NEW ARCHITECTURES FOR NON-VOLATILE MEMORY TECHNOLOGIES

BY

APOSTOLOS KOKOLIS

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois Urbana-Champaign, 2022

Urbana, Illinois

ABSTRACT

Over the last few decades, there has been a drastic shift toward applications that need to store and operate on a large volume of data and, at the same time, provide a high quality of service to millions of users. One popular example of these types of applications is online services. For these applications, Non-Volatile Memory (NVM) technologies, such as Intel 3D XPoint, are particularly interesting. NVM has high density, it is byte addressable, and offers an attractive balance between performance and non-volatility to store data. However, the success of NVM depends on the availability of architectures that gracefully facilitate NVM's integration to current systems and take advantage of its properties.

This dissertation develops computing architectures which ease the adoption of NVMs in contemporary systems. It addresses the challenges of utilizing NVMs both (a) in single node multicore systems, as well as (b) in multi-node distributed systems.

For single node systems, this thesis first addresses the problem of data placement in systems that use both DRAM and NVM in a hybrid main memory hierarchy, and introduces *PageSeer*, which leverages page walks to predict forthcoming requests to main memory, and swaps data between DRAM and NVM to achieve high performance. Next, it examines NVM as a cache replacement and suggests *Cloak*, a technique that can hide the increased latency of NVM and take advantage of its high density when used as a last level cache by leveraging address translation to identify page reuse. Next, this thesis addresses the problem of programming for NVMs. Easy to use programming frameworks that dynamically detect persistent objects suffer from low performance. Thus, this thesis proposes *P-INSPECT*, which uses hardware to accelerate such programming frameworks.

For distributed systems, this thesis introduces the concept of *Distributed Data Persistency (DDP)*, which is the binding of consistency models with different memory persistency models. To support DDP models, it develops low-latency protocols that provide a variety of different options for persisting and replicating data in distributed systems with NVMs. It also analyzes the advantages and disadvantages of each of these options. Moreover, it proposes *HADES*, which adds hardware support for distributed transactional consistency, to bypass the costly software abstractions that increase client request latency and programming complexity, and assesses the interaction of HADES with the different persistency models. It also utilizes the network hardware to speed up data persist operations.

Overall, this thesis proposes solutions to the different NVM challenges. These solutions can pave the way for the wide integration of NVM in future systems.

To my family and friends, for their love and support.

ACKNOWLEDGMENTS

The Ph.D. has been a long and memorable journey, and there are many people that I would like to thank for helping me throughout this process.

First, I would like to thank my advisor Professor Josep Torrellas, without whom I would not have been able to complete this thesis. Throughout my Ph.D., we spent numerous hours discussing research problems with Josep, he helped me organize my work, he taught me how to approach research problems and write papers. Most importantly, he gave me the freedom to pursue a research direction that I was most passionate about and he encouraged me to persist on my goals, while at the same time he helped me stay focused on my research. His guidance and advice, his own passion for research and his invaluable experience were key to keep me motivated, and inspired me to complete my Ph.D.

Next, I would like to thank the members of my committee for their help and guidance. I worked with Professor Jian Huang extensively during the last three years of my Ph.D. and he gave me valuable advice about Non-Volatile Memories and Network Systems. I am very thankful that he was always willing to help me whenever I needed it, he was willing to stay up after hours to help me during submissions, and helped me have the right infrastructure for conducting my thesis experiments. I was also fortunate to work closely with Dr. John Kalamatianos during my internship at Advanced Micro Devices (AMD) research. John gave me the flexibility to explore new research topics during my internship and we had multiple meetings to discuss research ideas together. He showed me how to write my first patent and we continued our collaboration even after the end of my internship. Also, I had the chance to work with Professor Tianyn Xu. Tianyn's optimism against any problem was very motivating, as well as the fact that during submissions he was always there to help, and he even brought us food for late night submissions! I am also grateful to the rest of my committee members, Professor David Padua, Professor Thomas Wenisch and Professor Vijay Nagarajan whose work I had been following for years and I was looking up to. Their insightful questions during my Ph.D. examination and their comments on my dissertation helped me to better organize my thesis and conclude my dissertation.

I was also fortunate to meet the members of the i-acoma group, both past and present. In i-acoma I found not only lab mates, but also friends that helped me along the way. I want to thank Antonio, Tom, Bhargava, Serif, Namrata, Dimitris, Mengjia, Jiho, Azin, Raghav and Yasser that welcomed me when I first arrived at the US and helped me at the early years of my Ph.D. I will always remember the time we spent together at the lab, rushing for deadlines and going to conferences. I also want to thank the new students of i-acoma that were there during my senior years, Gerasimos, Antonis, Jovan, Neil and Nam.

I would also like to thank my friends. My friends from Greece, Spiros, George, Bill, Kostis, Teo and Venia. Although they were thousands of miles away, they were always supportive, and we spent many hours video chatting. Also, the friends that I made in Champaign and were pivotal towards my Ph.D. Erman was there for me from the first day I arrived at University of Illinois Urbana-Champaign (UIUC). I will always cherish the time we spent together and how much time he spent listening to me talk about computer architecture. Also, I want to thank Thymios, Thodoris, Antonio and Rovatsos that were always eager to go out or play a video game together, and take our minds of the Ph.D. for a while. I also like to thank Gohar that I met during a parallel programming course and became very close friends, Gizem, and the members of the Hellenic Association of Champaign-Urbana.

This journey would not have been so impressive without Olnancy. I would like to give her a very special thanks, for always being by my side. She was there through disappointments to cheer me up, and at accomplishments to celebrate. We studied, travelled, cooked, played and finished the Ph.D. together. I could not have asked for a better person to share this experience with.

Last but not least, I would like to thank my family, which I hope I had more time to see during these years. My sister Korina for her constant enjoyable support, for the times that she came to visit me here and the roadtrips we organized together. Also, my parents Nikolaos and Ioanna, for their unconditional love and support. They were always believing in me and they supported me when I decided to leave Greece and travel to the US to pursue my dreams.

TABLE OF CONTENTS

CHAPTER 1: THESIS OVERVIEW

1.1 INTRODUCTION

Byte-addressable Non-Volatile Memory (NVM) technologies such as 3D XPoint [1], Phase Change Memory (PCM) [2, 3, 4], STT-RAM [5] and Resistive RAM (ReRAM) [6] have recently gained much attention. These technologies offer high storage density, low static power, non-volatility, and performance characteristics that are comparable to those of DRAM [7]. Thanks to these properties, NVM is expected to create disruptive changes to many application domains, hardware and software systems.

NVM can serve multiple different purposes in future systems. Taking advantage of its increased capacity and low static power, it can be used as part of main memory or it can replace current last level caches (LLC). Considering NVM's non-volatile properties, it can be used as a fast storage-class memory device for writing persistent applications or providing data durability in distributed environments that have low-latency and fault-tolerance requirements. The success of NVM depends on the availability of architectures and user-friendly programming frameworks that gracefully facilitate NVM's integration to current systems, take advantage of its properties and ease software development. In this thesis, I propose new architectures that tackle the challenges of adopting NVM in future systems.

In single node systems, the increased capacity and durability characteristics of NVMs show promise to satisfy the high memory demands of contemporary workloads and develop applications that can benefit from NVM persistency. NVMs can attain high capacity at low cost [2, 8]. However, NVMs have higher read and write latencies than DRAM main memories and cannot replace DRAM without some performance loss. Thus, future systems will probably opt for hybrid main memory systems that are composed of both DRAM and NVM. A major challenge in such memory systems is deciding the placement of data in DRAM or NVM to attain high performance.

NVMs can also be used as an SRAM replacement for on chip LLCs. SRAM technology suffers from scalability problems. It has high area overhead and substantial leakage power [9]. NVM technologies such as STT-RAM [5] are promising alternatives to replace SRAM in LLCs. STT-RAM offers higher density and lower leakage power [10] than SRAM, but higher latency for both read and write operations, and higher dynamic energy consumption per access. Thus, it is challenging to replace SRAM LLCs without incurring performance loss because of the increased access latency of NVMs.

Another challenge of NVMs is how to effortlessly develop applications that specify data structures that use persistent storage. Most programming frameworks rely heavily on the programmer to mark persistent objects [11, 12, 13, 14, 15, 16, 17, 18]), identify stores that target NVM [11, 12, 16, 17, 18, 19, 20, 21] and potentially augment the code with instructions that write back a cache line to NVM (*CLWB*) [22], order instructions (store fence or *sfence*), and log state in NVM. A class of NVM programming frameworks that reduces the programmer's effort to identify persistent objects is *Persistence by Reachability* (e.g., [23, 24, 25]). Such frameworks require the programmer to annotate only durable roots within the program data structures and then rely on the runtime system to ensure that all objects that are reachable from a durable root are stored in NVM. However, the appeal of these frameworks is hindered by the increased runtime overheads that they experience.

In distributed systems, applications such as key-value stores and databases tend to utilize the faster volatile memories and offer fault tolerance by replicating data to the memories of other nodes of the system, instead of writing to persistent storage. This decision is justified by the high latency of persistent storage. NVMs offer a promising approach to help distributed applications attain both high performance and data persistence. However, distributed persistency has only received limited attention from the research community [26, 27, 28], especially for systems that are using emerging NVM technologies. Thus, it is crucial to understand the interaction of consistency and persistency models in distributed systems with NVMs, and the implications that these models have on distributed applications in terms of performance, fault-tolerance and programmability.

Finally, the network hardware infrastructure has been improving swiftly in distributed systems. At the same time, the capabilities of Network Interface Cards (NIC) have increased and SmartNICs have been developed with advanced hardware support [29, 30, 31]. In such environments, current implementations of transactional consistency, a very popular model in databases and key-value stores, suffer from high software overheads. However, clients demand high performance and fault-tolerance guarantees. As a result, there is a need to provide low-latency distributed transactions with durability guarantees.

1.2 THESIS CONTRIBUTION

This thesis presents new computing architectures which ease the adoption of NVMs in contemporary systems. My research addresses the challenges of utilizing NVMs both (a) in single-node multicore systems, as well as (b) in multi-node distributed systems. In single-node systems, I first provide a solution for data placement in hybrid DRAM-NVM main memory systems by proactively swapping data between the two memories. Next, I propose a technique that allows NVM to be used as an SRAM LLC replacement by hiding its high read latency. Also, I develop an architecture that assists programmable NVM frameworks by performing runtime operations in hardware instead of the slower software system. In distributed systems, I define *Distributed Data Persistency*, which explores how persistency models interact with consistency models. Also, I propose new hardware for the NIC and the processor that enables efficient distributed transactions, and supports the different distributed persistency models.

1.2.1 Summary of Contributions

The integration of NVM in future systems inherently involves trade-offs between several metrics such as performance, energy efficiency and programmability. Moreover, it is crucial to understand how this new type of memory can be used in smaller and larger scale systems. I present below a brief summary of the challenges that each of my proposals addresses and the novel techniques that I suggested to overcome them.

Managing Data Placement in Hybrid Memory Systems. Hybrid main memories composed of DRAM and Non-Volatile Memory (NVM) combine the capacity benefits of NVM with the low-latency properties of DRAM. To achieve high performance, data segments should be exchanged between the two types of memories dynamically—a process known as segment *swapping*—based on the access patterns to the segments in the program. A major difficulty in hardware-managed swapping techniques is to identify the appropriate segments to swap between the memories at the right time during program execution.

To perform hardware-managed segment swapping both accurately and with substantial lead time, I propose to use hints from the page walk in a TLB miss. I call this scheme *PageSeer* [32]. During the generation of the physical address for a page in a TLB miss, the memory controller is informed. The controller uses historic data on the accesses to that page and to a subsequently-referenced page (i.e., its follower page), to potentially initiate swaps for the page and for its follower. I call these actions *MMU-Triggered Prefetch Swaps*. PageSeer also initiates other types of page swaps, building a complete solution for hybrid memory. The evaluation of PageSeer with simulations of 26 workloads shows that PageSeer effectively hides the swap overhead and services many requests from the DRAM. Compared to a state-of-the-art hardware-only scheme for hybrid memory management, PageSeer on average improves performance by 19% and reduces the average main memory access time by 29%.

Tolerating Non-Volatile Cache Read Latency. The increased memory demands of workloads is putting high pressure on Last Level Caches (LLCs). Unfortunately, there is limited opportunity to increase the capacity of LLCs due to the area and power requirements of the underlying SRAM technology. Interestingly, emerging Non-Volatile Memory (NVM) technologies promise a feasible alternative to SRAM for LLCs due to their higher area density. However, NVMs have substantially higher read and write latencies, which offset their area density benefit. Although researchers have proposed methods to tolerate NVM's increased write latency, little emphasis has been placed on reducing the critical NVM read latency.

To address this problem, I propose *Cloak* [33, 34]. Cloak exploits data reuse in the LLC at the page level, to hide NVM read latency. Specifically, on certain L1 DTLB misses to a page, Cloak transfers LLC-resident data belonging to the page from the LLC NVM array to a set of small SRAM Page Buffers that will service subsequent requests to this page. Further, to enable the high-bandwidth, low-latency transfer of lines of a page to the page buffers, Cloak uses an LLC layout that accelerates the discovery of LLC-resident cache lines from the page. I evaluate Cloak with full-system simulations of a 4-core processor across 14 workloads. I find that, on average, Cloak outperforms an SRAM LLC by 23.8% and an NVM-only LLC by 8.9%—in both cases, with negligible additional area. Further, Cloak's ED^2 is 39.9% and 17.5% lower, respectively, than these designs.

Architectural Support for Programmable Non-Volatile Memory Frameworks. The availability of user-friendly programming frameworks is key to the success of NVM. Unfortunately, most current NVM frameworks rely heavily on user intervention to mark persistent objects and even persistent writes. This not only complicates NVM programming, but also introduces potential bugs. To address these issues, researchers have proposed Persistence by Reachability frameworks, which require minimal user intervention. However, these frameworks are slow because their runtimes have to perform checks at program load and store operations, and move data structures between DRAM and NVM during program execution.

In this thesis, I introduce P-INSPECT [35], a novel hardware architecture targeted to speeding up persistence by reachability NVM programming frameworks. P-INSPECT uses bloom-filter hardware to perform various checks in a transparent and efficient manner. It also provides hardware for low-overhead persistent writes.

My simulation-based evaluation running a state-of-the-art persistence by reachability framework shows that P-INSPECT retains programmability and eliminates most of the overhead. I use real-world applications to demonstrate that, on average, P-INSPECT reduces an application's number of executed instructions by 26% and the execution time by 16%—delivering similar performance to an ideal runtime that has no persistence by reachability overhead.

Distributed Data Persistency. Distributed applications such as key-value stores and databases avoid frequent writes to secondary storage devices to minimize performance degradation. They provide fault tolerance by replicating variables in the memories of different nodes, and using data consistency protocols to ensure consistency across replicas. Unfortunately, the reduced data durability guarantees provided can cause data loss or slow data recovery. In this environment, NVM offers the ability to attain both high performance and data durability in distributed applications. However, it is unclear how to tie NVM memory persistency models to the existing data consistency frameworks, and what are the durability guarantees that the combination will offer to distributed applications.

In my work, I propose the concept of *Distributed Data Persistency* (DDP) model [36], which is the binding of the memory persistency model with the data consistency model in a distributed system. I reason about the interaction between consistency and persistency by using the concepts of *Visibility Point* and *Durability Point*. I design low-latency distributed protocols for DDP models that combine five consistency models with five persistency models. For the resulting DDP models, I investigate the trade-offs between performance, durability, and intuition provided to the programmer.

Hardware-Assisted Distributed Transactional Consistency. Transactional-based distributed storage applications such as key-value stores and databases are widely used in the cloud. Recently, the hardware infrastructure on which these applications run has been rapidly improving, with faster networks and powerful network interface cards (NICs). Unfortunately, as a result of these changes, these applications run increasingly inefficiently. They have bulky housekeeping software overheads that constrain performance.

To address this problem, I analyze the sources of software overhead in distributed transactional consistency applications and propose new hardware structures to eliminate them. The hardware includes bloom filters for a variety of tasks, and SmartNICs for efficient remote communication. I then develop *HADES*, a new distributed transactional protocol that leverages this hardware to provide low-overhead distributed transactions, and also propose a hybrid hardware-software implementation of HADES. Finally, I present how HADES interacts with different distributed data persistency models and how it enables support for in-network persist operations. The evaluation of HADES shows that a set of workloads running on five nodes of five cores each, execute on average 2.7× faster on HADES than on a state-of-the-art distributed transactional system. Moreover, we find that the persistency models can change the performance of our applications up to 77%, while supporting in-network persistency in the NIC can alleviate the overhead of data persistency, especially for strict persistency models.

1.3 THESIS ORGANIZATION

This thesis is organized as follows. Chapter 2 describes using NVM in a hybrid DRAM-NVM main memory system and presents PageSeer. Chapter 3 presents using NVM as an LLC replacement and introduces Cloak. Chapter 4 presents P-INSPECT to enable efficient NVM programming frameworks. Next, Chapter 5 presents the Distributed Data Persistency models, and Chapter 6 describes HADES for supporting efficient distributed transactional consistency. Finally, Chapter 7 concludes by summarizing the contributions of my thesis.

CHAPTER 2: USING PAGE WALKS TO TRIGGER PAGE SWAPS IN HYBRID MEMORY SYSTEMS

2.1 INTRODUCTION

Data-intensive applications demand large memory capacity and high bandwidth with low power consumption. While DRAM has been used as main memory for decades due to its relatively low latency and energy consumption, it is suffering from device scalability problems [37]. As a result, new memory solutions are required.

Non-Volatile Memory (NVM) technologies, such as PCM [2] and STT-RAM [38], show promise to satisfy the increasing memory capacity demands of workloads. These memories can attain high capacity at low cost [2, 8]. Memory vendors have announced the production of NVMs [39], and impending systems will include them.

Unfortunately, NVMs have drawbacks and cannot simply replace DRAM in their current form. Compared to DRAM, read and write accesses in NVMs have higher latencies and consume more energy. As a result, upcoming systems are likely to incorporate a hybrid main memory system composed of both DRAM and NVM.

The main challenge in a hybrid memory system is how to exploit the capacity benefits of NVM, while benefiting from the low latency of DRAM. Previous research has either used DRAM as a cache for the NVM [40] or used a flat address space configuration with both memories [41]. The latter organization provides both higher aggregate bandwidth and higher memory capacity. However, one needs to decide in what memory to place specific data structures.

Optimal placement of data structures in a hybrid memory system is hard to attain statically, as applications exhibit dynamic changes of behavior. It is better to dynamically move data segments between the two types of memory (i.e., to *swap* segments between the memories) based on the access patterns. Such a movement can be software- or hardware-managed.

Hardware-managed swap techniques [41, 42, 43] are generally preferred over software-managed ones because they induce much lower overhead. However, they require special hardware structures to track memory access activity, perform the data swaps, and record the remappings of segments between the memories. The efficient management of the relevant metadata is crucial for performance.

The key difficulty in hardware-managed swap techniques is to identify the appropriate segments to swap at the appropriate time in the execution. Aggressive schemes that move a segment to DRAM upon the first access to it introduce unnecessary traffic if the segment is not accessed much more after the swap. Alternatively, schemes that require a history of many accesses to the segment before moving it to DRAM may react too slowly and hence not improve performance.

To perform hardware-managed segment swapping both accurately and with substantial lead time, this chapter proposes to use hints from the page walk in a TLB miss. We call the scheme *PageSeer*. During the generation of the physical address for a page in a TLB miss, the memory controller is informed. The controller uses historic data on the accesses to that page and to a subsequently-referenced page (i.e., its *follower* page), to potentially initiate swaps for the page and for its follower. We call these actions *MMU-Triggered Prefetch Swaps*. They are transparent to the software. PageSeer also initiates other types of page swaps, building a complete solution for hybrid memory.

We evaluate PageSeer using simulations of 26 workloads. Our results show that PageSeer effectively hides the swap overhead, and services many requests from the fast memory. Compared to a state-of-the-art hardware-only scheme for hybrid memory management, PageSeer on average improves performance by 19% and reduces the average main memory access time by 29%. Further, MMU-triggered prefetch swaps accurately predict future memory accesses. Overall, PageSeer efficiently manages a hybrid memory system.

2.2 BACKGROUND & MOTIVATION

2.2.1 Hybrid Memories

Emerging NVM technologies, such as 3D XPoint [39] and Phase Change Memory [2], have recently gained a lot of attention. The high bit density, low static power, and non-volatile aspects of these memories appear as a viable solution to the increasing memory demands of workloads and the slowdown of DRAM scaling. However, NVMs exhibit higher latencies than DRAM, and therefore cannot replace DRAM entirely without performance loss. For this reason, the combination of DRAM and NVM has been proposed as a method to efficiently increase system capacity, performance, and reliability [40]. A memory system that integrates both DRAM and NVM is typically called a hybrid memory system.

A hybrid memory system can be configured in one of two ways. In one configuration, the faster and smaller DRAM is a hardware-managed cache for the slower and larger NVM [44, 45, 46, 47, 48, 49]. In the other, the DRAM and NVM are configured as a flat address space, where the OS is aware of both memories for page allocation [41, 42, 43, 50, 51, 52]. The first configuration has the advantage that it can be easily deployed and is transparent to the OS, with DRAM acting as an additional level of caching between the Last Level Cache (LLC) and main memory. However, it faces the challenge of efficiently storing and accessing a large amount of tags [45, 53]. Moreover, the overall capacity of the system decreases by a non-negligible amount, as long as the sizes of DRAM and NVM are comparable. Although previous work has shown performance improvements for latency critical applications [44, 47], capacity-limited applications do not benefit as much [50]. Also, the overall memory bandwidth is limited, since we cannot take advantage of the combined bandwidth of the two memories.

The flat address space configuration has the advantage that it provides both higher aggregate bandwidth and higher memory capacity. Moreover, there is no need for tag storage. However, it is challenging to decide the data placement and swapping of data between the two memories.

Data swaps between the two memories can be done either in software [52, 54, 55] or in hardware [41, 42, 43, 50, 51]. In both cases, we must identify data that are "hot" (i.e., accessed frequently) but reside in the slow memory, and swap them with data that are "cold" but reside in the fast memory. In a software-managed approach, the OS interrupts the processor, swaps the pages, performs a TLB shootdown to purge stale TLB entries, and continues execution. This procedure can take several microseconds [56], and constrains swaps to a coarse time granularity.

When data swaps are hardware-managed, they can happen at finer time granularity. However, there are several challenges in this method. The first one concerns the consistency between the OS view of memory and the data movements that the hardware has performed. Since the OS is not aware of any remapping, the hardware needs to keep track of the data remappings that have occurred. Second, we need dedicated hardware to decide when and what swaps to perform between fast and slow memory. This means that the hardware should be able to track memory activity, and trigger a swap accurately and promptly to tolerate the swap cost.

2.2.2 Hardware-Based Memory Management Techniques

Previous work has investigated hardware-only techniques for managing hybrid memory systems. Typically, these techniques rely on LLC misses to track main memory activity and determine data swaps between DRAM and NVM. The techniques differ in the size of the memory segments to swap and what triggers a swap.

Suppose that we perform a segment swap between the slow and the fast memory. Then, if a second segment from the slow memory needs to be moved to the exact same location in fast memory as the first one, it can do so with a *Slow* or a *Fast* swap. In a slow swap, the first swap is undone and then the second swap is performed. In a fast swap, only one swap is performed, exchanging the second segment from the slow memory with the segment currently in fast memory (which used to be in slow memory).

CAMEO [50] migrates data in 64B blocks, and a swap is triggered on every access to a block in slow memory. CAMEO restricts the swap flexibility by allowing a set of slow-memory blocks (which form a Swap Group) to be swapped only with a single block of fast memory. Also, only one of these slow-memory blocks can be in fast memory at a time. Further, a slow-memory block can reside anywhere within the slow-memory area assigned to its swap group. CAMEO uses fast swaps. While CAMEO keeps the required swap bandwidth low and is easy to implement, the small swap granularity requires substantial meta-data storage and misses the opportunity to take advantage of spatial locality. Moreover, the direct mapping of the swap groups to single fast-memory blocks may cause conflict misses.

PoM [41, 57] is similar to CAMEO, with the difference that swaps happen at the granularity of 2KB, and a swap is triggered when the number of accesses to a 2KB memory segment reaches a threshold. PoM is adaptive, and the swap threshold can change based on the program characteristics. PoM uses fast swaps and has direct-mapped swap groups.

SILC-FM [43] uses segments (e.g., 2KB), and optimizes the granularity of swaps, which can range from 64B sub-blocks to the whole segment. It supports sub-block interleaving, where two segments interleave data at sub-block granularity. SILC-FM also relaxes swap groups to be set-associative rather than direct-mapped. It uses slow swaps.

MemPod [42] further enhances swap flexibility by allowing any slow-memory segment to be swapped with any fast-memory segment within a Pod. This comes at the cost of a substantial increase in the metadata overhead. MemPod uses the Majority Element Algorithm [58] to identify memory segments that are to be accessed in the future, and migrates them to fast memory at 2KB granularity after predefined time intervals.

Other hardware schemes target different aspects of hybrid memory systems. For example, BATMAN [59] tries to optimize swaps so that the overall memory bandwidth utilization is maximized. ProFess [51] proposes a cost-benefit mechanism that decides swaps considering fairness between different programs that compete for fast memory.

One difficulty in this area is the need to make swap decisions early enough. Otherwise, it is likely that a swap for a memory segment will not be finished by the time memory requests for the segment arrive.

2.2.3 Page Walk

In current x86 systems, only 48 bits out of the 64-bit virtual address (VA) are used for addressing. Of those, the lower 12 bits are used for the offset within the 4KB page. When a virtual to physical page translation is not found in the TLB, the hardware initiates a page table walk. A page table walk consists of the hardware stepping over four levels of page tables (Figure 2.1): the Page Global Directory (PGD), the Page Upper Directory (PUD), the Page Middle Directory (PMD) and, finally, the Page Table Entry (PTE). The base of the PGD is obtained by using the CR3 register, which is unique to a process. Adding CR3 to bits 47-39 of the VA, we obtain a PGD entry, whose contents is the base of the PUD table. Adding this base to bits 38-30 of the VA, we obtain a PUD entry, whose contents is the base of the PMD table. This process repeats until we obtain the base of the requested physical page, which is finally added to the page offset.

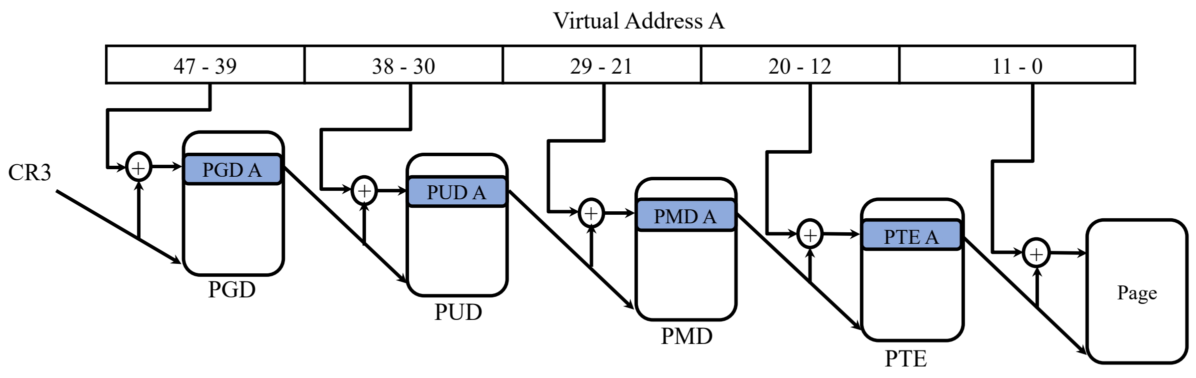


Figure 2.1: Page walk operation.

This process may require up to four memory accesses, to get the entries in the PGD, PUD, PMD, and PTE tables. To avoid main memory accesses, the data in these entries can be stored in the caches (except in L1), along with regular data. In addition, to further reduce the cost, modern processors have an intermediate translation cache called the Page Walk Cache (PWC), which stores a few entries per translation level (except for the PTE). The PWC is accessed before going to the L2 cache to obtain the entries. The four-step page walk and the PWC are in the core's Memory Management Unit (MMU).

2.3 DESIGN OF PAGESEER

2.3.1 Main Idea

PageSeer is a hardware mechanism that initiates early page swaps — also called *Prefetch Swaps* — between slow and fast memory. Prefetch swaps are initiated before the main memory receives multiple requests for the page in slow memory to be moved to fast memory. To initiate prefetch swaps, PageSeer uses state stored in a hardware table called Page Correlation Table (PCT). Prefetch swaps can be triggered by one of two events: (i) a hint from the MMU (*MMU-triggered Prefetch Swaps*), or (ii) a regular memory access (*Prefetching-triggered Prefetch Swaps*).

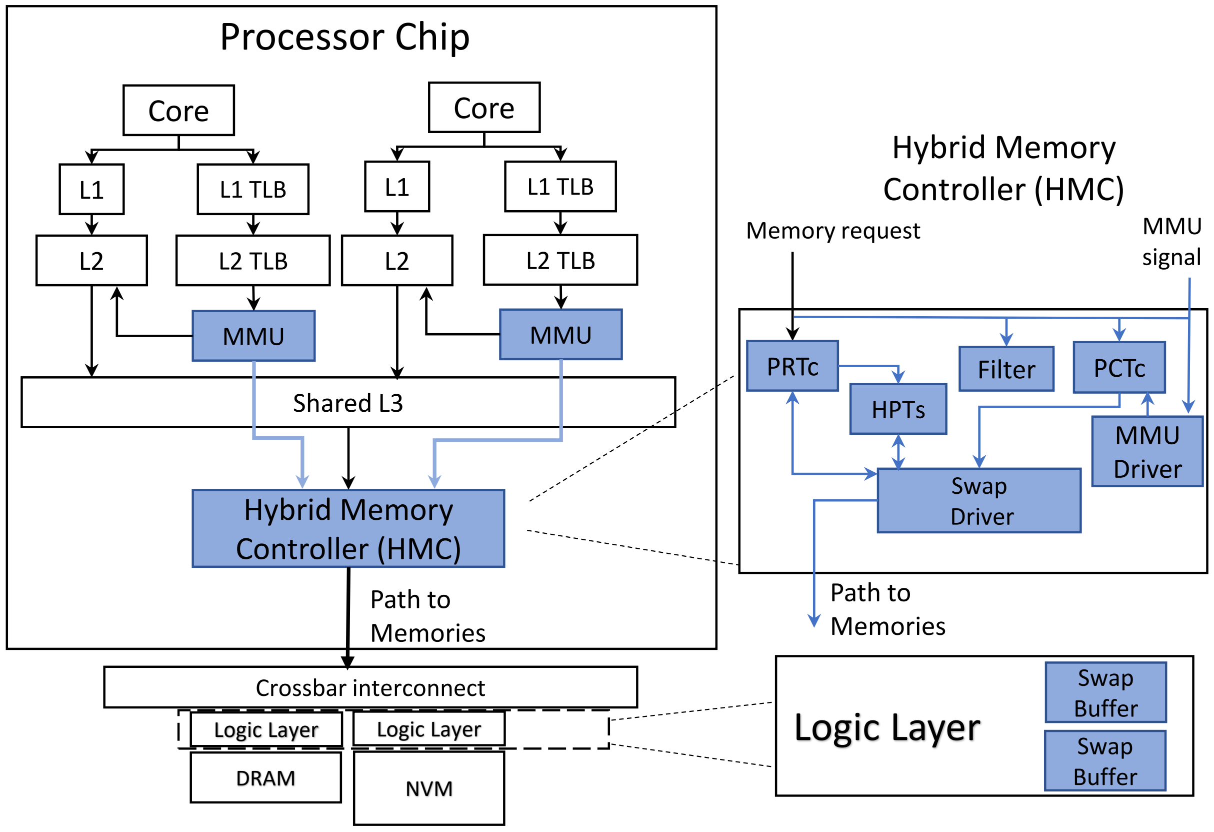


Figure 2.2: PageSeer architecture, where the new or modified hardware structures are shown shaded, and the added connections are shown in lighter color.

PageSeer also supports regular swaps, which are initiated when the main memory receives a certain number of requests for a page. To initiate regular swaps, PageSeer uses state stored in a hardware table called Hot Page Table (HPT).

Most of PageSeer's hardware structures are placed in the memory controller, which we call *Hybrid Memory Controller* (HMC). In addition, PageSeer needs swap buffers in the DRAM and NVM memory modules, and slightly modifies the MMUs. Figure 2.2 shows the architecture of PageSeer, where the new or modified hardware structures are shown shaded, and the added connections are shown in lighter color. In the following, we first describe the communication between MMU and HMC (Section 2.3.2), then the structures in the HMC (Section 2.3.3), and then the operation of PageSeer (Section 2.3.4).

2.3.2 MMU-Triggered Prefetch Swaps

Memory-intensive applications that access many pages are likely to miss in the TLB. Further, after a page walk, applications with large working sets are unlikely to find the requested PTE entry in the caches, and are likely to have to go to main memory. Under such conditions, PageSeer uses the time that it takes to satisfy a TLB miss to potentially perform a prefetch swap of the requested page and one additional page — bringing them to fast memory in expectation that they will be referenced very soon. We call these actions *MMU-Triggered Prefetch Swaps*.

This operation requires some hardware modifications. In conventional systems, as soon as a page walk reaches the fourth translation level and the address of the memory line with the needed PTE entry is known, the MMU sends a request to the L2 cache. In PageSeer, the MMU additionally sends a signal to the MMU Driver in the HMC. This is shown as action ① in Figure 2.3.

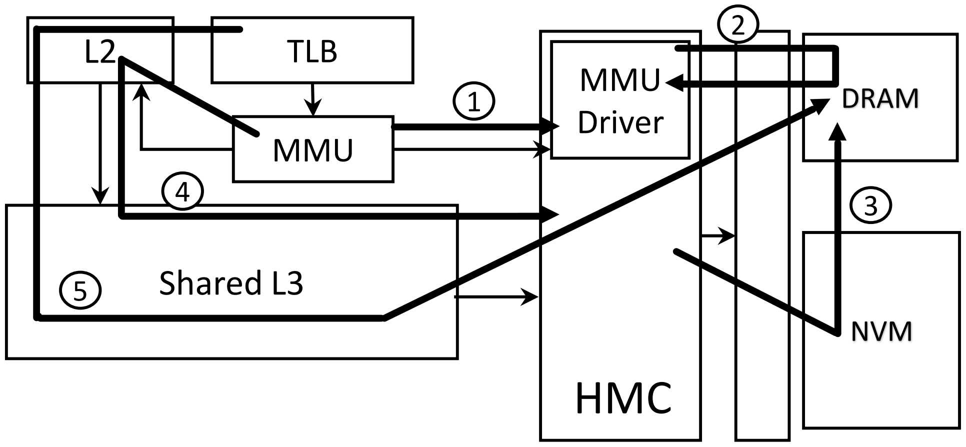


Figure 2.3: Performing MMU-triggered prefetch swaps in PageSeer.

When the MMU Driver in the HMC receives the signal, it sends a memory request to DRAM to obtain the PTE entry (action ② in Figure 2.3). When the HMC obtains the PTE entry, it extracts the Physical Page Number (PPN) from it. After that, based on the state of the PCT and other internal state, it decides whether to perform a prefetch swap of the requested page and one additional page (action ③ in Figure 2.3). In any case, the memory line with the needed PTE entry is cached in the MMU Driver of the HMC. Further, some internal state in the HMC is updated.

This design has two benefits. The first one occurs if, later, the request from the MMU to the L2 cache to obtain the PTE entry ends-up missing in both the L2 and L3. At that point, conventional systems send the request to the memory controller, which should initiate a main memory access (action ④ in Figure 2.3). However, since the MMU Driver in the PageSeer HMC does cache the line with the PTE entry (or, at least, it has already issued a request for it), the HMC provides the data faster. Note that the HMC needs to know that this is a request for a line with a PTE entry. To make this possible, PageSeer adds an identifying bit in the message that the MMU sends to the L2.

The second, more important benefit occurs when the TLB in updated with the new translation and the original memory request is replayed. If the request misses in the caches and is directed to a page that was in NVM, the prefetch swap triggered by PageSeer may have brought the page to DRAM. The result is a faster memory access for this request and potentially future ones (action ⑤ in Figure 2.3).

2.3.3 Structures in the Hybrid Memory Controller

In addition to the MMU signals described in the previous section, the HMC handles all the memory requests. The HMC includes some hardware structures that swap pages between the slow and fast memories, keep track of address re-mappings, monitor memory activity, and trigger swaps. We describe them in this section.

Page Re-mapping: PageSeer swaps pages without OS knowledge. As a result, the hardware needs to examine every request that reaches the HMC to determine if the location of the page has changed as a result of a swap. PageSeer accomplishes this with the *Page Remapping Table* (PRT). This hardware table is responsible for keeping information on all the current page re-mappings. The OS is oblivious to the re-mappings.

The information needed to keep track of all the current re-mappings is substantial. Moreover, the PRT is accessed on every main memory access and is on the critical path. Hence, we need to keep the access time to a minimum. Consequently, rather than having the whole PRT in the HMC, PageSeer saves storage and latency by keeping a cache of the PRT in the HMC, which holds only some of the PRT entries. We call it *PRTc*, for PRT cache. The rest of the entries are stored in DRAM, like in other designs [41, 42, 43, 51].

We design the PRT and the PRTc so that they can be accessed quickly and use the storage efficiently, minimizing the amount of metadata they need to hold. Specifically, we constrain the pairs of DRAM and NVM pages that can be swapped with each other. As shown in Figure 2.4, only DRAM and NVM pages of the same cache color can be swapped with each other. This means that an NVM page can only be swapped with DRAM pages that map to the same PRTc set.

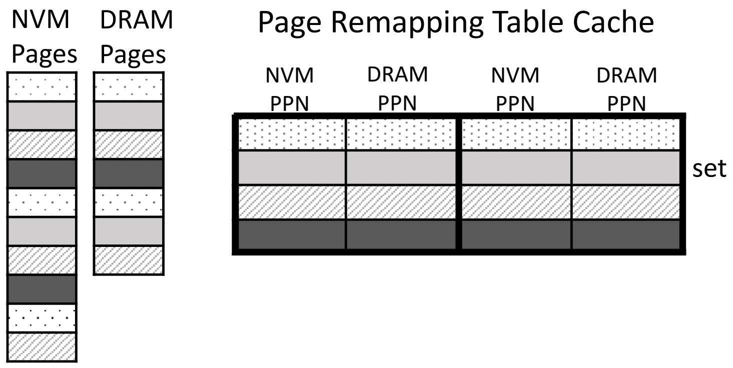


Figure 2.4: Page Remapping Table Cache (PRTc).

As shown in Figure 2.4, the PRTc is set-associative, and each entry has an NVM PPN and a DRAM PPN. The entry denotes that these two pages have been swapped — i.e., the NVM data is in DRAM, using the original location of the DRAM PPN, and vice-versa. With this design, PRTc queries are fast. When a request for a memory address arrives at the PRTc, the hardware extracts the address' PPN. Irrespective of whether this PPN is in the physical memory range of NVM or in the physical memory range of DRAM, the same PRTc set is accessed and the multiple entries are read out. Then, if this PPN is in the NVM range, the PPN is compared to the leftmost field in each of the selected entries; if this PPN is in the DRAM range, it is compared to the rightmost field in the entries.

This design requires that pages that are not currently swapped remain in their originally-assigned location. For example, an NVM page that is swapped to DRAM and then returns to NVM has to return to its original position. The same is true for a DRAM page. This design is very space efficient, as it requires minimal metadata. However, it cannot support fast swaps.

To reduce the cost of swaps, PageSeer uses what we call *Optimized* slow swaps. The idea is to reduce the number of read and write operations by temporarily keeping one of the pages in a swap buffer. Figure 2.5 shows an optimized slow swap. The figure considers three pages: DRAM page ① and NVM pages ② and ③. In the past, pages ① and ② have been swapped. As a result, the state of the memory is represented by the *dark* circles in the left figure labeled *Step 1*.

Suppose now that that we need to move page ③ to DRAM and, because of the state of the replacement algorithm, it has to go to the place currently occupied by page ②. A fast swap would simply swap pages ③ and ②, for a total of 2 page reads and 2 page writes. However, it would not bring ② to its original place in NVM (which is currently used by page ①). A

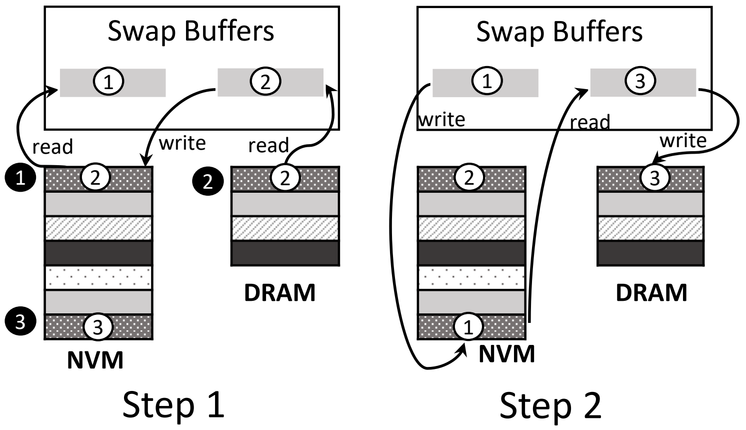


Figure 2.5: Optimized slow swap operation.

slow swap, instead, would swap ① and ②, and then ③ and ①, for a total of 4 page reads and 4 page writes.

An Optimized slow swap leverages the swap buffers to only perform 3 reads and 3 writes. This is shown by the white circles. In *Step 1*, pages ① and ② are read into the swap buffers, and page ② is written to its original NVM location. Then, in *Step 2*, page ③ is read into a free swap buffer, and pages ① and ③ are written to NVM and DRAM, respectively.

Initiating Prefetching-triggered Prefetch Swaps: Besides the MMU hints, the other trigger of PageSeer actions is LLC misses. PageSeer has two hardware structures that track LLC miss information and initiate swaps. They are the *Page Correlation Table* (PCT), which initiates prefetching-triggered Prefetch Swaps (in addition to assisting in MMU-triggered Prefetch Swaps), and the *Hot Page Tables* (HPTs), which initiate Regular Swaps. In this section, we describe the PCT; in the next one, we describe the HPTs.

When a page *P1* is accessed, the main memory system often observes a flurry of LLC misses on *P1* in a short period of time, followed by a flurry of misses on another page *P2*, and so on. Further, later, when *P1* is accessed again, *P1* is often seen to cause a similar flurry of misses, again followed by a flurry of misses by follower *P2*. For a page like *P1*, a PCT entry saves the number of LLC misses observed when *P1* is accessed, the PPN of its follower page *P2*, and the number of misses observed on *P2*. Later, when *P1* is accessed and triggers its first miss, if its PCT entry's miss count is higher than a threshold, and *P1* is in NVM, PageSeer issues a prefetching-triggered prefetch swap for *P1*. Further, if *P1*'s follower *P2* has a miss count higher than the threshold, and *P2* is in NVM, PageSeer also issues a prefetching-triggered prefetch swap for *P2*. With these early swaps, PageSeer can avoid repeated, costly NVM accesses. The miss count threshold is set so that the cost of a swap is lower than the expected savings to be attained.

In practice, the PCT is too large to keep in the HMC. Consequently, the HMC keeps a *PCT cache* (PCTc). A PCTc entry contains the PPN of a leader page, the number of LLC misses on the page per invocation, the PPN of the follower page, and the number of LLC misses on the follower per invocation. This is shown in the top part of Figure 2.6.

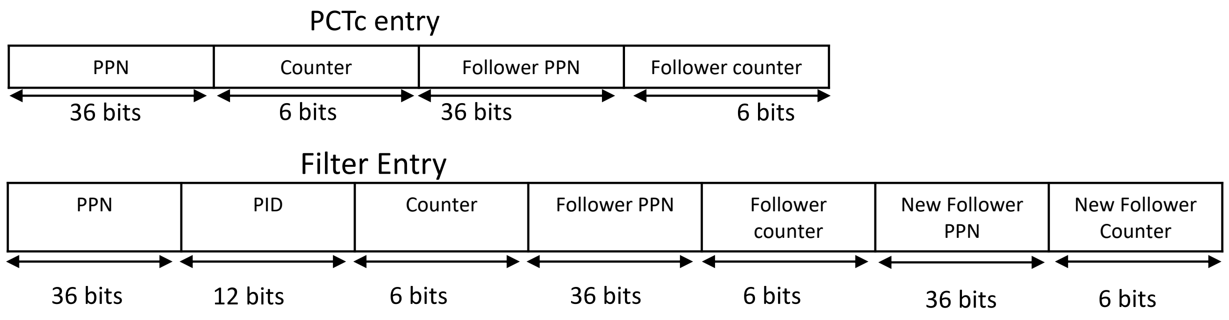


Figure 2.6: Structure of a PCTc and a Filter entry.

Since the miss patterns of a page change with time, PageSeer also uses a small hardware *Filter* table. Its purpose is to quickly update the information of a page's PCTc entry. The Filter table only has a few entries and works as follows. When the HMC observes an LLC miss for a page, it brings the page's PCTc entry (if it exists) into the Filter table. As execution proceeds, PageSeer recomputes a new miss count for the page by adding the number of misses observed in the current invocation plus half the value of the old miss count. This is done for both leader and follower pages. This approach is taken to reflect the new miss patterns, while retaining some history from past invocations. The new miss counts are stored back in the corresponding PCTc entry.

The structure of a Filter entry is shown in the lower part of Figure 2.6. In addition to the leader page's PPN and counter, and the follower page's PPN and counter, it has three more fields. One is the program identifier (PID) of the process accessing the leader and follower pages. This is needed so that, in a multi-program environment, PageSeer does not try to correlate pages that are accessed by different programs; we want to correlate leader-follower pages accessed by the same program.

The two additional fields are the PPN and counter for a new follower page. It is possible that, in this new use of leader page *P1*, *P1* is not followed by the use of page *P2*, but by the use of page *P3*. For this reason, the Filter entry has space to record the accesses to a new follower (*P3*). Later, when the Filter entry is to be saved into the PCTc, only the follower with the highest count is saved.

Each PCTc entry has one additional bit that indicates whether the entry's contents have effectively changed since it was last brought in from the PCT in main memory. An effective change is one that causes a different swap action for any of the pages involved. When a PCTc entry is evicted, it is written back to the PCT only if the change bit is set.

Finally, the PCTc state is also used for the MMU-triggered prefetch swaps of Section 2.3.2. The difference is that the trigger that causes the PCTc look-up and potentially the two swaps is not an access to a page. Instead, it is a signal from the MMU Driver that was initiated by a TLB miss.

Initiating Regular Swaps: The *Hot Page Tables* (HPTs) are two small hardware tables, one for DRAM pages and the other for NVM pages, that record the pages that are being frequently accessed (i.e., are "hot"). Each HPT entry has a PPN and a counter of how many LLC misses have been recorded on this PPN. Every time that the HMC receives an LLC miss for a page, the corresponding counter is incremented. The counters are automatically halved at regular intervals. If the counter in an entry reaches zero, the corresponding page is removed from the HPT.

The goal of the DRAM HPT is to lock hot pages in DRAM. A DRAM page that appears in the HPT is being accessed a lot and, therefore, should not be swapped out of DRAM. The goal of the NVM HPT is to identify NVM pages that are becoming hot and should be swapped to DRAM. When the count in an entry of the NVM HPT reaches a swap threshold, the hardware starts a regular swap operation for the corresponding NVM page. Note that the NVM HPT complements the PCTc. The latter may fail to initiate a swap for the page, either because the PCTc does not yet have an entry for the page, or because the current count value is too low to initiate a prefetching-triggered prefetch swap. The NVM HPT has a lower count threshold to initiate a swap than the PCTc. Both the HPTs and the PCTc are off the critical path.

Other Structures: The two other HMC structures in Figure 2.2 are the *MMU Driver* and the *Swap Driver*. The former gets a signal from the MMU on a page walk. It then obtains the memory line with the relevant PTE entry — either from memory or from the small set of lines with PTEs that it caches. After that, it generates the page PPN and checks the PCTc to determine if an MMU-Triggered Prefetch Swap needs to be started for the page. The MMU Driver also intercepts LLC misses requesting lines with PTE entries.

The Swap Driver initiates all page swaps. It receives requests from either the PCTc or the HPT. It also checks all memory accesses, to ensure that those directed to pages being swapped get the data from the swap buffers.

2.3.4 Putting All Together: PageSeer Operation

After having described the HMC structures, we can explain PageSeer's operation. We divide it into three flows: (i) a regular memory request reaches the HMC, (ii) an MMU signal or an LLC miss requesting a PTE entry reaches the HMC, and (iii) a regular memory request reaches the HMC while a swap is in progress.

A Regular Memory Request Reaches the HMC: The PRTc is accessed to find out the correct address in case the page is remapped. In parallel, the PCTc and Filter receive the request. Note that PCTc and Filter use addresses before remapping, to be able to retain their state across remappings. If the PRTc or PCTc miss, memory requests are sent to DRAM to fetch the appropriate entries.

Immediately after the PRTc look-up, we have the correct address, and the request is sent to main memory (with a Swap Driver look-up). In parallel, the request is sent to the DRAM and NVM HPTs. After the Swap Driver receives signals from the HPTs and the PCTc, it knows whether the NVM HPT or the PCTc request a swap (for this page and/or its successor), and which pages cannot be swapped out of DRAM (from the DRAM HPT). If appropriate, the Swap Driver initiates page swap(s).

In a swap operation, as the hardware reads a page into a swap buffer, it starts with the requested cache line first. It also provides the requested line right away to the processor.

The Swap Driver may refuse to perform a swap if, due to a large number of requests directed to the DRAM, the DRAM bandwidth is saturated and the NVM bandwidth is under-utilized. Performance is usually higher if saturation of the DRAM links is avoided.

An MMU Signal or an LLC Miss Requesting a PTE Entry Reaches the HMC: As indicated in Section 2.3.3, the MMU Driver intercepts these two types of requests. On an MMU signal, the MMU Driver obtains the PTE, then fetches the needed PRTc and PCTc entries (if missing) and, finally, it checks the PCTc to determine whether an MMU-Triggered Prefetch Swap needs to be initiated. On reception of an LLC miss requesting a line with a PTE entry, the MMU Driver provides it from its cache.

A Regular Memory Request Reaches the HMC while a Swap Is in Progress: The Swap Driver checks whether the request targets a page that is participating in the swap. If it does not, the request proceeds normally. Otherwise, the request obtains the data from the appropriate swap buffer. This helps avoid stalls for requests directed to these hot pages. The swap buffers temporarily act as prefetch buffers for these pages.

2.3.5 Page Swaps between Memory and Disk

PageSeer is compatible with DMA engines that swap pages between memory and disk. All DMA requests go through the HMC, which may change the address if the page has been remapped. As soon as the HMC receives the first DMA request to read/write a line, the HMC completes any swap in progress for that page, then freezes the page (preventing future swaps), and then allows the DMA requests for that page to proceed. After the DMA is done, the page is unfrozen. There is no need to change the state of the page in the HMC structures; the state will dynamically evolve based on the miss patterns of the new page.

2.4 EXPERIMENTAL METHODOLOGY

2.4.1 Evaluation Infrastructure

We use cycle-level simulations to model a server architecture with a 4-core multicore and a 4.5-GB main memory composed of 4 GBs of NVM and 512 MBs of DRAM. The architecture parameters are shown in Table 2.1. Each core is an out-of-order core with private L1 and L2 caches, and a shared L3 cache. It has private L1 and L2 TLBs and page walk caches for intermediate translations. Each core has a page walker. We integrate the Simics full-system simulator [60] with the SST [61, 62] framework, and the DRAMSim2 [63] memory simulator, similar to [64]. To model NVM, we modified the DRAMSim2 timing parameters as shown in Table 2.1, and disabled refreshes. We use CACTI [65] for energy and area analysis of the PageSeer structures. Additionally, we utilize Intel SAE [66] on top of Simics for OS instrumentation. The page walk is modeled after the x86 architecture, and leverages the 4-level page tables created and maintained by the OS to perform the page walk memory accesses. We accurately model the page swaps and the accesses to the HMC structures by issuing the appropriate read and write requests to memory. Our implementation is based on the Ubuntu 16.04 operating system.

2.4.2 Configurations

We compare our design to two state-of-the-art hardware-managed hybrid memory systems: PoM [41] and MemPod [42].

Processor/MMU Parameters	
Cores; Frequency	4 out-of-order cores; 2GHz
Cache line	64B
L1 cache	32KB, 8-way, 2 cycles access latency (AL)
L2 cache	256KB, 8-way, 8 cycles AL
L3 cache	8MB, 16-way, 32 cycles AL, shared
L1 TLB	64 entries, 4-way, 1 cycle AL
L2 TLB	1024 entries, 12-way, 10 cycles AL
Main-Memory Parameters	
Capacity	DRAM: 512MB; NVM: 4GB
Channels	DRAM 4; NVM: 2
t_{CAS}-t_{RCD}-t_{RAS}	DRAM: 11-11-28; NVM: 11-58-80
t_{RP},t_{WR}	DRAM: 11,12 NVM: 11,180
Ranks per Channel	DRAM: 1; NVM: 2
Banks per Rank	DRAM: 8; NVM: 8
Frequency; Data rate	1GHz; DDR
Bus width	64bits per channel
Operating System: Ubuntu Server 16.04	

Table 2.1: Configuration of the system evaluated.

PoM: We configure PoM according to the specification given in previous work [41], but we change the architecture-related parameters to adjust it better to our configuration. In the PoM paper, the authors manage die-stacked and DRAM memories with different latencies than ours. Thus, we modify their `K` parameter to 12 to be consistent with our memory timing model. For the `SRC`, which is the equivalent of our PRTc, we use a 32KB cache similar to PageSeer.

MemPod: MemPod uses the MEA algorithm to decide on memory swaps. Both PoM and MemPod swap at the granularity of 2KB. For MemPod, we use 64 MEA counters and make swap decisions every 50 μs, as described in the original work [42]. We also use a 32KB cache for the remapping table. MemPod also requires an inverted map table, but since we lack details about its implementation, and to be optimistic in our evaluation, we assume a zero cycle latency for this structure.

PageSeer: The parameters of our design are shown in Table 2.2. The goal is to keep the size of the entries in the PRTc and PCTc tables small. As a result, we can have more entries in the tables and increase their hit rate. The total size of the HMC structures is less than 72KB, which is very modest. We also need the full PRT and PCT tables in the DRAM, but they account for only 1% of our DRAM storage. In our experiments, we found that caching 16 lines with PTE entries in the MMU Driver is good enough. Doing so gives us a hit rate of over 99% for page walk requests that miss in the LLC and reach the MMU driver.

PageSeer Design Parameters	
Swap size	4KB (which is a page)
Counters	6 bits
MMU to HMC latency	2 cycles (at 2GHz)
PCTc prefetch swap threshold	14
HPT swap threshold	6
HPT counter decrease interval	50K cycles (at 1GHz)
PRT	4 way-set associative
PageSeer Hardware Structures	
PRTc and PCTc	32KB, 4-way, 1 cycle (at 1GHz)
HPT size (each table)	5.3KB,fully-assoc,4 cycle (at 1GHz)
Filter	2.2KB,fully-assoc,2 cycle (at 1GHz)
MMU Driver	16 lines with PTEs, 64B per line
PRTc,PCTc,HPT,Filter entry	3.5B, 10.5B, 5.25B, 17.25B
PageSeer Hardware Structures – Area and Energy per Access Area(A) $*10^{-3}mm^2$, Leakage(L) mW, Rd/Wr(R/W) pJ	
PRTc	A: 54.9, L: 11.4, R/W: 14.8/14.4
PCTc	A: 36.8, L: 11.4, R/W: 14.7/16.7
HPT	A: 23.7, L: 9.1, R/W: 1.8/2.6
Filter	A: 7.7, L: 2.3, R/W: 1.4/2.7
PageSeer Structures in DRAM	
PRT	426KB
PCT	7MB with follower 884.7KB without follower

Table 2.2: PageSeer parameters.

2.4.3 Workloads

To evaluate the efficacy of our design, we run 20 different benchmarks organized into 26 workloads. They are shown in Table 2.3, with the memory footprint for our simulated period when a *single* instance of the benchmark is running. We choose eight memory intensive benchmarks from the SPEC CPU2006 suite [67], six benchmarks from Splash-3 suite [68] and six benchmarks from CORAL [69], which are used for testing HPC systems. There are two types of workloads. The first twenty are unique-benchmark workloads, where we run multiple instances of the same benchmark on different cores. Typically, we run four instances. However, in cases where the memory footprint was not enough to stress our memory system, we increased the number of cores and run more instances of the same benchmark (see Table 2.3). The next workloads are 6 mixes of benchmarks, where different benchmarks are running on different cores.

For the unique-benchmark workloads, we simulate 2 billion instructions per core, while for the mixed-benchmark workloads, we simulate until a core reaches 2 billion instructions, or a program terminates. In both cases, we perform 1.5 billion instructions of warm-up per

Workload	MB(Single)	Workload	MB(Single)
lbm×4	422	luNCon×4	520
milc×4	380	oceanCon×4	887
bwaves×4	385	barnes×8	250
GemsFDTD×4	502	radix×4	648
mcf×8	290	stream×4	457
libquantum×6	267	miniFE×4	480
omnetpp×8	164	LULESH×4	914
leslie3d×12	62	AMGmk×4	350
fft×4	768	SNAP×4	441
luCon×4	520	MILCmk×4	480
mix1: lbm-LULESH-SNAP-leslie3d			
mix2: AMGmk-luCon-radix-barnes			
mix3: miniFE-oceanCon-barnes-AMGmk			
mix4: LULESH-MILC-miniFE-stream			
mix5: luCon-radix-oceanCon-barnes			
mix6: libquantum-lbm-mcf-bwaves			

Table 2.3: Workloads.

core.

2.5 EVALUATION

2.5.1 PageSeer Characterization

The goal of PageSeer is to identify pages that are "hot" and move them to DRAM as soon as possible, while preparing the HMC structures for accesses to these pages. As a result, PageSeer's effectiveness can be quantified by how accurately it recognizes present and future "hot" pages, and how fast it manages to move them to DRAM. In this section, our simulations do not take into account contention for main-memory system bandwidth. In reality, maximum performance will be obtained when some memory requests actually access the NVM rather than the DRAM, so that the overall bandwidth of both memories is effectively utilized. However, in this section, we want to know if PageSeer can identify the pages that are worth moving to DRAM.

In Figure 2.7, we present what fraction of the main-memory accesses were serviced from DRAM, NVM, or the swap buffers for the three configurations we are comparing. Each bar of the plot represents one of the three configurations (PoM, MemPod, and PageSeer), and each bar shows a breakdown of the memory requests serviced from each memory module. We present the results for each benchmark suite and our mixes. From this figure, we see that PageSeer directs a vast number of memory requests to DRAM (88.5% on average), a small but non-negligible number to the swap buffers (2.2% on average) and the rest to NVM. Compared to the other two schemes, PageSeer can better recognize and predict hot pages, and move them to fast memory on time.

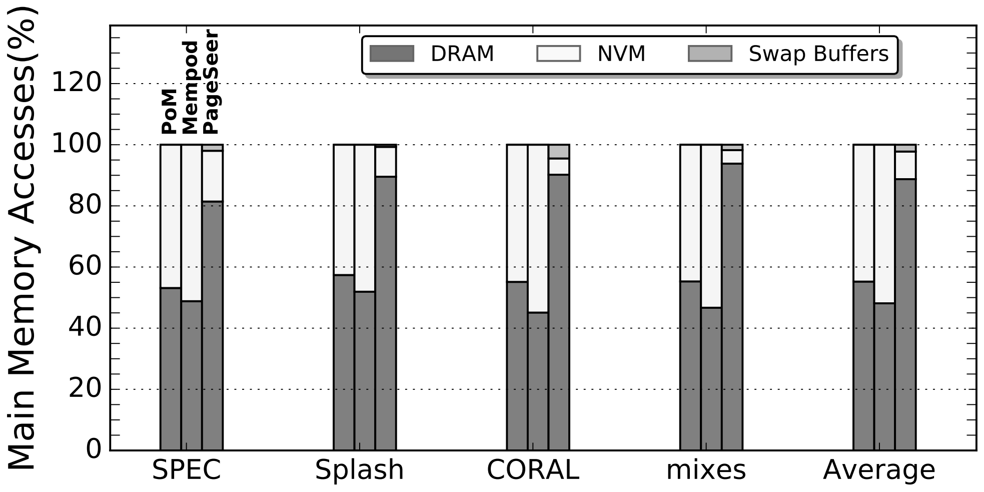


Figure 2.7: Percentage of main-memory accesses to each memory module for PoM, MemPod, and PageSeer.

