Exploring the Boundaries of Operating System in the Era of Ultra-fast Storage Technologies

R. Madhava Krishnan

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Engineering

Changwoo Min, Chair
Ali R. Butt
Dimitrios Nikolopoulos
Leyla Nazhand-Ali
Cameron Patterson

April 28, 2023
Blacksburg, Virginia

Keywords: Kernel-bypass storage stack, Non-volatile Memory, Concurrency, Multi-core Scalability, Operating System, Computational Storage

Exploring the Boundaries of Operating System in the Era of Ultra-fast Storage Technologies

R. Madhava Krishnan

(ABSTRACT)

The storage hardware is evolving at a rapid pace to keep up with the exponential rise of data consumption. Recently, ultra-fast storage technologies such as nano-second scale byte-addressable Non-Volatile Memory (NVM), micro-second scale SSDs are being commercialized. However, the OS storage stack has not been evolving fast enough to keep up with these new ultra-fast storage hardware. Hence, the latency due user-kernel context switch caused by system calls and hardware interrupts is no longer negligible as presumed in the era of slower high latency hard disks. Further, the OS storage stack is not designed with multi-core scalability in mind; so with CPU core count continuously increasing, the OS storage stack particularly the Virtual Filesystem (VFS) and filesystem layer are increasingly becoming a scalability bottleneck.

Applications bypass the kernel (kernel-bypass storage stack) completely to eliminate the storage stack from becoming a performance and scalability bottleneck. But this comes at the cost of programmability, isolation, safety, and reliability. Moreover, scalability bottlenecks in the filesystem can not be addressed by simply moving the filesystem to the userspace. Overall, while designing a kernel-bypass storage stack looks obvious and promising there are several critical challenges in the aspects of programmability, performance, scalability, safety, and reliability that needs to be addressed to bypass the traditional OS storage stack.

This thesis proposes a series of kernel-bypass storage techniques designed particularly for fast memory-centric storage. First, this thesis proposes a scalable persistent transactional memory (PTM) programming model to address the programmability and multi-core scalability challenges. Next, this thesis proposes techniques to make the PTM memory safe and fault tolerant. Further, this thesis also proposes a kernel-bypass programming framework to port legacy DRAM based in-memory database applications to run on persistent memory-centric storage. Finally, this thesis explores an application driven approach to address the CPU side and storage side bottlenecks in the deep learning model training by proposing a kernel-bypass

programming framework to move compute closer to the storage. Overall, the techniques proposed in this thesis will be a strong foundation for the applications to adopt and exploit the emerging ultra-fast storage technologies without being bottlenecked by the traditional OS storage stack.

Exploring the Boundaries of Operating System in the Era of Ultra-fast Storage Technologies

R. Madhava Krishnan

(GENERAL AUDIENCE ABSTRACT)

The storage hardware is evolving at a rapid pace to keep up with the exponential rise of data consumption. Recently, ultra-fast storage technologies such as nano-second scale byte-addressable Non-Volatile Memory (NVM), micro-second scale SSDs are being commercialized. The Operating System (OS) has been the gateway for the applications to access and manage the storage hardware. Unfortunately, the OS storage stack that is designed with slower storage technologies (*e.g.*, hard disk drives) becomes a performance, scalability, and programmability bottleneck for the emerging ultra-fast storage technologies. This has created a large gap between the storage hardware advancements and the system software support for such emerging storage technologies. Consequently, applications are constrained by the limitations of the OS storage stack when they intend to explore these emerging storage technologies.

In this thesis, we propose a series of novel kernel-bypass storage stack designs to address the performance, scalability, and programmability limitations in the conventional OS storage stack. The kernel-bypass storage stack proposed in this thesis are carefully designed with ultra-fast modern storage hardware in mind. Application developers can leverage the kernel-bypass techniques proposed in this thesis to develop new application or port the legacy applications to use the emerging ultra-fast storage technologies without being constrained by the limitations of the conventional OS storage stack.

This work is dedicated to my

Mom, Nappinnai Ramanathan,

Dad, Ramanathan Sivasankaran,

and my little brother, Hemanth Krishnan Ramanathan

Acknowledgments

I am blessed to work with and learn from some amazing people throughout my PhD journey. First, I want to thank my advisor Dr. Changwoo Min for his unparalleled support and commitment, without Changwoo none of this would be possible. I want to thank Changwoo for trusting my abilities even when the things were not in my favour. I can proudly say that I've learn the art of systems research from one of the best in the field. Next, I would like to thank my amazing postdocs Dr. Jaeho Kim and Dr. Wook-Hee Kim. Jaeho and Wookhee are the two most talented, hardworking, and humble people I've come across so far. They were also my very good friends and I am happy to share my research journey with them. I would like to thank my Masters advisor Dr. Wei-Ming Lin who introduced me to the systems research when I first came to the US right after my undergrad. Dr. Lin has always been a person from whom I take a lot of inspiration and I owe my deepest gratitude for all his support and encouragement.

Next, I would like to thank my collaborators, Dr. Sudarsun Kannan and Dr. Sanidhya Kashyap for their effort towards making this research possible and also for writing recommendation letters whenever I ask for one. I also thank Dr. Ali Butt and Dr. Dimitrios Nikolopoulos for serving in my PhD committee and for their invaluable support and encouragement. I've also been fortunate to collaborate with my amazing labmates particularly Shashwat, Sanjana, Xinwei, Ajit, Kris, and Honda.

I've been fortunate to have such a amazing group of friends who have been with me through the thick of things particularly, Raj Prabhu, Mahesh Narayanamurthi, Abinabh, Lakshman, and Sudharsan. I would like to thank the god almighty for giving me the strength and intent to get through this PhD journey. Finally, I am indebted to my dad, mom, and my brother for their unparalleled support and encouragement. They've always been my pillar of support and my well wishers, without their support none of this would be possible.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The storage hardware is evolving at a rapid pace to keep up with the exponential rise of data consumption. Modern Machine Learning, social media applications are driven by large amounts of user data to make user-specific decisions and recommendations [14, 15]. Storage hardware is evolving in the aspects of cost, performance, capacity etc. Recently, ultra-fast storage technologies such as nano-second scale byte-addressable Non-Volatile Memory (NVM) [20, 26, 34, 36], micro-second scale SSDs [41, 42, 70, 71] are being commercialized. Further, storage hardware nowadays are being equipped with compute capabilities which enable them to perform large compute intensive operations near the storage without having to rely on the CPUs [55, 58, 59, 207].

The Operating System (OS), particularly the storage stack has been the de-facto gateway for the applications to manage and access these storage hardware. However, the OS storage stack has not been evolving fast enough to keep up with these new ultra-fast storage hardware. For instance, NVMe SSDs have a read and write latency of $7\mu s$ and $18\mu s$ respectively [41, 71] and NVM's read and write latency are $169ns$ and $62ns$ [26, 136] respectively. At this scale, the latency due user-kernel context switch caused by system calls and hardware interrupts is no longer negligible as presumed in the era of slower high latency hard disks. In a nutshell,

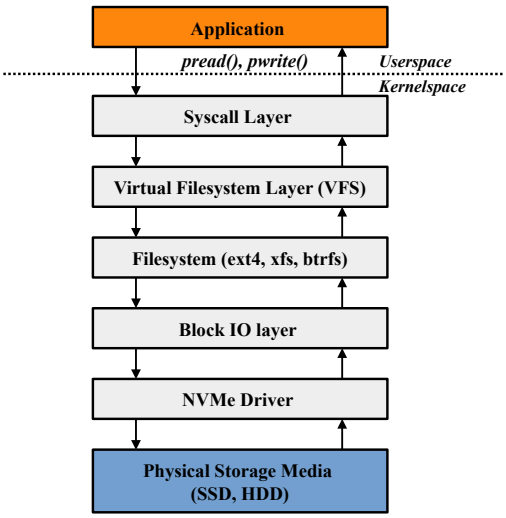

Figure 1.1: A simplified Linux storage stack

the arrival of ultra-fast storage hardware is exposing some of the hidden assumptions in the OS storage stack design.

1.1 Hidden Assumptions in the Storage Stack Design

First, the long held assumption that the OS storage stack latency is neglibile as compared to the storage media access latency is no longer true. Figure 1.1 shows IO path of a read/write request to a physical storage media through OS storage stack layers. It is estimated that ∼50% of time is spent on these storage stack layers when writing to a micro-second scale SSD [276]. This time will be much higher for nano-second scale devices such as the NVM and most probably for any future ultra-fast storage technology. With such a deep and high latency storage stack applications are incapable of tapping the benefits of these ultra-fast storage hardware to the fullest.

Second, the storage stack is designed with an assumption that a concurrent access is a special case. That is, the storage stack design does not primarily focus on concurrent applications as they were not common several decades back. This is true back in the time

when applications rely on the CPU vendors to increase the performance by improving the CPU frequency. But, as the CPU frequency increase is hitting physical limitations, CPU vendors scale the CPU performance by adding multiple CPU cores (multi-core CPU architecture) and building servers with NUMA architecture. Currently, server with multiple sockets each having multiple cores are pretty common and the number of CPU cores are increasing. Nowadays, a server with 4 NUMA sockets and 200 CPU cores are very common in the datacenters [5, 30]. Even the mobile CPU architectures are largely shifting towards the multi-core trend and 8-16 CPU cores in mobile processors are becoming common [6, 7, 24, 47]. Consequently, applications are becoming concurrent to exploit the multi-core CPUs and in fact this is inevitable for the application's performance scalability.

However, the OS storage stack is not designed for such a high concurrency and many scalability bottlenecks are being reveled by the OS researchers. One good example is the scalability bottlenecks in the VFS layer; the inode list lock (readers-writer lock) and directory access lock (mutex) becomes a scalability bottleneck under concurrent accesses even for read operations and direct IO [56, 197]. Similar scalability bottlenecks exist in the filesystem layer of the storage stack as well; popular filesystem such as the ext4, XFS suffer from poor scalability due to journalling operations as they are not designed for multicore CPUs [197]. This means that as much as the application is optimized for concurrent accesses it will be eventually bottlenecked by the poor scalability of the different storage stack layers. Nevertheless, for a foreseeable future multi-core machines are going to be ubiquitous and concurrency is inevitable for the application scalability.

The third and the final hidden assumption in the storage stack design is that ***the CPU and memory are faster than IO devices***; but this assumption is starting to change lately with the rapid performance improvement of the IO devices. Currently IO devices are becoming faster while the CPU scaling has plateaued as we approach the end of the

Moore's law. Also, the memory bandwidth and capacity scaling of traditional DDR interface is starting to saturate at its peak point [237]. Overall, in the last ten years, the CPU-DRAM bandwidth has improved 2× while the IO bandwidth has increased by more than 12× with the arrival of new ultra-fast storage technologies [4].

This trend now exposes the dichotomy of memory and storage by exposing the high data copy overhead. The data movement between storage and memory is increasingly becoming critical performance bottleneck particularly in the applications that operate on petabytes of data (*e.g.*, Deep Learning training) as they can not fit the entire dataset in the memory and consequently they rely on the storage devices to store the dataset. This results in applications constantly moving their working dataset between storage and memory.

1.2 Kernel-Bypass Storage Stack Design

Applications bypass the kernel completely to eliminate the storage stack overhead using the kernel-bypass libraries [17, 62, 266, 271, 276]. Typically the kernel-bypass storage stack involves moving the some of the components in the storage stack such as the filesystem, NVMe driver to the userspace so the application does not have pay the kernel-crossing overhead. For instance, SPDK [62] is a popular kernel-bypass storage stack from Intel and it implements the NVMe driver to the userspace. The applications can directly access the storage media and poll for IO completion directly from the userspace. However, such a kernel-bypass storage stack design comes with a lot of tradeoffs; applications using SPDK storage stack are required to implement their own filesystem in the userspace [10]. Further, *with kernel-bypass systems applications need to forgo the isolation and safety provided by the OS with ueer-kernel address space separation.* This is a serious problem because, now that the storage stack co-exists with application in the userspace a simple bug in the application code (*e.g.*, buffer overflow) can easily corrupt the storage stack and consequently corrupt the entire storage media.

Moreover, such kernel-bypass storage stack are still filesystem based; meaning that they are designed to support only block based IO and this makes new memory-centric storage technologies such as the byte-addressable NVM [20, 26], memory-semantic SSDs [34, 36], battery-backed DRAM [3, 21, 37] and computational SSDs [55, 59, 207] from leveraging such a storage stack for byte-addressability. Memory-centric storage offers byte-addressability, meaning that the applications can use load/store instructions to access the storage media. Essentially, memory-centric storage technologies aims to close the performance gap between memory and storage by bringing storage closer to the memory. Kernel-bypass storage stack is essential for these memory-centric storage technologies for two reasons, 1) they operate at very low latency ranging from few hundred nanoseconds to few microseconds and for such low latency devices the traditional OS storage stack would add more performance penalty by increasing the access latency. 2) applications can not exploit the byte-addressability feature of memory-centric storage, therefore, a new kernel-bypass storage stack that can offer byte-level access is necessary for memory-centric storage; meaning that there is a need to move away from filesystem abstraction provided by the OS storage stack. While designing a kernel-bypass storage stack looks obvious and promising for memory-centric storage there are several critical challenges in the aspects of programmability, performance, scalability, safety, and reliability when bypassing the traditional OS storage stack.

1.3 Challenges in Designing a Kernel-Bypass Storage Stack

Programmability. So far the filesystem based programming has been a successful programming model for storage systems, moving away from filesystem is a critical challenge. The filesystem layer in the OS storage stack handles crash consistency, concurrency, caching, and recovery which makes the application development easier. So one of the key challenge when designing a kernel-bypass storage stack for memory-centric storage is to identify the

right programming abstraction. The next challenge is to identify how the crash consistency and caching techniques need to evolve to keep up with the low latency memory-centric storage media. This is critical because the traditional journalling based crash consistency designed for block based storage are known to have high performance overhead and high write amplification [180, 198].

Scalability and Performance. As discussed in §1.1, the primitive lock-based concurrency is not scalable for multi-core CPUs. But the upside of such lock-based concurrency scheme is that they are simple and hence it is easy for application to reason about the system correctness. While concurrent programming is in itself challenging, designing a scalable concurrency mechanism without trading off the programmability is a much challenging endeavor. But a scalable concurrency mechanism is central in designing an efficient storage stack and more importantly it should not come at the cost of programmability and correctness.

Safety and Reliability. The OS storage stack provides isolation and safety using the userspace and kernelspace abstraction (*e.g.*, ring privileges in Linux OS). Since the kernel-bypass storage stack will co-exist with the application in the userspace a trivial bug in the application code can compromise the safety of the entire storage stack by corrupting the data stored in the underlying physical storage media. The memory safety bugs are common in applications [22, 38, 39] and data corruption due to such bugs have catastrophic effects which can compromise the safety of the entire system [165, 200, 233, 242, 273]. Although this problem is serious none of the existing kernel-bypass storage stack including the widely used SPDK [62] provide any sort of isolation. So a kernel-bypass storage stack should strive to provide some form of isolation and safety as provided by the traditional OS storage stack. Further, memory-centric storage media can experience device wear-out as it gets older similar to the traditional block based storage media [194, 205, 227]. Such a device wear-out can cause data corruption and comprise the reliability of the storage stack. Therefore it is necessary for

a kernel-bypass storage stack to be fault tolerant against device failures.

1.4 Thesis Contributions

This thesis proposes a kernel-bypass storage stack designed particularly for memory-centric storage by addressing the programmability, multi-core scalability, and safety challenges. *This thesis uses the recently commercialized byte-addressable NVM [26] and computational SSDs (CSD) [59] as the representative ultra-fast memory-centric storage technologies for evaluating and analyzing the implications of the proposed systems.* Overall this thesis makes the following contributions:

Thesis Statement

The OS storage stack is designed for slow block storage devices and is not equipped to efficiently support the new memory-centric storage technologies. Memory-centric storage exposes the programmability, performance, and scalability limitations in the OS storage stack. This hinders the applications from exploring the new memory-centric storage technologies to its true potential. Hence, this thesis proposes a series of kernel-bypass designs to overcome the constraints and limitations in the OS storage stack. This thesis introduces novel kernel-bypass programming frameworks to help developers design a high-performant, multi-core scalable, memory safe, and fault tolerant applications using memory-centric storage technologies.

1.4.1 Transactional Programming Framework for Scalable NVM Programming

Chapter 3 describes TimeStone, a scalable persistent transactional memory (PTM) programming framework for byte-addressable NVM. TimeStone advocates the use of transactional programming abstraction for NVM because of its capability to hide the complexities of concurrent and crash consistency programming. All developers need to do is to just write a

serial (single-threaded) code and enclose the code within the TIMESTONE's *ts_ begin() and ts_ end()* wrapper; then under the hood TIMESTONE makes the application concurrent, crash consistent, and recoverable. Internally TIMESTONE uses multi-version concurrency control to support concurrent non-blocking reads and concurrent disjoint writes which makes it scale for 100's of CPU cores even for a write-heavy workloads. For crash consistency, TIMESTONE implements a hybrid logging technique called the TOC logging, a combination of operational logging and REDO logging. TOC logging is optimized for a lower write amplification, it offers a lightweight crash consistency and a consistent loss-less recovery.

