memory operations are performed at the PM. Throughout this book, we refer to the act of performing a store operation to PM as a *persist*. Similar to the memory consistency models that enforce visibility of memory operations, memory persistency models (or simply, persistency models) define the order in which operations persist in PM. Persistency models allow programmers to reason about possible PM states in case of a failure. Memory persistency models, unlike their consistency model counterpart, need to specify program behavior for single-core operations as well. Failures can occur during single-threaded program execution; memory persistency models define the legal order of memory load and store operations with respect to post-failure recovery.

3.1 PERSISTENCY MODELS

Pelley and co-authors [57] define persistency models as an extension to the consistency model of the system. Persistency models prescribe ordering requirements on how memory operations persist to PM, similar to how memory consistency models define the order in which memory operations are visible in shared memory. Pelley classifies persistency models in three categories: *strict persistency*, *epoch persistency*, and *strand persistency*. These persistency models differ in their programmability, complexity, and the performance that they enable. For instance, strict persistency is simpler and easier to program, allows intuitive memory ordering in PM, but has high ordering overhead. In contrast, epoch and strand persistency relax the order of PM operations, remove redundant ordering overheads, and enable higher performance, yet are complex and non-intuitive, and impose a high programmability burden. The epoch and strand persistency models are analogous to relaxed consistency models, which impose minimal ordering constraints on memory operations in shared memory, but require complex programming annotations. The persistency models provide a framework for hardware and low-level software to order PM operations. We discuss these models, formalize them, and define their hardware implementations in this chapter.

Specifically, the rest of this chapter covers hardware implementations of several persistency models in detail. In contrast, language-level persistency models build upon hardware/ISA-level persistency models to provide mechanisms that define the semantics of PM accesses as an integral part of the high-level programming language. Language-level persistency models provide an ISA-agnostic framework in high-level languages such as C++ and Java that application developers can use to reason about PM state in the case of failure. Compilers can map high-level language frameworks to the low-level hardware ISA persistency models. We defer discussion of language-level persistency models to Chapter 5.

3.2 RECOVERY OBSERVER

Recovery mechanisms require that persist operations to PM are ordered. In the absence of ordering constraints on persists, persistent data structures are vulnerable to data corruption post-failure. Persistency models prescribe the order of memory operations to PM. Specifically, persistency models define *persist memory order* to specify ordering constraints required for correct recovery. This is analogous to the memory consistency models that introduce constraints on the visibility of the memory operations defined by the *volatile memory order*. The consistency and persistency models coexist to specify different sets of relations on the visibility order and persist order of PM operations. The two models may impose a different set of ordering constraints on memory operations; the persists may be concurrent under the persistency model, but the visibility of the corresponding store operations may be constrained by the consistency model. Similarly, stricter persistency models may serialize the order in which the memory operations persist to PM, but they may occur to the volatile shared memory out of order if allowed by the system's consistency model.

Consistency models define the allowed memory behavior when multiple cores access shared memory locations. Unlike consistency models (that only apply to multi-processor systems), persistency models must define the PM state for single- and multi-core systems: failures may occur in a single-core system and expose any memory reorderings to recovery.

To reason about the PM state after a failure, Pelley introduces the notion of a *recovery observer*, which atomically observes the entire content of the PM upon failure. The recovery observer may be conceptualized as a hypothetical execution thread, which executes concurrently to pre-failure execution on another processor and may observe the order of persists to PM. The ordering constraints on persists required for correct recovery imply a persist order with respect to the recovery observer. These persists are ordered in PM by the persistency model. In contrast, persists that are not ordered with respect to the recovery observer may be issued concurrently under the persistency model. These persists may drain to the PM in an arbitrary order.

Unlike consistency models, which apply only to multiprocessor systems, the notion of the recovery observer extends to both uniprocessor and multiprocessor systems. Any persists performed by any core may form a data race with the conceptual observations of the recovery observer. In case of a failure, the recovery observer may observe violation of ordering constraints required for correct recovery. Single- and multi-core systems that operate on PM must follow the ordering constraints defined by the persistency model.

The persistency model abstracts the persist ordering requirements needed for correct recovery from their actual implementation. It prescribes a set of ordering guarantees that high-level programs can rely on; hardware implementations can enable optimizations (e.g., persist reordering and coalescing) as long as they correctly honor the ordering requirements specified by the persistency model. Persistency models can be categorized into strict, epoch, and strand persistency models. Strict persistency provides an intuitive design, but comes at a high performance cost. Like relaxed consistency models, epoch and strand persistency models trade off simplicity to enable greater persist concurrency (and thereby, higher performance). We describe each of these models in detail next.

3.3 STRICT PERSISTENCY

Strict persistency tightly couples the persist order with the visibility order of memory operations as defined by the consistency model. That is, the persist memory order (PMO) is identical to the visibility order of the memory operations. In other words, PM persist operations precisely follow their visibility order in the shared memory system. Strict persistency provides an intuitive persist memory order. Any memory operations that are ordered through the consistency model must also persist in the same order. Thus, programmers are not required to reason about the persist order separately; existing hardware/programming abstractions that order the visibility of memory operations also establish the persist order. Unfortunately, the strict persistency model trades off simplicity for performance, especially in under strict consistency models.

Under the strict persistency model, persists inherit the visibility order of stores. Conservative consistency models, such as sequential consistency (SC) or total store order (TSO), serialize visibility of stores to shared memory. In a system that implements an SC or a TSO consistency model, the strict persistency model couples the persist order with their visibility, thus serializing all stores to PM. This serialization introduces frequent CPU pipeline stalls during execution; subsequent persists stall until prior persists drain completely to PM. The strict persistency model, with SC or TSO consistency, disables several common hardware performance optimizations that reorder or coalesce updates. For instance, modern CPUs exploit temporal and spatial locality of memory accesses (e.g., in write-back caches) by coalescing or reordering multiple updates to the same memory location before writing back the updates to PM. Under strict persistency, as persists are serialized, such performance optimizations are disallowed.

Systems that implement relaxed consistency models (e.g., ARMv8 consistency model and RMO) allow memory load and store reordering. Only the load and store operations that are separated by memory barriers are ordered. The operations that are not ordered by a memory barrier may execute in any order. Under such relaxed consistency models, strict persistency allows persists that are not separated by a memory barrier to drain concurrently to PM. Much like the consistency model, the programmability burden of annotating programs with necessary memory barriers falls on the programmer. The strict persistency model, under relaxed consistency, allows greater persist concurrency than in systems that require strict consistency. However, memory barriers still introduce long delays in the program execution; barriers stall until prior persists drain to the PM. The key deficiency of strict persistency is that it couples visibility of memory operations with their persistence. The long memory latency delays both the visibility and persistence of subsequent memory operations.

3.3.1 BUFFERED STRICT PERSISTENCY MODEL

The buffered strict persistency model couples persist order with the visibility order, while minimizing direct stalls resulting from the strict persist order. It does so by buffering the in-flight updates and allowing the persistent state to lag program execution. Persists still drain to the PM in a strict order (defined by their visibility), but execution does not need to stall until the prior persists drain to the PM. For instance, the buffered strict persistency model with the SC or TSO consistency model still requires persists to drain to PM in program order. However, the persists may be buffered and drained well after additional memory accesses have become visible according to the consistency model. In the case of a failure, recovery may observe an earlier program state than the latest state observed by the processors. But, it can never observe an illegal persist order, as persist order always corresponds to visibility order (i.e., recovery will observe a state that corresponds precisely to some execution state prior to the failure).

With the buffered model, persists drain asynchronously to PM. The persist latency is overlapped with useful volatile execution of the program. Given ideal (infinite) buffering, persist latency may fully overlap with useful program execution. However, in realistic scenarios with finite buffering, in-flight persists may still stall program execution if the rate of issue of persists during execution exceeds the rate at which persists drain to the PM. Under buffered strict persistency with conservative consistency models such as SC, persists within a thread are serialized to the PM. Stricter constraints on persists lowers the rate at which they may drain to the PM. Ongoing persists may fill up the buffers and stall program execution.

Hardware solutions that build buffered strict persistency models navigate the performance bottlenecks by minimizing stalls due to persist ordering constraints in the strict persistency model. Before we discuss those hardware mechanisms, we formally define the strict persistency model and provide ordering examples.

3.3.2 FORMALIZING THE STRICT PERSISTENCY MODEL

This section formally defines the persist order enforced by the strict persistency model. We use following notations to describe load and store memory operations to volatile or persistent addresses. "Thread" refers to the execution context that performs the memory operation. We use similar notation to formalize other persistency models in the rest of this chapter.

- M_x^i: A load or store operation to address x on thread i.
- L_x^i: A load operation to address x on thread i.
- S_x^i: A store operation to address x on thread i.

We consider two ordering relations to define the memory order. Volatile memory order (VMO) is an ordering relation over the memory operations that is prescribed by the memory consistency model. VMO encodes the visibility order of the memory operations. Persist memory order (PMO) is an ordering relation over the memory operations to PM prescribed by the memory persistency model. The following notations describe the order prescribed by VMO and PMO.

- $M_x^i \leq_v M_y^i$: M_x^i is ordered before M_y^i in VMO.
- $M_x^i \leq_p M_y^i$: M_x^i is ordered before M_y^i in PMO.

Under the strict persistency model, the persist order is same as the visibility order of memory operations. Ordering relations defined by VMO also apply to PMO. Thus,

$$M_x^i \leq_v M_y^j \rightarrow M_x^i \leq_p M_y^j . \tag{3.1}$$

If the store visibility is ordered by the consistency model, corresponding persists, as observed by the recovery observer, are also ordered by the persistency model. The ordering relation applies to the memory operations within and across the logical threads.

3.3.3 PERSIST ORDERING EXAMPLES

Figure 3.1 shows a single-threaded program execution on a system with sequential consistency and a strict persistency model. SC orders store visibility to the shared memory system, as shown in Figure 3.1. The strict persistency model further serializes persists. As per the persist order shown in Figure 3.1, persists to memory locations A and B are ordered. The strict persistency model does not allow persistent state to lag execution. As such, the subsequent stores are delayed until prior persist operations complete. Thus, in Figure 3.1, store C is visible only after the prior store to location B persists in PM.

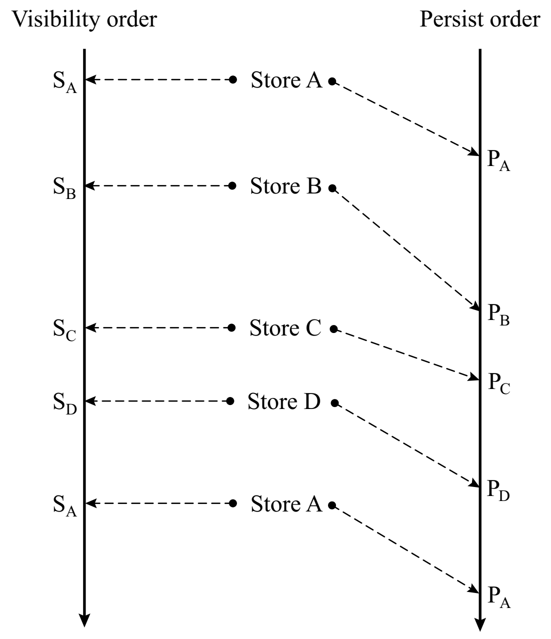


Figure 3.1: Visibility and persist order under sequential consistency and a strict persistency model.

The buffered strict persistency model allows persistent state of the program to lag that of its execution. Figure 3.2 shows the resulting visibility and persist order. As the persists still follow their visibility order, the persists to locations A, B, and C are serialized to PM. However, in the buffered model, persists may be buffered and delayed, allowing subsequent stores to be visible before the prior stores persist to PM. In Figure 3.2, visibility of store to location B is not delayed until the store to location A persists. Provided sufficient buffering, buffered strict persistency can reduce stalls during program execution compared to the strict persistency model, as seen in Figure 3.2.

3.4 HARDWARE IMPLEMENTATIONS OF STRICT PERSISTENCY

Multiple research works [60–62] propose hardware implementations of the buffered strict persistency model. To minimize any stalls resulting from serializing persists to PM, these works take two orthogonal approaches. Joshi et al. [60] builds the buffered strict persistency model with sequential consistency; the design serializes updates to PM, but updates occur in bulk to minimize the critical path of persists. We cover this work in detail in Section 3.4.2. Decoupled Persist Order (DPO) [61] implements a buffered strict persistency model with an ARMv7

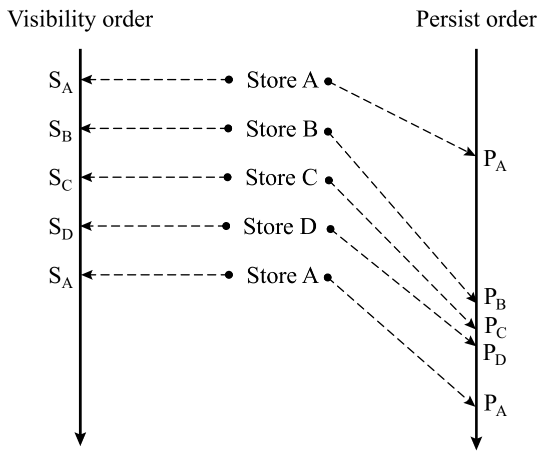


Figure 3.2: Visibility and persist order under sequential consistency and a buffered strict persistency model.

relaxed consistency model; it provides hardware structures that track intra- and inter-thread ordering dependencies enforced by the ARMv7 consistency model, and use them to drain the persists in the correct order. This work is described in detail in Section 3.4.3. Before we dive deeper into the proposed hardware designs, we first look at a naive hardware approach for implementing strict persistency.

3.4.1 NAIVE IMPLEMENTATION

The most straightforward implementation of the strict persistency model with sequential consistency ensures that each update becomes visible and the corresponding persist completes, before any subsequent store operations are issued. This constraint forbids concurrency among persists and introduces long stalls during execution. Buffered strict persistency may be implemented in hardware by employing a per-thread totally ordered queue in the cache hierarchy that issues persists to PM in order. Unfortunately, such an implementation also disallows any concurrency in persist operations, and runs the risk of the queues becoming full and delaying program execution. Prior works [60, 61] improve upon this naive design by taking two orthogonal approaches, discussed in the next two sections.

3.4.2 BULK PERSISTENCE

Joshi and co-authors [60] propose a hardware implementation of the buffered strict persistency model with sequential consistency with *bulk persistence*. Their proposed implementation forms batches of consecutive persists. The size of these batches is hardware configurable. Persists are allocated to their respective batches by the processor during execution; this allocation is transparent to the programmer. Persists that lie within the same batch need not be ordered; they are allowed to drain concurrently to PM. The boundaries of the consecutive batches enforce order as defined by the strict persistency model. That is, the persists that lie in different batches are serialized to the PM. The hardware flushes the updates to persistence at the end of each batch.

Although the persist order is enforced at the batch boundaries, a failure might occur when an batch is partially complete. As such, a failure might expose intra-batch persist reordering to recovery; such persist reordering is not allowed under strict persistency. To avoid partial completion of batches, Joshi et al. [60] employ logging to ensure atomicity of updates within a batch. Per update, the proposed hardware records the old value of the memory location to a pre-configured log area in PM. Before the cache line is modified within a batch, the old value of the cache line is recorded to the log entry in the PM. On failure, recovery inspects any partially completed batches and rolls back the PM state to the boundary of the previous complete batch.

The batch size determines the forward progress that is lost on failure. With a batch size of one, the persists are totally ordered as per an unbuffered strict persistency model. Recovery observes the latest PM state before the failure, but the total persist order incurs high performance overhead. Larger batches allow greater persist concurrency, but risk losing substantial forward progress on failure.

3.4.3 DELEGATED PERSISTENCE

Kolli and co-authors [61] propose delegated persist ordering (DPO), a hardware implementation for a buffered strict persistency model under the relaxed ARMv7 consistency model. ARMv7 requires programmers to appropriately annotate programs with memory barriers to establish order between memory operations. Any memory operations separated by a memory barrier are ordered. DPO relies on the relaxed memory ordering in ARMv7 consistency to enable higher persist concurrency.

Under the ARMv7 consistency model, memory operations within a thread are ordered by a FENCE operation. FENCE operations divide thread execution into epochs. Memory operations that lie in the same epoch are unordered, while those separated by a FENCE operation are ordered in shared memory. Intra-thread memory order is also established through data, address or control dependency between the memory operations [63]. Across threads, memory operations are ordered through their cache coherence order. That is, two stores to the same memory address on different threads are ordered globally; the order is determined by cache coherence in shared memory.

DPO builds a buffered strict persistency model. It buffers persists, records their visibility order, and enforces the ordering dependencies as persists drain to PM. Figure 3.3 shows its hardware architecture. It builds dedicated persist buffers alongside the L1 caches, which record persists and drain them to PM in order. The persist buffers track persists and fences executed by their respective cores to manage intra-thread persist order. Additionally, they monitor coherence traffic to log inter-thread persist order. A persist buffer records a request when a store operation

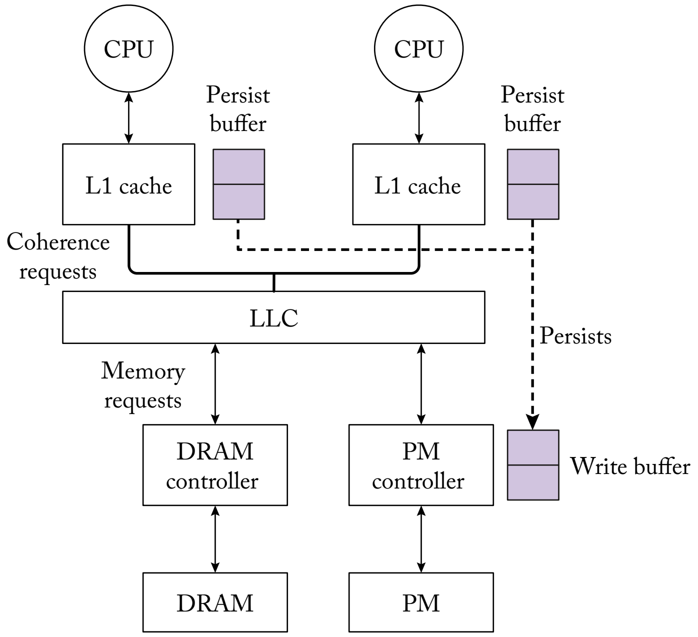


Figure 3.3: DPO hardware architecture, from [61].

to the address space in PM or a FENCE operation retires to the L1 cache. It is the responsibility of persist buffers to drain the persists separated by a FENCE operation in order. The persists that lie in the same epoch (i.e., between fences) may be reordered or coalesced as they drain to PM. As persist buffers are responsible for performing persist operations to PM, any write backs from the L1 cache are discarded.

The persist buffer also monitors incoming coherence requests to record inter-thread persist dependencies. Stores to the same memory location, on different threads, are ordered in the ARMv7 consistency model. DPO mandates that the two persists occur to PM in their coherence order. The persist buffer detects incoming read-exclusive requests that may steal ownership of the cache block in the ongoing epoch. For instance, a conflict is detected when CPU 1 tries to steal ownership of a dirty cache block from CPU 0. The persist buffer at CPU 0 annotates the coherence response to CPU 1 to log this persist dependency. The annotation prevents the persist at CPU 1 from draining prior to the persist at CPU 0. CPU 1 resolves the persist dependency and clears the annotation when it observes CPU 0 drain its persist to PM over the snoop network. As such, persist buffers drain persists to PM when their intra- and inter-thread dependencies resolve.

DPO ensures that program execution continues while the persists drain to PM in the background. Unlike strict consistency models, the ARMv7 consistency model allows memory operations (and therefore, also persists) to occur concurrently to shared memory (and thus, also to PM). Due to the relaxed persist ordering, DPO outperforms Intel's x86 persistency model by 1.28× on average on a set of write-intensive microbenchmarks [61].

3.5 DRAWBACKS OF STRICT PERSISTENCY

The strict persistency and strict consistency models, together, serialize memory operations to PM. As persists serialize to PM, the memory latency lies in the critical path of program execution. Serialization also disallows any common optimizations that reorder and coalesce PM operations.

The strict persistency model with relaxed consistency models, such as ARMv8 and Sun's Relaxed Memory Order (RMO), enable higher persist concurrency. Persists separated by memory barriers are ordered, but those that are not ordered by barriers (or other consistency model rules) may be issued concurrently to PM. However, these models, too, stall program execution at each memory barrier, until previous persists complete.

The buffered strict persistency model allows asynchronously draining persists to PM. However, buffered models are effective only when the rate at which CPUs issue persists is lower that the rate at which they may drain to PM. If the buffers fill up, buffered models suffer performance bottlenecks similar to the strict persistency model.

3.6 EPOCH PERSISTENCY MODEL

The epoch persistency model presents a relaxed model that allows higher persist concurrency than that allowed by the visibility restrictions on the corresponding memory operations as ordered by the system's consistency model. As such, epoch persistency decouples the persist order from the visibility of the memory operations. The visibility of the memory operations continues to follow the order defined by the consistency model. In contrast, the persist memory order enforces ordering constraints on persist operations that may differ from the memory operations' visibility order. To decouple persist order from visibility, programmers must annotate programs to indicate the order in which operations may persist to PM.

The epoch persistency model orders persists using *persist barriers*. Persist barriers divide execution into *epochs*; the persist order is managed at the granularity of epochs. Persists that lie within the same epoch may reorder or occur in parallel. Those that are separated by a persist barrier must occur in their program order. Upon failure, recovery never observes persists from different epochs out of order. That is, the persists from subsequent epochs are not visible to the recovery observer if any persist in a prior epoch is not visible.