The improvement over PoM and MemPod is mainly because of two reasons. First, PageSeer takes a swap decision ahead of time, so it does not wait until requests start hitting the NVM to initiate a swap. The second reason is that the MMU signal and the history of page accesses in the PCTc are an accurate indication of future accesses to a page. What is more, MemPod swaps pages at regular time intervals, which are not optimal for every application. In addition, all pages qualified for a swap start moving at the same time, causing swap bursts. As for PoM, it restricts its swap flexibility with a direct mapped re-mapping table, losing the opportunity to have multiple pages of the same swap group in DRAM.

Figure 2.8 depicts the result of the swaps for each configuration. The figure shows the positive, the negative, and the neutral main-memory accesses as a percentage of the total main-memory accesses for each configuration. We consider a main-memory access to be positive when it accesses DRAM instead of NVM thanks to a swap operation, and negative when the opposite happens. Neutral accesses are those that end-up accessing the same type of memory as a run without swaps. We see that, on average, PageSeer attains 16% and 13% more positive accesses than PoM and MemPod, respectively, and that it removes practically all of the negative accesses. To achieve that, PageSeer introduces 1% and 2.8% more swaps than PoM and MemPod, respectively.

The takeaway is that PageSeer is capable of identifying hot pages and swapping them to

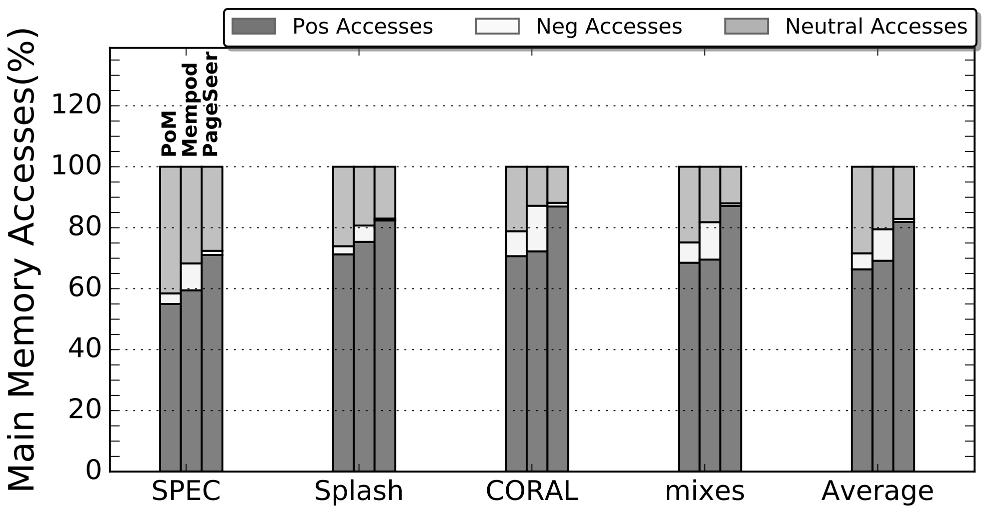


Figure 2.8: Characterization of swap effectiveness for PoM, MemPod, and PageSeer.

fast memory. The result is a high percentage of positive accesses (81.3% on average) and only 1% of negative accesses on average.

Next, we present the accuracy and effectiveness of the prefetch swap mechanism alone. Recall from Section 2.3.1 that PageSeer supports regular swaps (initiated by the HPT) and prefetch swaps (initiated by the PCTc). The latter can be triggered by either a hint from the MMU (MMU-triggered Prefetch Swaps), or a regular memory access (Prefetching-triggered Prefetch Swaps). In this discussion, we focus on prefetch swaps only.

Figure 2.9 presents the accuracy of prefetch swaps. A swap is deemed accurate when the number of accesses to the swapped page in fast memory is enough to justify the page swap cost. In our experiments, we want to attain at least 14 positive accesses to a page to recognize the prefetch swap of the page as accurate. As we can see from the figure, our mechanism is accurate in the vast majority of times. It has an average accuracy of 86.7%. `GemsFTDT` is the only benchmark for which the accuracy of our mechanism is low (28.3%) and we also perform lots of prefetches (as we will see later). This occurs because the pattern of accesses to pages changes with time. `luCon` experiences relatively low accuracy but, as we will see, the total number of prefetches is not high enough to cause an application slowdown.

In Figure 2.10, we present the percentage of swaps that are prefetch swaps. We break the prefetch swaps into prefetching-triggered and MMU-triggered. The remaining swaps to 100% in the figure are regular swaps. The benchmarks are organized into two groups. The group on the left contains those benchmarks for which PageSeer is not able to generate many prefetch swaps. This can happen because the pages for these benchmarks do not receive enough

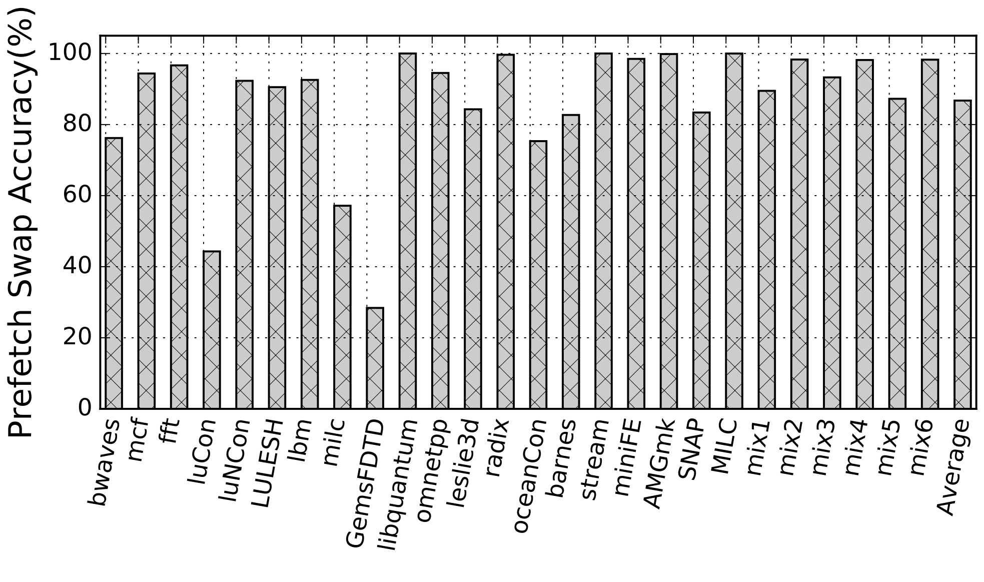


Figure 2.9: Accuracy of PageSeer's prefetch swaps.

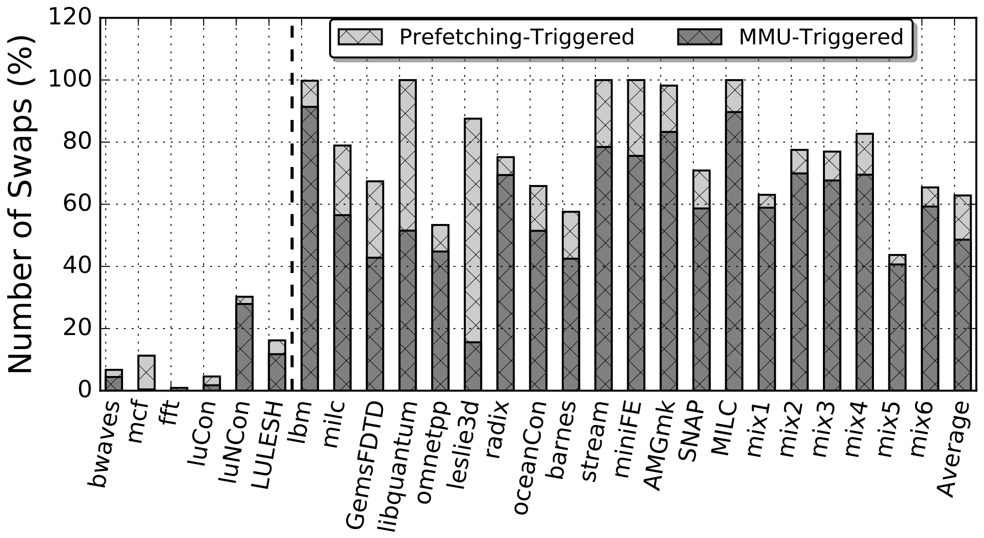


Figure 2.10: Percentage swaps that are prefetch swaps in PageSeer.

accesses to qualify for prefetching. Another reason may be that the highly-accessed pages of the application are moved to DRAM and the remaining pages are not worth swapping. However, even PageSeer's prefetch mechanism cannot find prefetch opportunities, PageSeer can still provide high performance through the use of the HPTs.

The group on the right contains those benchmarks for which PageSeer generates many prefetch swaps. We see that these are the most common benchmarks. Importantly, we see that MMU-triggered swaps are much more frequent than prefetching-triggered swaps. This shows the benefit of leveraging the MMU hints to initiate swaps.

Overall, on average for all the benchmarks, prefetch swaps account for 62.8% of all the swaps. In addition, 48.6% of all the swaps are MMU-triggered swaps.

2.5.2 PageSeer Performance

In this section, we evaluate the performance of PageSeer. We consider a variety of metrics that give insights into PageSeer. Our simulations in this section take into account the contention for the main-memory system bandwidth.

In this environment, we want to avoid saturating the DRAM channels and not using the NVM channels. Consequently, our Swap Driver uses a simple heuristic to avoid the most extreme imbalanced conditions. Specifically, when the Swap Driver observes that the DRAM channels are saturated, it considers declining some of the incoming swap requests. Specifically, it declines to swap requests if over 95% of the main-memory requests up to this point in the application have been satisfied by DRAM. While this is a primitive heuristic, it is effective.

To assess the impact of this heuristic, in Figure 2.11, we show the rate of swaps in each benchmark suite, in swaps per kilo-instruction. Each suite has two bars. *PageSeer w/ BW-opt* corresponds to PageSeer; *PageSeer w/o BW-opt* corresponds to PageSeer without the Swap Driver heuristic described above to limit DRAM channel saturation. On average, these rates are 0.19 and 0.35 swaps per Kinstruction, respectively. The heuristic has an impact.

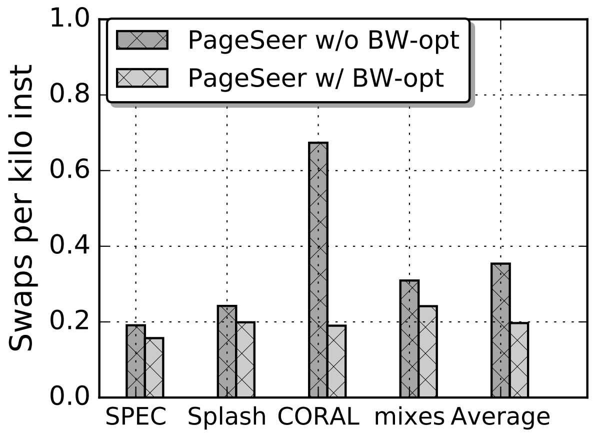


Figure 2.11: Swap rate.

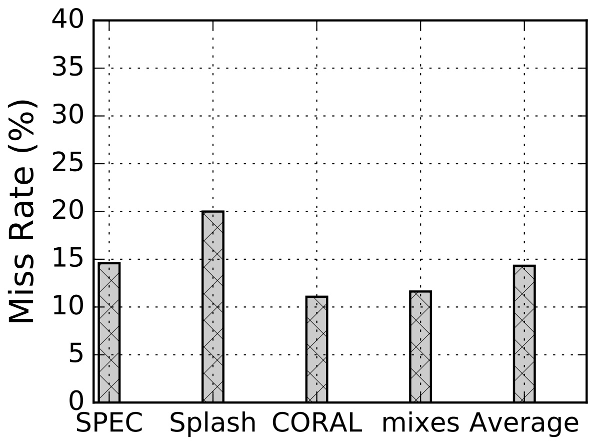


Figure 2.12: Cache miss rate of PTEs on a TLB miss.

We now examine one aspect of how PageSeer helps TLB misses. In Figure 2.12, we consider the TLB misses and record whether the resulting request for the PTE does miss in the caches (L2 or L3). The figure shows the miss rate of such requests. We see that, on average, 14.5% of these requests miss in the caches and reach the HMC. Fortunately, as indicated above, we find that over 99% of these misses are satisfied by the cache in the MMU Driver. Consequently, PageSeer is effective at reducing the latency of such accesses.

The performance benefits of PageSeer come from three factors. First, PageSeer predicts that a page in NVM will receive a large number of main memory accesses in the near future, and moves the page to DRAM before the requests arrive. The result is an increase in the number of accesses that are satisfied by DRAM. Figures 2.7 and 2.8 showed that PageSeer is very effective at exploiting this factor.

The second factor is that PageSeer can satisfy main-memory requests for pages that are currently taking part in a swap. Specifically, the swap buffers can serve memory requests if they contain the data requested. However, we showed in Figure 2.7 that only a small percentage of the total main-memory accesses are serviced from the swap buffers.

The third factor is that a request in PageSeer spends relatively little time in the HMC structures, and particularly in the PRTc. This is because, as soon as the HMC receives an MMU hint, or information from the PCTc that a follower page will be prefetched, the PageSeer hardware starts fetching the corresponding PRTc and PCTc entries for the pages. It is especially important to load the PRTc as soon as possible, since it stands on the critical path of a memory request. Note that the time wasted in a PRTc miss is not negligible. On a PRTc miss, the hardware has to access DRAM. The earlier that we fetch these entries, the better.

To assess this factor, Figure 2.13 compares the total time that requests in PageSeer and PoM spend waiting at the PRTc to load PRTc entries. Specifically, the figure shows the reduction of the waiting time in PageSeer compared to PoM. We use the same size PRTc (32KB) for both PageSeer and PoM.

The figure shows that, on average, the total request waiting time for the PRTc in PageSeer is 61.8% lower than in PoM. Of course, many of the requests may be serviced in parallel. However, this is sizable number that affects the performance of the schemes. The magnitude of this waiting time is equal to 18% of the total execution time of the benchmarks in PoM, and to 12.8% of the total execution time in PageSeer. In addition, the hit rate in the PRTc is 3.5 points higher in PageSeer than in PoM.

Taking all this into account, in Figure 2.14, we compare the overall performance of PoM, MemPod, and PageSeer. The top graph in Figure 2.14 shows the IPC of each application in PoM and PageSeer normalized to MemPod. The bottom graph depicts the Average Main-Memory Access Time (AMMAT) of each application in PoM and PageSeer, also normalized

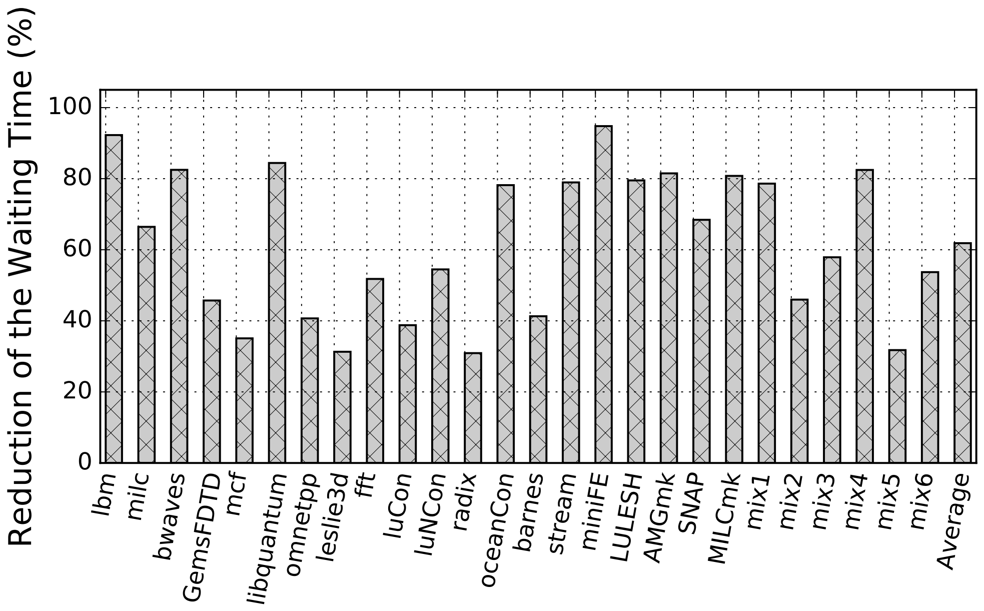


Figure 2.13: Reduction of the PRTc waiting time in PageSeer compared to PoM.

to MemPod. As in previous work [42], AMMAT is calculated as the average time spent by a main-memory request to go from the memory controller to main memory and back to the memory controller.

The top graph shows that PageSeer outperforms MemPod and PoM. On average across all the benchmarks, the IPC in PageSeer is 28% and 19% higher than in MemPod and PoM, respectively. Furthermore, the bottom graph shows that the main-memory requests in PageSeer take less time to access main memory than in the other architectures. Specifically, on average across all the benchmarks, the AMMAT in PageSeer is 37% and 29% lower than in MemPod and PoM, respectively.

Even for benchmarks where PageSeer cannot perform many prefetch swaps (shown in the leftmost side of Figure 2.10), PageSeer still delivers a performance equivalent or better than PoM. The reason is two-fold. First, the NVM HPT in PageSeer still triggers swaps. Second, PageSeer has less stall due to PRTc misses than PoM for all the benchmarks, as shown in Figure 2.13. For example, consider `mcf`. While PageSeer is unable to perform many prefetch swaps (Figure 2.10), its lower PRTc stall time induces speed-ups over PoM.

Out of the 26 workloads that we tested, PoM has a higher IPC than PageSeer only in `milc` and `GemsFTDT`; MemPod never has a higher IPC than PageSeer. These two benchmarks experience lower than average prefetch swap accuracy, as shown in Figure 2.9. As a result, their performance is harmed. This occurs because pages experience different access patterns

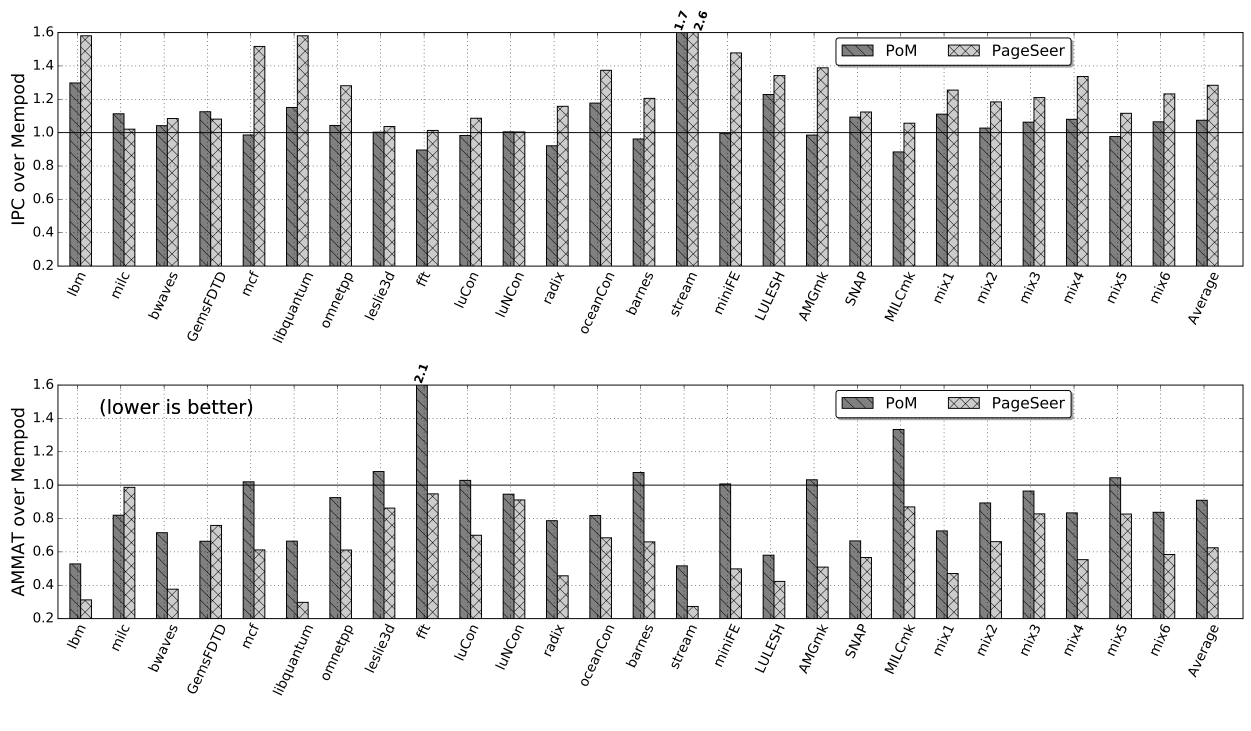


Figure 2.14: IPC and AMMAT in PageSeer and PoM normalized to the same measures in MemPod.

at different times.

Overall, our experimental results confirm that PageSeer efficiently manages a hybrid memory system, and that the communication between the MMU and the HMC increases performance.

2.5.3 Analysis of Page Correlation Prefetching

PageSeer uses page correlation prefetching to initiate some of the prefetch swaps. We want to understand the contribution of this technique to the overall performance. Consequently, we evaluate an architecture like PageSeer except that PCTc entries have no follower information. As a result, there is no correlation prefetching (*PageSeer-NoCorr*).

We find that, on average, PageSeer and PageSeer-NoCorr deliver similar performance. The reason is that, often, the MMU signal alone is able to notify the HMC about most of the future page accesses. Hence, the hardware is able to prefetch pages to DRAM without the correlation prefetching mechanism. However, the results vary across benchmarks. For example, we find that `radix` shows a performance improvement of 11% with PageSeer over PageSeer-NoCorr, while `LULESH` shows 3% performance reduction. In general, supporting correlation prefetching is advantageous when the MMU signal cannot notify the HMC about the forthcoming pages. This occurs when there are few TLB misses. On the other hand, supporting correlation prefetching is undesirable when the page access patterns change frequently.

2.6 OTHER RELATED WORK

Bhattacharjee [70] proposes TEMPO, which involves using the virtual to physical address translation process to prefetch a data cache line into the LLC. As a page walk request for the PTE reaches the memory controller, the memory controller fetches the cache line with the PTE, and uses the PTE value to fetch the cache line that triggered the page walk into the LLC. It also prepares the DRAM row buffer for accesses to nearby cache lines. TEMPO is different from PageSeer in three ways. First, TEMPO targets cache line prefetching, while PageSeer targets optimizing hybrid memory systems. Second, TEMPO prefetches a cache line into the LLC, while PageSeer initiates page swaps between slow and fast memory. Finally, TEMPO only initiates prefetches for page walk requests that miss in the LLC, while PageSeer initiates an MMU hint for every single page walk.

Section 2.2.2 described hardware schemes for managing hybrid memory systems, either as a flat memory address space or as a DRAM cache for NVM. Apart from the different swap triggers that we discussed, these schemes tackle some other aspects that can be incorporated into our scheme without major modifications. For instance, SILC-FM [43] suggests a method to swap only a portion of a page. This method can be adopted by PageSeer and save memory bandwidth. A bitmap for a page can tell us which cache lines from a page are worth swapping, and avoid moving 4KB of data. MemPod [42] mentions a clustered architecture that groups together memory controllers to be more scalable. The same approach can be embraced by PageSeer.

In Section 2.2.1, we mentioned software schemes for hybrid memory management. Besides the aforementioned methods, hybrid memory systems are an active area of research. There are proposals for hardware/sofware techniques for effective page placement [55, 71, 72, 73]. These techniques require either annotations to the applications and compiler support to identify "hot" data structures at compile time, or OS involvement to track memory activity and swap pages. The role of the hardware is to inform the OS about pages that need to be swapped. In other schemes [72, 74], the OS involvement is proposed as an optimization, to periodically update the page table entries, and relieve some pressure from the hardware remapping tables. The remapping table entries can be freed if the OS updates its page tables with the remapping information.

A well-known mechanism to hide memory latency is data cache prefetching. In theory, a data cache prefetcher could be used in PageSeer if it could help identify pages that will soon be accessed and, therefore, trigger DRAM-NVM swaps. However, data cache prefetches are not issued as early as PageSeer's MMU hints. Specifically, a regular data prefetcher will not issue any prefetches until the TLB has the translation for the page. On the other hand, our MMU hints give early information to the HMC about future accesses — before the TLB entry is filled and memory requests for the page are generated. Thus, PageSeer can start the swap earlier and prepare the HMC structures.

Researchers have examined the use of prefetching for hybrid memory systems [54, 75, 76], to prefetch pages from the slow to the fast memory. In this chapter, we examine correlation prefetching. Correlation prefetching has been used in the past for cache lines [77, 78]. Here, we use it to prefetch pages. The intuition behind it is that, many times, a program accesses a set of pages multiple times during the execution, in the same or similar order. Keeping information about the correlation between these pages can help us swap future pages before we see any requests for these pages. Our mechanism has to identify the page patterns using only LLC misses, coming from shared caches in a possibly multi-programmed environment. Of course, the page access patterns may change during program execution. Thus, we need a page correlation mechanism that is able to identify access patterns adaptively across different programs.

2.7 CONCLUSION

This chapter presented *PageSeer*, a scheme that performs hardware-managed page swapping in a hybrid memory system using hints from the page walk in a TLB miss. During the generation of the physical address for a page in a TLB miss, the memory controller is informed. The controller uses historic data on the accesses to that page and to its follower page to potentially initiate MMU-Triggered Prefetch Swaps for the page and for its follower. PageSeer also initiates other types of page swaps, as it builds a complete solution for hybrid memory. Our evaluation of PageSeer with simulations of 26 workloads showed that PageSeer effectively hides the swap overhead and services many requests from the DRAM. Compared to a state-of-the-art hardware-only scheme for hybrid memory management, PageSeer on average improved performance by 19% and reduced the average main memory access time by 29%.

CHAPTER 3: TOLERATING NON-VOLATILE CACHE READ LATENCY

3.1 INTRODUCTION

The popularity of data intensive workloads, such as HPC applications, cloud applications and databases, has intensified capacity pressure on Last-Level Caches (LLCs). While much larger LLCs are desired, especially as more cores are integrated in a chip, SRAM technology suffers from high area overhead (exacerbated by the increasing manufacturing cost at leading edge technologies [79, 80]), substantial leakage power, and scalability problems [9].

For these reasons, researchers have examined alternative memory technologies, such as eDRAM and Non-Volatile Memory (NVM). In particular, NVM technologies such as PCM [2] and, especially, STT-RAM [5] are promising candidates to replace SRAM in LLCs. Compared to SRAM, STT-RAM offers higher density and lower leakage power [10]. Unlike eDRAM, NVM is compatible with current logic and SRAM, and can be easily integrated in the same die. Further, compared to eDRAM, NVM offers lower complexity (no refresh, activate, or precharge operations), comparable read access time, and improved power-efficiency due to its ability to power gate without losing state.

However, NVMs have two main shortcomings over SRAM, namely, higher latency for both read and write operations, and a higher dynamic energy consumption per access. Moreover, read and write latencies in NVMs change based on the targeted lifetime endurance (wear-out) of the device. Therefore, replacing an SRAM LLC with an NVM one becomes a trade-off between latency, capacity, reliability and energy consumption. In this chapter, we focus on mitigating the longer NVM read latency for highly-reliable NVM caches.

Table 3.1 compares the characteristics of SRAM and STT-RAM cells. We can see that STT-RAM cells are ~4x smaller in area, while their read and write latencies are 10–30x and 25–100x higher, respectively, than SRAM's. The table does not include the energy and power numbers because the literature provides wide ranges of values, dependent on implementation and manufacturing technology [81, 82, 83, 84, 85]. Specifically, STT-RAM's leakage power is 0.15–0.48x that of SRAM's, and its dynamic access energy is 0.8–2.5x higher than SRAM's for reads and 1.5–15x higher for writes.

Prior work has tried to overcome NVM's problems of higher latency—primarily write latency—and dynamic power using solutions spanning the device, circuit, and architecture levels. At the device and circuit levels, the write access latency (primarily) can be reduced by sacrificing the retention time and non-volatility of the STT-RAM cells [100, 101, 102, 103]. Also, the transistor size can be adjusted for faster write operation [93], at the cost of higher

Characteristic	SRAM [86, 87, 88, 89]	STT-RAM [90, 91, 92, 93, 94] [95, 96, 97, 98, 99]
Area (F^2)	70-150	15-40
Read Latency (ns)	0.3	3 - 10
Write Latency (ns)	0.3	8 - 30

Table 3.1: Comparing SRAM and STT-RAM characteristics.

power, lower density, and lower reliability [10]. Such approaches limit the full potential of NVM caches and do not solve the increased read latency problem. Indeed, degrading NVM characteristics to reduce the write latency introduces the need for periodic refresh, which increases design complexity and energy consumption, and hinders non-volatility [104, 105]. Additionally, adjusting the NVM cell size to reduce write latency limits NVM capacity and introduces higher error rates [10, 106].

At the architecture level, the most popular solutions to address NVM's higher latency and dynamic power combine smaller SRAM caches with larger STT-RAM caches in a hybrid cache hierarchy [82, 84, 107, 108]. However, these solutions focus mostly on write latency (not the focus of this chapter) or use inclusive caches (not so popular today). In addition, they use a considerable amount of SRAM storage, plus complex logic to decide which cache lines to swap between SRAM and STT-RAM. As a result, they limit the area savings from NVM and increase the energy consumption.

In terms of access latencies, several proposals mitigate the performance impact of long NVM write latencies [83, 85, 109, 110, 111, 112]. Interestingly, little emphasis has been placed on mitigating the NVM read latency, based on the common assumption that the SRAM and STT-RAM read latencies are similar. However, measurements on fabricated STT-RAM caches and observations by industry vendors show a significant difference in read latency between STT-RAM [81, 82, 90, 91, 92, 93, 95, 96, 97, 98, 99, 113] and SRAM [86, 87, 88]. Specifically, as shown in Table 3.1, FinFET-based 6T SRAM arrays perform read operations with a 300ps latency [88], while the fastest STT-RAMs can only attain 3ns latencies at best [94].

To take advantage of NVM for LLCs, we need a low-cost architectural solution that can tolerate the higher read latency of STT-RAM without sacrificing capacity, reliability, or non-volatility. This chapter proposes such an architectural solution, which we call *Cloak*. Cloak exploits page-level data reuse in the LLC to hide NVM read latency. Specifically, on certain L1 DTLB misses to a page, the hardware transfers LLC-resident lines of the TLB-missing page from the LLC NVM array to a set of small SRAM page buffers. Such buffers will service future requests to this page. To enable the low latency detection and high-bandwidth transfer of lines of a page to the SRAM page buffers, Cloak uses a LLC layout that accelerates the discovery of LLC-resident cache lines from the page. Further, we develop an adaptive replacement policy for the page buffers to increase their utilization and achieve better performance and energy consumption.

We evaluate Cloak with full-system simulations of a four-core processor running 14 workloads. On average, Cloak outperforms an SRAM LLC by 23.8% and an NVM-only LLC by 8.9%—in both cases, with negligible additional area. Further, Cloak improves the ED^2 metric by 39.9% and 17.5% relative to these designs.

3.2 BACKGROUND

3.2.1 STT-RAM Limitations and Opportunities

STT-RAM has emerged as a promising candidate to replace SRAM in LLCs [82, 108, 113, 114] because it provides higher density and lower leakage than SRAM. However, the viability of STT-RAM is inhibited by higher read and write access latencies, and by higher dynamic energy than SRAM. This is because STT-RAM requires a high thermal barrier, which in turn increases the switching current of the device and the cell access latency. The high thermal barrier is a consequence of the requirement for a large retention period, typically expected of non-volatile memory cells.

Past research has exploited a trade-off that exists between retention time and write access latency, to design STT-RAM cells whose write access latency is tolerable for practical on-chip integration [100, 101, 102, 103]. Such techniques help lower the device write time to 8–30ns, as shown in Table 3.1. These proposals focus on reducing write latency because read latencies tend to be smaller, namely 3–10ns.

One major caveat is that the STT-RAM array latency cannot be *pipelined*. During a cell access with a latency reported in Table 3.1, the STT-RAM array is blocked from servicing other requests. In contrast, SRAM array accesses are pipelined, and data can move to/from the cache array every cycle, achieving higher throughput than STT-RAM.

STT-RAM write latency is constrained by bit-level error guarantees to ensure reliable operation across the lifetime of the chip. In our case, with 16MB of STT-RAM per LLC slice (Table 3.2), the Bit Error Rate (BER) of the STT-RAM cell needs to be lower than 10^{-10} to ensure a 99.99999% yield with SECDED ECC. This is based on the assumption that a cache line is fetched from a single STT-RAM array block of 2MB. Based on results from prototype devices[93, 97, 115], STT-RAMs with such error rate guarantees can only achieve a bitcell write latency of ≈8ns and a read latency of ≈3.2ns. Furthermore, recent innovations in the quality of magnesium oxide (MgO), which acts as the dielectric material, pave the way for high endurance STT-RAM cells in future designs [116]. As a result, the literature reports that STT-RAM is a competitive alternative to SRAM for LLC caches. It can have an endurance in the order of 10^{12} to 10^{13} write cycles [107, 109, 112, 116], especially under normal temperature environments as the one we target [117].

Overall, given the reasonable tradeoffs between STT-RAM write latency and BER, the most pressing problem is to mitigate the impact of the higher read latency of large STT-RAM LLCs operating in ambient conditions. This is the focus of this chapter.

3.2.2 NVM Cache as an SRAM Replacement

Prior work on NVM caches [83, 85, 109, 110, 111, 112] has focused on mitigating the effect of long-latency write operations rather than read operations. The work can be categorized into three groups: methods to reduce, stall or bypass writes to NVM caches, NVM cell optimizations, and hybrid SRAM/NVM caches.

Solutions in the first group identify write contention in the NVM cache that can stall latency-critical reads, and try to take writes off the critical path of subsequent reads [83, 85, 109, 110]. These techniques assume the same read access latency for NVM and SRAMs.

Proposals that optimize NVM cells improve NVM cache write performance at the expense of retention time and area [93, 100, 101, 102]. These optimizations are not trivial, given the trade-offs between access latency, area, and retention time of NVMs [5, 106, 113]. In addition, decreasing the retention time of NVMs introduces refreshes, similar to DRAM, which increase energy consumption and complicate the design. Moreover, it limits the capacity of NVMs and introduces higher error rates. Importantly, it does not address the problem of the non-pipelined and higher read latency.

Hybrid cache proposals [82, 84, 107, 108] split a cache into an NVM and an SRAM portion, typically by partitioning a cache set into SRAM and NVM ways. These proposals monitor address reuse [82, 84, 108] and migrate frequently used data to the SRAM portion of the cache, and rarely used data to the NVM portion. However, these techniques have several shortcomings. First, they only target inclusive caches, which are not widely used nowadays. Second, they dedicate a large portion of cache capacity to SRAM, therefore reducing the density benefit of NVM and increasing leakage power. The area overhead of the SRAM portion is 25–100%, assuming a 4:1 density between NVM and SRAM [82, 84, 108]. Third, they need large structures of several KBs to accurately monitor cache line activity. Fourth, they must migrate data between SRAM and NVM, which further increases the number of writes to NVM, the energy consumption, and the area overhead. Fifth, in exclusive and victim LLCs, it is hard to monitor the reuse of individual addresses because LLC hits result in promoting lines to faster levels of the cache hierarchy. As a result, the overhead and complexity of recording data reuse increases. Finally, these techniques do not consider that the non-pipelined nature of NVM accesses introduces higher overhead to line migration.

3.3 MOTIVATION

To increase the cache capacity in multi-cores, designers are architecting LLCs as victim caches. We propose using NVM in the LLC, to enable higher LLC capacity for the same area with high retention time and low error rates. To this end, in this chapter, we observe that an L1 DTLB miss on a page *that was already referenced in the past* is a good hint that some LLC-resident cache lines of the page will be reused soon. Consequently, we identify such DTLB refills and bring likely-to-reuse lines from the LLC into a small SRAM structure.

To support this insight, we model a 3-level cache hierarchy with a 16MB LLC and 4KB pages. Figure 3.1 shows that 94.9% of LLC hits originate from pages whose translation is re-filled back into the L1 DTLB. This data implies that, upon an L1 DTLB page re-fill, there should be enough LLC-resident data from this page that will be re-referenced.

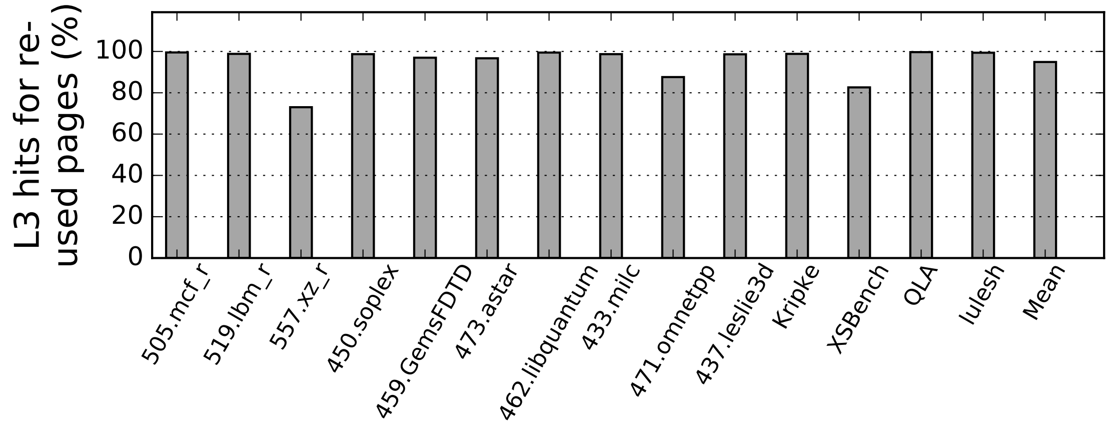


Figure 3.1: Percentage of LLC hits that originate from accesses to pages that were re-filled into the L1 DTLB.

We also measure the number of LLC-resident cache lines belonging to a 4KB page at the point when the page is re-filled into the L1 DTLB. We use 4MB and 16MB LLCs. Figure 3.2 shows the frequency of the number of such lines. Even though in most cases 0–3 lines are resident, there is a long tail of up to 60-64 resident lines, which increases with larger L3

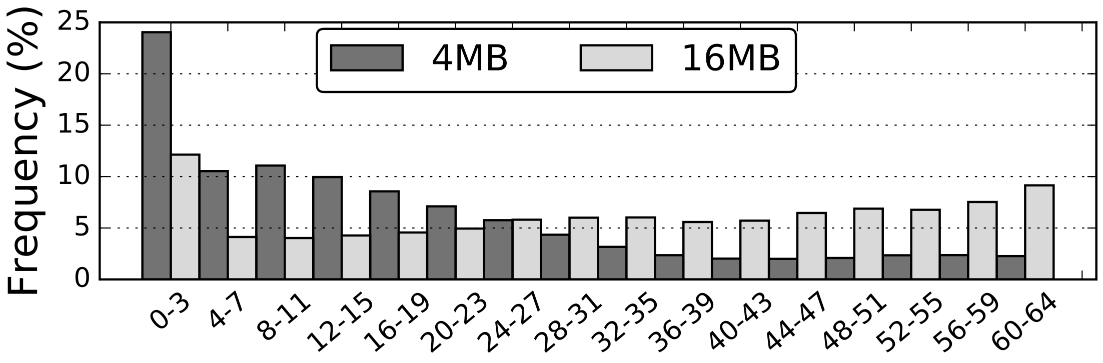


Figure 3.2: Frequency of the number of LLC-resident cache lines from a 4KB page that is re-filled into the L1 DTLB. The figure shows data for 4MB and 16MB LLCs.

size (16MB). Therefore, we conclude that the number of requests hitting in the LLC and originating from a L1 DTLB refilled page likely rises with LLC size. Cloak builds on these observations to architect a solution that hides the increased read latency of NVM caches and increases overall throughput.

3.4 DESIGN OVERVIEW OF CLOAK

3.4.1 Main Idea

Cloak is a hardware mechanism that takes advantage of certain L1 DTLB misses to exploit data re-use in large NVM LLC caches and hide NVM's higher non-pipelined read latency. The NVM LLC is augmented with small SRAM buffers, which we call *Page Buffers (PB)*. PBs hold data transferred from the NVM LLC. Each PB can hold a copy of LLC-resident cache lines originating from the same page. To trigger the copy of data into a PB, Cloak leverages the L1 DTLBs. When a miss in the L1 DTLB occurs for a previously-accessed page, hardware passes this information to the LLC, which finds and copies the LLC-resident lines of this page to a PB.

Previously accessed data from the page have a high chance of being accessed again when the page translation is re-filled in the L1 DTLB—due to temporal locality. To facilitate the retrieval of a page's cache lines from the LLC, we introduce a new LLC data layout that places the lines of a given page in the same physical cache row.

Figure 3.3 shows the architecture of Cloak, where the new or modified hardware structures are shaded, and the added connections between the L1 DTLBs and the L3 Controllers are

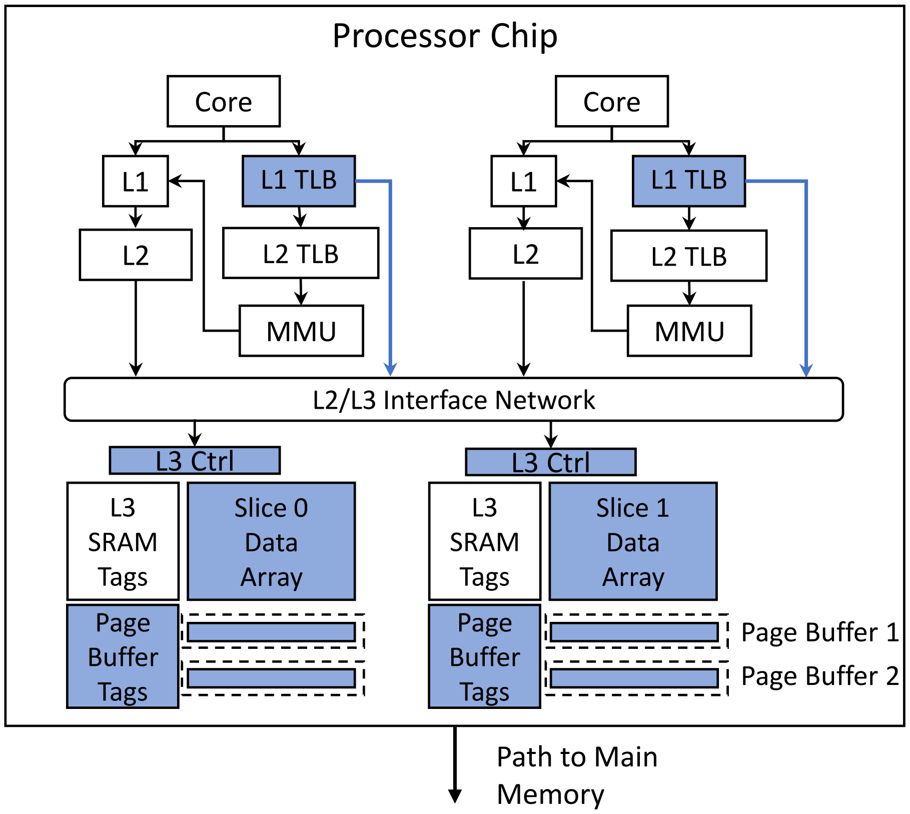


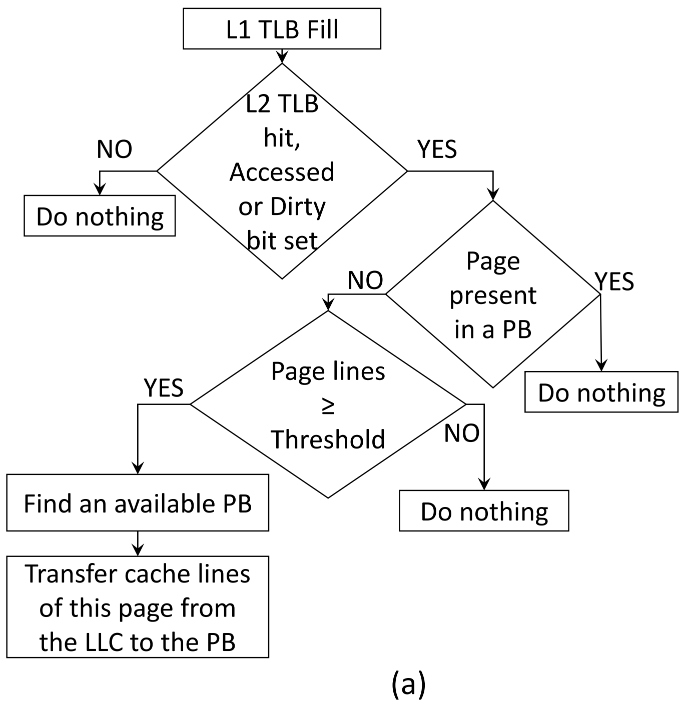
Figure 3.3: Cloak architecture, with the new or modified hardware structures and connections colored.

shown in lighter color. In this section, we describe the overall operation of Cloak to fetch data to the PBs and service memory requests. Subsequently, Section 3.5 discusses the architectural details of the Cloak design.

3.4.2 Cloak Overview

The core operations of Cloak consist of data movement from the NVM LLC data array to the PBs on a DTLB signal, and servicing of subsequent requests either from the PBs or the NVM cache array. Figure 3.4 shows the control flow diagrams of these actions.

TLB triggered Page Buffer transfer. PBs are small SRAM-based cache structures which act as fast access buffers to the NVM LLC. Each PB can hold a subset of the NVM-resident cache lines of a given page. Promoting NVM-resident lines from a page into a PB can hide the read latency of a future NVM LLC access, by exploiting intra-page spatial



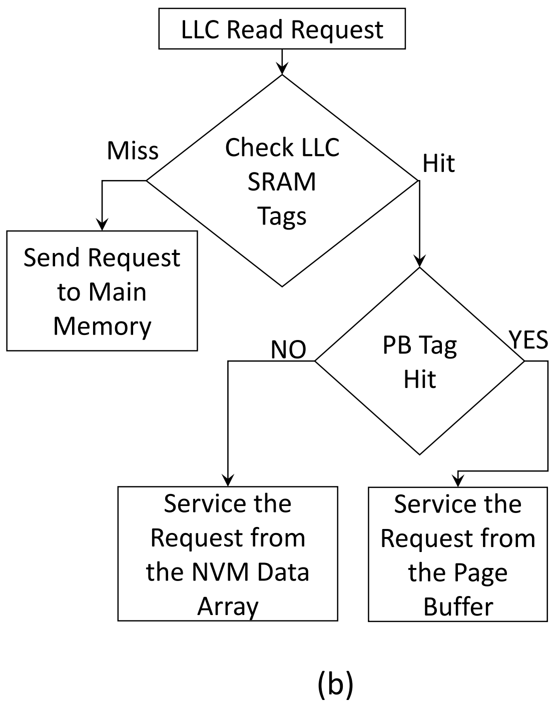

LLC Read Request
Check LLC SRAM Tags
Miss
Hit
Send Request to Main Memory
PB Tag Hit
NO
YES
Service the Request from the NVM Data Array
Service the Request from the Page Buffer
(b)

Figure 3.4: Control flow diagrams: (a) promotion of lines from a page to a PB, and (b) servicing a read request from the LLC.

locality [118] and by reducing the number of accesses to the NVM data array.

The promotion of a page's cache lines to one of the PBs is depicted in Figure 3.4a. When an L1 DTLB miss occurs, the PTE for the page is fetched and Cloak determines whether this page was previously referenced. To identify whether the page was referenced in the past, Cloak checks if the page is in the L2 DTLB or, if it is not, if either the *Accessed* or *Dirty* bits of its PTE [119] is set. A set Accessed bit indicates that the page was accessed in the past. This bit is set by hardware when the page is first read or written, and is only reset by the OS to track the frequency of accesses to the page. A set Dirty bit indicates that the page was written, and hence, was referenced before. The Dirty bit is set by the processor the first time that the page is written to, and is only cleared by software.

If Cloak finds that the page was used in the past, it sends a signal to the LLC controller containing the physical address of the request that caused the DTLB refill. The LLC controller first determines if the PBs already contain lines from this page. If not, Cloak decides whether to transfer the page to a PB and, if so, which PB to use. To decide whether to transfer the page's lines, Cloak checks the LLC tags to calculate the population of NVM-resident lines from this page. A promotion to a PB occurs only if the population size exceeds a programmable threshold, so that the cost of fetching the lines to a PB can be amortized across the expected number of PB hits. Then, Cloak selects an available PB to promote the page's cache lines according to the PB replacement policy (Section 3.5.4).

Servicing requests to the LLC. In Cloak, a hit in the LLC can obtain the data from the NVM cache array or from a PB. The PB contents are kept coherent with the NVM cache. Thus, writes to the LLC (e.g., due to an L2 eviction) also check the PBs and, on a hit, update both the NVM data array and the PB. Completion of write requests is signaled to L2 when the request is buffered in the LLC queues. It does not wait until when the request updates the NVM data array and PB.

Figure 3.4b shows servicing a read request to the LLC. The LLC controller checks the Physical Page Numbers (PPNs) in the PB Tags and in the LLC SRAM tags in parallel, to determine a hit or a miss. If the LLC tag check determines that there is an LLC miss, the request is forwarded to main memory. If the LLC tag check determines an LLC hit and the PB tag check indicates a PB hit, the request is serviced from the PBs otherwise it is serviced from the NVM data array. An LLC NVM hit can be serviced in parallel with a PB hit to a different address. In-flight read accesses to the slow, non-pipelined NVM data array do not block younger reads to the PBs.

3.5 CLOAK IMPLEMENTATION

3.5.1 Data Layout

Transferring the lines of a page from the LLC to a PB requires finding all NVM-resident lines of that page. To avoid massive tag lookups and NVM cache read operations, we introduce an alternative data layout for the NVM cache. The proposed layout forces all the lines of a given page to be placed in a single physical row of the LLC. A physical row contains multiple cache sets, each with multiple cache lines, and each physical row may contain lines from multiple pages.

For the Cloak LLC, we assume a physically distributed, logically shared LLC cache that acts as a victim of private L2 caches. While we evaluate a specific design point, the design of Cloak itself does not preclude other potential organizations of the cache hierarchy. The cache is split into equally-sized slices. Each slice has its own controller and can independently service any type of request. We use STT-RAM for the data array and SRAM for the tag array for two reasons. First, the tag array is much smaller than the data array and its relative area overhead is small. Second, LLCs are typically highly set-associative, and tags are accessed before data to minimize the dynamic energy of the data array access. Having NVM tags would add significant latency to all LLC traffic.

The LLC tag array supports both conventional accesses (e.g., triggered by L2 misses) and 4KB-aligned page-level data transfer requests triggered by L1 DTLB fills. We distinguish the two by referring to the former as Cache Line Requests (CLR) and to the latter as Page Transfer Requests (PTR).

By placing all the lines of a given page in the same LLC physical row, we limit tag searching to only 64 entries for every incoming PTR (assuming 4KB pages and a 64B line size). We alter the LLC cache indexing to support our new data layout for both PTR and CLR accesses. Our addressing scheme is presented in Figure 3.5. To pick the physical row, we use some bits of the Physical Page Number (PPN) called Row Index. Once the row is selected, we use a subset of the Physical Page Offset (PPO) bits called Set Index to select a set within the physical row. Then, some of the PPN bits (*Tag-High*) and of the PPO bits (*Tag-Low*) are used as tag. Finally, the remaining PPO bits are used as block offset (Figure 3.5).

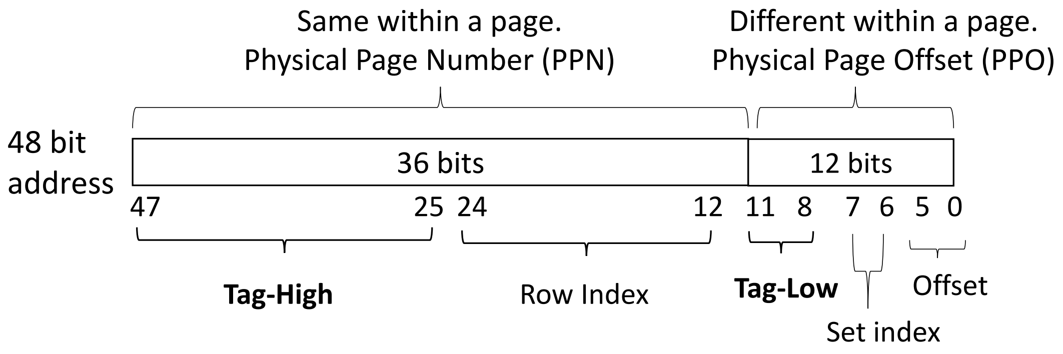


Figure 3.5: Cloak LLC addressing scheme.

PTRs and CLRs differ in the tag match logic. Specifically, for tag matching, a PTR access ignores the page offset bits and uses *Tag-High* bits only. In contrast, a CLR access uses both *Tag-High* and *Tag-Low* bits for tag matching.

The lines of a page could be split across LLC slices, mapping to the same physical row in each slice. However, in order to simplify the tag hit logic and NVM to PB data movement, we choose to map the entire page in the same LLC slice. Note that our layout does not impose any restrictions on the LLC organization (e.g., line size, associativity, etc.).

Example. To illustrate the proposed layout, we show an example with a single-slice of 32MB size, 16-way set-associative LLC with 64B cache lines and 4KB pages. The LLC has

8192 physical rows, each organized in 4 sets, 16 ways each. Each physical row is 4KB and can be banked if needed. As shown in Figure 3.5, the physical address (PA) has 48 bits, the 12 least significant ones are the PPO, and bits 0-5 form the line offset.

The cache index bits include the row index bits (bits 24:12), which select the row of the cache, and the set index bits (bits 7:6), which select the cache set within a row. Note that the row index bits (bits 24:12) do not include any PPO bits. Bits 11:8 and 47:25 form the tag, split into Tag-Low and Tag-High parts, respectively. For tag matching, a CLR selects a row and a set using indexing bits 24:12 and 7:6, respectively, and finds a match using the tag bits (bits 47:25 and 11:8). For tag matching, a PTR selects a row using the row index bits (bits 24:12) and finds all matches using the subset of tag bits lying outside the PPO, namely the Tag-High (bits 47:25). Thanks to this layout, the PTR does not search the entire cache; it only checks the Tag-High (bits 47:25) of the 64 lines in the selected cache row.

The dynamic energy of a CLR tag access is proportional to the 16 lines x 27 tag bits comparison (432 bits). The dynamic energy of a PTR tag access is proportional to the 64 lines x 23 tag bits comparison (1,472 bits). A PTR tag access consumes 3.4x more energy than a conventional CLR tag access. Triggering PTR tag searches only on DTLB re-filled pages, lowers the overall energy cost of accessing the PBs.

Our data layout could be extended to optimize for huge pages. However, we find that such a design is not efficient, as huge pages increase the overhead of tag lookup and data movement, while it is unlikely that a large fraction of their lines will be LLC-resident. In Section 3.5.5, we discuss how to efficiently handle huge pages.

3.5.2 Promotion of Cache Lines to Page Buffers

Cloak populates each PB with LLC-resident cache lines from the same page. The process is as follows. Once a PTR reaches the LLC controller, the hardware checks if any PB already has lines from the accessed page. If not, the hardware checks the LLC tags to find if the LLC holds more than a threshold number of cache lines of the accessed page. If so, the LLC-resident lines of the page are transferred to a PB. The NVM cache is inclusive of the PBs. Hence, the transferred lines are not invalidated from the NVM data array.

Cloak provides hardware to bring cache lines to the PBs. According to Figure 3.2, the LLC may only contain a few of the lines from a given page. Hence, PBs will be sparsely populated. In order to increase PB utilization, we propose PBs that are smaller than a page. However, given that the size of a physical row is equal to a page, steering the data from a row to a PB is not trivial and may require additional metadata and complex routing logic. To simplify both the metadata and the routing overhead, we propose using PBs of size equal to half a page (2KB), and multiplex the two halves of a page into the same PB.

Example. Figure 3.6 shows an example that promotes cache lines from page A into a PB. Since we have 4KB pages (and, hence, 4KB physical rows) and use 2KB PBs, we logically partition a physical row into two 2KB regions, and promote the lines of a page into the PB in two steps: first from the leftmost region of the physical row, and then from the rightmost region of the row.

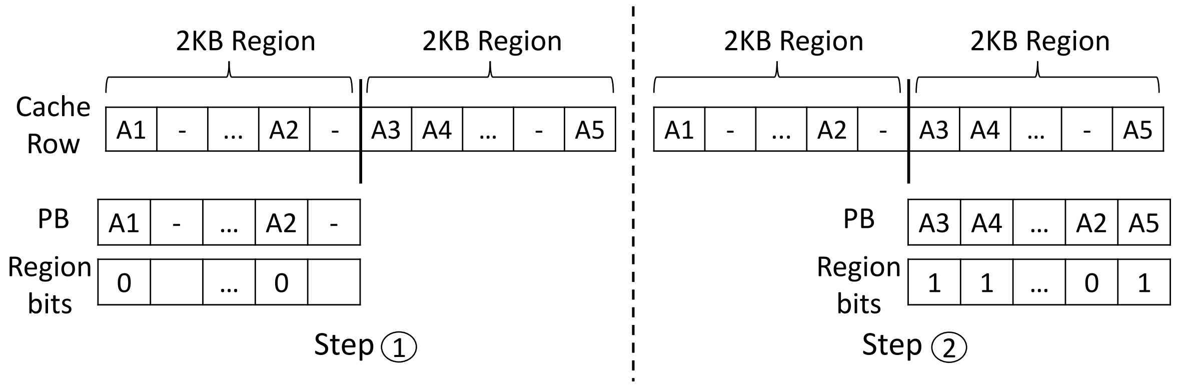


Figure 3.6: Promotion of the lines of a page to a PB.

The left side of Figure 3.6 shows the first step. At the top, we see the the state of the cache row. The leftmost region has lines in its first position (*A1*) and in the one before last (*A2*). Hence, we promote these lines into the PB. To simplify the routing, as shown in the figure, the lines are placed in the PB in the same slots that they use in the 2KB region. The PB does not need to store any address tags because any memory access will check the SRAM LLC tags first, to decide whether the corresponding PB line is valid. However, each PB slot has a bit to identify which region the line comes from. This bit is needed to fully identify the line. In the example, since the two lines come from the leftmost region, the bits are 0. In our example, we need 32 such bits, which we call *Region* bits.

The right side of Figure 3.6 shows the second step. The top part repeats the cache row state. The rightmost region has lines in its first (*A3*), second (*A4*), and last (*A5*) positions. Hence, we promote these lines into the PB and set the Region bits of the entries to 1.

Note that the two lines in the first position of the two regions wanted to use the same PB slot, and we had to pick a winner. In the example, we picked *A3* over *A1*. To pick a winner, Cloak uses a simple algorithm that guesses which of the two lines is more likely to be used in the future. Specifically, Cloak records if the address A that triggered the DTLB miss belongs to the first or second half of the page. Then, when populating a PB, on a conflict in a PB entry, the line from the same half of the page as A overwrites the line from the other half of the page. This algorithm guesses that, because of spatial locality, the former is more likely to be accessed soon that the latter.