1.4.2 Making PTM Memory Safe and Fault Tolerant

Chapter 4 describes TENET, which essentially builds on the TIMESTONE PTM to enable memory safe and fault tolerant transactions. TENET uses simple techniques like canary bits, pointer tagging to guarantee both spatial and temporal memory safety. It leverages the RCU style grace-period and operational logging to design a asynchronous data replication while guaranteeing no-loss recovery. Overall, TENET breaks the fundamental belief that memory safety comes at a high performance overhead by making TIMESTONE memory safe and fault tolerant with ∼20% average performance overhead. The key insights in TENET is to show how can the concurrency guarantees of the transactional programming model be leveraged to provide a optimized, strong, and lightweight memory safety and fault-tolerance. In a nutshell, with TENET as a userspace storage stack, developers can build a high-performant, multi-core scalable, memory safe, and fault-tolerant applications albeit with a programming abstraction different from the traditional filesystem but a feature rich one best enough to exploit the ultra-fast memory-centric storage hardware.

1.4.3 Programming Framework for Porting Legacy Applications to NVM

Chapter 5 presents Tips, a programming framework for porting legacy in-memory key-value store (KVS) and database applications to memory-centric storage. Before memory-centric storage, in-memory KVS and database applications run on DRAM forgoing consistency, data persistence, and better capacity scaling in the interest of performance. The low latency memory-centric storage provides a great opportunity for such in-memory applications to achieve strong consistency, better capacity scaling, and data persistence at the memory level instead of relying on traditional block based storage. But porting an existing DRAM based in-memory KVS is reported to be cumbersome, hard and error-prone requiring a significant amount of engineering effort [45, 185] for several reasons particularly the crash consistency requirements. Tips acts a gateway to port such DRAM based in-memory KVS to use memory-centric storage by providing a generic framework to which the developers can *plug-in* the KVS of their interest. Tips automatically makes the *plugged-in* KVS persistent, crash consistent, and recoverable in the event of crash. To ensure correctness, Tips guarantees *durable linearizability* for all its conversions which is the gold-standard correctness condition. Overall, we converted 7 state-of-the-art volatile KVS index structures and the real-world Redis KVS using Tips with less than 20 lines of code changes in each of the applications. All the Tips-enabled applications showed excellent performance, multi-core scalability, and correct recovery.

1.4.4 Unified KVS Framework for Near-Storage Programming

Chapter 6 describes Retina, an unified KVS farmework for computational storage. Unlike the other memory-centric storage technologies like the byte-addressable NVM computational SSDs [59] are capable of performing compute on the data with help of near-storage accelerator (*e.g.*, FPGA) which is shipped as a part of the SSD. Unlike the TimeStone and Tips,

RETINA takes an application driven approach by focusing on solving the CPU side and storage side bottlenecks in the Deep Learning (DL) model training. The DL training is primarily bottlenecked by the stalls due to, 1) fetching data from the storage, and 2) preprocessing the data using the CPUs [88, 163, 199, 239]. RETINA focuses on addressing the two aforementioned problems by leveraging the computational SSDs.

However, the OS storage stack in its current form is not designed to enable near-storage compute and neither there exists any standard programming model for applications to leverage computational SSDs. Hence RETINA tackles the programmability challenges by proposing a near-storage compute capable kernel-bypass storage stack for computational SSDs. To this end, RETINA proposes a KVS framework with a computational pipeline implemented on the near-storage accelerator. We integrated the RETINA to the Tensorflow framework and conducted ResNet50 DL model training by offloading the entire image preprocessing steps (JPEG decoding, crop, flip) to the FPGA inside the computational SSD. Offloading preprocessing steps proved to be beneficial as it decreased the training latency by 36%, reduced the CPU utilization and GPU idle time by 60% and 43% respectively when compared to the traditional CPU side preprocessing.

1.5 Thesis Outline

This thesis is organized into six chapters. Chapter 3 discusses the programmability and scalability challenges in the kernel-bypass storage stack and introduces TIMESTONE PTM as a potential solution to the programmability and multi-core scalability problems. Chapter 4 discusses the importance of memory safety and fault tolerance support in the kernel-bypass storage stack and introduces TENET, an extension of TIMESTONE PTM to enable memory safe and fault tolerant transactions. Chapter 5 introduces TIPS framework and discusses the challenges and the importance of porting legacy volatile in-memory applications to

memory-centric storage. Chapter 6 discusses the CPU side and storage side bottlenecks in the DL model training and discusses how those bottlenecks can be addressed using computational SSDs by introducing Retina. Chapter 8 concludes this thesis by discussing how the techniques proposed in thesis can be a strong foundation to design kernel-bypass storage stack for future memory-centric technologies.

Chapter 2

Background

Memory-centric storage technologies [20, 26, 34] provide the best of both memory and storage; they are byte-addressable like the main memory and they are persistent like the traditional block storage devices. With byte-addressability, applications can use simple load/store instructions to program memory-centric storage and this enables to develop fast, crash consistent data structures. Throughout this thesis, we consider the recently commercialized byte-addressable Non-volatile memory (NVM) [26] and computational SSDs (CSD) [59] as the representative ultra-fast memory-centric storage technologies for evaluating and analyzing the implications of the proposed systems. However, the problems identified and the solutions proposed in this thesis are pertinent for the other future ultra-fast memory-centric storage technologies [2, 3, 20, 21, 34, 36, 37, 40, 111, 116, 225]. The next two sections (§2.1 and §2.2) introduces the architecture of the NVM and the CSD.

2.1 Byte-addressable NVM

NVM provides byte-addressability like the DRAM and it is persistent like the SSD. NVM operates at nanosecond latency, it has a worst case read and write latency of 80 ns and 230 ns [136] which is ∼100× faster than the high-performance microsecond scale NVMe

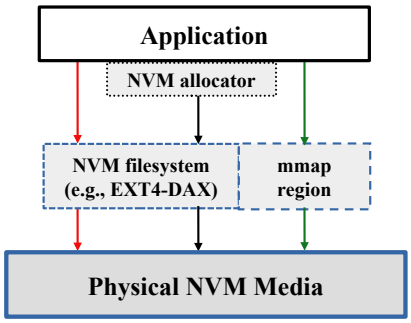

Figure 2.1: NVM software stack. NVM is mapped to the userspace and directly accessed via load/store instructions (green arrow).

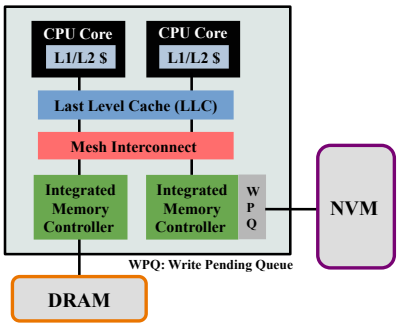

Figure 2.2: NVM hardware architecture. NVM sits on the CPU's memory bus alongside the DRAM.

SSDs [41, 42, 70, 71] NVM has high capacity scaling than DRAM, the current generation of NVM can scale up to 6TB on a single machine. It also offers a higher write endurance than all the disk based storage medias. Essentially, in terms of latency NVM sits in between the DRAM and storage, and thus closing the large performance gap that exists between the memory and block based storage. NVM provides the applications particularly the data center scale and compute-centric ones to achieve (1) high performance at a reasonable cost, (2) low durability cost, and (3) to considerably reduce the data movement costs as the persistent data is located closer to the CPU.

Figure 2.1 illustrates the NVM software stack. Byte-addressability in the NVM is enabled by mapping (*mmap*) the filesystem to the applications address space (green arrows). Then the applications use the load/store instructions to access the mapped region. The mapped NVM region is typically managed using a NVM memory allocator [98]. NVM can also be accessed via the OS storage stack similar to the block based storage like the SSDs (red arrows). However, in this model the NVM is not byte-addressable *i.e.*, the applications can not access NVM using load/store instructions.

Figure 2.2 illustrates the NVM architecture with Intel's DC persistent memory (DCPMM) [26]; NVM like the DRAM sits right on the CPU's memory bus alongside the DRAM and the

CPU cache hierarchy. The NVM's integrated memory controller is equipped with a write pending queue (WPQ) and it is considered as part of the non-volatile domain *i.e.*, when the stores reach the NVM's memory controller it is considered persistent. The stores are batched in the WPQ and is written to the physical storage media. However, the L1, L2, and the Last Level (LL) cache are volatile *i.e.*, stores in the cache hierarchy are not persistent and such stores will be lost in the event of a power failure or a program crash. So applications need to flush the stores in the L1 cache using cacheline flush instructions (*e.g.*, `clwb`). To guarantee store ordering, that is the order in which stores reach NVM is can be controlled by using explicit fence instructions (*e.g.*, `sfence`). In the Intel systems that support Extended ADR (eADR) [18], the L1, L2, and the LL cache are non-volatile; so the stores in the cache hierarchy are persistent and the applications do not need explicit cacheline flushing to make the stores (<= 8 bytes) persistent. For larger size stores (>= 8 bytes) applications may still need to use `clwb` and `sfence` to guarantee consistency depending on the situations. We discuss more about this in the next two sections. Overall, although the NVM technology is promising, the store ordering and crash consistency requirements opens up a lot of intricate challenges.

2.1.1 Store Ordering in NVM

Stores to the NVM are guaranteed to be persisted only when the stores reach the NVM's memory controller *i.e.*, stores have to be explicitly flushed from the CPU cache to reach the NVM media. This creates a huge number of correctness and ordering issues. For instance, consider the example of adding a new link list node, it is a two step process. (1) allocate a new node and populate it and (2) modify the *next* pointer of the previous node to make the new node visible. Normally in the DRAM world this is straightforward and intuitive. But for NVM there is a caveat, *the order of persistence should be same as the execution store order* , *i.e..*, in case of NVM the new node must be first persisted before updating the next pointer

of the previous node. If this order is not maintained then it can cause memory corruption which in turn will make the application irrecoverable. For instance, say if the next pointer is modified and persisted before the new node (*i.e.*, new node is still in the CPU cache but has not reached the NVM media yet while the next pointer modification has reached the NVM media) and the system crashes. Upon recovery, the next pointer would point to garbage address as the new node has not been persisted before the crash.

This is a complicated problem and it becomes more serious when the application logic is complex such as having to update multiple pointers at the same instance like in case of a B+tree split operation. This is further complicated by the random cacheline evictions. In order to guarantee the correct persistence ordering developers are required to use explicit cacheline flush and fence instructions (*e.g.*, `clwb` [12] and `sfence` [57]). Again these instructions should be inserted in the right part of the application logic to ensure correctness and even a small/trivial mistake can render the application irrecoverable. Moreover such programming bugs are hard to spot and it requires considerable amount of experience and in-depth knowledge on the NVM programming.

The systems with eADR support (non-volatile CPU cache) neutralizes the store ordering problem to some extent; particularly, for the stores to NVM that follow 8-byte atomicity guaranteed by the hardware [18]. For instance, in our linked list example if a new node is added in lock-free manner using the atomic store instructions then the order in which the stores reach NVM would not compromise the correctness. However, for the stores that are greater than 8 bytes, applications still need to handle the store ordering. For instance, consider the leaf node split operation in a B+tree. The split operation typically happens in three steps, 1) allocate a new leaf node and copy half of the key range from the old leaf node to the new one, 2) modify the old leaf node and parent leaf node to point to the new leaf node, and 3) remove the redundant key ranges in the old leaf node. If a power failure happens

after step 3 then there is no correctness violation. But say that a program crashes in between step 2 and step 3; in that case, after recovery, the new leaf node and old leaf node will have overlapping key ranges which can make the B+tree inconsistent. This problem becomes more serious when in the event of a cascading split operation where the updating the parent node to point to the new leaf node results in the parent node split. Such cascading split can render the entire B+tree inconsistent and useless. So in such scenarios the applications need to control the order in which the stores reach NVM or implement some form of logging to recover the inconsistent B+tree during the recovery. Overall, for NVM programs that performs atomic lock-free updates eADR systems can reduce the complexities due to store ordering, otherwise the applications are required to maintain the correct store order for consistency. However, it is a challenging task if not impossible to make stores greater than 8 bytes atomic and there are several research works [174, 250] in this area.

Additionally there is also the problem of persistent memory leaks. While memory leaks are serious security threat even for DRAM applications it is more serious in case of NVM. Because a memory leak can stay forever in the NVM if it goes undetected as NVM is non-volatile; *i.e.*, a simple system restart can not fix the memory leaks in the NVM like in the DRAM. So upon recovery it is critical that NVM based systems should be capable of detecting and fixing the memory leaks.

2.1.2 Crash Consistency in NVM

Unlike the DRAM, all writes to NVM should be crash consistent. The writes to NVM must not cause inconsistency if the program crashes say due to a power failure. Crash consistency is required to ensure that all the NVM data can be correctly recovered to either the state before the write happened or the state after. Recovering to an intermediate state leads to memory corruption which in turn will lead to application crash and data loss. Although, the

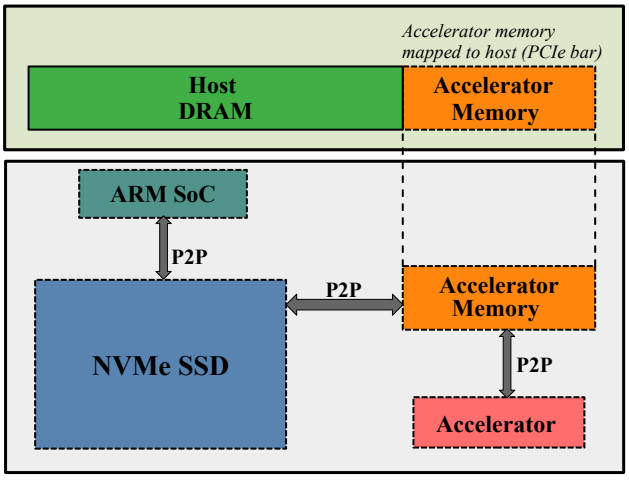

Figure 2.3: Computational SSD Architecture.

concept of crash consistency is not new to the OS and storage developers, designing crash consistency mechanism for NVM has its own challenges. The concept of crash consistency and techniques to achieve it has been well studied in the traditional block based storage software (*e.g.*, filesystem). Such systems relies mostly on *logging* to guarantee crash consistent writes. Particularly, the *UNDO* and *REDO* logging are the two widely used logging techniques to guarantee failure-atomic updates. While these logging techniques can be used for NVM it comes with a different set of challenges. It is because NVM has almost 100× lower read/write latency than the block storage media. So, using the logging techniques designed for the block based storage will cause high performance overhead and can potentially defeat the latency benefits of using NVM. In a nutshell, guaranteeing crash consistency is inevitable for NVM but at the same time it is critical to design a crash consistency mechanism with low performance and space overhead.

2.2 Computational Storage

Computational SSDs (CSD) are capable of performing compute near the storage independent of the CPU. Figure 2.3 illustrates the CSD architecture; the striking difference from the

traditional SSD is the presence of an accelerator (*e.g.*, FPGA) and the ARM SoC inside the storage media. The accelerator can be leveraged to perform compute on the data stored in the NVMe SSD without having to move the data back and forth to the CPU memory. The ARM SoC inside the computational storage acts as a control plane; it moves the required data from the SSD to the accelerator's memory (DRAM) for the accelerator to perform the compute. The ARM SoC runs an embedded OS [43] which will trigger reads and writes to the SSD. Further, the accelerator memory is typically mapped to the PCIe bar address and it becomes the part of the CPU's memory hierarchy. This mapped memory is used as a shared memory region between the CSD and the host CPU for communication and data movement.

While CSD shows a lot of potential to reduce the data movement, it is still a nascent technology. There is no standard programming model for CSD until recently proposed by SNIA [13]. The lack of a standard programming interface for CSD will be primary impediment for the applications to explore the CSD. Because, offloading compute to storage requires deviating from the traditional simple filesystem based programming and currently there are no specific OS support for the CSD. So solving the programmability challenges by designing a high-level programming interface will be an important first step in exploring the CSD.

2.3 Kernel-Bypass Storage Stack for Memory-Centric Storage

Sections §2.1 and §2.2 explains how the memory-centric storage particularly the byte-addressable NVM and CSD are different from traditional block based storage. This begs the question, *how does the conventional OS storage stack fit in for such memory-centric storage?* Apart from the performance and scalability bottlenecks in the OS storage stack (§1.1), the OS storage stack can not be used to fully exploit these new ultra-fast memory-centric storage at least not in its current form. Of course, both NVM and CSD can be accessed using a traditional filesystem programming model but in doing so the applications have to forgo the

NVM's byte-addressability feature and the CSD's near-storage compute feature.