Under epoch persistency, persist barriers have no effect on the visibility of memory operations. Memory operations that lie in different epochs may be made visible out of order, provided the memory consistency model allows it. The persistency model only restricts the order by which persists that may drain to PM. Figure 3.4a shows example code that orders persists to PM locations A and B using a persist barrier. Persists to locations A and B are separated by a persist

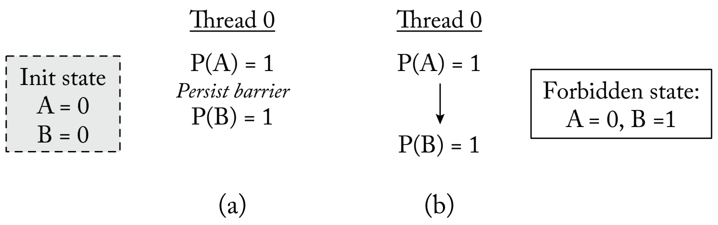


Figure 3.4: (a) Example code under epoch persistency model and (b) intra-thread persist order in epoch persistency model.

barrier and lie in different epochs. Stores to memory locations A and B may become visible in any order in shared memory, provided the visibility does not violate the system's consistency model (e.g., under relaxed consistency models such as RMO or ARMv7). However, persists to A and B are ordered by the persist barrier. That is, the persist to PM location B may not drain to PM before the persist to PM location A drains to PM. As shown in Figure 3.4b, the PM state $A = 0, B = 1$ is forbidden by the epoch persistency model.

3.6.1 STRONG PERSIST ATOMICITY

Memory consistency models often guarantee that store operations to the same or overlapping memory locations are serialized [58, 64]. Hardware coherency mechanisms serialize such memory operations, providing *store atomicity* [64]. Similarly, Pelley [57] proposes that a persistency model should also provide *strong persist atomicity* to serialize persists to the same or overlapping memory locations. With strong persist atomicity, recovery always observes a unique value of the memory location. Under strong persist atomicity, the persist order to the same or overlapping memory locations is derived from their visibility order. This restriction prevents recovery from observing any non-intuitive persist re-orderings to a location and their subsequent side-effects. Programmers can rely on strong persist atomicity to order persists across threads, as the racing persists are serialized by the persistency model. For instance, any synchronizing lock and unlock operations to a mutex that resides in PM are serialized, due to strong persist atomicity. Thus, recovery observes correct lock state, which it may use, for instance, to reset or re-acquire it after failure.

The epoch persistency model provides strong persist atomicity. Persists to the same memory location within racing epochs (on the same or different threads) follow the visibility order of the corresponding stores.

The persist ordering is cumulative. That is, the inter-thread persist order due to strong persist atomicity additionally orders any subsequent persists that are separated by a persist barrier on the subsequent threads. Figure 3.5a–b shows an example of this relation. The persists to the memory location A on threads 0 and 1 establish an inter-thread order between the two persists. That is, the persist to location A on thread 1 is ordered after the persist to location A on thread 0,

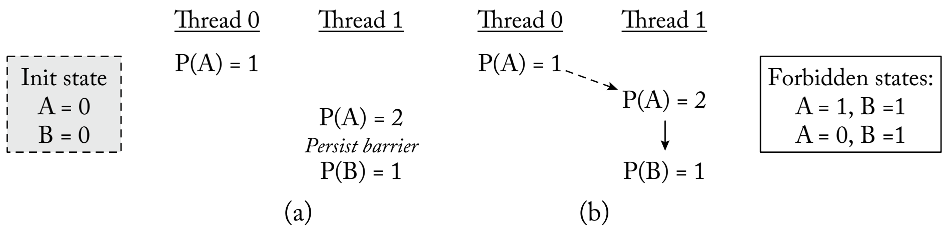


Figure 3.5: (a) Example code under epoch persistency model and (b) inter-thread persist order in epoch persistency model.

following their visibility order. Additionally, the persist to location B is ordered after the persist to location A on thread 1 by an intermediate persist barrier. The cumulativity orders the persist to location B on thread 1 after the persist to location A on thread 0.

To summarize, the epoch persistency model establishes intra- and inter-thread persist order using (a) a persist barrier within a thread and (b) a combination of racing persists and a following persist barrier across the threads.

The epoch persistency model provides higher persist concurrency compared to the strict persistency model. The persist ordering is not bound to the consistency model of the system. However, programmers need to manage the persist order separately from the visibility order of the operations. Much like relaxed consistency models, it is the programmer's responsibility to annotate programs (e.g., using persist barriers) to obtain the desired order.

3.6.2 BUFFERED EPOCH PERSISTENCY MODEL

Although epoch persistency allows persist concurrency within epochs, the persists delay program execution at the epoch boundaries. The program execution stalls at each persist barrier to allow prior persists to complete, before any subsequent stores can execute. This effect is amplified in programs with smaller epochs.

The buffered epoch persistency model (similar to the buffered strict persistency model discussed in Section 3.3.1) allows persistent state to lag its execution. Program execution continues beyond a persist barrier to execute memory operations in the subsequent epochs. Subsequent updates may be made to the volatile shared memory system. However, persists must drain to the PM in their epoch order. Hardware implementations must guarantee that updates drain from buffers and write-back caches to PM in their epoch order.

On failure, under buffered epoch persistency, recovery is allowed to observe the earlier state of the program in PM, but it may never observe persist reorderings across epochs. That is, persists from subsequent epochs may not be visible if any persist from a prior epoch is not visible to recovery. The buffered model eliminates long latency stalls at each persist barrier, provided hardware has enough buffering to hoist ongoing incomplete persists.

3.6.3 FORMALIZING THE EPOCH PERSISTENCY MODEL

We formally define the persist order enforced by the epoch persistency model. We use similar notations described earlier in Section 3.3.2.

- M_x^i: A load or store operation to address x on thread i.
- L_x^i: A load operation to address x on thread i.
- S_x^i: A store operation to address x on thread i.
- PB^i: A persist barrier issued by thread i.

As described earlier in Section 3.3.2, volatile memory order (VMO) defines an ordering relation prescribed by the memory consistency model. Persist memory order (PMO) encodes an ordering relation on memory operations to PM prescribed by the memory persistency model.

We use the following notations to describe ordering relation.

- $M_x^i \leq_v M_y^i$: M_x^i is ordered before M_y^i in VMO.
- $M_x^i \leq_p M_y^i$: M_x^i is ordered before M_y^i in PMO.

Intra-thread persist order. Under epoch persistency, memory operations separated by a persist barrier on a logical thread are ordered in PMO.

Thus,

$$M_x^i \leq_v PB^i \leq_v M_y^i \rightarrow M_x^i \leq_p M_y^i. \tag{3.2}$$

Persists within an epoch are concurrent and may occur to PM in parallel.

Strong persist atomicity. Conflicting accesses to the same or overlapping memory locations, when at least one of them is a store, are ordered by strong persist atomicity.

$$\begin{aligned} M_x^i \leq_v S_x^j &\rightarrow M_x^i \leq_p S_x^j \\ S_x^i \leq_v M_x^j &\rightarrow S_x^i \leq_p M_x^j. \end{aligned} \tag{3.3}$$

Conflicting accesses in PMO assume their visibility order from VMO. Strong persist atomicity disallows recovery to observe any non-reachable program state in case of a failure.

Transitivity. Persist order in PMO is transitive and irreflexive.

$$\left(M_x^i \leq_p M_y^j\right) \wedge \left(M_y^j \leq_p M_z^k\right) \rightarrow M_x^i \leq_p M_z^k. \tag{3.4}$$

Thus, persist order within and across threads is the combination of persist order due to Equations (3.2) and (3.3).

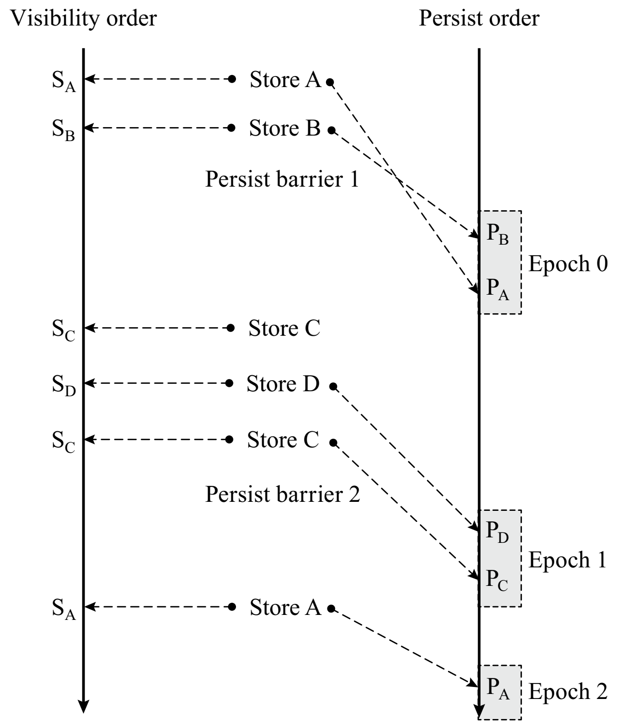


Figure 3.6: Intra-thread visibility and persist order in epoch persistency model.

3.6.4 PERSIST ORDERING EXAMPLES

We discuss examples of intra- and inter-thread persist ordering constraints formalized by Equations (3.2)–(3.4).

Intra-thread persist order. Figure 3.6 shows the visibility and persist order with sequential consistency and epoch persistency models. Under sequential consistency, visibility of store operations is ordered. The epoch persistency model decouples the persist order and allows the persists corresponding to each store to reorder within the epochs. Thus, although the visibility of stores to locations A and B are ordered (as shown by visibility order in Figure 3.6), the corresponding persist operations may occur to PM in any order. The epoch persistency model also allows persist coalescing to the same PM location within an epoch. In epoch 1, persists to the PM location C may be coalesced, before being written back to the PM. As shown in Figure 3.6, epoch 1 results in two persist operations to PM locations C and D.

Under epoch persistency, program execution stalls at each persist barrier before prior persists complete. For instance, persist barrier 1 stalls execution of store operations to locations C and D, until persists to PM locations A and B complete in PM.

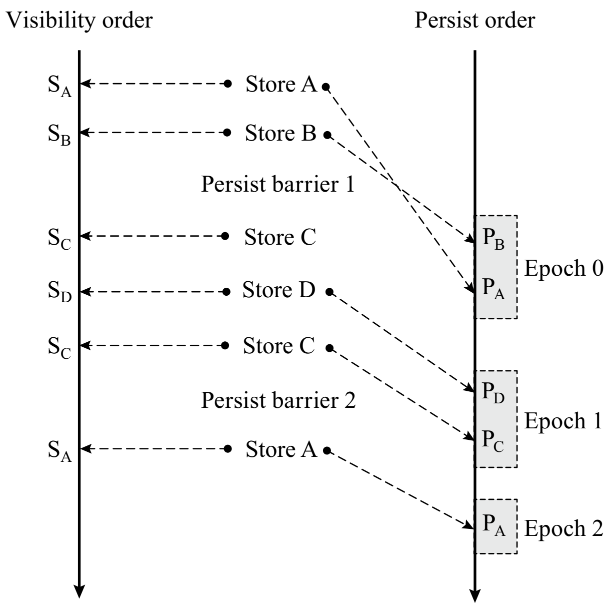


Figure 3.7: Intra-thread visibility and persist order in buffered epoch persistency model.

The buffered epoch persistency model further allows persistent state to lag program execution. Thus, ongoing persists may be buffered and written back to PM at a later time. Figure 3.7 shows an example of visibility and persist ordering under the buffered epoch persistency model. Persist barrier 1 does not delay execution and visibility of subsequent stores to locations C and D. Persists in epoch 1 and epoch 2 are still ordered in the buffered model.

Inter-thread persist order. Persists to the same or overlapping PM locations establish inter-thread persist order, as defined by strong persist atomicity in Equation (3.3). Figure 3.8 shows program execution on two threads; thread 0 and thread 1 update the same memory location A. Thus, the persist to PM location A on thread 0 is ordered before the persist to PM location A on thread 1. These persists assume their visibility order, as per Equation (3.3). Due to transitivity (as per Equation (3.4)), the subsequent persist barrier 1 on thread 1 orders the persist to location B after persists location A on threads 0 and 1.

3.7 HARDWARE IMPLEMENTATIONS OF EPOCH PERSISTENCY

Next, we discuss hardware implementations of the epoch and buffered epoch persistency models that have been proposed in the research literature. Before we dive deep into various state-of-

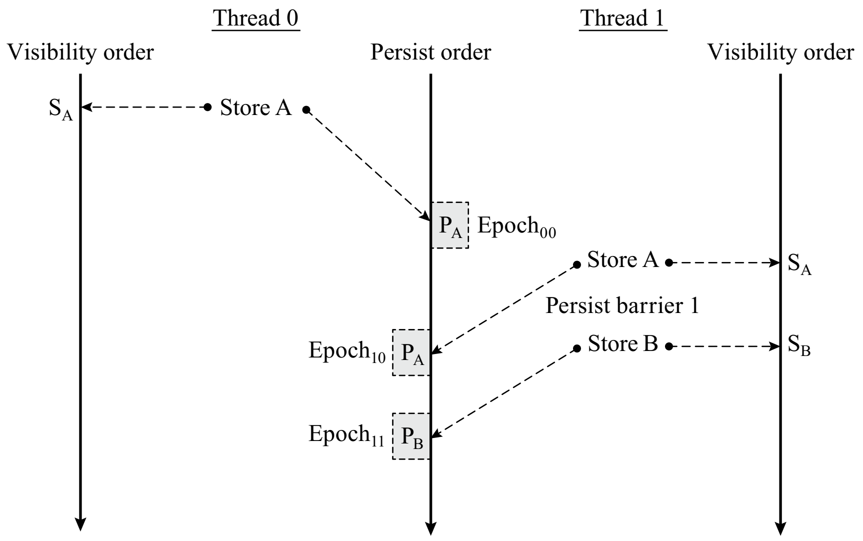


Figure 3.8: Inter-thread visibility and persist order in epoch persistency model.

the-art hardware designs, we first discuss the most straightforward approach to build epoch persistency in hardware.

3.7.1 NAIVE IMPLEMENTATION

The most naive implementation of the epoch persistency model achieves epoch ordering by forcing persists from the prior epochs to complete in PM before any subsequent stores are issued by the processor. In this naive implementation, the processor is responsible for enforcing the correct persist order.

As outlined earlier in Section 2.2, persists can drain to PM from write-back caches: (a) voluntarily due to explicit cache flush operations issued to the cache hierarchy by the processor; or (b) involuntarily due to cache-line writebacks triggered by cache replacement policies. To prevent persist reordering across persist barriers, processors must prevent out-of-order writebacks, both due to voluntary cache flushes and involuntary cache-line replacements.

To avoid persist reordering, processors may commit a persist barrier only when the prior persists complete in PM. Subsequent store operations may update the cache only after the persist barrier commits. Processors must ensure that updates within an epoch are indeed committed to PM before the persist barrier. Tracking this constraint requires significant bookkeeping. Processor must track the memory addresses to which stores were performed in the ongoing epoch, and make sure that these updates reach PM before a persist barrier commits. Several different design choices can achieve this goal.

(a) The processor may flush the cache line as soon as an update is written to the write-back cache. This approach is similar to write-through caching, except that the non-invalidating cache flush operations may retain the newly clean cache line in the cache. However, multiple flush operations negate any opportunities to coalesce persists to the same memory locations within an epoch. Multiple flush operations also increases PM write bandwidth.

(b) The processor can track memory locations that were updated within an epoch, and issue writebacks to the corresponding cache lines at the end of the epoch. Under this approach, processors may coalesce updates to the same memory locations performed within an epoch, thereby minimizing PM writes. However, a stream of write-back operations issued to PM at once can saturate hardware queues (e.g., write-pending queues in PM controller), and stall commit of a persist barrier.

(c) Alternatively, software can communicate the memory locations that need to be flushed to PM within each epoch (e.g., by annotating the program with `CLWB` operations in Intel x86). The processor must ensure, at the end of each epoch, that the cache flush operations issued during that epoch have completed to PM. Such an approach requires programmers to annotate software applications with explicit cache flush operations. Programmers need to ensure that the correct set of memory locations are flushed to PM within an epoch for correct recovery.

Intel x86. The Intel x86 ISA provides an epoch persistency model. Under the x86 consistency model, processors serialize store visibility to shared memory. That is, processors perform stores to the write-back caches in order.

Additionally, Intel x86 also defines extensions required to apply updates directly to PM and order them with respect to prior updates. As described earlier in Chapter 2, Intel x86 provides the `CLWB` operation that performs a non-invalidating cache flush to PM, and the `SFENCE` instruction that orders prior stores and `CLWB`s with subsequent stores and `CLWB`s in a logical thread. Thus, `SFENCE` defines epoch boundaries. Within an epoch, `CLWB`s are allowed to reorder, coalesce and/or occur in parallel. `SFENCE` operations act as persist barriers that stall subsequent stores until prior stores and `CLWB`s complete.

3.7.2 BUFFERED EPOCH PERSISTENCY IN HARDWARE

Condit and co-authors [65] propose BPFS, which provides hardware mechanisms to build a buffered epoch persistency model. BPFS guarantees the epochs are ordered, but it additionally allows program execution to run ahead while persists issued in the prior epochs complete. The processor communicates epoch ordering constraints to the cache subsystem. In BPFS, the cache subsystem must ensure that persists are performed to PM in the correct order.

BPFS uses various hardware structures and metadata to track intra- and inter-epoch dependencies. The processor manages an epoch counter per hardware context to record an ongoing epoch. The epoch counter increments when a persist barrier commits. The processor assigns an ongoing epoch ID to each PM store as it is performed to the cache. The epoch ID propagates through the cache subsystem before the corresponding store persists to PM. To store the epoch ID associated with each update, BPFS extends each cache block with an *epoch ID* and *persistence* bits associated with the cache line. The persistence bit is set if the cache line belongs to the PM location. The cache controller enforces epoch ordering constraints by ensuring that cache lines that belong to younger epochs do not drain from the cache out of order. When all the cache lines from the oldest epoch evict to PM, the subsequent epoch is allowed to drain.

BPFS does not actively seek to drain dirty cache lines from prior epochs, unless there is an epoch conflict (two epochs access the same location). It preserves strong persist atomicity when epochs conflict. If any processor performs a read or write to a cache line that holds dirty data from a prior epoch, the cache controller in BPFS ensures that the prior epoch flushes to PM before the operation is performed to the cache. It passively flushes such dirty cache lines from the prior epochs to the PM on a conflict. This protocol guarantees that updates within the older epochs persist before other processors may observe the conflict-triggering update or its side-effects.

3.7.3 OFFLINE CONFLICT RESOLUTION

The key drawback of the BPFS implementation, as Joshi and co-authors [60] note, is that persists drain to PM lazily. Persists drain to PM due to cacheline replacement or due to any future epoch conflicts. On an epoch conflict—when the processor issues a load or a store operation to the memory location that is not yet written back to PM from the cache—the BPFS cache controller flushes updates so that the earlier epochs are synced to PM. This behavior stalls program execution at each epoch conflict until the cache controller drains those persist operations to PM. Such epoch conflicts occur frequently under BPFS because updates are drained lazily to PM.

Joshi and co-authors [60] navigate this challenge by performing offline epoch conflicts resolution. They propose two hardware mechanisms: (a) inter-thread dependency tracking that records epoch dependencies so that they may be resolved offline; and (b) proactive flushing that actively flushes updates when epochs terminate to minimize the possibility of epoch conflicts. Figure 3.9 shows a high-level overview of their proposed hardware architecture. Much like BPFS, they tag each cache line with an epoch ID.

Inter-thread dependency tracking. Inter-thread dependency tracking (IDT) allows program execution to proceed even when epochs conflict. These epochs are drained offline in the correct order, without stalling program execution. IDT detects a conflict when a *dependent epoch* performs a load or a store operation to a location that was modified by a previous *source epoch*. Per ongoing epoch, IDT maintains incoming and ongoing dependencies in the cache controller as shown in Figure 3.9. Specifically, IDT maintains two lists per epoch ID: (a) epochs that need to be notified when an epoch is entirely flushed to PM and (b) epochs upon which the current epoch depends. IDT delays an epoch flush if it depends on other epochs that are yet to persist.

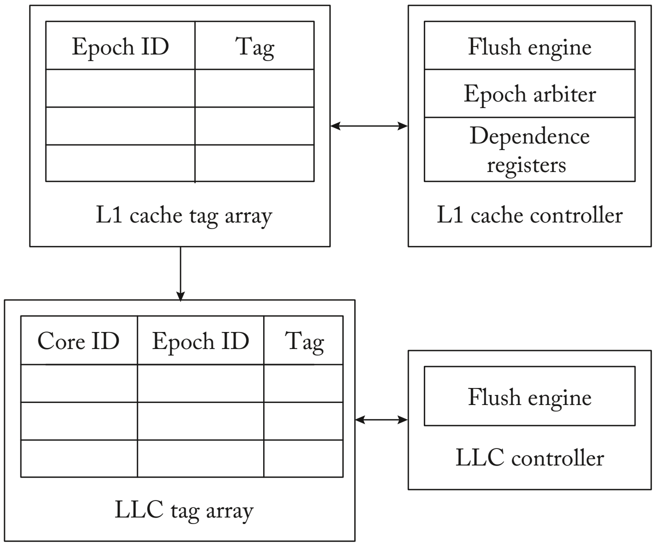


Figure 3.9: Hardware implementation of inter-thread dependency tracking and proactive flushing mechanisms in Joshi et al. [60].