Given Cloak's proposed LLC organization, the operation of promoting the lines from the two regions (and, in another design, from potentially more regions) into a PB does not stall the LLC pipeline more than a single read access would. Indeed, all the cache lines of a page are on the *same physical row*, and thus they are promoted to a PB from the NVM data array with *a single read operation*. The writes into the PB are also pipelined: as the first region is written, the second performs the checks.

3.5.3 Tag Checks

To keep track of the pages and cache lines that are present in the PBs, Cloak employs an array called the *Page Buffer Tags* (PB Tags) (Figure 3.7). Each entry in the PB Tags corresponds to one PB. An entry has: (a) the PPN of the page whose lines are stored in the PB, (b) a *Replacement* counter to manage PB replacement, (c) a *Residency* counter that tracks the number of valid lines in the PB, and (d) the *Region* bits discussed above.

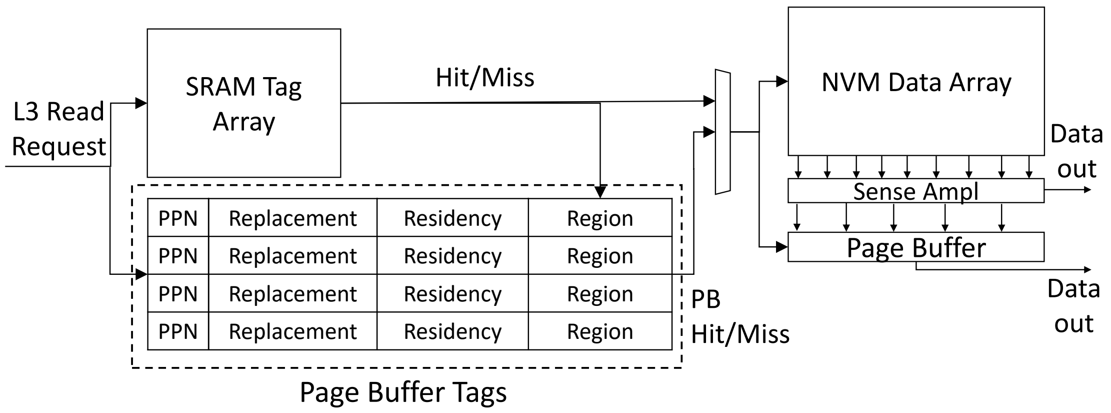


Figure 3.7: LLC read request path in Cloak.

Both PTR and CLR use the PB Tags to determine whether a PB contains data for the page requested. In the case of a PTR, the PB Tag search uses the PPN of the requested page. In a CLR, the PB Tag search uses the PPN of the requested page and the correct bit in the Region field corresponding to the requested address.

The CLR operation starts by accessing both the LLC tag array and PB Tags simultaneously (Figure 3.7). It uses the PPN bits of the PB Tags to identify if a PB contains lines from the page accessed. It uses the LLC tag array to identify if and where the requested line resides in the LLC. If the address of the line is not found in the LLC tag array, an LLC miss is declared.

However, if the LLC tag array indicates a cache hit, Cloak checks for a PB hit. A PB hit will occur if the region of the physical row with the matching address is the same as the one indicated by the Region bit of the corresponding location of the PB. In this case, the line is accessed from the PB in the same position. Recall that, during data transfer, lines were moved from the LLC to the PB without reordering. If the Region bit does not match or the PPN bits do not match, the line is accessed from the NVM-LLC data array.

Note that the access to the PPNs in the PB tags overlaps with the access to the LLC tag array. The access to the Region bit in the PB tags is only performed after the LLC tag array access (Figure 3.7). However, accessing the Region bit only extends the critical path by one cycle (when both the PPN and the LLC tag array hit).

The PB contents are always kept synchronized with the NVM LLC. When a line is written to the LLC, the corresponding line in the PB, if present, is updated. For this reason, there is no need to write back PBs to the NVM cache. There is also no need to keep valid or coherence state bits in PB Tags because the LLC tags provide such information. Whenever an LLC line is invalidated (due to an external probe or L2 promotion), or evicted (due to an LLC replacement), the corresponding valid bit of the line in the LLC tags is reset. No other action is needed: given that the PB hit logic waits for the LLC tag search to complete, a CLR will not read the PB slot data if its corresponding LLC line is invalid, even if the data is still resident in the PB entry.

The Residency counter in the PB Tags tells how many cache lines are valid per PB entry. This counter is set when the lines of a page are moved from the NVM cache to the PB. It is decremented when one of the lines is invalidated or evicted from the LLC. It is incremented when an L2 victim is installed in the LLC and copied into the PB. The Residency and Replacement counters are used to handle PB replacement, as we discuss in Section 3.5.4.

Example. The PBs add little area overhead to the LLC. To see why, consider an example based on Figure 3.5. A PB is composed of tag and data. For the tag, we have a 36-bit PPN and assume a 10-bit Replacement counter. The Residency counter needs $log_2(PBsize/linesize)$ bits, which is 5 in our example. The Region bits are $PBsize/linesize * log_2(4KB/PBsize)$, which is 32 in our case. The total comes to 83 bits per PB tag entry. The size of the PB data per entry is 2KB. Based on this, each PB adds a ≈0.05% area overhead over a 16MB LLC NVM slice.

3.5.4 Page Buffer Replacement Policy

Cloak needs to find an available PB to promote a page's cache lines, when all PBs are in use. To pick a PB, Cloak uses a PB replacement algorithm that considers: (i) how many cache lines are resident in a PB, and (ii) the frequency of accesses to the page in the PB. The goal is to capture the dynamic behavior of accesses to each PB, and neither replace a PB too early (before its entries are accessed), nor keep a page resident in the PB if it is not being accessed.

Specifically, when a PB is loaded, its Replacement counter is set to the product of the Residency counter and a programmable constant called *Activation Period*. At every cycle, the Replacement counter is decreased by one. When a PB entry is accessed, either for a read or a write operation, its Replacement counter is recalculated by multiplying the current value of the Residency counter with the Activation Period. Furthermore, PB accesses change the Residency counter. On a PB read, Cloak decrements the Residency counter because a line from the PB is moved to the private caches. On a PB write, Cloak increments the Residency counter because a line is written back from L2. Once the Replacement counter reaches zero, the PB entry is subject to replacement. If there are multiple PBs with a replacement counter of zero, we can pick any of them at random.

3.5.5 Huge Page Management

Modern systems support huge pages, such as 2MB and 1GB, to alleviate TLB pressure. Even though we described Cloak in the context of 4KB pages, Cloak can support huge pages without any modification to the NVM cache layout.

We envision a physical row in the LLC cache to still hold 4KB of data. However, if we chose to transfer a whole 2MB page into a PB, we would need to search many rows. Moreover, larger PB entries would likely be underutilized.

Consequently, we use a different design where Cloak only transfers individual 4KB chunks of data at a time from a huge page into a PB. Specifically, when a huge page entry is re-filled into the L1 DTLB, Cloak only brings into a PB the 4KB chunk of this huge page that contains the address that triggered the DTLB miss. In addition, the L1 DTLB records this 4KB chunk that triggered the DTLB miss. Subsequently, when an access to the same huge page, but a different 4KB chunk occurs, Cloak triggers a transfer of the new 4KB chunk, and again records the chunk in the DTLB. In this way, Cloak can have multiple 4KB chunks of the same huge page active in the PBs.

Cloak adds this support for 2MB and 1GB pages. For the 2MB pages, Cloak needs to add

9 bits per L1 DTLB entry to record the most-recently-promoted 4KB chunk. For the 1GB pages, Cloak needs to add 18 bits per L1 DTLB entry. These are minimal area overheads.

3.6 EVALUATION METHODOLOGY

3.6.1 Modeled Architecture and Infrastructure

We use full-system cycle-level simulations to model a server architecture with 4 cores and 64 GB of main memory. The main architecture parameters are shown in Table 3.2. Each core is out-of-order with private L1 and L2 caches, and a shared LLC. The L1 and L2 caches use stride and next-line prefetchers, respectively, as implemented by the SST [61] framework. The L2 cache is inclusive of L1, while the LLC is populated by L2 victims. The baseline system uses an SRAM-based physically distributed, logically shared, victim LLC. For Cloak, we modeled an increased-latency, non-pipelined, STT-RAM-based LLC and the proposed hardware modifications. We use the published data (Table 3.1) to estimate the minimum read latency of the STT-RAM LLC data array (i.e., 3 ns). Note that the higher STT-RAM latency as presented in Table 3.1 refers to the STT-RAM LLC *data array* (i.e., 10 cycles at 3.2GHz), and not the *total round trip latency* to access the LLC from the core (as shown in Table 3.2). There are private L1 and L2 DTLBs, and a page walker per core. For our evaluation, we integrate the Simics full-system simulator [60] with the SST [61] framework. To model main memory we used the DRAMSim2 [63] memory simulator. We use Intel SAE [66] on top of Simics for OS instrumentation. Finally, we use CACTI [120] and McPAT [121] to calculate the timing and energy parameters of our processor, all SRAM-based tag and data arrays required by Cloak, and the Baseline SRAM-based LLC. We scaled the STT-RAM cache energy parameters according to prior work [83, 84]. We model one extra clock for determining a PB hit/miss because the LLC tag search must first determine the line location inside a PB.

3.6.2 Configurations and Workloads

We compare four different design configurations.
Baseline: SRAM-based LLC with the latency and size parameters described in Table 3.2.
NVM-Only: LLC with STT-RAM for the data array and SRAM for the tag array with the parameters of Table 3.2, but without PB support and with conventional indexing.
Cloak: LLC with STT-RAM for the data array and SRAM for the tag array with the proposed data layout and PB support.

Processor Parameters	
Multicore chip	4 OoO cores, 4-issue, 22nm, 3.2GHz
Ld-St queue; ROB	92 entries; 192 entries
L1 cache	32KB, 8-way, 2 cycles round trip latency (RT), 64B line
L2 cache	512KB, 8-way, 14 cycles RT, 64B line
Prefetchers	L1 cache: stride prefetch.ß L2 cache: next-block prefetch
LLC SRAM cache	4MB/core, 16-way, 1 slice/core, 64B line
	53 cycles RT, 2 cycles tag latency, 12 cycles data latency
	Energy: Read/Write 0.47/0.48nJ, Tags 4pJ, Leak: 1.4W
L1 DTLB	64 entries, 4-way, 2 cycles RT
L2 DTLB	1024 entries, 12-way, 12 cycles RT
NVM cache parameters	
LLC NVM (STT-RAM) cache	16MB/core, 16-way, 1 slice/core, 64B line
	63 cycles RT read latency, 78 cycles RT write latency
	2 cycles tag access latency, 22 cycles data access latency
	(of which 10 cycles are not pipelined)
	Energy: Read/Write 0.95/6.3nJ, Tag 7pJ , Leak: 829mW
Page Buffers (PB)	20 PBs, 2KB/each, 43 cycles RT
	Energy: Read/Write 12/13pJ, Tag 12pJ, Leak: 4.1mW
PTR signal latency	6 cycles
STT-RAM cache to PB threshold	6 cache lines
PB activation period	20 cycles per active cache line
PB area overhead	1% area overhead over LLC data array (0.05% per PB)
Main-Memory Parameters	
Capacity	64GB
Channels; Banks	2; 8
Latency	190 cycles RT (on average)
Freq; Bus width	1.6GHz DDR; 64 bits per channel
System Parameters	
Host OS	Ubuntu Server 16.04

Table 3.2: Architectural parameters used for evaluation. In the table, RT means round trip latency from the core.

O-SRAM: Optimistic hybrid design with conventional indexing, pipelined access latency and energy characteristics of *Baseline*, combined with STT-RAM area density.

To evaluate the efficacy of our design, we simulate 14 benchmarks. The benchmarks are shown in Table 3.3 with their memory footprint and L2 misses per kilo instructions (MPKI). We chose ten benchmarks from the SPEC CPU 2017 (Group A) [122] and SPEC CPU 2006 (Group B) [67] benchmark suites with high MPKI to stress the memory subsystem. We also run four benchmarks from the CORAL [69] and CORAL2 [123] suites (Group C), as representative HPC applications. The memory footprint of our benchmarks does not fit in

Workload	Footprint (MB)	L2 MPKI	Workload	Footprint (MB)	L2 MPKI
Group A			**Group B**		
505.mcf_r	613	39			
519.lbm_r	409	10	450.soplex	436	10
557.xz_r	800	3	459.GemsFDTD	146	4
Group C			473.astar	372	18
Kripke	608	39	462.libquantum	267	11
XSBench	110	63	433.milc	123	7
QLA	375	11	471.omnetpp	388	12
lulesh	110	15	437.leslie3d	62	3

Table 3.3: Workloads.

the L2 and can stress the LLC. We select the region of interest (ROI) with SimPoint [124] for the SPEC® workloads and we instrument the source code for the others. Starting from a checkpoint inside the ROI, we warm-up the architectural state by running 500 million instructions before simulating 1.5 billion instructions.

3.7 EVALUATION

3.7.1 Cloak Performance and Energy

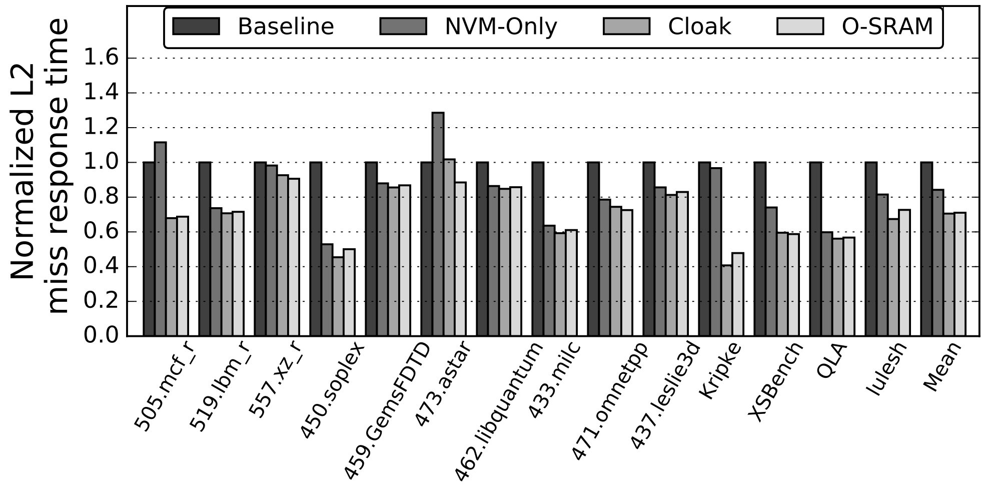


Figure 3.8: Normalized L2 miss response time - w/ 4KB Pages

In this section, we evaluate the performance of Cloak. When replacing an SRAM-based cache with NVM (STT-RAM), there are two factors that affect application performance. The first is the higher read and write latencies of STT-RAM. The second is the lower cache

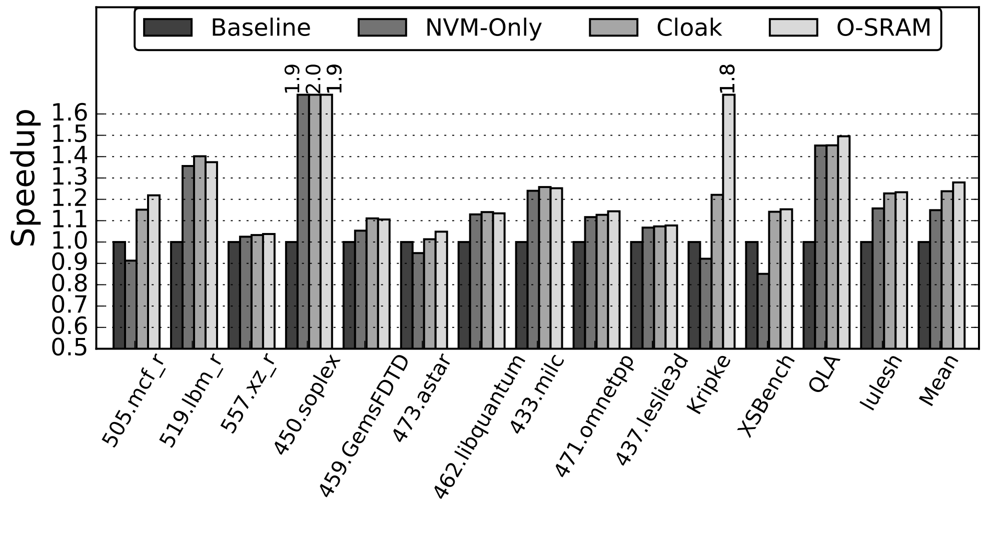


Figure 3.9: Speedup - w/ 4KB Pages

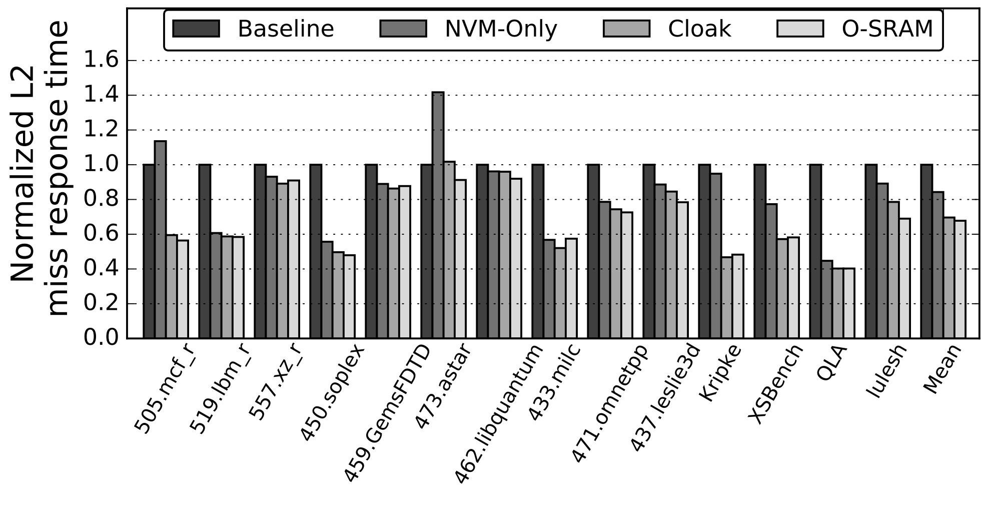


Figure 3.10: Normalized L2 miss response time - w/ 4KB+Huge Pages

miss rate due to the higher area density of NVM technology. We consider two different metrics in the following figures to show the performance impact of Cloak. Figures 3.8 and 3.10 show the L2 miss response times for read CLRs, while Figures 3.9 and 3.11 depict the application speedup. All figures are normalized to the Baseline configuration. We conducted experiments on a system with 4KB pages only, and a system with Transparent Huge Pages enabled (2MB and 1GB pages).

Figures 3.8 and 3.10 show the L2 miss response time which is calculated as the total

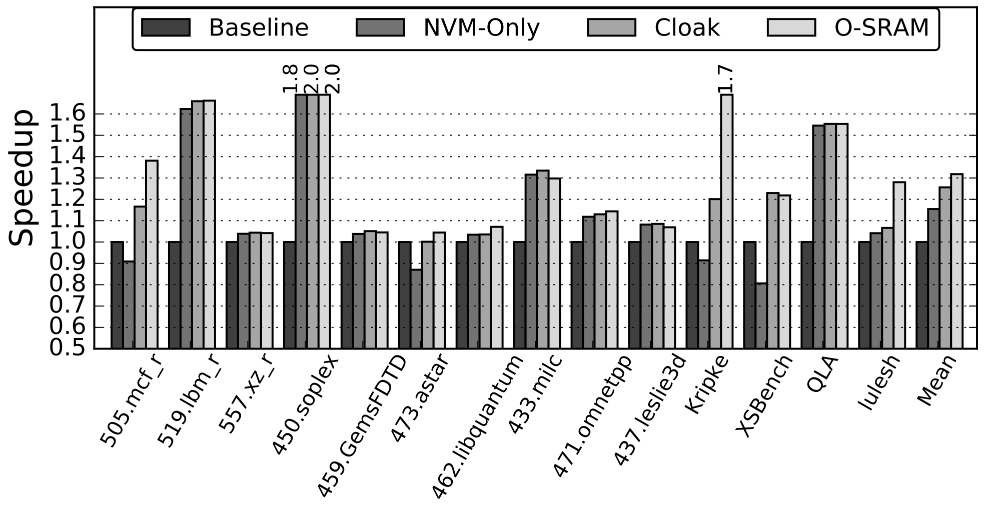


Figure 3.11: Speedup - w/ 4KB+Huge Pages

number of cycles from issuing an L2 miss until the miss response reaches back to the L2. On average, Cloak reduces the L2 miss response time by 30.0% and 30.5% over Baseline, with and without Huge Pages. This impact is really close to that of the O-SRAM configuration. The NVM-Only configuration lowers the L2 miss response time by only 15.8% and 15.9%. It does not achieve the same reduction as Cloak or O-SRAM because of its higher and non-pipelined LLC hit latency. These results indicate that the PBs are effective at reducing the NVM cache read latency—practically as much as O-SRAM.

Figures 3.9 and 3.11 show the application speedup over Baseline. We see that NVM-Only LLCs can increase performance. The reason is the larger LLC capacity achieved via NVM technology, which can greatly decrease the LLC miss rate. However, there are benchmarks where NVM-Only experiences performance degradation compared to Baseline (505.mcf_r, 473.astar, Kripke, and XSBench), because the lower LLC miss rate cannot compensate for the higher LLC hit latency. Benchmarks with high L2 MPKI and high LLC hit rate suffer more from the increased read latency of an NVM-based LLC. For instance, 473.astar and XSBench with Huge Pages experience 13% and 19% lower performance than Baseline, respectively.

On the other hand, Cloak consistently attains higher performance than Baseline and NVM-Only. There are times when it even outperforms O-SRAM. This can happen for benchmarks with high PB hit rate because a PB hit has lower access latency than an SRAM-based LLC hit. This is due to the lower routing latency observed retrieving data from the PB data array compared to a much larger SRAM-based LLC slice. On average, Cloak is 25.6% and

23.8% faster than Baseline with and without Huge Pages, respectively, while NVM-Only is 15.5% and 14.9% faster than Baseline. For some benchmarks, Cloak outperforms Baseline by up to 97%, effectively hiding the increased read latency of STT-RAM.

We also tested Cloak's efficacy by running mixes of four benchmarks. We find that NVM-Only and Cloak outperform baseline by 24% and 31%, without Huge pages and by 27% and 33% with Huge pages, respectively. Additionally, we also tested all the low L2-MPKI SPEC benchmarks and find that Cloak always achieves the same or slightly higher performance (1-2%) than SRAM and NVM-Only cases. The rest of our evaluation focuses on a system that utilizes only 4KB pages. However, the performance trends remain the same when huge pages are enabled.

To further understand the performance impact of Cloak, we present two more performance metrics in Figures 3.12 and 3.13. In Figure 3.12, we show the drop in LLC Misses per Kilo-Instructions (MPKI) for the four configurations. The increased capacity with STT-RAM greatly reduces the LLC MPKI of the applications. The MPKI drops by up to 55% across all benchmarks—including those with the highest MPKI such as XSBench (50% drop). In most cases, Cloak achieves an LLC MPKI close to that of O-SRAM.

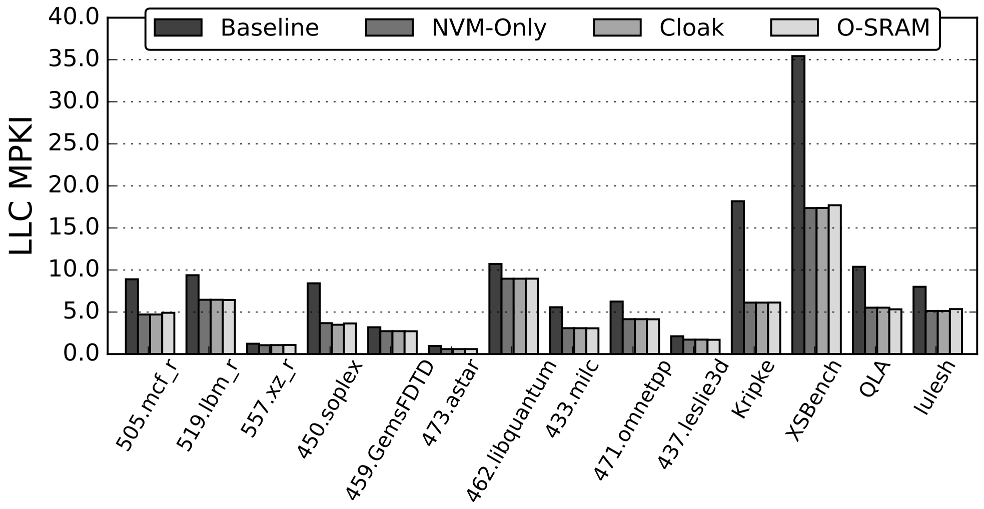


Figure 3.12: LLC Misses per Kilo-Instructions (MPKI).

To isolate the performance impact of PBs, Figure 3.13 compares the total time that read requests spent in the LLC in NVM-Only and Cloak. This time is calculated as the total number of cycles from when an L2 miss is issued until the response is sent back to L2 from the LLC (in case of an LLC hit), or until the LLC declares a miss (in case of an LLC miss). Note that the two configurations have similar LLC MPKIs. Therefore, their cycle count difference

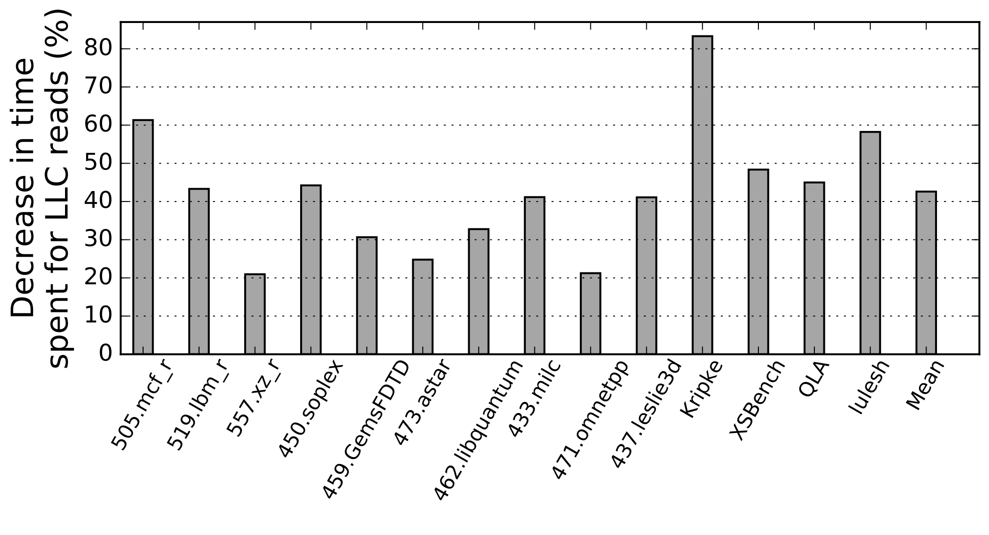


Figure 3.13: LLC read latency reduction of Cloak over NVM-Only.

depends on the PB hit rate in Cloak. Figure 3.13 shows that Cloak notably reduces these LLC read latency cycles and, therefore, accelerates the LLC read traffic. On average, LLC CLRs spent 42.5% less time in the LLC with Cloak than with NVM-Only. The PBs are able to speed-up Cloak because they service CLRs much faster than the LLC NVM-based data array (both in latency and bandwidth). Moreover, when CLRs are serviced from the PBs, they do not block the LLC data array pipeline, giving the opportunity to subsequent CLRs that do not target PB-resident regions, to proceed in parallel. As a result, the PBs not only service requests faster, but also increase the overall throughput of the LLC.

Figure 3.14 shows the ED^2 of the different configurations normalized to Baseline. The bars are broken down into the contributions of the core plus private caches, the LLC, and main memory. Overall, we see that all STT-RAM designs have a lower ED^2 than Baseline. On average, NVM-Only reduces the ED^2 by 22.4%, while Cloak reduces it by 39.9%. The reasons are the lower execution times of the STT-RAM configurations, the lower leakage power of the STT-RAM cache (which is the main energy contributor of the LLC), and the reduced number of accesses to main memory. Compared to the NVM-Only design, Cloak consumes more energy in the LLC. This is because Cloak performs more tag checks and read accesses to the NVM data array to fetch data to the PBs, a Cloak request needs to check both the LLC and PB tags, and the PBs consume extra leakage power. However, this extra energy is compensated by a faster execution of Cloak because it services many requests from the PBs. O-SRAM reduces the ED^2 by 43.3% over Baseline on average, delivering the best energy efficiency. However, we see that, in some cases, Cloak is better. O-SRAM has the same leakage power as baseline. So, in cases where the performance of O-SRAM cannot make up for its extra leakage, Cloak is better.

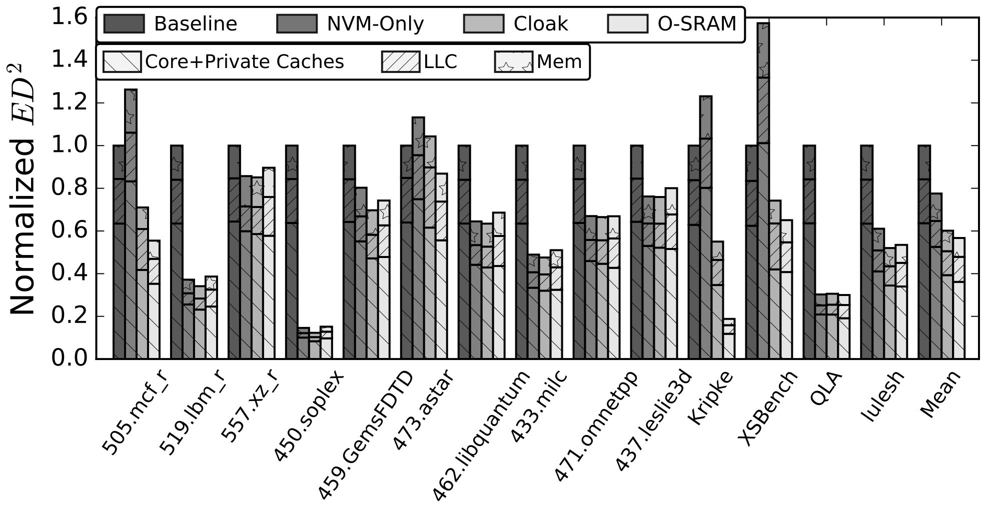


Figure 3.14: ED^2 normalized to Baseline.

3.7.2 Cloak Characterization

To achieve high performance, it is crucial to maximize the use of PBs. We find that 84.4% of the PTRs sent by Cloak to the LLC are for pages that have at least 6 cache lines in the LLC. Moreover, 99% of the PTRs are able to find a PB to promote the lines.

To get further insight, Figure 3.15 shows the percentage of LLC hits serviced from the PBs (instead of from the STT-RAM data array). On average, 54% of the hit CLRs are serviced from the PBs instead of the STT-RAM data array. The benchmarks with the highest LLC hit traffic, such as XSBench (45 HPKI or Hits per Kilo-Instructions) and Kripke (32 HPKI) hit in the PB 57% and 48% of the time, respectively. This leads to substantial performance gains of Cloak over NVM-Only, as shown in Figure 3.10.

We now quantify the coverage of PTRs to the LLC in Figures 3.16a and 3.16b.[1] Figure 3.16a shows what percentage of the cache lines promoted into PBs are actually accessed from the PBs. We see that, on average, CLRs reference 51.1% of the cache lines promoted to the PBs. This hit ratio is due to the PB replacement algorithm that favors the victimization of PBs with few lines or a low number of hits. Note that when we promote cache lines into a PB, we do not pollute the LLC or L2. This is because the PBs simply hold a copy of the data present in the STT-RAM data array.

[1]Group A, Group B, and Group C are the mean of the SPEC CPU® 2017, SPEC CPU® 2006, and Coral benchmarks of Table 3.3, respectively.

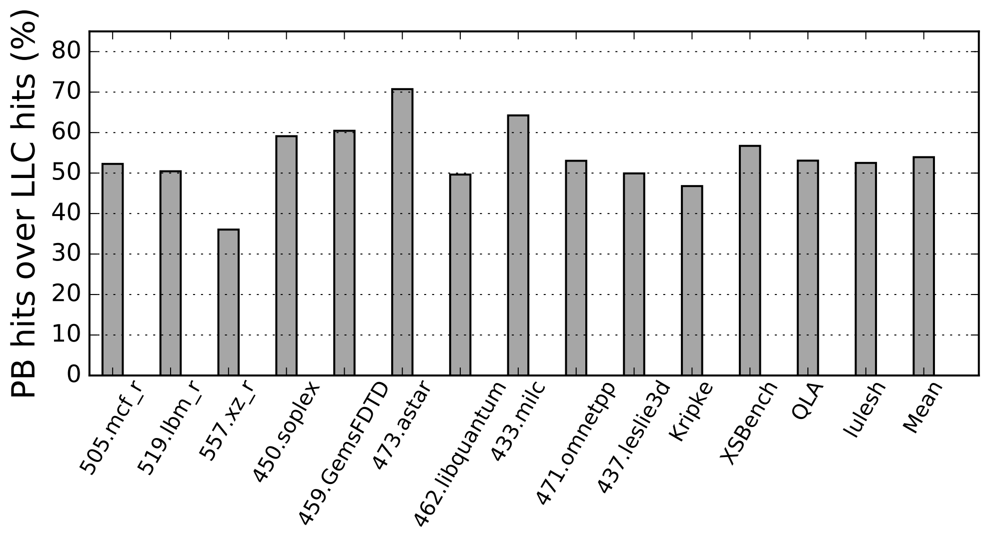


Figure 3.15: Percentage of LLC hits that are serviced by PBs.

Figure 3.16b shows what percentage of the LLC-resident cache lines of a page are promoted to a PB in a PTR. This number is not 100% for two reasons. First, for a given page, some of the lines from different regions in the same physical row may conflict with each other, and cannot all be promoted to the PB. Second, if the number of LLC-resident lines is less than a threshold, Cloak does not promote the page. On average, Cloak promotes 68% of the LLC-resident cache lines to a PB—or about 26 lines.

3.7.3 Alternative Cloak Design

To further highlight the benefits of Cloak, we evaluate a scheme that fetches NVM-resident cache lines to the L2 cache instead of the PBs, using the same trigger as Cloak. For this experiment, we keep the data layout we introduced for the NVM cache, so that we can identify the cache lines of a page with a single read operation. Moreover, as an optimization, we make sure that the L2 cache always prioritizes read requests from the core over LLC-to-L2 prefetches. In addition, when the L2 MSHR entries are heavily utilized (i.e., ~90%), we drop outstanding LLC-to-L2 prefetches.

We find that this design is not competitive with Cloak: on average, it is 19.8% slower than Cloak and increases the writes to NVM by 183%. This is because bulk prefetches from LLC to L2 saturate the interconnect, causing core requests to stall while arbitrating for the bus. Moreover, fetching many lines to L2 causes L2 thrashing, which in turn increases L2 misses. This is especially the case for benchmarks with high L2 MPKI such as XSBench. This

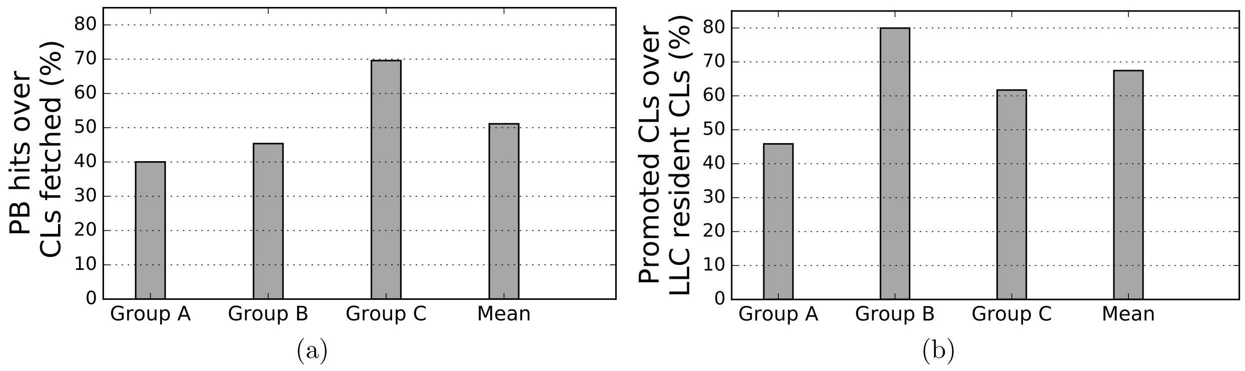


Figure 3.16: PB use characterization: (a) percentage of cache lines promoted into PBs that are actually accessed from the PBs, and (b) percentage of the LLC-resident cache lines (CLs) of a page that are promoted to a PB in a PTR.

benchmark takes ~90% longer to complete with the new design than with Cloak because of the increased traffic between the L2 and the LLC. Only benchmarks with a small L2 MPKI and a high PB hit ratio, such as 450.soplex and 437.leslie3d, can benefit from this design, and attain a performance that is comparable to Cloak's.

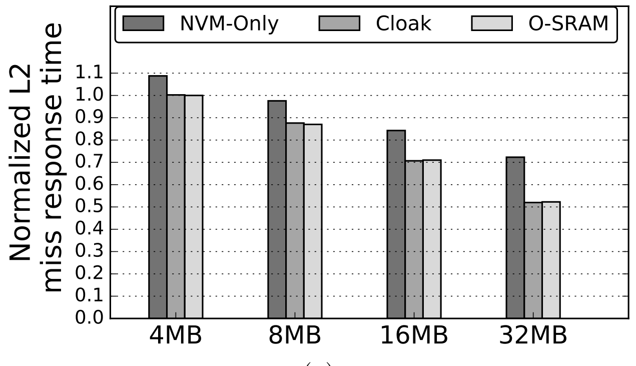
An aggressive L2 prefetcher, that tries to prefetch the same cache lines as Cloak, faces the same performance bottleneck. Furthermore, if the Cloak LLC data layout is not used, read requests from the core suffer from low LLC read bandwidth due to the non-pipelined STT-RAM data array access.

3.7.4 Sensitivity Analysis

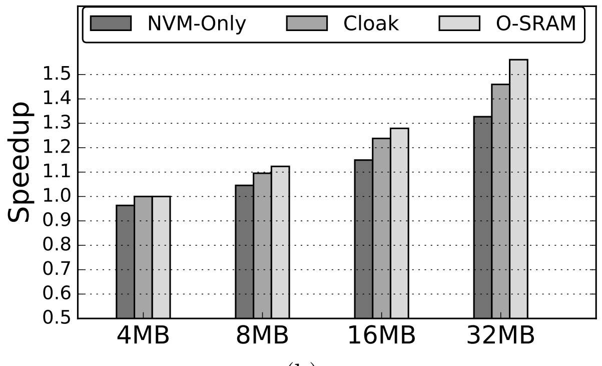
Finally, we perform two sensitivity analyses. First, we examine the sensitivity of Cloak to the LLC cache size, which is the primary parameter dictating the LLC hit/miss rate. Figures 3.17a and 3.17b show the average L2 miss response time and the average speedup, respectively, across all benchmarks, as the size of the LLC cache increases from 4MB to 32MB per core. All results are normalized to Baseline, which has an SRAM-based LLC with 4MB per core.

Figure 3.17a shows that the relative L2 miss response time drops with the increase in LLC size for all the schemes. Cloak has lower L2 miss response time than NVM-Only for all configurations. It has practically the same L2 miss response time as O-SRAM because the PBs provide even faster access than a larger SRAM LLC slice due to their smaller routing overhead.

Figure 3.17b shows that the speedup of all the schemes increases with the LLC size.



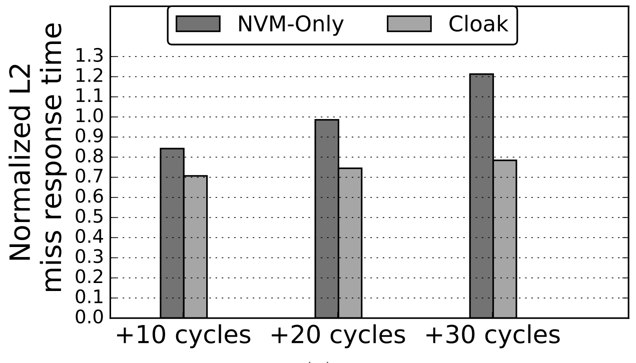
(a)



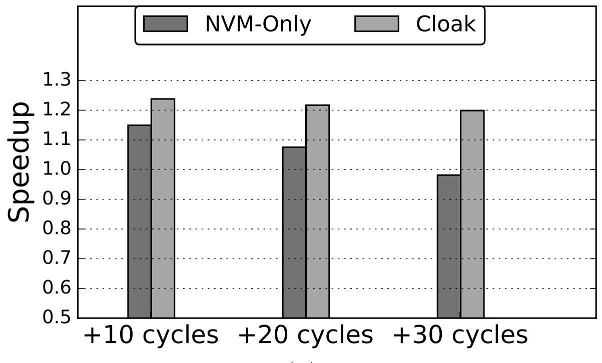
(b)

Figure 3.17: Sensitivity analysis of different LLC sizes per core over Baseline with an SRAM-based LLC of 4MB per core: (a) Normalized L2 miss response time and (b) speedup.

This is because of the increasingly lower LLC miss rate. For all LLC sizes, Cloak delivers higher speedups than NVM-Only and lower speedups than O-SRAM. Interestingly, Cloak can tolerate the higher read latency of STT-RAM and achieve equal performance to Baseline with a 4MB LLC.



(a)



(b)

Figure 3.18: Sensitivity analysis of different LLC read latencies over Baseline with an SRAM-based LLC of 4MB per core: (a) Normalized L2 miss response time and (b) speedup.

We also analyze the effects of increasing the read latency of STT-RAM LLC caches, while keeping the cache size at 16MB per core. Figure 3.18a and Figure 3.18b show the average L2 miss response time and the average speedup, respectively, across all benchmarks, as the LLC read latency is increased. We increase the latency by lengthening the NVM-based LLC data array read latency by 10, 20 and 30 cycles over the SRAM baseline. The configuration with +10 cycles represents the NVM cache configuration we simulated in our prior experiments.

All results are normalized to Baseline, which has an SRAM-based LLC of 4MB per core. The three designs represent an STT-RAM with a minimum read latency of ~3ns, ~6ns and ~9ns.

As the STT-RAM LLC read latency increases, the relative L2 miss response time increases, and the speedup drops. These trends occur for both NVM-Only and Cloak, although they are less prominent for Cloak. In all cases, Cloak has a lower L2 miss response time and a higher speedup than NVM-Only. This is because the PBs can tolerate part of the higher STT-RAM array read latency. Even with an STT-RAM of 30 cycles, Cloak is faster than Baseline.

3.8 OTHER RELATED WORK

Page Caches. Prior work has looked into the use of die-stacked eDRAM as large LLCs [125, 126, 127, 128, 129, 130, 131, 132]. eDRAM-based caches are typically organized in pages (Page Caches) instead of blocks to avoid massive tag storage. When a request reaches a page cache and the page is not cached, the whole or a subset of the page [126, 127] is brought from main memory, generating off-chip traffic. The capacity of page caches is underutilized, since a page allocates cache space even for lines that are not fetched. This reduces cache capacity. In addition, page caches add extra overhead to keep track of a page's useful footprint. Cloak does not sacrifice any LLC capacity, it does not need to track any footprint, and does not generate any off-chip traffic. Instead, it brings the LLC-resident lines into the PBs. Page caches cannot be easily designed as victim or non-inclusive LLC caches—e.g., storing a victim line requires the allocation of space for the whole page. Instead, Cloak can be integrated with LLCs of different inclusion properties. Finally, Cloak can still replace an SRAM-based L3 in a system with a page cache employed as an extra cache (e.g., L4).

Techniques to Hide High Latency. To hide the increased latency of NVM caches, in addition to the advanced techniques discussed in Section 3.3, one can use conventional techniques such as prefetching and dead block elimination [85]. These proposals are orthogonal to Cloak and can be used in conjunction with it. However, LLC prefetchers incur increased complexity and can saturate memory bandwidth (Section 3.7.3) when using NVM caches [83, 112, 133].

The advantage of using the address translation to make early decisions has been demonstrated before for page walks. Specifically, TEMPO [70] uses PTE page walk requests that miss in the cache hierarchy to prefetch the cache line that caused the page walk to LLC from main memory. PageSeer [32] uses page walk information to swap pages in a DRAM-NVM hybrid main memory system. Cloak is different because it hides the higher read latency of

NVM-based LLCs. Second, it uses a different trigger, the DTLB miss on a page used in the past.

3.9 CONCLUSION

This chapter presented Cloak, a novel, low cost NVM LLC architecture that uses small SRAM-based page buffers to tolerate the higher and non-pipelined latency of NVM reads. An L1 DTLB miss on certain pages triggers the data transfer of LLC-resident lines belonging to the page from the NVM LLC to the page buffers. The buffers will service subsequent requests for this page, and use a novel replacement algorithm to achieve high performance and low energy consumption. Cloak effectively hides the higher latency of NVM reads. On average, Cloak outperformed an SRAM LLC by 23.8% and an NVM-only LLC by 8.9%—in both cases, with negligible additional area. Further, the ED^2 of Cloak was 39.9% and 17.5% lower, respectively, than these two designs.

CHAPTER 4: ARCHITECTURAL SUPPORT FOR PROGRAMMABLE NON-VOLATILE MEMORY FRAMEWORKS

4.1 INTRODUCTION

Byte-addressable Non-Volatile Memory (NVM) technologies such as 3D XPoint [1], Phase Change Memory (PCM) [2, 3, 4], and Resistive RAM (ReRAM) [6] have recently gained much attention. They offer high storage density, low static power, non-volatility, and performance characteristics that are comparable to those of DRAM [7]. Thanks to these properties, NVM is expected to create disruptive changes to many application domains and software systems.

To a large extent, the success of NVM depends on the availability of user-friendly programming frameworks for software development [134]. For this reason, many NVM programming frameworks have been proposed (e.g., [11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 23, 24, 25, 135, 136, 137, 138]). Although they have different implementations, they share the same challenges, including what persistence abstractions to provide, how to identify the objects to allocate in NVM versus DRAM, and how to identify the stores that modify the persistent state.

Most of the frameworks rely heavily on programmer involvement. They require programmers to mark all the objects that must be allocated in NVM (e.g., [11, 12, 13, 14, 15, 16, 17, 18]). Many frameworks also require programmers to identify the stores that modify NVM (e.g., [11, 12, 16, 17, 18, 19, 20, 21]), and potentially augment the code with instructions that write back a cache line to NVM (*CLWB*) [22], order instructions (store fence or *sfence*), and log state in NVM. This results in programming difficulty, introduces software bugs, and generates nonreusable code.

Ideally, NVM frameworks should assume all of these aforementioned responsibilities. One class of NVM frameworks that come close to this ideal is *Persistence by Reachability* frameworks (e.g., [23, 24, 25]). The idea is that the programmer only identifies the *durable roots*—i.e., the few entry points into the program's data structures that should reside in NVM. There is no need to mark all the persistent objects. Then, during execution, the runtime software ensures that all the data structures that are reachable from the durable roots are crash-consistent. The runtime does so by moving the data structures to NVM when needed, identifying persistent stores and adding CLWB and sfence instructions, and performing logging when required.

Unfortunately, while these frameworks are user-friendly, their runtime adds substantial performance overhead [23, 139]. Since the properties of program structures change dynamically, the framework has to perform checks at every program load and store, and move data structures between DRAM and NVM at runtime. To make persistence by reachability frameworks attractive to the community, and the paradigm of choice among programmers, they must have lower overhead.

The operation of such frameworks dictates that they possess fine-grain dynamic information about the program's data structures, such as memory location and persistence properties. Currently, there is no efficient hardware technique to provide such information. Approaches like Intel's MPX [140] can only provide limited information (i.e., pointer bounds) about program structures, while approaches that rely on tagging memory locations [141, 142, 143] incur too much overhead to be used in production code.

In this chapter, we introduce novel hardware support targeted to accelerate NVM programming frameworks that provide persistence by reachability. Our scheme, named P-INSPECT, focuses on reducing the main source of overhead in these NVM frameworks: state checks performed before read and write accesses. Specifically, in P-INSPECT, an application read-/write includes inexpensive hardware checks of the state of the accessed data structure. In the common case, the hardware checks conclude that no special action is needed, and the read/write completes normally. Otherwise, a runtime software handler is automatically invoked, which performs any needed framework operations. To perform these checks efficiently, P-INSPECT uses cache-coherent hardware bloom filters. In addition, P-INSPECT also speeds up the execution of persistent writes. It does so by combining writes with CLWB and sfence instructions in hardware.

Our evaluation shows that P-INSPECT retains the programmability advantages of persistence by reachability frameworks while removing most of their execution overhead. Specifically, we run a key-value store with various Yahoo Cloud Service Benchmarks, and several kernels on a state-of-the-art persistence by reachability framework. With P-INSPECT hardware support, real-world applications reduce their number of executed instructions and their execution time by an average of 26% and 16%, respectively. We also compare P-INSPECT to an ideal runtime that has no persistence by reachability overhead, and demonstrate similar performance improvement.

This chapter presents the following contributions:

- We propose P-INSPECT, the first hardware architecture to accelerate persistence by reachability NVM frameworks in a transparent manner.

- We develop hardware designs to minimize persistence checks in software and to speed up persistent writes.

- We evaluate the proposed hardware on a state-of-the-art persistence by reachability NVM framework, and demonstrate its improved performance.

4.2 BACKGROUND: USING NVM

Programming an NVM system is challenging [134, 144]. To use NVM, programming frameworks must offer users an appropriate interface. For instance, the Storage Networking Industry Association (SNIA) has created a low-level programming model for NVM developers [137]. Intel has produced a tool set that is compatible with the SNIA model, which is called Persistent Memory Development Kit (PMDK) [138], and is a collection of libraries in C/C++ and Java. In addition to the SNIA and Intel efforts, there are many other NVM programming frameworks [11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 23, 24, 25, 135, 136]. Their goal is to facilitate NVM programming and ease the adoption of NVM.

The two main differentiating features of these frameworks are: (i) how they identify which data objects to place in NVM, and (ii) what persistence abstractions they provide. Many frameworks [11, 12, 13, 14, 15, 16, 17, 18, 20, 138] expect the user to explicitly mark all the objects that must be placed in NVM. Such a limited abstraction places a heavy burden on the programmer and creates many opportunities for bugs [144, 145, 146]. Furthermore, these frameworks cannot be directly used with existing codes; they require applications to be rewritten to include the necessary markings.

In contrast, several new frameworks [23, 24, 25] use a more programmer-friendly model known as *Persistence by Reachability*. In these frameworks, the runtime automatically ensures that, given a set of entry points into the persistent data structures in the program, all the data in the program reachable from such entry points is in NVM. This support significantly reduces the programmer burden and allows for the reuse of existing code.

Frameworks also differ in the persistence abstractions provided. Frameworks supporting epochs [16, 17, 18, 19, 21] couple logging regions to epochs (e.g., critical sections), and stores only need to be persisted at the end of the epochs. Other frameworks allow the logging regions to be specified by the programmer, and are not tied to a programming construct [11, 12, 13, 14, 20, 23, 24, 25]

Another difference between frameworks is whether the user is expected to explicitly identify the persistent stores in the program. Some frameworks expect the user to mark persistent stores manually [11, 12, 14, 15, 16, 17, 18, 19, 138], introducing many opportunities for mistakes. However, others leverage both compile time and runtime techniques [13, 20, 21, 23, 24, 25, 135, 136] to obviate the need for the programmer to label persistent stores. As persistence by reachability NVM frameworks already leverage runtime techniques, they free the programmer from having to label persistent stores.

4.3 PERSISTENCE BY REACHABILITY

4.3.1 Benefits of the Reachability Abstraction

In *Persistence by Reachability* [23, 24, 25], the programmer is only tasked with identifying the few entry points into the data structures that should be made persistent. These are called *durable roots*, and could be, e.g., the dominator pointer to a graph structure, or the root node of a tree. The framework is responsible to ensure that, dynamically, all objects reachable from a durable root (i.e., the durable root's *transitive closure*) reside in NVM.

This model is attractive because it puts a minimal burden on programmers. Programmers only have to think about which data structures must be in NVM, instead of explicitly identifying all the objects that must be in NVM. Throughout execution, data structures are modified and the durable transitive closure changes. Therefore, the framework monitors program writes and moves objects to NVM as needed.

These frameworks have three other advantages. First, they simplify programming by automatically inserting the necessary CLWBs, sfences, and logging operations. Second, they are compatible with existing code, in that neither the code must be rewritten to identify the persistent state nor a different library must be used. Finally, by dynamically placing data in NVM only when needed, they save power and latency.

4.3.2 Reacting to Updates to the Durable Transitive Closure

At runtime, these frameworks guarantee that the durable root set's transitive closure always resides in NVM. This requires objects to be moved to NVM as they become reachable from a durable root. To understand the operations needed, Figure 4.1(a) shows a heap containing objects A, B, and C in DRAM, and E and F in NVM. F is a durable root. An arrow from object i to j means that a field in i has a reference to j. Each object has a header state with 2 bits: the *Forwarding* and *Queued* bits. Their functionality is explained below.

Suppose that F is updated to point to A instead of E. To ensure that the durable transitive closure is completely within NVM, before performing the write that makes F point to A, the framework must transparently move both A and its transitive closure (i.e., the objects pointed by A, recursively) to NVM. This is done iteratively, using a worklist of objects starting with A. For each object *obj* in the worklist (e.g., A), the framework does:

1. Create a copy of *obj* in NVM with a set Queued bit.
2. Update the original *obj* by setting its Forwarding bit and making it point to the object's

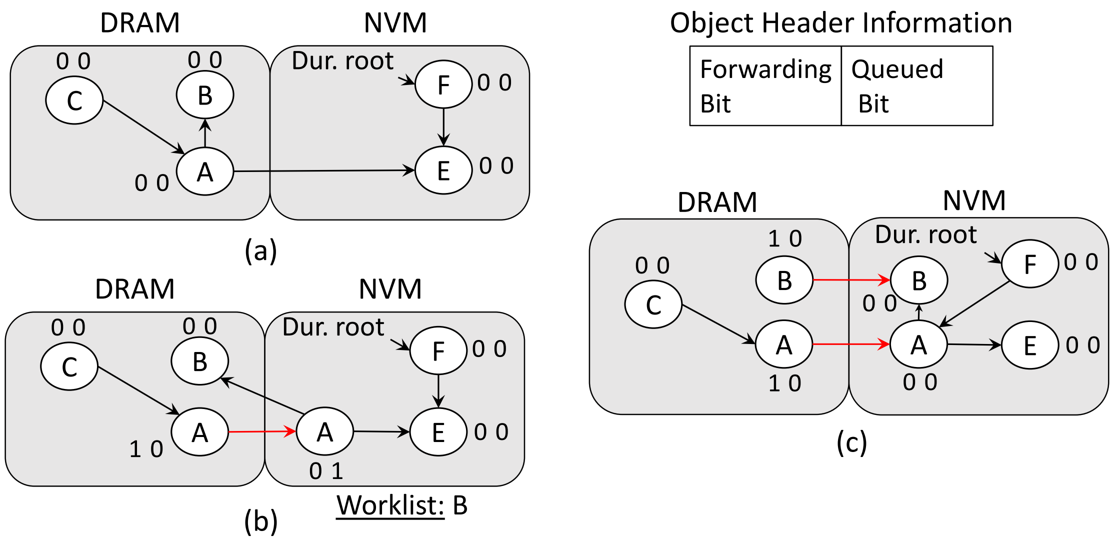


Figure 4.1: Persistence by reachability example.

new location.

3. Search *obj*'s fields for references to other objects to be added to the worklist.

In the first step, the object is copied to NVM with a Queued bit set to indicate that the object's transitive closure is still being processed. All the objects in the transitive closure being moved will set their Queued bit, which will only get reset once there are no more objects within the worklist to move to NVM.

Setting the Queued bit is necessary for correct execution in multithreaded environments. It ensures that another thread does not prematurely update a field somewhere to point to an object copied to NVM before the object's transitive closure is made durable. Hence, any write must check the Queued bit and wait to perform until the bit is reset.

Step two repurposes the original object to act as a *Forwarding* object pointing to the object's new NVM location. Forwarding objects are essential to allow pointers in DRAM to be lazily updated. The alternative would be to stop the execution of all the threads and update all the pointers that point to the old object in DRAM to point to the new object in NVM. This is eschewed by persistence by reachability frameworks because it would have prohibitive performance overheads. Instead, when accessing an object, the framework first checks whether the object is a forwarding object by checking its Forwarding bit. If it is, the NVM location of the object is accessed. Note that forwarding objects are always in DRAM and point to NVM.

The third step is to search for objects that need to be added to the worklist. Specifically, the framework checks all of the objects pointed to by *obj* and sees whether they are already in NVM. If not, they are added to the worklist. It is necessary to move these objects to NVM because they are now reachable from the durable root set. For each of the objects moved to the worklist, steps one to three are repeated, iteratively.

Figure 4.1 (b) shows the state of the heap after A has been processed. The original A is now a forwarding object to A in NVM, and the latter has the Queued bit set. Also, B is added to the worklist since it is pointed to by A.

Figure 4.1 (c) shows the final state after the object movement completed. F points to A in NVM, there are two forwarding objects, and all Queued bits are clear. Forwarding objects are only temporary; during garbage collection, this level of indirection is removed and forwarding objects are deallocated.

Finally, the framework also automatically inserts any necessary instructions before and after the write. They may be CLWB and sfence after the write and, if execution is within a transaction, a logging write (plus its CLWB and sfence) before the write.

4.3.3 Required Software Checks

A persistence by reachability framework needs to include checks around loads and stores. First, if a persistent object is being updated to point to an object in DRAM, before the write takes place, the actions in Section 4.3.2 are performed. Determining whether an object is in the NVM or DRAM heap requires a check on the object's virtual address.

Moreover, before any read or write to an object, one needs to determine whether the object is a forwarding one (i.e., its Forwarding bit is set). If it is, its forwarding pointer must be followed to retrieve the object's true location in NVM. Note that, if the object is in NVM, it cannot be a forwarding one.

Furthermore, before an object can be pointed to by a durable object, one needs to determine whether the object is being processed by another thread to become part of the durable root's transitive closure. This check is required to avoid the inconsistency of a durable object pointing to an object that is not yet part of the durable set. Hence, the software checks if the object is in NVM and has its Queued bit set. If both are true, the object's transitive closure is being processed by another thread, and pointing to the object must be delayed until the Queued bit is cleared.

More explanation can be found in the description of the AutoPersist persistence by reachability NVM framework [23].

4.4 P-INSPECT: MAIN IDEA

In persistence by reachability frameworks [23, 24, 25], all the software checks described, plus some runtime decisions based on the results of these checks [139], and the insertion of CLWB, sfence, and logging operations have a large performance impact. We will show in Section 4.9 that they contribute with 22–52% of the instructions in a set of workloads. This is a significant price to pay for the programmability afforded by persistence by reachability. Importantly, most of the checks turn out to require no action. This is because finding objects that are forwarding or whose transitive closure is being moved to NVM is *not the common case*. In this chapter, we propose hardware support to eliminate most of this overhead.

We call our scheme *Persist-Inspect*, or P-INSPECT. It focuses mostly on reducing the overhead of the checks performed before read and write accesses. It also reduces the overhead of the CLWB and sfence instructions that are added to persistent writes. We consider each case in turn.

4.4.1 Minimizing the Overheads of Checks

In P-INSPECT, we envision every application read and write to check certain state in hardware. If the check resolves that no special action is needed—recall that this is the common case by far—the read or write completes normally. Otherwise, the hardware automatically invokes a software handler, which performs some checks and extra operations, and also performs the read or write.

To understand the types of checks performed by the hardware, consider the most general operation, where a field in an object (call it *Holder* object) is set to point to another object (call it *Value* object). We represent this as $\text{Obj}_H.\text{field} = \text{Obj}_V$.