Kernel-bypass storage stack is a potential solution to overcome the performance, scalability, and the feature limitations of the OS storage stack. The idea of kernel-bypass existed even for the block storage [17, 62, 266, 271, 276], but the focus was to reduce the performance overhead of the OS storage stack by moving a part of storage stack (*e.g.*, filesystem, NVMe driver) to the userspace. However, designing a kernel-bypass storage stack for memory-centric storage is quite different from the ones designed for the block storage. For instance, the kernel-bypass storage stack must be capable of enabling byte-addressable access. This means that a memory-centric kernel-bypass storage stack must deviate from the traditional filesystem based programming model. As discussed in the previous chapter (§1.3) designing a kernel-bypass storage stack for memory-centric storage opens up a lot of new challenges in the aspects of programmability, multi-core scalability, performance, safety, and isolation. Below we explain the prior research efforts geared towards solving these problems, their shortcomings, and the key insights proposed in this thesis to address these challenges.

2.3.1 Programming Model for Memory-Centric Storage

There have been lot of prior research works focusing on developing new filesystems for NVM [102, 144, 259, 260, 277], however as stated earlier with filsystem programming model the byte-addressability of NVM can not be exploited. However, NVM filesystems have good use cases particularly in making legacy applications to work on NVM. Some of the NVM filesystems are implemented in the userspace [102, 144] to reduce the kernel storage stack overhead by delegating all file operations to the userspace and using a trusted kernel component for updating the filesystem metadata.

Many research works also take application-specific approach by designing a NVM index structures [77, 87, 127, 167, 176, 179, 183, 204, 206, 213, 247, 263, 278] and use those index

structures as a core building block to design a crash consistent NVM based memory-centric key-value stores [46, 258]. These approaches exploit the byte-addressability of NVM to build crash consistent data structures by leveraging the data structure or application specific logic. Such approaches are not generic enough and lacks flexibility in terms of programmability. For instance, the crash consistency techniques in these approaches are highly data structure specific and it can not be applied to other systems or in some cases even for a different type of data structures. Overall, application specific data structure approach is performance-efficient, and easy to scale yet it lacks the generality and the flexibility offered by the filesystem programming model.

In the chapter §3 of this thesis, we propose to design a transactional programming model by designing a persistent transactional memory (PTM). PTMs provide ACID guarantees (**A**tomicity, **C**onsistency, **I**solation, **D**urability) for all the updates that occurs within the transaction boundary. Unlike the data structure approach, PTMs are generic, consistent, and composable. Unlike the NVM filesystem, PTMs support byte-addressable programming and does not rely on the OS storage stack for storage operations which makes it performance efficient. The idea of using PTMs for NVM has been explored prior to this thesis [131, 177, 246] but such PTMs simply repurpose the software transactional memory (STM) [103, 107, 122] designed for DRAM by adding a naive layer of logging for crash consistency. STMs are notorious for their poor multi-core scalability and high performance overhead; this is one of the reasons why the idea of STMs failed for DRAM [84]. Adding a naive UNDO/REDO logging based crash consistency to these already slow and poorly scalable STMs further increases the performance overhead and the write amplification. Overall, the promising PTM programming model is marred by the performance and scalability challenges. Section §2.3.2 discusses the reasons for poor scalability and high performance overhead in PTM systems.

Similar to the NVM, there is a lack of generic programming framework for CSD. Most prior

works focus on adhoc compute offloads to the CSDs such as SQL processing [101, 138, 254], graph analytics [141, 166, 186], filesystem [80, 181, 265]. Only a very few works focusses on addressing the programmability of CSDs [224, 226, 264]. Among them, MetalFS [226] and Insider [224] propose CSD programming framework using UNIX pipes and virtual filesystem respectively. Both the works focus on offloading compute to the CSD but suffers from high performance overhead. Chapter §6 of this thesis introduces Retina a compute capable KVS framework for CSDs and discusses how Retina addresses the programmability and performance challenges in the CSD programming.

2.3.2 Performance and Multi-Core Scalability

As stated earlier, prior PTM works repurposed the STMs designed for the DRAM [103, 107] by simply adding additional crash consistency layer. The DRAM based STMs scales poorly due to the globally shared lock table and the high performance overhead is due to the write amplification in the word-based STMs [103, 107]. Consequently, the PTMs such as the DudeTM [177], and Mnemosyne [246] which are essentially built on top of the tinySTM [107] inherits the same scalability and performance bottlenecks. None of the existing PTMs scales beyond 8 CPU cores even for a read-heavy workload (98% reads and 2% updates).

Further, the logging operations in the PTMs such as the DudeTM (REDO logging), and the PMDK's libpmemobj (UNDO logging) [130] causes a very high write amplification. The write amplification comes from additional writes performed to the logs to ensure crash consistency for every write operation to the NVM. For instance, PMDK's libpmemobj and DudeTM incurs up to 70× and 4× write amplification. This means that for every byte of user data written to the NVM, libpmemobj and DudeTM writes up to 70 bytes and 4 bytes of data to the NVM. Such a high write amplification easily becomes the performance and scalability bottleneck in these PTMs. With regards to multi-core scalability, In chapter §3, we introduce

TIMESTONE PTM and discuss how it overcomes the challenges to scales up to 100's of CPU cores and guarantee a < 1 write amplification at all times. With TIMESTONE, this thesis breaks the fundamental notion that the PTMs are slow and poorly scalable by showing how the transactional programming model can be scaled to 100's of CPU cores.

On the other hand, it is not just the PTMs that suffer from high crash consistency overhead, even some of the data structures designed specifically for the NVM incur high performance overhead due to crash consistency operations [77, 174, 176, 250]. So to reduce performance overhead the NVM data structures forgo the critical features (*e.g.*, not supporting variable length keys) [87, 127, 213, 263] and sometimes even resort to supporting weaker consistency guarantee such as buffered durable linearizability (BDL) [118, 169, 257]. Supporting a weaker consistency such as BDL makes the applications using these data structures difficult to guarantee correctness and also it comes at the cost of loosing data in the event of a crash or a power failure. However, one of the main benefits of using NVM in an application is to guarantee a consistent loss-less recovery and supporting a weaker consistency totally defeats the purpose. In chapter §5 of this thesis, we introduce TIPS, a programming framework specifically designed for porting the legacy DRAM based volatile index structures and also to design new NVM index structures without trading off the consistency for performance. With TIPS, this thesis defeats the fundamental notion that the strong consistency comes at a performance tradeoff by making TIPS perform on par with the systems that support a weaker consistency guarantee.

2.3.3 Memory Safety and Fault Tolerance

It is critical for any NVM storage stack whether it is a filesystem or a PTM or a index structure to guarantee memory safety and fault tolerance. Because, these systems coexist with the application in the userspace and any trivial application bug (*e.g.*, buffer overflow,

dangling pointers) can corrupt the storage stack. Corrupting a storage stack may result in the loss of critical metadata and in many cases it can render the entire application irrecoverable, this defeats the purpose behind using NVM. Further, the Random Bit Error Rate (RBER) of NVM is reported to be quite high on par with the traditional NAND flash [245, 270]. Hence unlike the DRAM, NVM can wear out in time and such a cell wear out can silently corrupt the NVM data without the knowledge of the application or the NVM storage stack. Such data corruptions due to NVM hardware failure is called as media errors. To reduce the impact of such media errors the NVMDIMMs are usually equipped with hardware ECC which is capable of up to 2-bit errors [27]. The errors that can not be corrected by the hardware ECC are reported to the application (by the OS) as Uncorrectable Media Error (UME). It is then the responsibility of the application to either correct such errors by restoring the NVM data from the backup (if exists) or to endure data loss by ignoring the error report. Further, the OS reports the UME in the page granularity *i.e.*, even if only a small portion of data is corrupted the OS offlines the entire NVM page where the corruption occurred. So the applications are in the danger of loosing an entire page of data in the event of an UME. Also, UMEs are random *i.e.*, it can happen at any random offset so loss of critical metadata means the entire application is at the risk of being rendered irrecoverable. Consequently, it is critical for a NVM storage stack to be fault tolerant against such UMEs.

Unfortunately, most of the existing NVM filesystems, PTMs, and index strictures do not provide any memory safety or fault tolerance guarantees. Only a few PTMs [82, 272] and filesystems [260] support some form of memory safety and fault tolerance. Pangolin [272] and SafePM [82] extend the libpmemobj transaction to support a partial memory safety. Meaning that both the techniques have lot of safety vulnerabilities in their system that can be exploited with a trivial application bug. Further, these systems introduce high performance overhead and high write amplification in the quest to achieve memory safety. For

instance, the libpmemobj transaction slows down by up to 2× when Pangolin's and SafePM's techniques are applied to make it memory safe. Obviously, libpmemobj's already high write amplification is also further increased by these techniques. Eventually with such systems, memory safety (albeit only partial) comes at very high performance tradeoff. Chapter §4 of this thesis introduces TENET and discusses how TENET makes TIMESTONE memory safe and fault tolerant with just ∼20% performance overhead. Chapter §4 also discusses how TENET overcomes the performance challenges to guarantee full memory safety and fault tolerance. Before TENET, NVM systems (even many DRAM systems) were either high-performant and no memory safety or poor performance and full memory safety. This thesis breaks this barrier by designing TENET which guarantees full memory safety and fault tolerance at modest performance overhead.

2.4 Chapter Summary

This chapter introduces memory-centric storage technologies using NVM and CSD as representative examples. This chapter also discusses how the NVM and CSD are different from traditional storage devices and the challenges in the programming NVM and CSD. Section §2.3 introduces the need for kernel-bypass storage stack to program NVM and CSD as the OS storage stack can not exploit the NVM's byte-addressability and CSD's near-storage compute capabilities. Further, Sections §2.3.1-§2.3.3 discuss the challenges in designing a kernel-bypass storage stack for memory-centric storage. In a nutshell, a kernel-bypass stack should support better programmability, better performance and multi-core scalability, and provide memory safety and fault tolerance for the applications to efficiently use the memory-centric storage technologies. Chapters §3-§6 delves deep into each of these challenges and introduces new system designs to overcome the challenges. Next chapter introduces TIMESTONE PTM and discusses the programmability and multi-core scalability challenges in the NVM.

Chapter 3

Designing a Scalable PTM for NVM

New emerging non-volatile main memory (NVM) technologies, such as Intel Optane [74, 196], provide persistence along with traditional main memory characteristics [244, 274], such as byte-addressability and low access latency. In addition, the NVM offers data durability and larger in-memory capacity at a significantly lower `$/GB` compared to traditional DRAMs [90, 182, 218, 238, 267]. Although NVMs incur higher read-write latency compared to traditional DRAMs [92, 147], they enable software to have a larger capacity and almost attain free durability of data.

While NVM technology is promising, it poses system developers with several new challenges such as guaranteeing crash consistency with a minimal write amplification, scalability, and high performance at high core counts. Even for byte-addressable NVMs, guaranteeing crash consistency requires high latency logging operations in the critical path, complicated by the modern out-of-order processors that can reorder cacheline evictions. As a consequence, achieving crash consistency without impacting the manycore scalability and performance has become an onerous task.

Nevertheless, manycore scalability is becoming an inevitable design principle when designing

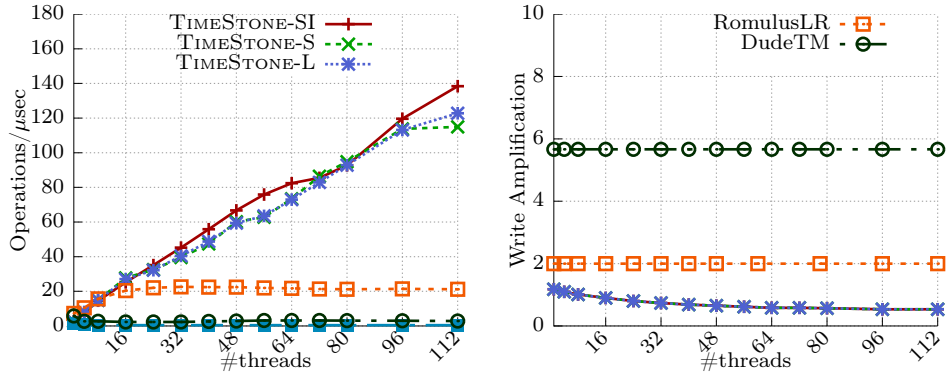

Figure 3.1: Performance comparison of PTM systems for concurrent hash tables with 2% update. Except TIMESTONE, prior systems suffer from poor scalability and high write amplification.

NVM software as NVMs are expected soon to be a part of data center manycore servers [81]. For example, the first public Cloud service of DCPMM used by SAP HANA, an in-memory database system, which requires manycore parallelism [81]. So a competent NVM library should provide better performance and scalability, have a minimal write amplification, be memory efficient, and have broad-ranging applicability.

Unfortunately, none of the prior work exhibit all the above capabilities. For instance, prior concurrent durable data structure (CDDS) libraries [77, 87, 127, 146, 167, 204, 206, 212, 213, 248, 263, 275, 278] leverage application's data structure knowledge to achieve better scalability but do not guarantee atomicity of multiple operations, such as atomically adding two nodes to a list (durable composability). Moreover, current CDDS libraries do not guarantee consistency for data (full-data consistency), rather delegate it to the application developers. Failing to guarantee durable composability and full-data consistency delimits the usage of such libraries.

Contrary to CDDS libraries, existing PTM approaches [93, 115, 131, 177, 192, 246] support durable composability and provides full-data consistency. However, our analysis shows that none of the PTM systems scales beyond 16 cores (see Figure 3.1). For example, DudeTM [177] and Mnemosyene [246] scale poorly because of the underlying STM, which is known for its poor scalability [84]. They extend STM with an extra durability layer, which incurs a

PTM Systems	Isolation Level	Parallelism RR	RW	WW	Durability How	When	Additional Memory DRAM	NVM	NVM Write Amplification
PMDK [131]†+	LN	●	▲	×	UNDO	⇒]	-	log	≥ 2
Mnemosyne [246]‡	LN	●	▲	▲	REDO	⇒]	-	log	4-7
KaminoTX [192]+	LN	●	▲	▲	Backup	⇒]	-	NVMM	≥ 2
DudeTM [177]*	LN	●	▲	▲	REDO	⇛	log + NVMM	log	4-6
Romulus [93]*	LN	●	×	×	Backup	⇒]	-	NVMM	2
Pisces [115]+	SI	●	●	▲	REDO	⇒]	-	log	≥ 2
TENET	SI/SR/LN	●	●▲	▲	TOC	⇒]	TLog	OLog + CLog	≤ 1

NOTE: SI: snapshot isolation SR: serializability LN: linearizability

×: not supported ▲: partially supported ●: fully supported ⇒]: immediate durability ⇛ : eventual durability

Table 3.1: High-level comparison of PTM systems for NVM. TIMESTONE is a PTM based on MVCC (Multi-Version Concurrency Control), which makes supporting multiple isolation levels (*i.e.*, mixed isolation levels) possible. Our novel TOC logging absorbs NVM writes through three layers of logging so TIMESTONE can provide extremely low write amplification (below 1) on NVM while providing immediate durability resulting in high performance and scalability. † While PMDK does not provide isolation, we assume that a PMDK transaction uses a readers-writer lock. We get write amplification by measuring by ourselves for ∗ or by analyzing the design of + or by referring the measured value for ‡ in Romulus [93]. We define write amplification as the ratio of the actual NVM writes by application requests while Romulus counts only the *additional* data written in NVM.

high write amplification (∼4-7×), as shown in Figure 3.1 and Table 3.1 in the course of guaranteeing crash consistency. On the other hand, Romulus [93] and KaminoTX [192] minimize write amplification by maintaining a full backup of the NVM, which derails the cost effectiveness of NVM. Moreover, existing PTM systems support limited write parallelism (refer to Table 3.1) impacting their scalability, or leaving it entirely to the application developers to use locks [131]. More recently, Pisces [115] attempts to provide scalability by providing snapshot isolation. Unfortunately, only providing snapshot isolation delimits the use for applications requiring stronger isolation model. Importantly, the dual version concurrency control and the synchronous write during log reclamation in Pisces are bound to affect scalability [152], also increasing write amplification like other PTM approaches.

To address all these problems, we propose a new PTM system, named TIMESTONE, which achieves *1) scalability across multiple cores*, guarantees *2) crash consistency with a significantly lower write amplification (< 1)* and also maintains *3) minimal additional memory footprint.* At its core, TIMESTONE adopts MVCC to achieve high concurrency and scalability but

introduces several new principles to MVCC for scalable persistency in NVM.

We believe that MVCC is a better design choice for PTM frameworks because of its inherent capabilities to support full-data consistency and the ability to run concurrent transactions with different isolation guarantees. To tackle the write amplification problem, we propose a novel multi-layered hybrid DRAM-NVM logging scheme called the *TOC logging* with the ability to absorb the write-traffic to NVM and significantly reduce write amplification. Further, to overcome the garbage collection and log reclamation overheads of MVCC [256], which impacts write throughput, we equip the TOC logging with a *scalable and concurrent log reclamation scheme.* TIMESTONE makes the following contributions:

- We introduce TIMESTONE, which is the *first highly scalable MVCC-based PTM system designed for NVM.*
- We propose a *novel multi-layered hybrid DRAM-NVM logging scheme, named TOC logging* to significantly reduce the write traffic and write amplification in the NVM.
- We propose a *scalable and concurrent log reclamation scheme* to avoid log reclamation becoming a bottleneck in our MVCC-based design.
- We design TIMESTONE to concurrently support three different isolation levels (*i.e.*, linearizability, serializability, and snapshot isolation) on the same data set. To the best of knowledge, TIMESTONE is the first PTM framework to support *mixed isolation levels.*
- We provide a *familiar programming model* hiding the complexities of MVCC, concurrency, and durability from the user with C++ API.
- We evaluate TIMESTONE with key data structures and real-world workloads and our results show that TIMESTONE outperforms state-of-the-art PTM systems with significantly higher performance and lower write amplification.