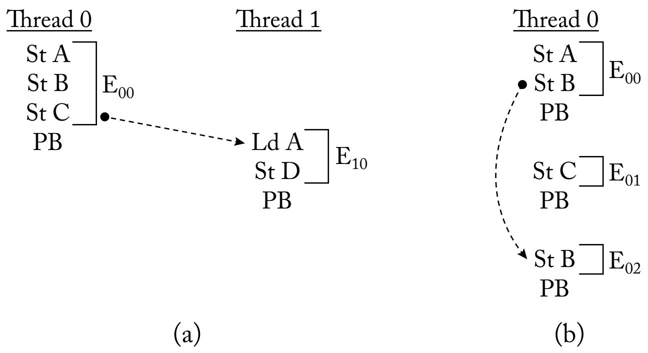


Figure 3.10: (a) Inter-thread epoch conflict and (b) intra-thread epoch conflict.

An epoch flushes to PM when all the prior epochs it depends upon complete. The epoch signals completion to its dependent epochs when it is entirely flushed.

Intra- and inter-thread epoch conflicts may arise in various scenarios, shown in Figure 3.10. In Figure 3.10a, inter-thread conflicting epochs arise when thread T_1 performs a load from epoch E_{10}. IDT logs the dependency between epochs E_{00} and E_{10}, and allows the load operation to proceed without stalling for the source epoch E_{00} to persist. When E_{00} completes, IDT notifies its completion to the cache controller hosting updates in epoch E_{10}. Epoch E_{10} can then persist.

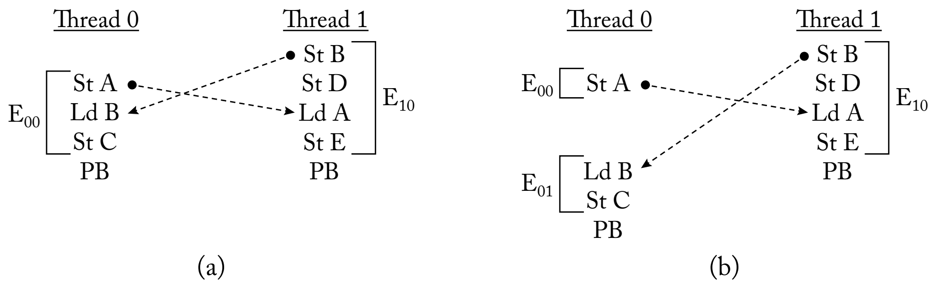


Figure 3.11: (a) Example of epoch deadlock and (b) epoch deadlock resolution.

Proactive epoch flush. Intra-thread epoch conflicts occur when a store operation updates a cache line that had been updated by a previous epoch in the same thread. In such a scenario, the prior epoch must be flushed before the conflicting update may overwrite the cache line. Figure 3.10b shows an example of an intra-thread epoch conflict. Epochs E_{00} and E_{02} conflict as they update the same PM location B. Epoch E_{00} needs to be flushed completely, before the update to location B in epoch E_{02} overwrites the cache line. If such a scenario arises in BPFS, execution stalls until the prior epoch is flushed to PM.

Joshi and co-authors [60] propose proactive flushing (PF) to actively flush epochs when they complete. Whenever an epoch completes, the proactive flush mechanism in the cache controller proactively flushes it to PM, provided there are no ongoing inter-thread epoch conflicts. This reduces the possibility of any future intra- and inter-thread conflicts, as the cache controller flushes epochs as soon as they terminate. In Figure 3.10b, proactive flush seeks to flush the updates in epoch E_{00} as soon as it terminates, without waiting for the future conflict with epoch E_{02}.

Epoch dependency deadlocks. A deadlock may arise if two epochs are inter-dependent. Figure 3.11a shows an example of a deadlock. Epochs E_{00} and E_{10} perform load operations to the memory location previously updated by the other epoch. Due to the conflict, epoch E_{00} may not persist before epoch E_{10} and vice versa, resulting in a deadlock. IDT provisions hardware to detect such a deadlock. When another processor issues a load or a store operation to the memory location currently being updated by the ongoing epoch, IDT conservatively tags such a scenario as a deadlock. To resolve such deadlocks, IDT breaks the source epoch when it observes a load or a store operation from another processor. As shown in Figure 3.11b, IDT breaks the epoch on thread T_0 into epochs E_{00} and E_{01}. Epochs E_{00}, E_{10}, and E_{01} are then serialized to PM.

Such deadlocks can be avoided in software by appropriately defining epochs in a logical thread. In a properly labeled program without any data races, memory operations to the same or overlapping memory locations across the threads can be serialized to PM through user-defined synchronization primitives. We discuss software strategies for PM later in Chapter 5.

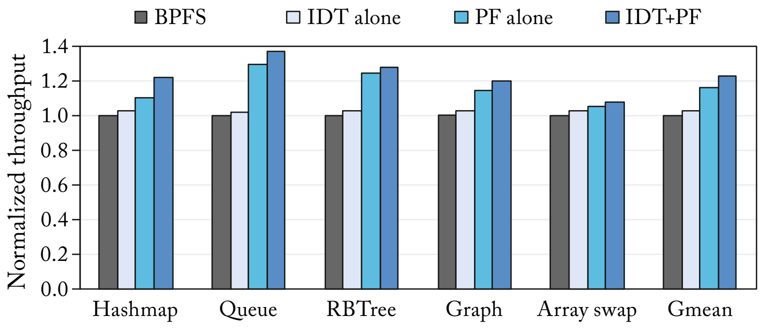


Figure 3.12: Throughput of epoch persistency implementation proposed by Joshi et al. [60] compared to BPFS [65] hardware.

Joshi and co-authors [60] compare the performance of IDT and PF mechanisms with the BPFS hardware implementation. Figure 3.12 shows the throughput comparison of the two mechanisms with BPFS, measured over a set of micro-benchmarks designed to access PM data. IDT improves application throughput by 3% over BPFS as it reduces stalls resulting from inter-thread epoch conflicts. In comparison, PF results in a higher improvement of 17% on average over BPFS, as it reduces the frequency of conflicts. Overall, both mechanisms together result in a throughput improvement of 22% on average over BPFS.

3.7.4 SEPARATE ORDERING AND DURABILITY BARRIERS

HOPS [66] takes a different approach to the hardware design for buffered epoch persistency. It separates ordering and durability of persists by introducing two distinct types of persist barriers. The ordering barrier, ofence, divides program execution into epochs, but ensures persist ordering alone. The ofence barrier does not ensure durability of prior persists. Alternatively, the durability barrier, defence, enforces durability of prior persists. The defence barrier assures that the prior persists complete to PM.

Unlike prior solutions like BPFS [65] and IDT [60] that maintain persist metadata per cache line, HOPS builds persist buffers adjacent to the L1 cache to manage ongoing persists. Figure 3.13 shows the high-level hardware implementation of HOPS. The persist buffers track ongoing updates. When the processor performs a PM update to the L1 cache, an entry containing the updated cache line is also inserted into the persist buffer. HOPS also maintains an epoch ID (similar to BPFS [65]) in the L1 cache controller that denotes the ongoing epoch. It records the ongoing epoch ID per PM update in the persist buffer.

The offence barrier increments the epoch ID in the cache controller to commence a new epoch. This is a low-latency operation as it does not enforce durability of the prior updates. Any subsequent updates are annotated with the updated epoch ID in the persist buffer as they belong to the subsequent epoch. Each persist buffer proactively flushes updates that belong to

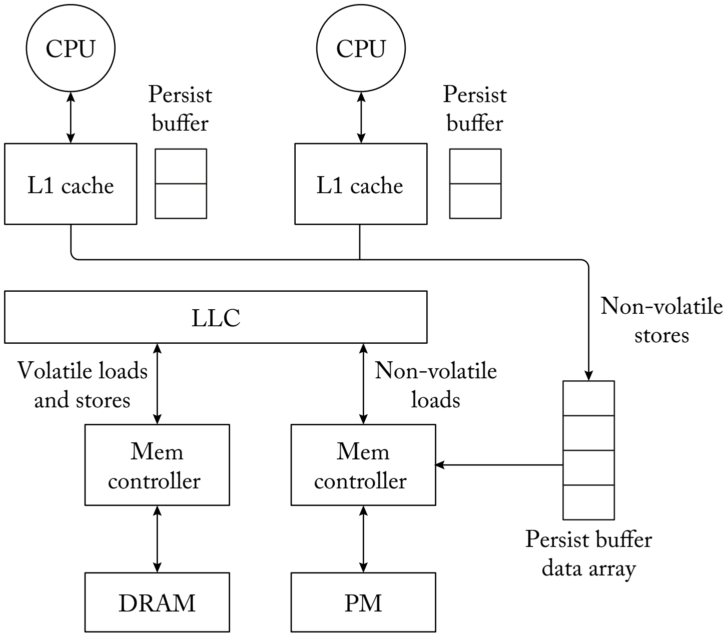


Figure 3.13: HOPS [66] hardware architecture.

the same epoch concurrently to PM. As persist buffers are responsible for writing back updates to PM, HOPS discards any writebacks triggered due to cache replacement. The persist buffers guarantee epoch order. That is, the updates in the subsequent epochs are flushed only when all the updates in the prior epochs complete in PM. The defence barrier increments the epoch ID to mark the new epoch, and also flushes the ongoing updates in the persist buffers to PM. It is a heavy-weight operation as it drains the persist buffer before it completes.

Like BPFS [65] and IDT [60], HOPS also resolves intra-thread and inter-thread epoch dependencies in hardware. It takes two different approaches to manage and resolve these conflicts.

Intra-thread epoch order. Epochs within a thread conflict when a subsequent epoch updates the PM location that was also updated by a prior epoch, and is yet to be flushed to PM (as shown previously in Figure 3.10b). In HOPS, the L1 cache retains the latest data, but the persist buffer can record multiple versions of a cache line. The persist buffer drains the two copies of a cache line in epoch order.

Inter-thread epoch order. Inter-thread epoch conflicts occur when an epoch on one thread performs a load or a store operation to the location updated by an epoch on another thread (as shown previously in Figure 3.10a). HOPS preserves the inter-thread epoch order by employing

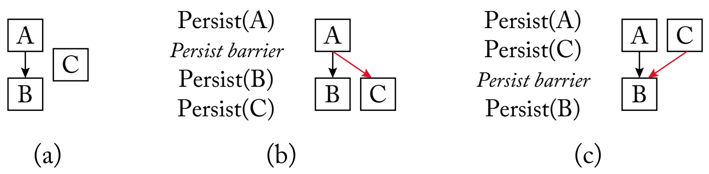


Figure 3.14: (a) Desired order on persists A, B, and C, (b) persist barrier additionally orders persists A and C, and (c) persist barrier additionally orders persists C and B.

a dependency tracking mechanism similar to IDT [60]. It identifies a conflict when the L1 cache receives a read-exclusive access from another core to a cache line that is currently in the modified state. On detecting a conflict, the L1 cache controller responds with exclusive permission to the cache line, along with a dependency pointer to the source epoch. The dependent epoch is not flushed until the source epoch is entirely flushed to PM.

3.8 DRAWBACKS OF EPOCH PERSISTENCY

Epoch persistency allows higher persist concurrency at the granularity of epochs. Unfortunately, it labels only consecutive persists that lie within an epoch as concurrent. It does not relax constraints on persists that lie in different epochs. The model does not provide primitives that can be used to encode an arbitrary set of ordering constraints on persists.

Figure 3.14a shows the desired order on persists A, B, and C. Suppose persist A needs to be ordered before persist B for correct recovery, but persist C may be issued concurrently to PM. Unfortunately, epoch persistency model fails to apply such fine-grained ordering constraints on persists. That is, the persist barrier (required to order persist A before persist B) additionally constrains persist C when it is issued in either of the epochs as shown in Figure 3.14b–c. Thus, the persist barrier precludes persist C from being concurrent to persists A and B, in both the cases, in Figure 3.14b–c.

3.9 STRAND PERSISTENCY MODEL

The strand persistency model allows programmers to specify arbitrary order on persists. Similar to epoch persistency, it decouples visibility order of memory operations from the persist memory order; the visibility of memory operations is governed by the consistency model, while the persistency model codifies the order in which persists occur to PM.

The strand persistency model manages persist order within *strands*. It divides program execution on a logical thread into strands; persists are ordered within a strand, but those that lie on different strands are concurrent. A strand provides the abstraction of an independent stream of persists that happen to lie on the same logical thread, but may be issued concurrently to PM.

The strand persistency model provides three primitives to specify persist order. A persist barrier orders persists within a strand. This is similar to the epoch persistency model, wherein persist barriers order persists within a logical thread. A new strand primitive initiates a new strand; the following persists that now lie on a separate strand may reorder or be issued concurrently to the persists on prior strands. Persist barriers do not order persists that lie on different strands. Thus, strands behave as separate logical threads as far as persist memory order is concerned. It is important to note that visibility of memory operations that lie on different strands remains unchanged and is governed by the memory consistency model. The notion of strands is restricted to persist memory order alone. Finally, a join strand primitive merges prior strands. Any persists separated by a join strand primitive are ordered, even if they lie on separate strands. The idea of a join strand primitive, with respect to persist order, is analogous to a thread join operation that merges prior logical threads. The join strand primitive ensures that prior persists complete, before any subsequent persists are issued.

3.9.1 STRONG PERSIST ATOMICITY

Similar to the epoch persistency model, the strand persistency model also ensures that persists to the same or overlapping memory locations are serialized to PM. These persists to conflicting memory addresses can lie on the same or different strands within a thread, or on different threads. The persist memory order assumes the visibility order of the memory operations under strong persist atomicity.

The combination of the three primitives, persist barrier, new strand, and join strand, enforces the persist order. Recovery may never observe any persist mis-orderings post failure; persists ordered by a persist barrier within a strand, or those separated by a join strand, are always ordered with respect to recovery. Additionally, conflicting persists on the same strand, separate strands, or different logical threads, ordered through strong persist atomicity, are observed in order by recovery.

3.9.2 FORMALIZING THE STRAND PERSISTENCY MODEL

We formally define the persist order enforced by the strand persistency model. We use notations defined earlier in Section 3.3.2 to describe persist order.

- M_x^i: A load or store operation to address x on thread i.
- S_x^i: A store operation to address x on thread i.
- PB^i: A persist barrier issued by thread i.
- NS^i: A new strand issued by thread i.
- JS^i: A join strand issued by thread i

As described earlier in Section 3.3.2, persist memory order (PMO) describes an ordering relation on memory operations prescribed by the persistency model.

Intra-strand ordering. The new strand primitive begins a new strand. It does not apply the prior persist ordering constraints to the persists on subsequent strands. Meanwhile, a persist barrier continues to order memory operations within a strand.

$$\left(M_x^i \leq_v PB^i \leq_v M_y^i\right) \wedge \left(\nexists NS^i : M_x^i \leq_v NS^i \leq_v M_y^i\right) \rightarrow M_x^i \leq_p M_y^i. \tag{3.5}$$

Two memory operations that are not separated by a new strand are ordered by a persist barrier in PMO.

The join strand primitive additionally orders memory operations that lie on separate strands on a logical thread.

$$M_x^i \leq_v JS^i \leq_v M_y^i \rightarrow M_x^i \leq_p M_y^i. \tag{3.6}$$

Memory operations separated by join strand on a logical thread are ordered in PMO.

Strong persist atomicity. Persists to the same or overlapping memory locations in PMO assume the order from VMO. As described earlier in Equation (3.3), conflicting persists on different strands or threads are ordered in PMO.

$$\begin{aligned} M_x^i \leq_v S_x^j &\rightarrow M_x^i \leq_p S_x^j \\ S_x^i \leq_v M_x^j &\rightarrow S_x^i \leq_p M_x^j. \end{aligned} \tag{3.7}$$

The strand persistency model preserves strong persist atomicity so that recovery never observes side-effects due to reorderings that would never occur in fault-free program execution.

Transitivity. Finally, persist order is transitive and irreflexive, as described in Section 3.4.

$$\left(M_x^i \leq_p M_y^j\right) \wedge \left(M_y^j \leq_p M_z^k\right) \rightarrow M_x^i \leq_p M_z^k. \tag{3.8}$$

Thus, the two persists are ordered in PMO due to the combination of persist ordering constraints in Equations (3.5)–(3.7).

3.9.3 PERSIST ORDERING EXAMPLES

This section discusses persist order due to the primitives in the strand persistency model.

Intra-strand persist concurrency. Figure 3.15a shows example code that employs the `NewStrand` primitive to perform independent persists concurrently on different strands. A `NewStrand` operation initiates a new strand (Strand 1) that performs persist C concurrent to persists A and B. The `NewStrand` operation clears any prior persist ordering constraints. Thus,

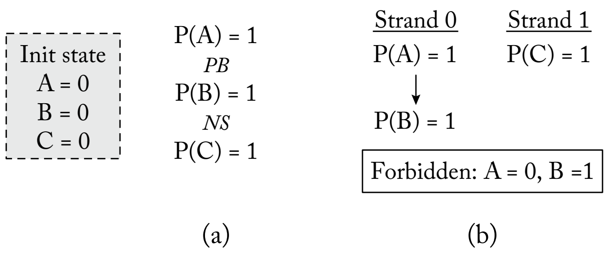


Figure 3.15: Persist ordering due to `NewStrand` in strand persistency model.

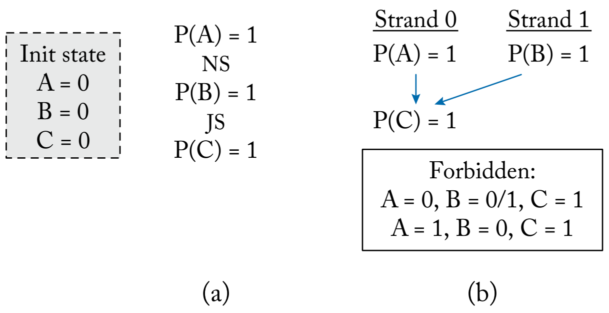


Figure 3.16: Persist ordering due to `JoinStrand` in strand persistency model.

persist barrier `PB` does not impose any ordering constraints on persist C. Persist barrier `PB` orders persist A before B on strand 0. As such, a `NewStrand` primitive may be used to achieve the desired persist ordering, which is not possible under epoch persistency (as explained earlier in Section 3.8).

Inter-strand persist concurrency. Figure 3.16 shows the inter-strand persist ordering due to a `JoinStrand` primitive within a thread. The `JoinStrand` primitive merges prior strands 0 and 1 to guarantee that persists A and B complete before persist C. Persists A and B lie on different strands and are concurrent.

Inter-strand persist order. Strong persist atomicity governs the order of persists to the same or overlapping memory locations that lie on different strands or threads. It derives persist order from the order in which updates become visible. Figure 3.17 shows an example code that establishes inter-strand persist order through strong persist atomicity. Strong persist atomicity orders persist A on strand 0 before persist A on strand 1. Due to transitivity, persist B on strand 1 is also ordered after persist A on strand 0.

Inter-thread persist order. Strong persist atomicity also orders persists to the same or overlapping memory locations on different threads. Figure 3.18 shows one such example. Persist B

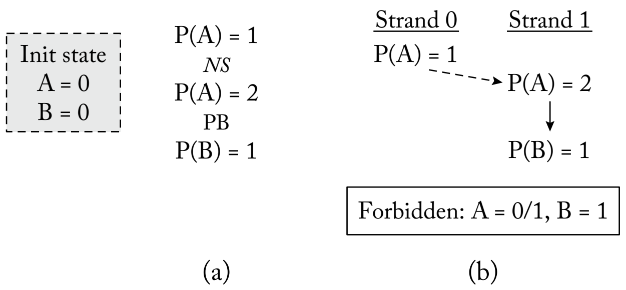


Figure 3.17: Inter-strand persist ordering due to strong persist atomicity in strand persistency model.

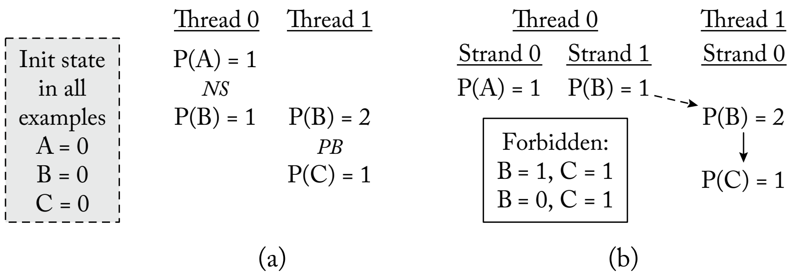


Figure 3.18: Inter-thread persist ordering due to strong persist atomicity in strand persistency model.

on thread 0, strand 1 is ordered before persist B on thread 1, strand 0. Due to transitivity, persist C on thread 1 is also ordered after persist B on thread 0. As persists A and B lie on a different strands in thread 0, persist A may occur concurrent to the other persists on threads 0 and 1.

3.10 HARDWARE IMPLEMENTATIONS OF STRAND PERSISTENCY

StrandWeaver [67] builds the strand persistency model in hardware to minimally constrain persists to PM. It decouples the volatile and persist memory order and relaxes persist order even when the system builds upon a conservative consistency model like Intel TSO. StrandWeaver implements strand primitives, persist barrier, `NewStrand`, and `JoinStrand` as hardware ISA extensions. It uses `CLWB` operations to flush dirty cache lines to PM. `CLWB` operations complete when the CPU receives an acknowledgement from the PM controller.