Based on the discussion in Section 4.3, we propose to perform the four checks shown in Table 4.1 in hardware, rather than in software as in current systems. The first check is whether the holder and/or the value objects are allocated in NVM or DRAM. If the code is attempting to have an NVM holder object point to a DRAM value object, a software handler will be invoked. The software will copy the value object and its transitive closure to NVM before performing the write.

The second check is whether the holder and/or the value objects are forwarding objects. If any is, the software will be invoked. It will follow the forwarding link(s) to obtain the correct object(s) before performing the read or write.

The third check is whether the transitive closure of the value object is currently being processed by another thread. Recall from Section 4.3.3 that this check is required to avoid

HW Check	HW Needed
Holder and/or value objects in NVM or DRAM?	Virtual addresses
Holder and/or value objects are forwarding?	Bloom filter
Is the value object's transitive closure being processed?	Bloom filter
Is execution inside a Xaction?	Register bit

Table 4.1: Hardware checks performed by P-inspect.

the inconsistency of a durable holder object pointing to a value object that is not yet part of the durable set. Depending on the conditions, the software will be invoked. The software will wait until the transitive closure is completely processed before pointing to the value object.

The final check is whether the execution is inside a transaction (Xaction). If so, the software may need to perform a logging operation before the write.

Table 4.1 also shows the simple hardware that P-inspect uses to perform these checks quickly. Specifically, whether the objects reside in NVM or DRAM can be determined by their virtual addresses. To detect whether an object is a forwarding one, P-inspect uses a bloom filter that keeps the addresses of all the current forwarding objects. Similarly, whether the transitive closure of an object is being processed is detected by accessing another bloom filter that keeps the addresses of all such current objects. Finally, whether the execution is inside a Xaction is determined by a register bit that is automatically set and cleared in hardware when a Xaction starts and ends, respectively.

With this hardware support, P-inspect minimizes the checking overheads of persistence by reachability. Specifically, write and read instructions automatically trigger the checking hardware described. In the common case when the hardware determines that no special action is needed, the write or read is performed at high speed.

In the uncommon case when the hardware determines that a special action is needed, a software handler is automatically invoked. The handler reads information from the header of the software object structures to determine what actions to take before performing the access— i.e., copy objects, follow forwarding pointers, wait until a transitive closure is fully processed, or log a value inside a Xaction.

4.4.2 Minimizing Persistent Write Overheads

A second overhead in practically all NVM frameworks is the fact that performing a persistent write typically requires executing a CLWB and, depending on the persistency model, an sfence (Section 4.2).

P-INSPECT minimizes this overhead by supporting one type of persistent write operation that combines the write, CLWB, and sfence. Such operation interacts efficiently with the cache coherence protocol. It can be automatically invoked when the hardware performs the write, and can also be invoked by the programmer.

4.5 P-INSPECT DESIGN

4.5.1 Bloom Filter Support

As indicated in Section 4.3, *Forwarding* objects are a key component of a high-speed persistence by reachability NVM framework. When an object A needs to be moved from DRAM to NVM, setting-up a forwarding object induces only modest overhead on the critical path of the application. The alternative is: (i) block all the other threads of the application, (ii) traverse the heap to find all the pointers to object A and to its transitive closure [25], and (iii) update these pointers to point to new objects now allocated in NVM. These actions are on the critical path of the application.

Unfortunately, in frameworks with forwarding objects, on an access to an object, one needs to check the Forwarding bit of the object and, if set, follow a pointer to access the correct object in NVM. In addition, on an access to a value object, one needs to check its Queued bit, and only proceed with the access when the bit is clear.

These checks are on the critical path. However, typically, the bits are clear. Hence, P-INSPECT introduces two bloom filters that quickly return whether an object's Forwarding or Queued bits are set. These filters are called Forwarding (*FWD*) and Transitive Closure (*TRANS*), respectively.

FWD Bloom Filter. Immediately before the runtime sets up a forwarding object in DRAM, the runtime inserts the base address of the object in the FWD bloom filter. Later, on a read or write to an object, the hardware searches the FWD filter and determines whether the object is a forwarding one.

When the FWD bloom filter fills-up to a certain threshold, the hardware wakes up a *Pointer Update Thread* (PUT). The PUT traverses all live objects of the volatile heap. When it identifies a pointer to a forwarding object, it updates the pointer to point to the corresponding NVM object. When the PUT has traversed all the objects, it clears the FWD bloom filter in bulk. The now inaccessible forwarding objects will later be reclaimed by the garbage collector.

Section 4.6 discusses the implementation of the FWD filter so that the PUT can operate in the background without stalling program threads or losing any filter information.

TRANS Bloom Filter. Immediately before a value object in the worklist for a transitive closure being processed is moved by the runtime to the NVM, the runtime inserts the base address of the object in the TRANS filter. Later, on an access to a value object, the hardware searches the TRANS filter and determines whether the object is a queued one.

As soon as the thread that is processing the transitive closure sets-up forwarding objects for all the objects in the transitive closure, it clears the TRANS bloom filter in bulk.

4.5.2 New Operations

To access the bloom filters, P-INSPECT introduces the new operations shown in Table 4.2. Six of them operate as store instructions and one as a load. They all take at most a memory address and a register as arguments. A possible implementation in x86 could use existing store and load opcodes preceded by a prefix. In the table, *Ha* is the address of a field in a holder object, *Va* is the address of the base of a value object, and *Addr* is the address of the base of an object.

Name	What it Does
checkStoreBoth [Ha],Va	Performs checks, then Mem[Ha] = Va
checkStoreH [Ha],value	Performs checks, then Mem[Ha] = value
checkLoad [Ha],dest	Performs checks, then dest = Mem[Ha]
insertBF$_{\text{FWD}}$ Addr	Inserts Addr in the FWD bloom filter
insertBF$_{\text{TRANS}}$ Addr	Inserts Addr in the TRANS bloom filter
clearBF$_{\text{FWD}}$	Clears the FWD bloom filter
clearBF$_{\text{TRANS}}$	Clears the TRANS bloom filter

Table 4.2: New operations. In the table, *Ha* is the address of a field in a holder object, *Va* is the address of the base of a value object, and *Addr* is the address of the base of an object.

checkStoreBoth [Ha],Va performs some hardware checks detailed in Section 4.5.3 and, if successful, stores the base address of the value object into a field of the holder object. *checkStoreH [Ha],value* also performs hardware checks and, if successful, stores the value into a field of the holder object. *checkLoad [Ha],dest* performs hardware checks and, if successful, loads the contents of a field of the holder object into a destination register. *insertBF$_{FWD}$ Addr* and *insertBF$_{TRANS}$ Addr* insert the base address of an object (Addr) into the FWD and TRANS bloom filter, respectively. *clearBF$_{FWD}$* and *clearBF$_{TRANS}$* clear the FWD and

TRANS bloom filter, respectively.

In the first three operations, if the hardware checks are unsuccessful, the write/read is not performed and a software handler is invoked. We now examine the checks and the software handlers.

4.5.3 Hardware Checks Before Writes/Reads

Table 4.3 shows the hardware checks performed by the *checkStoreBoth*, *checkStoreH* and *checkLoad* operations.

Hardware Check	Operation		
	CSB	CSH	CL
Is Base(Ha) in NVM or DRAM?	✓	✓	✓
Is Va in NVM or DRAM?	✓		
Is Base(Ha) in the FWD bloom filter?	✓	✓	✓
Is Va in the FWD bloom filter?	✓		
Is Va in the TRANS bloom filter?	✓		
Is execution inside a Xaction?	✓	✓	

Table 4.3: Checks performed by the checkStoreBoth (CSB), checkStoreH (CSH), and checkLoad (CL) operations.

checkStoreBoth (CSB). This operation checks the most conditions. The first two conditions are whether the base addresses of the two objects accessed (i.e., Base(Ha) and Va) are in NVM or in DRAM. This information is attained by examining the objects' virtual addresses, which tell whether the objects are in NVM or DRAM. The Base(Ha) is directly obtained from the instruction, which contains base plus offset for Ha. The next two conditions are whether the two objects are forwarding objects. This is obtained by hashing Base(Ha) and Va for membership in the FWD bloom filter. The next condition is whether the value object is currently being processed as part of a transitive closure worklist. This information is obtained by hashing Va for membership in the TRANS bloom filter. Finally, the last check is whether the execution is inside a transaction. This is obtained by reading a register bit.

Given all these checks, there are three cases when the hardware can complete the checkStoreBoth operation by performing the write without invoking the software. These three cases are shown in the top three rows of Table 4.4.

The first row is the case when both objects are in NVM, the value object is not in the

Conditions						Action Taken
Where is Base(Ha)?	Base(Ha) in FWD?	Where is Va?	Va in FWD?	Va in TRANS?	In Xaction?	
NVM	-	NVM	-	false	false	HW
DRAM	false	DRAM	false	-	-	HW
DRAM	false	NVM	-	-	-	HW
DRAM	true / -	- / DRAM	- / true	-	-	SW: ①
NVM	-	DRAM / NVM	-	- / true	-	SW: ②
NVM	-	NVM	-	false	true	SW: ③

Table 4.4: Execution flows for stores. Empty boxes (-) can take any values.

TRANS bloom filter, and execution is not inside a Xaction. Since the value object is not in the TRANS bloom filter, there is no need to wait. Further, since execution is not in a Xaction, there is no need to log. Hence, checkStoreBoth performs a read and a persistent write.

The second row is when both objects are in DRAM and not in the FWD bloom filter. They are volatile, non-forwarding objects. checkStoreBoth simply performs a read and a non-persistent write to the holder. There is no need to wait for any transitive closure or perform any logging.

The third row is when the holder is a non-forwarding object in DRAM, and the value object is in NVM. Since having a pointer from DRAM to NVM is always fine, checkStoreBoth simply performs a read and a non-persistent write to the holder.

checkStoreH (CSH). This operation is like checkStoreBoth, except that it does not read from a value object. Instead, it reads from a primitive type like an integer. Hence, checkStoreH does not check anything related to any value object Va (Table 4.3).

As a result, checkStoreH can perform the operation in hardware under the same three cases as checkStoreBoth (Table 4.4)— except that there is no check for Va conditions. Specifically, checkStoreH performs the read and write and completes if (i) the holder is in NVM and execution is not in a Xaction (first row) or (ii) the holder is a non-forwarding object in DRAM (second and third rows).

checkLoad (CL). Since checkLoad is a read of the holder object, we only need to ensure that the read gets the correct address. Hence the only checks performed are whether the base of Ha is in NVM or DRAM, and whether the object is a forwarding one (Table 4.3). There is no value object and no logging is ever needed.

Given these checks, there are two cases when checkLoad can perform the read and complete. They are the top two rows of Table 4.5. One is when the object is in NVM; the other is when it is in DRAM and is not in the FWD bloom filter.

Conditions		Action
Where is Base(Ha)?	Base(Ha) in FWD?	Taken
NVM	-	HW
DRAM	false	HW
DRAM	true	SW: ④

Table 4.5: Execution flows for loads.

4.5.4 Software Handlers

In all the other conditions not covered in Section 4.5.3, the hardware does not perform the read or write. Instead, it invokes one of the following 4 software handlers.

Handlers for Writes. As can be seen in Table 4.4, there are three possible cases when checkStoreBoth or checkStoreH invoke a software handler.

The first case, shown in Row 4, invokes handler ① when: (i) the holder object is in DRAM, and (ii) the value or holder objects are in the FWD bloom filter (i.e., either one or both). In this case, the software needs to check if indeed these objects are forwarding and, if so, follow the forwarding pointer(s) to get to the correct object(s). Recall that a bloom filter can produce false positives (but never false negatives). Hence, to be certain that the object(s) are forwarding ones, the software needs to check the actual Forwarding bits in the headers of the object(s)' software structures.

Further, before performing the write, the software will have to wait until the completion of any in-progress transitive closure processing that includes the value object, and will have to create a log entry if execution is in a Xaction. After that, the software may perform a persistent or a regular write.

We refer to handler ① as *checkHandV*, and show it in Algorithm 4.1. In Lines 2 and 4, it checks the Forwarding bits of the holder and value objects, respectively, and follows the forwarding links, if needed.

Then, *CheckHandV* determines if the holder object is persistent (i.e., it is in NVM or is Forwarding) (Line 6). If it is not, the algorithm performs a non-persistent write (Line 21). Otherwise, we need to check if the value object is persistent (Line 7) and, if it is not, make it persistent (Line 10). It will not be persistent if either its virtual address is not in NVM or it is part of an in-progress transitive closure processing. The latter condition is indicated

Algorithm 4.1: Software handlers.

```
Function ① checkHandV(Ha, Va, Xaction):
    Read H header & update Ha if needed // forwarding case;
    if isObject(Va) then
        Read V header & update Va if needed // forwarding case;
    end
    if isPersistent(H) then
        if !isPersistent(V) then
            // not in NVM or the Queued bit is set;
            // may wait until Queued is cleared;
            makeRecoverable(V);
        end
        if Xaction then
            Write to log // includes a CLWB and sfence;
            persistentWrite [Ha], Va // persistent program store;
            // it includes CLWB but not sfence;
        else
            persistentWrite [Ha], Va // persistent program store;
            // it includes CLWB and possibly also sfence;
        end
    else
        store [Ha],Va // non-persistent program store
    end
End Function
Function ② checkV(Ha, Va, Xaction):
    Read V header & update Va if needed // forwarding case;
    if !isPersistent(V) then
        // not in NVM or the Queued bit is set;
        // may wait until Queued is cleared;
        makeRecoverable(V);
    end
    if Xaction then
        Write to log // includes a CLWB and sfence;
        persistentWrite [Ha], Va // persistent program store;
        // it includes CLWB but not sfence;
    else
        persistentWrite [Ha], Va // persistent program store;
        // it includes CLWB and possibly also sfence;
    end
End Function
Function ③ logStore(Ha, Va, Xaction):
    Write to log // includes a CLWB and sfence;
    persistentWrite [Ha], Va // persistent program store;
    // it includes CLWB but not sfence;
End Function
Function ④ loadCheck(Ha):
    Read H header & update Ha if needed // forwarding case;
    load [Ha];
End Function
```

with the Queued bit set in the header of V. The *makeRecoverable* function (Line 10) will make it persistent.

After that, if we are in a Xaction, the software creates a log entry and performs a write to NVM. The instruction used, called *persistentWrite*, is discussed in Sec. 4.5.5. This flavor of persistentWrite includes a CLWB but not an sfence, since we are inside a Xaction; we will need an sfence at the end of the Xaction. If we are not in a Xaction, the software performs a write to NVM. In this case, we use a persistentWrite flavor that includes a CLWB and also possibly an sfence.

The second case when a software handler needs to be invoked is shown in Row 5 of Table 4.4. In this case, we invoke handler ② when: (i) the holder object is in NVM, and (ii) the value object is in DRAM (may or may not be a forwarding object), or in NVM and its Queued bit is set (i.e., it is part of an in-progress transitive closure processing). Only checkStoreBoth can invoke it.

We refer to handler ② as *checkV* in Algorithm 4.1. The flow is like *CheckHandV* except that no check needs to be performed on the holder object. This is because the object is in NVM.

Finally, the third case when a software handler needs to be invoked is Row 6 of Table 4.4. This case invokes handler ③ when: (i) both holder and value objects are in NVM, (ii) the value object's Queued bit is clear, and (iii) we are in a Xaction.

This is a simple case. Handler ③, which we call *logStore*, is shown in Algorithm 4.1. The handler creates a log entry and performs a store to NVM. Again, this write includes a CLWB but no sfence.

Handlers for Reads. As can be seen in Table 4.5, there is one case when checkLoad needs to invoke a software handler. It invokes handler ④ when the holder object is in DRAM and in the FWD bloom filter. This means that the object may be forwarding. We refer to handler ④ as *loadCheck* in Algorithm 4.1. The software checks the Forwarding bit in the object's header and, if set, follows the forwarding link (Line 46). Then, it reads the field.

4.5.5 Low-Overhead Persistent Write

A write to NVM can be expensive, since it is often followed by a CLWB, and sometimes even by an sfence. Consider the worst case when all three operations need to take place (Figure 4.2(a)). First, the write itself may have to bring the line from main memory, as it loads it into an L1 cache in Dirty state (Steps ① to ④). The CLWB then needs to find a copy of the line—which is likely to be in the L1 cache but may be in any cache in any state, if the line has been accessed by other cores since the above write. Once the line is found in a cache, it is written to main memory and a copy is kept in that cache (Steps ⑤ to ⑥). Once the acknowledgment of the CLWB completion reaches the originating core (Steps ⑦ to ⑧), the sfence retires, allowing a subsequent write to be merged with the memory system. In the worst case, the combined operations may require two round trips to memory.

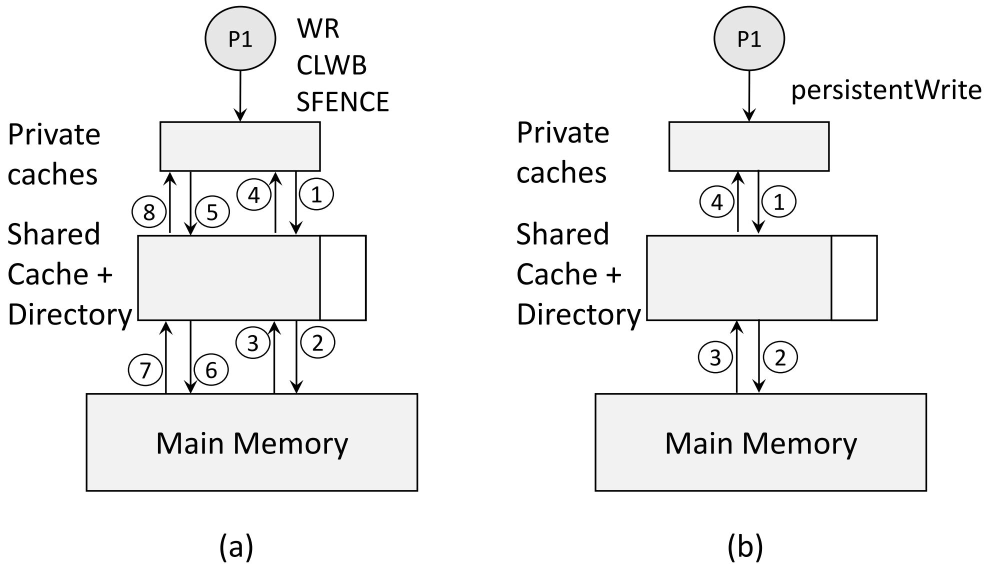


Figure 4.2: Conventional persistent write, CLWB, and sfence (a), and proposed advanced persistentWrite flavor (b).

We propose a new instruction called *persistentWrite* that speeds-up a write to NVM. *persistentWrite* has three flavors: one that simply performs a write; one that combines a write with a CLWB; and one that combines a write with a CLWB and an sfence. The flavor used depends on whether the write needs to be followed by CLWB and sfence operations. In our discussion, we assume that the NVM can be written at a granularity finer than a cache line [147, 148, 149, 150, 151, 152].

The first flavor is a simple write. The third flavor is the one that improves performance the most over the state of the art. The three operations (write, CLWB, and sfence) are performed with at most one single round trip to memory (Figure 4.2(b)). Specifically, the originating core's persistentWrite operation first sends the update down the cache hierarchy (Step ①). If the transaction finds a copy of the line, it is incorporated in the message. Once the message reaches the directory, the directory is locked. If the directory indicates that the line is in state Exclusive/Dirty in another cache, that line is recalled and the owner cache is invalidated. In any case, all cached copies of the line are invalidated (except in the cache of the originating core, if it is there). Finally, the update—combined with the corresponding cache line if the line was dirty in the cache hierarchy—is sent to NVM to persist (Step ②).

The NVM returns an acknowledgment, potentially with the updated line, to the directory (Step ③) and then to the originating core (Step ④). The directory marks that core as having the line in Exclusive state, and is unlocked. Once the core receives the acknowledgment, it allows a subsequent write to proceed. The result is at most a single round trip to memory.

The flavor that combines the write and the CLWB proceeds in a similar manner.

In all cases, persistent writes from different cores are not ordered unless they access the same cache line. In such case, the corresponding directory module serializes and orders them based on arrival order.

Finally, the checkStoreBoth and checkStoreH operations also have the three flavors described. The flavor used depends on whether the write, if it succeeds, needs to be followed by CLWB and sfence operations.

4.6 P-INSPECT IMPLEMENTATION ISSUES

4.6.1 Operation of the FWD Bloom Filter

As a program executes, threads insert the base addresses of forwarding objects into the FWD filter. Recall from Section 4.5.1 that when the FWD filter fills to a certain threshold, the system wakes up the PUT thread. In the background, PUT updates any pointers to forwarding objects to point to the corresponding NVM objects, and clears the FWD filter.

To enable this operation, P-INSPECT uses two FWD bloom filters—call them red and black. They have one extra bit called *Active*, which is set for the single FWD filter that is currently being inserted to.

Suppose that the red FWD filter is currently the active one. When it fills up to the threshold, PUT wakes up and toggles the Active bit in both FWD filters. PUT then performs a sweep of the objects in the volatile heap, processing encountered pointers to forwarding objects as discussed above. During this time, since the black FWD filter is now the active one, all object insertions required by the program are performed on the black FWD filter. However, object lookups are performed on *both* FWD filters. If an object is a member of either of the two FWD filters it is considered a Forwarding object. When PUT finishes its traversal, it clears the red FWD filter and goes back to sleep.

With this approach, PUT execution happens in the background, without stalling program threads, and no filter information is ever lost. It is likely that PUT's execution also updates pointers to some forwarding objects that were inserted in the black FWD filter. Hence, the black FWD filter may now include some objects that are not forwarding anymore. This is no problem since, at worse, this effect increases the number of false positives in the FWD bloom filter.

4.6.2 Implementing the Bloom Filters

Each process has two FWD bloom filters, each with 2047 bits for the data and 1 bit (the most significant one) for the Active bit. Hence, a FWD filter covers 4 cache lines in a machine with 64-byte cache lines. The TRANS bloom filter only uses 512 bits, which is 1 cache line. Overall, the bloom filters for a process use 9 cache lines. For each filter, we use two hash functions, H_0 and H_1. Each process has all of its bloom filters in memory in a single page, at a fixed virtual address. The filters are accessible with virtual addresses.

The operations performed on the FWD bloom filters are shown in Table 4.6. The operations on the TRANS filter are fewer and simpler; we do not show them for simplicity. In an *Object Lookup*, we take the object's address and hash it using H_0 and H_1. Then, we read the 2 FWD bloom filters and check for membership in them.

Operation	What it Does
Object Lookup	Check both FWD filters for address membership
Object Insert	Insert address in the active FWD filter
Inactive FWD Filter Clear	Zero-out the inactive FWD filter
Change Active FWD Filter	Toggle the active bit in both FWD filters

Table 4.6: Operations performed on the FWD filters.

In an *Object Insert*, we hash the object's address using H_0 and H_1 as before. Then, we read the most-significant cache line of the two filters and identify which filter is the active one. Finally, we read the three remaining lines of the active filter and set the appropriate bits in the filter.

In an *Inactive FWD Filter Clear*, we read the most-significant line of the two filters and identify the inactive filter. Then, we read the three remaining lines of the inactive filter and clear the filter. Finally, in a *Change Active FWD Filter* operation, we read the most-significant line of the two filters, and toggle the active bit in both.

P-INSPECT implements these operations in hardware. Specifically, P-INSPECT has a *BFilter_FU* functional unit that helps execute the instructions of Table 4.2. This functional unit knows the address and layout of the page of bloom filters, and the hash functions H_0 and H_1. Given an address, it can perform H_0 and H_1 on it, and access the filters.

The left part of Figure 4.3 shows the P-INSPECT hardware in the core. In addition to the *BFilter_FU* functional unit with the two hash functions, there is a bit that indicates if execution is inside a transaction, and the virtual addresses for: the bloom filter page, the software handlers, and the base and limit of the persistent heap.

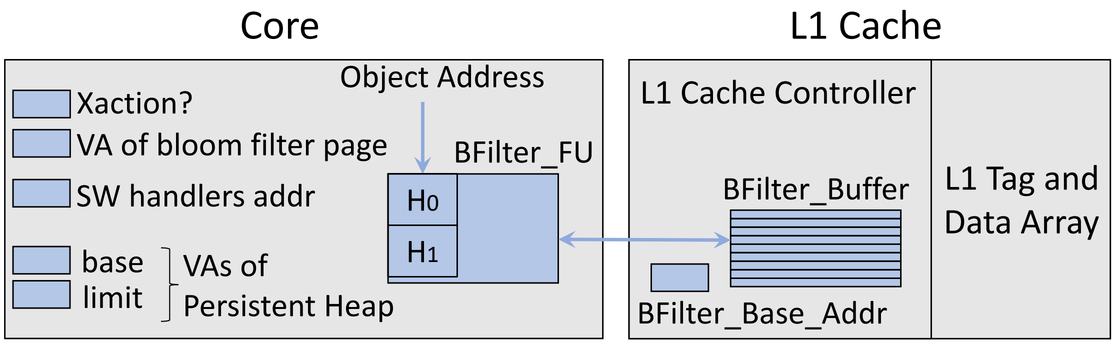


Figure 4.3: Implementation of the P-INSPECT architecture.

The operations in Table 4.6 are implemented as follows. An Object Lookup is performed in hardware, as part of the checkStoreBoth, checkStoreH, and checkLoad operations. The lookup uses the BFilter_FU functional unit, and is fully overlapped with the store or load.

An Object Insert is performed by the runtime executing the insertBF$_{\text{FWD}}$ or insertBF$_{\text{TRANS}}$ operations in Table 4.2 for the FWD or TRANS filter, respectively. An Inactive FWD Filter Clear is performed by the PUT thread executing the clearBF$_{\text{FWD}}$ operation in Table 4.2 when it has completed its operations on the volatile heap. Similarly, the TRANS filter is cleared by a thread executing the clearBF$_{\text{TRANS}}$ operation in Table 4.2 when it has completed processing a transitive closure. Finally, the Change Active FWD Filter operation is performed by the PUT thread when it wakes up.

4.6.3 Keeping Bloom Filter Data Coherent

In a multithreaded program running on a multiprocessor, we need to ensure the coherence of the bloom filter data across cores. To understand the problem, recall that while Object Lookup is a read-only operation, the other three operations (Object Insert, Inactive FWD Filter Clear, and Change Active FWD Filter) are read-write operations that need to be performed atomically. Note that insertBF$_{\text{TRANS}}$ and clearBF$_{\text{TRANS}}$ also include write operations. However, as we will see, the read-only Object Lookup operation is on average over one million times more frequent than the operations that involve writes. We want to keep the former fast.

To keep bloom filter data coherent, a simple approach is to use the protection bits in the TLB for the bloom filter page. Specifically, when a thread wants to perform a read-write operation, it sets the owner bit in its TLB entry and invalidates the TLB entry from the TLBs of the other cores. Unfortunately, this approach is too slow, as it involves the OS.

A faster approach, used by P-INSPECT, is to use the cache coherence protocol to maintain the coherence of bloom filter data. To understand the approach, assume a single bloom filter that spans a single cache line. When performing an object lookup, the hardware requests the cache line in Shared state. When performing the other operations, the hardware requests the line in Exclusive state. In this case, when the cache obtains the line, the cache refuses any incoming transaction to the line (i.e., it locks it) until all the local reads and writes to the bloom filter are done. Then, it unlocks the line.

A difficulty with this approach is that the bloom filters in P-INSPECT span 9 contiguous cache lines. In effect, we would like the coherence protocol to operate on these lines as if they were "glued" together—i.e., all 9 lines should be fetched at a time, and all 9 kept in the cache at a time.

To solve this problem, P-INSPECT augments the L1 cache controller with a register that contains the address of the bloom filter lines used by the currently-executing process. Since these 9 lines are contiguous, the register only keeps the base address. We call it the *BFilter_Base_Addr* register. In addition, the controller has a small buffer that has space for the 9 lines of bloom filter data. We call this buffer the *BFilter_Buffer*, and its lines are visible to the cache coherence protocol. These components are shown in the right part of Figure 4.3.

In an Object Lookup operation, as the BFilter_FU functional unit requests the bloom filter lines, the cache controller attempts to read all 9 lines into the BFilter_Buffer in Shared state. If, as it reads lines, some get invalidated by writes from other cores, it retries the read. When all 9 lines are obtained, they are read by the BFilter_FU.

In the bloom-filter operations that involve read-write accesses, we use the most significant line of the red FWD bloom filter as the *Seed*. This means that, as the BFilter_FU functional unit requests the bloom filter lines in Exclusive state, the cache controller attempts to obtain the Seed cache line in Exclusive state *first*. Once it attains it, it locks it in the BFilter_Buffer and proceeds to obtain the remaining lines in the BFilter_Buffer in Exclusive state. These lines are also locked. Once all the lines are present, they are read by the BFilter_FU, which performs the read-write operations. After that, the lines are unlocked and accept external requests. Since obtaining the Seed cache line in Exclusive state serializes all the other operations, there is no data incoherence or deadlock.

With this design, at a context switch, the OS simply writes back the dirty lines in the BFilter_Buffer, invalidates the BFilter_Buffer lines, and updates the BFilter_Base_Addr register with the base address of the bloom filters for the new process. The lines with the bloom filter data of the new process will be brought into the BFilter_Buffer on demand.

4.7 RELATION TO FAILURE RECOVERY

If an NVM system is to recover from failures, it needs software that performs operations such as undo/redo logging or checkpointing. Such software has to be aware of the memory persistency model [104] used by the system, since the memory persistency model determines when, after persistent objects are written, will the updates reach NVM.

A persistence by reachability framework such as P-INSPECT does not impact failure recovery. The framework's goal is to simplify the programmer's job by performing two main tasks automatically: (i) ensure that objects that need to be persistent are moved from volatile memory to NVM at the right time, and (ii) ensure that the updates to such objects are marked as being persistent—and therefore are accompanied by the correct CLWB and sfence instructions. The actual CLWB and sfence instructions added with the updates depend on the memory persistency model used by the system. Hence, the persistence by reachability framework is cognizant of the persistency model used; at no point, however, it affects or needs to know about the failure recovery algorithms.

A related issue is that prior literature has proposed per-core buffers that buffer stores destined for NVM [105]. Then, there is hardware that optimizes the write back of these stores to NVM to improve performance. One concern in this environment is to honor all inter-thread persistence dependencies. Once again, these hardware optimizations are orthogonal to the persistence by reachability framework. The latter simply ensures that the updates to the buffers are performed transparently to the programmer, efficiently, and following the memory persistency model used.

4.8 EVALUATION METHODOLOGY

Modeled Architecture and Infrastructure. We use cycle-level simulations to model a server architecture with 8 cores and a main memory of 32 GBs of NVM and 32 GBs of DRAM. The main architecture parameters are shown in Table 4.7. We integrate the Simics full-system simulator [60] with the SST [61] framework and the DRAMSim2 [63] memory simulator. To model NVM, we modified the DRAMSim2 timing parameters as shown in Table 4.7, and disabled refreshes. We use Intel SAE [66] on top of Simics for OS instrumentation, the Synopsys Design Compiler [153] to evaluate the RTL implementation of CRC hash functions, and CACTI [120] for the area and energy analysis of the hardware structures at 22nm.

For the runtime system, we use the most recent version of the AutoPersist framework [23]. AutoPersist is built within the Maxine Java Virtual Machine (JVM) [154]. In addition, to perform a lengthy analysis of the behavior of the architecture, we use Pin [155]. We augment the Java compiler and runtime to communicate the required information to our simulation infrastructure, both to Simics and Pin.

Processor Parameters	
Multicore chip	8 OoO cores, 2GHz, 2 issue (and 4 issue)
Ld-St queue; ROB	92 entries; 192 entries
Cache Line Size	64 bytes
DL1 cache	32KB, 8-way, 2-cycle access latency
L2 cache	256KB, 8-way, 8-cycle data, 2-cycle tag lat.
L3 cache	1MB/core, 16-way, 22-cycle data, 4-cycle tag latency
Cache coherence	MESI protocol
L1 TLB	64 entries, 4-way, 2-cycle latency
L2 TLB	1024 entries, 12-way, 10-cycle latency
Bloom Filter Parameters and Analysis	
Size	FWD: 2047 bits; TRANS: 512 bits
Hash function	CRC; 2-cycle latency; Area: $1.9 * 10^{-3} mm^2$; Dyn. energy: $0.98pJ$; Leak. power: $0.1mW$
Call to PUT	When 30% of FWD bloom filter bits are set
BFilter_Buffer	Area: $0.023mm^2$; Leakage power: $1.9mW$; Rd/Wr energy per access: $12.8/13.1pJ$ Lookup access overlaps with ld/st (2 cycles)
Main-Memory Parameters	
Channels; Banks	DRAM: 2; 8 NVM: 2; 8
t_{CAS}-t_{RCD}-t_{RAS}	DRAM: 11-11-28; NVM: 11-58-80
t_{RP},t_{WR}	DRAM: 11,12 NVM: 11,180
Freq; Bus width	1GHz DDR; 64 bits per channel
Host and Runtime System Parameters	
Host OS; Runtime	Ubuntu Server 16.04; Maxine JVM 2.0.5

Table 4.7: Architectural parameters used for evaluation.

Configurations. We compare four different designs.

Baseline: It uses the unmodified AutoPersist [23], a Java programming framework that provides persistence by reachability. AutoPersist performs all the runtime checks and object moves.

P-INSPECT--: AutoPersist plus our proposed P-INSPECT hardware to perform the required checks for loads and stores, but does not include the optimization for speeding-up persistent writes (Section 4.5.5).

P-INSPECT: AutoPersist plus the complete P-INSPECT design, including the optimization for persistent writes.

Ideal-R: AutoPersist without all the checks and object moves required to implement persistence by reachability. It is an ideal runtime where the user identified all persistent objects. It does not include the persistent write optimization.

Workloads. We do experiments on a key-value store using different backends, and also on several kernel applications.

Key-Value Store: We implement a persistent version of a key-value store by modifying QuickCached [156] to use the AutoPersist framework to persist its internal key-values. For our evaluation, we use four different, commonly used backends: (i) ***pTree*** uses a Java implementation of the IntelKV B+ tree [157] and persists both all inner and leaf nodes; (ii) ***HpTree*** uses the same data structure as pTree, but it is a hybrid design that only persists the leaf nodes of the tree, similar to IntelKV; (iii) ***hashmap*** uses a HashMap data structure for its internal storage; and (iv) ***pmap*** uses the Map implementation from the Java PCollections library [158].

To evaluate the performance of our key-value stores we use the Yahoo! Cloud Serving Benchmark (YCSB) [159]. This benchmark suite is commonly used for the evaluation of cloud storage services. We populate our key-value stores to a memory footprint of 12.5GB for pTree and HpTree, 12.4GB for hashmap, and 12.8GB for pmap. In our evaluation, we run three of the YCSB workloads: (i) the write-intensive workload A, (ii) the read-intensive workload B, and (iii) workload D, which is both read intensive and also inserts new values into the data structures.

Kernel Applications: These are several kernels that perform a collection of read, write, insert, and delete operations on persistent data structures. In total, we use six different kernel applications: (i) ***ArrayList*** is a persistent version of the Java ArrayList; (ii) ***ArrayListX*** is identical to the previous kernel, but uses transactions to perform in-place insertions and deletions; (iii) ***LinkedList*** is a doubly linked list implementation; (iv) ***HashMap*** is a HashMap implementation; (v) ***BTree*** is a B-tree implementation; and (vi) ***BPlusTree*** is a B+ tree implementation. We populate the kernel applications with one million elements before simulation.

Simulation Methodology. We perform two types of simulations. One is architectural simulations to evaluate performance using a cycle-level simulator. The other is behavioral simulations using Pin to characterize the behavior of applications and bloom filters over long execution intervals.

For the detailed architectural simulations, we perform full-system simulations. We warm-up the architectural state by running, per core, 200M instructions before simulating 1B instructions. For the behavioral simulations, we run hundreds of billions of instructions with Pin.

4.9 EVALUATION

4.9.1 Architectural Evaluation

The goal of P-INSPECT is to minimize the overheads of programmable NVM frameworks. P-INSPECT targets two main sources of overhead in such frameworks: (i) the runtime checks that need to be performed on the objects to determine their state, and (ii) the cost of the persistent write operations. In this section, we consider these overheads.

Figure 4.4 shows the instruction count of the kernels for the different configurations. The count is normalized to that of the baseline configuration. We see that P-INSPECT-- and P-INSPECT greatly reduce the number of instructions executed across the board. On average, they reduce the number of instructions by 46%. Kernels that have a larger number of stores like ArrayList attain higher instruction reductions than those that are more read-intensive like BTree. Recall that stores require more checks.

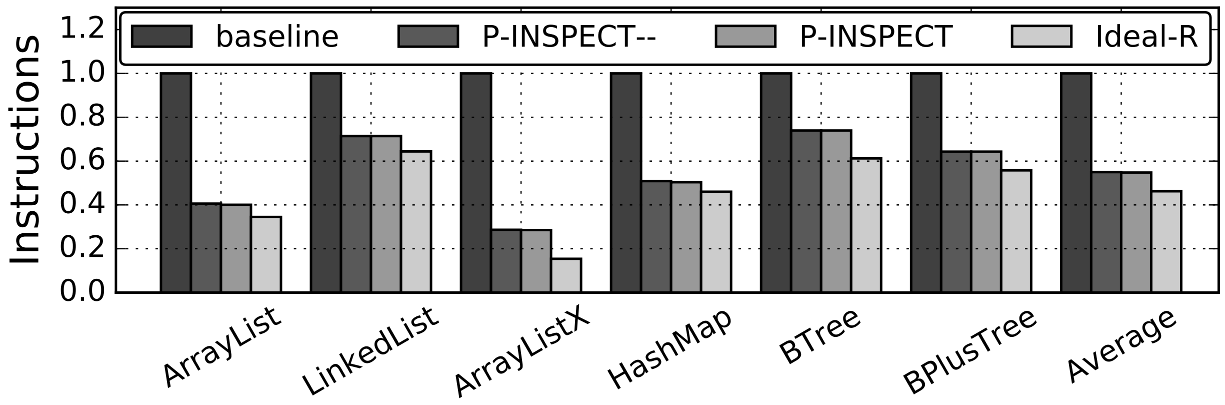


Figure 4.4: Instruction count of the kernel applications.

The figure also shows that P-INSPECT-- and P-INSPECT have approximately the same instruction count. P-INSPECT can reduce the number of executed instructions slightly more, because of the persistent write optimization that can combine the write, the clwb and the sfence into a single instruction. Further, they are fairly close to the Ideal-R configuration, which achieves an average 54% reduction.

Figure 4.5 shows the total execution time of the kernels. The figure is organized as Figure 4.4, except that we have broken down the baseline bar into time to perform: (i) the checks (baseline.ck), (ii) the persistent writes, approximately (baseline.wr), (iii) the persistency by reachability runtime operations such as logging and copying objects between DRAM and NVM (baseline.rn), and (iv) the rest (baseline.op). Baseline.op corresponds to a true ideal system with no persistency by reachability and (unlike Ideal-R) no NVM.

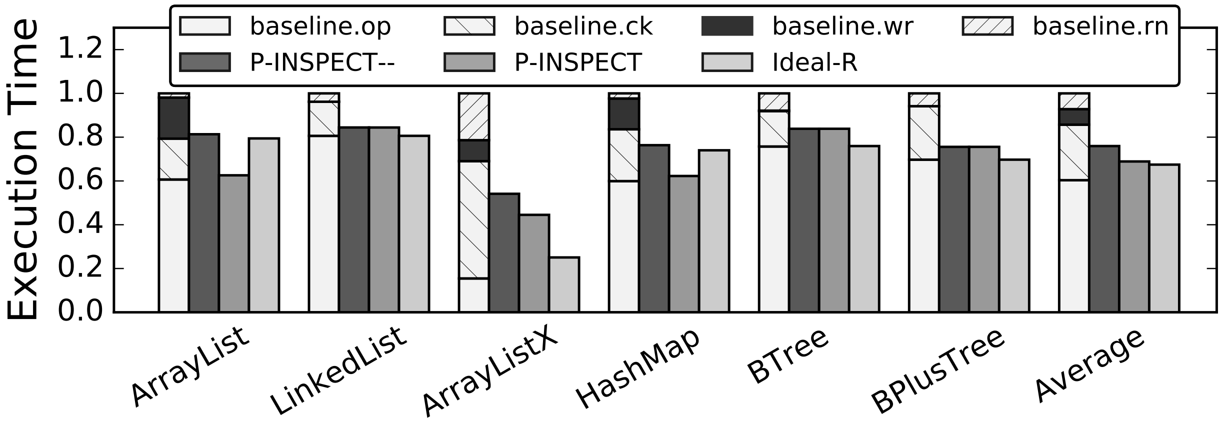


Figure 4.5: Execution time of the kernel applications.

We see that, in the baseline, the checking overhead is substantial, while the persistent write overhead is sometimes significant, and the runtime overhead is only significant when there is logging (ArrayListX). Both P-INSPECT-- and P-INSPECT deliver significant speed-ups. On average, they are 24% and 32% faster than baseline, respectively. The speed-ups, of course, are smaller than the instruction count reductions. However, the trends are largely similar.

We see a difference between P-INSPECT-- and P-INSPECT for some applications. Although these two configurations have similar instruction counts in Figure 4.4, P-INSPECT has a lower execution time for applications that have many persistent writes, like ArrayList and HashMap. This is especially the case when these writes miss in the cache hierarchy. In this case, thanks to our design, combining a write with a CLWB and an sfence has a substantial performance impact.

Finally, the average speed-up of P-INSPECT is very close to that of Ideal-R, which reduces execution time by 33%. In some cases, P-INSPECT is even faster than Ideal-R. The reason is that P-INSPECT includes the persistent write optimization of Section 4.5.5, while Ideal-R does not.

Figures 4.6 and 4.7 show the instruction count and the execution time, respectively, for the key-value stores. The figures are organized as the previous ones. In general, the trends for instruction reduction and execution time reduction are like for kernels, except that the improvements are relatively smaller. This is because these workloads perform relatively more non-memory access instructions than the kernels.

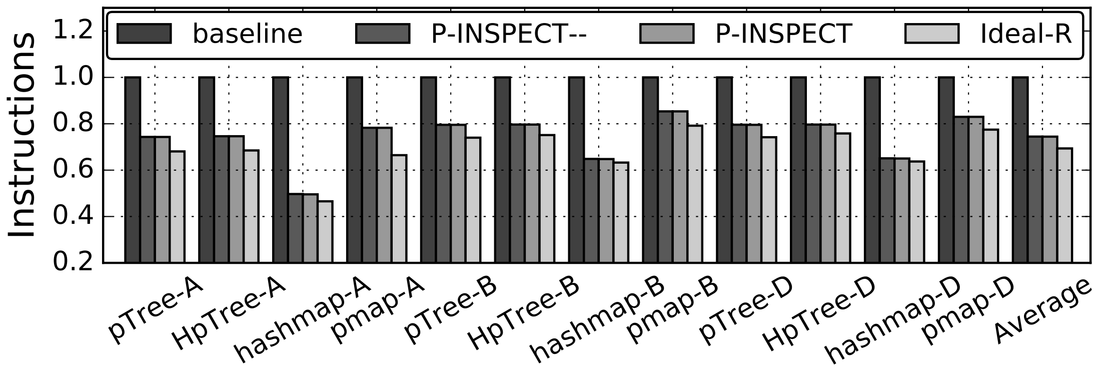


Figure 4.6: Instruction count of the YCSB workloads.

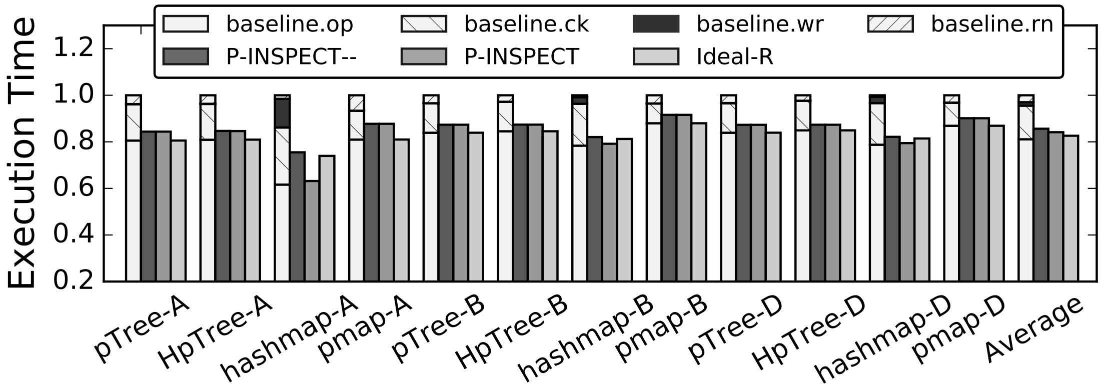


Figure 4.7: Execution time of the YCSB workloads.

From Figure 4.6, we see that, on average, P-INSPECT reduces the executed instructions by 26%. The same reduction is obtained by P-INSPECT--. This reduction is close to the 31% reduction attained by Ideal-R. The instruction reduction is larger in the write-heavy workload A than in the other workloads. This is because writes are more check-intensive. In hashmap-A the reduction reaches 50%.

Figure 4.7 shows that, on average, P-INSPECT-- and P-INSPECT reduce the execution-time of the key-value stores by 14% and 16%, respectively, relative to baseline. These are substantial reductions for real-world workloads. Note that Ideal-R delivers a 17% reduction in execution time, which is only 1 percentage point more than P-INSPECT. Hence, the P-INSPECT hardware effectively hides the overhead of persistence by reachability. Further, for some persistent write intensive workloads such as hashmap-A, P-INSPECT is faster than Ideal-R.

Figure 4.7 breaks down the execution time of the applications in baseline like in Figure 4.5. We see that, as in Figure 4.5, the checking overhead is the most dominant one.

To get another insight into the benefit of accelerating persistent writes, we have isolated the persistent writes within all the applications, and added-up the total time it takes for them to complete (i.e., we do not consider any overlapping with other instructions). We compare the time it takes to execute the separate write, CLWB, and sfence instructions against when the instructions are combined in a *persistentWrite* P-INSPECT operation. This metric ignores any overlap that these instructions may have with any other instruction.

We find that our *persistentWrite* operations that combine writes, CLWBs, and sfences take, on average, 15% less time than the instructions separated. In fact, for ArrayList, the reduction is 41%.

4.9.2 Bloom Filter Evaluation

We now characterize the behavior of P-INSPECT and its bloom filters over long execution runs using Pin. The TRANS bloom filter is cleared often, when the processing of a transitive closure completes. Because of that, we find that the TRANS bloom filter has a false positive rate close to zero. Hence, we do not consider it further.

The FWD bloom filter is cleared less frequently, by the PUT thread. PUT is woken up when FWD has over 30% of its bits set. We find that, on average, 357 forwarding objects are inserted in FWD before this threshold is reached. Our experiments also show that the average false positive rate of FWD across all the benchmarks is 2.7%. However, the actual rate of calling a software handler because of a false positive is less than 1%. This is because, in many scenarios, the outcome of the FWD check does not determine whether to call a software handler (Tables 4.4 and 4.5).

To characterize FWD, we run all the applications with the same configuration parameters as before, but with the ratio of operations of the YCSB workloadd: 5% of inserts and 95%

reads. We collect 50 samples per application and report the mean. Table 4.8 shows the results for each application.

Applic.	# Inst. between PUT calls (mill.)	# FWD checks per insert (thous.)	Avg. FWD occup.	PUT instr.
ArrayList	26,326	3,006.0	14.5%	0.0%
LinkedList	3,175	163.5	15.9%	0.2%
ArrayListX	43,778	4,937.4	15.8%	0.0%
HashMap	928	134.8	15.9%	1.6%
BTree	237	10.4	15.9%	6.5%
BPlusTree	45,367	3,201.0	15.9%	0.0%
pTree-D	478	22.4	16.0%	3.6%
HpTree-D	426	11.1	15.8%	3.8%
hashmap-D	969	85.2	16.1%	1.8%
pmap-D	92	1.9	15.9%	18.4%
Average	12,177	1,157.4	15.8%	3.6%

Table 4.8: Characterization of the FWD bloom filter.

Column 2 shows the number of instructions executed between invocations of the PUT. We see that this number ranges from 92 million to 45 billion. Generally, PUT is invoked rarely. Column 3 shows the number of FWD checks divided by the number of FWD insertions. We see that FWD reads are much more frequent than writes: on average, we have 1.15 million reads per write. Column 4 shows the average FWD occupancy. We take a sample every time that the program performs a FWD lookup. We see that this number is low. Its range is 14-16%. Column 5 shows the additional instructions executed by the PUT relative to the application instructions. On average, we see that the PUT overhead is very small.

The net effect of these measurements is that the PUT does not need to be frequently invoked for the FWD to exhibit a low false positive rate. Moreover, the results show that bloom filters are a low cost hardware mechanism that is very effective to handle object checks accurately.

Finally, we perform a sensitivity analysis of the size of FWD. The goal is to show how the FWD size affects the frequency of calling the PUT. Figure 4.8 shows the normalized number of instructions between PUT invocations for FWD sizes ranging from 511 bits to 4095 bits. We use the same target occupancy as before. Instruction counts for each application are normalized to the 2047-bit case. The numbers on the bars are the % increase in instruction count due to PUT.

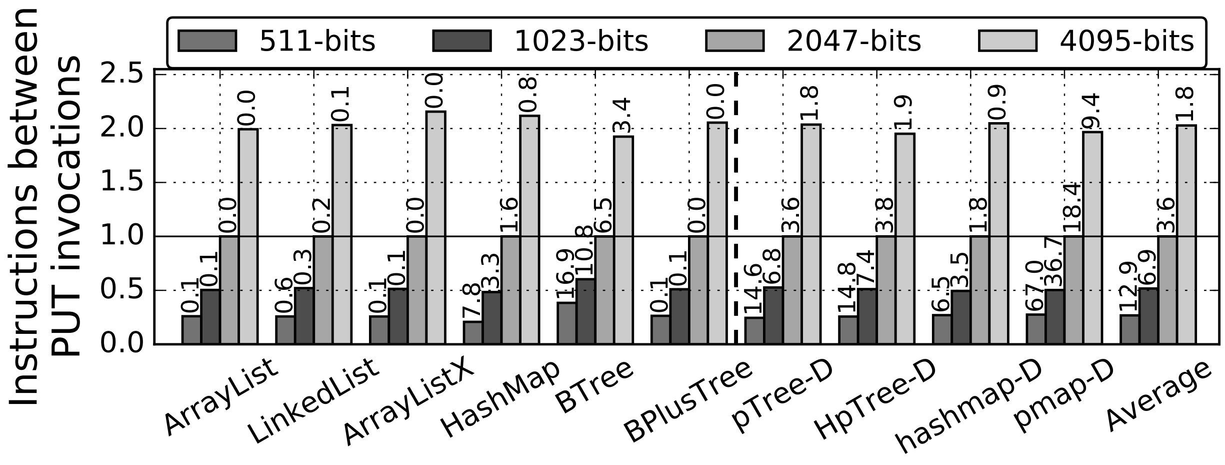


Figure 4.8: Normalized number of instructions between PUT invocations for different FWD sizes. The numbers on the bars are the % increase in instruction count due to PUT.

We see there is an almost linear relationship between the FWD size and the frequency of PUT invocations to clear the FWD. There is a tradeoff between FWD size and frequency of PUT invocation. Our 2047-bit design is a good design point.

4.9.3 Other Evaluation

To gain more insight into P-INSPECT, we perform two additional experiments. In the first one, we measure, for each application, the percentage of accesses to NVM addresses and the reduction in execution time of P-INSPECT over baseline. Table 4.9 shows the results. We see that both metrics are broadly correlated. There are some cases, however, where the execution time reductions are higher than expected from the fraction of NVM accesses. This effect is due to a higher fraction of persistent writes that miss in the caches, and benefit from our persistentWrite optimization.

In a second experiment, we re-run our evaluation with 4-issue (rather than 2-issue) cores. The average speed-ups of P-INSPECT−−, P-INSPECT and Ideal-R over baseline are 23%, 31% and 33% for the kernels, and 14%, 16% and 17% for the workloads, respectively. These numbers are practically the same as for 2-issue. The reason is twofold. First, all the environments become faster (including baseline and P-INSPECT); second, the long-latency NVM accesses stall the pipeline for both issue width designs.

Application	NVM accesses	Execution Time Reduction
ArrayList	13.3%	37.4%
LinkedList	6.4%	15.6%
ArrayListX	14.8%	55.9%
HashMap	8.3%	37.7%
BTree	6.3%	16.2%
BPlusTree	11.3%	24.4%
pTree-D	6.1%	12.8%
HpTree-D	2.8%	12.7%
hashmap-D	7.2%	20.5%
pmap-D	1.0%	9.9%

Table 4.9: Application NVM accesses and reduction in execution time.

4.10 RELATED WORK

System Support for NVM. The systems and storage communities have proposed many systems and frameworks to assist NVM integration. Besides the programming frameworks of Section 4.2, researchers have proposed to redesign the software systems. Examples include BPFS [19], NOVA [160, 161, 162, 163], Aerie [164] and PMFS [165]. These systems have different ordering and consistency attributes, and they handle their metadata information differently, but all aim for high performance NVM accesses in hybrid memory.

Hardware Optimizations for NVM. Many works provide hardware optimizations for NVM operations. In general, their goal is to reduce the overhead of persistent writes, either by introducing new persistency models [15, 104], or by removing persistent writes from critical paths and minimizing logging overheads [12, 105, 166, 167, 168, 169, 170, 171, 172, 173, 174]. As NVM normally relies on transactions for memory persistency, most of the optimizations focus on reducing the persistency overhead by relaxing the persistence order. For instance, WHISPER [12] proposes high-level ISA primitives to decouple ordering from durability. Other works propose hardware to optimize different aspects of NVM memories: efficient checkpointing [175, 176], improved NVM encryption operations [177, 178, 179, 180], and persistent object translation to accelerate the process of identifying the addresses of persistent objects [181, 182]. There is no work that proposes hardware techniques for supporting programmable NVM frameworks per se. These proposals are orthogonal to our work and, in fact, many can be combined with our work.

Recognizing Object State. Persistence by reachability requires dynamic fine-grain state information about each individual object. Existing hardware cannot provide the information needed or as fast as it is needed. For instance, bounds checking hardware [140] cannot be used to find out conditions such as whether an object is a Forwarding one or whether its Transitive Closure is being processed (Table 4.1). On the other hand, ARM's Memory Tagging Extension (MTE) [141], SPARC's Application Data Integrity (ADI) [183] or CHERI [142, 143] could be used for fine-grain identification. These proposals tag memory locations with bits that identify their state. However, these approaches are too slow for production code. As documented in [142, 184], MTE's, ADI's, and CHERI's precise exception mode introduces significant performance overheads. Since the hardware first needs to fetch the state (tag or capabilities) of the memory location and check if a precise exception needs to be raised, the original operation can only be performed after this load and check. In P-INSPECT, this overhead does not exist. By using bloom-filter hardware checks, P-INSPECT removes the loading of state from the execution's critical path.

4.11 CONCLUSION

To attain both user-friendly and high-performance NVM frameworks, this chapter introduced P-INSPECT, the first hardware architecture targeted to speeding-up persistence by reachability. In P-INSPECT, reads/writes leverage bloom-filter hardware to check some key states of the data accessed. Typically, no special action is needed and the read/write completes normally. Otherwise, a runtime software handler is invoked. P-INSPECT retains programmability and eliminates most of the execution overhead. For a set of workloads, it reduces the number of instructions executed by 26%, and the application execution time by 16%, delivering similar performance to that of an ideal runtime.

CHAPTER 5: DISTRIBUTED DATA PERSISTENCY

5.1 INTRODUCTION

Over the past decades, distributed storage systems such as key-value stores and transactional databases have become a core component of the cloud infrastructure [185, 186, 187, 188, 189, 190]. To meet ever-increasing performance requirements, these distributed applications typically avoid frequent accesses to slow storage devices such as solid-state drives (SSDs). This is because an access to such devices can take tens of microseconds. Instead, many applications store data in main memory and provide fault tolerance by making replicas (i.e., copies) of variables in other nodes' memories.

These replicas are managed by the runtime system using a *data consistency model*. There are many different consistency models in use [191], which differ in their strength. Strong consistency models strive to ensure that reading different replicas in different nodes returns similar, largely up-to-date versions of the variable. In contrast, weak models permit reads to different replicas to return inconsistent, sometimes stale versions. Commercial applications support a variety of models—e.g., Apache's ZooKeeper [192, 193] supports the strong Linearizable consistency, while Google's Bigtable [188] provides the weak Eventual consistency.

Unfortunately, due to the reduced support for data durability in these environments, distributed applications can suffer from data loss or slow data recovery. For example, a Facebook key-value store cluster needs hours to recover using remote data replicas, and days to recover using a backend storage [194, 195]. To make matters worse, a failure of the entire system can cause the permanent loss of in-memory state [26, 194].

The recent arrival of non-volatile memory (NVM) [1, 2, 6] offers a promising approach to help distributed applications attain both high performance and data persistence. Indeed, NVM can provide data durability in about 100-400 ns [7, 40, 196]. This is faster than a network round trip in data centers with high-performance interconnects such as InfiniBand [197, 198, 199, 200, 201].

To facilitate the use of NVM, researchers have developed a framework of *data persistency models* for a single machine with hardware-managed cache hierarchies (e.g., [104, 202]). These models vary in how eagerly they persist writes to NVM. For example, Strict persistency requires that a variable be persisted as soon as it is updated, while Epoch persistency only requires that updated variables be persisted at certain program locations.

As we use NVM in distributed applications, we have to carefully manage both the consistency and the persistency of the data. Although distributed data consistency has been well studied (e.g., [185, 191, 203, 204, 205, 206, 207, 208, 209]), it has almost always been used in systems which, at best, use slow storage devices for durability [26, 27, 28]. Hence, it is unclear how to best incorporate the NVM memory persistency models into these data consistency frameworks. In fact, it is unclear how these two classes of models interact with each other, and how their combination impacts data durability, performance, and programmer intuition in applications.

This chapter addresses these limitations. We introduce the concept of *Distributed Data Persistency* (DDP) model, which is the binding of the memory persistency model with the data consistency model in a distributed system. To reason about the interaction between data consistency and memory persistency, we use the concepts of *Visibility Point* and *Durability Point* of an update. The former is when the update is visible for consumption, and is specified by the consistency model; the latter is when the update is durable, and is specified by the persistency model.

To understand the tradeoffs, we consider five consistency models (Linearizable, Read-Enforced, Transactional, Causal, and Eventual), and five persistency models (Synchronous, Strict, Read-Enforced, Scope, and Eventual), pair-wise combine them, and design a low-latency distributed protocol for each of the resulting DDP models. Using these protocols, we investigate the trade-offs that DDP models offer in terms of performance, durability, intuition provided to the programmer, programmability, and implementability.

Our analysis shows that different DDP models deliver substantially different performance—e.g., one model delivers a 3.3x higher throughput than another for 100 clients. However, any fair comparison between the models has to consider other dimensions as well, such as durability and intuitiveness. In our analysis, we find that, typically, latency-sensitive applications that can tolerate some data staleness work best with DDP models that combine weak consistency with strong persistency. On the other hand, consistency-sensitive applications benefit from stricter consistency and relaxed persistency. For a broad class of applications, an intermediate model that combines Causal consistency with Synchronous persistency appears to be a good choice.

Overall, this chapter makes the following contributions:

• The concept of Distributed Data Persistency (DDP) model, which integrates a memory persistency model with a data consistency model in distributed systems. To reason about DDPs, we use the interaction between the Visibility and Durability points of an update.

• The design of novel, low-latency distributed protocols for many DDP models. These protocols are tailored to contemporary hardware, which provides low-latency, high-bandwidth network and storage through RDMA and NVM.

• A thorough trade-off comparison of different DDP models, in terms of performance, durability, intuitiveness, programmability, and implementability.