3.1 Overview of TimeStone

We first elucidate our design goals and how we incorporate them in TimeStone, and then describe the key features of Tenet with an illustrative example (see Figure 5.1).

3.1.1 Design Goals

Write-Aware System Design. Given the higher write latency, limited endurance, and high energy consumption of NVM writes [92, 126, 136, 145, 161, 217, 218, 235], it is essential that NVM applications should be write-aware. Unlike previous PTM systems [93, 131, 177, 192, 246] that suffer from high write amplification, we aim to significantly reduce amplification by making TimeStone write-aware. We adopt a hybrid multi-layered logging design (TOC logging) to absorb and coalesce a large chunk of redundant NVM writes.

Full-Data Consistency Guarantee. To support crash consistency, it is critical to guarantee consistency for applications' data stores (data) as well as applications' internal data structure (metadata), which we term as *full-data consistency*. Failing to guarantee full-data consistency affects recovery and can lead to data corruption in the NVM. For example, recent log-free data structure [97] designs guarantee only the consistency of pointers in a durable data structure (called *link consistency*) and delegates the data consistency to the application developer. Ignoring data consistency adds burden to the developer as it demands proper knowledge on correctly flushing the stores to ensure a proper recovery without any data corruption. We aim to provide full-data consistency without compromising the system performance and scalability.

Immediate Durability. For PTM systems, it is important to make updates *immediately durable* upon transaction commit. Prior PTM systems (*e.g.*, DudeTM [177]) defer durability to reduce commit latency; however adopting relaxed durability (*i.e.*, eventual durability) has

the problem of losing some updates while recovering from a failure. In TIMESTONE, we make all the successfully committed updates immediately durable which enables our system to guarantee a deterministic and loss-less recovery and all of this without compromising the performance of the live transactions unlike some of the prior techniques [85, 113, 177, 178].

Mixed Isolation Levels. The level of required isolation guarantee depends on the application semantics and there is no single isolation level that can suit all application types. For example, even though snapshot isolation can provide good performance with more parallelism, it cannot be used for developing data structures without addressing write skew anomaly [83, 115, 152, 188]. On the other hand, stronger isolation levels such as linearizability might be an overkill for OLAP-class applications, which can tolerate weaker isolation reading slightly stale data [232]. We believe that supporting multiple isolation levels for the same data set is essential considering the rapid growth of data size, and the varied requirements of applications. Now, the beauty of MVCC is that it provides a way to represent multiple isolation levels. TIMESTONE supports any number of concurrent transactions running with three different isolation levels (linearizability, serializability and serializable snapshot isolation) to operate on the same data set.

Decentralized Design for Scalability. To achieve scalability on a manycore system, we should avoid any centralized scalability bottleneck in TIMESTONE. Prior PTM techniques [177, 192] use a centralized lock table notorious for scalability bottlenecks, and similarly, transaction techniques that use copy-on-write (CoW) for performing updates [112, 192] have centralized address mapping table that hinders scalability. We designed TIMESTONE not to have such a central entity by allocating resources at the thread level (*e.g.*, per-thread logging) and using hardware timestamps for coordination among threads.

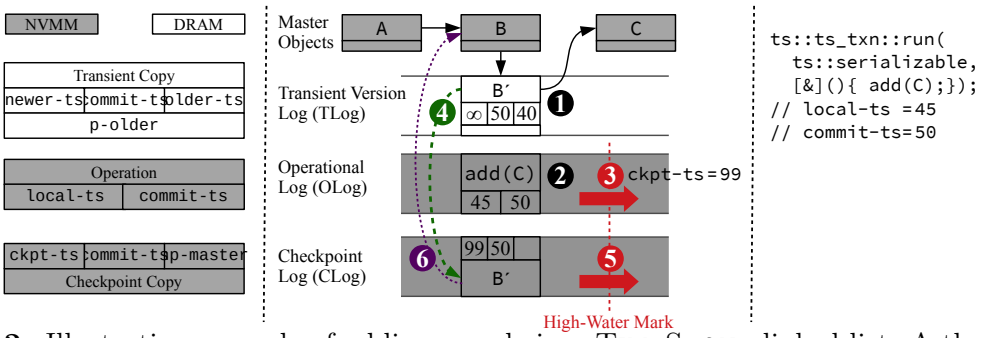

Figure 3.2: Illustrative example of adding a node in a TimeStone linked list. A thread adds a node `C` to a linked list in a TimeStone transaction (`ts_txn::run()`) with a serializable isolation level (`ts::serializable`). Consider a transaction that starts at timestamp 45 (*i.e.*, `local-ts = 45`) and commits at 50 (*i.e.*, `commit-ts`= 50). TimeStone first creates a copy of node `B` in `TLog` (`B'`) and updates its next pointer to node `C` ❶. When the transaction commits, TimeStone persists the executed operation (`add(C)`) to `OLog` making the transaction immediately durable ❷. Steps ❸ and ❺ denotes the log capacity crossing the high-water mark and this triggers the checkpointing for `TLog` reclamation ❹ and writeback for `CLog` reclamation ❻. During checkpointing, TimeStone checkpoints the latest transient copy (node `B'`) to the `CLog` so the `TLog` can be reclaimed ❹. In the `CLog` reclamation, TimeStone writes back the latest checkpoint copy (node `B'`) to the master object and the checkpoint log can be reclaimed safely ❻. The reclamation process is detailed in §3.2.9

3.1.2 Design Overview

We explain the key design features of TimeStone and how we realized our design goals with an example in Figure 5.1.

3.1.2.1 Multi-Versioning

We adopt MVCC in TimeStone to exploit its inherent benefits for key features of Tenet. Since MVCC makes out-of-place updates by composing a new version, which is a full replica of the respective original object, making the version durable guarantees full-data consistency. Importantly, because each version is discrete, it can concurrently support different isolation levels. Given these benefits, naive adoption of MVCC will incur a lot of write traffic, and affect the write endurance of NVM with frequent writes defeating our design goals of minimizing write amplification and achieving high scalability. We solve these problems by using *TOC logging* and a scalable log reclamation scheme (see Figure 5.1 and §5.3).

3.1.2.2 TOC Logging

As illustrated in Figure 5.1, in TOC logging, we use a volatile log on the DRAM, named transient version Log (`TLog`), and two non-volatile logs, namely operational log (`OLog`) and checkpoint log (`CLog`). Essentially, each one of the logging layers is key to realizing our design goals. The volatile `TLog` reduces write amplification by absorbing redundant writes to NVM and it is also key to achieving full-data consistency. The `OLog` is important to guarantee immediate durability and the `CLog` guarantees a deterministic recovery. For scalability without a central bottleneck, all three logs are per-thread logs and updates are synchronized using the hardware timestamp. The *TOC logging* is a combination of the traditional redo logging and the operational logging, but our novelty lies in the multi-layered hybrid placement of logs (in DRAM and NVM) and their usage for PTM in tandem. Next, we explain how the three logs are used in a typical TIMESTONE transaction.

Transient Version Log. As shown in the step ❶ in Figure 5.1, before modifying an object, the thread first makes the full copy of the respective master object (transient copy) in the `TLog` by locking the object (`ts_lock` in Figure 3.3). It then executes transactions on the respective copy, and upon successful commit, the transient copy object is added to the version chain. During the NVM writeback, the thread writes only the latest transient copy (❹ in Figure 5.1). Consequently, a large chunk of redundant NVM writes is absorbed in the `TLog`, which is the key to achieving a lower NVM write amplification.

Operational Log. The `OLog` is pivotal in guaranteeing immediate durability upon transaction commit. `OLog` stores only the transaction semantics (❷ in Figure 5.1), which is essentially a function pointer and its argument of a transaction, and re-executes them during recovery. Unlike the traditional undo or redo logging, `OLog` does not log the entire transaction data, and neither incurs read indirection nor requires multiple store flushes. As a result, `OLog` reduces

write amplification, improves scalability, and guarantees durability using a single persistent barrier (`clwb` and `sfence`) per transaction.

Checkpoint Log. The `CLog` is essential to maintaining the master object in a consistent state and to tolerate any potential failure during writing back the checkpoint copy (❻ in Figure 5.1). If master objects are inconsistent, re-executing `OLog` does not guarantee recovery. During recovery, the master objects are first reset to the most recent checkpointed status available in the `CLog`.

3.1.2.3 Mixed Isolation Levels

With the goal of making TimeStone a generic framework suitable for a wide range of applications, TimeStone supports multiple isolation levels–linearizability, serializability, and snapshot isolation–on the same instance of an application such that transactions with different isolation levels can run concurrently. Applications that require high performance but can tolerate write-skew or slightly stale reads should use TimeStone's snapshot isolation, while applications that needs stricter isolation levels can fall back to linearizability or serializability. For linearizability and serializability, TimeStone additionally employs a read set validation at commit time where we check if any of the objects dereferenced has been updated during the course of current transaction; If so, we simply abort and retry again. Note that the serializability and linearizability differs in the object derefernce semantics and still follow the same read set validation procedure in our design.

3.1.2.4 Scalable Garbage Collection

TimeStone maintains fixed size logs and hence the memory usage of logs are limited. If one or more logs becomes full, this could block all writes until logs are reclaimed. Hence garbage collection can directly impact write throughput. Also, a synchronous and non-scalable

garbage collection scheme can quickly become a bottleneck hampering the performance of the system [256].

For the garbage collection to be scalable, TIMESTONE must identify safe objects to reclaim without any centralized lookup or coordination. Importantly, garbage collection must be NVM-write aware so that it does not increase direct writes to NVM. Hence, TIMESTONE employs a *timestamp-based reclamation* scheme where decisions like what/when to reclaim are solely made based on the object-local timestamp without accessing shared structures. To harness concurrency in the garbage collection, TIMESTONE delegates responsibility of reclamation to each thread that holds the log itself (*i.e.*, *concurrent reclamation*). To further reduce NVM writes, we introduce *best-effort reclamation*, which reclaims objects that do not incur NVM writes. We explain the details in §3.2.9.

3.1.2.5 Programming Model

TIMESTONE follows the programming model of object-level, lock-based software transactional memory providing full ACID guarantee on NVM. TIMESTONE provides C++ API so programming in TENET is just writing a typical C++11 code using TIMESTONE base classes and APIs as shown in Figure 3.3.

User-defined persistent structures (*e.g.*, `struct node` in Figure 3.3) that inherit `ts_object` (line 1, 6) will be allocated on the NVM (line 20). To hide the complexity of NVM memory management, concurrency control, and version resolution in MVCC, TIMESTONE provides two smart pointers; `ts_master_ptr` points to a master node on NVM and `ts_copy_ptr` points to a version-resolved copy, which is part of the version chain on `TLog`. Essentially, `ts_copy_ptr` should be used in the function that accesses the `ts_object` such as `list::add()`. Type conversion between two smart pointer types involves version resolution (line 13, 24) in Figure 3.4. To modify an object, first it should be locked using `ts_lock` (line 18). Once a

```
1  struct node : public ts::ts_object { // Inherit ts_object
2   int64_t val; // data on NVMM
3   // p_next: persistent pointer to a master object
4   ts::ts_master_ptr<node> p_next;
5  };
6  class list : public ts::ts_object { // Inherit ts_object
7   // p_head: persistent pointer to a master object
8   ts::ts_master_ptr<node> p_head;
9  public:
10  bool add(int64_t val) {
11   // p_prev, p_next: version-resolved pointer to a copy object
12   ts::ts_copy_ptr<node> p_prev = p_head;
13   ts::ts_copy_ptr<node> p_next = p_prev->p_next;
14   while (p_next) {
15    if (p_next->val >= val) {
16     // Lock the object (p_prev) before update
17     // creating a transient copy of p_prev in TLog
18     ts::ts_lock(p_prev);
19     // Allocate a master object on NVMM
20     ts::ts_copy_ptr<node> p_new_node = new node;
21     p_new_node->val = p_next->val;
22     // In assigning to a persistent pointer, a persistent
23     // master object pointer of a copy will be assigned.
24     p_new_node->p_next = p_next;
25     p_prev->p_next = p_new_node;
26     return true; } // end of if
27    // In assigning to a copy pointer, a version-resolved
28    // copy pointer will be assigned after version resolution.
29    p_prev = p_next;
30    p_next = p_prev->p_next; } // end of while
31   return false; } // end of add()
32 };
33 void thread_main(int64_t v) {
34  // Run a transaction with given isolation level and function.
35  // Upon abort, ts_txn::run internally re-tries the transaction.
36  ts::ts_txn::run(ts::serializable,
37                  [&]() { p_list->add(v); p_list->add(v+1); });
38 }
39 int main(int argc, char *argv[]) {
40  // Spawn a thread for concurrent transaction execution.
41  ts::ts_thread worker(thread_main, argc);
42  worker.join();
43  return 0;
44 }
```

Figure 3.3: A linked list adding two nodes in one transaction.

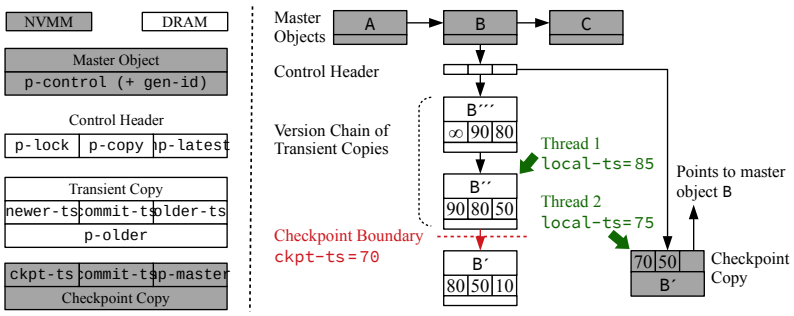

Figure 3.4: Object representation and version resolution in TIMESTONE.
An object consists of one *master object* on NVM and one or more *transient copies* in a transient version log (`TLog`) on DRAM and *checkpoint copies* in a checkpoint log (`CLog`) on NVM. Each copy has a its commit timestamp (`commit-ts`) as well as timestamps of older and newer versions (`older-ts`, `newer-ts`) to make decisions without pointer chasing. A version chain is ordered from the latest to the oldest version starting at a *control header* on DRAM. Object dereferencing finds the closest past version (`commit-ts`) to when a thread starts a transaction (`local-ts`). For example, when `local-ts` of a thread is 85, the thread will read `B"` with `commit-ts` 80 (*e.g.*, `Thread 1`). The version chain traversal should stop at the last checkpoint boundary because transient versions older than the last checkpoint timestamp (`ckpt-ts`) would already have been reclaimed. In this case, a thread should fall back to the latest version on NVM pointed to by `np-latest` on a control header (*e.g.*, `Thread 2`).

lock is acquired, `ts_copy_ptr` will be updated to pointing a new transient copy of the object (❶ in Figure 5.1). A code executed within `ts_txn::run` is a full ACID transaction and the required isolation level can be specified per transaction (line 37). Upon abort, `ts_txn::run` internally re-tries the transaction. `ts_thread` is a shallow wrapper inheriting `std::thread` that registers and deregisters a thread to TIMESTONE (line 41).

3.2 Design of TIMESTONE

We first describe the basic metadata structures of TIMESTONE's transactional object followed by versioning, logging, committing, and recovery mechanisms.

3.2.1 Object Representation

In TIMESTONE, updates to NVM are in object granularity. To make TENET NVM write-aware, we maintain frequently modified metadata (control headers) and persistent intermediate

copy objects on DRAM in addition to application data objects. We next discuss their details.

Master Object on NVM. In TimeStone, every persistent data structure is represented by a non-volatile *master object* that acts as a handle for these persistent structures. To reduce overheads of frequent access of a master object on slow NVM, TimeStone also maintains a volatile *control header* per master object on DRAM as shown in Figure 3.4. This volatile pointer (`p-control`) is validated by matching the `gen-id` cached in the *master object* and a global `gen-id`, which increments each time the non-volatile heap is loaded.

Copy Object. TimeStone maintains two different copy objects: a *transient copy* and a *checkpoint copy*. A transient copy is created on `TLog` during update operations while a checkpoint copy is created on `CLog` when the transient copy is checkpointed during `TLog` reclamation (❹ in Figure 5.1). As illustrated in Figure 3.4, a copy object caches essential timestamp information that is required to make object-local decisions during version resolution and log reclamation.