Figure 3.19 shows the high-level architecture of StrandWeaver. It features persist queue and strand buffer units that jointly enforce persist ordering. The persist queue is implemented alongside the load-store queue. It ensures that `CLWB` and store operations are *issued* to the L1

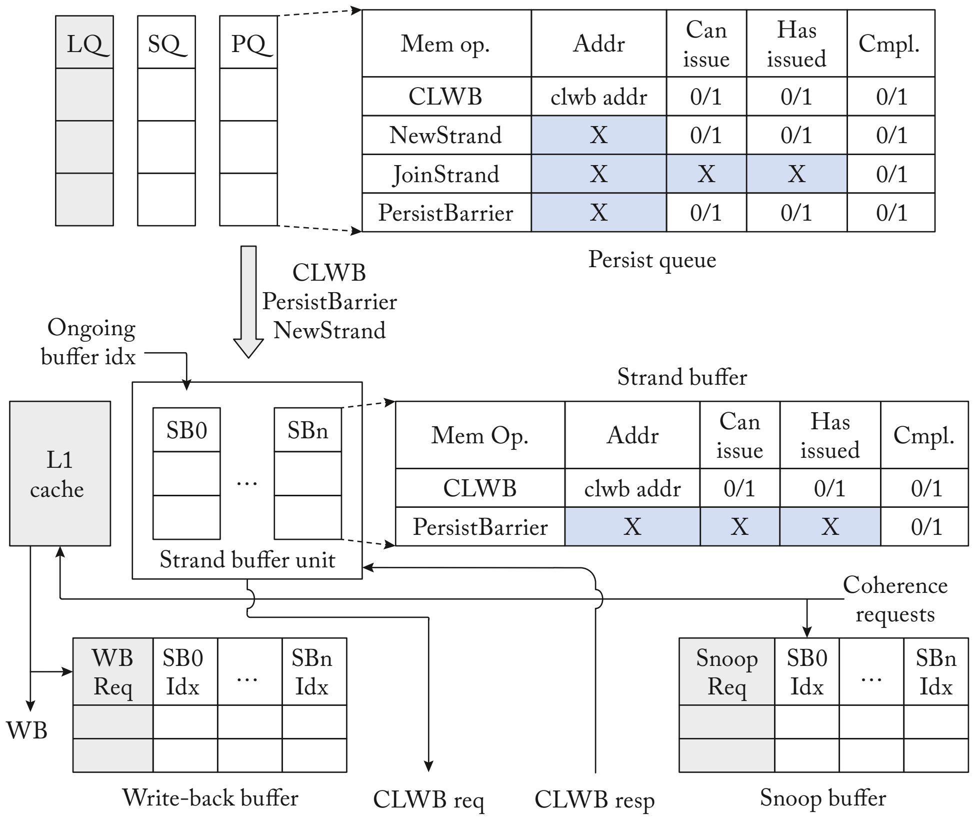


Figure 3.19: StrandWeaver [67] hardware architecture.

cache in order, and CLWBs separated by JoinStrand operation *complete* in order. The strand buffer unit is implemented adjacent to the L1 cache and manages the order in which persists drain to PM. It comprises an array of strand buffers; each strand buffer is responsible for ordering persists within a strand and ensures that persists separated by a persist barrier within a strand complete in order. It also monitors coherence messages to preserve inter-thread persist order.

The persist queue records ongoing CLWBs, persist barriers, NewStrand, and JoinStrand operations, and issues them to the strand buffer unit when any ordering constraints on them resolve. It coordinates with the store queue to order any stores and CLWBs separated by a persist barrier within a strand. It also orders CLWBs separated by JoinStrand operation. JoinStrand stalls issue of subsequent CLWBs until prior CLWBs complete. CLWBs, persist barriers, and NewStrand operations complete when the persist queue receives a completion acknowledgement from the strand buffer unit.

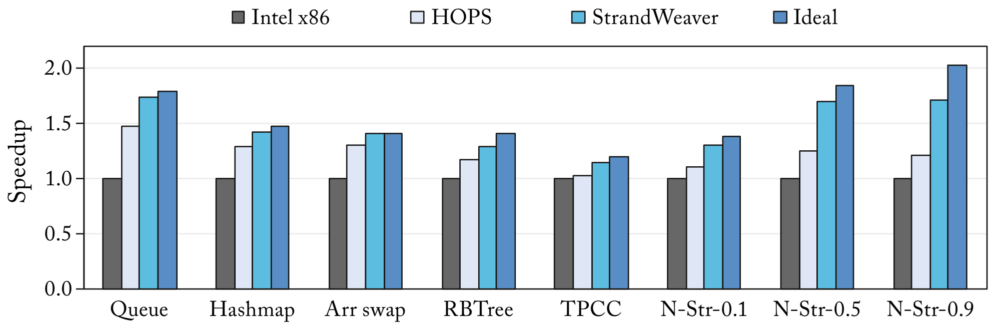


Figure 3.20: StrandWeaver performance comparison. StrandWeaver outperforms Intel x86 by 45% and HOPS by 20% on a set of microbenchmarks studied by Gogte et al. [67], as it enables higher persist concurrency and removes redundant persist dependencies.

The strand buffer unit receives strand primitives from the persist queue. Each strand buffer in the strand buffer unit manages intra-strand persist order; `CLWB`s separated by a persist barrier are ordered. `CLWB`s that lie in different strand buffers are issued concurrently to PM. On receipt of a `NewStrand` operation, the strand buffer unit updates the ongoing strand buffer to which subsequent `CLWB`s and persist barrier would be added. Subsequent `CLWB`s and persist barriers are added to a different strand buffer. A persist barrier completes when `CLWB`s ahead of it complete and retire from the strand buffer. `CLWB`s are issued when they are not ordered by prior persist barriers, and complete when strand buffers receive acknowledgement from PM controller.

PM writes can also occur due to involuntary write backs by the L1 cache of the same or another core. The strand buffer unit coordinates with the write-back buffer and L1 cache to ensure that cache write backs update PM before ongoing `CLWB` operations. It also ensures inter-thread persist order by monitoring incoming coherence traffic. It coordinates with the snoop buffer to prevent another core from "stealing" cache line ownership and prematurely writing an update back to PM, before ongoing `CLWB` operations in the strand buffers.

StrandWeaver enables higher persist concurrency. Figure 3.20 shows StrandWeaver's performance compared to an Intel x86 [68] system than implements the epoch persistency model and HOPS [66] that implements the buffered epoch persistency model. Gogte et al. [67] use strand persistency to relax persist order between the independent undo logging operations. Undo log operations created within an atomic section only need to be ordered to PM before their corresponding store operations; different undo log operations may be created and issued concurrently to the PM. Gogte et al. [67] issue undo logs concurrently to the PM by creating and issuing them on separate strands. Within each strand, they ensure a pairwise ordering between an undo log and its corresponding store operation using an intermediate persist barrier. A `JoinStrand` operation issued at the end of an atomic section assures that the undo logs and the corresponding stores issued on separate strands complete in PM before the logs are committed.

Gogte et al. [67] note that Intel x86 and HOPS designs implement epoch persistency models and so they restrict persist concurrency by serializing independent undo logs (and the corresponding stores) to the PM. StrandWeaver outperforms Intel x86 and HOPS design in all the benchmarks and achieves an average speedup of 45% and 20%, respectively. Under the Intel x86 persistency model, `SFENCE`s stall issue for subsequent updates until prior `CLWB`s complete. HOPS buffer updates within an epoch, but imposes additional ordering constraints on persists due to the deficiencies of the epoch persistency model detailed in Section 3.8. On the contrary, StrandWeaver allows precise persist order under strand persistency and improves persist concurrency.

CHAPTER 4

Hardware Mechanisms for Atomic Durability

PM programming requires that data is consistent in PM in case of a failure to enable recovery and resume execution. Recovery software inspects PM data, ensures data consistency, and restores the system. Post-failure recovery can be greatly simplified by providing failure atomicity for a set of PM updates. Failure atomicity ensures that either all or none of the updates are visible to recovery in the case of a failure. It reduces the state space that recovery might observe upon failure.

Previously, in Chapter 3, we discuss how PM systems may order memory operations to PM for correct recovery. These hardware mechanisms assure failure atomicity for individual persist operations. For instance, Intel x86 provides failure atomicity for aligned 8-byte updates to PM. That is, on failure, either the entire persist occurs in PM, or the PM location retains its prior state. Recovery would never observe partial updates. As we discuss in Section 4.1, failure atomicity at such a fine granularity complicates recovery. As such, we require additional mechanisms, either implemented in hardware, compiler systems, or as software libraries, that provide failure atomicity for a larger granularity of updates. These failure-atomic implementations rely on persistency models (discussed previously in Chapter 3) to correctly order updates to PM. In this chapter, we discuss how these implementations construct atomicity for larger updates.

Section 4.1 discusses the need for such failure-atomic mechanisms and how it simplifies post-failure recovery. Further, the chapter outlines write-ahead logging, shadow paging, and log-structuring mechanisms that allow failure atomicity for a larger granularity of updates. Sections 4.3 and 4.4 describe hardware proposals that build failure-atomic mechanisms. These proposals require minimal software annotations and enable easy programming. Finally, Section 4.5 covers mechanisms that provide coarse-grained checkpointing in hardware.

4.1 FAILURE ATOMICITY

Failure atomicity ensures data consistency. Consider a banking transaction between Alice and Bob, where Alice transfers \$50 to Bob, as shown in Figure 4.1a. Within this transaction, the amount is first deducted from Alice's account, and subsequently deposited in Bob's account. In a fault free execution, as per Figure 4.1c, Alice's balance shrinks to \$50 and Bob's balance increases to \$150 provided both of them had \$100 in their account initially. Suppose a failure occurs between the two operations as shown in Figure 4.1d. The amount is deducted from Alice's

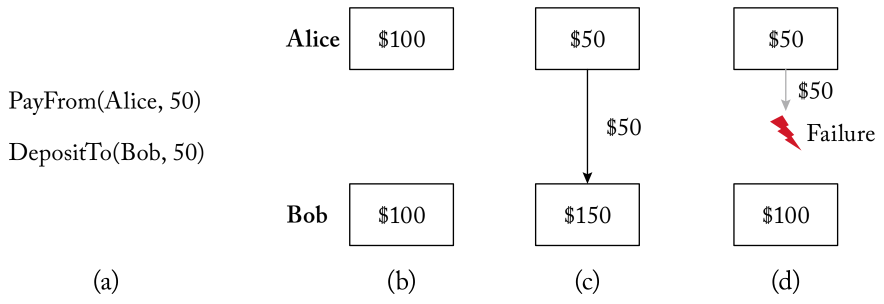


Figure 4.1: (a) Banking transaction between Alice and Bob, (b) initial amount in the bank accounts, (c) final state of the accounts when transaction completes without any failure, and (d) state when failure occurs before amount is deposited to Bob's account.

account, but not deposited to Bob's account. To avoid such a scenario, the two operations must be failure atomic. On failure, either both or none of the operations complete in PM. That is, either both Alice's and Bob's account have $100 (as per their initial state in Figure 4.1b), or the transaction between them is complete (as per Figure 4.1c).

Hardware ISA such as Intel x86 provides failure atomicity of 8-byte stores [68]. If failure occurs during an aligned 8-byte store, PM either has the new 8-byte update due to the completed store operation, or the old 8-byte value when the store operation is lost. Operations that are larger than 8 bytes are not failure atomic. Thus, Intel x86 does not guarantee that the two separate 8-byte updates to Alice's and Bob's accounts in Figure 4.1a atomically modify PM.

Failure atomicity for such updates can be achieved via write-ahead logging [32, 33, 39, 48], shadow paging [65], or log-structuring [30, 69, 70]. The mechanisms have different overheads, and their suitability for different persistent data structures depend on several factors. We discuss these mechanisms below.

4.1.1 WRITE-AHEAD LOGGING

Write-ahead logging (or journaling) is extensively used in file-systems [71–73] and databases [74–76]. It records an update to a log space in PM, before an in-place update is made. Per update, it maintains old and/or new versions of data in PM. On failure, the log is used to replay partial updates in an incomplete failure-atomic section. The logs may record the old or new version of data depending on the type of logging, namely, undo logging or redo logging.

Undo logging techniques copy the old value of a location to the log space before performing an in-place update in PM. Figure 4.2a–c shows the steps of undo logging. First, the old values are copied to the log space for all memory locations being updated in the failure-atomic region. The undo log must persist before in-place updates persist in PM. Undo logs created in the failure-atomic region are committed when all the in-place updates persist in PM. An uncommitted log indicates a partially complete failure-atomic region. On failure, any uncommitted

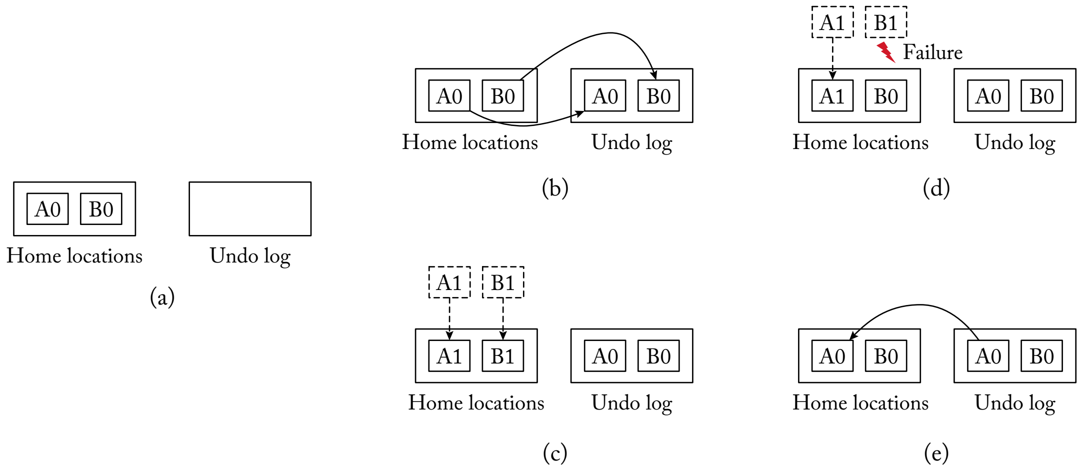


Figure 4.2: (a) Initial PM state, (b) undo logs record old values of PM locations, (c) in-place updates are made to PM after undo logs persist, (d) failure occurs before update to location B persists in PM, and (e) recovery rolls back update to location A after failure.

undo logs are used to roll back partial PM updates, as shown in Figure 4.2d–e. The old values in the log entries are copied to PM locations to restore data to its state as of the start of the failure-atomic region.

Redo logging performs updates to PM twice. Figure 4.3a–c shows the steps in a redo-logging mechanism. First, it records intent logs that contain the new values in the log space in PM. Once all the updates within a failure-atomic region are recorded in the log, it then makes in-place updates to PM. Redo logs are committed when all the modifications within a failure-atomic region are recorded in the logs. Any uncommitted redo logs indicate the failure-atomic regions that have partially recorded modifications in the log entries. On failure, such log entries are discarded as no in-place updates were made to PM data structures. The committed redo logs are used to replay updates to PM, as shown in Figure 4.3d–e. As committed redo logs maintain the new version of data, they are used to apply updates to PM data structures.

The undo and redo logging schemes have different overheads. Both logging schemes performs two writes per PM data update, either to copy the old value of data to the log under undo logging, or to apply the new value of data from the logs to the actual PM location under redo logging. This entails additional PM write traffic. Additionally, undo logging requires ordering of each log operation before each PM in-place update. This introduces significant ordering constraints on each PM update within a failure-atomic region.

To reduce ordering constraints, undo logging may require identifying the entire write set within a failure-atomic region, so that the old values of those locations may be logged, prior to any in-place updates. Unfortunately, such an approach requires identification of the write set

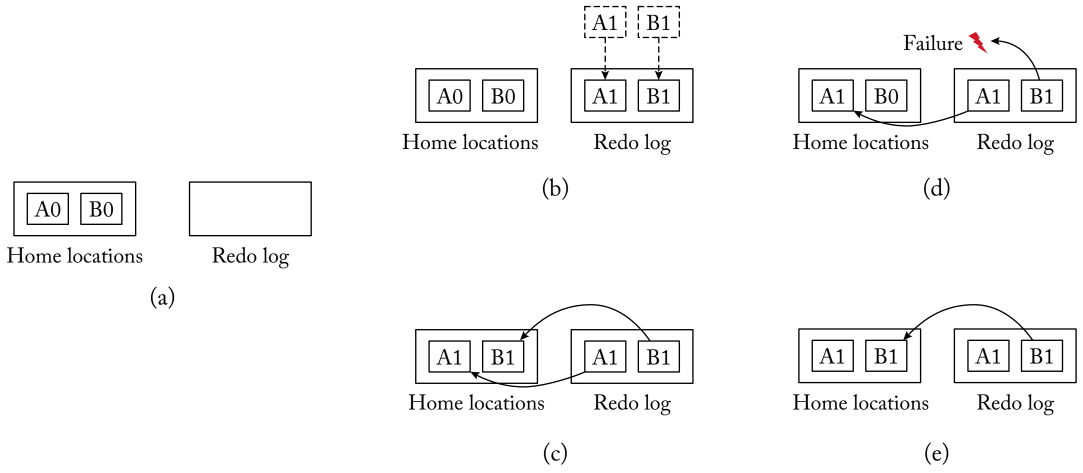


Figure 4.3: (a) Initial PM state, (b) redo logs record new values of PM locations, (c) in-place updates are made to PM after redo logs persist, (d) failure occurs before update to location B persists in PM, and (e) recovery reapplies update to location B from redo logs after failure.

so that logging operations may be coalesced. It is challenging for compilers to perform static analysis at compile time and coalesce logging operations [77–79] in the presence of ambiguous memory dependencies.

Moreover, undo logging commits logs after the in-place updates persist to PM. Any uncommitted logs discard the updates made to PM. This limits the forward progress of the program, in case of frequent failures. Failures roll back program state.

On the contrary, redo logging reduces ordering constraints on PM writes. It only requires log entries to be written completely to PM, before any in-place updates are made. On failure, redo logging applies updates recorded in the committed log entries to the PM data. It only discards uncommitted logs. The amount of forward progress of the program lost is lower in the case of redo logging.

The drawback of redo logging is that it requires re-direction of PM reads to the log space. Each read operation to PM must locate the latest copy of data. Thus, per read operation, a lookup has to performed to the log space to determine if an earlier update to the same location was recorded in the redo log. On a hit, the redo log serves the read operation. Otherwise, the read proceeds normally. The additional re-direction layer delays PM read operations. The key decision at play between choosing undo- vs. redo-logging is the trade-off between the overhead of the additional ordering constraints between logs and updates (in undo logging), and the extra re-direction layer for read operations (in redo logging).

4.1.2 LOG STRUCTURING

Log structuring [30, 69, 70] records updates sequentially in a log space in PM. Log structured updates were originally designed for hard disk-based storage that benefit from sequential block accesses [30, 69]. In disk-based storage, fixed-size blocks with updates are appended sequentially to a free region in the disk. Any reads happen to these data blocks in the log area. For instance, in a standard file system, if a data block is updated, the inode pointer is also modified to point to the new data location in the log area. Stale data blocks are subsequently reclaimed by garbage collection to free up disk space, and logs are compacted to prevent disk fragmentation.

The key benefit of log structuring is that it requires a single PM write per update (unlike a write-ahead logging mechanisms that requires new or old data to be written twice per PM update). However, log structured updates are suited to software systems that manage data in fixed-sized blocks (e.g., blocks in file system [30, 69, 80] and tuples in a database [51, 70]). PMs enable random byte-addressable access. Log structuring requires limiting the size of accesses to a block granularity. It also requires subsequent read operations to locate data in the logs. The mapping table increases required memory footprint and delays read operations. Finally, it also entails a garbage collection-like mechanism that can clean stale data.

4.1.3 SHADOW PAGING

Shadow paging is a copy-on-write mechanism that is used by several file-systems [65, 81, 82] and databases [51, 83]. It relies on a tree-like data structure in the file systems to atomically perform updates to file pages. It performs update to a free location in storage keeping the old data untouched, and atomically updates the parent node in the tree to point to the newly created page. This results in cascaded updates up the tree to the root node, as parent nodes also need to be merged up in the tree via copy-on-write.

Shadow paging, unlike write-ahead logging, does not maintain a separate log area to record updates. However, an update to a page in the tree results in several updates in the tree to merge that update. The copying overhead per update is significant. PMs enable fine-grained data manipulation. An update to few bytes under shadow paging result in amplified overhead due to large page (~4 KB) copies. Shadow paging is best suited for data structures that manage atomic updates at a page granularity.

4.2 FAILURE-ATOMIC MECHANISMS

The failure-atomic mechanisms discussed above may be built in hardware [38–40, 43, 47, 84–89], compiler systems [44, 46, 48, 90–97], or software libraries and file systems [29, 30, 32, 33, 47, 65, 98–100]. Several research proposals have emerged in the past few years that build failure atomicity of PM operations in hardware or a storage stack.

(a) Expertly designed software libraries [32, 33, 47] may implement optimized mechanisms that provide failure atomicity for PM updates. Programmers may rely on interfaces exposed by these libraries for PM programming. We discuss these software solutions in Section 5.2.

(b) Compiler mechanisms may instrument PM updates with logging code [44, 48]. The compiler builds failure atomicity as a part of high-level language implementations. In such a model, programmers need not build additional logging code for ensuring consistent PM updates. Instead, they may only annotate programs with failure-atomic regions, while compilers transparently emit logging code to ensure that the updates within the region are failure atomic. We discuss these language-level implementations in Sections 5.2.4–5.2.5.

(c) Processor systems may also provide failure atomicity for PM operations larger than a cacheline granularity. Mechanisms that provide failure atomicity in hardware focus mainly on write-ahead logging, as it is better suited for random updates at cacheline granularity. The key objective of these proposals is to bypass software overheads and build undo and/or redo logging mechanisms in hardware. The hardware logging mechanisms [38–40, 84–87] aim to provide efficient implementations for ensuring failure atomicity for PM updates in hardware. They require programmer annotations that determine the regions of code that must be failure atomic. These works ensure failure atomicity for the regions by emitting logging code for PM updates transparent to the program. They may use programmer-specified annotations to transparently emit logs for PM updates, elide or coalesce operations to reduce logging overhead, or may use temporary buffers in ADR-supported PM controller to minimize PM bandwidth requirements.

Below, we discuss proposals in the literature that build undo logging and redo logging mechanisms in hardware.