• A performance evaluation of different DDP models using distributed applications.

5.2 BACKGROUND

5.2.1 Data Consistency Models

To provide fault tolerance and performance, distributed computing applications replicate variables in the volatile memory hierarchy of multiple nodes. The replicas of a variable in different nodes may be read and updated concurrently by different processes. The consistency model of a system defines the requirements and guarantees of what data values can processes read. Many consistency models exist, as described in the distributed-computing literature (e.g. [191, 207, 208, 209]). Typically, there is a performance vs. data staleness trade-off: strict models require writes to update the replicas very soon, while weaker models sacrifice this requirement for higher performance. Next, we describe several models.

Linearizable Consistency or Linearizability. Linearizable consistency is the strongest consistency model for distributed systems. It requires that all writes to all variables be seen by all processes in the same order and, additionally, that all reads and writes be ordered by their timestamps [191, 210, 211]. This model is highly intuitive but may deliver low performance.

Causal Consistency. In this model, accesses are partially ordered according to the happens-before relationship. Specifically, two accesses within the same thread are ordered based on program order. Moreover, a read from a thread that obtains a value written by a write from another thread is ordered after the write. Further, this relation is built transitively. In this model, a thread can observe a write w only after it observes every previous write in w's happens-before history. Note that writes do not need to be applied instantly and, therefore, reads can return stale values. Replicas only need to reflect causally-related writes in order.

Eventual Consistency. In this model, writes are propagated lazily. The model only guarantees that all the replicas will eventually see all the writes. This model provides very weak consistency guarantees, and processes might read unexpected values. However, it offers great performance.

Transactional Consistency. Many contemporary databases organize their operations in transactions (Xactions). While different variations of this model exist, this chapter uses a simple one. The writes in a Xaction only need to be propagated to all the replicas by the end of the Xaction. If the Xaction fails, none of the updates are performed. Moreover, the operations within a Xaction can only see the effects of other Xactions that have completed prior to it.

Read-Enforced Consistency. In this chapter, we introduce this new model, which is slightly weaker than Linearizable consistency. It is inspired by the Read-Enforced durability of Ganesan et al. [26]. In this model, a write only needs to be visible to all the replicas at the point when a subsequent read tries to read any of the replicas. Compared to linearizability, this model allows faster completion of writes at the potential expense of delaying reads.

Current systems support most of these models, although linearizability is often eschewed for performance reasons. The availability and performance of Causal consistency make it an attractive choice for many applications [191, 207, 209, 212], such as online services. Eventual consistency is one of the most widely deployed [186, 191, 213] because of its performance.

5.2.2 Memory Persistency Models

The availability of NVM has led to the creation of multiple memory persistency models for single-server platforms. These models differ in how eagerly writes are persisted to NVM [104]. The models range from a strict one, where a write is immediately persisted to NVM, to relaxed ones, where writes are persisted lazily under certain conditions. These models need to be adapted to work in a distributed system, where nodes use asynchronous messages to coordinate.

In this chapter, we build on these models and on more traditional durability protocols that distributed systems have used to persist data to SSDs (e.g., [26, 27, 28]). In this section, we describe several persistency models.

Synchronous Persistency. In this chapter, we introduce this new model as the adaptation of the Strict memory persistency model from single-server systems [104] to distributed systems. In this model, when a replica is updated in volatile memory, it is immediately persisted to NVM. This model is strict, but the time of the persist depends on when the replica is updated, which in turn depends on the data consistency model of the system. For this reason, we call it *Synchronous*. It is the most intuitive model.

Read-Enforced Persistency. This model was introduced by Ganesan et al. [26]. It is more relaxed than Synchronous. Replicas do not need to be persisted when they are updated. Instead, the requirement is that all the updated replicas are persisted before any of them is read. This model guarantees that any value that has been read is also recoverable. However, there is no guarantee for updates that have not yet been read by processes—such updates may be lost in a crash or program failure.

Eventual Persistency. In the Eventual persistency model, persist operations are performed lazily. They happen whenever it is possible, without any concern about the order of persists. No other guarantees are provided. In case of a volatile storage failure, an arbitrary number of updates may be lost.

Scope Persistency. In the context of NVM persistence, there are proposals that persist a set of writes as a group. They include Strand [104, 202] and Epoch persistency [104, 214]. In this chapter, we propose a generalized approach where writes belong to *Scopes*—a concept reminiscent of Fence Scoping [215]. Every write is augmented with a *Scope ID*, and the application can invoke a *Persist* on a given Scope ID. Writes can be persisted in the background, but the model guarantees that all the writes in a scope are persisted by the time the Persist call for that scope terminates. The scopes in a program may be totally ordered, partially ordered, or not ordered at all, based on their Scope IDs. In our design, we use total order within a process and no order across processes. In all cases, if there is a volatile storage failure, the state of all the completed scopes is recovered, and that of those partially executed is discarded.

Strict Persistency. This is the strictest model. It dictates that a write should be persisted in the NVM of all the replica nodes by the time the write completes—possibly even before the replicas in the volatile memories of the replica nodes receive the update[216, 217]. On a failure of volatile storage, no update is lost. This model is relatively less interesting because of its high strictness.

Most existing systems use a persistency model close to Eventual persistency. This is because they value performance and do not want to pay the cost of persisting to SSDs or disks in the critical path. Some systems such as Redis [218] give the user a choice of models, ranging from Eventual-like to more strict. Some systems provide Synchronous-like persistency, such as LogCabin [219]. Read-enforced persistency has been recently proposed [26] and, to our knowledge, it is not used yet.

5.3 MOTIVATION: IMPACT OF CONSISTENCY AND PERSISTENCY MODELS

To motivate the importance of understanding how consistency and persistency interact, we perform a simple experiment. We take the Odyssey system [220] and implement a strict environment where both writes to volatile replicas and persists to NVM happen synchronously (i.e., a client's write does not return to the client until all replicas are updated and persisted). We then repeat the experiments without synchronously persisting to NVM, but still updating the volatile replicas in the critical path before returning to the client. Finally, we repeat the experiments without persisting to NVM or updating the volatile replicas before returning to the client. For all these experiments, we use a 3-node cluster, and every variable is replicated in all nodes. Each node has 24 Xeon E5-2687W cores, and connects to other nodes with Mellanox ConnectX-4 NICs that perform RDMA over Infiniband. The nodes run client threads issuing write requests and worker threads processing requests. These threads execute on separate cores. Table 5.1 shows the relative throughput of the three environments.

Volatile Updates in Critical Path?	NVM Updates in Critical Path?	Normalized Throughput
Yes	Yes	1
Yes	No	1.32
No	No	4.08

Table 5.1: Relative throughput of three environments.

As can be seen from Table 5.1, the throughput (normalized to the first environment) is significantly different. A relaxed environment that completes writes locally without updating or persisting replicas delivers a 4x higher throughput. Given the large number of consistency and persistency models, we need to develop a framework to examine the interactions between consistency and persistency models, and investigate the tradeoffs between the different combinations. These are the goals of the rest of the chapter.

Note that writing correct protocols for distributed systems is complex. For example, the Hermes distributed consistency protocol [204, 221] is 16K lines of code (LOC), and ZooKeeper [192, 193, 222] is 294K LOC. Hence, it is important to understand how consistency and persistency interact and how to systematically design protocols to support combinations of them.

5.4 INTEGRATING PERSISTENCY AND CONSISTENCY IN DISTRIBUTED SYSTEMS

We propose to bind memory persistency models with data consistency models in distributed platforms, creating what we call *Distributed Data Persistency* (DDP) models. To understand our approach, consider a distributed computer (e.g., a datacenter) where each node has a volatile memory hierarchy and some NVM. This is the architecture we will use in this chapter. Figure 5.1 shows two nodes of such a platform.

When an application such as a key-value store or a database runs on this platform, the

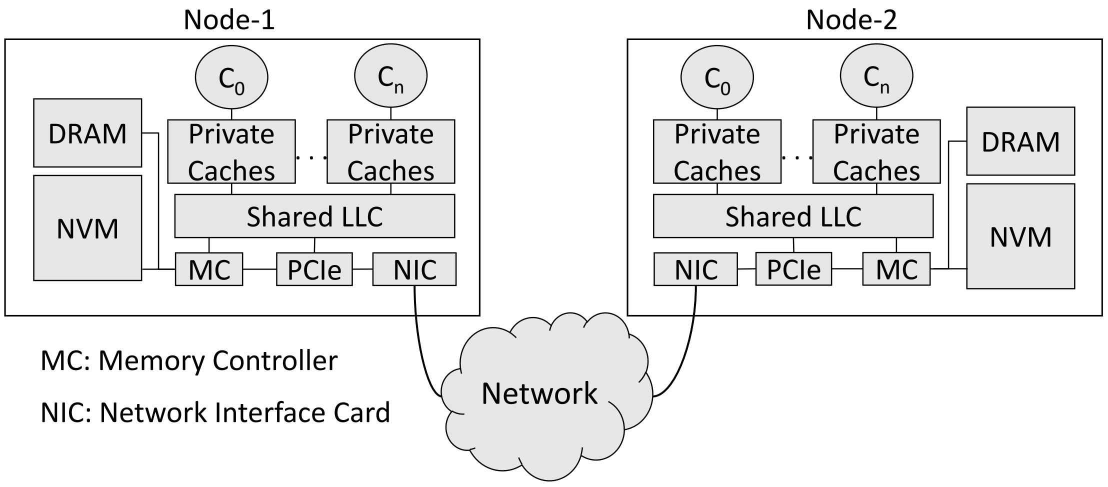


Figure 5.1: Distributed computer with NVM.

runtime typically makes copies of keys (or records) in the volatile memory hierarchy of multiple nodes—e.g., there are *N replicas* of every key. With today's Data Direct I/O (DDIO) technology [223, 224], the hardware makes the copies in the Last Level Caches (LLC) of the nodes. Such replication is performed for fault tolerance and performance, and may be later followed by the persistence to durable storage—NVM in our case.

In such a system, we decouple data consistency models from memory persistency models by using the concepts of *visibility* and *durability*. The consistency model is concerned with *when to propagate* the update of a key to the key's replicas in the volatile hierarchies of nodes; the persistency model is concerned with *when to persist* the update to the NVMs of nodes. To be specific:

The consistency model defines the *Visibility Point* (VP). The VP of an update with respect to a node is when the update becomes available for consumption at that node. The persistency model defines the *Durability Point* (DP). The DP of an update is when the update is made durable (in the necessary number of nodes, as required by the recovery system) and, hence, cannot be wiped out by a failure.

Broadly speaking, it helps to think as follows. Consistency models are more or less strict depending on how eagerly they propagate the update of a key to the volatile memory hierarchy of the nodes with replicas. Persistency models are more or less strict depending on how eagerly they persist the update to the NVMs of the nodes with replicas. This separation of concerns provides an intuitive way to plug the framework of persistency models into the framework of consistency models, creating what we call DDP models.

Table 5.2 shows the VP and DP of an update in the different consistency and persistency models, respectively, that we consider in this chapter. The models are listed from more to less strict. In the following, for a given variable, we call "replica nodes" all the nodes that contain a copy of the variable.

Consistency	Visibility Point (VP) of an Update
Linearizable	Wrt all nodes: when the update takes place
Read-Enforced	Wrt all nodes: before the update is read
Transactional	Wrt all nodes: at the transaction end
Causal	Wrt a node: after the VPs wrt the same node of all the updates in the happens-before history
Eventual	Wrt a node: sometime in the future
Persistency	**Durability Point (DP) of an Update**
Strict	When the update takes place
Synchronous	At the visibility point of the update
Read-Enforced	Before the update is read
Scope	Before or at the scope end
Eventual	Sometime in the future

Table 5.2: Visiblity and durability points of an update for different data consistency and memory persistency models, respectively. "Wrt all/a node(s)" stands for "with respect to all/a replica node(s)".

Consistency Models. In the *Linearizable* model, the VP of an update with respect to all replica nodes is when the update takes place. A client's write in a node is not completed until the volatile memories of all the replica nodes have been updated.

In *Read-Enforced* consistency, the VP of an update with respect to all replica nodes is sometime before the update is read by a node. A client's write completes as soon as the local key is updated; the update propagates to the replica nodes in the background. However, a read to any replica will stall until all the replica nodes have been updated.

In *Transactional* consistency, the code is annotated with transactions, and the VP of an update with respect to all replica nodes is at the transaction end. A write completes as soon as the local key is updated; the update propagates to the replica nodes in the background. The end-transaction operation stalls until all the writes in the transaction have updated all the replica nodes.

In *Causal* consistency, the VP of an update u with respect to a given replica node is sometime after the VPs with respect to the same replica node of all the updates U (to any key) that are in u's happens-before history. We call the *list* of U updates the *Causal History* (or *cauhist*) of u.

Finally, in *Eventual* consistency, the VP of the update with respect to a given replica node is sometime in the future. The update is propagated lazily. The model only guarantees that the update will eventually reach its VPs with respect to the different replica nodes.

Persistency Models. In the *Synchronous* persistency model, the DP of an update is at the VP of the update. In other words, when a volatile replica is updated (according to the consistency model), the replica is also persisted to NVM.

Table 5.2 also shows the even stronger but unintuitive *Strict* persistency, where the DP is when the update takes place. In this model, briefly mentioned by Talpey [217], when a node writes a key, the update has to be immediately persisted in the replica nodes, even if the volatile replicas in such nodes are not updated. In this chapter, we de-emphasize this model.

In *Read-Enforced* persistency, the DP is before the update is read. Specifically, a read to any replica will stall until all the replicas have been persisted to NVM.

In *Scope* persistency, every update is annotated with a Scope ID. The DP of an update is before or at the point when execution reaches the *end-of-scope* annotation for that Scope ID. When the annotation is reached, execution stops until all the writes in the scope are persisted in the replica nodes.

Finally, in *Eventual* persistency, the DP of the update is sometime in the future; the update is persisted lazily in the replica nodes.

Distributed Data Persistency (DDP) Models. We define a DDP model as the binding of a memory persistency model with a data consistency model. We represent it as <consistency model, persistency model>.

5.5 DDP PROTOCOLS FOR MODERN HARDWARE

Based on the insights from the previous section, we now design new distributed protocols for several DDP models. We target a modern data center architecture, where nodes communicate with low latency with advanced RDMA [198, 199, 225, 226] and use NVM for persistency. In this setting, where a round trip between nodes takes single-digit μs, and data persistency can be obtained in a few-hundred *ns*, we design protocols that emphasize low latency. Specifically, we design protocols that have no single leader—i.e., a client read or write request can be received and processed at *any node*. Moreover, on reception of a client's write, a node broadcasts messages to all the other replica nodes, instead of sending a message that sequentially visits all the other replica nodes.

Our designs are based on the linearizable-consistent (no persistency) protocol used by

Hermes [204]. Following their terminology, we call *Coordinator* the node that receives the client's read/write request for a key, and *Followers* all the other nodes with a replica of the key. Finally, for simplicity and like in Hermes, we assume that keys are replicated in all the nodes; reducing the number of replica nodes does not change the protocols conceptually, but may affect performance.

Before we describe the design of the DDP protocols, we outline the protocol operations.

5.5.1 Overview of the Protocol Operations

Table 5.3 shows the types of messages in our protocols. The basic protocol operation can be illustrated with a write in a strict model. When the coordinator receives a write from the client, it broadcasts an *INV (+data)* message to all the followers. This is an invalidation message that also includes the update. On reception of the message, a follower invalidates its current value of the key, sets the key's state to transient, and buffers the new key value (*data*). Then, it acknowledges the operation with an *ACK* message back to the coordinator. When the coordinator has received all the ACKs, it broadcasts a validation *VAL* message to all the followers. On reception, each follower knows that all the followers have been notified, and sets the key to the new value. As shown in Table 5.3, our ACKs and VALs may combine consistency and persistency information or may only apply to consistency (ACK_c, $VAL_$c) or persistency (ACK_p, $VAL_$p) events.

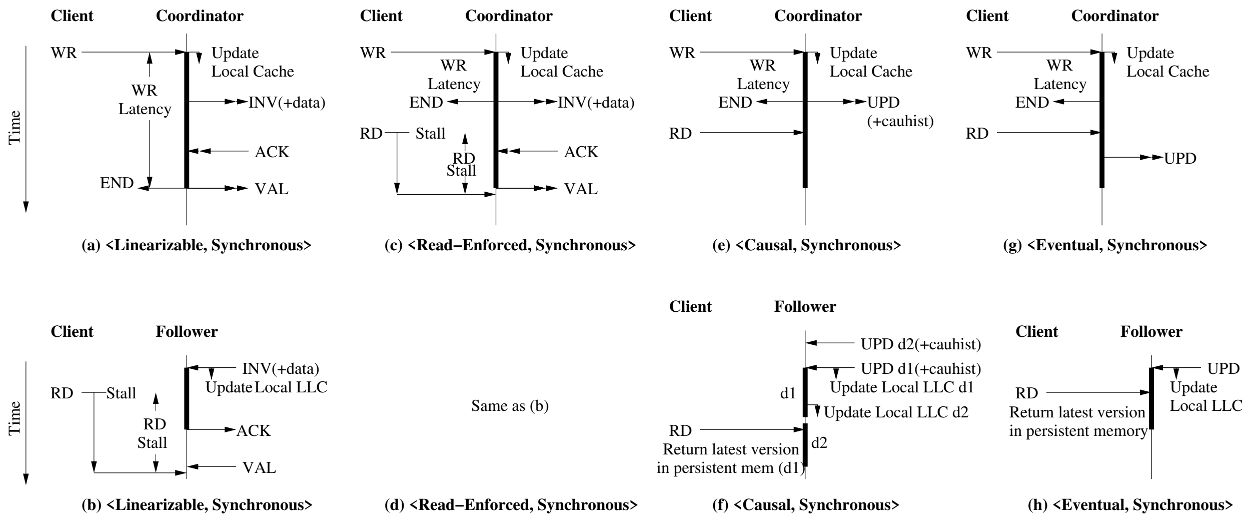

Client
Coordinator
Time
WR
Update Local Cache
WR Latency
INV(+data)
ACK
END
VAL
(a) <Linearizable, Synchronous>
RD
Stall
RD Stall
(c) <Read−Enforced, Synchronous>
UPD (+cauhist)
(e) <Causal, Synchronous>
UPD
(g) <Eventual, Synchronous>
Follower
INV(+data)
Update Local LLC
(b) <Linearizable, Synchronous>
Same as (b)
(d) <Read−Enforced, Synchronous>
UPD d2(+cauhist)
UPD d1(+cauhist)
Update Local LLC d1
Update Local LLC d2
d1
d2
Return latest version in persistent mem (d1)
(f) <Causal, Synchronous>
Return latest version in persistent memory
(h) <Eventual, Synchronous>

Figure 5.2: Timelines of the protocols for the DDP models that bind Synchronous persistency with various consistency models. The top row corresponds to the coordinator and the bottom one to a follower.

Message	Explanation
INV (+data)	Invalidates the current value of a key and provides its updated value (*data*)
ACK ACK_c ACK_p	Acknowledges an event Acknowledges a consistency event Acknowledges a persistency event
VAL VAL_c VAL_p	Marks the termination of an event Marks the termination of a consistency event Marks the termination of a persistency event
UPD (+cauhist)	Provides an updated value for a key plus the causal history of this update (*cauhist*)
INITX	Informs of the beginning of a transaction
ENDX	Informs of the end of a transaction
[PERSIST]s	Informs of the end of scope *s*
[XXX]s	[INV]s [ACK_c]s [ACK_p]s [VAL_c]s [VAL_p]s for scope *s*

Table 5.3: Types of messages in our protocols.

Causal and Eventual consistency protocols do not use *ACK* or *VAL* because there is no need for global information about *when* an update becomes visible. Hence, the coordinator simply sends update *UPD* messages with the data. In the case of Causal consistency, UPD includes the causal history (*cauhist*) of the write.

In Transactional consistency, the coordinator also sends begin and end transaction messages (INITX and *ENDX*). Finally, under Scope persistency, all messages are tagged with the scope *s* they belong to (e.g., *[ACK_c]s*). Furthermore, the coordinator also sends an end-of-scope message (*[PERSIST]s*) when execution reaches the end of the scope.

5.5.2 DDP Models with Synchronous Persistency

Figure 5.2 shows the timelines of the protocols for the DDP models that bind Synchronous persistency with various consistency models. The top row corresponds to the coordinator and the bottom row to a follower. In each subfigure, the left part shows the requests issued by a client. The thicker line shows the time during which a persist operation to NVM takes place. Furthermore, a down arrow means that the write updates the local node's cache. Finally, a two-headed arrow means that the message is sent to or received from multiple followers.

<Linearizable, Synchronous>. The coordinator is shown in (a) and a follower in (b). When the coordinator receives a write from a client, it updates its local cache and sends an INV (+data) to all the followers. On reception of INV (+data), a follower updates its local LLC and, to satisfy Synchronous persistency, persists the update to NVM before retuning an ACK to the coordinator. On reception of all the ACKs, the coordinator finishes persisting the update (to satisfy Synchronous persistency) before broadcasting a VAL to all the followers to indicate the operation is complete. *Only then* can the coordinator tell the client that the write is complete: all nodes have updated their volatile replica (required by Linearizable consistency) and have persisted it in their NVM (required by Synchronous persistency). Overall, we see that writes have high latency in this DDP model.

Consider now a client read to the same key. The coordinator cannot process any other request while the write is in progress. A follower stalls the read until all replicas have been updated (required by Linearizable consistency) and persisted (required by Synchronous persistency). Such condition is only guaranteed when VAL is received. Overall, reads also have high latency.

<Read-Enforced, Synchronous>. The coordinator is shown in (c) and a follower in (d). When the coordinator receives a write, it updates its local cache and sends an INV (+data) to all the followers. Read-Enforced consistency does not require the volatile replicas to be updated before completing the write—only when one of the replicas is read. Hence, the coordinator immediately tells the client that the write is complete, while the local persist and the remote updates and persists are in progress. In this DDP model, writes have low latency. In the follower, the operation is the same as in (b): once the LLC is updated and the update is persisted (required by Synchronous persistency), an ACK is sent. The coordinator collects all ACKs and finishes its persist. Then, it sends a VAL that indicates that *all replicas* have been updated and persisted.

A client read for the same key can only be serviced by the coordinator after sending VAL, and by a follower after receiving VAL. The reason is that Read-Enforced consistency requires a read to stall until all volatile replicas are updated. Moreover, Synchronous persistency requires that, at the same time as the replicas are updated, they are also persisted. Overall, reads have high latency.

<Causal, Synchronous>. The coordinator is shown in (e) and a follower in (f). On a write, the coordinator updates the local cache, sends the UPD with the cauhist of the write to all the followers, and returns to the client. Causal consistency only requires that a replica be updated after it has been updated with the updates in the causal history of the write. Subfigure (f) shows the follower receiving two updates (UPD d2 and UPD d1) in an order opposite to their cauhist. In this case, the first one (UPD d2) is buffered. When the second one (UPD d1) is received, it updates the LLC and, because of Synchronous persistency, it is persisted right away. Then, the first update is performed on the LLC and is persisted.

At any time, any read for the key that arrives at the coordinator or a follower proceeds with no stall. This is because Causal consistency places few constraints on when the replicas are updated, and Synchronous persistency only requires that, when the replica *is* updated, it is persisted. However, Synchronous persistency requires that the read get the latest *persisted* version, so that the version is recoverable on a failure. In our example in the follower, it is version d1. Overall, in this DDP model, both writes and reads have low latency.

<**Eventual, Synchronous**>**.** The coordinator is shown in (g) and a follower in (h). Using Eventual consistency makes this DDP model even more relaxed. Indeed, Eventual consistency adds no cauhist to the UPD messages. Updates are performed on the LLC of the follower in the order they arrive—but, because of Synchronous persistency, they are immediately persisted when they do. We denote the relaxed nature of Eventual consistency by delaying the sending of UPD. Overall, both writes and reads have low latency.

5.5.3 DDP Models with Read-Enforced Persistency

For brevity, we only discuss protocols for two DDP models that bind Read-Enforced persistency with consistency models: <Linearizable, Read-Enforced> and <Causal, Read-Enforced> (Figure 5.3).

<**Linearizable, Read-Enforced**>**.** The coordinator is shown in (a) and a follower in (b). This protocol decouples ACKs for consistency (ACK_c) from those for persistency (ACK_p). When the coordinator receives a write, it updates its local cache and sends INV (+data) to all followers. On reception of INV (+data), a follower updates its local LLC and immediately sends an ACK_c. When the coordinator receives all ACK_c, it knows that all the replicas have been updated. This is the condition that Linearizable consistency requires to tell the client that the write terminated. Read-Enforced persistency does not pose any condition on write termination. Overall, writes still have substantial latency, but less than in <Linearizable, Synchronous>.

A read in either the coordinator or follower, however, has to stall until all the replicas have persisted to NVM—as required by Read-Enforced persistency. Therefore, after a follower persists the update, it sends an ACK_p to the coordinator. When the coordinator receives all ACK_p messages and persists its version, it sends VAL_p. Because of Read-Enforced persistency, a read stalls in the coordinator until VAL_p is sent. Further, a read stalls in a

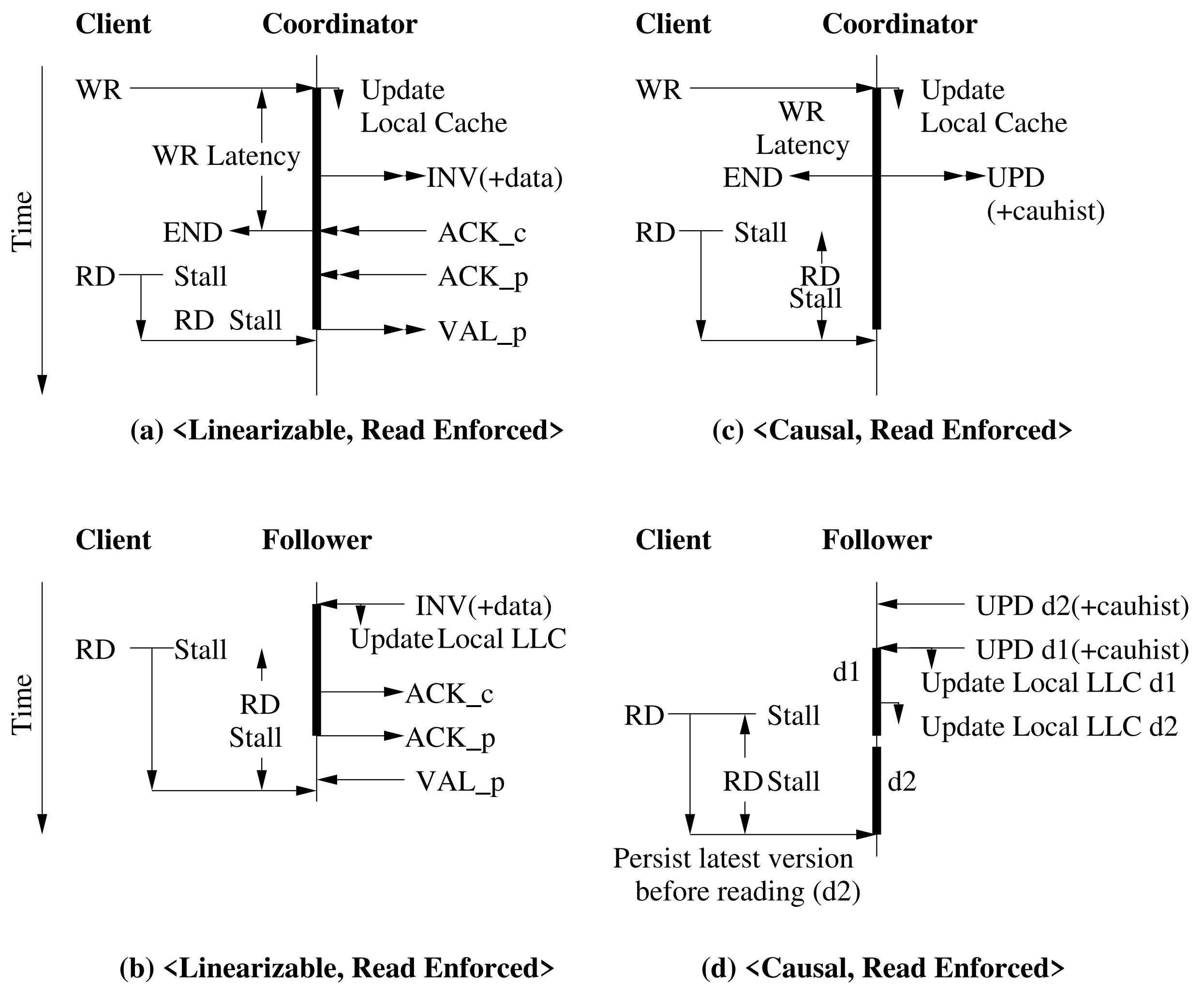


Figure 5.3: Timelines of the protocols for the DDP models that bind Read-Enforced persistency with Linearizable ((a) and (b)) or Causal ((c) and (d)) consistency.

follower until VAL_p is received. Hence, reads have high latency in this DDP model.

<Causal, Read-Enforced>. The coordinator is shown in (c) and a follower in (d). The operation of a write is the same as in <Causal, Synchronous> because the change in persistency model does not impact the actions on a write. However, reads now may have to stall longer. The reason is that, while Synchronous persistency only required that a read obtained a persisted version of the key, Read-Enforced persistency prevents a read to proceed unless the latest visible version of the key is persisted. Hence, as shown in (c), a read in the coordinator stalls until the update is persisted (thick line). Further, as shown in (d), a read in the follower stalls until the follower's latest visible version persists—i.e., the read waits until d2 persists and then reads it. Overall, in this DDP model, writes have low latency but reads do not.

5.5.4 <Transactional, Synchronous> DDP Model

To understand the protocol of a DDP model that includes Transactional consistency, we consider <Transactional, Synchronous> in Figure 5.4 (coordinator in (a) and follower in (b)). The client performs a transaction by issuing an *Init Xaction* request followed by multiple writes and reads, and then an *End Xaction* request. When the coordinator receives *Init Xaction*, it sends an INITX to all the followers, which persist the event (because of Synchronous persistency) and send an ACK to the coordinator. The Init Xaction request only completes when the coordinator has received the ACK from all the followers and persisted the event locally. At this point, all replica nodes are aware of the transaction. From then on, a write causes the coordinator to update the local cache, send the INV (+data) and immediately acknowledge to the client without waiting for the ACKs from the followers. Hence, a write is fast. The followers send their ACK after updating their LLCs, without waiting for the update to be persisted to NVM.

On reception of the *End Xaction* request, the coordinator broadcasts an ENDX to the followers. A follower, before returning an ACK for ENDX, must complete all the updates in the transaction to both the volatile LLC (as required by Transactional consistency) and to the NVM (as required by Synchronous persistency). When the coordinator has received all the ACKs and completed all its updates in the transaction to both the volatile caches and the NVM, it acknowledges the *End Xaction*.

During the transaction, a read in the coordinator or followers is fast. It does not wait; it simply reads the latest visible version—i.e., the latest one in the volatile memory hierarchy.

On top of this protocol, there is additional software infrastructure that detects and handles transactional conflicts. Specifically, at every read and write at the coordinator or followers, the address to be accessed is compared to those of all the reads and writes in the currently-active transactions. If a conflict is detected, different actions can be taken, such as transaction squashes or stalls, depending on the flavor of transactional model supported.

5.5.5 <Linearizable, Scope> DDP Model

Finally, to understand the protocol of a DDP model that binds Scope persistency with a consistency model, we consider <Linearizable, Scope> in Figure 5.5 (coordinator in (a) and follower in (b)). Recall that, in Scope persistency, writes, persist operations, and messages are tagged with a scope ID s.

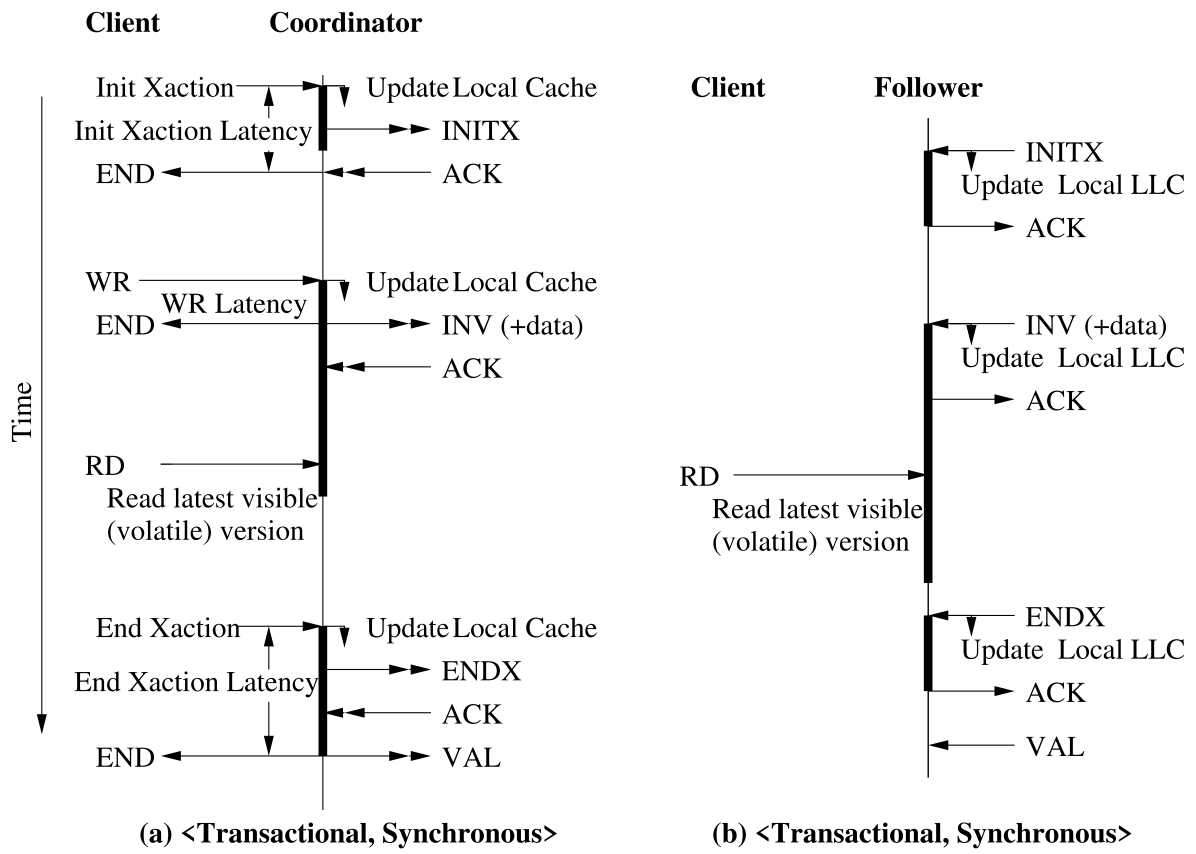


Figure 5.4: Protocol timeline for the DDP model that binds Synchronous persistency with Transactional consistency. The figure shows the coordinator (a) and a follower (b).

When the coordinator receives a write, it updates its local cache and broadcasts an INV (+data) to all followers. Each follower must update its volatile replica before returning an ACK_c. When the coordinator has received all the ACK_c messages, it broadcasts a VAL_c. At this point, since all the volatile replicas have been updated (as required by Linearizable consistency), the write is acknowledged. Writes are relatively slow because of Linearizable consistency.

When the coordinator receives a persist request for this scope, it broadcasts a PERSIST to all the followers. The latter persist to NVM all the updates in the scope and respond with ACK_p. When the coordinator has collected all the ACK_p and locally persisted all the updates in the scope, it broadcasts VAL_p. Since now the scope is fully persisted (as required by Scope persistency), the client is acknowledged.

A read in the coordinator or the followers is typically fast. It reads the latest visible version in the volatile hierarchy. However, sometimes it has to stall. Consider the read in Figure 5.5(b). There is a new version in the follower but, because of Linearizable consistency,

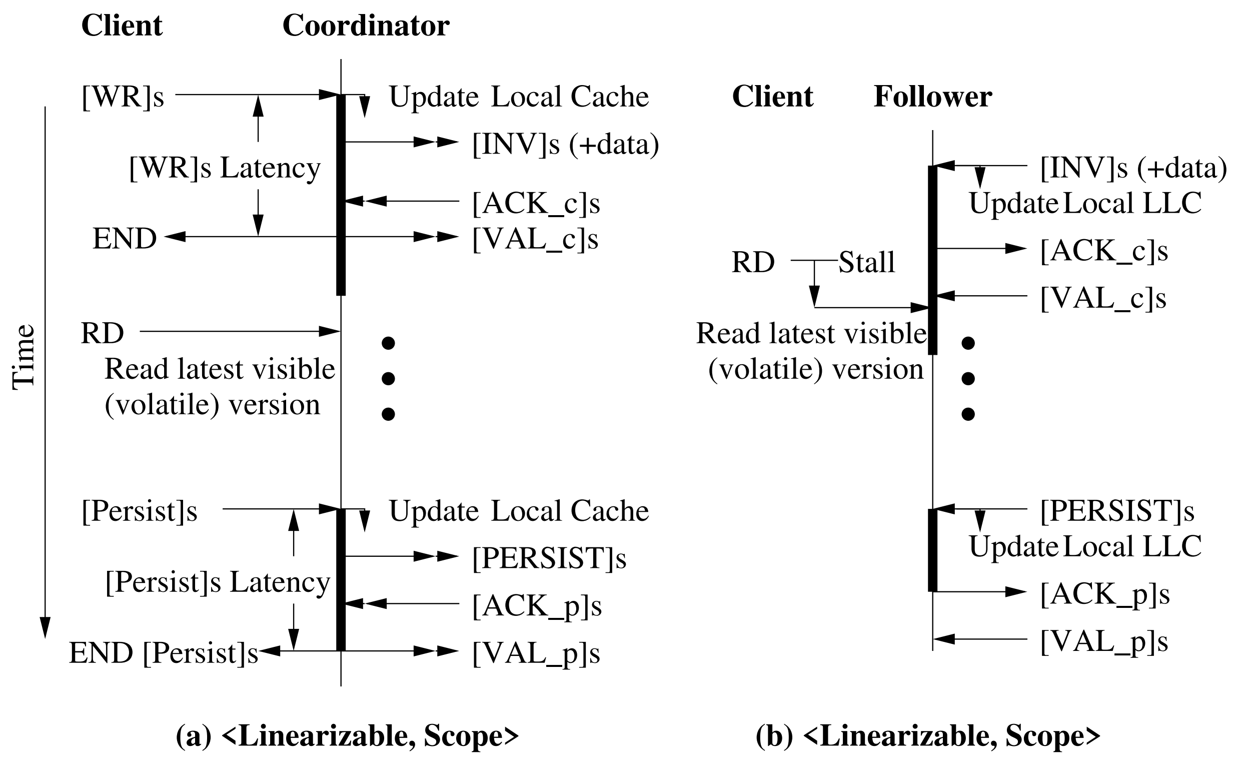


Figure 5.5: Timeline of the protocol for the DDP model that binds Scope persistency with Linearizable consistency. The figure shows the coordinator (a) and a follower (b).

the read cannot read it until VAL_c is received—i.e., when all the followers have this version as well.

5.6 TRADEOFFS BETWEEN DDP MODELS

The different DDP models provide different tradeoffs between durability, performance, intuition provided to the programmer, programmability, and implementability. Durability refers to how capable the system is to retain a consistent state after a failure that causes the loss of some or all the volatile state. Performance depends on three main factors: the speed of reads, the speed of writes, and the volume of traffic generated.

Programmer intuition is determined by what values a read can return. In particular, we consider whether the system supports monotonic reads and/or non-stale reads [26]. A system supports monotonic reads if, given two system-wide reads to the same variable, the later read always provides the same or a later version of the variable that the earlier read provided. A system fails to provide non-stale reads if a read that follows a write system-wide fails to provide the value of the write. The most obvious case is when a failure between the

			Performance				Programmer Intuition			Other	
Consistency Model	Persistency Model	Dura-bility	Wr Opt?	Rd Opt?	Traf-fic	Over-all	Monot. Rds?	Non Stale Rds?	Over-all	Program-mability?	Implemen-tability?
1. Linearizable		⇑	✘	✘	⇔	⇓	✔	✔	⇑	⇑	⇑
2. Read-Enfor.		⇔	✔	✘	⇔	⇔	✔	✘	⇔	⇑	⇑
3. Transactional	Synchronous	⇑	✔	✔	⇑	⇑	✔	✔	⇑	⇓	⇓
4. Causal		⇔	✔	✔	⇑	⇑	✔	✘	⇔	⇑	⇓
5. Eventual		⇓	✔	✔	⇓	⇑	✘	✘	⇓	⇑	⇑
6. Linearizable	Read-Enfor.	⇔	✔	✘	⇑	⇔	✔	✘	⇔	⇑	⇑
7. Causal		⇔	✔	✘	⇑	⇑	✔	✘	⇔	⇑	⇓
8. Linearizable	Eventual	⇓	✔	✔	⇓	⇑	✘	✘	⇓	⇑	⇑
9. Linearizable	Scope	⇑	✔	✔	⇑	⇑	✘	✘	⇑	⇓	⇓
10. Transactional		⇑	✔	✔	⇑	⇑	✘	✘	⇑	⇓⇓	⇓⇓

Table 5.4: Comparing different DDP models. "Opt" means optimized.

write and the read causes the loss of the written version. Intuitive systems support both monotonic and non-stale reads.

Programmability refers to the developer's ease of writing the application. For example, if the developer has to include annotations for transactions or scopes, programmability is hurt. Finally, implementability refers to the simplicity of the algorithms in the model. For example, keeping track of the happens-before histories of writes in the Causal consistency model complicates the implementation.

5.6.1 Specific DDP Model Analysis

Table 5.4 compares ten representative DDP models: five that bind Synchronous persistency, two that bind Read-Enforced persistency, one that binds Eventual persistency, and two that bind Scope persistency to consistency models. We consider durability, performance, programmer intuition, programmability, and implementability. In the table, upward, flat, and downward arrows mean high, medium and low; crosses mean no and tick marks yes.

Combinations with Synchronous Persistency. Row 1 shows the very strict < Linearizable, Synchronous>. Durability is high because a write does not return until it is persisted in all replica nodes. In terms of performance, writes are not optimized because a write in the coordinator only returns when all ACKs are received and the VALs are sent out; reads are not optimized either because a read in a follower is blocked until the VAL from the coordinator is received. For these reasons, even though we can say that the traffic is medium, the overall performance is low. In terms of intuitiveness, this model is highly intuitive because it provides both monotonic reads and non-stale reads. Finally, both programmability and implementability are high.

Row 2 shows <Read-Enforced, Synchronous>, which relaxes consistency by allowing writes to return as soon as the coordinator sends the INVs. Reads, however, are not optimized and still need to wait until a prior write to the same address is propagated to all the replicas and persisted. In this model, durability is medium because, if a failure occurs between the write and the subsequent read, the written version may fail to be persisted and be lost. Since writes are optimized but reads are not, and the traffic is medium, overall performance is medium. Monotonic reads are guaranteed but not non-stale reads, due to the failure just described: as the system recovers from the failure, a read will not return the value produced by the lost write. Hence, intuitiveness is medium. Programmability and implementability are high.

Row 3 shows <Transactional, Synchronous>, which is similar to <Linearizable, Synchronous > except that it operates at the transaction level. It has high durability—completed transactions are never lost. It optimizes writes through overlapping them inside a transaction, and reads by not stalling them. As a result, although traffic is high due to transaction begin/end messages, its performance is high. It provides both monotonic reads and non-stale reads and, hence, intuitiveness is high. However, programmability is low due to the need to annotate code with transactions, and implementability is low due to the need to implement transactions and their conflict detection and resolution.

Row 4 shows <Causal, Synchronous>, which optimizes both writes and reads. Neither of them stalls: writes return as soon as the coordinator sends the updates, and reads return the latest version in persistent memory. Since the write optimization may result in a write to be lost in a failure, durability is medium. Both reads and writes are fast but the traffic is high because each write carries its cauhist. Still, performance is high. Monotonic reads are guaranteed because, even if updates arrive at a follower out of order, the system buffers them and performs them in order based on their cauhist. However, non-stale reads are unsupported because writes can be lost to failures. Hence, intuition is medium. Programmability is high but implementability is low because of the need to buffer and enforce the cauhists.

Row 5 shows <Eventual, Synchronous>. As it provides practically no guarantees on when writes update replicas and persist, its durability is low. It has optimized reads and writes, and low traffic. Hence, performance is high. However, since neither monotonic reads nor non-stale reads are supported, intuitiveness is low. Programmability and implementability are high.

Relaxing Persistency. As we relax persistency and go from <Linearizable, Synchronous> to <Linearizable, Read-Enforced> in Row 6, we optimize writes by returning before they are persisted, but not reads. Writes can be lost in a failure. Consequently, durability decreases to medium. Despite the higher traffic due to double ACKs (Figures 5.3(a)-(b)), performance increases to medium. Further, since non-stale reads are not guaranteed in a failure, intuitiveness decreases to medium.

Row 7 shows <Causal, Read-Enforced>, which mostly optimizes writes over <Linearizable, Synchronous> (Figure 5.3). Because of this change, and despite the high traffic, its performance is high. However, since writes can be lost, durability is medium and non-stale reads are not supported. As a result, intuitiveness is medium. Implementability is low because of the need to keep cauhists.

Further relaxing persistency to <Linearizable, Eventual> in Row 8 creates a system with both read and write optimization but neither monotonic nor non-stale reads. The result is low durability, high performance, and low intuitiveness.

Finally, we consider Scope persistency. In <Linearizable, Scope> (Row 9) and < Transactional, Scope> (Row 10), we have systems with high durability: in a volatile storage failure, the state of all the completed scopes is recovered, and that of those partially executed is discarded. Within a scope, writes are optimized because they do not serialize their persists, and so are reads, which can read before the scope persists. As a result, despite the higher traffic caused by scope-persist messages, performance is high. Neither monotonic reads nor non-stale reads are guaranteed: on a failure, a group of writes may be discarded after being read because the scope did not persist. However, intuitiveness is still high because either the whole scope survives or no part of it does. Finally, both programmability and implementability are low due to the need to mark and support scopes. Further, both properties are worse if scopes are combined with transactions (Row 10).

5.7 EVALUATION METHODOLOGY

Modeled Architecture. We model the architecture of a distributed system with 5 servers. Each server is a 20-core multicore with 80 GBs of main memory composed of 64 GBs of NVM and 16 GBs of DRAM. The architecture parameters are shown in Table 5.5. Each core is an out-of-order core with private L1 and L2 caches, and a shared LLC. A 10% portion of the LLC can be used for direct cache access with DDIO [223, 224]. The servers' Network Interface Card (NIC) supports Remote Direct Memory Access (RDMA), which enables a server to access the remote memory of other servers.

We use RDMA because it supports low-latency data transfers across nodes without involving the remote processor. Unfortunately, current RDMA support is limited, in that an RDMA transaction provides no guarantees that the data have been successfully per-

Server Architecture Parameters	
Servers; Clients	5 servers; 20 clients per server
Multicore chip	20 out-of-order cores, 6-issue, 2GHz
Ld-St queue; ROB	92 entries; 192 entries
L1 cache	64KB, 8-way, 2 cycles round trip latency (RT)
L2 cache	512KB, 8-way, 12 cycles RT
LLC cache	2MB/core, 16-way, 38 cycles RT, 10% for DDIO
Network Parameters	
Network latency	1μs RT NIC-to-NIC
Network Bandwidth	200Gb/s
Queue Pairs	Up to 400
Per-Server Main-Memory Parameters	
Capacity	DRAM: 16GB; NVM: 64GB
Channels, Banks	DRAM: 4, 8; NVM: 2, 8
Latency	DRAM: 100ns read/write RT
	NVM: 140ns read, 400ns write RT
Freq; Bus width	1GHz DDR; 64 bits per channel

Table 5.5: Architectural parameters used for evaluation.

sisted in remote NVM. However, recent work has proposed RDMA extensions that facilitate operations with NVM [216, 227, 228]. In particular, in our evaluation, we follow SNIA's proposals [216] and model RDMA update commands that guarantee that, on acknowledgment, the remote volatile memory or the remote NVM (depending on the type of command) has been successfully updated. We also model an RDMA command that flushes data from a remote volatile memory to the remote NVM.

We model a high-end NIC with a bandwidth of 200Gb/s [199], and up to 400 Queue Pairs [229] for scheduling messages. Further, we model a 1μs round-trip latency for a message between two NICs [197, 198, 199].

Modeling Approach. Since we model non-existing, future RDMA primitives and high-end NICs, and want to do sensitivity analyses of even faster NICs and networks, we model performance using simulations. We use the SST simulator [61], Pin [155], and the DRAMSim2 memory simulator [63]. To model NVM, we modified the DRAMSim2 timing parameters and disabled refreshes. With Pin, we collect instruction traces for N cores processing read and write client requests to our key-value stores locally. Traces have no timing information. Then, we take these traces and simulate N cores in our distributed architecture. Timing is dynamically determined by the simulator. The simulation inserts all the protocol messages for correct operation of individual reads and writes.

With our simulation-based approach, we build an infrastructure that can be easily parameterized with new technology advancements and be used to perform sensistivity analyses.

Configurations and applications. We model the protocols for the DDP models that combine all 5x5 <consistency, persistency> pairs shown in Table 4.1. To minimize the effort of annotating codes for Transactional consistency and Scope persistency, we artificially select transactions to be 5 client requests and scopes to be 10 client requests. For our experiments, we use popular applications. Specifically, we use the widely used memcached [230] application and some simpler in-memory key-value stores such as HashTable, Map, B-Tree [231] and BPlusTree [232]. We evaluate all of them with Yahoo! Cloud Serving Benchmark (YCSB) [159] using different workloads with varying read and write ratios. In our experiments, we warm up the architectural state by running 1 billion instructions before simulating a total of 10 billion instructions. For brevity, the results show the average across all our applications.

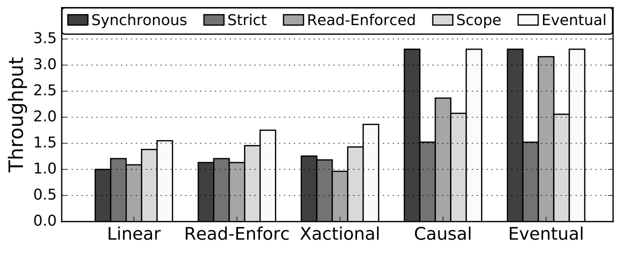
5.8 EVALUATION

5.8.1 Performance Analysis

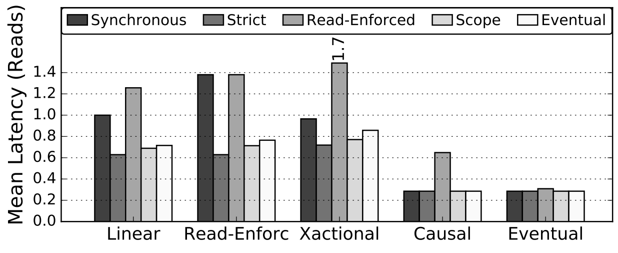
Figure 5.6 compares the performance of our 25 DDP models from Table 4.1 by showing throughput (measured in client requests/second) (*a*), mean read latency (*b*), mean write latency (*c*), mean access latency (*d*), 95th percentile read latency (*e*), and 95th percentile write latency (*f*). We find it easier to organize the discussion based on consistency models. Hence, in a plot, each group of bars is labeled with a consistency model, and each bar in the group corresponds to a persistency model, as shown in the legend. In a plot, all bars are normalized to <Linearizable, Synchronous>. We run YCSB workload-A, which has 50% read and 50% write requests.

General Observations. Focusing on throughput (Plot *a*), we see that models with Linearizable consistency have the lowest throughput, while those with Causal and Eventual consistency have the highest (often 2-3x higher). Those with Transactional consistency fail to deliver high throughput, mostly due to transaction conflicts—since ≈30% of the transactions conflict. In settings with minimal conflicts, we expect models with Transactional consistency to perform well.

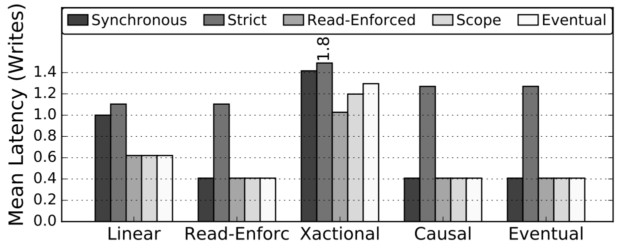
Typically, throughput is inversely correlated with mean read (Plot *b*) and write (Plot *c*) latencies. Models with Causal and Eventual consistency have low read and write latencies. The exception is the combinations with Strict persistency. The latter stalls writes until the updates are persisted everywhere. Models with Transactional consistency have high write la-



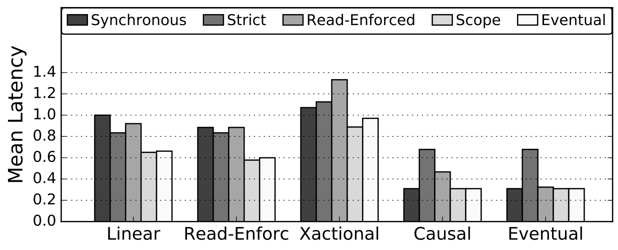
(a) Throughput



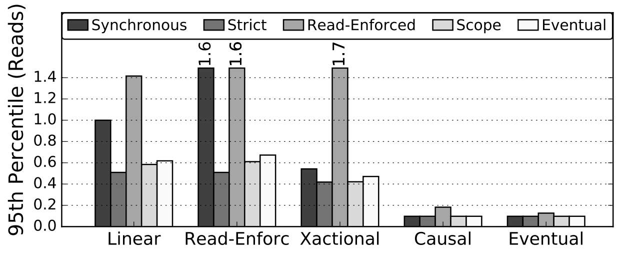
(b) Mean Read Latency



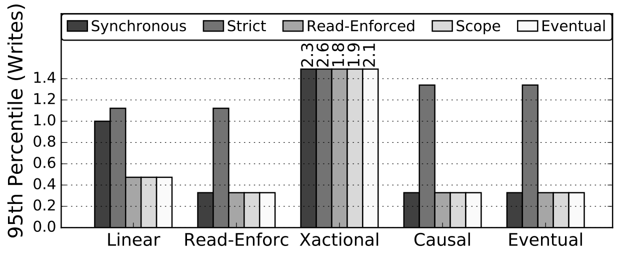
(c) Mean Write Latency



(d) Mean Latency



(e) 95th Percentile Read Latency



(f) 95th Percentile Write Latency

Figure 5.6: Performance of the different DDP models. In a plot, each group of bars is labeled with a consistency model, and each bar in the group corresponds to a persistency model as described in the legend. In a plot, all bars are normalized to <Linear, Synchronous>.

tencies, both because of conflicts—a request will not be satisfied until the transaction restarts and completes—and because writes bunch-up at Xaction end. This is especially the case for strong persistency models such as Strict and Synchronous. On the other hand, some models with Read-Enforced consistency have high read latency. This is because, by enabling more write overlapping than Linearizable consistency, they induce more NVM pressure, causing reads to stall longer for writes to persist. As a result, the throughputs of Read-Enforced consistency in Plot *a* are only modestly higher than those of Linearizable consistency

NVM pressure causes unexpected results. Specifically, under Linearizable consistency, Synchronous persistency has a lower read latency than Read-Enforced persistency. Read-Enforced persistency allows more outstanding writes to NVM, which increases NVM pressure and causes subsequent reads that conflict with those writes to take longer.

The 95th percentile plots mostly magnify the effects described. Models with Transactional consistency have a high write tail; some of those with Read-Enforced consistency have high a read tail.

Interesting Combinations. The plots also show that, for a fixed consistency model, which persistency model is used can make a big difference, changing the throughput by up to 2x. In aggregate, models with Strict persistency are the slowest ones, while those with Eventual persistency perform the best.

However, we note that binding a strict persistency model with a relaxed consistency model often delivers high throughput. In particular, the combinations <Causal, Synchronous> and <Causal, Read-Enforced> are attractive because they deliver high throughput (Plot *a*) while retaining medium durability and medium intuitiveness (Table 5.4). Of course, models that include an Eventual consistency or persistency model, such as <Causal, Eventual> or <Eventual, Synchronous> can perform great. However, as shown in Table 5.4 for the latter, both durability and intuitiveness are low.

Models with Causal consistency have good performance, but may require substantial buffering of writes with the more strict persistency models [212]. This is because writes need to be buffered until all their happens-before updates are persisted. In our experiments, we measure that Causal with Synchronous persistency needs about 1-2 orders of magnitude more buffered writes than with Eventual persistency.

Across consistency models, using Read-Enforced persistency delivers a throughput that is only slightly higher or typically lower than using Synchronous persistency. As indicated above, this is because Read-Enforced persistency forces many reads to stall. This poor performance is in contrast to the results of Ganesan et al. [26]. The reason is that our experiments use a higher number of clients (100 instead of 10), and we implement low-latency protocols with no designated leader. As a result, we find that over 30% of the read requests conflict with a yet-to-persist write in <Read-Enforced, Read-Enforced>, instead of 5.1% in Ganesan's work.

Overall, we conclude that different DDP models deliver quite different throughput values. In the extreme case, <Eventual, Eventual> delivers a 3.3x higher throughput than <Linearizable, Synchronous>. As shown in Table 5.4, however, performance is only one of our considerations. Consequently, different applications may prefer different DDP models. We discuss this issue in Section 5.9.

5.8.2 Sensitivity Analysis

To get a better understanding of what determines the performance, we perform three sensitivity analyses. Due to space reasons, we only show data for Linearizable and Causal consistency with all the persistency models.

First, Figure 5.7 shows the throughput as we vary the number of clients from 10 to 100 (the default) and to 150. The bars are normalized to <Linearizable, Synchronous> with 100 clients. The number of clients affects the traffic and the probability of conflicts between reads and writes. The figure shows that, in most of the DDP models, the throughput improves substantially as the number of clients decreases. For example, <Linearizable, Synchronous> increase the throughput by 2.2x when going from 100 to 10 clients. Conversely, the throughput decreases as the clients increase.

The exceptions are <Causal, Synchronous> and <Causal, Eventual>, which are largely unaffected by the number of clients. The reason is that, in these models, reads and writes do not stall. However, in Causal with Strict persistency, writes stall until they are persisted; in Causal with Read-Enforced persistency, conflicting reads stall; and in Causal with Scope persistency, reads and writes stall until the scope persists.

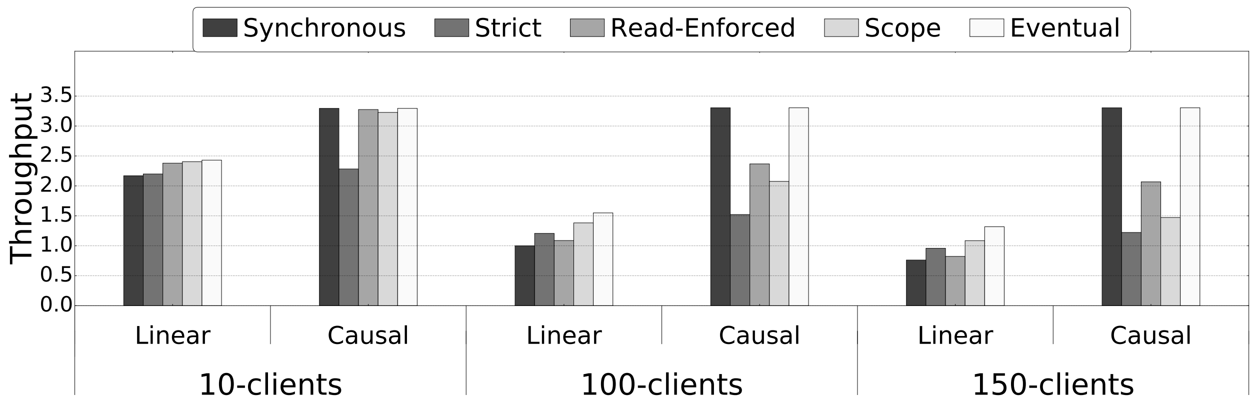


Figure 5.7: Sensitivity analysis for different clients.