Control Header on DRAM. As shown in Figure 3.4, *control header* stores per-object run time metadata such as the version chain head (`p-copy`) and lock-status (`p-lock`). The control header is created when the master object is first updated. The decentralization of metadata enables faster metadata lookup and update performance and also significantly reduces the frequency of NVM access. The control header serves as the entry point to access copy objects and it also helps copy objects to reference their respective master.

3.2.2 Version Chain Representation

In TimeStone, the version chain is a singly linked list with the newest object at the head of the linked list. As illustrated in Figure 3.4, version chain traversal is delimited by the latest checkpoint timestamp (`ckpt-ts`) because transient copies older than the checkpoint boundary

would have already been reclaimed. For any access request beyond the checkpoint boundary, the latest checkpoint copy is dereferenced via the control header. The latest transient copy is always stationed at the head, so new threads starting a transaction may traverse until the checkpoint boundary in the worst case. Therefore, the length of a version chain does not affect performance or chain traversal cost.

3.2.3 Object Version Dereferencing

Our design provides a generic versioning support that can satisfy snapshot isolation, serializability, and linearizability. A thread does a control header lookup to get to the version chain head and traverses the chain. The thread compares its `local-ts` against the `commit-ts` of the transient copies and dereferences the closest past version, which is the most recent transient copy with `commit-ts` lesser than the `local-ts`. If a thread reaches the checkpoint boundary (`ckpt-ts`), it falls back to the control header and then the latest checkpoint copy (`np-latest`) is dereferenced. If a control header does not exist yet, the thread dereferences the master object. Note that these dereference semantics are the same for both snapshot isolation and serializability. In case of linearizability, we dereference the latest transient copy in the version chain without any further traversal. This is because linearizability requires reading the latest object and in the situation where the latest transient copy is a future version for the current transaction, we abort and retry to preserve the object dereferencing correctness.

3.2.4 Updating an Object

Before updating an object, the writer-thread attempts to lock the control header of the associated master object; if the lock could be acquired, a transient copy of the respective master object is created on `TLog` (❶ in Figure 5.1) and `p-lock` in the control header (pointing to `NULL`) is atomically modified to point this transient copy. A non-`NULL` `p-lock` means that there is an ongoing update and hence the thread aborts (see §3.2.6). Note these lock failures

(*i.e.*, `p-lock` is not `NULL`) are hidden from the user and in such cases, the writer-thread aborts and retries. The absence of a control header indicates no updates to the master object. Hence the current writer-thread is responsible for creating and locking the control header.

3.2.5 Committing a Transaction

A writer-thread maintains a private write set on `TLog` consisting of all updates made in a transaction. We first persist our `OLog` entries to make them durable and then we make all updates in the write set atomically visible; we add each of the copy objects (`p-lock`) to respective version chain head (`p-copy`) and then atomically update the `commit-ts` of the write set to the current hardware clock. Finally, we update the `commit-ts` field in the new copy objects with the write set `commit-ts`. For stricter isolation levels such as linearizability and serializability, an additional read set validation is carried out at the beginning of a commit procedure. The read set validation checks if the view of an object has changed since the arrival of this transaction. If so, then the current transaction is aborted and all updates are discarded.

3.2.6 Aborting a Transaction

A writer-thread aborts a transaction upon a `ts_lock` failure or in the event of stale-reads upon read set validation or reading a future version in linearizability. When aborting a transaction, the writer-thread unlocks all control headers by resetting `p-lock` to `NULL` and rolls back the log space. We also free the new master objects that were allocated inside the aborting transaction.

3.2.7 Timestamp Allocation

For timestamp allocations, we leverage the hardware clock (`rdtscp` in x86 architectures) to prevent the timestamp allocation from becoming a scalability bottleneck [148, 154, 175,

240, 268]. As hardware clocks can have a constant skew between processor cores which can lead to incorrect ordering, we use the ORDO primitive [148] and avoid this inconsistency. ORDO assumes there is a constant clock drift among cores and it compensates for the drift using the pre-measured ORDO boundary. ORDO is a software-based technique and only assumes invariant timestamp counter, which is already supported in x86 and many other architectures [148].

3.2.8 TOC Logging

All the logs in TIMESTONE are modeled as per-thread circular logs with new entries updated at the tail. `TLog` is created in the volatile memory while `CLog` and `OLog` are placed in the non-volatile heap. `CLog` and `OLog` are accessible from the root object of the non-volatile heap. Before terminating TIMESTONE, but after all threads safely exit the TIMESTONE transaction, we free all logs on the non-volatile heap and make the root object to point to `NULL`. Thus, a `NULL` root object upon starting TIMESTONE signifies safe termination in the previous run. We leverage this design invariant to trigger the recovery.

3.2.9 Log Reclamation

Log reclamation or garbage collection (GC) is critical as it directly impacts the write performance of the system.

Concurrent Reclamation. To prevent garbage collection being bottlenecked either by a single thread or due to synchronous waiting [188], we employ a self-reclamation scheme [152] in which, the thread that holds the log is responsible for the reclamation of its state. This design is scalable, asynchronous, thread-local, cache- and prefetcher-friendly [136]. Each thread at the transaction boundary checks if it needs to perform a log reclamation. If so, it triggers the `gp-detector` thread to broadcast the last detected grace period timestamp,

which we will define shortly. To avoid race conditions, we add a reclamation barrier to avoid any new trigger before the currently running reclamation is finished. To ensure liveness of log reclamation, the `gp-detector` thread reclaims the log of a thread which did not initiate reclamation.

What to Reclaim? We employ a RCU-style grace period detection algorithm to identify the safe reclaimable objects [152, 188, 190] Grace period is an interval in which all threads in TimeStone are outside or have exited the transaction boundary. Grace period timestamp is the time at which the a grace period detection begins. A background `gp-detector` thread continuously detects the grace period and broadcasts the grace period timestamp to all thread when log reclamation is requested.

An object is obsolete if it has a newer version, so it is no longer visible to new threads entering transactions and does not have any new references in a transaction. When all threads reading an obsolete object exit the transaction, the obsolete object becomes invisible and is safe to reclaim. Thus, as per grace period semantics, *a copy object can be safely reclaimed if one grace period has elapsed since it became obsolete*. The grace period semantics guarantee that there cannot be any thread in a transaction with `local-ts` less than two grace periods. So TimeStone always waits for at least two grace periods to elapse before reclaiming any copy object. Note that we cannot reclaim a copy object if it is the latest version of the associated master object as it is still visible to all threads. The copy object has to be checkpointed, followed by the completion of one more grace period before the copy object can be safely reclaimed.

When to Reclaim? Ideally, deferring reclamation until the log comes to capacity improves the chance for coalescing updates which will reduce the frequency of NVM writebacks. However, we can not afford the log resources to become full as it can block writers hampering system performance. Keeping this in mind, we maintain a preset *high-water mark* and

low-water mark for all three logs. When a log utilization exceeds high-water mark, the log is fully reclaimed incurring NVM writes (checkpoint reclamation in `TLog` and writeback reclamation in `CLog`). When the utilization is between low and high-water mark, the log is reclaimed in a *best-effort mode*. In the best-effort reclamation, a thread reclaims its log until the first writeback to NVM is required. The first writeback is the point at which the thread encounters the latest transient copy object of the respective master object. Stopping at the first writeback allows coalescing updates as future updates on the same object is expected. The deferred object will be reclaimed or written back in the next reclamation cycle. Below we explain how each of log reclaims in detail.

Transient Version Log Reclamation. In checkpoint reclamation passing the high-water mark, a thread first checks if this transient copy object is the latest version, then it checkpoints the copy to the `CLog` if one grace period has elapsed since this copy object is committed. After checkpointing, any future reference to this object is served from `CLog` and the checkpoint boundary (`ckpt-ts`) is set to the grace period timestamp when starting the reclamation. We then wait for one more grace period to pass and then safely reclaim the copy from `TLog`. After this, all versions with `commit-ts` less than `ckpt-ts` are deemed safe to be reclaimed. This ensures that the object does not have any references. Note that we wait for one grace period before checkpointing and one additional grace period after checkpointing making it at least two grace periods since the arrival of the copy object and that makes it safe to reclaim as per grace period semantics. Alternatively, if the entry is not the latest version then it is simply skipped so that the thread that has the latest version in its `TLog` will checkpoint that entry.

In the best-effort reclamation of `TLog`, the thread reclaims its `TLog` entries until it encounters the first writeback to NVM. It is prudent and optimal to defer writeback until log utilization goes above the high-water mark. This deferring helps reduce the frequency of writeback to NVM.

Checkpoint Log Reclamation. Writeback reclamation in `CLog` works similar to checkpoint reclamation in `TLog` except that it writebacks the checkpoint copy to its corresponding master object. A thread performs writeback only if it is the latest checkpoint copy, else it is skipped and the thread that has the latest checkpoint copy in its `CLog` will writeback during its reclamation. Again, similar to `TLog`, we wait for two grace periods to pass before safely reclaiming an object. This asynchronous waiting for at least two grace period is to ensure that there is no any existing reference to this checkpoint copy in TimeStone transactions.

The best-effort reclamation of `CLog` also follows a similar semantics of `TLog` where the thread stops reclaiming its `CLog` when it encounters the first writeback to the master. Again, this deferring the writeback is done with a goal to coalesce multiple checkpoint copies associated with the same master object and then writeback the latest copy when the thread reclaims its `CLog` in the writeback reclamation.

Operational Log Reclamation. An `OLog` entry is deemed to be reclaimable based on the last checkpoint boundary (`ckpt-ts`). So all the entries with `commit-ts` less than the `ckpt-ts` can be reclaimed anytime, independent of any grace period semantics. When `OLog` utilization goes above the high-water mark, it triggers checkpoint reclamation of `TLog` and as a result, `ckpt-ts` is updated. At this point, `OLog` can discard all the entries with the `commit-ts` lesser than the `ckpt-ts`.

3.2.10 Freeing an Object

To free a master object, we first lock the object to avoid any race condition while freeing it. If the object to be freed is in `TLog` then we cannot immediately free it as there might be one or more checkpoint copies in `CLog` that would still require a reference to the master object during its reclamation. Hence, we insert a tombstone of the master object to `CLog`. Tombstone-marked master objects will be freed when we find them upon `CLog` reclamation.

Note that the lock will not be released until the master object is freed to prevent double-free and update-after-free of the master object.

3.2.11 Recovery

TIMESTONE's recovery procedure guarantees no-loss recovery. As mentioned in §3.2.8, on safe termination of TIMESTONE, we free all logs in the non-volatile heap. When there are non-volatile logs upon start, TIMESTONE triggers recovery procedure. The recovery procedure is a two-step process, which first replays `CLog` and then replays `OLog`.

Checkpoint Log Replay. The goal of `CLog` replay is to find the latest checkpoint copy associated with each master object by sequentially examining all `CLog`s in NVM. The replay routine constructs a control header for each master object that has a corresponding copy in the checkpoint log. `np-latest` field in the control header is updated to the newest copy by comparing `commit-ts` of all the copies of the same master object. This sets up the master objects to the last consistent state before the failure and thus prepares it for `OLog` replay.

Operational Log Replay. The goal of `OLog` replay is to restore back to the latest committed state before the failure occurred. To restore the application back to the last consistent state before the failure, the transactions that happened after the last checkpoint timestamp (*i.e.*, `ckpt-ts` stored in NVM) should be re-executed from `OLog`. *The transactions in* `OLog` *should be re-executed in the original* `local-ts` *order and should be committed in the original* `commit-ts` *order to avoid any inconsistent view.* `local-ts` ordering is essential to ensure a consistent version view for replay transactions as it had during the live execution while a proper `commit-ts` ordering will bring back the application to the same old status that existed before the failure. Note that the `OLog` replay does not require any global state as the `CLog` replay will establish the required state (as in the live execution before failure) for the `OLog` entry to be correctly executed.

Recovery Time. The recovery cost in TIMESTONE is roughly constant and it is directly proportional to the utilization of `OLog` and `CLog` (number of entries in them) and the worst case being both the `CLog` and `OLog` are full. Apparently, the `OLog` entries have to be executed only for the `TLog` entries that have not been checkpointed yet. Since our log size is limited and our TOC logging scheme keeps checkpointing the updates regularly and hence we do not have to re-execute all `OLog` entries that occurred before the failure.

3.3 Implementation

We implement TIMESTONE in C and C++. The core library written in C, which comprises of around 7000 lines of code and C++ API comprised of 800 lines of code. To abort a transaction, we use `setjmp` and `longjmp` instead of C++ exception because we found throwing an exception in C++ is not scalable [228]. We use modified jemalloc allocator as our non-volatile memory allocator, `nv-jemalloc`, similar to some of the previous works [97, 177]. We could not use the PMDK allocator [131] or Makalu allocator [79] because of their poor scalability.

3.4 Evaluation

In this section, we first show that TIMESTONE achieve better scalability and performance across different data structures under various workload configurations compared to state-of-the-art PTM systems (§3.4.1). We then show the effectiveness of TIMESTONE for real-world workloads (§3.4.2). Finally, we thoroughly analyze the effectiveness of TOC logging in reducing write amplification (§3.4.3).

Evaluation Platform. We use a system with Intel Optane DC Persistent Memory (DCPMM) for our evaluation. The machine consists of two sockets with Intel Xeon Platinum 8280M processors equipped with 28 cores (56 logical cores) per socket (112 logical cores in total), 1.5 TB of NVM (12 × 128 GB), and 768 GB of DRAM (12 × 64 GB). We used `gcc`

9.1.1 with `-O3` flag to compile benchmarks and ran all experiments on Linux kernel 5.0.9.

Configuration. We preset the size of `TLog` and `OLog` to 1 MB and `CLog` to 4 MB. We also set the high-water mark to 75% for all logs. We set low-water mark to 50% and 62.5% to `TLog` and `CLog`, respectively. We also present the performance analysis for varying log sizes and analyze the sensitivity of TOC logging in §3.4.3. We ran TIMESTONE for different isolation levels. Note that TIMESTONE-SI denotes snapshot isolation whereas TIMESTONE-L and TIMESTONE-S represents linearizability and serializability versions, respectively. In order to evaluate the data structures under the snapshot isolation, we removed the write-skew by locking the adjacent nodes in addition to the nodes that are being updated as prior work [115, 152, 188] did. We compare our TIMESTONE with the state-of-art PTM systems: DudeTM, Romulus, and Intel's PMDK. Both DudeTM and Romulus provide their own memory allocator, and we ported them such that they allocate memory on the NVM. For Romulus, we handpicked RomulusLR with the best performance. PMDK's transactional library `libpmemobj` does not support isolation, so we use a standard readers-writer lock to protect a transaction from concurrent accesses.

3.4.1 Concurrent Durable Data Structures (CDDS)

We evaluate three persistent data structures–linked list, hash table, and binary search tree–for three types of workloads similar to several prior works [75, 128, 235, 278]: 1) read-mostly (2% update), 2) read-intensive (20% update), and 3) write-intensive (80% update) operations.

We present the performance and scalability in Figure 4.6 and present abort ratios in Figure 3.6 for further analysis of each PTM systems.

Linked List. We use a singly linked list with 10,000 items for our evaluation. TIMESTONE exhibits relatively a better scalability across all the workloads than the other PTM systems but the performance of DudeTM and Romulus are upto 2× better than TIMESTONE only

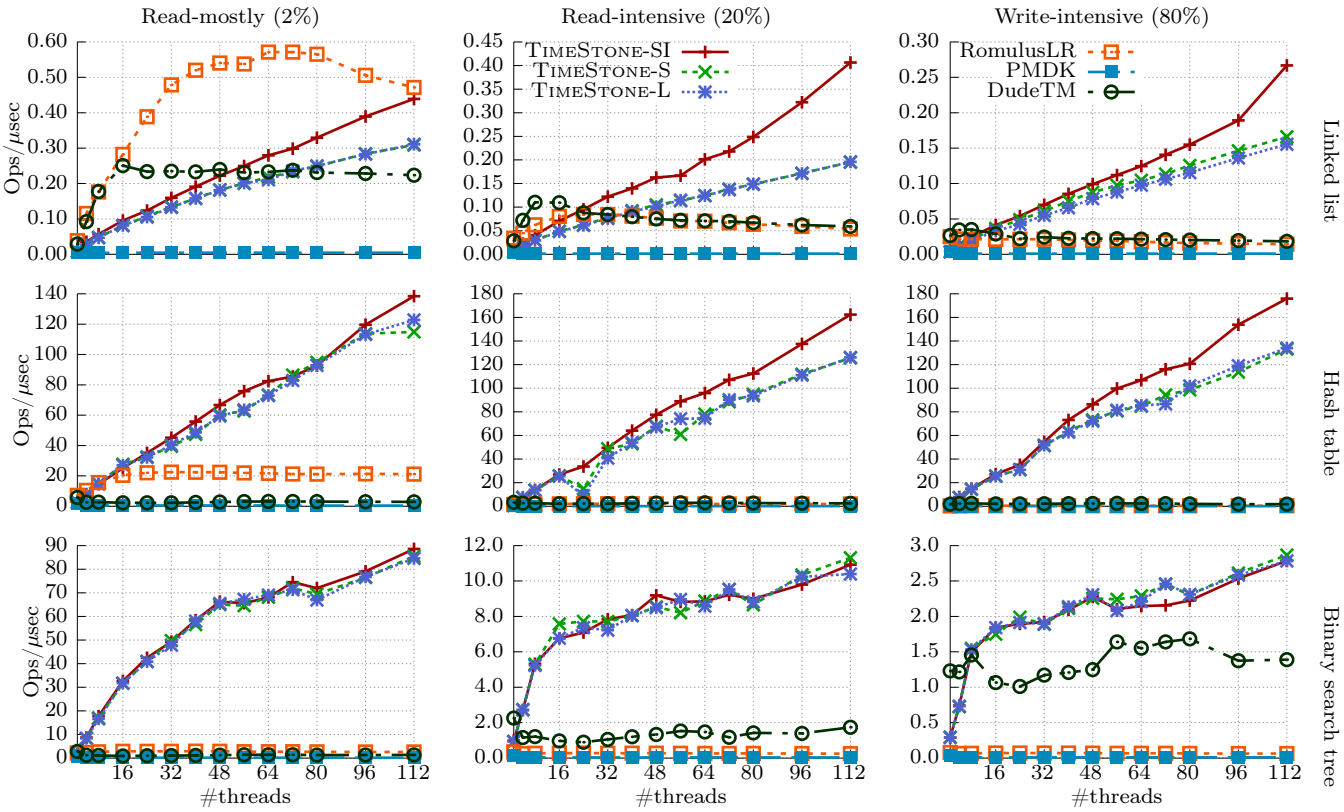

Figure 3.5: Performance and scalability of concurrent data structures: a 10,000 item linked list, hash table (1K buckets), and binary search tree with read-mostly, read-intensive, and write-intensive workloads.

for the read-mostly workload and this happens only in the case of linked list.