4.3 HARDWARE UNDO LOGGING

This chapter discusses undo logging mechanisms built in hardware.

4.3.1 NAIVE IMPLEMENTATION

Undo logging manages log space in PM to maintain undo log entries. The undo log entries record the old value of PM locations before they are updated in-place during the execution of a failure-atomic region. The key correctness requirement under undo logging is that the log is created and flushed to PM before the in-place updates persist. This ordering guarantees that recovery can roll back partial updates to the start of the failure-atomic region.

In a naive implementation of undo logging, hardware creates a log entry per PM update, records the value of the PM location, and flushes the log entry to PM, before the in-place update. Figure 4.4 shows the required ordering. First, hardware records the old data in the undo log entry. Once created, the log entry may reside in the volatile write-back cache. Hardware must ensure

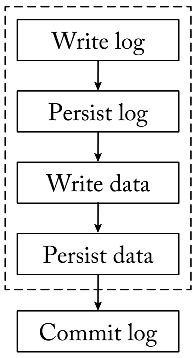


Figure 4.4: Ordering required for undo logging.

that the log entry is persistent, before it performs the in-place update, as shown in Figure 4.4. The log entries are created in PM for all the updates within a failure-atomic region. Before the log entries may commit, hardware must also ensure that in-place updates are persistent.

Hardware may record PM locations that were updated within a failure-atomic region, and flush them together at the end of the region. Delaying flushing to the end of the region allows coalescing of persist operations to the same memory locations. However, it may saturate PM write bandwidth, as the persist operations all occur at once. Alternatively, hardware may flush data from write-back caches as soon as it performs in-place updates. This negates coalescing opportunities, but evenly spreads persist operations over time.

Once values persist in-place to PM, hardware commits the log created during the execution of failure-atomic region. Committed log entries will no longer be applied after a failure to revert locations to prior values. Committed logs may be pruned and the log space reclaimed for future log entries.

The key bottleneck in such an approach is that undo logs and in-place updates persist during the critical execution path. The log-update persist ordering delays program execution. ATOM [39] notes that undo logs may persist in the background, and moves undo logging persists out of the critical execution. We discuss their proposal next.

4.3.2 OPTIMIZED UNDO LOGGING

ATOM [39] performs undo logging in hardware transparent to the programmer and implements several optimizations to move undo logging operations out of the critical execution path. ATOM ensures log-to-store ordering in hardware. It ensures that the log entry is created before the actual in-place update executes. It also guarantees that the log persists to PM before the in-place update.

It introduces two primitives, Atomic_Begin and Atomic_End, that programmers can use to define the boundaries of a failure-atomic region. Figure 4.5b shows the programming model

```
Atomic_Begin()

While (!Done) {
    WriteLog()
    PersistLog()
    WriteData()
}

PersistAllData()
CommitLog()

Atomic_End()
```

(a)

```
Atomic_Begin()

While (!Done) {
    WriteData()
}

FlushAllData()

Atomic_End()
```

(b)

```
Atomic_Begin()

While (!Done) {
    log_load()
    log_flush()
    WriteData()
}

Atomic_End()
```

(c)

Figure 4.5: (a) Programming model for naive undo logging, (b) ATOM's [39] programming model, and (c) Proteus's [40] programming model.

in ATOM. The Atomic_Begin primitive initiates logging in hardware for the updates in the failure-atomic region. Atomic_End terminates the region and commits the logs. ATOM still requires programmers to flush the in-place updates to PM before the failure-atomic region terminates. These primitives provide durability, but not isolation. ATOM relies on programmers to synchronize operations to shared memory locations and avoid data races.

Figure 4.6 shows hardware architecture of ATOM. ATOM maintains a log manager adjacent to the L1 cache that manages the logging operations. The operating system initializes log space in PM, where the log entries are written. When a processor issues a PM update to the L1 cache, assuming a hit in the cache, the log manager creates a log entry that records the old value and memory address of the PM location. It issues the log entry to PM and stalls completion of the in-place update until it receives acknowledgement of log completion from the PM controller. The in-place update modifies L1 cache when the log entry persists in PM.

The processor can perform in-place updates to the same PM location multiple times in a failure-atomic region. For correct recovery, it is sufficient that the undo log records the oldest value (the value prior to the start of the region) of the PM location. ATOM obviates the need to create separate log entries for each update to the same PM location. It creates the log entry for the first update. ATOM maintains a log bit in L1 cache, that is set when the log entry for the first update persists. It does not create logs for the subsequent updates to the same location, provided the log bit is set in PM. Of course, if the cache line is evicted (e.g., due to a cache conflict) between the two subsequent updates, ATOM performs logging for the second update as well. On failure, undo logs are replayed in their reverse order of creation to restore PM state to the start of the partially completed failure-atomic region.

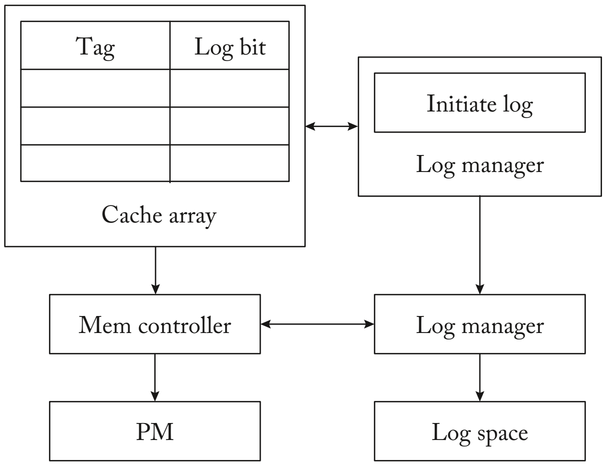


Figure 4.6: ATOM's [39] hardware architecture.

ATOM [39] does not assume an ADR-supported PM controller (possibly because it was proposed before Intel mandated ADR-support for PM controllers [55]). Consequently, it must also ensure that the in-place update does not reorder before the log operation in the PM controller. To avoid the potential reordering, it locks the cache line in the PM controller until the log persists. This prevents the corresponding in-place update from bypassing the log in the PM controller and modifying the PM location before the log persists. With ADR-support, this additional functionality of the PM controller is not required: logs are persistent as soon as they reach the PM controller. Logs are ordered before in-place updates to the PM controller by the log manager in the L1 cache.

On each cache miss, ATOM notes a redundant movement of data during log creation: L1 cache controller fetches cache block from PM into the cache, the log manager records this old data into the log entry, and it then rewrites the log entry back to the PM. ATOM eliminates this redundant movement of old data on a cache miss. On each cache miss, it actively creates a log entry in the PM controller, copies the old data fetched from PM to the log entry and then persists the log entry back to the PM. This removes logging operation from the critical execution path of the store operations that miss in the L1 cache.

4.3.3 SOFTWARE-ASSISTED HARDWARE LOGGING

Proteus [40] proposes optimized hardware logging, but expects programmers to annotate programs with logging operations. Similar to ATOM [39], it provides failure atomicity for a region of code bounded by atomic begin and end primitives. Further, it exposes log_load and log_flush operations to create a log entry and to flush the created log entry to persistence, respectively. As shown in Figure 4.5c, programmers (or compiler mechanisms) must insert the logging operations for the PM updates.

Proteus hardware mechanisms order log writes with respect to the corresponding in-place updates. This ordering ensures that log operations reach ADR-supported PM controller before the subsequent stores update the cache. As it can distinguish between log operations and in-place updates in hardware, Proteus can perform several optimizations.

First, it builds a log lookup table adjacent to the L1 cache that may coalesce logging operations to the same cache line. The log lookup table records recently issued log operations in the ongoing transaction. Proteus discards the log operation if the previous operation to the same location exists in the lookup table. This prevents duplicate logging for temporally local PM accesses.

Second, Proteus makes an interesting observation that, once written, logs are never used in a failure-free program execution. When an ongoing failure-atomic region terminates, logs are invalidated. Proteus manages ongoing log operations in the log pending queue in the ADR-supported PM controller. When the failure-atomic region terminates, it flash clears the logs that were created during that region. Proteus prioritizes in-place updates to PM before the logs, so that the log-pending queue may be flashed cleared in the PM controller, without ever being written to PM. This ensures that PM write bandwidth is utilized primarily by the in-place updates. It also reduces write traffic to PM, as the logs are discarded in the controller. The reduction in writes to PM reduces its wear out and improves device lifetime.

4.4 HARDWARE REDO LOGGING

Redo logging mechanisms log the new value of the PM location in the log, keeping the old value in place in PM. When the logs for all the updates in a failure-atomic region persist in PM, redo logs commit and the new values are copied from the logs to the home locations in PM. On failure, the committed logs in PM are used to reapply updates to their home locations. Uncommitted redo logs are discarded on failure.

Hardware mechanisms that construct redo logging mechanisms must ensure that no in-place updates persist in PM before the redo logs commit. The updates must first be recorded in the logs. As the PM home locations may record stale data, hardware must also redirect any subsequent loads to the log space to find the latest data values. All load operations must probe log entries to check if the update exists in the logs. On a miss, loads read the latest value from the home locations in PM. The additional indirection layer in hardware increases latency of load operations. The lookup operation may grow expensive as the number of log entries that need to be probed increases.

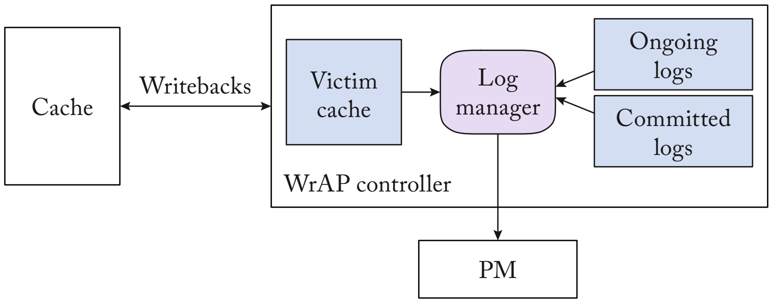


Figure 4.7: WrAP's [38] hardware architecture.

4.4.1 OPTIMIZED REDO LOGGING

Doshi et al. [38] build WrAP, a hardware redo logging mechanism that ensures redo logs persist before the corresponding in-place updates and eliminates expensive log lookups for read operations. Figure 4.7 shows the high-level hardware architecture of WrAP.

WrAP builds failure atomicity for the region of code bounded by atomic_begin and atomic_end primitives. It manages the log space in PM as a sequence of redo log entries. A redo log entry is created per PM update in the atomic region and holds the new value and address of the PM location. WrAP does not stall in-place updates until the redo logs persist in PM. It performs the updates to PM simultaneously with the redo logs, but caches those updates in the write-back cache. It does not allow in-place updates to drain out of the last-level cache and update the home locations in PM. Instead, it buffers any cache lines evicted by the LLC in a victim cache. WrAP builds a victim cache adjacent to the LLC that holds any dirty cache line evictions to PM. The victim cache is volatile and may lose its contents on failure.

WrAP issues log operations to PM using non-temporal streaming stores that bypass the volatile caches. As a failure-atomic region terminates, WrAP ensures that the log records created in the region persist to PM. It commits the redo logs once the logs persist. As the redo logs commit, PM home locations still hold the stale data, as the in-place updates are cached in LLC or in the victim cache. The logs may now be used to reapply updates to the home locations in PM, in case of a subsequent failure. The WrAP controller asynchronously copies the updates recorded in the committed redo log entries to the home locations in PM. Once all the updates are applied to the home locations in PM, committed redo logs are retired and pruned. The WrAP controller limits the log size by constantly applying updates to the home locations and pruning the retired log entries.

As the redo logs retire, WrAP also removes the updates in the victim cache that were performed during the corresponding failure-atomic region. These updates in the victim cache are no longer needed, as PM home locations record the new data. In WrAP, cache evictions never update PM home locations.

Load operations find the latest data through the shared cache hierarchy, similar to the DRAM locations. In case of a hit, a load operation finds the latest value in the cache. Otherwise, load operations look in the victim cache, as the victim cache buffers the LLC evictions. If the victim cache does not hold the update, the load operation fetches the latest data from the home location in PM. WrAP eliminates the need to traverse the redo logs, as the write-back cache and victim cache hold any intermediate updates.

WrAP proposes an elegant redo logging solution in hardware. However, it requires re-reading the redo logs and copying the updates to the PM home locations. The additional read bandwidth due to logging may interfere and delay the application reads to PM. WrAP is also limited by the victim cache size. If the victim cache overflows, WrAP records the evicted LLC cache lines in volatile DRAM buffers. The lookup latency to these DRAM buffers may also delay application reads.

4.4.2 DURABLE HARDWARE TRANSACTIONS

DHTM [84] proposes a redo logging approach to provide ACID transactions in hardware. ACID transactions ensure that either all or none of the operations are performed (atomicity), provide data consistency, have isolated local view of memory per logical thread (isolation), and assure that the updates are durable (durability). Prior works [38–40] that we discussed in Sections 4.3.2–4.4.1 provide atomicity, consistency, and isolation of failure-atomic regions, either via software transactional memory or through locking. DHTM leverages hardware transactional memory (HTM) (e.g., Intel's Restricted Transactional Memory) to provide atomicity, consistency, and isolation, and builds redo logging in hardware to ensure durability of transactions.

DHTM combines commercial HTMs and hardware redo logging to provide failure-atomic transactions. DHTM ensures atomic visibility of transactions, similar to HTMs. It maintains read and write sets for the ongoing transaction in the L1 cache. The L1 cache detects conflicts when it receives incoming coherence requests from another core for a cache line in its read and write sets. One of the two conflicting transactions aborts, as determined by the conflict resolution policy. When a transaction commits, the read and write sets are cleared.

DHTM builds on top of atomic transactions to provide durability. Similar to WrAP, DHTM manages in-place updates in the L1 cache. The L1 cache controller creates a redo log entry when an in-place update is performed in the L1 cache. DHTM builds a log buffer alongside the L1 cache that coalesces any redo log entries created for the same cache line. This minimizes redo logging traffic to PM. DHTM commits the transaction by persisting all the redo log entries created during that transaction. At this point, the transaction is failure atomic, and redo logs are used to replay the in-place updates in case of a failure. Once the transaction commits, the L1 cache controller flushes in-place updates to PM.

Commercial HTMs manage transaction metadata in the L1 cache and disallow any updates to overflow out of the L1 cache. This is because the conflict detection and metadata management designed for smaller L1 caches fails to scale to larger shared LLCs. DHTM maintains an overflow list in PM to allow L1 cache overflow. The overflow list records the cache lines that overflowed from the L1 cache to the shared LLC during an ongoing transaction. On commit, DHTM uses the overflow list in PM to identify the cache lines that overflowed and flushes them to PM.

DHTM uses a sticky coherence state in the LLC (similar to LogTM [101]) to detect transaction conflicts. During an ongoing transaction, if a core modifies a cache line in the L1 cache that overflows to the LLC, the LLC still records that core as the owner of that cache line. Any subsequent coherence requests from another core are directed to the owner, where the transaction conflict is detected.

4.5 HARDWARE CHECKPOINTING MECHANISMS

Write-ahead logging mechanisms in hardware provide fine-grained logging of PM state. They provide failure atomicity for programmer-defined failure-atomic regions, which are typically small, ranging from a few tens to hundreds of PM updates. Fine-grained logging imposes frequent ordering constraints on logs and in-place updates. It also results in frequent copying of data between logs and home locations in PM.

Several coarse-grained checkpointing mechanisms have emerged [42, 43, 102], which eliminate data copy per PM update. These mechanisms are similar to earlier frameworks [103–105] proposed for crash-recovery systems for volatile memory, which frequently log program execution via checkpoints. Checkpointing mechanisms involve storing a snapshot of the current working copy of the data persistently in PM. These mechanisms checkpoint program state at a fixed interval in PM. Volatile execution operates on a working copy of data, which is then copied to the checkpoint in PM at the end of each interval. On failure, recovery uses a checkpoint to recover the system. Depending on the checkpointing granularity, these mechanisms may lose substantial application forward progress on a failure. Below, we discuss mechanisms that build checkpointing in hardware.

4.5.1 COARSE-GRAINED CHECKPOINTING

ThyNVM [42] provides a software-agnostic checkpointing mechanism in hardware for PM systems. It checkpoints thread context (e.g., CPU registers and dirty cache blocks) and PM data at a fixed time interval. On failure, the checkpoint stored in PM is used by recovery software to restore the system and restart program execution.

ThyNVM hardware divides program execution into fixed time intervals, 10 ms each. Each interval consists of two phases: an execution phase and a checkpointing phase. During the execution phase, ThyNVM operates on the working copy of the PM data (similar to redo logging), leaving the prior checkpoint copy in PM untouched. At the end of each execution phase, ThyNVM checkpoints the working copy of the PM data created during that interval. In a trivial checkpointing framework, hardware may checkpoint the program state at the end of each execution phase, thereby stalling program execution. However, ThyNVM notes that the checkpoint phase incurs a performance overhead of as much as 35% if it is serialized with program execution. Instead, ThyNVM overlaps the checkpointing phase of the prior interval with the execution phase of the ongoing interval. In doing so, checkpoint and execution phases of the two consecutive intervals may operate on the same data blocks. ThyNVM ensures concurrency of the two phases without any data corruption. This concurrency allows program execution to proceed while the checkpoint is created for the prior interval.

ThyNVM also leverages a hybrid DRAM+PM memory system to manage its working copy during the execution phase. It aims to minimize the latency required to checkpoint the working copy to PM and the space overhead of the metadata that records the locations of the working and checkpoint copies of data. The two metrics change depending on the location of the working copy. Maintaining the working copy in DRAM allows faster access (as DRAM is faster than PM), but requires copying data from DRAM to PM. Keeping the working copy in PM requires no data copy, as the working copy may be checkpointed in place. However, this approach requires large mapping tables to record locations of the working copy and checkpoint.

ThyNVM manages large working sets (e.g., at a page granularity) with dense, sequential updates in DRAM, to allow faster updates with small metadata overhead. Conversely, it maintains small working sets (e.g., updates at a cache line granularity) with sparse, scattered updates in PM, to enable faster checkpoints. It builds a translation layer in hardware to map accesses to working and checkpoint copies. ThyNVM dynamically switches between the two checkpointing strategies as the application access pattern changes.

4.5.2 CHECKPOINTING ON POWER FAILURE

Whole-system persistence (WSP) [52] proposes checkpointing the program context, including CPU registers, program stack, and caches, using the residual supply power on a power failure. WSP adopts NVDIMM-based persistent storage (previously discussed in Section 1.2) to store the checkpoint before each power failure.

WSP uses a programmable microcontroller that monitors the power supply to the system. As soon as it detects the power failure, it triggers an interrupt to a control processor via a serial line. The control processor is a processor responsible for managing the flush operations in the system. WSP estimates that the time window available to complete the flush operations to NVDIMM ranges from 10 ms to 400 ms, measured on AMD and Intel platforms. The control processor notifies other processors via interprocessor interrupts and initiates routines to flush the program context to the NVDIMM. WSP evaluates that the hardware caches (as big as 12 MB size) may be entirely flushed to NVDIMM within 5 ms, well within the power window. Once the context is flushed, the control processor notifies the NVDIMM controller of the imminent power failure. NVDIMM then flushes DRAM contents to the backup Flash using the reserve power backup stored in its ultracapacitors. On recovery, WSP resumes program context from the known location in NVDIMM, and restores the program state.

CHAPTER 5

Programming Persistent Memory Systems

In prior chapters, we discussed different kinds of hardware support for building high-performance PM systems. In this chapter, we will focus on some of the software work that has been done to achieve the same objective. PMs represent a significant departure from conventional storage devices and a significant body of work aims to identify and fill in any performance or ease of programming gaps that arise in the transition to PM systems. While covering the entire body of this work [29, 30, 43, 44, 47–51, 65, 91, 97–100, 106–125] is out of scope for this chapter, we hope to highlight some of the most interesting work in this space. Specifically, for advances in databases and data structures for PM, we refer the reader to the synthesis lecture by Arulraj et al. [126].

5.1 FILE SYSTEMS

In this section, we describe file systems that were built specifically for PM systems.

5.1.1 EXT4 DAX

Ext4 DAX [127] is one of the most widely used and developed PM file systems. It is a variant of the popular Ext4 file system with Direct Access (DAX) extension to optimize it for PMs. DAX allows applications to bypass the page cache and access file data directly via processor load and store instructions via the mmap system call. DAX effectively enables applications to directly access their storage data without having to be mediated through a number of slow systems software storage layers.

Conventionally, file systems employ a DRAM page cache. While the file system data resides primarily in a persistent medium like SSD or HDD, those devices are too slow to keep up with the processing capabilities of modern processors. So, frequently used data is brought into memory (into the page cache) where the processor is able to quickly access it. However, DRAM is not persistent, so periodically (or when the application requests it), file systems move dirty data from the page cache to persistent storage.

On a common case read or write operation, the file system first looks in the page cache and services the request if the data is present. If the data is not present in the page cache, it accesses the slower persistent storage device. This serialization of the lookup process works well

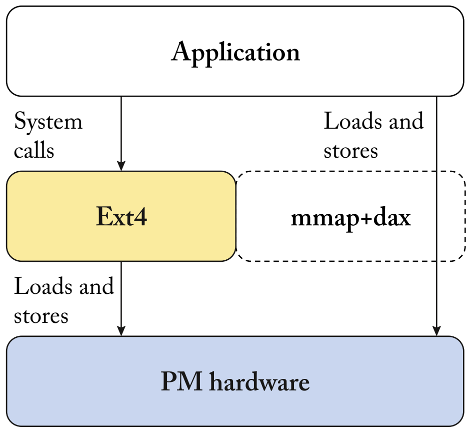


Figure 5.1: Ext4's Direct Access (DAX) extensions to accelerate file system calls on PM systems.

with slower storage devices—(1) in the case of a page cache miss, the request latency is not meaningfully impacted, the storage device access latency being the bottleneck; and (2) in the case of a page cache hit, the request latency incurred is a DRAM access rather than an SSD/HDD access, which is a significant improvement.