Although not shown in the figure, we also run Transactional consistency. The experimental results show that, as the number of clients reduces from 100 to 10, the number of transaction conflicts decreases by roughly 50%, and Transactional consistency becomes more competitive.

Figure 5.8 shows the throughput as we vary the NIC-to-NIC round-trip latency from 500ns to 1μs (the default), and to 2μs. All bars are normalized to <Linearizable, Synchronous> with 1μs. The figure shows that the network latency affects mostly the models with Linearizable consistency, while those with Causal consistency are barely affected. The former are affected because network transfer is in the critical path. For example, the throughput of <Linearizable, Synchronous> decreases by 12% when going from 1μs to 2μs. Models with Causal consistency are not affected because updates are generally communicated to other servers in the background.

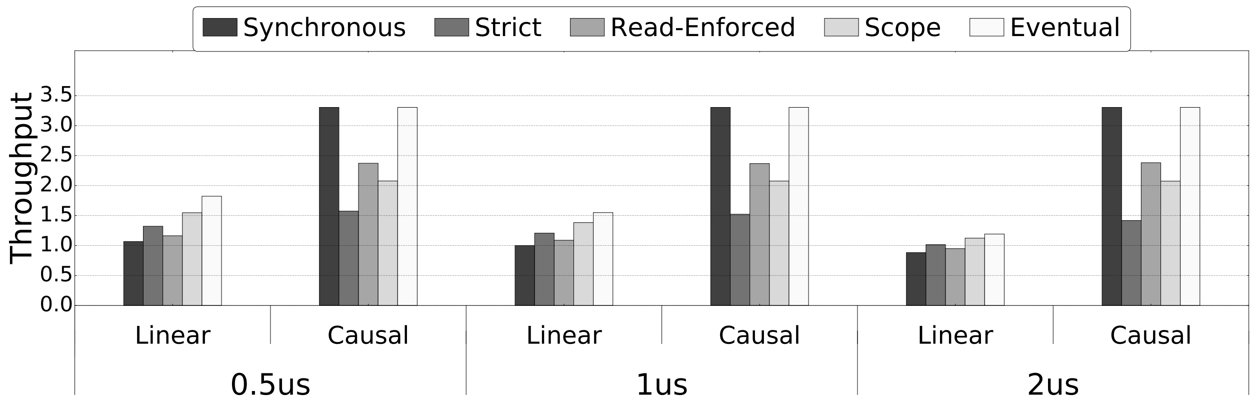


Figure 5.8: Sensitivity analysis for different NIC-to-NIC round-trip latencies.

Figure 5.9 shows the throughput as we vary the relative fractions of reads and writes in the workload. We consider workload-B (95% reads and 5% writes), workload-A (the default, which has 50% reads and 50% writes), and our defined workload-W (95% writes and 5% reads). All bars are normalized to <Linearizable, Synchronous> for workload-A. From the figure, we see that the more read-intensive a workload is, the less affected by the consistency and persistency models it is. This is because such models dictate when writes are propagated and persisted. Reads are affected indirectly.

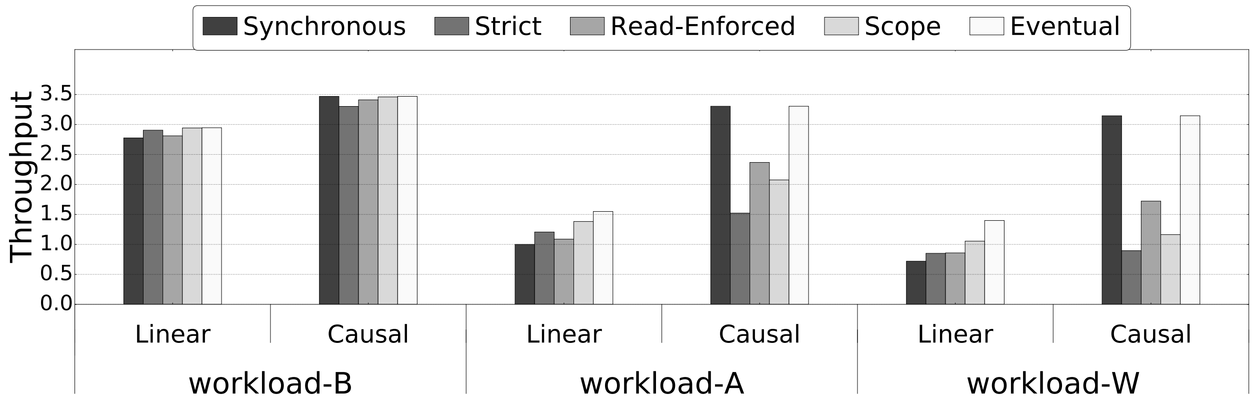


Figure 5.9: Sensitivity analysis with different relative fractions of reads and writes.

5.9 IMPLICATIONS FOR APPLICATIONS

Application developers choose the consistency model according to their needs. Our evaluation in Section 5.8.1 has suggested which persistency models should go with which consistency models and, therefore, which DDP models to use. We now summarize the main insights.

Latency-sensitive applications that can tolerate a certain level of data staleness such as web browsing and social networking often use Eventual consistency [233, 234]. In this case, using Synchronous persistency is a good choice in terms of performance (Figure 5.6(a)), programmability, and implementability (Table 5.4).

For consistency-sensitive applications that require bounded staleness but can accept modest latencies, such as certain web search services [191, 233], stricter consistency models such as Read-Enforced consistency are good choices. In this case, Figure 5.6(a) suggests to combine them with Scope or Eventual persistency, which results in high throughput and low tail latency. Some of these applications aggregate data from thousands of anonymous users and, therefore, the loss of a certain amount of recent data is acceptable.

Applications that want to attain both reasonable consistency guarantees and high performance such as photo sharing and news readers [207, 212] often use Causal consistency. In this case, Figure 5.6(a) suggests to use Synchronous persistency. In fact, the figure shows that Causal consistency delivers some of the best performance of all cases in combination with multiple persistency models. Therefore, developers can select the appropriate persistency model based on the data durability requirements of their application.

Applications that require transactional guarantees [186, 235, 236, 237] such as Google's globally distributed Spanner database [237] use a form of Transactional consistency. This consistency model can deliver high throughput, but its performance suffers if transaction conflicts are frequent. In this case, Figure 5.6 shows that using Read-Enforced persistency is not a good choice, since reads end-up suffering long stalls. Other persistency models such as Scope or Eventual should be used.

Many systems use hybrid consistency models [233, 234, 238, 239, 240]—e.g., Linearizable or Read-Enforced consistency in a local cluster, and Eventual consistency across the entire distributed system in a data center [238]. In this case, our results suggest using Scope or Eventual persistency for the local cluster, and Synchronous persistency across the system.

Generally, we find that Causal consistency combined with either Synchronous or Eventual persistency is highly competitive, and robust to increases in number of clients, network latencies, and write traffic. In these DDP models, reads and writes do not stall. Beyond this, models combining strong consistency with weak persistency, or weak consistency with strong persistency are typically best. Finally, as RDMA advances improve remote communication, and NVM usage speeds-up durability, companies will increasingly favor stronger consistency models and stronger persistency models, respectively.

Irrespective of the DDP model, a recovery algorithm is invoked on a crash. The complexity of the recovery is higher in the weaker models than in the stricter ones. For example, strict models like <Linearizable, Synchronous> have a simple recovery process because all nodes have the same persistent view of the data. On the other hand, weaker DDP models such as those with Eventual consistency or persistency may need an advanced recovery algorithm, such as a voting-based one [241].

5.10 RELATED WORK

Distributed Consistency and Persistency Models. Many distributed data consistency models have been studied over the past decades (e.g., [191, 203, 205, 207, 208, 209, 242]). However, few of them have decoupled the discussion of data consistency from data durability. With the advent of NVM, memory persistency models have been proposed [104, 243]. They mainly target a single machine with a hardware-controlled cache hierarchy. Recently, Katsarakis et al. [204] developed Hermes, a broadcast-based replication protocol for in-memory datastores. Hermes uses Linearizable consistency and does not persist data to durable storage. It relies on remote replicas for data recovery, which may cause long recovery delay and even data loss upon full datacenter crashes [26, 194]. Ganesan et al. [26] proposed read-enforced persistency to attain a strong model with low performance overhead. In this work, we decouple consistency from persistency in distributed systems and show how they interact.

Distributed NVM Systems. Recently, many distributed systems have been built based on persistent memory (e.g., [162, 194, 244, 245, 246, 247, 248, 249]). For instance, FileMR [162, 248] developed a distributed NVM file system through RDMA. FaRM [245] implemented a distributed transactional system with battery backed DRAM and RDMA, which supports strict serializability and data durability. Most of these distributed systems follow one of the conventional distributed data consistency models, and develop optimization techniques to reduce remote persistency overhead. We believe the work presented in this chapter will help the future development of such systems by offering insights into DDP models.

Network Support for NVM. To improve the performance of remote data persistency, some architectural techniques have been proposed [162, 227, 228, 250]. Hu et al. [228] present persistence parallelism techniques to improve the network bandwidth utilization using RDMA. Industry plans to extend RDMA to support atomic write and flush operations for NVM [250]. Our work is orthogonal, and can benefit from such hardware. The DDP models we propose can take advantage of network hardware optimizations.

5.11 CONCLUSION

This chapter proposed the concept of Distributed Data Persistency (DDP) model, which is the binding of the memory persistency model with the data consistency model in a distributed system. We reasoned about the interaction between consistency and persistency using the Visibility and Durability points. We designed low-latency distributed protocols for DDP models that combine five consistency models with five persistency models. For the resulting DDP models, we studied the trade-offs between performance, durability, intuitiveness, programmability, and implementability.

We found that, in general, models combining strong consistency with weak persistency, or weak consistency with strong persistency are typically highly competitive. In addition, Causal consistency combined with different memory persistency models is often a good choice.

CHAPTER 6: HARDWARE ASSISTED DISTRIBUTED TRANSACTIONS

6.1 INTRODUCTION

The popularity of cloud systems is due in great measure to the existence of many distributed applications that provide services across multiple nodes in a data center or across data centers. Some of these applications are user facing [233], such as web browsing, while others provide background services such as distributed data storage or processing [193, 251]. Of particular importance to the cloud infrastructure are distributed storage systems, such as key-value stores and databases (e.g., [185, 186, 187, 188, 189, 190]). Such applications ensure that distributed data is safely stored and accessible to users on demand. Many of these storage systems use the transactional model, whereby queries are written as transactions that either complete or fail without leaving any side effect. Using transactions in storage systems is very popular [245, 252, 253, 254, 255] due to the transactional model's clear ACID (Atomicity, Consistency, Isolation, and Durability) semantics [256, 257] and resulting simpler application design and implementation.

Recently, the hardware infrastructure on which these applications run has been improving rapidly. On the one hand, networking hardware has become steadily faster. Both commercial [258, 259] and, especially, custom-designed network solutions have substantially reduced the round-trip latency of node-to-node communication—to under one microsecond in a data center [197]. On the other hand, so called smart network interface cards (SmartNICs) are including progressively more advanced hardware support [29, 30, 31, 199, 260]. Such support can enable the development of efficient RDMA operations, further reducing communication overheads and possibly off-loading computation from the host processor.

A result of these hardware changes is that existing distributed application software runs increasingly inefficiently. Applications wait for short times that cannot be effectively hidden using current hardware and software latency-hiding techniques (i.e., the well-documented killer microsecond [261, 262]). More importantly for our analysis, applications have hefty housekeeping software overheads on the critical path that limit their performance.

Specifically, state-of-the-art distributed transactional storage systems such as Microsoft's FaRM [245, 254] have major software overheads. They result from managing and checking the read and write sets of transactions—i.e., the records that a transaction accesses plus their metadata, including versions, values, and source nodes. Other overheads result from the fact that reads and writes are supported at record granularity—forcing whole-record transfers when only some fields may be really needed. Finally, other software overheads result from many operations to lock and unlock records, poll for lock and unlock completion, and re-read records before committing to check for transaction conflicts. In our analysis, we find that such overheads are responsible for 60-70% of the execution time of various workloads on FaRM.

Given that these trends are only likely to accelerate, in this chapter, we introduce new hardware structures to eliminate high-overhead software operations in distributed transactional systems. We start by analyzing the sources of software overhead. Based on the analysis, we propose novel hardware that includes bloom filters for a variety of tasks and SmartNIC support for efficient remote communication. We then develop *HADES*, a new optimistic concurrency control (OCC)-based distributed transactional protocol that leverages this hardware to provide high-performance distributed transactions. HADES is easy to use in different transactional systems, as it is agnostic to the data layout and does not require any extension to the data records. We also propose a cheaper, hybrid hardware-software implementation of HADES. Finally, Chapter 5 indicated that persistency models affect the throughput of distributed transactional consistency applications. In this chapter, we evaluate the performance impact of persistency to HADES, and devise an in-network persistency mechanism that accelerates persistency operations at remote nodes. HADES' hardware enables this optimization by utilizing the SmartNIC to persist data in-network, instead of the slower NVM for better performance.

Using a simulation-based evaluation, we estimate that a set of distributed transactional workloads running on five nodes of five cores each, execute on average 2.7× faster on HADES than on FaRM. Further, they run on average 2.3× faster on Hybrid HADES than on FaRM. Also, the different persistency models greatly affect the performance of distributed transactions, up to 77% difference, and in-network support for persistency can decrease the overhead of persistency operations.

Overall, this chapter's contributions are:

- An analysis of the main sources of software overhead in a state-of-the-art distributed transactional system.
- New hardware structures to eliminate high-overhead software operations in distributed transactional systems.
- A new distributed transactional protocol (HADES) that uses this hardware to provide low-overhead distributed transactions.
- A performance evaluation of HADES and HADES-Hybrid, and their comparison to a state-of-the-art transactional system.
- An analysis of the performance impact of distributed persistency on HADES.
- New hardware to extend HADES for supporting in-network persistency operations for

higher throughput.

6.2 BACKGROUND

Distributed transactional systems are a key component of the storage infrastructure in modern data centers [186, 245, 263, 264, 265, 266, 267]. They enable multiple clients to access shared data structures correctly across distributed servers at scale in a concurrent manner. To attain high concurrency and performance for distributed transactions, state-of-the-art systems usually leverage RDMA primitives to enable fast remote data accesses [245, 254, 268, 269].

These systems augment the data records with extra fields that the software uses to manage the structures. Figure 6.1 shows a typical example. In this case, a record is augmented with fields that include the record version, a lock, the incarnation to detect whether the record has been freed, and a per-cache-line version V_{Ci} to support optimistic concurrency control and conflict detection between concurrent transactions that operate on the same record.

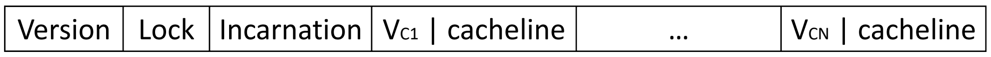


Figure 6.1: Augmented record to support transactions in a typical distributed transactional system.

To ensure that concurrent transactions are executed in a proper way by following the ACID guarantees, these systems use a distributed transactional protocol. Typically, this protocol has three main phases: *Execution*, *Validation*, and *Commit*. Figure 6.2 shows a phase-by-phase example of the protocol. In the example, a coordinator node executes a transaction with a mix of accesses to the memory of the local node and to remote nodes. Records A and C are read, and B and D are written.

The three phases work as follows:

Execution Phase. During the *Execution* phase, reads of local records are performed locally, while reads of remote records are executed by sending RDMA read operations to the nodes that have the corresponding records. All the read operations in the transaction are recorded in the transaction's Read Set, with the version of the records. Before a read can be recorded in the Read Set, the atomicity of the read must be validated. This involves checking that all the cache lines of the record have the same version and, therefore, no write is interfering with the read.

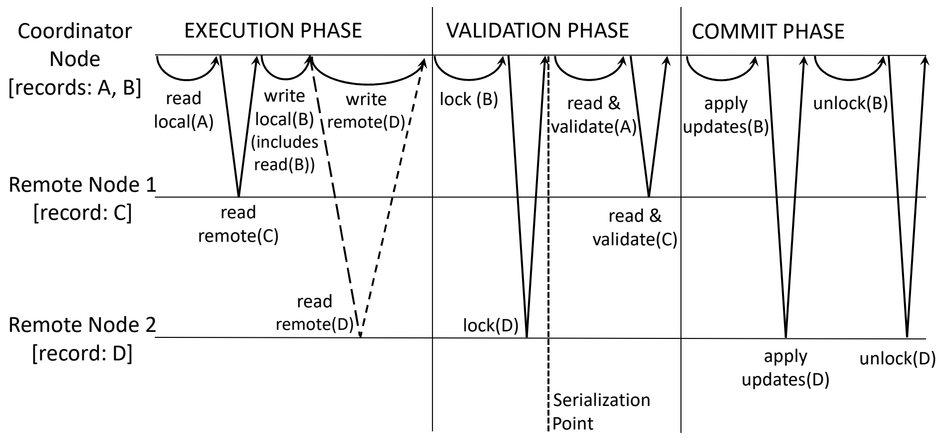


Figure 6.2: Typical protocol for distributed transactions.

Both local and remote writes are buffered locally until the transaction can safely commit. The version, the addresses, and the data of all written records in a transaction are stored in the transaction's Write Set.

Transactional systems usually operate at record-level granularity. This means that, even though an update is only modifying part of a record, the system needs to first fetch the whole record before the write, apply the update, and buffer the record in the Write Set until the transaction commits.

Validation Phase. During the *Validation* phase, the coordinator needs to confirm that the transaction does not conflict with any other transaction executing on the local node or on a remote one. For this reason, it first locks its local and remote Write Sets. This can be done using the Compare-and-Swap (CAS) and RDMA CAS atomic operations. Once locking succeeds, it is guaranteed that the transaction can be serialized. Then, the coordinator fetches the data versions of all the records read, re-reads their current version number, and compares it to the version that was read during the Execution phase. The goal is to identify conflicts. If there is no conflict, the transaction can proceed to the Commit phase. Otherwise, the transaction is squashed and needs to be re-executed.

Commit Phase. During the *Commit* phase, the data versions of all the written records are updated, and the writes are performed for both local and remote records. After that, the local and remote Write Sets are unlocked to allow future accesses from other transactions.

6.3 SOFTWARE OVERHEAD IN A STATE-OF-THE-ART DISTRIBUTED TRANSACTIONAL SYSTEM

To assess the software overheads in current distributed transactional systems, we implement the state-of-the-art Microsoft FaRM protocol [245, 254] on top of a HashTable key-value store. FaRM is a popular system, which has inspired many distributed transactional protocols [265, 268, 269, 270]. We implement the records of the key-value store as FaRM requires (Figure 6.1). We instrument the code to capture the overheads of the software.

Operation with High Software Overhead in FaRM	**Proposed Hardware to Minimize Overhead**
Manage and check the Read and Write sets of a transaction.	Bloom Filters (BFs) next to the directory/LLC (for local accesses) and in remote NICs (for remote accesses).
Before performing a write, update the version of the record.	No record versions.
On a record read, check for read atomicity. Unable to do zero-copy reads.	Use the BFs to partially lock the directory while reading multiple lines.
Operation at record granularity, which causes: (i) On a read/write, bring the whole record, and (ii) Potential increase in number of transaction conflicts.	Operation at cache line granularity.
Perform many RDMA and local operations beyond reads and writes. They include: (i) lock/unlock, (ii) poll for lock/unlock completion, and (iii) re-read record versions at validation time, to check for conflicts.	Eliminate some RDMA and local operations. Support some new RDMA messages, including *Intend-to-commit*, *Ack*, and *Validation*. Off-load RDMA operations from the core to the NIC. Use the BFs to partially lock the directory while a transaction is committing.

Table 6.1: Reducing the overhead of distributed transactional systems.

The left column of Table 6.1 lists the major sources of software overhead that we have observed in FaRM. The first source is managing and checking the read and write sets of transactions. The *Read Set* of a transaction is the set of records that the transaction reads plus their metadata (e.g., their versions and the nodes where they live). The *Write Set* is the set of records that the transaction writes, the values written, and the metadata. In FaRM, writing a record involves two reads and two writes: first, a read of a record or metadata (from either a remote or a local node), then a write to the write set, and then, at commit time, a read from the write set and a write to the final location.

FaRM also adds other software overheads in every write and read. Specifically, before performing a record write, the software needs to update the record's version. Further, on a record read, the software needs to check that all its fields have the same version (i.e., that the read is atomic: there is no transaction writing to the record while the record is being read). This means that one cannot do zero-copy reads: one reads into a temporary location, checks for atomicity, and then copies the record to the destination location.

Other overheads of FaRM stem from the fact that reads and writes are performed at record granularity. On an access, the whole record is read—rather than a few fields. Moreover, transactions conflict even when they access different fields of a record.

Finally, FaRM performs many RDMA and local operations beyond the basic reads and writes. They include operations to lock and unlock, poll for lock and unlock completion, and re-read records in the Validation phase before committing, to check for conflicts (Section 6.2). These operations add overhead and consume core cycles.

To quantify these overheads, we execute three different workloads using the Yahoo! Cloud Serving Benchmark (YCSB) [159]. The first workload performs only write operations (100% WR), the second one performs the same number of read as write operations (50% WR - 50% RD), and the third one performs only read operations (100% RD). Based on previous work [271, 272, 273] we selected five requests from a client to form a transaction. The workloads run on a 4-node cluster, where each node has 48 Xeon E5-2687W cores, and connects to each other with Mellanox ConnectX-4 NICs that perform RDMA over InfiniBand.

The execution time of the workloads, with the contribution of different components, is shown in Figure 6.3 , which was created with the support of graduate student A. Psistakis. The overheads in Table 6.1, from top to bottom, are labeled as *Manage RD/WR Sets*, *Update Version*, *Read Atomicity*, *RD before WR*, and *Conflict Detection*. In the figure, all execution times are normalized to the case of 100% WR. From the data, we see that the software overheads are very significant. Their combined contribution is 59%, 65%, and 71% of the total execution time for the 100% WR, 50% WR - 50% RD, and 100% RD workloads, respectively.

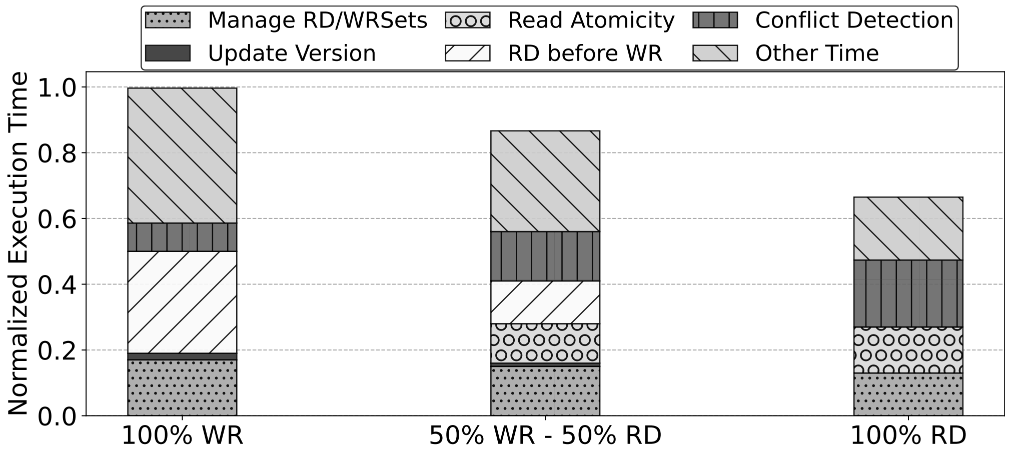


Figure 6.3: Execution time of some workloads running the FaRM protocol, showing the contribution of the main software overheads.

In the 100% WR workload, the highest overheads are reading records before writing them, due to operating at record granularity, and recording records in the Write Set. In the 100% RD workload, the main overheads are: (i) re-reading the version of all records in the Read

Set during validation to check for conflicts, (ii) ensuring the atomicity of read operations, and (iii) recording the Read Set. Finally, for the 50% WR - 50% RD workload, the dominant overheads are a combination of the main overheads of the other two bars.

6.4 DESIGN OF HADES

To improve distributed transactional applications, in this chapter, we introduce new hardware to eliminate some of the high-overhead software operations described above. Then, we develop *HADES*, a new distributed transactional protocol that leverages this hardware to provide low-overhead distributed transactions.

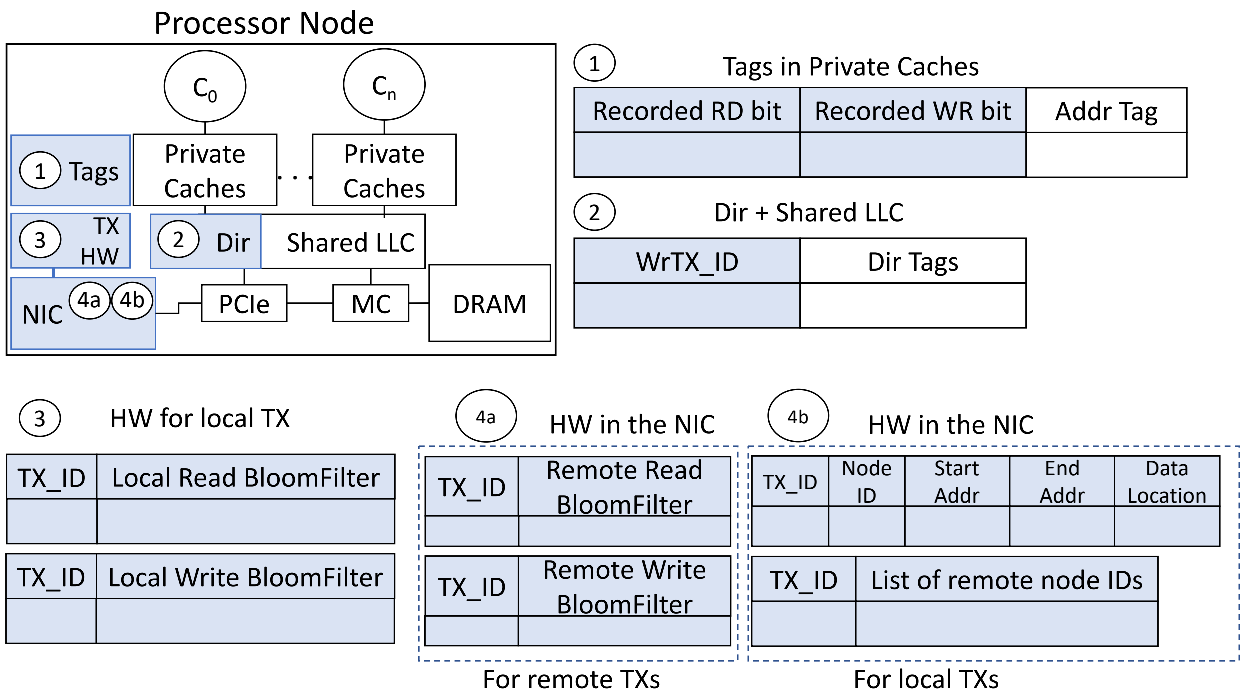


Figure 6.4: Node with the HADES hardware modules painted with a shade.

6.4.1 Proposed Hardware to Minimize the Software Overheads

The right column of Table 6.1 lists our proposed hardware designs to minimize the software overheads. First, to minimize the overhead of managing and checking the Read and Write sets of transactions, HADES uses read and write hardware Bloom Filters (BF). Transparently to the software, these structures (i) encode the addresses of the Read and Write sets, and (ii) help perform conflict detection between transactions.

A transaction owns a pair (read and write) of *local* BFs in the local node and a pair of *remote* BFs in each of the remote nodes from where the transaction accesses data. The pair of local BFs are next to the local directory/LLC and record accesses to the local node memory. A pair of remote BFs exist in the NIC of a remote node, and record accesses by the transaction to the remote node's memory.

HADES does not have the software overhead of updating the versions of records because there are no versions. Further, HADES does not have the software overhead of checking for read atomicity either. The reason is that HADES introduces a hardware mechanism where a transaction can use its BF to partially lock the directory while reading multiple lines, preventing other transactions from concurrently writing the same lines.

HADES does not have the overheads stemming from performing reads and writes at record granularity because its hardware nature enables it to operate at cache line granularity.

Finally, to further reduce overhead, HADES eliminates some of the RDMA and local operations performed by the conventional system. Further, HADES supports some efficient new RDMA operations, including *Intend-to-commit*, *Ack*, and *Validation*, that trigger actions on the receiving NIC. In addition, several of these operations are off-loaded from the core and executed in the NIC. Finally, HADES uses the partial directory-locking hardware mechanism while a transaction commits, preventing other transactions from issuing conflicting accesses.

6.4.2 Choice of Distributed Transactional System

To showcase the impact of HADES, we assume a distributed computer composed of N nodes, each of which has C cores that share memory. Each of the database records has its home in one of the nodes. Hence, when a core accesses a record for the first time in the transaction, it issues a local or a remote access depending on whether the record's home is the local node or a remote one. During the transaction, the record is reused locally. Finally, when the transaction commits, all the remote records that it has updated are written to their home nodes.

Remote records are accessed via RDMA requests that take as argument the range of contiguous addresses of the record. We use one-sided RDMA since it reduces core costs and latency [270]. Local records are accessed with loads and stores. The same is the case for accesses to local copies of remote records.

Both remote and local accesses from a transaction i can conflict with accesses from another transaction j running on the same node (i.e., a local transaction) or on another node (i.e., a remote transaction). Figure 6.5a shows a local (L) and a remote (R) access, and Figure 6.5b shows an L-to-R conflict and an L-to-L conflict.

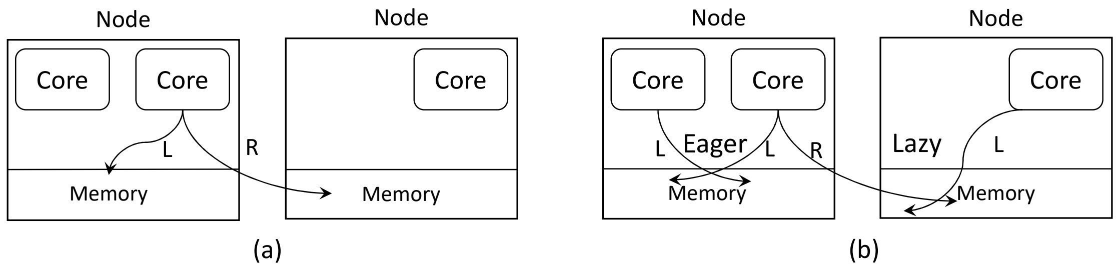


Figure 6.5: Example of transactional conflicts.

We design the HADES' transactional protocol as follows. Conflicts that involve at least one R access of a transaction are detected *lazily* when the first of the two conflicting transactions commits (Figure 6.5b); the transaction that commits first squashes the other one. On the other hand, conflicts where both of the accesses are L are detected *eagerly* as soon as the second access occurs (Figure 6.5b). Moreover, the transaction that issues the second access squashes itself. It will be apparent when we describe the protocol in Section 6.5 that these conflict detection decisions are natural given the hardware envisioned.

6.4.3 Overview of the HADES Hardware

The top left part of Figure 6.4 shows a node with multiple cores, with the added hardware shaded. There are five modules. Module ② is a *Writing-Transaction* ID tag (WrTX_ID) added to each directory/LLC entry. It records the ID of the latest transaction to write that line. Module ① are bits in the private caches that act as filters to avoid accessing WrTX_ID at every access.

Module ③ keeps the *Local Read Bloom Filters* and the *Local Write Bloom Filters* of all the local transactions. They encode the local addresses read and written, respectively, by the local transactions.

Module 4a keeps the *Remote Read Bloom Filters* and the *Remote Write Bloom Filters* of all the remote transactions that have accessed data homed in this node. They encode the local addresses read and written, respectively, by the remote transactions.

Finally, module 4b records the addresses of the remote locations written by the local transactions—together with a pointer (*Data Location* in the figure) to a local buffer that contains the values written. In addition, it records the IDs of the remote nodes that home any data read/written by local transactions. All this information is used when a local transaction commits. Each entry in modules ③, 4a, and 4b is tagged with the ID of the owner transaction (TX_ID).

6.4.4 HADES: Data Buffering and Conflict Detection

Consider a HADES transaction running on Core i of Node x. Data is local if its home is x and remote otherwise. The local data written by the transaction is buffered in the local cache hierarchy (including the shared LLC). The record of local lines read is encoded in a Local read bloom filter (BF). The record of local lines written is encoded in a Local write BF and in Writing-Transaction ID tags in the directory/LLC.

The remote data written by the transaction is buffered in the local NIC. The record of remote lines read and remote lines written by i that are homed in a remote Node y is encoded in a Remote read BF and a Remote write BF, respectively, at the NIC of remote Node y.

We now consider how conflicts between transactions are detected. Recall that L-to-L conflicts are detected eagerly. On a local read by transaction i, the address is checked against the Writing-Transaction ID tag in the directory/LLC. On a local write by transaction i, the address is checked against the Writing-Transaction ID tag in the directory/LLC and against the Local read BFs of all the other local transactions.

Recall that L-to-R conflicts are detected lazily when transaction i commits. At the local node, the local addresses written by i are checked against the Remote read and Remote write BFs in the NIC of all the remote transactions that accessed data homed in x. In addition, at any remote Node y that homes remote data written by i, the following is done: the addresses of the remote data written by i are checked (i) against the Remote read and Remote write BFs of all the other remote transactions in y and (ii) against the Local read and Local write BFs of all the local transactions in y.

6.5 HADES' TRANSACTIONAL PROTOCOL

We propose two versions of the HADES transactional protocol: a hardware-only one and a Hybrid one. The latter replaces the local component of the hardware-only protocol with software, to simplify the hardware design. In the following, we describe both.

6.5.1 Hardware-only HADES Protocol

In our description, we label transactions (sometimes referred to as Cores) as $\{i, j, k, ...\}$ and nodes as $\{x, y, ...\}$. Table 6.2 overviews the protocol followed by Transaction i running on Core i of Node x. We now describe each operation in turn.

Remote Read/Write. Assume that i accesses a remote datum homed in y. In this case, Core i sends an RDMA request to Node y. If this is a read, the addresses of the range

Remote Read/Write by i
* Send request to Node y
* If Read
- Add addresses read to RemoteReadBF$_i$ in NIC of Node y ④ⓐ
- Fetch the data to local node
* If Write
- Add the addresses of partially written lines to RemoteWriteBF$_i$ in NIC of Node y ④ⓐ
- Fetch the partially written lines to local node
- From then on: buffer the updates to all these addresses (not just the partially written ones) in Node x
Local Read/Write by i
* Use tag in the local directory ② to check if another local transaction wrote the line. If so, squash yourself
* If Write
- *Additionally* check the other LocalReadBF$_{j,k,..}$ ③ to see if another local transaction read the line. If so, squash yourself
* If Read
- Add address read to LocalReadBF$_i$ ③
* If Write
- Add address written to LocalWriteBF$_i$ ③
- Update the tag in the local directory ②
Transaction Commit by i. At Local Node x
* i partially locks the local directory or gets squashed
* Detect any conflict on local data between i and a remote transaction
- Find the lines with i's tags in the local directory ② and probe for membership in all RemoteReadBF$_{j,k,..}$ and RemoteWriteBF$_{j,k,..}$ in x's NIC ④ⓐ
- Send squashes to any conflicting remote transactions
* Request the commit of i in remote nodes
- Send *Intend-to-commit* RDMA message to all remote nodes involved in the transaction, passing the addresses written
- Receive *Acks* from all the remote nodes involved in the transaction
- After this, i cannot be squashed anymore
* Clear i's local speculative state
- Find the lines with i's tags in the local directory ② and clear their tag
* Send *Validation* plus *updates* to all the remote nodes involved in the transaction to clear i's remote state
* Unlock local directory and clear LocalReadBF$_i$ and LocalWriteBF$_i$ ③
Transaction Commit by i. At Remote Node y
* NIC receives i's *Intend-to-commit* RDMA message with the addresses written
* Partially lock y's directory for i or squash i
* Detect any conflict on y's local data between i and any transaction local or remote to y
- Take each address written by i homed in y, and check for membership in:
= All other RemoteReadBF$_{j,k,..}$ and RemoteWriteBF$_{j,k,..}$ in y's NIC ④ⓐ and
= All LocalReadBF$_{l,m,..}$ and LocalWriteBF$_{l,m,..}$ in y ③
- Squash all transactions conflicting with i
* Send *Ack* to i
* Receive *Validation* plus *updates* from i
* Push the updates to y's local memory or LLC
* Unlock y's directory for i and clear RemoteReadBF$_i$ and RemoteWriteBF$_i$ ④ⓐ

Table 6.2: Operation of a Transaction i running on a Node x. The references in circles correspond to the hardware modules in Figure 2.2.

of cache lines requested are encoded in the Remote read BF (RemoteReadBF) of i in the NIC of y (Module ④ⓐ of Figure 2.2). Finally, the data is fetched to Node x. If this was a write, a similar process is followed, except that we only need to care about the cache lines that are partially written. Such lines can be found at the beginning and end of the range of addresses written. Such lines are fetched to Node x and their addresses encoded in the

RemoteWriteBF of i in the NIC of y (Module $\textcircled{4a}$). The other lines do not need to be fetched to Node x because they will be overwritten and do not need to be inserted in the BF because they cannot cause squashes. From now on, Node x buffers i's updates to all these addresses.

Local Read/Write. Assume that i accesses a local datum homed in Node x. The hardware accesses the Writing-Transaction ID tag in the directory (Module ② of Figure 2.2) to check if another local transaction has written the line. If so, i is squashed. Note that the filter bits in the private caches (Module ①) are first checked and, if the *Recorded WR* bit is set, there is no need to access the directory because it is guaranteed that the Writing-Transaction ID tag in the directory is set to i. For simplicity, in this discussion, we do not consider Module ①.

On a write, we *additionally* check the LocalReadBF$_{j,k,..}$ of all the other local transactions (Module ③) to see if another local transaction read the line. If so, i is squashed.

If i survives, the action is: on a read, the address read is encoded in LocalReadBF$_i$ (Module ③); on a write, the address written is encoded in LocalWriteBF$_i$ (Module ③) and the Writing-Transaction ID tag is set (Module ②).

Transaction Commit. To commit i, HADES requires several steps in x and some steps in each of the remote nodes from where i accessed remote data $\{y, z...\}$.

Actions in Node x. There are six steps in Node x (Table 6.2):

Step 1. To ensure that commits have a total order, the commit of i starts with i partially locking the local directory. This mechanism is explained in Section 6.5.2 and consists of using the LocalReadBF$_i$ and LocalWriteBF$_i$ (Module ③) to temporarily and selectively block accesses to lines in the directory that are encoded in these bloom filters. This operation prevents other transactions from performing conflicting accesses while i commits. If i fails to lock the directory because another transaction is already locking common lines, i gets squashed.

Step 2. HADES detects any conflict on local data between i and a remote transaction. For this, the hardware takes each of the directory lines whose Writing-Transaction ID tag (Module ②) matches i, and checks them for membership in all the RemoteReadBF$_{j,k,..}$ and RemoteWriteBF$_{j,k,..}$ in the NIC of Node x (Module $\textcircled{4a}$). If a match is detected, a squash is sent to the conflicting remote transaction. Section 6.5.3 shows the hardware structures proposed to easily obtain the directory lines whose tag matches a certain transaction, and those to check for BF membership.

Step 3. HADES requests the commit of i in remote nodes. For this, the local NIC sends an *Intend-to-commit* RDMA message to all remote nodes involved in the transaction, passing the addresses written. Such nodes $\{y, z...\}$ will inititate the commit of i by performing the actions that will be described below. If the operations are successful, the nodes will return an *Ack* to i. When x's NIC has received all *Acks*, it knows that i cannot be squashed anymore.

Before i receives all the *Acks*, i can still receive squashes, which will result in the abort of i and the notification to all the nodes involved in the transaction.

Step 4. Since i is free of squashes, it clears i's local speculative state. Specifically, HADES finds all the lines with i's tags in the local directory (Module $\textcircled{2}$) and clears their tag.

Step 5. The NIC in x sends a *Validation* RDMA message to all remote nodes involved in the transaction, asking them to clear i's remote state. The message includes i's updates to the data homed in the remote node, if any. The receiving nodes clear the RemoteReadBF$_i$ and RemoteWriteBF$_i$ in their NIC (Module $\textcircled{4a}$) and push the updates to their local memory or LLC.

Step 6. As the *Validation* messages are sent, i unlocks the local directory (Section 6.5.2) and clears LocalReadBF$_i$ and LocalWriteBF$_i$ (Module $\textcircled{3}$). All of i's state has disappeared.

Actions in Nodes $\{y, z...\}$*.* In the meantime, remote nodes $\{y, z...\}$ receive the *Intend-to-commit* message from i, with the addresses of the data in those nodes that i wrote, if any. Each of the nodes, say y, performs five steps (Table 6.2):

Step 1. To ensure correctness, the hardware attempts to partially lock y's directory for i. The operation involves using RemoteReadBF$_i$ and RemoteWriteBF$_i$ (Module $\textcircled{4a}$) to temporarily and selectively block access to lines in y's directory that are encoded in these bloom filters. If the hardware fails to lock the directory, a squash is sent to i.

Step 2. HADES detects any conflict on y's local data between i and any transaction local or remote to y. For this, the hardware takes each address written by i that is homed in y and checks for membership in: (i) all other RemoteReadBF$_{j,k,..}$ and RemoteWriteBF$_{j,k,..}$ in y's NIC (Module $\textcircled{4a}$) and (ii) all LocalReadBF$_{l,m,..}$ and LocalWriteBF$_{l,m,..}$ in y (Module $\textcircled{3}$). If a match is detected, a squash is sent to the transaction conflicting with i.

Step 3. y's NIC sends an *Ack* message to i and waits for a *Validation*.

Step 4. On reception of the *Validation* plus the local *updates* from i, HADES clears i's state in y. This involves pushing the updates to y's memory or LLC.

Step 5. y unlocks its directory for i and clears RemoteReadBF$_i$ and RemoteWriteBF$_i$ (Module $\textcircled{4a}$).

Transaction Squash. The previous discussion showed that i is squashed while trying to commit if it fails to partially lock the directory. However, i is also squashed when a conflict with another transaction is detected—either while i is not committing or while i is committing but before it receives all *Acks*.

Conflicts leading to squashes of i can be triggered when another local or a remote transaction conflicts with i on local or remote data. A conflict with another local transaction j on local data is detected eagerly when i attempts to write to an LLC line that has been read or written by j, or i reads an LLC line that has been written by j. A conflict with a local transaction j on remote data or with a remote transaction k on local or remote data is detected when j or k are committing.

Finally, i is also squashed when a line written by i is evicted from the LLC.

6.5.2 Hardware Primitive to Support Atomicity

For correct operation, while transaction i is committing, no other transaction should perform accesses to addresses that conflict with i's accesses. To ensure this capability, HADES introduces a hardware primitive that allows i to *partially lock the directory*, preventing other transactions from performing conflicting accesses. As a transaction i proceeds to commit, it first invokes such primitive.

The idea is to copy the Read and Write BFs of i to a *Locking Buffer* next to the directory (Figure 6.6). Then, every write that accesses the directory/LLC is checked for membership in the Read and Write BFs, while every read is checked for membership in the Write BF. If any of these checks is positive, the access is denied and needs to retry. Otherwise, the access proceeds as usual. Note that these checks are performed in parallel with the directory/LLC tag check.

In the HADES protocol (Section 6.5.1), when transaction i commits, it first locks its local directory with LocalReadBF_i and LocalWriteBF_i, and then the directory in each relevant remote Node y with RemoteReadBF_i and RemoteWriteBF_i. Note that blocking access to the directory/LLC is enough. There is no need to block access to the private cache hierarchies because every first write and first read of a transaction to an address needs to propagate to the directory/LLC to check and potentially set the Writing-Transaction ID tag. The purpose of the *Recorded RD and WR* bits in Module ① of Figure 2.2 is to filter the second and subsequent accesses.

At a given node, multiple transactions can commit at a time if they do not have conflicts. To support this case, as shown in Figure 6.6, our hardware has multiple Locking Buffers to store the bloom filters of multiple transactions. To see how it works, consider the case when transaction i wants to commit in Node x and finds that j is already partially locking the directory.

i's first step is to generate the list of line addresses it wrote. Generating such list is easy. If x is a remote node for i, then the list is available in the just-received *Intend-to-*

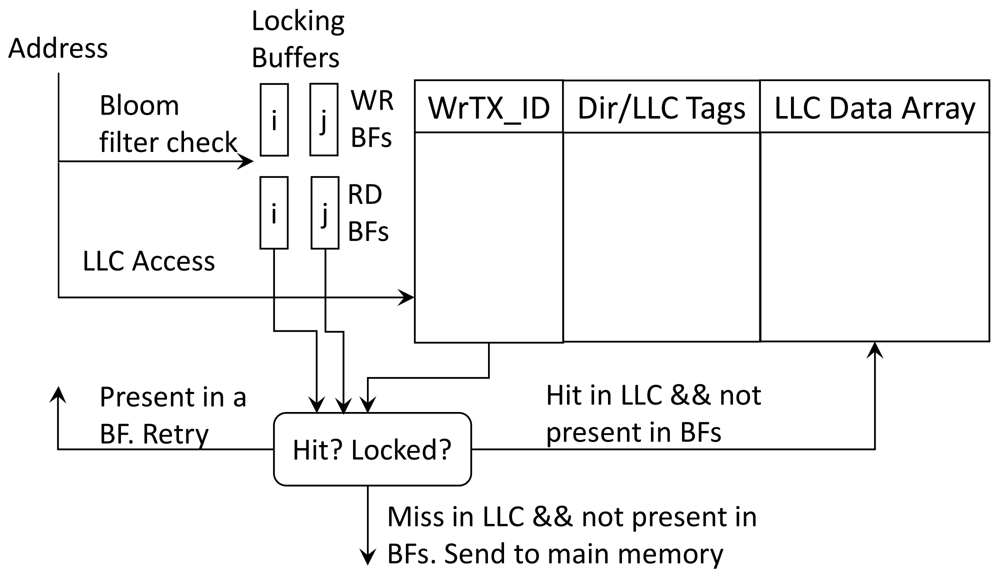


Figure 6.6: Partially locking the directory by transactions i and j using their bloom filters (BFs).

commit message. If x is the local node for i, the list is obtained from the directory/LLC's Writing-Transaction ID tags with the hardware that will be described in Section 6.5.3.

Once the list of write addresses is available, the addresses are checked for membership in the Read and Write BFs of j. If there is a match, the two transactions conflict and i is squashed—they cannot both commit concurrently. Otherwise, i's BFs are loaded into the second buffer of Figure 6.6, effectively adding a second partial lock to the directory. The BFs loaded are RemoteReadBF_i and RemoteWriteBF_i or LocalReadBF_i and LocalWriteBF_i depending on the case.

At the end of the commit, the unlock operation simply clears the Locking Buffer in the local node and in any relevant remote node (Section 6.5.1).

Finally, this primitive is also used by HADES to avoid checking for read atomicity when a transaction performs a read that covers multiple cache lines (Row 3 of Table 6.1). In this case, the hardware hashes the addresses of the multiple lines into one of the read registers of the Locking Buffers. Any concurrent write that attempts to access the directory/LLC while the reads are being performed is delayed.

6.5.3 Other Hardware Primitives

HADES includes two more hardware primitives that operate on BFs: one detects the membership of an address in a BF; the other quickly identifies the set of lines in the directory/LLC that have been written by a given transaction.

Detecting membership is implemented by hashing the address, bit-XORing the set bits in the result with the corresponding BF bits and then comparing the result with zero. If the result is zero, membership is detected (Figure 6.7).

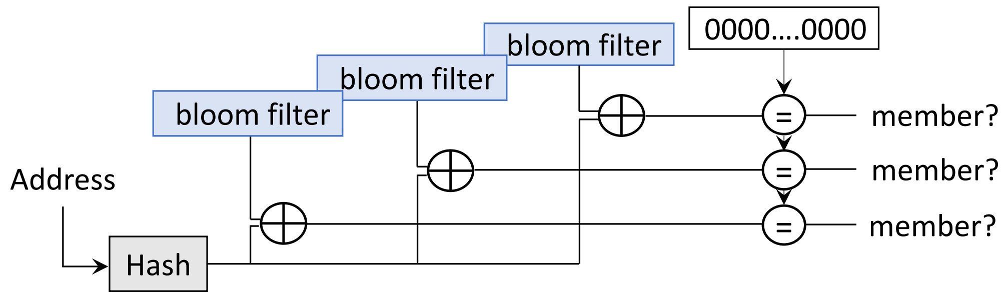


Figure 6.7: Bloom filter checks for address membership.

Identifying the set of lines in the LLC that have been written by a given transaction is needed in three operations. The first one occurs in a transaction squash: all the LLC lines tagged with the transaction's Writing-Transaction ID (WrTX_ID) need to be identified and invalidated. The second operation occurs at the end of a transaction commit: all the LLC lines tagged with the transaction's WrTX_ID need to have their WrTX_ID cleared because they become non-speculative.

The third operation occurs when a local transaction i commits, and needs to check for conflicts against remote transactions {j, k, ...} on local data. Transactions {j, k, ...} have their RemoteReadBF$_{j,k,..}$ and RemoteWriteBF$_{j,k,..}$ in the NIC of the local node (Module 4a). To check for conflicts, one needs to first obtain the set of local cache line addresses written by i. These addresses are then checked for membership in RemoteReadBF$_{j,k,..}$ and RemoteWriteBF$_{j,k,..}$. If any address is found to be a member, a conflict is detected.

Note that the opposite case, where a remote transaction j needs to check for conflicts against a local transaction i is easier: we already have a list of local line addresses written by j; they are included in the *Intend-to-commit* message received from the node where j runs (Table 6.2).

To identify the set of lines in the LLC written by a given transaction, we propose a new organization of the write BF. We logically divide it into two sections: *WrBF1* and *WrBF2*. *WrBF1* is set by hashing addresses using a conventional hash function (e.g., CRC [274, 275]); *WrBF2* is set by taking the LLC index bits of addresses and applying modulo WrBF2 size. This design is shown in Figure 6.8. As a result, each bit of WrBF2 corresponds to a few sets in the LLC (e.g., 4 or 8) and tells whether or not the bloom filter contains an address that

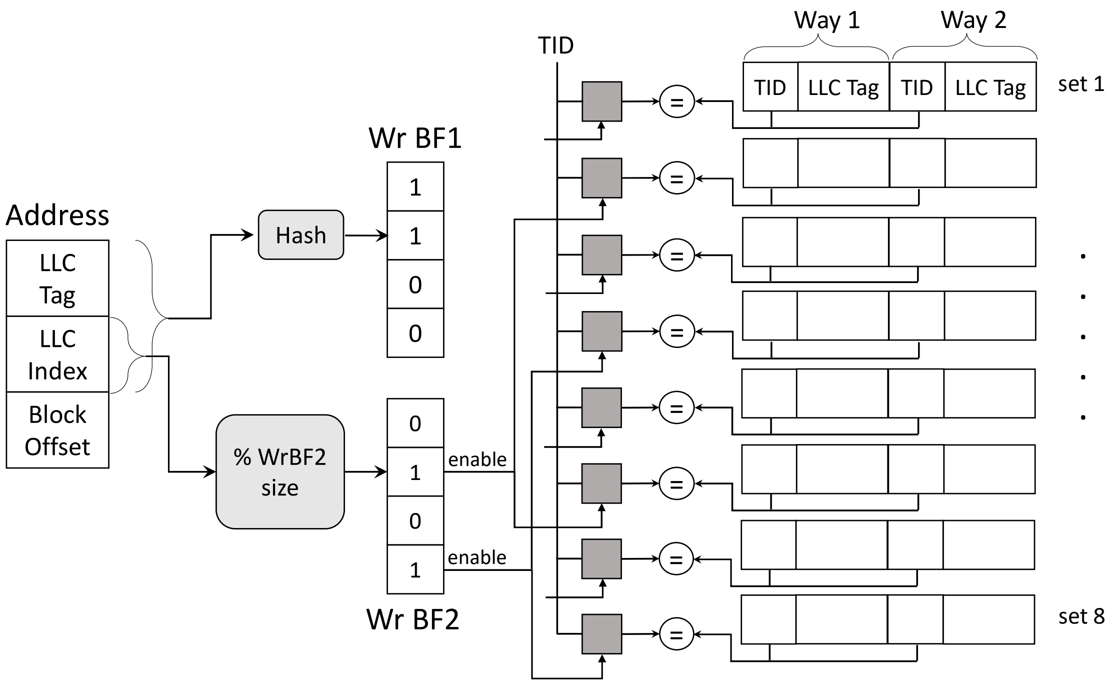


Figure 6.8: New write BF design to quickly identify the set of lines in the LLC written by a given transaction.

maps to such sets. For example, in Figure 6.8, the WrBF2 size is 4, and the LLC has 8 sets. Hence, if bit 2 is set, it means that the bloom filter has at least a line that maps to sets 2 or 6.

With this BF design, address insertion in the BF and membership detection work as usual. However, this design allows fast, parallel detection of the LLC lines tagged with a given WrTX_ID. Indeed, as shown in Figure 6.8, each set bit in WrBF2 enables a group of sets. The enabled sets compare an input WrTX_ID with all the WrTX_ID tags of all their ways. (For space reasons, the figure uses TID rather than WrTX_ID.) Once the matches are found in parallel, retrieving the corresponding address tags does not consume excessive time, since the number of matches is typically modest—around 5 for our platform of 5 nodes. These addresses are the local ones written by the WrTX_ID transaction.

6.5.4 HADES-H: A Hybrid Protocol

We also design a hybrid transactional protocol that combines a subset of the HADES hardware with the baseline software-only FaRM protocol. We call the design HADES-Hybrid (HADES-H).

The idea is to support the local operations in software, like in FaRM, as described in

Section 6.2, and the remote operations in hardware, like in HADES. To support the latter, the NIC has the same hardware structures as in HADES. Specifically, of the hardware structures in Figure 2.2, HADES-H only keeps Modules $\textcircled{4a}$ and $\textcircled{4b}$. In addition, HADES-H retains the hardware primitive that partially locks the directory/LLC (Section 6.5.2), which adds efficiency to the protocol. With this overall design, we reduce the hardware modifications made to the non-NIC part of a node. Data records are augmented as in Figure 6.1. For local read and write operations HADES-H performs them in software at record granularity and records them in the read/write sets like FaRM. No other modification is needed for the execution phase of a transaction. At a high level, the commit follows the operations in Table 2.2. Note, however, that local transactions do not have Local BFs, as Module $\textcircled{3}$ in Figure 2.2 does not exist. Consequently, when a local transaction i attempts to commit, the software passes the list of local lines written by i to the NIC, which builds the LocalWriteBF_i for i. The software then places these BFs in the Locking Buffers to lock the directory. Next, software validates local reads as in FaRM by re-reading their version. Overall, during the commit, operations such as setting the versions of the written records to detect conflicts and re-reading the versions of the previously-read records to ensure read atomicity are performed in software like in FaRM.

6.5.5 Comparison to Other Transactional Memory Designs

HADES builds on Hardware Transactional Memory (HTM) [276] concepts, and uses Bloom filters for conflict detection. However, the design is very different than past work. Consider local transactions first. HADES: (i) allows context switches inside a transaction without aborts; (ii) minimizes the modifications to L1; (iii) uses the LLC as speculative storage to allow for large transactions; (iv) allows more concurrent transactions than cores. Regarding remote transactions, HADES: (i) is the first design to propose hardware for remote transactions in a distributed system, allowing RDMA operations within a transaction without aborting; (ii) proposes new NIC hardware, and (iii) introduces a new protocol.

Two related designs are DrTM [266] and DrTM+R [269]. DrTM locks and fetches all the remote records that a transaction will use. Then, it uses HTM to execute the transaction atomically. Instead, HADES uses OCC to execute the transaction and needs no a priori knowledge of the records in the transaction. DrTM+R extends DrTM to use OCC. It uses a software mechanism like FaRM for conflict detection. In addition, inside the distributed transaction, it uses HTM to guarantee the atomicity of local record reads and writes. Instead, HADES uses hardware-supported local and remote operations in a transaction.

6.6 HADES INTERACTION WITH DISTRIBUTED DATA PERSISTENCY

Distributed applications that use transactional consistency often require fault-tolerance guarantees. Different degrees of fault-tolerance can be achieved by replicating the data across the different nodes of the system, or by persisting the data written by the transaction according to a persistency model. The HADES protocol can support data replication across different nodes, and interact efficiently with the different persistency models.

Supporting Data Replication. For data replication, the HADES protocol needs to propagate the updates of a transaction to all replica nodes that hold a copy of the written records. There can be multiple different options for replicating data in a distributed system, e.g., primary-backup or broadcast-based replication. However, as we discussed in Chapter 5, broadcast-based replication protocols, like Hermes [204], are very efficient and increase system throughput. Thus, we will describe the HADES protocol for replication assuming broadcast-based replication, where each node of the system can act as the Coordinator for a transaction executing on this node (local node), or as a Follower for a transaction executing on another node (remote node). The execution phase of the transaction under replication is the same for HADES as it was described before (Section 6.5). Note, that while a record can be in multiple different replica nodes, during the execution phase it is sufficient to select one of these nodes to perform remote read and write operations. However, on commit, the addresses and the updated data of all written records, both local and remote, need to propagate to the replica nodes of these records. Thus, an *Intend-to-commit* RDMA message is sent by the Coordinator, the local node of the transaction, to all replica nodes. The replica nodes execute the steps of Table 6.2. They need to lock in the directory the addresses that participate in the transaction that intends to commit. Potentially, a replica node might need to create a write bloom filter for this transaction if one was not already present. This can happen in the case that this replica node was not aware of the committing transaction yet, as the remote writes of the transaction targeted a different replica node during the execution phase. Then, conflict detection is performed in all remote nodes and they reply back to the Coordinator with an ACK if the transaction can commit. The replicas wait for a validation message from the Coordinator node to apply the transaction updates in memory and unlock the directory. Since we broadcast all transaction updates to all replica nodes for the written records during commit phase, a conflict with a different transaction will be detected even though during the execution phase a transaction selects one of the replica nodes to perform remote read and write operations. This saves unnecessary network traffic during the execution phase.

Supporting Data Persistency. HADES can work with any of the persistency models of Chapter 5. The Visibility Point of updates for a distributed transaction is when the transaction commits. The durability point of the transaction updates depends on the persistency model. Table 6.3 presents the durability point for every persistency model that we considered in Chapter 5.

Persistency	Durability Point (DP) of an Update
Strict	Log to a persistent storage when the update occurs
Synchronous	At the visibility point of the update, when the transaction commits
Read-Enforced	Before the update is read. This can be after transaction commit time
Scope	Before or at the scope end
Eventual	Sometime in the future

Table 6.3: Durability points of transaction updates for different memory persistency models.

In the *Synchronous* persistency model, the updates of a transaction need to be persisted at transaction commit time. As such, the Coordinator node will receive all ACK messages from the remote nodes and then send the *Validation* message to all remote nodes, as was previously described in Table 6.2. However, a reply to the client that the transaction has completed will be sent only after the Coordinator and the remote nodes have successfully persisted the data. A second ACK_p message from each of the remote nodes is needed to indicate that the updates have successfully persisted in the NVM or secondary storage. In *Read-Enforced* persistency, the transaction is considered completed as soon as the *Validation* message is sent from the Coordinator node to all remote nodes. However, a subsequent transaction will not be able to read any updated data until the data persist. For the case of *Scope* persistency, every update that happened within a Scope needs to persist before the *end-of-scope* annotation completes for a specific Scope ID. Note, that a Scope ID might contain multiple transactions and the updates of all transactions need to persist when the end-of-scope for this Scope ID is reached. Finally, in *Eventual* persistency, the updates of a transaction will persist sometime in the future lazily, while for *Strict* persistency, every transaction update needs to be logged into persistent storage before the transaction execution continues, instead of waiting for the transaction to commit. The latter is a relatively unintuitive model that we include for completeness.