For the linked list, DudeTM performs well at a lower core counts and starts to collapse beyond 16 cores. Since DudeTM supports decoupled durability–replicating *all* the persistent data and logs on the DRAM, the foreground thread just accesses the replica on DRAM and writes to its volatile log and the background thread persists the log entries later. That makes foreground writes much faster and for the same reason persisting by the background thread becomes a high latency operation. The background thread becomes a bottleneck since it cannot keep up with the rapidly filling of log entries as the number of foreground thread increases. This eventually blocks incoming writes and leads to a performance collapse.[1] This

[1]The DudeTM code supports only single-threaded background persist.

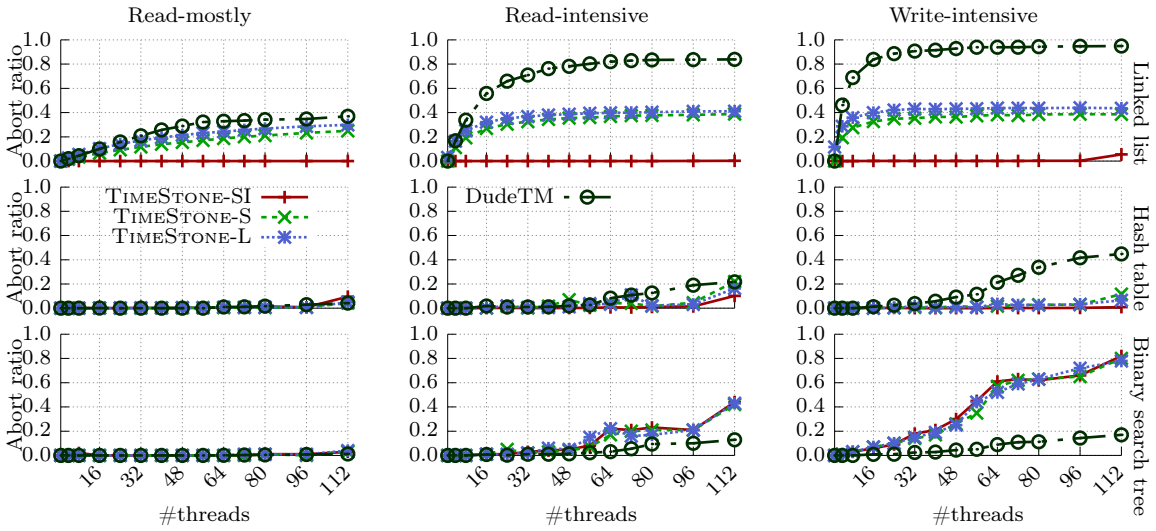

Figure 3.6: Abort ratio of concurrent data structures.

is more evident in read-intensive and write-intensive workloads as it collapses after 6 cores. Unlike DudeTM, TimeStone guarantees immediate durability and hence it has to persist its `OLog` entry upon commit. TimeStone shows upto 10× lesser abort ratio than DudeTM and even the stricter isolation version of TimeStone shows 2.5× lesser abort ratio. This is because the underlying fixed-size central lock table in DudeTM causes spurious aborts. The decentralized resource allocation helps in achieving better scalability and lower abort ratio in TimeStone.

As with Romulus, the single writer thread becomes a bottleneck in read-intensive and write-intensive workloads and for the same reason Romulus performance starts to saturate after 40 cores in the read-mostly workload. Because of the inherently larger critical section in the linked list, the single writer latency is masked in the read-mostly scenario. Note that Romulus never aborts so its abort ratio is always zero.

Note that although PMDK uses a readers-writer lock due its additional logging and NVM allocator overhead in the critical path causes the performance collapse and poor scalability.

Hash Table. For the evaluation, we create a hash table with 1,000 buckets, where each

bucket points to the head of a singly linked list. TimeStone outperforms all the other PTM systems by upto 30× and exhibits a near-linear scalability. The cause for the performance collapse in the other PTM systems is same as observed in the linked list. Pisces [115] implements a similar hash table with the same load factor but with 10× more buckets and TimeStone still outperforms Pisces by 2×-6×. Note that higher number of buckets increases the concurrency and thereby aborts are reduced.[2] We believe that having multiple versions and employing TOC logging for effectively handling crash consistency makes TimeStone performs better than Pisces.

Binary Search Tree. In BST, TimeStone performs upto 10× better than the other and exhibits a better scalability. The slightly skewed scaling and the TimeStone-SI performing same as the stricter versions of Tenet can be attributed to higher chances of lock failure in common ancestor nodes (*e.g.*, root node) causing a spike in the abort ratio. The cause for the performance collapse in the other PTM systems is same as observed in the linked list.

Scalability across Isolation Levels. From our analysis, it is evident that TimeStone shows a better scalability for all the three isolation levels. TimeStone-S and Tenet-L shows a superior scalability when compared to the linearizable PTM systems such as DudeTM and Romulus. Particularly, for hash table and binary search tree, TimeStone-S and Tenet-L shows a similar throughput and abort ratio as that of TimeStone-SI because of better concurrency of those data structures. This is perceptible from the abort ratio also as all the three versions of TimeStone exhibits a similar abort ratio. In DudeTM, the lesser aborts of a hash table can also be attributed to the better concurrency levels in the data structure. Overall, the scalability across read workloads can be attributed to our MVCC-based design while our TOC logging equipped with efficient garbage collection makes TimeStone scalable

[2]Because Pisces code is not publicly available, we compare the performance reported in their paper against our similar hash table configuration.

even for the write-intensive workloads.

3.4.2 Real World Workload

To analyze the impact of TIMESTONE for real-world workloads, we use Kyoto Cabinet [72] and YCSB benchmark [91].

KyotoCabinet. KyotoCabinet is an in-memory database which is internally divided into a number of slots and each slot hosts a number of buckets that point to a binary search tree. Concurrent access of each slot is protected by a per-slot lock. We replaced the binary search tree to the TIMESTONE binary search tree to provide synchronization and crash consistency for database operations. We compare our implementation (KyotoCabinet-TIMESTONE) against the vanilla KyotoCabinet (KyotoCabinet-vanilla) that runs on the faster DRAM and KyotoCabinet-NVM where the binary search trees are allocated on NVM. Note that KyotoCabinet-NVM does not provide crash consistency so there is no logging operations involved. As Figure 3.7 shows, TIMESTONE outperforms other systems and scales even with an additional overhead of providing crash consistency. It is important to note that KyotoCabinet in general is not scalable [100] and the performance starts to saturate after 16 cores. Using TIMESTONE, we make KyotoCabinet scale beyond 16 cores in addition to making KyotoCabinet crash consistent.

YCSB. We implemented a B+-tree for TIMESTONE and DudeTM to evaluate YCSB benchmarks. We set the B+-tree fan-out to 8 and ran 20 million operations using an index benchmarking tool, index-microbench [251]. TIMESTONE significantly outperforms DudeTM.[3] TIMESTONE scales reasonably well for read-dominant YCSB B, C, and D while TIMESTONE shows performance dip for YCSB A. That is because high update ratio in workload A (50%) causes the high latency split and merge operations in the B+-tree. Overall,

[3]We could not show full scalability results of DudeTM because the DudeTM crashes after 30 cores.

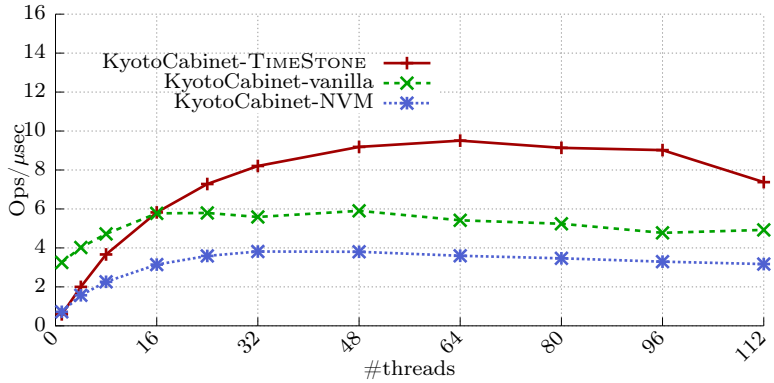

Figure 3.7: Performance comparison of TIMESTONE on KyotoCabinet for read-mostly workload.

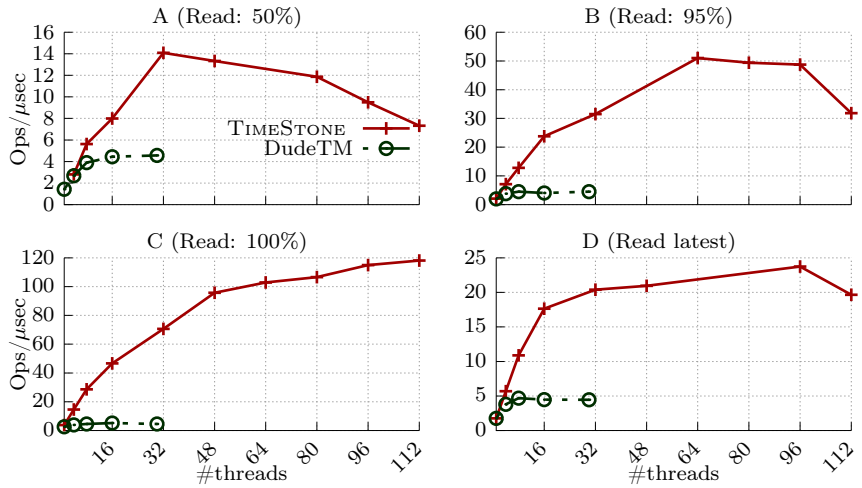

Figure 3.8: Performance of TIMESTONE and DudeTM with YCSB.

TIMESTONE performs upto 6× more and scales better than DudeTM but there is 0.5× dip in the performance for write workloads caused due to the split and merge in B+-tree.

3.4.3 Analysis on Design Choices

In this section, we show how the TOC logging benefits the write amplification, how it provides stability to TIMESTONE even with a larger dataset size and smaller log size.

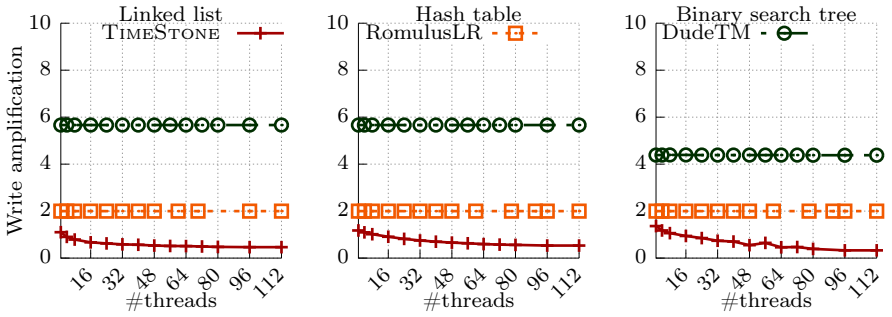

Figure 3.9: Comparison of write amplification incurred in different PTM systems for a write-intensive workload.

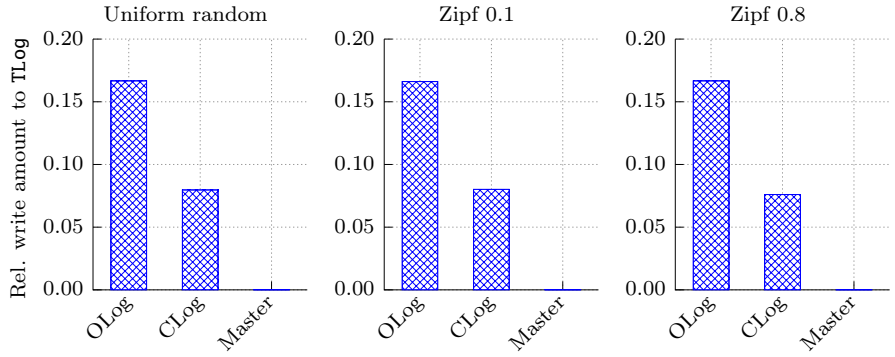

Figure 3.10: The total bytes written for each log in TimeStone for the varying skewness of read-intensive workload. Y-axis is relative to `TLog`.

3.4.3.1 Write Amplification

We ran the persistent data structures for write-intensive workloads and compared it with Romulus and DudeTM in Figure 3.9. The write amplification is defined as ratio of the actual amount of data written to NVM by the amount of user request data.

We infer that TimeStone consistently reduces direct NVM writes compared to RomulusLR and DudeTM for the following reasons. First, RomulusLR must propagate every updates to the NVM backup incurring 2× write amplification. Similarly, DudeTM also writes the user request data to redo log in NVM first and writes it back to data location of the NVM. Redo logging in DudeTM significantly increases the write amplification since it involves not only data but also other metadata (*e.g.*, address) per transaction. But in TimeStone only the

`OLog` write goes to NVM to guarantee immediate durability. Second, `TLog` absorbs a large chunk of redundant NVM writes as Figure 3.10 shows. Only $\sim$ 6% of the total `TLog` writes is being checkpointed to `CLog` and less than 1% of it is written back to the master. The presence of `TLog` becomes more significant if the skewness of the access increases. Third, as depicted in Figure 3.9, the write amplification decreases as the thread count increases. This can be attributed towards higher write coalescing in `TLog`. So overall, the TOC logging reduces write amplification by absorbing redundant writes in `TLog` along with our garbage collection mechanism which prudently controls the NVM writes. Note that write amplification for TIMESTONE and DudeTM reduces about $\sim$0.3$\times$ and 2$\times$, respectively, in case of binary search tree. This is due to the better coalescing chance in BST as common ancestor nodes more frequently updated. Romulus write amplification is unchanged because it has to propagate the updates to the backup heap irrespective of the underlying data structure.

To glimpse the write amplification in PMDK, we modified the `pmemcheck` [132] and observed cacheline flushes incurred in a PMDK transaction with our hash table benchmark. In PMDK, each insert operation incurs 18 flushes consisting of 2 flushes for transaction initialization, 6 flushes for NVM allocation, and 10 flushes for user data update and logging. Even a read-only transaction incurs 2 flushes for transaction initialization. *Overall we observed surprisingly high write amplification, 73.5$\times$, for 1-million transactions with 2% writes.*

3.4.3.2 Sensitivity Analysis

Dataset Size Sensitivity. In order to see how TIMESTONE behaves with varying dataset size, we ran the hash table benchmark for 10$\times$ and 100$\times$ larger dataset (*i.e.*, 100K and 1M elements) than Figure 4.6. As shown in Figure 3.11, TIMESTONE consistently scales even for the 10$\times$ and 100$\times$ larger datasets. However, we observe $\sim$8$\times$ drop in the overall throughput

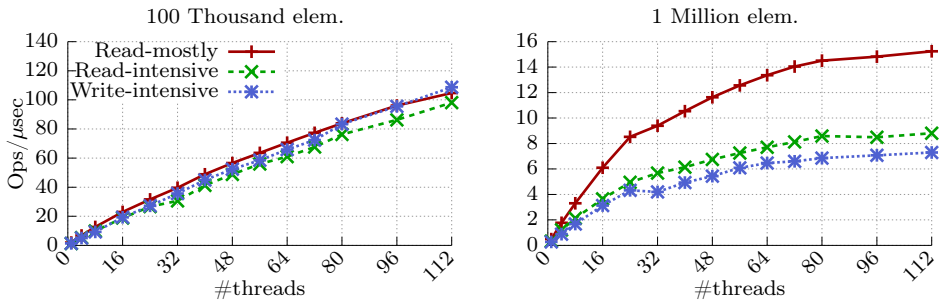

Figure 3.11: Performance of TIMESTONE for a larger hash table size.

for 1M elements; further analysis reveals high cache miss rate (∼60%) for 1M elements due to increased memory footprint compared to mere 2% for 100K elements. Overall, TIMESTONE shows a stable performance and scalability for both small (Figure 4.6) and large dataset (Figure 3.11).