However, with PMs being the storage device, page caches become less attractive. If a page cache based design is employed with a PM + DRAM device, on a page cache hit, the request latency is the DRAM access latency, only a marginal improvement over the PM access latency, while in the case of a page cache miss, both DRAM access and PM access latencies are incurred. The high costs of the page cache misses are not adequately offset by the savings due to page cache hits.

Ext4 DAX is an alternative solution for PM file system design, where the page cache is removed altogether and read/write requests are sent directly to PM, as shown in Figure 5.1. Note that other structures maintained by the file system in DRAM (e.g., indexes) are not impacted by this change. An interesting implication of this change is that when a process memory maps (mmaps) a file into its user space, the process can directly access file data from user space with regular processor load and store operations and not have to navigate a number of software layers to reach file data. In conventional file systems, when a process mmaps file data, the application only is given a DRAM replica of actual file data and applications must invoke msync system calls to persist their changes. DAX alleviates the need for an msync system call to persist data and removes the task from copying data from the storage device to the DRAM when an mmap call is issued.

Due to these changes, Ext4 DAX outperforms Ext4 for a number of workloads on PM [128]. Furthermore, due to an active development community and integration into Linux, it is the most widely used PM file system.

5.1.2 NOVA FILE SYSTEM

Non-Volatile Memory Accelerated (NOVA) file system [30] is one of the more popular PM file systems. NOVA focuses most on how to provide the failure atomicity guarantees expected of file systems. Journaling/logging is a common approach, however it incurs at least double the number of writes to PM (one to the log and one to the actual location). Shadow logging is another approach, but it requires cascading the writes all the way to the top of the file system tree if multiple leaf nodes in the tree are atomically modified. So, both approaches can incur significant write amplification.

NOVA instead proposes to use log-structuring approaches. The key idea behind log structured file systems [129], originally proposed for disks, is to buffer write updates in DRAM and once enough updates have been buffered, coalesce them and write to the file system log sequentially. So, the file system data is written only once, directly to the log. And, for hard drives and SSDs, writing data sequentially (i.e., contiguous locations) is extremely important and can provide significant cost savings. When some file system data is to be read, the log is searched to find the data and returned to the user. Since the data resides only in the log, log structuring requires writing to storage only once. To minimize the costs of searching the log on reads, one can maintain indexes in DRAM to map file system data to its exact location in the log.

NOVA brings this approach to PM file systems. Conventionally, log structured file systems are viewed as being optimized for SSDs and hard drives (due to the serial write traffic they produce). NOVA instead capitalizes on the fact that log structured file systems produce much less write traffic than journaling and shadow logging based ones. However, one of the important limitations of log structured file systems is the sequential nature of the log. To ensure a constant supply of empty sequential regions, the file system has to frequently coalesce log entries and compact the log, adding overhead. One of the key ideas behind NOVA's design is the observation that since random writes are almost as fast as sequential writes on PM, the file system log does not have to be physically contiguous, it suffices if it is logically contiguous. This observation allows NOVA to design the log as a singly-linked linked list with an index in DRAM. On every write, data is appended to the log, and the index in DRAM is updated to point to the new data location (a PM page) in the log. The old data page can be reclaimed for future use.

While employing a singly-linked list based log structured file system is the key innovation behind NOVA, it also incorporates a number of design choices and optimizations to both improve performance and ease of use of the file system. For example, using a single file system log poses a scalability bottleneck when multiple threads/processes are writing to the file system concurrently. NOVA employs per-inode logs so that only writes to the same file experience delays due to contention. Furthermore, the use of a linked list for the log makes it easy to implement copy-on-write techniques and NOVA is able to provide failure atomicity guarantees for every write to the file system, alleviating programmers the worry of torn or partial writes [130]. NOVA also uses this approach to provide an atomic msync operation that atomically applies all the changes made to a mapped region (prior to the msync call) to the file system on the msync.

5.1.3 SPLITFS

Traditional file systems add large overheads to each file system operation, especially on the write path. The overhead comes from performing expensive operations on the critical path, including allocation, logging, and updating multiple complex structures. The systems community has proposed different architectures to reduce overhead. BPFS [65], PMFS [26], and NOVA [30] redesign the in-kernel file system from scratch to reduce overhead for file-system operations. Aerie [99] and Strata [131] advocate for user-space library file systems and seek to reduce overhead by not involving the kernel for most file-system operations. Despite these efforts, file-system data operations, especially writes, can have significant software overhead, 3.5–12.4x [132].

SplitFS [132] is a PM file system that seeks to reduce software overhead via a novel split architecture: a kernel PM file system (ext4 DAX) that handles metadata operations (uncommon case, these are operations that modify the file system metadata, e.g., file appends, mkdir()) and a separate user-space library file system that handles data operations (common case). The novelty of SplitFS lies in how responsibilities are divided between the user-space and kernel components, and the semantics provided to applications. Unlike prior work like Aerie, which used the kernel only for coarse grained operations, or Strata, where all operations are in user-space, SplitFS routes all metadata operations to the kernel. At a high level, the SplitFS architecture is based on the belief that if we can accelerate common-case data operations, it is worth paying a cost on the comparatively rarer metadata operations. This is in contrast with in-kernel file systems like NOVA which extensively modify the file system to optimize the metadata operations.

The key idea behind SplitFS is to transparently intercept POSIX file system calls made by the application, memory map the underlying file and serve file reads and overwrites using processor load and store instructions. SplitFS also optimizes file append operations by introducing a new file system primitive, "relink," that logically and atomically moves a file extent from one file to another without actually physically copying the data. The relink operation can be thought of as an atomic swap of file contents that can be used to implement copy-on-write mechanisms and help provide strong failure atomicity guarantees.

Apart from lowering software overhead, the SplitFS architecture leads to several interesting benefits. First, instead of re-implementing kernel file-system functionality, SplitFS can take advantage of the mature, well-tested code in ext4 DAX for metadata operations. This decision has both advantages and disadvantages. Using a robust, widely used and actively worked on file system like ext4-DAX allows SplitFS to leverage the work being done on ext4-DAX. However, ext4-DAX has higher software overhead than a more kernel optimized file system like NOVA, and this higher overhead is incurred by SplitFS users too, albeit much less frequently. Second, the user-space library file system in SplitFS allows each application to run with different independent failure atomicity guarantees. Since not all applications require the same guarantees, allowing individual applications to choose an appropriate failure atomicity guarantee makes it easier for the developer to trade-off performance for programming ease.

5.1.4 STRATA

Strata [131] is a user-space file system that targets a different kind of memory hierarchy than the one we have been reasoning about so far in this book. Strata is built with the vision that due to costs, it is likely that a future computing system might have PM, SSDs, and even HDDs all organized as layers of storage devices layered in descending order of performance and increasing order of density. Strata is also inspired by the trend in modern applications to bypass the kernel for common case operations and alleviate costs incurred due to making kernel calls (context switch overheads, kernels have long, complex code paths [133]).

Strata organizes the file system as two major components, a fast user-space log in PM that can quickly absorb application writes (and service reads where possible) and an asynchronously-managed longer term storage that resides partly in PM and then spans SSD, and HDD or multiple layers of PMs, SSDs, and HDDs based on the memory hierarchy of the system. In the common case, applications read and write to the PM log and incur little software overhead. If the log fails to service a read, then the longer-term storage is accessed. Periodically, the contents of the log are coalesced and applied to the longer term storage and once the changes have been propagated to long-term storage, the log is truncated. Applying changes from the log to the long-term storage is performed asynchronously to application reads and writes to the log. Furthermore, the log is private to each process while the long-term storage is shared among all processes. Strata implements appropriate concurrency control mechanisms to handle both inter-process communication and conflicting updates from the private logs to the shared longer term storage.

Strata organizes the user-space log to be a synchronous data structure, so applications can be sure that their data has persisted when a write call returns. This approach is made possible by the low-latencies of PMs and alleviates developers from having to contend with asynchronous system calls, a complex and error-prone approach to high-performance storage software. With Strata, the developer never has to invoke the fsync operation. The long-term storage area is organized as an LSM-style tree spanning layers of PM, SSD, and HDD with hotter data being maintained in the upper layers of the tree for faster access and colder data being pushed down to the bottom for more efficient storage.

Strata's design choice of decoupling the client facing portion of the file system (the user-space log) and the shared long-term storage area allows Strata to tailor the format and management of data to the different logical components and the hardware layers across which the file system is operating. For example, in a system where the user-space log is maintained in PM and the long-term storage is maintained in SSDs, when moving contents from the log to the long-term storage, Strata batches and sequentializes updates to the long-term storage area to extract the best performance from the SSD.

5.2 PROGRAMMING PM SYSTEMS

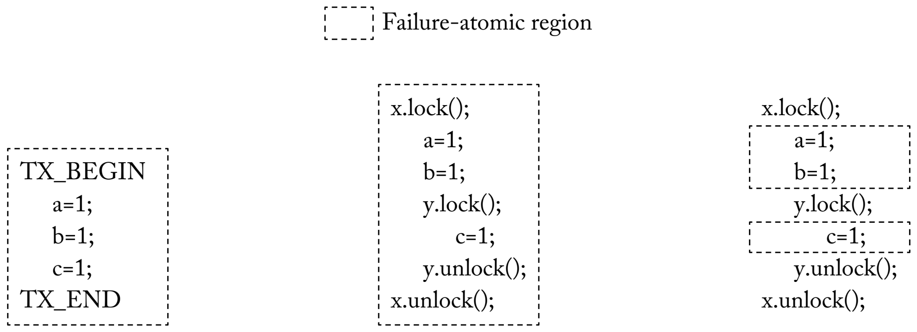
Most proposed persistency models have been specified at the instruction set architecture (ISA) level. That is, programmers must reason about recovery correctness at the abstraction of assembly instructions, an approach which is error prone and places an unreasonable burden on the programmer. The programmer must invoke ISA-specific mechanisms (via library calls or inline assembly) to ensure persist order, and often must reason carefully about compiler optimizations that may affect the relevant code. Since the ISA mechanisms differ in sometimes subtle ways, it is hard to write portable recoverable programs. Kolli et al. [45] argue for a language-level persistency model that provides mechanisms to specify the semantics of accesses to PM (including with respect to program failures) as an integral part of the programming language, just as language-level memory consistency models enable precise specification of the semantics of memory accesses from concurrent threads.

A language-level persistency model provides a single, ISA-agnostic framework for reasoning about persistency and can enable portability of recoverable software across language implementations (compiler, runtime, ISA, and hardware). Furthermore, a language-level model prescribes precise requirements on the implementation, allowing implementers to reason about the correctness of compiler and hardware optimizations. They explore a taxonomy of guarantees that a language-level persistency model might provide. Stronger guarantees (e.g., failure-atomicity of critical sections) make writing recoverable software easier but impose substantial requirements on the implementation, which entail performance penalties. Weaker guarantees complicate reasoning about recovery, but provide greater implementation freedom and performance. The weaker guarantees relax atomicity of critical sections and instead provide only ordering guarantees for individual persists. Ordering guarantees on individual persists allow synthesis of higher granularities of atomicity via logging.

5.2.1 TRANSACTIONAL FAILURE ATOMICITY

One approach to simplifying PM programming is to borrow the concept of "Transactions" from databases. A transaction is simply a set of PM operations that are all guaranteed to be failure-atomic, as shown in Figure 5.2a. Developers instrument their programs identifying the start and end of transactions and a corresponding transaction library or transaction processing engine is responsible for tracking all the memory operations within a transaction and appropriately logging them to ensure failure-atomicity.

Some of the earliest systems to propose and implement transactional failure-atomicty semantics for PM systems are NV-Heaps [32] and Mnemosyne [33]. Both systems not only provide transactional semantics for their users, but additionally provide a memory-like interface that developers can use to allocate and discover persistent objects. Furthermore, NV-Heaps provides numerous features like pointer safety that are critical for PM systems. For example, a pointer to a volatile memory location (in DRAM) from a PM location inherently loses meaning after a failure and dereferencing it post-failure could cause incorrect program outcomes. So, NV-



```
TX_BEGIN
    a=1;
    b=1;
    c=1;
TX_END
```

(a) Failure-atomic transactions

```
x.lock();
    a=1;
    b=1;
    y.lock();
        c=1;
    y.unlock();
x.unlock();
```

(b) Outer critical section

```
x.lock();
    a=1;
    b=1;
    y.lock();
        c=1;
    y.unlock();
x.unlock();
```

(c) Synchronization-free regions

Figure 5.2: This figure shows the different granularities of failure atomicity that a language could provide: (a) transactions, (b) outer critical sections, and (c) synchronization-free regions.

Heaps protects users from such scenarios by disallowing pointers to volatile memory locations from ever being stored in PM locations. Both NV-Heaps and Mnemosyne provide a number of low-level primitives that developers can use to directly and safely access PM.

While NV-Heaps and Mnemosyne are not the first works to provide persistent objects with a transactional interface, they were the first to design such systems for PM. The low-latency of PMs compared to traditional storage devices requires PM object stores to not involve the operating system and other systems software in the critical path of PM accesses. Furthermore, both systems provide transactional semantics by using logging techniques (redo logging for Mnemosyne and undo logging for NV-Heaps) optimized for PM. Similar to NV-Heaps and Mnemosyne, Intel developed and released Persistent Memory Development Kit (PMDK) [90] that provides an open-source set of libraries directed toward enabling easier PM programming.

5.2.2 INTEL'S PMDK LIBRARIES

PMDK provides different libraries that enable developers with a varying set of tools to program for PMs. The `libpmem` library [90] provides a low-level PM programming support that allows programmers to manually annotate and track every PM update and explicitly flush it from volatile to persistent domain. It requires programmers to build custom logging mechanisms, if failure atomicity coarser than an individual PM update is desired. To enable developers that do not desire such custom programming support, `libpmemobj` library simplifies PM programming by building on top of low-level `libpmem` library and provides a transactional interface to access PM data structures. It enables a transactional object store interface to PM, with additional memory management and locking capabilities. PMDK also provides other specialized libraries, such as `libpmemblk` and `libpmemlog`, that aid in atomic updates to PM-resident blocks and logs.

5.2.3 DEFERRED COMMIT TRANSACTIONS

Kolli et al. [50] considers mechanisms to implement software transactions for PM with minimal persist dependencies. Implementing failure-atomic transactions for PMs requires that writes to the PM are ordered. For instance, under write-ahead logging, ongoing transactions may commit only after the log records for that transaction persist in PM. Kolli et al. [50] notes that, while ordering operations to the PM within a transaction is required for correctness, minimizing such ordering dependencies is crucial for enabling high-performance transactions. In particular, they consider undo logging as an example to show how an intuitive transaction implementation, which they call synchronous commit transactions (SCT), introduces unnecessary persist dependencies and overconstrains the persist critical path. Under SCT, all steps within a transaction—acquiring locks for isolation, preparing an undo log entry and recording it in the PM, mutating data in-place in the PM, committing the transaction and releasing locks—must be performed in order (we discussed undo logging in detail earlier in Section 4.1). Whereas performing these steps in order admits simple and intuitive design, it also requires performing all the logging operations under the locks which overconstrains persist dependency graph. Kolli et al. [50] proposes deferred commit transactions (DCT) that aims to generate shorter persist dependency graph than SCT.

DCT reduces persist dependencies by releasing locks after mutating data in-place within a transaction and deferring commit until later. This ensures that the subsequent independent transactions may proceed without waiting for prior transactions to commit. However, deferring commits until later also leads to a new challenge of correctly ordering the commit operations across all the outstanding (potentially conflicting) transactions. That is, a lack of proper synchronization may result in conflicting transactions to commit out of order, resulting in an incorrect recovery after failure. The commit order must match the order in which the locks were originally acquired for the transactions to ensure transaction serializability. DCT guarantees transaction serializability by modifying transactions to explicitly track their predecessor conflicting transactions and stalling their commit until all the prior conflicting transactions commit. To enable this, DCT extends each lock with a pointer to the last transaction that acquired that lock. When a transaction acquires all the locks within its lock set, it records pointers to all the prior outstanding transactions that conflict with the current transaction. Before that transaction commits, it must verify that all the prior conflicting transactions have committed. Consequently, it polls on the recorded pointers to the prior transactions until they commit, before committing the current transaction. This allows independent transactions to proceed and commit in any order, but guarantees that the conflicting transactions never commit out of order.

5.2.4 FAILURE ATOMICITY FOR OUTER-CRITICAL SECTION

While the approaches discussed earlier provide transactional failure atomicity, Chakrabarti et al. [48] take an alternative approach by proposing durability semantics for lock-based programs. While durable transactions provide an attractive model for PM programming as they provide durability, atomicity, consistency and isolation for transactions, they argue that significant effort is required to convert lock-based programs to transaction-based programs. Moreover, widely used synchronization mechanisms like conditional waits or complex locks do not readily compose with transactional models. Chakrabarti et al. [48] propose ATLAS, which provides durability semantics for lock-based multi-threaded programs. ATLAS guarantees failure atomicity for outermost critical sections, where a critical section is the code region bounded by lock and unlock synchronization primitives, as shown in Figure 5.2b. In a properly labeled program, data structures are inconsistent only within the failure-atomic regions. ATLAS extends the durability guarantee to the outermost critical sections, so that recovery always observes a consistent PM state as it existed when no locks were held in the program. Failure atomic outermost critical sections are appealing, as they ensure that recovery always observes a sequentially consistent PM state.

Unlike transactions, locks may nest as shown in Figure 5.2b. ATLAS must ensure durability of *outermost* critical sections so as to ensure that recovery never observes a partially completed critical section, which may have been interrupted by a failure. It employs undo logging for the updates within an outermost critical section to ensure its failure atomicity. On failure, it uses undo logs to roll back the program state to the last completed failure-atomic region on each thread. ATLAS must ensure that recovery never observes any side-effects of the updates that were rolled back after failure. That is, if the updates made by inner critical sections (within a partially complete outermost critical section) propagate to other threads, those updates (and their side-effects) need to be rolled back as well on failure. ATLAS logs these happens-before ordering relations between the threads through the synchronizing operations. It uses these happens-before ordering relations to roll back the updates in their reverse order of creation upon failure.

5.2.5 FAILURE ATOMICITY FOR SYNCHRONIZATION FREE REGION

Gogte et al. [44] note that failure atomicity at the granularity of outer-most critical sections is appealing to programmers as they prevent any non-sequentially consistent updates to be visible to recovery. However, they also note that such an approach provides unclear semantics for updates that lie outside of critical sections. They also do not generalize to several synchronization constructs, such as condition variables. Moreover, ATLAS requires high-overhead cycle detection among several threads to identify updates that must be jointly made failure atomic. Gogte et al. [44] propose failure atomicity at the granularity of synchronization-free regions (SFRs), which represent the code regions delimited by synchronization operations (such as acquire and release operations) or system calls, as shown in Figure 5.2c. Failure atomicity for SFRs ensures that the state after failure always conforms to the program state at a synchronization operation on each thread. As the SFRs are data-race free during fault-free program execution, failure atomicity for SFRs additionally enables a sequential consistency guarantee to failure recovery.

For programs without nested critical sections, SFRs and critical sections represent the same regions of the code. However, with nested critical sections, multiple SFRs may span an outermost critical section, allowing partially completed critical sections to be visible post recovery. Programmers must be cognizant of such a possibility; if courser-grained failure atomicity is desired, programmers must build logging mechanisms for partially completed critical sections.

Gogte et al. [44] specify persistency semantics for multi-threaded programs in the C++ memory model. The C++ memory model uses synchronization operations to provide intra- and inter-thread memory order. They extend durability semantics by ensuring that the memory operations become durable in an order that is consistent with the order in which they are allowed to become visible. They log happens-before ordering relations between synchronization operations to ensure that updates are made durable in the same order. They propose two designs, Coupled-SFR and Decoupled-SFR, that provide failure atomicity for updates through undo logging and that vary in simplicity and performance.

Coupled-SFR design. This design couples the persistence of a program's state with its visibility. That is, on failure, recovery is guaranteed to observe the program state at the latest completed synchronization operation on each thread; persistent state lags the execution by at most one (incomplete and ongoing) SFR. In this design, PM mutations are flushed to PM at the end of each SFR and the undo logs created within that SFR are immediately committed. The key advantage of the Coupled-SFR design is that each thread is required to track only log entries for the updates performed within its ongoing SFR. The thread-private nature of Coupled-SFR design stands in stark contrast to ATLAS that must log inter-thread ordering dependencies to identify log entries that must be made durable atomically. The PM state after recovery is easy to interpret, as it conforms to the latest synchronization operation on each thread. The primary disadvantage of the Coupled-SFR design is that it must flush all PM updates and commit log entries at the end of each SFR, exposing much of the PM persist latency on the critical execution path.

Decoupled-SFR design. The Decoupled-SFR design trades off simplicity for performance. It decouples visibility of PM updates from the frontier of persistent state and allows persistent state to lag execution on each thread. Failure recovery still requires that updates with the SFRs are failure atomic and that they become persistent in the same order as the SFRs execute. Absent this guarantee, recovery may observe PM state that does not correspond to the state of a fault-free program execution. To ensure correct ordering, Decoupled-SFR records happens-before ordering relations between synchronization operations by: (1) adding acquire and release annotations to the undo logs, (2) creating and maintaining per-thread logs in program order to capture intra-thread ordering, and (3) tracking inter-thread order by maintaining a monotonic sequence number across synchronizing acquire and release pair.