6.6.1 Supporting In-Network Persistency

Persisting data affects the performance of applications. Chapter 5 shows that depending on the persistency model, the throughput and latency of transactions can change significantly.

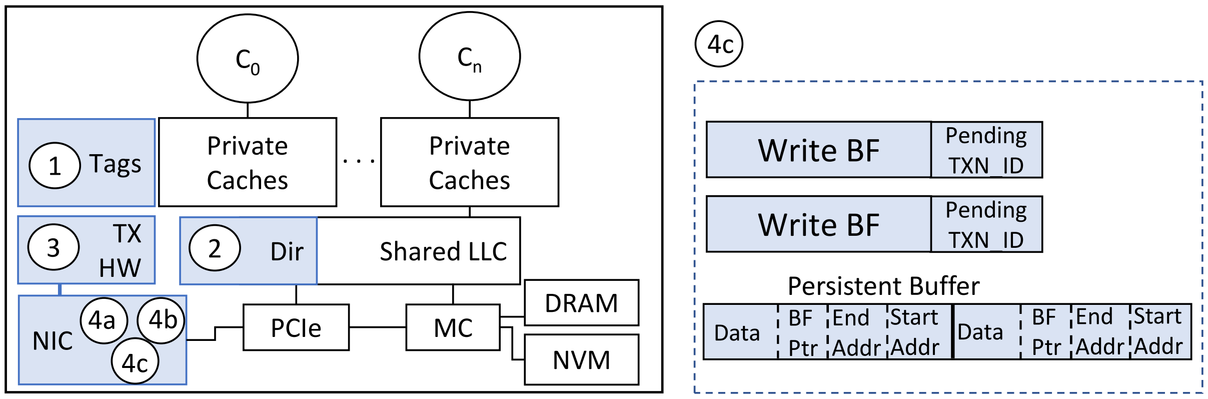


Figure 6.9: Node with the HADES hardware modules painted with a shade.

This is especially the case for persistency models where persisting the data stands on the critical path of execution. However, such models, like Synchronous and Scope persistency, provide the best programmer intuition. The ability to speed-up such models is of great importance. An optimization to enhance the performance of such models is to utilize the network hardware for persisting. Luckily, HADES' hardware can be extended to support persisting in a durable or battery backed buffer in the NIC and avoid waiting for the data to persist in NVM. This optimization can help to avoid the increased latency and limited bandwidth of NVM that can become a bottleneck for distributed transactions.

To achieve higher throughput for persistency, we propose to enhance the NICs with an extra hardware structure that can persist data in the NIC. Figure 6.9 shows the extra hardware structure that we add in the NIC, namely Module 4c. This module consists of two parts. The first is a Persistent Buffer that holds the updated values of the written data from a transaction. This buffer can be NVM, or volatile memory (SRAM or DRAM) that is battery backed. The buffer contains the data that are to be flushed to the NVM or secondary storage, the start and end addresses of the updates, and a pointer to the bloom filter that contains the signature of these updates. The second structure is the bloom filter that has the write signature of the updates that persisted in the NIC for this transaction. This is required to guarantee the correct ordering of updates to non-volatile storage. Thus, when a node sends the updates to a remote node with the *Intend-to-commit* message, an entry in the Persistent Buffer and a write bloom filter are filled for the committing transaction in Module 4c until the updates are flushed to NVM.

Commit operation with in-network persist support. The commit operation at a local and a remote node needs to account for the buffered writes at the NIC. Note, that while we are buffering the data at the NIC, we should not alter the order at which the data is flushed to NVM, when multiple transactions operate on the same data. This means that if a transaction has buffered data in the NIC and a subsequent transaction performs a write to the same data, the latter must not send any updates to NVM before the first transaction. Consider the operations of Table 6.2 for the case that a transaction i commits at remote node y. y's NIC receives the *Intend-to-commit* message for transaction i and partially locks the directory or squashes the transaction. Then, it checks for any conflict between i and any transaction local or remote to y. On top of that, it needs to check if there are any buffered updates at y's NIC Persist Buffer that conflict with the updates of the committing transaction. If any write bloom filter in Module $\textcircled{4c}$ indicates a conflict and the Persist Buffer has free space, then the updates of i will be buffered in an entry of the Persist Buffer and the write bloom filter of the transaction will be kept in Module $\textcircled{4c}$. That is because the Persist Buffer will flush the updates to NVM in order. So, no transaction needs to be squashed. However, if there is no space, then the committing transaction needs to be squashed to maintain the correct ordering of persist operations in NVM. On the other hand, if no conflict is detected between the committing transaction i and the write bloom filters of Module $\textcircled{4c}$, then the transaction can commit. The first option for committing is to buffer the updates in the NIC, when there is enough space in the Persist Buffer. In the case where the Persist Buffer does not have enough space to hold the data, we can buffer the data in the LLC and only keep the write BF and the start and end addresses in the Persist Buffer. Then a flush command will push the data to the NVM later. If there is no space in the Persist Buffer even for the metadata, the transaction is squashed.

Consider the case where transaction i commits at local node x. i needs to partially lock the directory and check for conflicts between i and any remote transaction at x. Additionally, it needs to check if the NIC has any buffered updates in the Persist Buffer, and decide if there is a conflict with them or not. Similar to before, if a conflict is detected and the Persist Buffer has no free space, transaction i is squashed. Otherwise, if there is free space in the Persist Buffer, i's updates can persist there until they are flushed to NVM. If no conflict is detected, the updates can be persisted in the Persist Buffer if it has free space, or buffer the data in LLC and keep the write BF in the NIC with the metadata. If there is no space for the metadata, the transaction is squashed.

The Persist Buffer accelerates persisting the data in the NIC instead of the slower NVM. The bloom filter support of HADES is utilized to dictate the ordering of updates between the buffer in the NIC and the NVM, so that no udpates for the same address are re-ordered between local or remote transactions and the NIC's Persist Buffer. The updates from the Persist Buffer are lazily flushed into the NVM. When all updates of a transaction are flushed to persistent storage, the data and the transaction's write bloom filter in Module $\textcircled{4c}$ are cleared.

6.7 EVALUATION METHODOLOGY

Modeled Architecture. We model the architecture of a distributed system with five servers. On each server, we have one client per core executing transactions. The servers utilize 64 GBs of main memory and they have a Network Interface Card (NIC) that supports Remote Direct Memory Access (RDMA), which enables a server to access the remote memory of other servers. The architecture parameters are shown in Table 6.4. Each core is an out-of-order core with private L1 and L2 caches, and a shared LLC. For HADES we include the bloom filter support at the LLC and the NIC and extend the directory and private caches' tags to support conflict detection of transactions.

We use RDMA to support low-latency data transfers across nodes without involving the remote processor and augment it with the operations required by HADES. These operations include support for remote read and write operations that need to be recorded in the HW structures of the NIC, support for the "Intention to commit", Ack and Val messages of our protocol, and squashing transactions on a conflict. We model a high-end NIC with a bandwidth of 200Gb/s [199], and up to 400 Queue Pairs [229] for scheduling messages. Further, we model a 2μs round-trip latency for a message between two NICs [198, 199, 265, 277]. Moreover, we model the latency of adding elements to the bloom filters, checking for conflicts and locking an address range to the directory using the bloom filters. Finally, when we evaluate the interaction of HADES with the different persistency models, we have 16GB of DRAM and 64GB of NVM, while the NIC has a Persist Buffer of 512KB.

Modeling Approach. For our simulations, we use the SST simulator [61], Pin [155], and the DRAMSim2 memory simulator [63]. With Pin, we collect instruction traces for 25 cores processing read and write client requests for all our benchmarks. Traces have no timing information. Then, we take these traces and simulate 25 cores in our distributed architecture. Timing is dynamically determined by the simulator. We inserted in the simulator all the protocol messages required for the execution, validation and commit phases of transactions. All the records accessed by the client requests are statically distributed across the five servers for all of our benchmarks. Thus, the probability of a request targeting remote data is 80%.

Configurations and applications. For our experiments we compare 3 different configurations: (a) **Baseline:** This the a SW only approach for distributed transactions based on FaRM [254, 268, 269], (b) **HADES-H** that is utilizing the SW based approach for local transaction operations and the HW based approach for remote operations, and (c) **HADES**

Server Architecture Parameters	
Servers; Clients	5 servers; 5 clients/server.
Multicore chip	5 out-of-order cores per server, 6-issue, 2GHz
Ld-St queue; ROB	92 entries; 192 entries
L1 cache	64KB, 8-way, 2 cycles round trip latency (RT)
L2 cache	512KB, 8-way, 12 cycles RT
LLC cache	4MB/core, 16-way, 40 cycles RT
LLC Bloom filters	Read: 1024bits, Write: 512bits with CRC hashing plus 4096bits with cache indexing hashing 10 read, 10 write bloom filter per node (7KB storage)
Find LLC TX writes	80 - 120 cycles
CRC hash function	2-cycle latency; Area: $1.9 * 10^{-3} mm^2$; Dyn. energy: $0.98pJ$; Leak. power: $0.1mW$
Network Parameters	
Network latency	$2\mu s$ RT NIC-to-NIC RDMA latency
Network Bandwidth	200Gb/s
Queue Pairs	Up to 400
NIC Bloom filters	1024 bits for Read and Write bloom filters 160 read, 160 write bloom filters/NIC (40KB storage)
Other NIC hardware	Total size of structures 4b in Figure 2.2: 1.2KB storage
Persist Buffer	Size: 512KB, Latency: 12 cycles
Per-Server Main-Memory Parameters	
DRAM Capacity	64GB w/o NVM and 16GB w/ NVM
NVM Capacity	64GB
Channels, Banks	DRAM: 4, 8; NVM: 2, 8
Latency	DRAM: 100ns read/write RT NVM: 140ns read, 400ns write RT
Freq; Bus width	1GHz DDR; 64 bits per channel

Table 6.4: Architectural parameters used for evaluation.

that is utilizing the proposed HW based approach both for local and remote requests. We use three popular distributed transactional applications and four key-value stores. Specifically, we use the widely used **TPC-C** [278]. TPC-C is an OLTP benchmark that simulates an order processing application. We fill the TPC-C warehouses with 10 million items. TPC-C is write intensive and has many read and write record accesses per transaction at a fine granularity. We used **TATP** [279], an OLTP benchmark that simulates a telecommunication database with 1 million subscribers. TATP has 80% read and 20% write requests and a small number of requests per transaction. We also used **Smallbank** [280, 281], a write intensive OLTP benchmark (46% write requests) that simulates bank account transactions on 5 million accounts.

We also used multiple commonly used key-value stores such as **HashTable (HT)**, **Map**, **B-Tree** [231] and **B+Tree** [232]. We evaluate them with Yahoo! Cloud Serving Benchmark

(YCSB) [159] running write intensive workload-A (wA) with 50% write, 50% read requests, and read intensive workload-B (wB) with 5% write and 95% read requests. We fill the key-value stores with 4 million keys. Similarly to previous work [271, 272, 273] we selected transactions to be 5 client requests.

In our experiments, we warm up the architectural state by running 1 billion instructions before simulating a total of 25 billion instructions.

6.8 EVALUATION

6.8.1 Improving Transaction Throughput

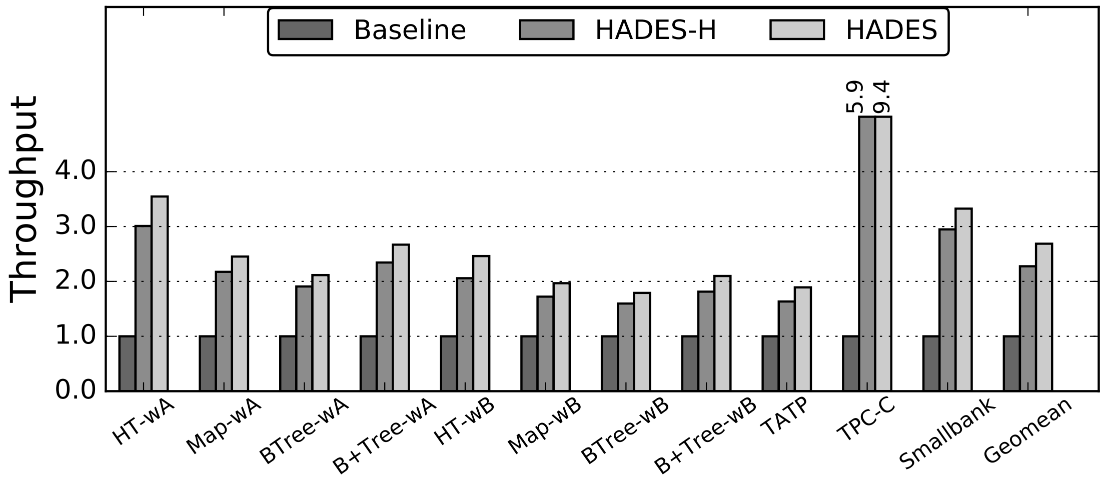


Figure 6.10: Throughput of HADES normalized to the state-of-the-art software-based approach (Baseline).

In Figure 6.10 we evaluate the throughput benefit of HADES-H and HADES. In the figure, the throughput for each benchmark is normalized to that of Baseline. Both HADES-H and HADES can achieve great speedups over the state-of-the-art software-based implementation. On average, we see that they achieve 2.3× and 2.7× higher throughput, respectively. For the TPC-C benchmark, our solutions can obtain even higher speedup. This is because TPC-C is characterized by many small client read and write requests per transaction (`~13.5` requests), and only a few benchmark instructions are executed to serve these requests. This exacerbates the added software overheads for such requests. Specifically, the cost of managing the Read and Write sets, as well as ensuring read atomicity and operating at record granularity introduces significant performance overheads. For the key-value stores running with YCSB workloads, we observe that we can achieve better performance for write intensive workload-A, than the read intensive workload-B. This is because writes have extra overheads of fetching the record version before performing the write and updating the record version before the transaction commits. Moreover, read-only transactions, in the case of Baseline, do not need to lock any records before the commit time. Such transactions validate the read set for conflicts by re-reading the version during the Validation Phase and if no conflict is detected, the transaction can safely commit. This saves a network round-trip for locking remote records, and the execution time for locking local records. We observe the same performance behavior for TATP (read-intensive) and Smallbank (write-intensive) workloads.

6.8.2 Reducing Transaction Latency

We show the average latency of the transactions for the different benchmarks in Figure 6.11. Similar to Figure 6.10, we normalize the latency numbers to that of the state-of-the-art software-based approach (Baseline). We break down the transaction latency into three parts: the time it takes for the Execution, the Validation, and the Commit phases. Compared to the Baseline, HADES-H and HADES achieve 54% and 60% lower latency, on average. The Execution Phase accounts for most of the total transaction time. HADES is able to mitigate most of the overheads during this phase. In HADES, there is no need to manage Read and Write sets, to validate the atomicity of read operations, and also, we do not need to operate at record granularity. As a result, we avoid all the redundant reads and writes of software, as well as all the checking and bookkeeping overheads that were outlined in the first, third and fourth rows of Table 6.1.

The Validation Phase is the second dominant factor for the total transaction time. HADES spends less time in this phase due to fact that the bloom filters perform fast conflict detection, as opposed to the Baseline that needs to re-read the record versions. Moreover, in HADES, we do not serialize locking the written records with re-reading record versions. Our "Intention to commit" message is broadcasted to the participating nodes of a transaction simultaneously, which improves the concurrency of the transaction validation.

Finally, Baseline spends some time in Commit Phase to update the record version, apply the updates and unlock the records. Instead, HADES does not include record versions and offloads the operations of (i) sending the updates to the remote nodes, (ii) making the local updates non-speculative, and (iii) unlocking the bloom filters to the NIC and the LLC hardware.

We also evaluate the tail latency of transactions, and show the 95th percentile latency

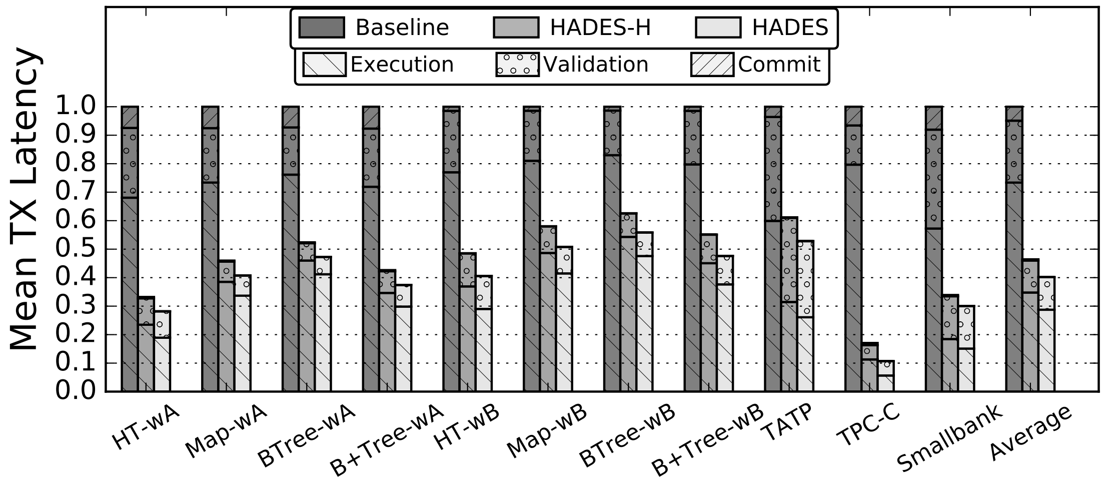


Figure 6.11: Mean transaction latency normalized to Baseline.

in Figure 6.12. As shown in Figure 6.12, the performance trend is similar to mean latency (Figure 6.11). HADES achieves slightly more improvement in tail latency than in mean latency. The reason is that HADES offloads operations to the NIC and has less operations for locking and unlocking record versions. This decreased memory and network traffic results in less interference during the execution of transactions, which achieves reduced tail latency for HADES, compared to Baseline and HADES-H.

6.8.3 Characterization of the LLC and the Bloom Filters

We performed two more analyses to further assess our design. We first check whether the LLC can accommodate the transactions we are executing, or whether the evictions of written cache lines are causing a transaction to abort. We run all our benchmarks and force every client request to target local nodes. This setting maximizes the pressure we put on the LLC. We modified the LRU replacement policy to avoid evicting speculatively written cache lines from a transaction, if non-speculative cache lines could be evicted instead. We find that 0.1% of the executed transactions need to be aborted, on average, because of an LLC eviction. At most, TPC-C aborts 0.7% of the transactions when all the requests are local. This is a very small percentage for a worst case scenario, and it is even more negligible when transactions involve remote nodes.

Second, we also evaluate the effectiveness of the bloom filters used in HADES. In our experiments, we find that the bloom filters have an average false positive probability of 0.02% for the case of HADES-H and 0.04% for the case of HADES. This is because the

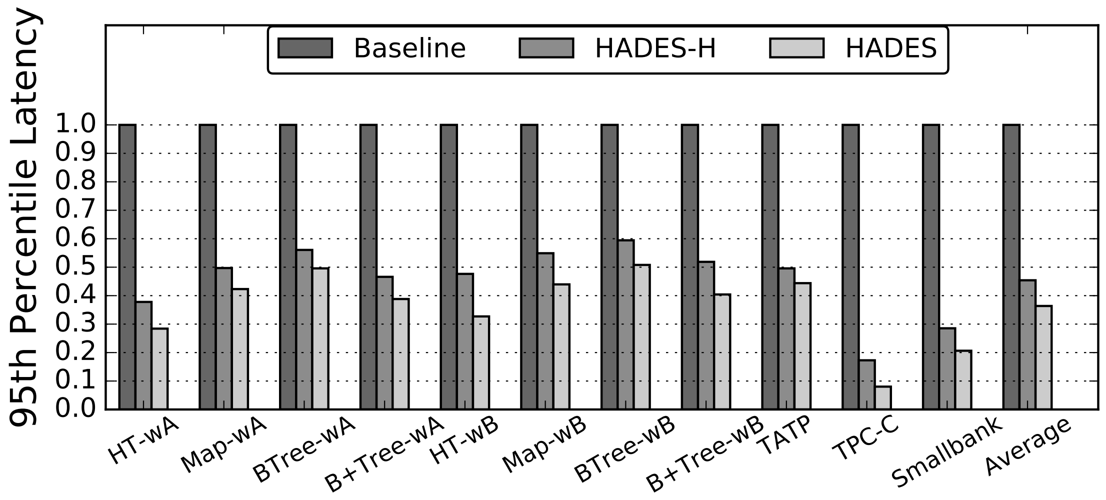


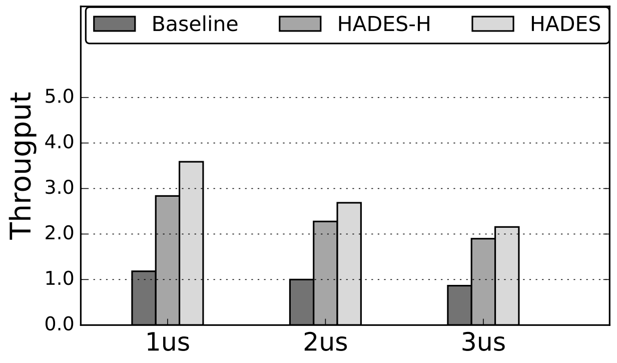
Figure 6.12: Tail (95th percentile) latency normalized to Baseline.

transactions do not read or write multiple cache lines in a concentrated manner. At most 76 cache lines are read and 40 are written by a transaction in our benchmarks, and these lines are spread across the nodes of the system. We find that the average percentage of bits set across all the available bloom filters is 0.32% for HADES-H and 0.37% for HADES. To further assess the effectiveness of our bloom filters, we consider a scenario where all requests target a specific node. This leads to ~13% bits set for a 1-Kbit bloom filter resulting in a false positive probability of ~2%. Thus, we conclude that the bloom filters are a very efficient solution for conflict detection in hardware-assisted transactional systems.

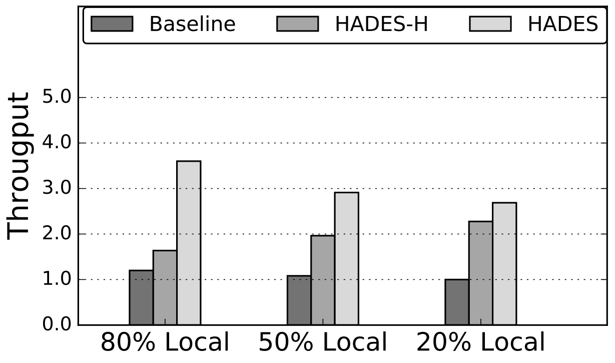
Overall, we find that, for the evaluated setup, all the configurations have a negligible rate of aborting transactions for all workloads, and that the hardware structures introduced by HADES do not add much hardware cost and performance overhead.

6.8.4 Sensitivity Analysis for HADES

Finally, we perform two sensitivity analyses. First, we examine the sensitivity of HADES and HADES-H to different network latencies. Figure 6.13a presents the average throughput across all benchmarks normalized to the Baseline with 2μs network latency. HADES achieves higher speedup as the network latency decreases (3.6× for 1μs latency). That is because the software overheads of the Baseline are becoming a more serious bottleneck when the network round-trip latency decreases. Our design eliminates these software overheads. As such, network latency has more impact on HADES and HADES-H, compared to the Baseline, in which the execution time is dominated by the software operations.



(a)



(b)

Figure 6.13: Throughput for: (a) different network latencies normalized to the 2μs Baseline and (b) different ratio of local to total requests normalized to the 20% local Baseline configuration.

Next, we perform a sensitivity analysis on the ratio between requests that target a local or remote nodes. The bars show the average throughput across all benchmarks and are normalized to the case of 20% Local Baseline (20% of the requests in transactions target a local node), which is the configuration we used in all the previous experiments. As the number of local requests increases, Baseline and HADES achieve higher throughput. Baseline has less impact because the software overheads are the major bottleneck, while HADES achieves greater speedup. However, this is not the case for HADES-H. Its performance decreases as we have more requests targeting the Local node. This is because HADES-H is using the software-based approach for local operations, which introduces dramatic software overheads. As a result, HADES-H obtains more performance benefits for transactions that involve a large number of remote requests. Overall, we observe that our solutions always perform better than Baseline, making us believe that our proposed hardware-assisted transactional system is a promising approach to accelerate distributed transactions in data centers.

6.8.5 Performance of HADES with Persistency

We combine HADES with five different persistency models to assess the performance impact of persistency.

We compare two different configurations, one that does not use the Persistent Buffer optimization that was described in Section 6.6 and one that does utilize the NIC hardware to persist data. Figure 6.14 presents performance across the different models. Each bar represents the geomean across all our benchmarks, but for this analysis we exclude benchmarks

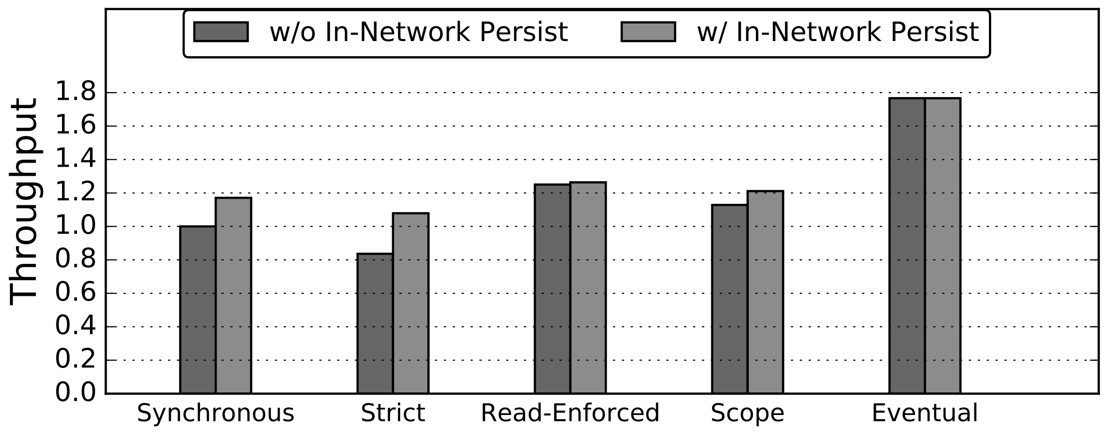


Figure 6.14: Throughput of HADES for different persistency models with and without in-network persistency support, normalized to the case of Synchronous persistency without in-network persist support.

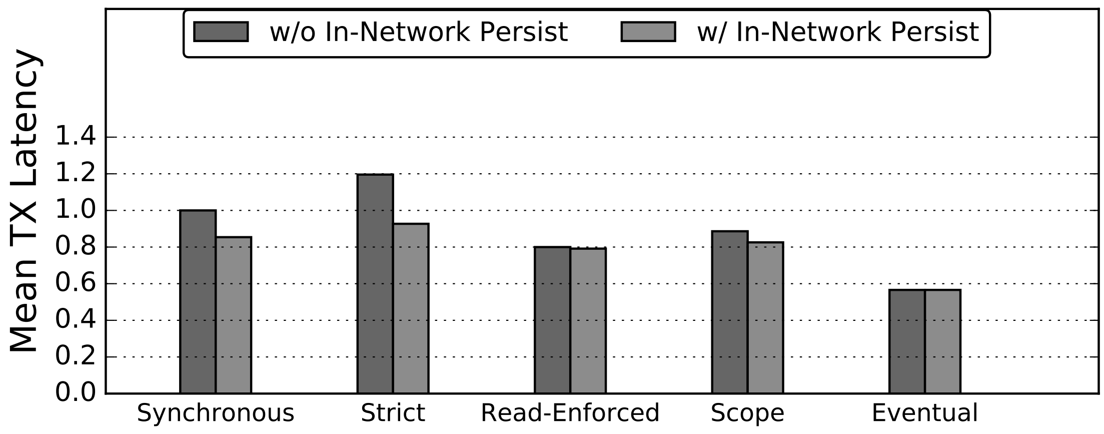


Figure 6.15: Mean transaction latency of HADES for different persistency models with and without in-network persistency support, normalized to the case of Synchronous persistency without in-network persist.

that use workload-B, because it has only a small percentage of write operations (5%). All the results are normalized to the case of Synchronous persistency with no In-Network Persist support. It is obvious from the figure that the choice of persistency model can have a huge impact on the throughput of the applications. For example, Eventual persistency delivers on average 77% higher throughput than Synchronous persistency. A second observation is that persisting in-network with the use of the Persist Buffer can improve the performance of the stricter persistency models, namely Synchronous, Strict and Scope persistency. These models put the persist operations on the critical path of the transaction execution. Thus, avoiding the increased NVM latency improves performance. For the case of Synchronous persistency, throughput increases by 17.1% and achieves performance that is comparable to that of the more relaxed Read-Enforced persistency. A similar behavior is observed for the case of Scope persistency which achieves 21% higher throughput than Synchronous when we use in-network persistency as opposed to 12.8% when in-network persist is not used. The performance benefit of Scope persistency is less than that we observe for Synchronous, since we selected the *end-of-scope* to be after two consecutive transactions. Strict persistency sees the highest performance impact from the persist buffer, since it is the model that stalls for every write operation to persist before continuing execution. When we persist in the NIC it achieves higher throughput than Synchronous persistency with no in-network persistency support. The more relaxed persistency models, Read-Enforced and Eventual, do not have a great impact from the optimization.

Figure 6.15 shows the mean latency of transactions under different persistency models. The figure is organized as before. We see that mean latency is inversely correlated to throughput. Stricter persistency models see a bigger improvement from persisting in the NIC, while relaxed persistency models achieve high performance and are impacted less by the NIC's Persist Buffer.

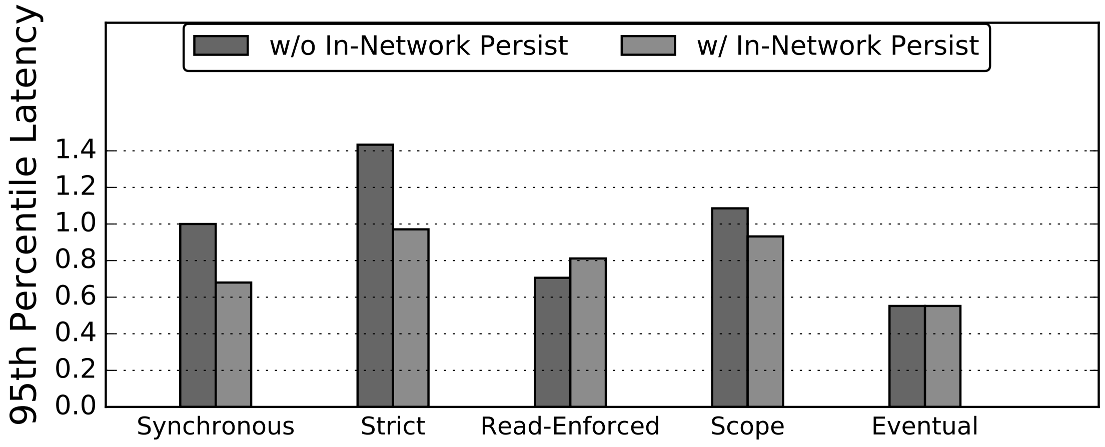


Figure 6.16: 95th percentile transaction latency of HADES for different persistency models with and without in-network persistency support, normalized to the case of Synchronous without in-network persist.

Figure 6.16 shows the 95th percentile transaction latency. Interestingly, the tail latency of our applications does not have the exact same behavior as the mean latency. While utilizing the Persist Buffer in the NIC decreases the mean and tail latency in most cases, we find that it can increase the tail latency for some benchmarks with Read-Enforced persistency.

The reason is the increased traffic of the NVM memory. For the case of Read-Enforced persistency we saw earlier a similar behavior (Chapter 5). Not waiting for the writes to persist until a subsequent read needs to access the data increases the traffic that targets NVM. Thus, when a read tries to access data that has not yet persisted it might need to wait for a larger amount of time. In our evaluation we find that Read-Enforced persistency stalls 15% of the total execution time waiting for data to persist before satisfying a read request. When the Persist Buffer is utilized, some of the reads that conflict with a previous write are satisfied faster since they persist in the NIC. Thus, more traffic is introduced to the NVM that can hurt tail latency in the unlucky case that a read needs to wait a write which is targeting the busy NVM.

Persist Buffer Characterization We perform a sensitivity analysis to further assess the capability of persisting in the NIC for the case of HADES. We change the size of the Persist Buffer in the NIC and measure the total number of requests that are able to persist in the NIC across the different persistency models.

Persist Buffer Size	32KB	512KB	1MB
Synchronous	90.2%	99.6%	99.8%
Strict	98.6%	99.9%	99.9%
Read-Enforced	50.3%	67.3%	74.1%
Scope	83.5%	99.3%	99.7%
Eventual	99.9%	99.9%	99.9%

Table 6.5: Percentage of persist operations that are satisfied by the Persist Buffer in the NIC for different Persist Buffer sizes.

We select three different sizes for the Persist Buffer, 32KB, 512KB and 1MB. Table 6.5 shows the percentage of persist requests that were able to persist in the NIC for the different persistency models. These percentages are the average across our benchmarks. We see that as we increase the size of the Persist Buffer, a larger percent of requests are able to persist in the NIC. For stricter persistency models such as Synchronous and Strict, even a small Persist Buffer of 32KB was able to satisfy most of the requests. That is because such models stall execution until the persist operations complete. On the other hand, Read-Enforced persistency does not stall execution waiting for write operations to persist. As a result, it introduces a lot more writes to the system that cannot be satisfied by the NIC if the Persist Buffer is not large enough.

In all cases the number of requests that can be persist in the NIC is quite high and does

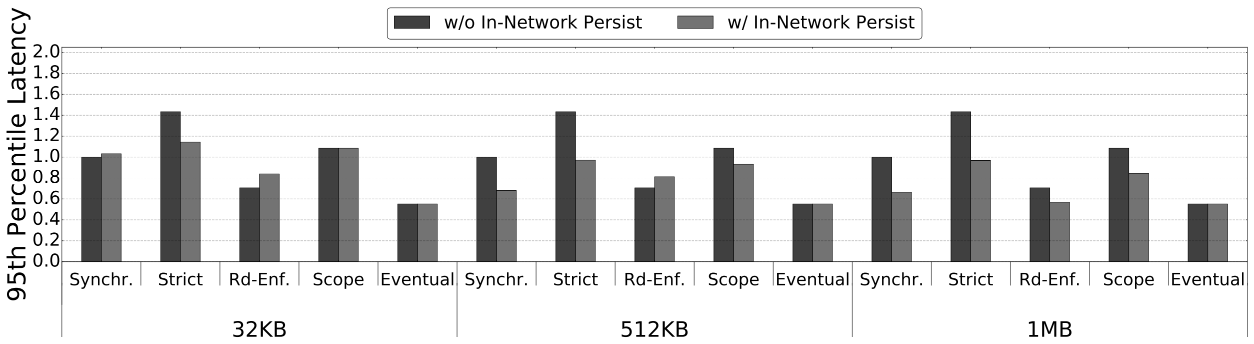


Figure 6.17: Normalized 95th percentile tail latency for each persistency model for different sizes of the Persist Buffer. For each size of the Persist Buffer the results are normalized to Synchronous persistency without In-Network Persist support.

not change much between the different Persist Buffer sizes to affect throughput and mean transaction latency considerably. However, the Persist Buffer size affects the tail latency of applications. In Figure 6.17 we show the 95th percentile tail latency for the different persistency models as we change the size of the Persist Buffer. The bars represent the geomean across all benchmarks and for each Persist Buffer size they are normalized to the case of Synchronous persistency. From the Figure, we see that as we increase the size of the Persist Buffer the tail latency decreases for all persistency models, except Eventual Persistency that never stalls waiting for a write which remains the same. We saw in Table 6.5 that Read-Enforced persistency is affected the most from the Persist Buffer size. As its size increases the tail latency is decreasing and for 1MB it is better than the case where we have no in-network persistency support. That is because more reads that conflict with a previous write only need to wait until data persists in the NIC, instead of the slower NVM. For the stricter persistency models, namely Synchronous, Strict and Scope, tail latency also improves as more requests can persist in the NIC. For example, for the case of Synchronous persistency, tail latency decreases by 37% when we increase the Persist Buffer from 32KB to 1MB.

Overall, we observe that even a small Persistent Buffer in the NIC is able to increase the performance of persistency models considerably, with only a few additional hardware changes in HADES.

6.9 RELATED WORK

Software Optimizations for Distributed Transactions. To optimize the performance of distributed transactions, the system community has developed many software-based systems recently, based on modern hardware techniques [186, 245, 254, 264, 265, 282]. Typical examples include FaRM [254] and its extension [245], which utilize the RDMA primitives to accelerate the remote data access in distributed transactions. Most recently, Opacity [270] advanced the FaRM implementation by enabling strict searializability for all transactions using global timestamps. However, all these schemes are software based and limited by the data access protocols provided by the existing network and memory devices. They suffer from significant performance overhead as presented in Section 6.3. In this chapter, we conduct a characterization study of the entire distributed transactional stack, quantify its overhead sources, and further develop a hardware-based scheme to accelerate the concurrent execution of distributed transactions.

Distributed Transactional Protocol Development. Researchers also reexamined distributed transactional protocols by exploring advanced hardware features in the transactional systems [255, 265, 268, 269, 283, 284, 285, 286]. For instance, DrTM+R [269] used both RDMA and Hardware Transactional Memory (HTM) to obtain the performance benefits from advanced hardware features by leveraging the strong atomicity of HTM and fast remote data access of RDMA. DrTM-H [268] used both one-sided and two-sided RDMA operations in different phases of the distributed transactional protocol. FaSST [265] replaced the one-sided RDMA with fast RPCs using two-sided unreliable datagrams, based on the observation that packet drops happen extremely rarely on modern RDMA networks. PRISM [287] proposed four new RDMA primitives for distributed systems without modifying the underlying hardware. Different from these works, we rethink the distributed transactional protocols by offloading the transactional operations into the NICs. Our evaluation shows that HADES can significantly eliminate the software overhead of existing transactional procotols. HADES also develops three new RDMA operations for efficient transaction executions in distributed environment.

Network Support for Distributed Transactions. Recently, researchers proposed to exploit the compute capability of network devices (i.e., SmartNIC) to accelerate conventional host-based distributed systems and services [288, 289], such as key-value stores [263], RPC [290], remote storage accesses [291, 292], processor-NIC integration [226], network functions [293, 294], and distributed file systems [295]. Unlike these works, HADES focuses on accelerating distributed transactions with SmartNICs. HADES shares the same motivation as the most recent work Xenic [267] that takes advantages of the SmartNIC to reduce the data lookup overhead for distributed transactions, however, HADES takes a more holistic approach by rethinking the design and implementation of distributed transactional protocols with the in-network computing paradigm, and addressing the fundamental issues with the data coherence between the host and NIC.

Moreover, researchers have proposed hardware support to accelerate persisting the data at the network switches [296]. HADES is utilizing the SmartNIC of each node to persist the data and is concerned with the ordering of persists as well as conflict detection between the executing transactions.

6.10 CONCLUSION

Recent hardware trends have exacerbated the software overheads in distributed transactional applications. To address this problem, this chapter analyzed the sources of software overhead in these applications and proposed new hardware structures to eliminate them. The hardware includes bloom filters for read and write set management, and SmartNIC support for efficient remote communication. The chapter then introduced *HADES*, a new distributed transactional protocol that leverages this hardware to provide low-overhead distributed transactions. We find that our transactions execute on average 2.7× faster on HADES and 2.3× faster on a hybrid hardware-software implementation of HADES than on the state-of-the-art distributed transactional system called FaRM system. Finally, this chapter analyzed how HADES interacts with the different persistency models, and proposed hardware support in the NIC to accelerate persistency operations.

CHAPTER 7: CONCLUSIONS

The emergence of non-volatile memories shows promise to drastically alter the landscape of single- and multi-node computer systems. NVM has appealing characteristics and can be used as part of main memory for extra capacity, as a fast storage device or as a cache replacement. However, for future systems to be able to take advantage of the NVM properties, new architectures should be designed that allow for NVM integration. To enable NVM integration, this thesis proposes architectural techniques that exploit the NVM capabilities.

This thesis first started by presenting how NVM can be used in single-node systems. It introduced PageSeer, which manages data placement in hybrid DRAM-NVM main memory systems by proactively swapping data between the two memories. PageSeer takes advantage of the NVM capacity and minimizes the penalty of the additional NVM access latency. Next, it described Cloak, a method that hides the increased read latency of NVM and enables using NVM as an LLC replacement. Cloak paves the way for larger LLCs with the use of NVM. Third, it introduced P-INSPECT, which adds hardware support in the processor to facilitate programmable NVM frameworks. Easy-to-use programming frameworks are vital for the success of NVM. P-INSPECT tackles the performance overheads of *Persistence by Reachability* frameworks to make them an attractive choice for NVM application development.

Besides single-node systems, this thesis examined the use of NVM in distributed architectures with fast networks and powerful NICs. The arrival of NVMs in conjunction with the technological advances of networks can help applications attain high performance and data persistence. This thesis introduced the concept of Distributed Data Persistency, which binds persistency and consistency models in distributed systems and developed low latency protocols that support them. It provided a trade-off analysis of these models and analyzed the implications of the models on distributed applications. Finally, it analyzed the sources of overhead in distributed transactional applications and proposed HADES which added new hardware for the processor and the NIC to provide low-latency distributed transactions. The thesis also showed how HADES can be combined with the different persistency models and utilize the NIC to offer in-network persistency.

The proposed techniques can improve the performance and programmability of future systems that utilize NVMs, and multi-node systems that use fast network interconnects and SmartNICs.

APPENDIX A: OTHER WORK

In this Section I briefly describe other research projects that I conducted during my Ph.D. in parallel with this thesis.

Virtual Memory Translation. The current implementation of page tables organizes the translations in a multi-level radix tree, namely Radix page tables. However, the increasing memory demands of modern applications is putting high pressure on the translation mechanism that requires serially searching the levels of the radix tree to find a translation. To tackle this problem we proposed *Elastic Cuckoo Page Tables* [297] that introduces a hashed page table design. The main idea of this design is to transform the sequential process of pointer chasing that exists in radix page tables into a fully parallel translation lookup that takes advantage of memory level parallelism.

The problem of Radix page tables is further exacerbated in virtualized environments, where a single translation might require up to twenty four sequential memory accesses to traverse the page tables. To this end, we proposed *Nested Elastic Cuckoo Page Tables* [298] that supports nested parallel address translations.

Wireless Network On-Chip. As the core count is increasing in shared memory multi-cores, the communication between the different cores can become a performance bottleneck. In such environments cache coherence protocols can suffer from great inefficiencies for cache lines that are shared between multiple cores. Wireless Network on Chip is a technology that can alleviate the problem of communicating between the different cores by providing low-latency message transfers and broadcast communication. To enable efficient coherence in multi-cores we propose *WiDir* [299] that uses Wireless Network on Chip and augments an invalidation-based directory cache coherence protocol with wireless transactions for cache lines that are heavily shared between many cores.

REFERENCES

[1] Intel, "3D XPoint: A Breakthrough in Non-Volatile Memory Technology." https://www.intel.com/content/www/us/en/architecture-and-technology/intel-micron-3d-xpoint-webcast.html, 2018.

[2] M. K. Qureshi, S. Gurumurthi, and B. Rajendran, *Phase Change Memory: From Devices to Systems.*, 1st ed. Morgan & Claypool Publishers, 2011.

[3] S. Raoux, G. W. Burr, M. J. Breitwisch, C. T. Rettner, Y. C. Chen, R. M. Shelby, M. Salinga, D. Krebs, S. H. Chen, H. L. Lung, and C. H. Lam, "Phase-change random access memory: A scalable technology," *IBM Journal of Research and Development*, vol. 52, no. 4.5, pp. 465–479, July 2008.

[4] P. Zhou, B. Zhao, J. Yang, and Y. Zhang, "A durable and energy efficient main memory using phase change memory technology," in *36th International Symposium on Computer Architecture (ISCA'09)*, Austin, TX, 2009.

[5] D. Apalkov, A. Khvalkovskiy, S. Watts, V. Nikitin, X. Tang, D. Lottis, K. Moon, X. Luo, E. Chen, A. Ong, A. Driskill-Smith, and M. Krounbi, "Spin-transfer Torque Magnetic Random Access Memory (STT-MRAM)," *J. Emerg. Technol. Comput. Syst.*, May 2013.

[6] H. Akinaga and H. Shima, "Resistive Random Access Memory (ReRAM) Based on Metal Oxides." *Proceedings of the IEEE*, vol. 98, no. 12, pp. 2237–2251, Dec 2010.

[7] J. Izraelevitz, J. Yang, L. Zhang, J. Kim, X. Liu, A. Memaripour, Y. J. Soh, Z. Wang, Y. Xu, S. R. Dulloor, J. Zhao, and S. Swanson, "Basic Performance Measurements of the Intel Optane DC Persistent Memory Module." *CoRR*, vol. abs/1903.05714, 2019. [Online]. Available: http://arxiv.org/abs/1903.05714

[8] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting Phase Change Memory As a Scalable DRAM Alternative," in *the 36th Annual International Symposium on Computer Architecture*, 2009. [Online]. Available: http://doi.acm.org/10.1145/1555754.1555758 pp. 2–13.

[9] M. Chang, P. Rosenfeld, S. Lu, and B. Jacob, "Technology comparison for large last-level caches (L3Cs): Low-leakage SRAM, low write-energy STT-RAM, and refresh-optimized eDRAM," in *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, 2013.

[10] A. V. Khvalkovskiy, D. Apalkov, S. Watts, R. Chepulskii, R. S. Beach, A. Ong, X. Tang, A. Driskill-Smith, W. H. Butler, P. B. Visscher, D. Lottis, E. Chen, V. Nikitin, and M. Krounbi, "Basic principles of STT-MRAM cell operation in memory arrays," *Journal of Physics D: Applied Physics*, feb 2013.

[11] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson, “NV-Heaps: Making Persistent Objects Fast and Safe with Next-generation, Non-volatile Memories,” in *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVI. New York, NY, USA: ACM, 2011. [Online]. Available: http://doi.acm.org/10.1145/1950365.1950380 pp. 105–118.

[12] S. Nalli, S. Haria, M. D. Hill, M. M. Swift, H. Volos, and K. Keeton, “An Analysis of Persistent Memory Use with WHISPER,” in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2017, Xi’an, China, April 8-12, 2017*, 2017, pp. 135–148.

[13] J. E. Denny, S. Lee, and J. S. Vetter, “NVL-C: Static Analysis Techniques for Efficient, Correct Programming of Non-Volatile Main Memory Systems,” in *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC ’16. New York, NY, USA: ACM, 2016. [Online]. Available: http://doi.acm.org/10.1145/2907294.2907303 pp. 125–136.

[14] M. Wu, Z. Zhao, H. Li, H. Li, H. Chen, B. Zang, and H. Guan, “Espresso: Brewing java for more non-volatility with non-volatile memory,” in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS ’18, New York, NY, USA, 2018.

[15] A. Kolli, V. Gogte, A. Saidi, S. Diestelhorst, P. M. Chen, S. Narayanasamy, and T. F. Wenisch, “Language-level Persistency,” in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA ’17. New York, NY, USA: ACM, 2017. [Online]. Available: http://doi.acm.org/10.1145/3079856.3080229 pp. 481–493.

[16] H. Volos, A. J. Tack, and M. M. Swift, “Mnemosyne: Lightweight persistent memory,” in *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVI. New York, NY, USA: ACM, 2011. [Online]. Available: http://doi.acm.org/10.1145/1950365.1950379 pp. 91–104.

[17] D. R. Chakrabarti, H.-J. Boehm, and K. Bhandari, “Atlas: Leveraging Locks for Non-volatile Memory Consistency,” in *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, ser. OOPSLA ’14. New York, NY, USA: ACM, 2014. [Online]. Available: http://doi.acm.org/10.1145/2660193.2660224 pp. 433–452.

[18] T. C.-H. Hsu, H. Brügner, I. Roy, K. Keeton, and P. Eugster, “NVthreads: Practical Persistence for Multi-threaded Applications,” in *Proceedings of the Twelfth European Conference on Computer Systems*, ser. EuroSys ’17. New York, NY, USA: ACM, 2017. [Online]. Available: http://doi.acm.org/10.1145/3064176.3064204 pp. 468–482.

[19] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee, "Better I/O Through Byte-addressable, Persistent Memory," in *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, ser. SOSP '09. New York, NY, USA: ACM, 2009. [Online]. Available: http://doi.acm.org/10.1145/1629575.1629589 pp. 133–146.

[20] N. Cohen, D. T. Aksun, and J. R. Larus, "Object-oriented recovery for non-volatile memory," *PACMPL*, vol. 2, no. OOPSLA, pp. 153:1–153:22, 2018.

[21] V. Gogte, S. Diestelhorst, W. Wang, S. Narayanasamy, P. M. Chen, and T. F. Wenisch, "Persistency for Synchronization-free Regions," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2018. New York, NY, USA: ACM, 2018. [Online]. Available: http://doi.acm.org/10.1145/3192366.3192367 pp. 46–61.

[22] "Intel 64 and IA-32 Architectures Software Develop's Manual," https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf, 2015.

[23] T. Shull, J. Huang, and J. Torrellas, "AutoPersist: An Easy-to-use Java NVM Framework Based on Reachability," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2019, 2019, pp. 316–332.

[24] J. Guerra, L. Mármol, D. Campello, C. Crespo, R. Rangaswami, and J. Wei, "Software Persistent Memory," in *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, ser. USENIX ATC'12. Berkeley, CA, USA: USENIX Association, 2012. [Online]. Available: http://dl.acm.org/citation.cfm?id=2342821.2342850 pp. 29–29.

[25] L. Marmol, M. Chowdhury, and R. Rangaswami, "LibPM: Simplifying Application Usage of Persistent Memory," *ACM Trans. Storage*, vol. 14, no. 4, pp. 34:1–34:18, Dec. 2018. [Online]. Available: http://doi.acm.org/10.1145/3278141

[26] A. Ganesan, R. Alagappan, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, "Strong and Efficient Consistency with Consistency-Aware Durability." in *18th USENIX Conference on File and Storage Technologies (FAST 20)*. Santa Clara, CA: USENIX Association, Feb. 2020. [Online]. Available: https://www.usenix.org/conference/fast20/presentation/ganesan pp. 323–337.

[27] E. B. Nightingale, K. Veeraraghavan, P. M. Chen, and J. Flinn, "Rethink the Sync." *ACM Trans. Comput. Syst.*, vol. 26, no. 3, Sep. 2008. [Online]. Available: https://doi.org/10.1145/1394441.1394442

[28] Y. Won, J. Jung, G. Choi, J. Oh, S. Son, J. Hwang, and S. Cho, "Barrier-Enabled IO Stack for Flash Storage." in *16th USENIX Conference on File and Storage Technologies (FAST 18).* Oakland, CA: USENIX Association, Feb. 2018. [Online]. Available: https://www.usenix.org/conference/fast18/presentation/won pp. 211–226.

[29] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. M. Caulfield, E. S. Chung, H. K. Chandrappa, S. Chaturmohta, M. Humphrey, J. Lavier, N. Lam, F. Liu, K. Ovtcharov, J. Padhye, G. Popuri, S. Raindel, T. Sapre, M. Shaw, G. Silva, M. Sivakumar, N. Srivastava, A. Verma, Q. Zuhair, D. Bansal, D. Burger, K. Vaid, D. A. Maltz, and A. G. Greenberg, "Azure Accelerated Networking: SmartNICs in the Public Cloud," in *15th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2018, Renton, WA, USA, April 9-11, 2018*, S. Banerjee and S. Seshan, Eds. USENIX Association, 2018. [Online]. Available: https://www.usenix.org/conference/nsdi18/presentation/firestone pp. 51–66.

[30] J. Lin, K. Patel, B. E. Stephens, A. Sivaraman, and A. Akella, "PANIC: A High-Performance Programmable NIC for Multi-tenant Networks," in *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020.* USENIX Association, 2020. [Online]. Available: https://www.usenix.org/conference/osdi20/presentation/lin pp. 243–259.

[31] Mellanox Technologies, "NVIDIA Mellanox BlueField SmartNIC for Ethernet High Performance Ethernet Network Adapter Cards," 2020. [Online]. Available: https://www.mellanox.com/files/doc-2020/pb-bluefield-smart-nic.pdf

[32] A. Kokolis, D. Skarlatos, and J. Torrellas, "PageSeer: Using Page Walks to Trigger Page Swaps in Hybrid Memory Systems," in *2019 IEEE 25th International Symposium on High Performance Computer Architecture*, Feb 2019.

[33] A. Kokolis, N. Mantri, S. Ganapathy, J. Torrellas, and J. Kalamatianos, "A Method for Hiding the Increased Non-Volatile Cache Read Latency," 2021.

[34] J. Kalamatianos, A. Kokolis, and S. Ganapathy, "Latency hiding for caches ," U.S. Patent Application No. US16/683,142, 2019.

[35] A. Kokolis, T. Shull, J. Huang, and J. Torrellas, "P-INSPECT: Architectural Support for Programmable Non-Volatile Memory Frameworks," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020, pp. 509–524.

[36] A. Kokolis, A. Psistakis, B. Reidys, J. Huang, and J. Torrellas, "Distributed Data Persistency," in *2021 54rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2021.

[37] J. A. Mandelman, R. H. Dennard, G. B. Bronner, J. K. DeBrosse, R. Divakaruni, Y. Li, and C. J. Radens, “Challenges and future directions for the scaling of dynamic random-access memory (DRAM),” *IBM Journal of Research and Development*, March 2002.

[38] E. Kültürsay, M. Kandemir, A. Sivasubramaniam, and O. Mutlu, “Evaluating STT-RAM as an energy-efficient main memory alternative,” in *2013 IEEE International Symposium on Performance Analysis of Systems and Software*, April 2013.

[39] F. T. Hady, A. Foong, B. Veal, and D. Williams, “Platform Storage Performance With 3D XPoint Technology,” *Proceedings of the IEEE*, Sept 2017.

[40] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, “Scalable High Performance Main Memory System Using Phase-change Memory Technology.” in *the 36th Annual International Symposium on Computer Architecture*, 2009. [Online]. Available: http://doi.acm.org/10.1145/1555754.1555760

[41] J. Sim, A. R. Alameldeen, Z. Chishti, C. Wilkerson, and H. Kim, “Transparent Hardware Management of Stacked DRAM As Part of Memory,” in *the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, 2014. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2014.56

[42] A. Prodromou, M. Meswani, N. Jayasena, G. Loh, and D. M. Tullsen, “MemPod: A Clustered Architecture for Efficient and Scalable Migration in Flat Address Space Multi-level Memories,” in *2017 IEEE International Symposium on High Performance Computer Architecture*, Feb 2017.

[43] J. H. Ryoo, M. R. Meswani, A. Prodromou, and L. K. John, “SILC-FM: Subblocked InterLeaved Cache-Like Flat Memory Organization,” in *2017 IEEE International Symposium on High Performance Computer Architecture*, Feb 2017.

[44] D. Jevdjic, G. H. Loh, C. Kaynak, and B. Falsafi, “Unison Cache: A Scalable and Effective Die-Stacked DRAM Cache,” in *the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-47, 2014. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2014.51

[45] G. H. Loh and M. D. Hill, “Efficiently Enabling Conventional Block Sizes for Very Large Die-stacked DRAM Caches,” in *the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, 2011. [Online]. Available: http://doi.acm.org/10.1145/2155620.2155673

[46] J. Meza, J. Chang, H. Yoon, O. Mutlu, and P. Ranganathan, “Enabling Efficient and Scalable Hybrid Memories Using Fine-Granularity DRAM Cache Management,” *IEEE Computer Architecture Letters*, July 2012.

[47] M. K. Qureshi and G. H. Loh, “Fundamental Latency Trade-off in Architecting DRAM Caches: Outperforming Impractical SRAM-Tags with a Simple and Practical Design,” in *the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, 2012. [Online]. Available: https://doi.org/10.1109/MICRO.2012.30

[48] J. Sim, G. H. Loh, H. Kim, M. O'Connor, and M. Thottethodi, "A Mostly-Clean DRAM Cache for Effective Hit Speculation and Self-Balancing Dispatch," in *the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, 2012. [Online]. Available: https://doi.org/10.1109/MICRO.2012.31

[49] L. Zhao, R. Iyer, R. Illikkal, and D. Newell, "Exploring DRAM cache architectures for CMP server platforms," in *2007 25th International Conference on Computer Design*, Oct 2007.

[50] C. Chou, A. Jaleel, and M. K. Qureshi, "CAMEO: A Two-Level Memory Organization with Capacity of Main Memory and Flexibility of Hardware-Managed Cache," in *47th Annual IEEE/ACM International Symposium on Microarchitecture*, 2014. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2014.63

[51] D. Knyaginin, V. Papaefstathiou, and P. Stenstrom, "ProFess: A Probabilistic Hybrid Main Memory Management Framework for High Performance and Fairness," in *2018 IEEE International Symposium on High Performance Computer Architecture*, Feb 2018.

[52] M. R. Meswani, S. Blagodurov, D. Roberts, J. Slice, M. Ignatowski, and G. H. Loh, "Heterogeneous memory architectures: A HW/SW approach for mixing die-stacked and off-package memories," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture*, Feb. 2015. [Online]. Available: doi.ieeecomputersociety.org/10.1109/HPCA.2015.7056027

[53] Y. Lee, J. Kim, H. Jang, H. Yang, J. Kim, J. Jeong, and J. W. Lee, "A Fully Associative, Tagless DRAM Cache," in *the 42nd Annual International Symposium on Computer Architecture*, 2015. [Online]. Available: http://doi.acm.org/10.1145/2749469.2750383

[54] M. Oskin and G. H. Loh, "A Software-Managed Approach to Die-Stacked DRAM," in *2015 International Conference on Parallel Architecture and Compilation*, Oct 2015.

[55] N. Agarwal and T. F. Wenisch, "Thermostat: Application-transparent Page Management for Two-tiered Main Memory," in *the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, 2017. [Online]. Available: http://doi.acm.org/10.1145/3037697.3037706

[56] T. Straumann, "Open Source Real-Time Operating System Overview (Invited)," in *Accelerator and Large Experimental Physics Control Systems*, H. Shoaee, Ed., 2001, p. 235.

[57] J. B. Kotra, H. Zhang, A. R. Alameldeen, C. Wilkerson, and M. Kandemir, "CHAMELEON: A Dynamically Reconfigurable Heterogeneous Memory System," in *the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, 2018.

[58] R. M. Karp, C. H. Papadimitriou, and S. Shenker, "A Simple Algorithm For Finding Frequent Elements In Streams and Bags," *ACM Transactions on Database Systems*, vol. 28, p. 2003, 2003.

[59] C. Chou, A. Jaleel, and M. Qureshi, "BATMAN: Techniques for Maximizing System Bandwidth of Memory Systems with Stacked-DRAM," in *the International Symposium on Memory Systems*, 2017. [Online]. Available: http://doi.acm.org/10.1145/3132402.3132404

[60] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner, "Simics: A full system simulation platform," *Computer*, vol. 35, no. 2, pp. 50–58, Feb 2002.

[61] A. F. Rodrigues, K. S. Hemmert, B. W. Barrett, C. Kersey, R. Oldfield, M. Weston, R. Risen, J. Cook, P. Rosenfeld, E. CooperBalls, and B. Jacob, "The Structural Simulation Toolkit," *SIGMETRICS Perform. Eval. Rev.*, vol. 38, no. 4, Mar. 2011.

[62] A. Awad, S. D. Hammond, G. R. Voskuilen, and R. J. Hoekstra, "Samba: A Detailed Memory Management Unit (MMU) for the SST Simulation Framework," Sandia National Laboratories, Tech. Rep. SAND2017-0002, January 2017.

[63] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "DRAMSim2: A Cycle Accurate Memory System Simulator," *IEEE Computer Architecture Letters*, Jan 2011.