Log Size Sensitivity. To see how TIMESTONE behaves with different log sizes, we ran the hash table benchmark with the read-intensive workload. We varied all three log sizes from `1/8`× to `8`× and measured throughput, the amount of reclamation, and the number of triggered reclamation. We did not observe any drop in performance even when log size is reduced by `1/8`×. The amount of log reclamation is about the same because we ran the same workload. The number of times the log reclamation triggered increases proportionally with the decrease in the log size. For example, if the log size is reduced by 4× then the number of times the log reclamation is triggered increases by roughly 4×. Since our log reclamation is asynchronous triggering it more frequently does not impact the throughput of the system. Overall, our effective log reclamation technique makes TIMESTONE insensitive to the varying log sizes and achieves a high throughput even for a smaller log size.

3.5 Related Work

NVM Optimized Logging. There have been a lot of active research in storage systems to optimize the logging protocols for NVM [76, 125, 142, 153, 155, 210, 211, 214, 216, 229, 249]. Such log optimization techniques consider NVM as a fast caching layer for the disk and leverage it for reducing the recovery cost or the durability cost incurred in the traditional disk-based logging. Techniques such as [125, 210, 249] proposes asynchronous commit policies to reduce the durability cost and to hide the long latency disk persist operations. Other techniques such as [76, 142, 216] leverages NVM to correctly restore the partial disk writes upon recovery. While the database log optimization techniques primarily focuses on reducing the durability cost, we in TimeStone propose *TOC* logging which is geared not only towards reducing the durability cost but also focuses on reducing write amplification to achieve better performance and scalability.

FASE Techniques. Another line of research for developing crash consistent NVM applications utilize a failure-atomic critical section (FASE) approach, guaranteeing atomicity at the level of a critical section granularity [85, 113, 124, 160, 178]. This approach focuses on providing failure atomicity to the legacy lock-based code with little or no focus on the scalability and write amplification issues. Moreover, the traditional FASE-based techniques such as [85, 113, 124, 160] suffers from complex runtime dependency tracking. While the state-of-art iDO logging [178] reduces the dependency tracking overhead but still it needs a specialized compiler support.

Hardware Assisted Techniques. This class leverages STM- or FASE-based approaches and propose new hardware support for guaranteeing atomic durability [112, 134, 137, 139, 158, 159, 203, 208, 222, 234, 275]. They interface with hardware buffers to speed up logging [139, 234, 275] or delegate the process of ordering stores to hardware [112, 158, 159, 203], clearly

demanding significant hardware changes and introducing new logging instructions. Some approaches in this class propose extending hardware transactional memory (like Intel RTM) for making atomic updates to NVM [140, 140, 164]. The performance of these techniques are bound by the L1-L3 cache size and requires changes in the existing cache-coherence protocol [140]. Unlike these techniques, TIMESTONE is completely software-based capable of running on the modern hardware.

3.6 Chapter Summary

In this chapter, we discussed TIMESTONE, a scalable and high-performing PTM framework. We propose *TOC logging* to keep write amplification under the check. MVCC-based design helps TIMESTONE to achieve better scalability and full-data consistency. Also, we support three different isolation levels to improve the applicability of TIMESTONE. We evaluated the TENET against all of the latest PTM works and we showed that TIMESTONE outperforms all of them upto 40× and shows a better scalability. While the prior PTM systems suffers from 2×-6× write amplification, TIMESTONE maintains it below 1. We also presented the real world impact of TIMESTONE by evaluating it with KyotoCabinet and YCSB workloads. The TIMESTONE enabled KytoCabinet and B+-tree shows better performance and scalability. Next chapter discusses how to make TIMESTONE memory safe and fault tolerant.

Chapter 4

Making TimeStone Memory Safe and Fault Tolerant

Direct persistence of NVM opens several challenges in protecting data from software bugs (*e.g.*, "memory scribbles") and media errors. NVM data can be permanently corrupted due to a single memory scribble, which roots from a spatial safety violation (*e.g.*, buffer overflow) or a temporal safety violation (*e.g.*, use-after-free) in a program. Previous studies [78, 95, 104, 105, 165, 200, 201, 202, 209, 230, 233, 242, 243, 269, 273] have shown that such memory safety violations are prevalent in programs (*e.g.*, 70% of CVEs [22, 38, 39]). Since NVM is mapped to the same address space as DRAM, memory safety violations in NVM and DRAM can corrupt NVM data. Besides these software bugs, dense NVMs have a higher random raw bit error rate (RBER) than DRAMs, with RBER closer to NAND flash [245, 270]. Hence, NVM (*e.g.*, Intel Optane) adopts stronger ECC for error correction. Unfortunately, certain hardware errors can still escape the error correction, leading to Uncorrectable Media Errors (UME) in NVM [28, 48].

PTMs [130, 162, 221, 255] are one of the most popular NVM programming models because of their ability to exploit direct persistence. A few recent PTM systems, such as SafePM [82]

and Pangolin [272], attempt to provide NVM data protection by extending `libpmemobj` [130]. A desirable PTM system that offers NVM data protection should (1) offer extensive data protection: protect against both NVM media errors and software memory safety violations in both DRAM and NVM, and (2) incur lower performance overhead and storage costs.

Unfortunately, existing works fail to meet the above criteria. SafePM [82] provides NVM memory safety by instrumenting every NVM access. It does not protect against media errors and memory safety violations in DRAM. The memory instrumentation and the associated metadata incur high performance overhead and storage cost. Pangolin offers data protection with checksum and parity while `libpmemobj` provides fault tolerance by simply replicating the NVM data to a backup NVM region. However, both systems are still vulnerable to memory safety violations, incur high NVM storage cost, and suffer from high performance overhead. As further explained in §4.1.2, in summary, prior approaches compromise the protection coverage [82, 123, 272] while also incurring high storage cost and high performance overhead [82, 130, 272].

This chapter introduces TENET, a principled PTM-based approach that offers an enhanced memory safety and fault tolerance guarantees at a significantly lower performance overhead and storage costs than prior works. Leveraging off-the-shelf hardware features and the concurrency properties of PTM, TENET reduces performance overhead and storage costs without compromising its protection coverage. We realize TENET's memory-safe design principles using the state-of-the-art and highly scalable PTM framework, TIMESTONE [162] that does not provide NVM data protection. In particular, key techniques of TENET are as follows:

- **Hardware-enforced memory domain separation.** Instead of instrumenting every memory access to check for memory safety violations, TENET exploits an existing hardware feature: Intel Memory Protection Keys (MPK) [133, 215], to separate the address space into NVM domains and a DRAM domain. Only the trustworthy TENET library can write to the

NVM domains. Thus, outside the Tenet library, Tenet offloads NVM data protection against memory scribbles to hardware. This enables data protection for most memory access with almost zero overhead.

- **On-first-read and on-commit memory safety enforcement.** Enforcing memory safety at every NVM access in the Tenet library incurs high overhead. Instead, leveraging PTM semantics, Tenet enforces the temporal safety violation only at the first reference of an NVM object and the spatial safety violation only at the commit of a persistent transaction. This, in tandem with the memory domain separation technique, prevents the corrupted data from reaching NVM with very low runtime overhead.
- **Asynchronous hybrid NVM-SSD replication.** Protecting against NVM media errors fundamentally requires creating redundancy. Tenet asynchronously replicates the NVM data to SSD off the critical path to tolerate any number of NVM media errors. It thus offers low storage cost fault tolerance without hindering performance.
- We design Tenet using the above approaches, which to the best of our knowledge is the first high-performance PTM with memory safety and fault tolerance guarantees.
- We evaluate two different versions of Tenet– (1) memory safety only (Tenet-MS) and (2) memory safety and fault tolerance (Tenet) with key data structures and real-world workloads. Our results indicate that Tenet offers enhanced protection at a modest performance overhead and storage cost as compared to state-of-the-art systems.

4.1 Background on NVM Memory Safety and Fault Tolerance

This section first introduces NVM media errors (§4.1.1) and memory safety violation in NVM programs (§4.1.2), followed by discussing the prior PTM works that address the media errors and memory safety violations (§4.1.3).

4.1.1 NVM Media Errors

Figure 4.1 shows the classification of potential errors in NVM. These errors can be classified into hardware errors and software errors. Hardware errors can be further classified into media errors (MEs) and silent data corruptions (SDCs). Media errors are caused by faults in the NVM media such as exceeding the write endurance, power spikes, soft media faults etc that directly corrupt data in the NVM media [245, 270]. SDCs are caused by faults that occur outside NVM media, which indirectly causes data corruption. Examples of SDCs are buggy NVM firmware, faults in CPUs, memory controllers, or other hardware components [89, 149]. Handling SDCs is a separate research area and it is out of scope of this work.

Hardware media error (ME) correction.. Commercially available NVMs implement error-correction code (ECC) in hardware to detect and correct media errors. For example, Intel Optane DCPMM uses hardware parity to detect any-bit errors, and it can correct up to two 2-bit errors [27]. The NVM hardware transparently fixes such correctable media errors (CMEs). However, uncorrectable media errors (UMEs) will be reported for software intervention as detailed below.

Reporting uncorrectable media error (UME) to software.. The OS receives the reports of UMEs; and it can pass it to the application. Specifically, when a CPU accesses an NVM page affected by UMEs, the NVM hardware sends a poison bit along with the relevant data to the CPU. Upon encountering the poison bit, the CPU raises a memory check exception (MCEs) for the OS to handle. Currently, Linux handles the MCE by adding the corrupted page to the bad block list and sends a `SIGBUS` signal to the application [19, 48]. Then the OS leaves the responsibility to the application for fixing UMEs during the recovery phase [28]. *We note that, although the NVM is byte-addressable, UMEs are reported to the software at the page granularity due to the blast radius effect [11].*

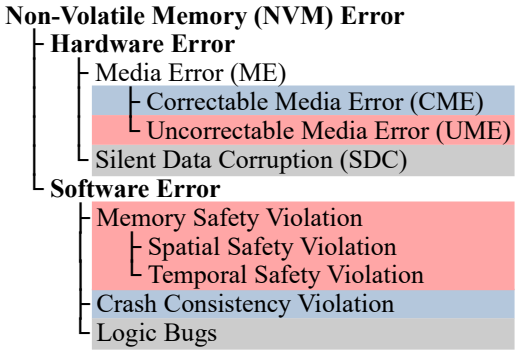

Figure 4.1: Classification of errors in NVM. Tenet handles *UME*, *Spatial and Temporal Safety Violation* bugs (red). Tenet relies on the hardware ECC to fix *CME* and the underlying PTM to handle *Crash Consistency Violations* such as atomicity and persistence ordering (blue). *Silent Data Corruption* in the hardware (*e.g.*, CPU faults) and *logical bugs* in the application are out of scope (grey).

4.1.2 Memory Safety in NVM Programs

We categorize software "scribbles", which corrupt NVM data, as spatial and temporal memory safety violations (Figure 4.1). *Spatial safety violations* happen when memory is accessed beyond its allocated range. Buffer overflows and array out-of-bound accesses are classical examples. *Temporal safety violations* happen due to dangling pointers; *i.e.*, when accessing an already freed (*use-after-free*) or accessing a reallocated address range (*use-after-realloc*). These memory safety bugs are even more dangerous in NVM than DRAM because the NVM data will be corrupted forever and a simple system restart would not fix these issues. *Note that memory safety bugs on either DRAM or NVM region of an application can cause NVM data corruption since the NVM region is mapped directly to application's address space.*

4.1.3 Prior NVM Data Protection Approaches

Memory safety in NVM programs.. Prior works – Pangolin [272], SafePM [82], and Corundum [123] – include mechanisms to protect NVM data from memory safety violations. Pangolin extends `libpmemobj` [130] and uses per-object checksum to detect spatial safety violations. SafePM adds AddressSanitizer [231] to `libpmemobj` transaction to detect spatial

and temporal safety violations on NVM data. Corundum is a Rust-based NVM programming library and leverages Rust's type system to statically enforce spatial and temporal memory safety. However, they have some critical limitations. *First, none of these approaches prevent NVM data corruption due to memory safety violations on "DRAM data".* Suppose that the buggy code inside a transaction causes a buffer overflow on DRAM data; such spatial safety violations on DRAM can scribble arbitrary memory location, including NVM data. *Moreover, none of them guarantee to protect NVM data from temporal safety violations.* Pangolin does not check temporal safety violations. Meanwhile, SafePM does not detect use-after-realloc bugs. Even with Corundum, the developers still have the responsibility to guarantee type and memory safety for the "unsafe" Rust code, both can result in spatial and temporal safety violations.

Both Pangolin and SafePM suffer from high performance overhead and introduce additional performance bottlenecks. Pangolin calculates and verifies checksums on the critical path, imposing high performance overhead. Furthermore, it verifies checksum only for write transactions (*i.e.*, read transactions are unprotected). SafePM instruments every NVM access to check for memory safety violation, which is costly. SafePM further introduces extra UNDO logging overhead over the already existing expensive logging in the `libpmemobj` to guarantee crash consistency for its memory safe metadata.

Fault tolerance against UME.. To protect against UME, `libpmemobj` supports replicating data on NVM. However, it replicates data on the write critical path, leading to high performance overhead. Furthermore, storing the replicated data on NVM wastes the precious NVM space, doubling (2×) storage cost. Pangolin uses parity for fault tolerance; however, *parity calculation on the critical path causes high performance overhead and it unnecessarily serializes the transactions which affects the write scalability.* Further, Pangolin can recover up to one page within a parity region; a data loss will happen if UME occurs on more than a

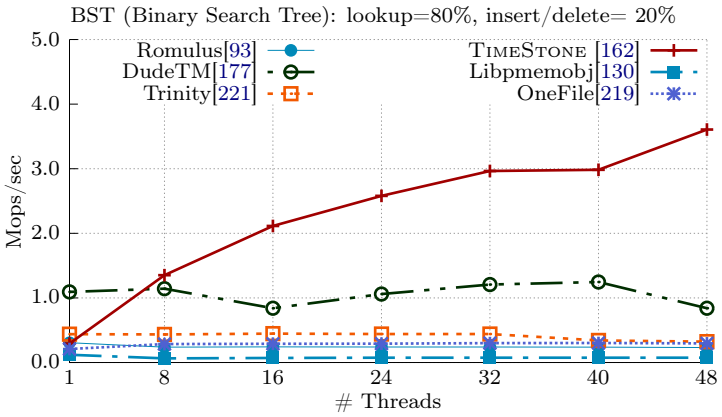

Figure 4.2: Performance of TimeStone against other PTMs. None of the PTMs are memory safe or fault tolerant against UME.

page. SafePM and Corundum do not provide any fault tolerance against UME.

4.1.4 Prior PTMs for NVM

`Libpmemobj` [130] has been the de-facto PTM. However, it suffers from high performance overhead and poor scalability. Thus, several new PTMs focus on addressing its limitations [93, 115, 177, 192, 219, 220, 221, 255]. Figure 4.2 shows that none of the existing PTMs, except TimeStone [162], scale beyond 8 cores even for a read-intensive workload. Further, TimeStone performs up to 8× better than the existing PTMs. Based on this observation, we chose TimeStone [64, 162] as the transaction abstraction for Tenet. Moreover, designing memory safety and fault tolerance techniques for such a high performance PTM is challenging as even a small bottleneck can compromise its original scalability and performance. We introduce the relevant design aspects of TimeStone below.

Multi-version concurrency control.. TimeStone follows multi-version concurrency control (MVCC). With MVCC, TimeStone supports non-blocking reads and concurrent disjoint writes, achieving high concurrency. For each object created by the application (*e.g.*, B-tree node), TimeStone allocates a *master object* on NVM (see Figure 4.3). On

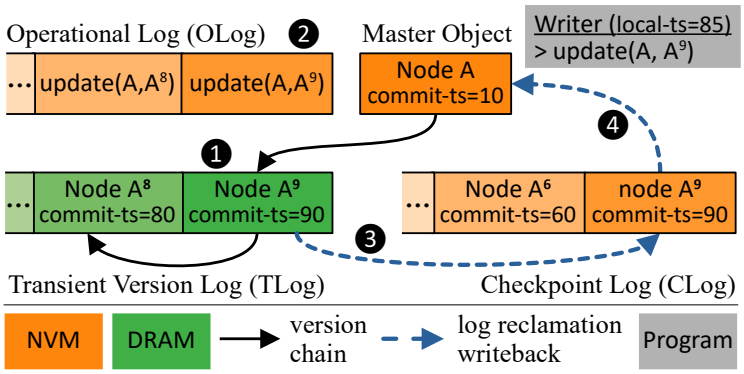

Figure 4.3: An illustrative example of updating `Node A` to its 9th version (`A`^{9}) in TIMESTONE.

updating a master object, TIMESTONE creates a new version (❶) on DRAM, chaining multiple version objects from new to old object's age. TIMESTONE dereferences the right version object during the dereference phase with the help of timestamps. Each version object gets assigned a timestamp when it is committed (`commit-ts`). Also, each transaction gets a timestamp (`local-ts`), which denotes the transactions' start time. TIMESTONE traverses the version chain and chooses the most recent version of an object based on these timestamps (*i.e.*, `commit-ts<=local-ts`). This guarantees a consistent snapshot of NVM data for all transactions at any given time.