During program execution, Decoupled-SFR periodically invokes a flush-and-commit mechanism to prune and commit log entries, following the recorded intra- and inter-thread program order. Periodic flush-and-commit also ensures that the persistent state does not fall too far behind execution, absent which it might risk losing forward progress in the event of failure. On failure, state observed by recovery always conforms to the program state at a frontier of past synchronization operations on each thread.

5.3 TESTING PM APPLICATIONS

While PMs offer many performance benefits over conventional storage devices, they present a new programming model and developing failure atomic software for them is error-prone. As discussed in Section 5.2, failure atomic software relies on guarantees from hardware to ensure that updates to persistent are (i) durable and (ii) they persist in a specified order. And, ISAs provide mechanisms to ensure both of these properties.

Developers of failure atomic storage software on PM systems have to reason about the ordering and durability constraints necessary to ensure failure atomicity and then instrument their programs with the necessary instructions to communicate these constraints to the hardware. However, relying on these low-level primitives is an error prone approach. While some software libraries provide higher-level primitives to simplify programming for PM systems [32, 33, 44, 45, 48, 90], developers still need to understand the guarantees provided by these software libraries to correctly use them and to compose operations from different libraries in their programs. In this context, it is easy for a developer to incur an incorrect instrumentation and render their application unrecoverable in the event of a failure. Furthermore, what makes tracking and debugging such bugs difficult is that even an incorrectly instrumented program can be correct during failure-free execution and even during some failures with errors manifesting in only a subset of failure cases. This non-deterministic behavior significantly complicates debugging PM software.

PMTest [35] is a tool that aims to help developers catch incorrect use of low-level PM primitives or higher-level PM software libraries. While earlier work like Yat [134] shares goals with PMTest, they rely on an exhaustive exploration of the possible failure state search space significantly slowing down test and development workflows. Furthermore, some of these testing frameworks are also specific to a particular ISA or software library and cannot be easily extended. PMTest instead offers developers a fast and flexible framework for testing their PM programs.

Unlike prior testing mechanisms, PMTest achieves flexibility by abstracting away the finer details of PM programming from the developer. Instead, PMTest provides developers with two key "assert-like" primitives that allow developers to check that the necessary ordering and durability constraints for ensuring crash consistency are met by the PM program under test. If the expected constraints are not met (for example, variable A must persist before variable B), PMTest throws an error that a developer may use to debug their program. The PMTest tool is responsible for synthesizing the assert-like primitives down to machine instructions, relieving the developer of this burden. Using these primitives developers can check if their PM programs are executing in the expected manner. However, the developers are still responsible for correctly using these primitives and rely on their knowledge of the program semantics to accurately instrument their program with these primitives. PM library developers can further compose these two fundamental lower-level primitives to construct higher-level primitives that are easier to use for their library users, further simplifying PM program debugging.

Apart from providing the flexibility of working across different ISAs, persistency models, and abstractions (library level versus machine instruction level), PMTest is also much faster than prior proposed PM program testing tools like Yat [134]. Prior work mainly relies on exhaustive state-space exploration of all potential failure PM states and checking for recoverability for each failure state. Instead, PMTest first prunes the search space by relying on developer instrumentation (using the above-mentioned assert-like primitives) of constraints for which to check. Furthermore, PMTest tracks PM operations and calculates the window of vulnerability for each PM write from when it gets executed by the core to when it becomes durable in PM. For writes with non-overlapping windows, it can be safely deduced that the corresponding writes will persist in order; however, writes with overlapping windows may persist in any order. Such deductions are then used to check for the validity of the conditions instrumented by the developer. These approaches make PMTest much faster than prior approaches that resort to exhaustive search space exploration of PM failure states.

One limitation of PMTest and prior PM testing tools is that they mainly limit to only investigating the pre-failure program execution phase. However, PM programs comprise another important phase, the recovery phase or post-failure execution phase, that gets executed in the event of a system failure. The interactions between the pre- and post-failure execution phases play an important role in the recoverability of a PM program. An error in either can lead to loss of recoverability. Furthermore, mismatches in "synchronization" of accesses within the pre- and post-failure executions could also lead to non-recoverability. Similar to how multi-threaded and concurrent data structures rely on an explicit orchestration of memory accesses from different threads, PM programs also rely on an explicit orchestration of PM access between the pre- and post-failure execution phases. Liu et al. define the scenario where the post-failure execution phase reads a PM location whose latest value has not yet been persisted as a "cross-failure race" and the scenario where the post-failure phase incorrectly accesses or interprets PM data as a "cross-failure semantic bug." Both these scenarios could lead to incorrect program recovery.

XFDetector [135] is a tool that tracks PM accesses both in the pre- and post-failure execution phases to check for crash consistency bugs. XFDetector works by tracking PM operations in pre-failure execution, automatically injecting system failures and then tracking PM accesses in the post-failure execution phase to determine if any cross-failure races or cross-failure semantic bugs exist within the program and flags them for developers to debug.

CHAPTER 6
Conclusion

Upcoming PMs aim to combine the byte-addressability and performance of DRAM, and the durability of traditional storage devices such as hard disks and SSDs. PMs may be accessed over a byte-addressable interface that is significantly faster than the block interface required to access traditional storage media. Unfortunately, the existing hardware, compiler, and software systems are not fully equipped to fully avail the full performance that the PMs offer.

This book surveyed a large class of works that have emerged both in the industry and the academic literature to integrate the PMs in hardware systems, compiler frameworks, and software applications. Correct recovery requires that PM operations are ordered to PM; this book detailed memory persistency models that prescribe the ordering constraints on PM operations. We defined strict and relaxed models formally, and also discussed the mechanisms proposed in the literature to implement these models in hardware. Further, this book described the logging mechanisms required to enable failure atomicity for operations that are larger than an individual persist. We detailed undo-, redo-, and shadow-logging mechanisms built in hardware that ensure that either all or none of the updates are visible to recovery in case of a failure. Finally, this book detailed several programming and software libraries that enable easier PM programming. It described software transactions, file systems, language persistency models, and testing frameworks designed for PM programming.

Bibliography

[1] Raid performance calculator, 2018. https://wintelguy.com/raidperf.pl 1, 4

[2] Jon L. Jacobi. NVMe SSDs: Everything you need to know about this insanely fast storage, 2019. https://www.pcworld.com/article/2899351/everything-you-need-to-know-about-nvme.html 1, 4, 5

[3] Jon L. Jacobi. Samsung 970 pro SSD review: The fastest m.2 NVMe drive yet, 2018. https://www.pcworld.com/article/3268829/samsung-970-pro-ssd-review.html 1, 4, 5

[4] Jon L. Jacobi. Toshiba RC100 NVMe SSD review: A good bargain SSD for laptops, 2018. https://www.pcworld.com/article/3279726/toshiba-rc100-nvme-ssd-review.html 1, 4, 5

[5] Thomas N. Theis and H.-S. Philip Wong. The end of Moore's law: A new beginning for information technology. In *Computing in Science Engineering*, 19(2):41–50, 2017. DOI: 10.1109/mcse.2017.29 1

[6] Seok-Hee Lee. Technology scaling challenges and opportunities of memory devices. In *IEEE International Electron Devices Meeting (IEDM)*, pages 1.1.1–1.1.8, 2016. DOI: 10.1109/iedm.2016.7838026 1

[7] Uksong Kang, Hak soo Yu, Churoo Park, Hongzhong Zheng, John Halbert, Kuljit Bains, SeongJin Jang, and Joo Sun Choi. Co-architecting controllers and dram to enhance dram process scaling. In *Presented at the Memory Forum*, 2014. 1

[8] Andres Lagar-Cavilla, Junwhan Ahn, Suleiman Souhlal, Neha Agarwal, Radoslaw Burny, Shakeel Butt, Jichuan Chang, Ashwin Chaugule, Nan Deng, Junaid Shahid, Greg Thelen, Kamil Adam Yurtsever, Yu Zhao, and Parthasarathy Ranganathan. Software-defined far memory in warehouse-scale computers. In *Proc. of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS'19*, pages 317–330, Association for Computing Machinery, New York, 2019. DOI: 10.1145/3297858.3304053 1

[9] INTEL OPTANE DC PERSISTENT MEMORY. https://www.intel.com/content/www/us/en/products/memory-storage/optane-dc-persistent-memory.html 1, 4

[10] Understand and deploy persistent memory. https://docs.microsoft.com/en-us/windows-server/storage/storage-spaces/deploy-pmem 1

[11] Available first on Google Cloud: Intel Optane DC Persistent Memory. https://tinyurl.com/gcp-release 1

[12] Mihnea Andrei, Christian Lemke, Günter Radestock, Robert Schulze, Carsten Thiel, Rolando Blanco, Akanksha Meghlan, Muhammad Sharique, Sebastian Seifert, Surendra Vishnoi, Daniel Booss, Thomas Peh, Ivan Schreter, Werner Thesing, Mehul Wagle, and Thomas Willhalm. SAP HANA adoption of non-volatile memory. In *Proc. VLDB Endow.*, 10(12):1754–1765, August 2017. DOI: 10.14778/3137765.3137780 1

[13] Micron NVDIMM. https://www.micron.com/products/dram-modules/nvdimm 2

[14] NVDIMM—CHANGES ARE HERE SO WHAT'S NEXT?, 2016. https://www.snia.org/sites/default/files/SSSI/NVDIMM%20-%20Changes%20are%20Here%20So%20What%27s%20Next%20-%20final.pdf 2

[15] Viking Technology NVDIMM. https://www.vikingtechnology.com/products/nvdimm/ddr4-nvdimm/ 2

[16] Rob Peglar and Wendy Elsasser. What you can do with NVDIMM-N and NVDIMM-P. https://www.snia.org/sites/default/files/PM-Summit/2019/presentations/07-PMSummit19-Peglar-Elsasser.pdf 2

[17] H. C. Yu, K. C. Lin, K. F. Lin, C. Y. Huang, Y. D. Chih, T. C. Ong, J. Chang, S. Natarajan, and L. C. Tran. Cycling endurance optimization scheme for 1 Mb STT-MRAM in 40 nm technology. In *IEEE International Solid-State Circuits Conference Digest of Technical Papers*, pages 224–225, February 2013. DOI: 10.1109/isscc.2013.6487710 3

[18] D. A. Buck. Ferroelectrics for digital information storage and switching, 1952. 3

[19] K. Aratani, K. Ohba, T. Mizuguchi, S. Yasuda, T. Shiimoto, T. Tsushima, T. Sone, K. Endo, A. Kouchiyama, S. Sasaki, A. Maesaka, N. Yamada, and H. Narisawa. A novel resistance memory with high scalability and nanosecond switching. In *IEEE International Electron Devices Meeting*, pages 783–786, December 2007. DOI: 10.1109/iedm.2007.4419064 3

[20] H. Y. Lee, Y. S. Chen, P. S. Chen, P. Y. Gu, Y. Y. Hsu, S. M. Wang, W. H. Liu, C. H. Tsai, S. S. Sheu, P. C. Chiang, W. P. Lin, C. H. Lin, W. S. Chen, F. T. Chen, C. H. Lien, and M. Tsai. Evidence and solution of over-reset problem for HfOX based resistive memory with SUB-ns switching speed and high endurance. In *International Electron Devices Meeting*, pages 19.7.1–19.7.4, December 2010. DOI: 10.1109/IEDM.2010.5703395 3

[21] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Architecting phase change memory as a scalable dram alternative. In *Proc. of the 36th Annual International*

Symposium on Computer Architecture, ISCA'09, pages 2–13, ACM, New York, 2009. DOI: 10.1145/1555815.1555758 3

[22] Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude A. Rivers. Scalable high performance main memory system using phase-change memory technology. In *Proc. of the 36th Annual International Symposium on Computer Architecture, ISCA'09*, pages 24–33, ACM, New York, 2009. DOI: 10.1145/1555815.1555760 3

[23] Moinuddin K. Qureshi, Sudhanva Gurumurthi, and Bipin Rajendran. *Phase Change Memory: From Devices to Systems*. Morgan & Claypool Publishers, 2011. DOI: 10.2200/S00381ED1V01Y201109CAC018 3

[24] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. Basic performance measurements of the intel optane DC persistent memory module. In *CoRR*, 2019. 4, 5, 15, 16

[25] Hiwot Tadese Kassa, Jason Akers, Mrinmoy Ghosh, Zhichao Cao, Vaibhav Gogte, and Ronald Dreslinski. Improving performance of flash based Key-Value stores using storage class memory as a volatile memory extension. In *USENIX Annual Technical Conference (USENIX ATC 21)*, pages 821–837, USENIX Association, July 2021. 5

[26] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. System software for persistent memory. In *Proc. of the 9th European Conference on Computer Systems, EuroSys'14*, pages 15:1–15:15, ACM, New York, 2014. DOI: 10.1145/2592798.2592814 5, 68

[27] Haris Volos, Guilherme Magalhaes, Ludmila Cherkasova, and Jun Li. Quartz: A lightweight performance emulator for persistent memory software. In *Proc. of the 16th Annual Middleware Conference, Middleware'15*, pages 37–49, Association for Computing Machinery, New York, 2015. DOI: 10.1145/2814576.2814806 5

[28] Zixuan Wang, Xiao Liu, Jian Yang, Theodore Michailidis, Steven Swanson, and Jishen Zhao. Characterizing and modeling non-volatile memory systems. In *53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 496–508, 2020. DOI: 10.1109/micro50266.2020.00049 5

[29] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. System software for persistent memory. In *Proc. of the 9th European Conference on Computer Systems, EuroSys'14*, pages 15:1–15:15, ACM, New York, 2014. DOI: 10.1145/2592798.2592814 9, 10, 11, 55, 65

[30] Jian Xu and Steven Swanson. Nova: A log-structured file system for hybrid volatile/non-volatile main memories. In *Proc. of the 14th USENIX Conference on File and Storage Technologies, FAST'16*, pages 323–338, USENIX Association, Berkeley, CA, 2016. 9, 10, 52, 55, 65, 67, 68

[31] MMAP. https://man7.org/linux/man-pages/man2/mmap.2.html 10

[32] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. NV-Heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. In *Proc. of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS XVI*, pages 105–118, ACM, New York, 2011. DOI: 10.1145/1950365.1950380 10, 11, 52, 55, 56, 70, 75

[33] Haris Volos, Andres Jaan Tack, and Michael M. Swift. Mnemosyne: Lightweight persistent memory. In *Proc. of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS XVI*, pages 91–104, ACM, New York, 2011. DOI: 10.1145/1950365.1950379 10, 11, 52, 55, 56, 70, 75

[34] Kumud Bhandari, Dhruva R. Chakrabarti, and Hans-J. Boehm. Implications of CPU caching on byte-addressable non-volatile memory programming. *Technical Report HPL-2012-236*, Hewlett-Packard, December 2012. 10

[35] Sihang Liu, Yizhou Wei, Jishen Zhao, Aasheesh Kolli, and Samira Khan. PMTest: A fast and flexible testing framework for persistent memory programs. In *Proc. of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS'19*, pages 411–425, ACM, New York, 2019. DOI: 10.1145/3297858.3304015 11, 75

[36] Sihang Liu, Korakit Seemakhupt, Gennady Pekhimenko, Aasheesh Kolli, and Samira Khan. Janus: Optimizing memory and storage support for non-volatile memory systems. In *Proc. of the 46th International Symposium on Computer Architecture, ISCA'19*, pages 143–156, ACM, New York, 2019. DOI: 10.1145/3307650.3322206 11

[37] S. Liu, A. Kolli, J. Ren, and S. Khan. Crash consistency in encrypted non-volatile main memory systems. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 310–323, February 2018. DOI: 10.1109/hpca.2018.00035 11

[38] K. Doshi, E. Giles, and P. Varman. Atomic persistence for SCM with a non-intrusive backend controller. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 77–89, March 2016. DOI: 10.1109/hpca.2016.7446055 11, 55, 56, 61, 62

[39] A. Joshi, V. Nagarajan, S. Viglas, and M. Cintra. Atom: Atomic durability in non-volatile memory through hardware logging. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 361–372, February 2017. DOI: 10.1109/hpca.2017.50 11, 52, 55, 56, 57, 58, 59, 62

[40] Seunghee Shin, Satish Kumar Tirukkovalluri, James Tuck, and Yan Solihin. Proteus: A flexible and fast software supported hardware logging approach for NVM. In *Proc. of the 50th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO-50'17*, pages 178–190, ACM, New York, 2017. DOI: 10.1145/3123939.3124539 11, 55, 56, 58, 59, 62

[41] Y. Lu, J. Shu, L. Sun, and O. Mutlu. Loose-ordering consistency for persistent memory. In *IEEE 32nd International Conference on Computer Design (ICCD)*, pages 216–223, 2014. DOI: 10.1109/iccd.2014.6974684 11

[42] Jinglei Ren, Jishen Zhao, Samira Khan, Jongmoo Choi, Yongwei Wu, and Onur Mutlu. ThyNVM: Enabling software-transparent crash consistency in persistent memory systems. In *Proc. of the 48th International Symposium on Microarchitecture, MICRO-48*, pages 672–685, ACM, New York, 2015. DOI: 10.1145/2830772.2830802 11, 63

[43] T. Nguyen and D. Wentzlaff. PiCL: A software-transparent, persistent cache log for non-volatile main memory. In *51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 507–519, October 2018. DOI: 10.1109/micro.2018.00048 11, 55, 63, 65

[44] Vaibhav Gogte, Stephan Diestelhorst, William Wang, Satish Narayanasamy, Peter M. Chen, and Thomas F. Wenisch. Persistency for synchronization-free regions. In *Proc. of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 46–61, ACM, New York, 2018. DOI: 10.1145/3192366.3192367 11, 55, 56, 65, 73, 74, 75

[45] Aasheesh Kolli, Vaibhav Gogte, Ali Saidi, Stephan Diestelhorst, Peter M. Chen, Satish Narayanasamy, and Thomas F. Wenisch. Language-level persistency. In *Proc. of the 44th Annual International Symposium on Computer Architecture, ISCA'17*, pages 481–493, ACM, New York, 2017. DOI: 10.1145/3079856.3080229 11, 70, 75

[46] Mahesh Dananjaya, Vasilis Gavrielatos, Arpit Joshi, and Vijay Nagarajan. Lazy release persistency. In *Proc. of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS'20*, pages 1173–1186, Association for Computing Machinery, New York, 2020. DOI: 10.1145/3373376.3378481 11, 55

[47] Mengxing Liu, Mingxing Zhang, Kang Chen, Xuehai Qian, Yongwei Wu, Weimin Zheng, and Jinglei Ren. DUDETM: Building durable transactions with decoupling for

persistent memory. In *Proc. of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS'17*, pages 329–343, ACM, New York, 2017. DOI: 10.1145/3037697.3037714 11, 55, 56, 65

[48] Dhruva R. Chakrabarti, Hans-J. Boehm, and Kumud Bhandari. Atlas: Leveraging locks for non-volatile memory consistency. In *Proc. of the ACM International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA'14*, pages 433–452, ACM, New York, 2014. DOI: 10.1145/2660193.2660224 11, 52, 55, 56, 65, 72, 73, 75

[49] Amirsaman Memaripour, Anirudh Badam, Amar Phanishayee, Yanqi Zhou, Ramnatthan Alagappan, Karin Strauss, and Steven Swanson. Atomic in-place updates for non-volatile main memories with Kamino-Tx. In *Proc. of the 12th European Conference on Computer Systems, EuroSys'17*, pages 499–512, ACM, New York, 2017. DOI: 10.1145/3064176.3064215 11, 65

[50] Aasheesh Kolli, Steven Pelley, Ali Saidi, Peter M. Chen, and Thomas F. Wenisch. High-performance transactions for persistent memories. In *Proc. of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS'16*, pages 399–411, ACM, New York, 2016. DOI: 10.1145/2872362.2872381 11, 65, 72

[51] Joy Arulraj, Andrew Pavlo, and Subramanya R. Dulloor. Let's talk about storage and recovery methods for non-volatile memory database systems. In *Proc. of the ACM SIGMOD International Conference on Management of Data, SIGMOD'15*, 2015. DOI: 10.1145/2723372.2749441 11, 55, 65

[52] Dushyanth Narayanan and Orion Hodson. Whole-system persistence. In *Proc. of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS XVII*, pages 401–410, ACM, New York, 2012. DOI: 10.1145/2150976.2151018 14, 64

[53] Open compute UPS & power supply, 2011. https://perspectives.mvdirona.com/2011/05/open-compute-ups-power-supply 14

[54] Jens F. Peters, Manuel Baumann, Benedikt Zimmermann, Jessica Braun, and Marcel Weil. The environmental impact of li-ion batteries and the role of key parameters—a review. In *Renewable and Sustainable Energy Reviews*, 67:491–506, 2017. DOI: 10.1016/j.rser.2016.08.039 14

[55] Intel. Deprecating the PCOMMIT instruction, 2016. https://software.intel.com/en-us/blogs/2016/09/12/deprecate-pcommit-instruction 15, 17, 59

[56] ARM. Armv8—a architecture evolution, 2016. https://tinyurl.com/arm-nvm 17

[57] Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. Memory persistency. In *Proc. of the 41st Annual International Symposium on Computer Architecture, ISCA'14*, pages 265–276, IEEE Press, Piscataway, NJ, 2014. DOI: 10.1109/isca.2014.6853222 20, 21, 31

[58] Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. In *Computer*, 29(12):66–76, December 1996. DOI: 10.1109/2.546611 21, 31

[59] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proc. of the 17th Annual International Symposium on Computer Architecture, ISCA'90*, pages 15–26, ACM, New York, 1990. DOI: 10.1145/325164.325102 21

[60] Arpit Joshi, Vijay Nagarajan, Marcelo Cintra, and Stratis Viglas. Efficient persist barriers for multicores. In *Proc. of the 48th International Symposium on Microarchitecture, MICRO-48*, pages 660–671, ACM, New York, 2015. DOI: 10.1145/2830772.2830805 26, 27, 28, 38, 39, 40, 41, 42, 43