[64] D. Skarlatos, N. S. Kim, and J. Torrellas, "PageForge: A Near-memory Content-aware Page-merging Architecture," in *the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, 2017. [Online]. Available: http://doi.acm.org/10.1145/3123939.3124540

[65] R. Balasubramonian, A. B. Kahng, N. Muralimanohar, A. Shafiee, and V. Srinivas, "CACTI 7: New Tools for Interconnect Exploration in Innovative Off-Chip Memories," *ACM Trans. Archit. Code Optim.*, vol. 14, no. 2, June 2017. [Online]. Available: http://doi.acm.org/10.1145/3085572

[66] N. Chachmon, D. Richins, R. Cohn, M. Christensson, W. Cui, and V. J. Reddi, "Simulation and Analysis Engine for Scale-Out Workloads," in *2016 International Conference on Supercomputing*, 2016.

[67] J. L. Henning, "SPEC CPU2006 Benchmark Descriptions," *SIGARCH Comput. Archit. News*, vol. 34, no. 4, Sep. 2006. [Online]. Available: http://doi.acm.org/10.1145/1186736.1186737

[68] C. Sakalis, C. Leonardsson, S. Kaxiras, and A. Ros, "Splash-3: A properly synchronized benchmark suite for contemporary research," in *2016 IEEE International Symposium on Performance Analysis of Systems and Software*, April 2016.

[69] "CORAL Benchmark Codes," https://asc.llnl.gov/CORAL-benchmarks/.

[70] A. Bhattacharjee, "Translation-Triggered Prefetching," in *the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, 2017.

[71] S. Kannan, A. Gavrilovska, V. Gupta, and K. Schwan, "HeteroOS: OS Design for Heterogeneous Memory Management in Datacenter," in *the 44th Annual International Symposium on Computer Architecture*, 2017. [Online]. Available: http://doi.acm.org/10.1145/3079856.3080245

[72] L. E. Ramos, E. Gorbatov, and R. Bianchini, "Page Placement in Hybrid Memory Systems," in *the International Conference on Supercomputing*, 2011. [Online]. Available: http://doi.acm.org/10.1145/1995896.1995911

[73] F. X. Lin and X. Liu, "Memif: Towards Programming Heterogeneous Memory Asynchronously," in *the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, 2016. [Online]. Available: http://doi.acm.org/10.1145/2872362.2872401

[74] X. Yu, C. J. Hughes, N. Satish, O. Mutlu, and S. Devadas, "Banshee: Bandwidth-efficient DRAM Caching via Software/Hardware Cooperation," in *the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, 2017. [Online]. Available: http://doi.acm.org/10.1145/3123939.3124555

[75] M. Islam, S. Banerjee, M. Meswani, and K. Kavi, "Prefetching as a Potentially Effective Technique for Hybrid Memory Optimization," in *the Second International Symposium on Memory Systems*, 2016. [Online]. Available: http://doi.acm.org/10.1145/2989081.2989129

[76] S. Volos, J. Picorel, B. Falsafi, and B. Grot, "BuMP: Bulk Memory Access Prediction and Streaming," in *the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, 2014. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2014.44

[77] Y. Solihin, J. Lee, and J. Torrellas, "Using a user-level memory thread for correlation prefetching," in *29th Annual International Symposium on Computer Architecture*, 2002.

[78] A.-C. Lai, C. Fide, and B. Falsafi, "Dead-block prediction and dead-block correlating prefetchers," in *28th Annual International Symposium on Computer Architecture*, 2001.

[79] "Big trouble at 3nm," https://semiengineering.com/big-trouble-at-3nm/.

[80] "Apple A13 & Beyond: How Transistor Count And Costs Will Go Up," https://wccftech.com/apple-5nm-3nm-cost-transistors/.

[81] Y. Zhang, Y. Li, Z. Sun, H. Li, Y. Chen, and A. K. Jones, "Read Performance: The Newest Barrier in Scaled STT-RAM," *IEEE Transactions on Very Large Scale Integration Systems*, June 2015.

[82] X. Wu, J. Li, L. Zhang, E. Speight, R. Rajamony, and Y. Xie, “Hybrid Cache Architecture with Disparate Memory Technologies,” in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, 2009.

[83] J. Wang, X. Dong, and Y. Xie, “OAP: An obstruction-aware cache management policy for STT-RAM last-level caches,” in *2013 Design, Automation Test in Europe Conference Exhibition*, March 2013.

[84] Z. Wang, D. A. Jimenez, C. Xu, G. Sun, and Y. Xie, “Adaptive placement and migration policy for an STT-RAM-based hybrid cache,” in *2014 IEEE 20th International Symposium on High Performance Computer Architecture*, Feb 2014.

[85] J. Ahn, S. Yoo, and K. Choi, “DASCA: Dead Write Prediction Assisted STT-RAM Cache Architecture,” in *2014 IEEE 20th International Symposium on High Performance Computer Architecture*, Feb 2014.

[86] H. M. Ahmed T. El-Thakeb, Hamdy A. Elhamid and Y. Ismail, “Performance evaluation of FINFET based SRAM under statistical VT variability,” in *Proceedings of the 2014 International Conference on Microelectronics*, ser. ICM’14. IEEE, 2014.

[87] S. S.R., B. Ramakrishna, Samiksha, R. Banu, and P. Shubham, “Design and Performance Analysis of 6T SRAM Cell in 22nm CMOS and FINFET Technology Nodes,” in *Proceedings of the 2017 International Conference on Recent Advances in Electronics and Communication Technology*, ser. ICRAECT’17. IEEE, 2017.

[88] E. Karl, Z. Guo, J. Conary, J. Miller, Y. Ng, S. Nalam, D. Kim, J. Keane, X. Wang, U. Bhattacharya, and K. Zhang, “A 0.6 V, 1.5 GHz 84 Mb SRAM in 14 nm FinFET CMOS Technology With Capacitive Charge-Sharing Write Assist Circuitry,” *IEEE Journal of Solid-State Circuits*, vol. 51, no. 1, pp. 222–229, 2016.

[89] A. Shafaei, Y. Wang, X. Lin, and M. Pedram, “Fincacti: Architectural analysis and modeling of caches with deeply-scaled finfet devices,” in *2014 IEEE Computer Society Annual Symposium on VLSI*, 2014, pp. 290–295.

[90] T. Ohsawa, H. Koike, S. Miura, H. Honjo, K. Kinoshita, S. Ikeda, T. Hanyu, H. Ohno, and T. Endoh, “A 1 Mb Nonvolatile Embedded Memory Using 4T2MTJ Cell With 32 b Fine-Grained Power Gating Scheme,” *IEEE Journal of Solid-State Circuits*, June 2013.

[91] T. Kawahara, R. Takemura, K. Miura, J. Hayakawa, S. Ikeda, Y. Lee, R. Sasaki, Y. Goto, K. Ito, T. Meguro, F. Matsukura, H. Takahashi, H. Matsuoka, and H. Ohno, “2Mb Spin-Transfer Torque RAM (SPRAM) with Bit-by-Bit Bidirectional Current Write and Parallelizing-Direction Current Read,” in *2007 IEEE International Solid-State Circuits Conference*, Feb 2007.

[92] Y. Chen, H. Li, X. Wang, W. Zhu, W. Xu, and T. Zhang, “A 130 nm 1.2 V/3.3 V 16 Kb Spin-Transfer Torque Random Access Memory With Nondestructive Self-Reference Sensing Scheme,” *IEEE Journal of Solid-State Circuits*, Feb 2012.

[93] H. Noguchi, K. Ikegami, K. Kushida, K. Abe, S. Itai, S. Takaya, N. Shimomura, J. Ito, A. Kawasumi, H. Hara, and S. Fujita, "A 3.3ns-access-time 71.2μW/MHz 1Mb embedded STT-MRAM using physically eliminated read-disturb scheme and normally-off memory architecture," in *2015 IEEE International Solid-State Circuits Conference*, Feb 2015.

[94] S. Sakhare, M. Perumkunnil, T. H. Bao, S. Rao, W. Kim, D. Crotti, F. Yasin, S. Couet, J. Swerts, S. Kundu, D. Yakimets, R. Baert, H. Oh, A. Spessot, A. Mocuta, G. S. Kar, and A. Furnemont, "Enablement of STT-MRAM as last level cache for the high performance computing domain at the 5nm node," in *2018 IEEE International Electron Devices Meeting (IEDM)*, 2018, pp. 18.3.1–18.3.4.

[95] Y. Lu, T. Zhong, W. Hsu, S. Kim, X. Lu, J. J. Kan, C. Park, W. C. Chen, X. Li, X. Zhu, P. Wang, M. Gottwald, J. Fatehi, L. Seward, J. P. Kim, N. Yu, G. Jan, J. Haq, S. Le, Y. J. Wang, L. Thomas, J. Zhu, H. Liu, Y. J. Lee, R. Y. Tong, K. Pi, D. Shen, R. He, Z. Teng, V. Lam, R. Annapragada, T. Torng, P. Wang, and S. H. Kang, "Fully functional perpendicular STT-MRAM macro embedded in 40 nm logic for energy-efficient IOT applications," in *2015 IEEE International Electron Devices Meeting (IEDM)*, Dec 2015.

[96] K. Ikegami, H. Noguchi, S. Takaya, C. Kamata, M. Amano, K. Abe, K. Kushida, E. Kitagawa, T. Ochiai, N. Shimomura, D. Saida, A. Kawasumi, H. Hara, J. Ito, and S. Fujita, "MTJ-based "normally-off processors" with thermal stability factor engineered perpendicular MTJ, L2 cache based on 2T-2MTJ cell, L3 and last level cache based on 1T-1MTJ cell and novel error handling scheme," in *2015 IEEE International Electron Devices Meeting (IEDM)*, 2015, pp. 25.1.1–25.1.4.

[97] G. Jan, L. Thomas, S. Le, Y. Lee, H. Liu, J. Zhu, R. Tong, K. Pi, Y. Wang, D. Shen, R. He, J. Haq, J. Teng, V. Lam, K. Huang, T. Zhong, T. Torng, and P. Wang, "Demonstration of fully functional 8Mb perpendicular STT-MRAM chips with sub-5ns writing for non-volatile embedded memories," in *2014 Symposium on VLSI Technology (VLSI-Technology)*, June 2014.

[98] H. Noguchi, K. Ikegami, S. Takaya, E. Arima, K. Kushida, A. Kawasumi, H. Hara, K. Abe, N. Shimomura, J. Ito, S. Fujita, T. Nakada, and H. Nakamura, "4Mb STT-MRAM-based cache with memory-access-aware power optimization and write-verify-write / read-modify-write scheme," in *2016 IEEE International Solid-State Circuits Conference*, Jan 2016.

[99] T. Ohsawa, S. Miura, K. Kinoshita, H. Honjo, S. Ikeda, T. Hanyu, H. Ohno, and T. Endoh, "A 1.5nsec/2.1nsec random read/write cycle 1Mb STT-RAM using 6T2MTJ cell with background write for nonvolatile e-memories," in *2013 Symposium on VLSI Circuits*, June 2013.

[100] C. W. Smullen, V. Mohan, A. Nigam, S. Gurumurthi, and M. R. Stan, "Relaxing non-volatility for fast and energy-efficient STT-RAM caches," in *2011 IEEE 17th International Symposium on High Performance Computer Architecture*, Feb 2011.

[101] A. Jog, A. K. Mishra, C. Xu, Y. Xie, V. Narayanan, R. Iyer, and C. R. Das, "Cache revive: Architecting volatile STT-RAM caches for enhanced performance in CMPs," in *DAC Design Automation Conference*, 2012.

[102] Z. Sun, X. Bi, H. Li, W. Wong, Z. Ong, X. Zhu, and W. Wu, "Multi retention level STT-RAM cache designs with a dynamic refresh scheme," in *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec 2011.

[103] D. Kang, S. Baek, J. Choi, D. Lee, S. H. Noh, and O. Mutlu, "Amnesic cache management for non-volatile memory," in *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*, 2015, pp. 1–13.

[104] S. Pelley, P. M. Chen, and T. F. Wenisch, "Memory Persistency," in *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, 2014.

[105] A. Kolli, J. Rosen, S. Diestelhorst, A. Saidi, S. Pelley, S. Liu, P. M. Chen, and T. F. Wenisch, "Delegated Persist Ordering," in *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, 2016.

[106] A. Chintaluri, H. Naeimi, S. Natarajan, and A. Raychowdhury, "Analysis of Defects and Variations in Embedded Spin Transfer Torque (STT) MRAM Arrays," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, Sep. 2016.

[107] Y. Chen, J. Cong, H. Huang, B. Liu, C. Liu, M. Potkonjak, and G. Reinman, "Dynamically reconfigurable hybrid cache: An energy-efficient last-level cache design," in *2012 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2012.

[108] G. Sun, X. Dong, Y. Xie, J. Li, and Y. Chen, "A novel architecture of the 3D stacked MRAM L2 cache for CMPs," in *2009 IEEE 15th International Symposium on High Performance Computer Architecture*, 2009.

[109] K. Korgaonkar, I. Bhati, H. Liu, J. Gaur, S. Manipatruni, S. Subramoney, T. Karnik, S. Swanson, I. Young, and H. Wang, "Density Tradeoffs of Non-Volatile Memory as a Replacement for SRAM Based Last Level Cache," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture*, June 2018.

[110] L. Zhang, B. Neely, D. Franklin, D. Strukov, Y. Xie, and F. T. Chong, "Mellow Writes: Extending Lifetime in Resistive Memories through Selective Slow Write Backs," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture*, 2016.

[111] H. Cheng, J. Zhao, J. Sampson, M. J. Irwin, A. Jaleel, Y. Lu, and Y. Xie, "LAP: Loop-Block Aware Inclusion Properties for Energy-Efficient Asymmetric Last Level Caches," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, 2016.

[112] J. Zhan, O. Kayiran, G. H. Loh, C. R. Das, and Y. Xie, "OSCAR: Orchestrating STT-RAM cache traffic for heterogeneous CPU-GPU architectures," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture*, Oct 2016.

[113] W. Xu, H. Sun, X. Wang, Y. Chen, and T. Zhang, "Design of Last-Level On-Chip Cache Using Spin-Torque Transfer RAM (STT RAM)," *IEEE Transactions on Very Large Scale Integration Systems*, March 2011.

[114] H. Noguchi, K. Ikegami, N. Shimomura, T. Tetsufumi, J. Ito, and S. Fujita, "Highly reliable and low-power nonvolatile cache memory with advanced perpendicular STT-MRAM for high-performance CPU," in *2014 Symposium on VLSI Circuits*, June 2014.

[115] J. G. Alzate, U. Arslan, P. Bai, J. Brockman, Y. J. Chen, N. Das, K. Fischer, T. Ghani, P. Heil, P. Hentges, R. Jahan, A. Littlejohn, M. Mainuddin, D. Ouellette, J. Pellegren, T. Pramanik, C. Puls, P. Quintero, T. Rahman, M. Sekhar, B. Sell, M. Seth, A. J. Smith, A. K. Smith, L. Wei, C. Wiegand, O. Golonzka, and F. Hamzaoglu, "2 MB Array-Level Demonstration of STT-MRAM Process and Performance Towards L4 Cache Applications," in *2019 IEEE International Electron Devices Meeting (IEDM)*, 2019, pp. 2.4.1–2.4.4.

[116] Z. Wang, X. Hao, P. Xu, L. Hu, D. Jung, W. Kim, K. Satoh, B. Yen, Z. Wei, L. Wang, J. Zhang, and Y. Huai, "Stt-mram for embedded memory applications," in *2020 IEEE International Memory Workshop (IMW)*, 2020, pp. 1–3.

[117] Y. Chih, Y. Shih, C. Lee, Y. Chang, P. Lee, H. Lin, Y. Chen, C. Lo, M. Shih, K. Shen, H. Chuang, and T. J. Chang, "13.3 A 22nm 32Mb Embedded STT-MRAM with 10ns Read Speed, 1M Cycle Write Endurance, 10 Years Retention at 150°C and High Immunity to Magnetic Field Interference," in *2020 IEEE International Solid- State Circuits Conference - (ISSCC)*, 2020, pp. 222–224.

[118] S. Gupta and H. Zhou, "Spatial Locality-Aware Cache Partitioning for Effective Cache Sharing," in *2015 44th International Conference on Parallel Processing*, Sep. 2015.

[119] "AMD64 Architecture Programmer's Manual Volume 2: System Programming," https://www.amd.com/system/files/TechDocs/24593.pdf.

[120] R. Balasubramonian, A. B. Kahng, N. Muralimanohar, A. Shafiee, and V. Srinivas, "CACTI 7: New Tools for Interconnect Exploration in Innovative Off-Chip Memories," *ACM Trans. Archit. Code Optim.*, vol. 14, no. 2, June 2017. [Online]. Available: http://doi.acm.org/10.1145/3085572

[121] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2009, pp. 469–480.

[122] "SPEC CPU2017," https://www.spec.org/cpu2017.

[123] "CORAL2 Benchmarks," https://asc.llnl.gov/coral-2-benchmarks/.

[124] Greg Hamerly and E. Perelman and Jeremy Lau and B. Calder, "SimPoint 3.0: Faster and More Flexible Program Phase Analysis," *J. Instr. Level Parallelism*, vol. 7, 2005.

[125] M. K. Qureshi and G. H. Loh, “Fundamental latency trade-off in architecting dram caches: Outperforming impractical sram-tags with a simple and practical design,” in *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, 2012, pp. 235–246.

[126] D. Jevdjic, G. H. Loh, C. Kaynak, and B. Falsafi, “Unison Cache: A Scalable and Effective Die-Stacked DRAM Cache,” in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, 2014, pp. 25–37.

[127] D. Jevdjic, S. Volos, and B. Falsafi, “Die-Stacked DRAM Caches for Servers: Hit Ratio, Latency, or Bandwidth? Have It All with Footprint Cache,” in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ser. ISCA ’13. New York, NY, USA: Association for Computing Machinery, 2013. [Online]. Available: https://doi.org/10.1145/2485922.2485957 p. 404–415.

[128] X. Jiang, N. Madan, L. Zhao, M. Upton, R. Iyer, S. Makineni, D. Newell, Y. Solihin, and R. Balasubramonian, “CHOP: Adaptive filter-based DRAM caching for CMP server platforms,” in *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, 2010, pp. 1–12.

[129] G. H. Loh and M. D. Hill, “Efficiently enabling conventional block sizes for very large die-stacked dram caches,” in *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture*, 2011, pp. 454–464.

[130] Y. Lee, J. Kim, H. Jang, H. Yang, J. Kim, J. Jeong, and J. W. Lee, “A fully associative, tagless DRAM cache,” in *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture*, 2015, pp. 211–222.

[131] H. Jang, Y. Lee, J. Kim, Y. Kim, J. Kim, J. Jeong, and J. W. Lee, “Efficient footprint caching for Tagless DRAM Caches,” in *2016 IEEE International Symposium on High Performance Computer Architecture*, 2016, pp. 237–248.

[132] D. H. Woo, N. H. Seong, D. L. Lewis, and H.-H. S. Lee, “An optimized 3d-stacked memory architecture by exploiting excessive, high-density tsv bandwidth,” in *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, 2010, pp. 1–12.

[133] M. Mao, G. Sun, Y. Li, A. K. Jones, and Y. Chen, “Prefetching techniques for STT-RAM based last-level cache in CMP systems,” in *2014 19th Asia and South Pacific Design Automation Conference*, Jan 2014.

[134] Intel, “Introduction to Programming with Intel Optane DC Persistent Memory,” 2018. [Online]. Available: https://software.intel.com/en-us/articles/introduction-to-programming-with-persistent-memory-from-intel

[135] T. Shull, J. Huang, and J. Torrellas, “Defining a High-level Programming Model for Emerging NVRAM Technologies,” in *Proceedings of the 15th International Conference on Managed Languages & Runtimes*, ser. ManLang ’18. New York, NY, USA: ACM, 2018. [Online]. Available: http://doi.acm.org/10.1145/3237009.3237027 pp. 11:1–11:7.

[136] H.-J. Boehm and D. R. Chakrabarti, "Persistence Programming Models for Non-volatile Memory," in *Proceedings of the 2016 ACM SIGPLAN International Symposium on Memory Management*, ser. ISMM 2016. New York, NY, USA: ACM, 2016. [Online]. Available: http://doi.acm.org/10.1145/2926697.2926704 pp. 55–67.

[137] "NVM Programming Model v1.2," https://www.snia.org/sites/default/files/technical_work/final/NVMProgrammingModel_v1.2.pdf.

[138] "Persistent Memory Development Kit," http://pmem.io/pmdk/.

[139] T. Shull, J. Huang, and J. Torrellas, "QuickCheck: Using Speculation to Reduce the Overhead of Checks in NVM Frameworks," in *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE 2019. New York, NY, USA: ACM, 2019. [Online]. Available: http://doi.acm.org/10.1145/3313808.3313822 pp. 137–151.

[140] O. Oleksenko, D. Kuvaiskii, P. Bhatotia, P. Felber, and C. Fetzer, "Intel MPX Explained: A Cross-Layer Analysis of the Intel MPX System Stack," *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 2, no. 2, June 2018. [Online]. Available: https://doi.org/10.1145/3224423

[141] ARM, "Arm Architecture Reference Manual," https://developer.arm.com/docs/ddi0487/latest.

[142] J. Woodruff, R. N. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe, "The CHERI Capability Model: Revisiting RISC in an Age of Risk," *SIGARCH Comput. Archit. News*, vol. 42, no. 3, pp. 457–468, June 2014. [Online]. Available: https://doi.org/10.1145/2678373.2665740

[143] R. N. M. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. H. Dave, B. Davis, K. Gudka, B. Laurie, S. J. Murdoch, R. M. Norton, M. Roe, S. D. Son, and M. Vadera, "CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization," *2015 IEEE Symposium on Security and Privacy*, pp. 20–37, 2015.

[144] J. Ren, Q. Hu, S. Khan, and T. Moscibroda, "Programming for Non-Volatile Main Memory Is Hard," in *Proceedings of the 8th Asia-Pacific Workshop on Systems (APSys '17)*, September 2017, pp. 13:01–13:08.

[145] S. Liu, Y. Wei, J. Zhao, A. Kolli, and S. Khan, "PMTest: A Fast and Flexible Testing Framework for Persistent Memory Programs," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19. New York, NY, USA: ACM, 2019. [Online]. Available: http://doi.acm.org/10.1145/3297858.3304015 pp. 411–425.

[146] "PMDK. An introduction to pmemcheck." https://pmem.io/2015/07/17/pmemcheck-basic.html, 2015.

[147] Zhou, Ping and Zhao, Bo and Yang, Jun and Zhang, Youtao, "A Durable and Energy Efficient Main Memory Using Phase Change Memory Technology," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ser. ISCA '09. New York, NY, USA: Association for Computing Machinery, 2009. [Online]. Available: https://doi.org/10.1145/1555754.1555759 p. 14–23.

[148] S. Cho and H. Lee, "Flip-N-Write: A simple deterministic technique to improve PRAM write performance, energy and endurance," in *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2009, pp. 347–357.

[149] J. Xu, D. Feng, Y. Hua, W. Tong, J. Liu, C. Li, G. Xu, and Y. Chen, "Adaptive Granularity Encoding For Energy-Efficient Non-Volatile Main Memory," in *2019 56th ACM/IEEE Design Automation Conference*, 2019, pp. 1–6.

[150] D. H. Yoon, M. K. Jeong, M. Sullivan, and M. Erez, "The dynamic granularity memory system," in *2012 39th Annual International Symposium on Computer Architecture*, 2012, pp. 548–560.

[151] Y. Joo, D. Niu, X. Dong, G. Sun, N. Chang, and Y. Xie, "Energy- and endurance-aware design of phase change memory caches," in *2010 Design, Automation Test in Europe Conference Exhibition (DATE 2010)*, 2010, pp. 136–141.

[152] Y. Xie, "Modeling, Architecture, and Applications for Emerging Memory Technologies," *IEEE Design Test of Computers*, vol. 28, no. 1, pp. 44–51, 2011.

[153] "Synopsys design compiler," https://www.synopsys.com/.

[154] C. Wimmer, M. Haupt, M. L. Van De Vanter, M. Jordan, L. Daynès, and D. Simon, "Maxine: An Approachable Virtual Machine for, and in, Java," *ACM Trans. Archit. Code Optim.*, vol. 9, no. 4, pp. 30:1–30:24, Jan. 2013.

[155] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '05. New York, NY, USA: ACM, 2005, pp. 190–200.

[156] "QuickCached." [Online]. Available: https://github.com/QuickServerLab/QuickCached

[157] "Pmemkv: Key/Value Datastore for Persistent Memory." [Online]. Available: https://github.com/pmem/pmemkv

[158] "PCollections." [Online]. Available: https://pcollections.org/

[159] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, “Benchmarking Cloud Serving Systems with YCSB.” in *Proceedings of the 1st ACM Symposium on Cloud Computing*, ser. SoCC ’10. New York, NY, USA: ACM, 2010. [Online]. Available: http://doi.acm.org/10.1145/1807128.1807152 pp. 143–154.

[160] J. Xu and S. Swanson, “NOVA: A log-structured file system for hybrid volatile/non-volatile main memories,” in *14th USENIX Conference on File and Storage Technologies (FAST 16)*. Santa Clara, CA: USENIX Association, Feb. 2016. [Online]. Available: https://www.usenix.org/conference/fast16/technical-sessions/presentation/xu pp. 323–338.

[161] J. Xu, L. Zhang, A. Memaripour, A. Gangadharaiah, A. Borase, T. B. Da Silva, S. Swanson, and A. Rudoff, “NOVA-Fortis: A Fault-Tolerant Non-Volatile Main Memory File System,” in *Proceedings of the 26th Symposium on Operating Systems Principles*, ser. SOSP ’17. New York, NY, USA: ACM, 2017. [Online]. Available: http://doi.acm.org/10.1145/3132747.3132761 pp. 478–496.

[162] J. Yang, J. Izraelevitz, and S. Swanson, “Orion: A Distributed File System for Non-Volatile Main Memories and RDMA-Capable Networks.” in *Proceedings of the 17th USENIX Conference on File and Storage Technologies*, ser. FAST’19. USA: USENIX Association, 2019, p. 221–234.

[163] S. Zheng, M. Hoseinzadeh, and S. Swanson, “Ziggurat: A Tiered File System for Non-Volatile Main Memories and Disks,” in *17th USENIX Conference on File and Storage Technologies (FAST 19)*. Boston, MA: USENIX Association, Feb. 2019. [Online]. Available: https://www.usenix.org/conference/fast19/presentation/zheng pp. 207–219.

[164] H. Volos, S. Nalli, S. Panneerselvam, V. Varadarajan, P. Saxena, and M. M. Swift, “Aerie: Flexible File-system Interfaces to Storage-class Memory,” in *Proceedings of the Ninth European Conference on Computer Systems*, ser. EuroSys ’14. New York, NY, USA: ACM, 2014. [Online]. Available: http://doi.acm.org/10.1145/2592798.2592810 pp. 14:1–14:14.

[165] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson, “System Software for Persistent Memory,” in *Proceedings of the Ninth European Conference on Computer Systems*, ser. EuroSys ’14. New York, NY, USA: Association for Computing Machinery, 2014. [Online]. Available: https://doi.org/10.1145/2592798.2592814

[166] J. Zhao, S. Li, D. H. Yoon, Y. Xie, and N. P. Jouppi, “Kiln: Closing the Performance Gap between Systems with and without Persistence Support,” in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-46. New York, NY, USA: Association for Computing Machinery, 2013. [Online]. Available: https://doi.org/10.1145/2540708.2540744 p. 421–432.

[167] S. Shin, S. K. Tirukkovalluri, J. Tuck, and Y. Solihin, "Proteus: a flexible and fast software supported hardware logging approach for NVM," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2017, Cambridge, MA, USA, October 14-18, 2017*, 2017, pp. 178–190.

[168] T. M. Nguyen and D. Wentzlaff, "PiCL: a software-transparent, persistent cache log for nonvolatile main memory," in *Proceedings of the 51th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2018, Fukuoka, Japan, October 20-24, 2018*, 2018, pp. 178–190.

[169] J. Jeong, C. H. Park, J. Huh, and S. Maeng, "Efficient hardware-assisted logging with asynchronous and direct-update for persistent memory," in *Proceedings of the 51th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2018, Fukuoka, Japan, October 20-24, 2018*, 2018, pp. 178–190.

[170] M. 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 2018, Vienna, Austria, February 24-28, 2018*, 2018, pp. 336–349.

[171] J. Izraelevitz, T. Kelly, and A. Kolli, "Failure-Atomic Persistent Memory Updates via JUSTDO Logging," in *Proceedings of 21th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'16)*, Atlanta, GA, 2016.

[172] A. Joshi, V. Nagarajan, S. Viglas, and M. Cintra, "ATOM: Atomic Durability in Non-volatile Memory through Hardware Logging," in *2017 IEEE International Symposium on High Performance Computer Architecture, HPCA 2017, Austin, TX, USA, February 4-8, 2017*, 2017, pp. 361–372.

[173] K. Doshi, E. Giles, and P. J. Varman, "Atomic persistence for SCM with a non-intrusive backend controller," in *2016 IEEE International Symposium on High Performance Computer Architecture, HPCA 2016, Barcelona, Spain, March 12-16, 2016*, 2016, pp. 77–89.

[174] M. Liu, M. Zhang, K. Chen, X. Qian, Y. Wu, W. Zheng, and J. Ren, "Dudetm: Building durable transactions with decoupling for persistent memory," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '17. New York, NY, USA: ACM, 2017. [Online]. Available: http://doi.acm.org/10.1145/3037697.3037714 pp. 329–343.

[175] J. Ren, J. Zhao, S. Khan, J. Choi, Y. Wu, and O. Mutlu, "ThyNVM: Enabling Software-transparent Crash Consistency in Persistent Memory Systems," in *Proceedings of the 48th International Symposium on Microarchitecture*, ser. MICRO-48. New York, NY, USA: ACM, 2015. [Online]. Available: http://doi.acm.org/10.1145/2830772.2830802 pp. 672–685.

[176] H. Elnawawy, M. Alshboul, J. Tuck, and Y. Solihin, "Efficient Checkpointing of Loop-Based Codes for Non-volatile Main Memory," in *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Sep. 2017, pp. 318–329.

[177] S. Liu, K. Seemakhupt, G. Pekhimenko, A. Kolli, and S. Khan, "Janus: Optimizing Memory and Storage Support for Non-volatile Memory Systems," in *Proceedings of the 46th International Symposium on Computer Architecture*, ser. ISCA '19. New York, NY, USA: ACM, 2019. [Online]. Available: http://doi.acm.org/10.1145/3307650.3322206 pp. 143–156.

[178] K. A. Zubair and A. Awad, "Anubis: Ultra-low Overhead and Recovery Time for Secure Non-volatile Memories," in *Proceedings of the 46th International Symposium on Computer Architecture*, ser. ISCA '19. New York, NY, USA: ACM, 2019. [Online]. Available: http://doi.acm.org/10.1145/3307650.3322252 pp. 157–168.

[179] M. Ye, C. Hughes, and A. Awad, "Osiris: A Low-Cost Mechanism to Enable Restoration of Secure Non-Volatile Memories," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Oct 2018, pp. 403–415.

[180] V. Young, P. J. Nair, and M. K. Qureshi, "DEUCE: Write-Efficient Encryption for Non-Volatile Memories," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '15. New York, NY, USA: ACM, 2015. [Online]. Available: http://doi.acm.org/10.1145/2694344.2694387 pp. 33–44.

[181] T. Wang, S. Sambasivam, Y. Solihin, and J. Tuck, "Hardware Supported Persistent Object Address Translation," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-50 '17. New York, NY, USA: ACM, 2017. [Online]. Available: http://doi.acm.org/10.1145/3123939.3123981 pp. 800–812.

[182] T. Wang, S. Sambasivam, and J. Tuck, "Hardware Supported Permission Checks on Persistent Objects for Performance and Programmability," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, June 2018, pp. 466–478.

[183] Oracle, "Using Application Data Integrity (ADI)." [Online]. Available: https://docs.oracle.com/cd/E53394_01/html/E54815/gqajs.html

[184] K. Serebryany, E. Stepanov, A. Shlyapnikov, V. Tsyrklevich, and D. Vyukov, "Memory Tagging and how it improves C/C++ memory safety," *CoRR*, vol. abs/1802.09517, 2018. [Online]. Available: https://arxiv.org/abs/1802.09517

[185] W. Vogels, "All Things Distributed." https://www.allthingsdistributed.com/2010/02/strong_consistency_simpledb.html, 2010.

[186] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's Highly Available Key-Value Store." *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 6, p. 205–220, Oct. 2007. [Online]. Available: https://doi.org/10.1145/1323293.1294281

[187] Y. Matsunobu, "MyRocks: A space- and write-optimized MySQL database." https://engineering.fb.com/2016/08/31/core-data/myrocks-a-space-and-write-optimized-mysql-database/.

[188] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A Distributed Storage System for Structured Data." *ACM Trans. Comput. Syst.*, vol. 26, no. 2, June 2008. [Online]. Available: https://doi.org/10.1145/1365815.1365816

[189] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni, "PNUTS: Yahoo!'s Hosted Data Serving Platform." *Proc. VLDB Endow.*, vol. 1, no. 2, p. 1277–1288, Aug. 2008. [Online]. Available: https://doi.org/10.14778/1454159.1454167

[190] Oracle, "Oracle NoSQL Database: Fast, Reliable, Predictable. An Oracle white paper." https://www.oracle.com/technetwork/database/nosqldb/learnmore/nosql-database-498041.pdf, June 2018.

[191] P. Viotti and M. Vukolić, "Consistency in Non-Transactional Distributed Storage Systems." *ACM Comput. Surv.*, vol. 49, no. 1, June 2016. [Online]. Available: https://doi.org/10.1145/2926965

[192] Apache, "ZooKeeper." https://zookeeper.apache.org/.

[193] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "ZooKeeper: Wait-Free Coordination for Internet-Scale Systems." in *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, ser. USENIXATC'10. USA: USENIX Association, 2010, p. 11.

[194] W. Zhang, S. Shenker, and I. Zhang, "Persistent State Machines for Recoverable In-memory Storage Systems with NVRam." in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*, Nov. 2020.

[195] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani, "Scaling Memcache at Facebook." in *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. Lombard, IL: USENIX Association, Apr. 2013. [Online]. Available: https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/nishtala pp. 385–398.

[196] Z. Wang, X. Liu, J. Yang, T. Michailidis, S. Swanson, and J. Zhao, "Characterizing and Modeling Non-Volatile Memory Systems." in *Proceedings of the 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020.

[197] R. Zambre, M. Grodowitz, A. Chandramowlishwaran, and P. Shamis, “Breaking Band: A Breakdown of High-Performance Communication.” ser. ICPP 2019. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3337821.3337910

[198] Y. Ajima, T. Kawashima, T. Okamoto, N. Shida, K. Hirai, T. Shimizu, S. Hiramoto, Y. Ikeda, T. Yoshikawa, K. Uchida, and T. Inoue, “The Tofu Interconnect D.” in *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, 2018, pp. 646–654.

[199] Mellanox Technologies, “White paper: Introducing 200G HDR InfiniBand Solutions.” 350 Oakmead Parkway, Suite 100, Sunnyvale, CA 94085, Tech. Rep. 060058WP, 2019. [Online]. Available: https://www.mellanox.com/files/doc-2020/wp-introducing-200g-hdr-infiniband-solutions.pdf

[200] R. Neugebauer, G. Antichi, J. F. Zazo, Y. Audzevich, S. López-Buedo, and A. W. Moore, “Understanding PCIe Performance for End Host Networking.” in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM ’18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3230543.3230560 p. 327–341.

[201] P. Bianco, “New Fabric Interconnects: A comparison between Omni-Path and EDR Infiniband Architectures.” 2017. [Online]. Available: https://agenda.infn.it/event/13040/contributions/17299/attachments/12506/14064/INFN_CCR_2017_-_DELLEMC_v2.pdf

[202] V. Gogte, W. Wang, S. Diestelhorst, P. M. Chen, S. Narayanasamy, and T. F. Wenisch, “Relaxed Persist Ordering Using Strand Persistency.” in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture*, 2020, pp. 652–665.

[203] A. S. Tanenbaum and M. v. Steen, *Distributed Systems: Principles and Paradigms (2nd Edition)*. USA: Prentice-Hall, Inc., 2006.

[204] A. Katsarakis, V. Gavrielatos, M. S. Katebzadeh, A. Joshi, A. Dragojevic, B. Grot, and V. Nagarajan, “Hermes: A Fast, Fault-Tolerant and Linearizable Replication Protocol.” in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS ’20. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3373376.3378496 p. 201–217.

[205] G. Coulouris, J. Dollimore, T. Kindberg, and G. Blair, *Distributed Systems: Concepts and Design*, 5th ed. USA: Addison-Wesley Publishing Company, 2011.

[206] D. Ongaro and J. Ousterhout, “In Search of an Understandable Consensus Algorithm.” in *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. Philadelphia, PA: USENIX Association, June 2014. [Online]. Available: https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro pp. 305–319.

[207] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen, "Don't Settle for Eventual: Scalable Causal Consistency for Wide-Area Storage with COPS." in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, ser. SOSP '11. New York, NY, USA: Association for Computing Machinery, 2011. [Online]. Available: https://doi.org/10.1145/2043556.2043593 p. 401–416.

[208] R. Alagappan, A. Ganesan, J. Liu, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, "Fault-Tolerance, Fast and Slow: Exploiting Failure Asynchrony in Distributed Systems." in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18).* Carlsbad, CA: USENIX Association, Oct. 2018. [Online]. Available: https://www.usenix.org/conference/osdi18/presentation/alagappan pp. 390–408.

[209] P. Bailis, A. Ghodsi, J. M. Hellerstein, and I. Stoica, "Bolt-on Causal Consistency." in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '13. New York, NY, USA: Association for Computing Machinery, 2013. [Online]. Available: https://doi.org/10.1145/2463676.2465279 p. 761–772.

[210] S. B. Davidson, H. Garcia-Molina, and D. Skeen, "Consistency in a Partitioned Network: A Survey." *ACM Comput. Surv.*, vol. 17, no. 3, p. 341–370, Sep. 1985. [Online]. Available: https://doi.org/10.1145/5505.5508

[211] J. Izraelevitz, H. Mendes, and M. L. Scott, "Linearizability of Persistent Memory Objects Under a Full-System-Crash Failure Model." in *Distributed Computing*, C. Gavoille and D. Ilcinkas, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 313–327.

[212] S. A. Mehdi, C. Littley, N. Crooks, L. Alvisi, N. Bronson, and W. Lloyd, "I Can't Believe It's Not Causal! Scalable Causal Consistency with No Slowdown Cascades." in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17).* Boston, MA: USENIX Association, Mar. 2017. [Online]. Available: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/mehdi pp. 453–468.

[213] W. Vogels, "Eventually Consistent: Building Reliable Distributed Systems at a Worldwide Scale Demands Trade-Offs? Between Consistency and Availability." *Queue*, vol. 6, no. 6, p. 14–19, Oct. 2008. [Online]. Available: https://doi.org/10.1145/1466443.1466448

[214] A. Joshi, V. Nagarajan, M. Cintra, and S. Viglas, "Efficient persist barriers for multicores." in *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture*, 2015, pp. 660–671.

[215] C. Lin, V. Nagarajan, and R. Gupta, "Fence Scoping." in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '14. IEEE Press, 2014. [Online]. Available: https://doi.org/10.1109/SC.2014.14 p. 105–116.

[216] SNIA, "NVM PM Remote Access for High Availability (Technical White Paper)." https://www.snia.org/sites/default/files/technical_work/Whitepapers/NVM-PM-Remote-Access-for-High-Availability.pdf, May 2019.

[217] T. Talpey, "RDMA Persistent Memory Extensions." https://www.openfabrics.org/wp-content/uploads/209_TTalpey.pdf, March 2019, (last accessed on 11/20/2020).

[218] "Redis." https://redis.io/, (last accessed on 11/20/2020).

[219] "LogCabin." https://github.com/logcabin/logcabin, (last accessed on 11/20/2020).

[220] G. Vasilis, A. Katsarakis, and V. Nagarajan, "Odyssey: The Impact of Modern Hardware on Strongly-Consistent Replication Protocols." in *Proceedings of the Sixteenth European Conference on Computer Systems (EuroSys '21)*. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3447786.3456240

[221] "Hermes Reliable Replication Protocol." https://github.com/ease-lab/Hermes, (last accessed on 04/07/2021).

[222] "Apache Zookeeper." https://github.com/apache/zookeeper, (last accessed on 04/07/2021).

[223] A. Farshin, A. Roozbeh, G. Q. M. Jr., and D. Kostić, "Reexamining Direct Cache Access to Optimize I/O Intensive Applications for Multi-hundred-gigabit Networks." in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, July 2020. [Online]. Available: https://www.usenix.org/conference/atc20/presentation/farshin pp. 673–689.

[224] Intel, "Intel Data Direct I/O Technology Overview." Feb 2012. [Online]. Available: https://www.intel.com/content/www/us/en/io/data-direct-i-o-technology-brief.html

[225] S. Novakovic, A. Daglis, E. Bugnion, B. Falsafi, and B. Grot, "Scale-out NUMA." in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '14. New York, NY, USA: Association for Computing Machinery, 2014. [Online]. Available: https://doi.org/10.1145/2541940.2541965 p. 3–18.

[226] A. Daglis, D. Ustiugov, S. Novaković, E. Bugnion, B. Falsafi, and B. Grot, "SABRes: Atomic Object Reads for in-Memory Rack-Scale Computing." in *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-49. IEEE Press, 2016.

[227] A. Tavakkol, A. Kolli, S. Novakovic, K. Razavi, J. Gómez-Luna, H. Hassan, C. Barthels, Y. Wang, M. Sadrosadati, S. Ghose, A. Singla, P. Subrahmanyam, and O. Mutlu, "Enabling Efficient RDMA-based Synchronous Mirroring of Persistent Memory Transactions." *CoRR*, vol. abs/1810.09360, 2018. [Online]. Available: http://arxiv.org/abs/1810.09360

[228] X. Hu, M. Ogleari, J. Zhao, S. Li, A. Basak, and Y. Xie, "Persistence Parallelism Optimization: A Holistic Approach from Memory Bus to RDMA Network." in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018, pp. 494–506.

[229] Z. Wang, X. Wang, Z. Qian, B. Ye, and S. Lu, "RDMAvisor: Toward Deploying Scalable and Simple RDMA as a Service in Datacenters." *CoRR*, vol. abs/1802.01870, 2018. [Online]. Available: http://arxiv.org/abs/1802.01870

[230] Brad Fitzpatrick, "Distributed Caching with Memcached." *Linux Journal*, Aug 2004.

[231] "Google Code, cpp-btree." https://code.google.com/archive/p/cpp-btree/, December 2007, (retrieved 27/08/2020).

[232] T. Bingmann, "TLX: Collection of Sophisticated C++ Data Structures, Algorithms, and Miscellaneous Helpers." 2018, https://panthema.net/tlx, retrieved Oct. 7, 2020.

[233] D. B. Terry, V. Prabhakaran, R. Kotla, M. Balakrishnan, M. K. Aguilera, and H. Abu-Libdeh, "Consistency-Based Service Level Agreements for Cloud Storage." in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13. New York, NY, USA: Association for Computing Machinery, 2013. [Online]. Available: https://doi.org/10.1145/2517349.2522731 p. 309–324.

[234] M. K. Aguilera and D. B. Terry, "The Many Faces of Consistency," *IEEE Data Engineering Bulletin*, vol. 39, pp. 3–13, 2016.

[235] C. Xie, C. Su, C. Littley, L. Alvisi, M. Kapritsos, and Y. Wang, "High-Performance ACID via Modular Concurrency Control." in *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP'15)*, Monterey, California, 2015.

[236] S. Mu, Y. Cui, Y. Zhang, W. Lloyd, and J. Li, "Extracting More Concurrency from Distributed Transactions." in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, Broomfield, CO, Oct. 2014.

[237] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford, "Spanner: Google's Globally-Distributed Database." in *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, Hollywood, CA, Oct. 2012.

[238] H. Lu, K. Veeraraghavan, P. Ajoux, J. Hunt, Y. J. Song, W. Tobagus, S. Kumar, and W. Lloyd, "Existential Consistency: Measuring and Understanding Consistency at Facebook." in *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP'15)*, Monterey, California, 2015.

[239] C. Li, J. Leitão, A. Clement, N. Preguiça, R. Rodrigues, and V. Vafeiadis, "Automating the Choice of Consistency Levels in Replicated Systems." in *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, Philadelphia, PA, June 2014.

[240] M. R. Rahman, L. Tseng, S. Nguyen, I. Gupta, and N. Vaidya, “Characterizing and Adapting the Consistency-Latency Tradeoff in Distributed Key-Value Stores.” *ACM Trans. Auton. Adapt. Syst.*, vol. 11, no. 4, Jan. 2017.

[241] W. J. Bolosky, D. Bradshaw, R. B. Haagens, N. P. Kusters, and P. Li, “Paxos Replicated State Machines as the Basis of a High-Performance Data Store.” in *NSDI*, 2011.

[242] C. Lee, S. J. Park, A. Kejriwal, S. Matsushita, and J. Ousterhout, “Implementing Linearizability at Large Scale and Low Latency.” in *Proceedings of the 25th Symposium on Operating Systems Principles*, ser. SOSP ’15. New York, NY, USA: Association for Computing Machinery, 2015. [Online]. Available: https://doi.org/10.1145/2815400.2815416 p. 71–86.

[243] A. Kolli, V. Gogte, A. Saidi, S. Diestelhorst, P. M. Chen, S. Narayanasamy, and T. F. Wenisch, “Language-Level Persistency.” in *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA’17)*, Toronto, ON, Canada, 2017.

[244] Y. Zhang, J. Yang, A. Memaripour, and S. Swanson, “Mojim: A Reliable and Highly-Available Non-Volatile Memory System.” in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS ’15. New York, NY, USA: Association for Computing Machinery, 2015. [Online]. Available: https://doi.org/10.1145/2694344.2694370 p. 3–18.

[245] A. Dragojević, D. Narayanan, E. B. Nightingale, M. Renzelmann, A. Shamis, A. Badam, and M. Castro, “No Compromises: Distributed Transactions with Consistency, Availability, and Performance.” in *Proceedings of the 25th Symposium on Operating Systems Principles*, ser. SOSP ’15. New York, NY, USA: Association for Computing Machinery, 2015. [Online]. Available: https://doi.org/10.1145/2815400.2815425 p. 54–70.

[246] Y. Lu, J. Shu, Y. Chen, and T. Li, “Octopus: An RDMA-Enabled Distributed Persistent Memory File System.” in *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference*, ser. USENIX ATC ’17. USA: USENIX Association, 2017, p. 773–785.

[247] Y. Shan, S.-Y. Tsai, and Y. Zhang, “Distributed Shared Persistent Memory.” in *Proceedings of the 2017 Symposium on Cloud Computing.* New York, NY, USA: Association for Computing Machinery, 2017. [Online]. Available: https://doi.org/10.1145/3127479.3128610 p. 323–337.

[248] J. Yang, J. Izraelevitz, and S. Swanson, “FileMR: Rethinking RDMA Networking for Scalable Persistent Memory.” in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20).* Santa Clara, CA: USENIX Association, Feb. 2020. [Online]. Available: https://www.usenix.org/conference/nsdi20/presentation/yang pp. 111–125.

[249] S.-Y. Tsai, Y. Shan, and Y. Zhang, "Disaggregating Persistent Memory and Controlling Them Remotely: An Exploration of Passive Disaggregated Key-Value Stores." in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, July 2020.

[250] "RDMA Extensions for Enhanced Memory Placement." https://tools.ietf.org/id/draft-talpey-rdma-commit-01.html, (last accessed on 11/20/2020).

[251] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. C. Li, M. Marchukov, D. Petrov, L. Puzar, Y. J. Song, and V. Venkataramani, "TAO: Facebook's Distributed Data Store for the Social Graph," in *2013 USENIX Annual Technical Conference, San Jose, CA, USA, June 26-28, 2013*, A. Birrell and E. G. Sirer, Eds. USENIX Association, 2013. [Online]. Available: https://www.usenix.org/conference/atc13/technical-sessions/presentation/bronson pp. 49–60.

[252] "Apache cassandra." https://cassandra.apache.org, 2015, (last accessed on 11/20/2021).

[253] "Mysql." https://mysql.com, 2015, (last accessed on 11/20/2021).

[254] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson, "FaRM: Fast Remote Memory," in *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. Seattle, WA: USENIX Association, Apr. 2014, pp. 401–414.

[255] A. Katsarakis, Y. Ma, Z. Tan, A. Bainbridge, M. Balkwill, A. Dragojevic, B. Grot, B. Radunovic, and Y. Zhang, "Zeus: Locality-Aware Distributed Transactions," in *Proceedings of the Sixteenth European Conference on Computer Systems (EuroSys'21)*, ser. EuroSys '21, Online Event, United Kingdom, 2021.

[256] T. Härder and A. Reuter, "Principles of Transaction-Oriented Database Recovery," *ACM Comput. Surv.*, vol. 15, no. 4, pp. 287–317, 1983. [Online]. Available: https://doi.org/10.1145/289.291

[257] J. Gray, "The Transaction Concept: Virtues and Limitations (Invited Paper)," in *Very Large Data Bases, 7th International Conference, September 9-11, 1981, Cannes, France, Proceedings*. IEEE Computer Society, 1981, pp. 144–154.

[258] Mellanox Technologies, "InfiniBand EDR 100Gb/s Routers," Tech. Rep., 2020. [Online]. Available: https://www.mellanox.com/files/doc-2020/pb-edr-ib-router.pdf

[259] P. Grun, "Introduction to infiniband for end users," White Paper, InfiniBand Trade Association, 2010. [Online]. Available: http://www.mellanox.com/pdf/whitepapers/Intro_to_IB_for_End_Users.pdf

[260] Mellanox Technologies, "ConnectX-6 VPI Card: 200Gb/s InfiniBand & Ethernet Adapter Card," 2020. [Online]. Available: https://www.mellanox.com/files/doc-2020/pb-connectx-6-vpi-card.pdf

[261] L. A. Barroso, M. Marty, D. A. Patterson, and P. Ranganathan, "Attack of the killer microseconds," *Commun. ACM*, vol. 60, no. 4, pp. 48–54, 2017. [Online]. Available: https://doi.org/10.1145/3015146

[262] S. Cho, A. Suresh, T. Palit, M. Ferdman, and N. Honarmand, "Taming the Killer Microsecond," ser. MICRO-51. IEEE Press, 2018. [Online]. Available: https://doi.org/10.1109/MICRO.2018.00057 p. 627–640.

[263] B. Li, Z. Ruan, W. Xiao, Y. Lu, Y. Xiong, A. Putnam, E. Chen, and L. Zhang, "Kv-direct: High-performance in-memory key-value store with programmable nic," in *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP'17)*, Shanghai, China, 2017.

[264] C. Xie, C. Su, C. Littley, L. Alvisi, M. Kapritsos, and Y. Wang, "High-performance acid via modular concurrency control," in *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP'15)*, Monterey, California, 2015.

[265] A. Kalia, M. Kaminsky, and D. G. Andersen, "FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided (RDMA) Datagram RPCs," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. Savannah, GA: USENIX Association, Nov. 2016. [Online]. Available: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/kalia pp. 185–201.

[266] X. Wei, J. Shi, Y. Chen, R. Chen, and H. Chen, "Fast in-memory transaction processing using rdma and htm," in *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP'15)*, Monterey, California, 2015.

[267] H. N. Schuh, W. Liang, M. Liu, J. Nelson, and A. Krishnamurthy, "Xenic: Smartnic-accelerated distributed transactions," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP'21)*, Virtual Event, Germany, 2021.

[268] X. Wei, Z. Dong, R. Chen, and H. Chen, "Deconstructing rdma-enabled distributed transactions: Hybrid is better!" in *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, A. C. Arpaci-Dusseau and G. Voelker, Eds. USENIX Association, 2018. [Online]. Available: https://www.usenix.org/conference/osdi18/presentation/wei pp. 233–251.

[269] Y. Chen, X. Wei, J. Shi, R. Chen, and H. Chen, "Fast and general distributed transactions using RDMA and HTM," in *Proceedings of the Eleventh European Conference on Computer Systems, EuroSys 2016, London, United Kingdom, April 18-21, 2016*, C. Cadar, P. R. Pietzuch, K. Keeton, and R. Rodrigues, Eds. ACM, 2016. [Online]. Available: https://doi.org/10.1145/2901318.2901349 pp. 26:1–26:17.

[270] A. Shamis, M. Renzelmann, S. Novakovic, G. Chatzopoulos, A. T. Gjerdrum, D. Alistarh, A. Dragojevic, D. Narayanan, and M. Castro, "Fast General Distributed Transactions with Opacity using Global Time," *CoRR*, vol. abs/2006.14346, 2020. [Online]. Available: https://arxiv.org/abs/2006.14346

[271] S. Das, S. Nishimura, D. Agrawal, and A. El Abbadi, "Albatross: Lightweight Elasticity in Shared Storage Databases for the Cloud Using Live Data Migration," *Proc. VLDB Endow.*, vol. 4, no. 8, p. 494–505, May 2011. [Online]. Available: https://doi.org/10.14778/2002974.2002977

[272] A. J. Elmore, S. Das, D. Agrawal, and A. El Abbadi, "Zephyr: Live Migration in Shared Nothing Databases for Elastic Cloud Platforms," in *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '11. New York, NY, USA: Association for Computing Machinery, 2011. [Online]. Available: https://doi.org/10.1145/1989323.1989356 p. 301–312.

[273] A. Dey, A. Fekete, R. Nambiar, and U. Röhm, "YCSB+T: Benchmarking web-scale transactional databases," in *2014 IEEE 30th International Conference on Data Engineering Workshops*, 2014, pp. 223–230.

[274] W. W. Peterson and D. T. Brown, "Cyclic Codes for Error Detection," *Proceedings of the IRE*, vol. 49, no. 1, pp. 228–235, 1961.

[275] M. Walma, "Pipelined Cyclic Redundancy Check (CRC) Calculation," in *Proceedings of the 16th International Conference on Computer Communications and Networks, IEEE ICCCN 2007, Turtle Bay Resort, Honolulu, Hawaii, USA, August 13-16, 2007*. IEEE, 2007. [Online]. Available: https://doi.org/10.1109/ICCCN.2007.4317846 pp. 365–370.

[276] T. Harris, J. Larus, and R. Rajwar, *Transactional Memory, 2nd Edition*. Morgan & Claypool, 2010.

[277] A. Ruhela, S. Xu, K. Manian, H. Subramoni, and D. Panda, "Analyzing and Understanding the Impact of Interconnect Performance on HPC, Big Data, and Deep Learning Applications: A Case Study with InfiniBand EDR and HDR," 05 2020, pp. 869–878.

[278] TPC-C. TPC benchmark C., http://www.tpc.org/tpcc/, 2018.

[279] Simon Neuvonen and Antoni Wolski and Markku Manner and Viho Raatikka, "Telecom Application Transaction Processing Benchmark," http://tatpbenchmark.sourceforge.net/, 2011.

[280] The H-STORE Team. Smallbank Benchmark., https://hstore.cs.brown.edu/documentation/deployment/benchmarks/smallbank/, 2013.

[281] M. Alomari, M. Cahill, A. Fekete, and U. Rohm, "The Cost of Serializability on Platforms That Use Snapshot Isolation," in *2008 IEEE 24th International Conference on Data Engineering*, 2008, pp. 576–585.

[282] C. Xie, C. Su, M. Kapritsos, Y. Wang, N. Yaghmazadeh, L. Alvisi, and P. Mahajan, "Salt: Combining ACID and BASE in a Distributed Database," in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14)*, Broomfield, CO, 2014.

[283] S. Mu, Y. Cui, Y. Zhang, W. Lloyd, and J. Li, "Extracting more concurrency from distributed transactions." in *OSDI*, 2014.

[284] I. Zhang, N. K. Sharma, A. Szekeres, A. Krishnamurthy, and D. R. K. Ports, "Building consistent transactions with inconsistent replication," in *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP'15)*, ser. SOSP '15, Monterey, California, 2015.

[285] X. Wei, R. Chen, H. Chen, Z. Wang, Z. Gong, and B. Zang, "Unifying timestamp with transaction ordering for MVCC with decentralized scalar timestamp," in *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI'21)*. USENIX Association, Apr. 2021.

[286] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi, "Calvin: Fast distributed transactions for partitioned database systems," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '12, Scottsdale, Arizona, USA, 2012.

[287] M. Burke, S. Dharanipragada, S. Joyner, A. Szekeres, J. Nelson, I. Zhang, and D. R. K. Ports, "PRISM: Rethinking the RDMA Interface for Distributed Systems," in *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*, R. van Renesse and N. Zeldovich, Eds. ACM, 2021. [Online]. Available: https://doi.org/10.1145/3477132.3483587 pp. 228–242.

[288] S. Di Girolamo, A. Kurth, A. Calotoiu, T. Benz, T. Schneider, J. Beránek, L. Benini, and T. Hoefler, "A risc-v in-network accelerator for flexible high-performance low-power packet processing," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA'21)*, 2021.

[289] B. Pismenny, H. Eran, A. Yehezkel, L. Liss, A. Morrison, and D. Tsafrir, "Autonomous nic offloads," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'21)*, Virtual, USA, 2021.

[290] N. Lazarev, S. Xiang, N. Adit, Z. Zhang, and C. Delimitrou, "Dagger: Efficient and fast rpcs in cloud microservices with near-memory reconfigurable nics," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'21)*, Virtual, USA, 2021.

[291] D. Kim, A. Memaripour, A. Badam, Y. Zhu, H. H. Liu, J. Padhye, S. Raindel, S. Swanson, V. Sekar, and S. Seshan, "Hyperloop: Group-based nic-offloading to accelerate replicated transactions in multi-tenant storage systems," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM'18)*, Budapest, Hungary, 2018.

[292] H. Li, M. Hao, S. Novakovic, V. Gogte, S. Govindan, D. R. K. Ports, I. Zhang, R. Bianchini, H. S. Gunawi, and A. Badam, "Leapio: Efficient and portable virtual nvme storage on arm socs," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'20)*, Lausanne, Switzerland, 2020.

[293] M. Tork, L. Maudlej, and M. Silberstein, "Lynx: A smartnic-driven accelerator-centric architecture for network servers," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'20)*, Lausanne, Switzerland, 2020.

[294] P. M. Phothilimthana, M. Liu, A. Kaufmann, S. Peter, R. Bodik, and T. Anderson, "Floem: A programming system for nic-accelerated network applications," in *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation (OSDI'18)*, Carlsbad, CA, USA, 2018.

[295] J. Kim, I. Jang, W. Reda, J. Im, M. Canini, D. Kostić, Y. Kwon, S. Peter, and E. Witchel, "Linefs: Efficient smartnic offload of a distributed file system with pipeline parallelism," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP'21)*, Virtual Event, Germany, 2021.

[296] K. Seemakhupt, S. Liu, Y. Senevirathne, M. Shahbaz, and S. Khan, "PMNet: In-Network Data Persistence," in *Proceedings of the 2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA'21)*, 2021.

[297] D. Skarlatos, A. Kokolis, T. Xu, and J. Torrellas, *Elastic Cuckoo Page Tables: Rethinking Virtual Memory Translation for Parallelism.* New York, NY, USA: Association for Computing Machinery, 2020, p. 1093–1108. [Online]. Available: https://doi.org/10.1145/3373376.3378493

[298] J. Stojkovic, D. Skarlatos, A. Kokolis, T. Xu, and J. Torrellas, *Parallel Virtualized Memory Translation with Nested Elastic Cuckoo Page Tables.* New York, NY, USA: Association for Computing Machinery, 2022.

[299] A. Franques, A. Kokolis, S. Abadal, V. Fernando, S. Misailovic, and J. Torrellas, "WiDir: A Wireless-Enabled Directory Cache Coherence Protocol," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2021, pp. 304–317.