[61] Aasheesh Kolli, Jeff Rosen, Stephan Diestelhorst, Ali Saidi, Steven Pelley, Sihang Liu, Peter M. Chen, and Thomas F. Wenisch. Delegated persist ordering. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO-49*, pages 58:1–58:13, IEEE Press, Piscataway, NJ, 2016. DOI: 10.1109/micro.2016.7783761 26, 27, 28, 29, 30

[62] Per Ekemark, Yuan Yao, Alberto Ros, Konstantinos Sagonas, and Stefanos Kaxiras. TSOPER: Efficient coherence-based strict persistency. In *IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 125–138, 2021. DOI: 10.1109/hpca51647.2021.00021 26

[63] Maranget Luc, Susmit Inria, Sarkar, and Peter Sewell. A tutorial introduction to the ARM and POWER relaxed memory models. 2012. 28

[64] Daniel J. Sorin, Mark D. Hill, and David A. Wood. *A Primer on Memory Consistency and Cache Coherence*, 1st ed., Morgan & Claypool Publishers, 2011. DOI: 10.2200/s00346ed1v01y201104cac016 31

[65] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. Better I/O through byte-addressable, persistent memory. In *Proc. of the ACM SIGOPS 22nd Symposium on Operating Systems Principles, SOSP'09*, pages 133–146, ACM, New York, 2009. DOI: 10.1145/1629575.1629589 37, 41, 42, 52, 55, 65, 68

[66] Sanketh Nalli, Swapnil Haria, Mark D. Hill, Michael M. Swift, Haris Volos, and Kimberly Keeton. An analysis of persistent memory use with whisper. In *Proc. of*

the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS'17, pages 135–148, ACM, New York, 2017. DOI: 10.1145/3037697.3037730 41, 42, 49

[67] Vaibhav Gogte, William Wang, Stephan Diestelhorst, Peter M. Chen, Satish Narayanasamy, and Thomas F. Wenisch. Relaxed persist ordering using strand persistency. In *ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 652–665, 2020. DOI: 10.1109/isca45697.2020.00060 47, 48, 49, 50

[68] Intel. Intel architecture instruction set extensions programming reference (319433-022), 2014. https://software.intel.com/sites/default/files/managed/0d/53/319433-022.pdf 49, 52

[69] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1):26–52, February 1992. DOI: 10.1145/146941.146943 52, 55

[70] Russell Sears and Raghu Ramakrishnan. BLSM: A general purpose log structured merge tree. In *Proc. of the ACM SIGMOD International Conference on Management of Data, SIGMOD'12*, pages 217–228, Association for Computing Machinery, New York, 2012. DOI: 10.1145/2213836.2213862 52, 55

[71] R. Hagmann. Reimplementing the cedar file system using logging and group commit. In *Proc. of the 11th ACM Symposium on Operating Systems Principles, SOSP'87*, pages 155–162, Association for Computing Machinery, New York, 1987. DOI: 10.1145/41457.37518 52

[72] Edward K. Lee and Chandramohan A. Thekkath. Petal: Distributed virtual disks. *SIGPLAN Notices*, 31(9):84–92, September 1996. DOI: 10.1145/248209.237157 52

[73] S. G. International. XFS: A high-performance journaling file system. http://oss.sgi.com/projects/xfs 52

[74] Voltdb. http://voltdb.com 52

[75] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. Aries: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems*, 17(1):94–162, March 1992. DOI: 10.1145/128765.128770 52

[76] Berkeley db. http://www.oracle.com/technology/products/berkeley-db/index.html 52

[77] David M. Gallagher, William Y. Chen, Scott A. Mahlke, John C. Gyllenhaal, and Wen-mei W. Hwu. Dynamic memory disambiguation using the memory conflict buffer. In

Proc. of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS VI, pages 183–193, ACM, New York, 1994. DOI: 10.1145/195473.195534 54

[78] Rakesh Ghiya, Daniel Lavery, and David Sehr. On the importance of points-to analysis and other memory disambiguation methods for C programs. In *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI'01*, pages 47–58, ACM, New York, 2001. DOI: 10.1145/378795.378806 54

[79] Simha Sethumadhavan, Rajagopalan Desikan, Doug Burger, Charles R. Moore, and Stephen W. Keckler. Scalable hardware memory disambiguation for high ILP processors. In *Proc. of the 36th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 36*, page 399, IEEE Computer Society, Washington, DC, 2003. DOI: 10.1109/MICRO.2003.1253244 54

[80] Qingda Hu, Jinglei Ren, Anirudh Badam, Jiwu Shu, and Thomas Moscibroda. Log-structured non-volatile main memory. In *Proc. of the USENIX Conference on Usenix Annual Technical Conference, USENIX ATC'17*, pages 703–717, USENIX Association, 2017. 55

[81] Dave Hitz, James Lau, and Michael Malcolm. File system design for an NFS file server appliance. In *Proc. of the USENIX Winter Technical Conference on, WTEC'94*, page 19, USENIX Association, 1994. 55

[82] Sun Microsystems. ZFS. http://www.opensolaris.org/os/community/zfs 55

[83] Jim Gray, Paul McJones, Mike Blasgen, Bruce Lindsay, Raymond Lorie, Tom Price, Franco Putzolu, and Irving Traiger. The recovery manager of the system R database manager. *ACM Computing Surveys*, 13(2):223–242, June 1981. DOI: 10.1145/356842.356847 55

[84] Arpit Joshi, Vijay Nagarajan, Marcelo Cintra, and Stratis Viglas. DHTM: Durable hardware transactional memory. In *Proc. of the 45th Annual International Symposium on Computer Architecture, ISCA'18*, pages 452–465, IEEE Press, Piscataway, NJ, 2018. DOI: 10.1109/isca.2018.00045 55, 56, 62

[85] Nachshon Cohen, Michal Friedman, and James R. Larus. Efficient logging in non-volatile memory by exploiting coherency protocols. *Proc. ACM Programming Languages*, 1(OOPSLA), October 2017. DOI: 10.1145/3133891 55, 56

[86] Jungi Jeong, Chang Hyun Park, Jaehyuk Huh, and Seungryoul Maeng. Efficient hardware-assisted logging with asynchronous and direct-update for persistent memory. In *51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 520–532, 2018. DOI: 10.1109/micro.2018.00049 55, 56

[87] Seunghee Shin, James Tuck, and Yan Solihin. Hiding the long latency of persist barriers using speculative execution. In *Proc. of the 44th Annual International Symposium on Computer Architecture, ISCA'17*, pages 175–186, ACM, New York, 2017. DOI: 10.1145/3079856.3080240 55, 56

[88] Jishen Zhao, Sheng Li, Doe Hyun Yoon, Yuan Xie, and Norman P. Jouppi. Kiln: Closing the performance gap between systems with and without persistence support. In *Proc. of the 46th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO-46*, pages 421–432, ACM, New York, 2013. DOI: 10.1145/2540708.2540744 55

[89] M. A. Ogleari, E. L. Miller, and J. Zhao. Steal but no force: Efficient hardware undo+redo logging for persistent memory systems. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 336–349, February 2018. DOI: 10.1109/hpca.2018.00037 55

[90] pmem.io: Persistent memory programming. https://pmem.io/pmdk/ 55, 71, 75

[91] Hans-J. Boehm and Dhruva R. Chakrabarti. Persistence programming models for non-volatile memory. In *Proc. of the ACM SIGPLAN International Symposium on Memory Management (ISMM)*, pages 55–67, ACM, New York, 2016. DOI: 10.1145/2926697.2926704 55, 65

[92] Joseph Izraelevitz, Hammurabi Mendes, and Michael L. Scott. *Linearizability of Persistent Memory Objects Under a Full-System-Crash Failure Model*, pages 313–327. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016. DOI: 10.1007/978-3-662-53426-7_23 55

[93] Mingyu Wu, Ziming Zhao, Haoyu Li, Heting Li, Haibo Chen, Binyu Zang, and Haibing Guan. Espresso: Brewing Java for more non-volatility with non-volatile memory. In *Proc. of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS'18*, pages 70–83, ACM, New York, 2018. DOI: 10.1145/3173162.3173201 55

[94] Terry Ching-Hsiang Hsu, Helge Brügner, Indrajit Roy, Kimberly Keeton, and Patrick Eugster. NVthreads: Practical persistence for multi-threaded applications. In *Proc. of the 12th European Conference on Computer Systems, EuroSys'17*, pages 468–482, ACM, New York, 2017. DOI: 10.1145/3064176.3064204 55

[95] Thomas Shull, Jian Huang, and Josep Torrellas. Autopersist: An easy-to-use Java NVM framework based on reachability. In *Proc. of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 316–332, ACM, New York, 2019. DOI: 10.1145/3314221.3314608 55

[96] Joel E. Denny, Seyong Lee, and Jeffrey S. Vetter. NVL-C: Static analysis techniques for efficient, correct programming of non-volatile main memory systems.

HPDC'16, pages 125–136, Association for Computing Machinery, New York, 2016. DOI: 10.1145/2907294.2907303 55

[97] Jorge Guerra, Leonardo Marmol, Daniel Campello, Carlos Crespo, Raju Rangaswami, and Jinpeng Wei. Software persistent memory. In *Presented as part of the USENIX Annual Technical Conference (ATC)*, pages 319–331, Boston, MA, 2012. 55, 65

[98] Xiaojian Wu and A. L. Narasimha Reddy. SCMFS: A file system for storage class memory. In *Proc. of International Conference for High Performance Computing, Networking, Storage and Analysis, SC'11*, pages 39:1–39:11, ACM, New York, 2011. DOI: 10.1145/2063384.2063436 55, 65

[99] Haris Volos, Sanketh Nalli, Sankarlingam Panneerselvam, Venkatanathan Varadarajan, Prashant Saxena, and Michael M. Swift. Aerie: Flexible file-system interfaces to storage-class memory. In *Proc. of the 9th European Conference on Computer Systems, EuroSys'14*, pages 14:1–14:14, ACM, New York, 2014. DOI: 10.1145/2592798.2592810 55, 65, 68

[100] Jian Xu, Lu Zhang, Amirsaman Memaripour, Akshatha Gangadharaiah, Amit Borase, Tamires Brito Da Silva, Steven Swanson, and Andy Rudoff. Nova-fortis: A fault-tolerant non-volatile main memory file system. In *Proc. of the 26th Symposium on Operating Systems Principles, SOSP'17*, pages 478–496, ACM, New York, 2017. DOI: 10.1145/3132747.3132761 55, 65

[101] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. LogTM: log-based transactional memory. In *The 12th International Symposium on High-Performance Computer Architecture*, pages 254–265, 2006. DOI: 10.1109/hpca.2006.1598134 63

[102] Amirhossein Mirhosseini, Aditya Agrawal, and Josep Torrellas. Survive: Pointer-based in-dram incremental checkpointing for low-cost data persistence and rollback-recovery. *IEEE Computer Architecture Letters*, 16(2):153–157, July 2017. DOI: 10.1109/lca.2016.2646340 63

[103] Y. Masubuchi, S. Hoshina, T. Shimada, B. Hirayama, and N. Kato. Fault recovery mechanism for multiprocessor servers. In *Proc. of IEEE 27th International Symposium on Fault Tolerant Computing*, pages 184–193, 1997. DOI: 10.1109/ftcs.1997.614091 63

[104] M. Prvulovic, Zheng Zhang, and J. Torrellas. Revive: Cost-effective architectural support for rollback recovery in shared-memory multiprocessors. In *Proc. 29th Annual International Symposium on Computer Architecture*, pages 111–122, 2002. DOI: 10.1109/isca.2002.1003567 63

[105] D. J. Sorin, M. M. K. Martin, M. D. Hill, and D. A. Wood. SafetyNet: Improving the availability of shared memory multiprocessors with global checkpoint/recovery. In

Proc. 29th Annual International Symposium on Computer Architecture, pages 123–134, 2002. DOI: 10.1109/isca.2002.1003568 63

[106] I-C. K. Chen, C-C. Lee, and T. N. Mudge. Instruction prefetching using branch prediction information. In *Proc. of the International Conference on Computer Design*, 1997. DOI: 10.1109/iccd.1997.628926 65

[107] Jian Xu, Juno Kim, Amirsaman Memaripour, and Steven Swanson. Finding and fixing performance pathologies in persistent memory software stacks. In *Proc. of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS'19*, pages 427–439, ACM, New York, 2019. DOI: 10.1145/3297858.3304077 65

[108] Tianzheng Wang and Ryan Johnson. Scalable logging through emerging non-volatile memory. *Proc. VLDB Endowment*, 7(10):865–876, June 2014. DOI: 10.14778/2732951.2732960 65

[109] Andreas Chatzistergiou, Marcelo Cintra, and Stratis D. Viglas. REWIND: Recovery write-ahead system for in-memory non-volatile data-structures. *PVLDB*, 8(5):497–508, 2015. 65

[110] Ismail Oukid, Daniel Booss, Wolfgang Lehner, Peter Bumbulis, and Thomas Willhalm. SOFORT: A hybrid SCM-dram storage engine for fast data recovery. In *Proc. of the 10th International Workshop on Data Management on New Hardware, DaMoN'14*, 2014. DOI: 10.1145/2619228.2619236 65

[111] Hideaki Kimura. FOEDUS: OLTP engine for a thousand cores and NVRAM. In *Proc. of the ACM SIGMOD International Conference on Management of Data, SIGMOD'15*, 2015. DOI: 10.1145/2723372.2746480 65

[112] Joy Arulraj, Matthew Perron, and Andrew Pavlo. Write-behind logging. *Proc. VLDB Endowment*, 10(4):337–348, November 2016. DOI: 10.14778/3025111.3025116 65

[113] Vaibhav Gogte, Stephan Diestelhorst, William Wang, Satish Narayanasamy, Peter M. Chen, and Thomas F. Wenisch. Failure-atomic synchronization-free regions. In *9th Annual Non Volatile Memories Workshop*, San Diego, CA, 2018. 65

[114] Vaibhav Gogte, William Wang, Stephan Diestelhorst, Aasheesh Kolli, Peter M. Chen, Satish Narayanasamy, and Thomas F. Wenisch. Software wear management for persistent memories. In *17th USENIX Conference on File and Storage Technologies (FAST)*, pages 45–63, USENIX Association, Boston, MA, February 2019. 65

[115] Iyswarya Narayanan, Aishwarya Ganesan, Anirudh Badam, Sriram Govindan, Bikash Sharma, and Anand Sivasubramaniam. Getting more performance with polymorphism

from emerging memory technologies. In *Proc. of the 12th ACM International Conference on Systems and Storage, SYSTOR'19*, pages 8–20, ACM, New York, 2019. DOI: 10.1145/3319647.3325826 65

[116] Jian Huang, Karsten Schwan, and Moinuddin K. Qureshi. NVRAM-aware logging in transaction systems. In *Proc. of the VLDB Endowment*, 8:389–400, 2014. DOI: 10.14778/2735496.2735502 65

[117] Joseph Izraelevitz, Hammurabi Mendes, and Michael L. Scott. Linearizability of persistent memory objects under a full-system-crash failure model. In Cyril Gavoille and David Ilcinkas, Eds., *Distributed Computing: 30th International Symposium (DISC), Proceedings*, Paris, France, September 27–29, pages 313–327, Springer Berlin Heidelberg, Berlin, Heidelberg, 2016. DOI: 10.1007/978-3-662-53426-7_23 65

[118] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy H. Campbell. Consistent and durable data structures for non-volatile byte-addressable memory. In *Proc. of the 9th USENIX Conference on File and Storage Technologies, FAST'11*, pages 5–5, USENIX Association, Berkeley, CA, 2011. 65

[119] Faisal Nawab, Dhruva Chakrabarti, Terence Kelly, and Charles B. Morey III. Procrastination beats prevention: Timely sufficient persistence for efficient crash resilience. *Technical Report HPL-2014-70*, Hewlett-Packard, December 2014. 65

[120] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. Endurable transient inconsistency in byte-addressable persistent B+-tree. In *16th USENIX Conference on File and Storage Technologies (FAST)*, pages 187–200, USENIX Association, Oakland, CA, 2018. 65

[121] Himanshu Chauhan, Irina Calciu, Vijay Chidambaram, Eric Schkufza, Onur Mutlu, and Pratap Subrahmanyam. NVMOVE: Helping programmers move to byte-based persistence. In *4th Workshop on Interactions of NVM/Flash with Operating Systems and Workloads (INFLOW)*, USENIX Association, Savannah, GA, November 2016. 65

[122] Daehyeok Kim, Amirsaman Memaripour, Anirudh Badam, Yibo Zhu, Hongqiang Harry Liu, Jitu Padhye, Shachar Raindel, Steven Swanson, Vyas Sekar, and Srinivasan Seshan. Hyperloop: Group-based NIC-offloading to accelerate replicated transactions in multitenant storage systems. In *Proc. of the Conference of the ACM Special Interest Group on Data Communication, SIGCOMM'18*, pages 297–312, ACM, New York, 2018. DOI: 10.1145/3230543.3230572 65

[123] Yiying Zhang, Jian Yang, Amirsaman Memaripour, and Steven Swanson. Mojim: A reliable and highly-available non-volatile memory system. In *Proc. of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS'15*, pages 3–18, ACM, New York, 2015. DOI: 10.1145/2694344.2694370 65

[124] Yanqi Zhou, Ramnatthan Alagappan, Amirsaman Memaripour, A. Badam, and D. Wentzlaff. HNVM: Hybrid NVM enabled datacenter design and optimization. *Microsoft Research, Technical Report MSR-TR-2017-8*, 2017. 65

[125] Youyou Lu, Jiwu Shu, Youmin Chen, and Tao Li. Octopus: An RDMA-enabled distributed persistent memory file system. In *USENIX Annual Technical Conference (ATC)*, pages 773–785, USENIX Association, Santa Clara, CA, 2017. 65

[126] Joy Arulraj and Andrew Pavlo. Non-volatile memory database management systems. In *Synthesis Lectures on Data Management*, 11(1):1–191, 2019. DOI: 10.2200/s00891ed1v01y201812dtm055 65

[127] kernel.org. Direct access for files. https://www.kernel.org/doc/Documentation/filesystems/dax.txt 65

[128] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. Basic performance measurements of the intel optane DC persistent memory module, 2019. 66

[129] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *SIGOPS Operating Systems Review*, 25(5):1–15, September 1991. DOI: 10.1145/121133.121137 67

[130] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*, 1st ed., Arpaci-Dusseau Books, August 2018. 67

[131] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. Strata: A cross media file system. In *Proc. of the 26th Symposium on Operating Systems Principles, SOSP'17*. DOI: 10.1145/3132747.3132770 68, 69

[132] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. Splitfs: Reducing software overhead in file systems for persistent memory. In *Proc. of the 27th ACM Symposium on Operating Systems Principles, SOSP'19*, pages 494–508, ACM, New York, 2019. DOI: 10.1145/3341301.3359631 68

[133] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The operating system is the control plane. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–16, USENIX Association, Broomfield, CO, October 2014. DOI: 10.1145/2812806 69

[134] Philip Lantz, Subramanya Dulloor, Sanjay Kumar, Rajesh Sankaran, and Jeff Jackson. Yat: A validation framework for persistent memory software. In *USENIX Annual Technical Conference (ATC)*, pages 433–438, USENIX Association, Philadelphia, PA, June 2014. 75, 76

[135] Sihang Liu, Korakit Seemakhupt, Yizhou Wei, Thomas Wenisch, Aasheesh Kolli, and Samira Khan. Cross-failure bug detection in persistent memory programs. In *Proc. of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS'20*, pages 1187–1202, Association for Computing Machinery, New York, 2020. DOI: 10.1145/3373376.3378452 76

Authors' Biographies

VAIBHAV GOGTE

Vaibhav Gogte is a Software Engineer at Google. He received his Ph.D. in Computer Science and Engineering from The University of Michigan, Ann Arbor in 2020. He received his Master's in Computer Science and Engineering from The University of Michigan in 2016 and graduated with a Bachelor's in Electrical and Electronics Engineering from Birla Institute of Technology and Science, Pilani, India in 2011. His research interests are in computer architecture, with a particular focus on architecture, compiler, and systems support for integrating byte-addressable non-volatile memories in future computing systems. His research has been published at several venues, including ISCA, MICRO, ASPLOS, PLDI, and FAST.

AASHEESH KOLLI

Aasheesh Kolli is a Software Engineer at Google. His interests are in computer systems architecture; his work includes processor architectures, memory subsystems, programming interfaces, and systems software. His recent research focuses on designing next-generation memory systems, particularly on persistent memory and disaggregated memory systems. Aasheesh received his Ph.D. (2017) and MS (2013) from the University of Michigan, and his BE (2011) from Birla Institute of Technology and Science, Pilani, India. His work has resulted in multiple research papers, including best paper award, at venues like SOSP, ISCA, ASPLOS, and MICRO. He was awarded the 2018 ACM SIGARCH/IEEE CS TCCA Outstanding Dissertation Award.

THOMAS F. WENISCH

Thomas F. Wenisch is Director of Engineering at Google and an Adjunct Research Scientist (formerly Professor of Computer Science and Engineering) at the University of Michigan, specializing in computer architecture. Wenisch received the NSF CAREER award in 2009, the 2012 University of Michigan Henry Russell Award, a University-wide award for extraordinary accomplishment in scholarship and teaching for early-career faculty, and the Michigan CSE Outstanding Achievement Award for accomplishments in scholarly research, classroom teaching, student mentoring, and leadership in service in 2016. In 2021, Wenisch was awarded the ACM SIGARCH Maurice Wilkes Award, the most prestigious mid-career award in computer architecture, for contributions to memory persistency and energy-efficient systems. He received his Ph.D. in Electrical and Computer Engineering from Carnegie Mellon University.