Operational log based immediate durability.. TIMESTONE uses a DRAM-NVM hybrid logging technique, named TOC logging for efficient crash consistency. The TOC logging consists of Transient Version Log (`TLog`) on DRAM, Operational log (`OLog`) and Checkpoint log (`CLog`) on NVM, as illustrated in Figure 4.3. TIMESTONE creates a new version on `TLog` (❶), and logs the performed operation to the `OLog` (❷) for immediate durability. An operational log entry is typically much smaller than the conventional undo/redo logging, which duplicates the data, thus making crash consistency efficient.

Asynchronous log reclamation and replay based recovery.. As more versions are created, `TLog` eventually becomes full, triggering log reclamation. When `TLog` is reclaimed, the latest version of an object on `TLog` (`A`^{9} over `A`^{8}) is checkpointed to the `CLog` (❸). Similarly,

when `CLog` becomes full, the latest checkpoint (`A`^{9} over `A`^{6}) is written back to the master object (❹). To recover from a crash, TimeStone first applies the checkpoints in `CLog` to the respective master objects and reverts them to a consistent snapshot. Then the `OLog` is executed to recreate all the updates that are lost on `TLog`.

4.2 Overview of Tenet

4.2.1 Threat Model and Assumptions

Tenet aims to protect against spatial and temporal memory safety violations in buggy application code. Furthermore, Tenet considers the possibility of a memory safety violation on DRAM data corrupting NVM. Tenet also aims to guarantee fault tolerance for NVM data against the uncorrectable media errors (UMEs). PTMs in general and TimeStone in particular cannot guarantee **ACID** properties for the application code that is outside the transaction or when the PTMs' APIs are misapplied. This applies to Tenet as well, *i.e.*, it cannot guarantee memory safety and fault-tolerance for the code outside the transaction. Tenet is not designed to handle SDC that occur outside the NVM media. Protection against the adversarial attacks (*e.g.*, control-flow attacks) is out-of-scope. However, the protection techniques and mechanisms against SDC and control-flow attacks can be orthogonally deployed to Tenet. In Tenet, application code is distrusted while Tenet library code and OS kernel are considered as a trusted computing base (TCB).

4.2.2 Design Goals

- **Protect NVM data from memory safety violations.** Tenet should detect all spatial and temporal safety bugs not only from NVM but also from DRAM. Any memory safety bugs either in DRAM or NVM code should not corrupt NVM data.
- **Protect NVM data against UMEs.** Tenet should provide a robust fault tolerance

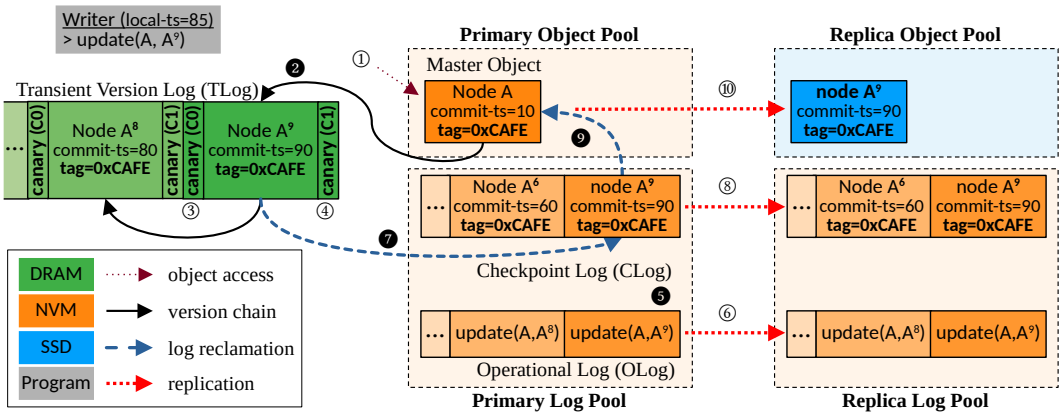

Figure 4.4: Overall architecture of TENET with an example of updating `Node A` to its 9th version (A^9). ⓫ denotes the newly added memory safety checks and replication to the TIMESTONE transaction. Note that the application has read/write access to DRAM and read-only permission for NVM. When accessing `Node A`, TENET validates its temporal safety by comparing the tags, `0xCAFE` (①). If the tags do not match, the transaction is aborted. Otherwise, the writer proceeds to traverse the `Node A`'s version chain, makes a copy of the latest version (A^8) in its `TLog` and updates it to A^9 (❷). Upon commit, `Node` A^9 is validated for spatial safety by checking the canary values (③ and ④). The transaction is aborted if the validation fails. Otherwise, the writer commits the transaction by updating its `OLog` (❺) for durability and it also synchronously updates the replica `OLog` for fault tolerance (⑥). When reclaiming the `TLog`, `Node` A^9 is once again validated for spatial safety before checkpointing it to the `CLog` (❼) followed by synchronously updating the replica `CLog` (⑧). Similarly, when the `CLog` is full, TENET writes back the latest checkpoint (`Node` A^9) to the original master object `Node A` (❾). The updated Node A is then *asynchronously* replicated to the disk (⑩).

mechanism to recover and restore NVM data from UMEs transparently.

- **Low performance and storage overhead.** TENET aims to be a *practical* system that offers an enhanced protection scope and strong fault tolerance at a minimal performance and storage overhead.

4.2.3 Design Overview

TENET re-purposes the multi-versioning and transactional semantics of TIMESTONE to achieve its design goals. Below we introduce TENET's main techniques as illustrated in Figure 4.4.

(1) Separation of NVM protection domain from DRAM. A memory safety bug (*e.g.*, out-of-bound write) either in DRAM or NVM can result in NVM data corruption. Enforcing

full memory safety in every single memory access incurs prohibitive runtime overhead as prior studies show [82, 272].

To prevent unauthorized NVM writes without checking every single memory access, Tenet grants the write permission to the NVM region only for the TCB *i.e.*, the Tenet library code. In other words, the application code has read-only permission for the NVM, and consequently, it only writes on DRAM. When the application commits its transaction, writer thread gets write permission to execute the Tenet library code which propagates the updates on DRAM to the NVM.

Tenet completely segregates DRAM and NVM regions so that all new version and master objects are created on DRAM (referred to as *DRAM Objects*). Therefore, Tenet application code does not require write access to NVM, as it writes only to the DRAM region. If a buggy application code tries to write to the NVM region, it will receive an exception (`SIGSEGV`) from Tenet and will be terminated. Tenet exploits Intel Memory Protection Keys (MPK) [133, 215] to efficiently switch NVM permissions for each thread.

(2) On-commit spatial safety enforcement. As applications can always write to the `TLog` (*i.e.*, DRAM), it is vulnerable to arbitrary memory scribble. A corrupted DRAM object can be eventually propagated to `CLog` (❻ in Figure 4.4) and the master object (❽), consequently corrupting the NVM data. We propose *on-commit spatial safety enforcement* to prevent corrupted DRAM objects from reaching NVM. Tenet adds eight byte canary values at the start and at the and of a DRAM object during its creation (❷). Specifically, Tenet assigns a random value to `C0`, and the hash of `C0` and its location (`xor(C0,&C1)`) to `C1`. When an application commits the transaction, Tenet inspects the integrity of canary values of all DRAM objects in that transaction (③ and ④). If the canaries are compromised (*i.e.*, `C0 != xor(C0,&C1)`), then Tenet aborts the transaction and gracefully terminates without propagating the corrupted objects to NVM.

Our on-commit spatial safety enforcement is efficient with minimal performance overhead. Unlike the prior approaches [82, 202, 230, 273], our technique avoids reading additional metadata, and it checks the integrity only once during the transaction commit. Note that NVM objects do not have canary values and thus no NVM space overhead.

(3) On-first-dereference temporal safety enforcement. Even after an NVM (master) object is freed (and then reallocated), a program still can reference it via dangling pointers which can corrupt the NVM data in unintended ways.

We propose *on-first-dereference temporal safety enforcement* to efficiently enforce temporal safety of NVM objects with a minimal runtime overhead. TENET uses a tag-based approach, which essentially checks if a pointer points to the right object by comparing tags associated with the pointer and the pointed object. When TENET creates an NVM (master) object, it assigns a 2-byte random integer as a tag of the object (*e.g.*, `0xCAFE` for `Node A` in Figure 4.4). We encode this 2-byte tag in the upper 16-bit of a pointer, which is unused in the x86 architecture. When the object is freed, its associated tag on the header set to zero for detecting use-after-free. When the object is dereferenced *first time* in a TENET transaction (①), TENET checks whether the encoded tag in the pointer matches with the tag in the pointed object. If the tags do not match, it means the pointer points to the already-freed/-reallocated object (*i.e.*, dangling pointer), which violates temporal safety. In this case, TENET aborts the transaction immediately.

Our approach is efficient and imposes minimal performance overhead because it checks the temporal safety of each object only once in a transaction. Also, accessing the inlined tags is cache-friendly, which, unlike prior approaches [82, 201, 230, 273], requires no additional metadata lookup.

(4) Off-critical path NVM replication to SSD. TENET replicates all NVM data; in

the case of a UME, corrupted NVM pages can be restored using the replica. The main challenge in designing a replication scheme is minimizing the performance overhead and storage cost. While replication to another NVM region can be performance efficient, it incurs 2× higher capacity cost. Instead, we propose a hybrid NVM-SSD replication technique; Tenet *asynchronously* replicates the master objects to SSD (⑩) and *synchronously* replicates the transaction logs (`CLog` and `OLog`) to NVM (⑥, ⑧). Master objects, are application data structures, which can be large and also potentially occupy the entire NVM space. Hence, Tenet replicates master objects to the SSD off the critical path to reduce both storage cost and performance overhead. Although the replication is asynchronous, Tenet guarantees *loss-less* NVM data recovery by prudently leveraging the transaction logs and grace period semantics. Meanwhile, transaction logs are small and finite, so Tenet replicates them to NVM to reduce performance overhead. Further, Tenet is also capable of recovering from multiple simultaneous UMEs occurring in one or multiple NVM pages. We explain this design, its correctness and recovery guarantees in §4.3.4 and §4.3.5.

4.2.4 Putting It All Together For TimeStone

Tenet makes the NVM read-only for all except the Tenet's library code. So the NVM objects in TimeStone do not need spatial safety checks as they are read-only objects. Tenet enforces temporal safety checks for all NVM objects (using pointer tags) during the object dereferencing to detect dangling pointers. On the contrary, DRAM objects are vulnerable to application scribbles (due to write permission) hence Tenet enforces on-commit spatial safety checks using the canary bits. DRAM objects do not need separate temporal safety checks as they are managed internally by Tenet; *i.e.*, as DRAM objects are accessed via the respective NVM object, enforcing temporal safety for NVM objects indirectly guarantees it for DRAM objects. We discuss the correctness of these techniques in §4.3.3. TimeStone can not handle UMEs, so Tenet proposes to replicate master objects and transaction logs

to SSD and NVM respectively; in the event of a UME, NVM data can be restored using the NVM/SSD backup. In a nutshell, we optimally apply TENET's memory safety techniques to the vulnerable parts of TIMESTONE and organically redesigned it to guarantee full memory safety. If TENET was to be used for other PTMs, then its techniques can well be applied, albeit it may require some engineering effort. We discuss this further in §4.5. Refer to Figure 4.4 for a summary on lifecycle of a TENET transaction.

4.3 TENET Design

In this section, we first describe TENET transaction design (§4.3.1) followed by the design of memory safety (§4.3.2-§4.3.3), fault tolerance replication (§4.3.4), and recovery (§4.3.5).

4.3.1 TENET Transaction

Below we explain how TIMESTONE transaction is redesigned using TENET to enforce memory safety and fault tolerance.

4.3.1.1 NVM Object Dereference

Object dereferencing in TIMESTONE (§4.1.4) only traverses the version chain and returns the correct version, whereas in TENET, object dereferencing is a two-step process.

(1) Temporal safety validation. TENET validates the master object pointer for temporal safety (§4.3.3.2) to detect dangling pointers; transaction aborts if the validation fails (§4.3.1.4).

(2) Version chain traversal. If the object passes the validation, then TENET dereferences the correct DRAM object or directly the master object if the version chain does not exist.

4.3.1.2 Updating an Object

In TENET, a writer updates a master object by creating a new DRAM object as done in the TIMESTONE. However, TIMESTONE allows its users (application) to allocate and write to the NVM when creating new master objects. Thus, a buggy application can easily corrupt the NVM region. In TENET, this is restricted to prevent direct NVM writes; so the application allocates and writes to a new master object (*shadow master object*) on the DRAM and then during the commit phase TENET library creates a corresponding NVM copy only if the writes pass the spatial safety violation checks.

4.3.1.3 Committing a Transaction

In TIMESTONE, the commit procedure updates the `OLog` to guarantee durability and then makes all the updates atomically visible. TENET's commit procedure happens in three phases:

(1) Spatial safety validation. All the new versions and shadow master objects created in a transaction are validated for spatial safety violations (§4.3.3.1). Upon successful validation, TENET allocates and updates the persistent master object from the corresponding shadow master object.

(2) Transaction durability and replication. Updating `OLog` guarantees durability, and replicating it ensures fault tolerance (§4.3.4.1). Also, TENET adds all the newly created master objects in **(1)** to the replica buffer to trigger async disk writes using background workers (§4.3.4.2).

(3) Publishing the updates atomically. TENET makes the updates atomically visible by adding the new versions to their respective version chain, and this procedure is exactly the same as TIMESTONE. Additionally, TENET frees all the shadow master objects, if any, and exits the critical section.

4.3.1.4 Aborting a Transaction

Common abort procedure. TENET rolls back any used log space, lock status, and reclaims all the shadow master objects and also its NVM counterpart if one exists. This is common for all three abort cases described below.

Abort due to lock conflict. During the object update (§4.3.1.2), if the writer fails to acquire a lock, it aborts the transaction. This is a benign abort *i.e.*, no memory safety violations, so TENET performs the common abort procedure and retries the transaction after the backoff period.

Abort due to memory safety violation. All ongoing transactions are aborted if a transaction aborts due to spatial safety or temporal safety violation. TENET executes the common abort procedure and returns an exception.

Abort due to a UME. The OS notifies a UME by sending a `SIGBUS` signal. TENET's signal handler catches the signal, returns a UME exception to notify the application, and gracefully terminates the process. TENET fixes the affected NVM region during the recovery process (§4.3.5).

4.3.2 Unauthorized NVM Write Prevention

TENET already prevents application code from directly writing to the NVM by using DRAM objects for the updates. However, a buffer overflow on DRAM can corrupt the NVM data as NVM is directly mapped to the applications' address space. TENET employs Memory Protection Keys (MPK), a hardware feature available in the Intel systems [98, 102, 120, 215, 241] to detect NVM writes out of TENET library code.

Using MPK to enforce read-only NVM access. With MPK, a page can be assigned

to one of the 16 available protection domains. The assigned protection domain is encoded in the page table entry. A thread's access permission to the protection domains is controlled at the per-thread level via a user-accessible register, `PKRU`. A thread can switch its access permissions to the protection domains by writing to the `PKRU` register, which only costs 20 CPU cycles. In TENET, each NVM pool is assigned a unique protection key during pool creation. Only the TCB (*i.e.*, TENET library code) is allowed to write to the NVM pool. Thus, a thread grants itself read-write permissions to the corresponding NVM pool during the library code execution and revokes it before exiting the library. As a result, if the application writes to NVM (*e.g.*, due to buffer overflow), MMU prevents the access and OS sends a `SIGSEGV` signal. Thus, any spatial safety violations due to a buggy write is contained within the DRAM region.

4.3.3 Enforcing Memory Safety

In this section, we explain the spatial (§4.3.3.1) and temporal safety design (§4.3.3.2). In §4.3.3.3, we explain the array interface as an example, and how the interface provides memory safety.

4.3.3.1 On-commit Spatial Safety Design

TENET enforces spatial safety for all DRAM objects to prevent NVM data corruption due to a buggy DRAM write.

Technique. As illustrated in the Figure 4.4, all DRAM objects are assigned two 8-byte canaries at the start `C0` and at the end `C1`. Specifically, `C0` is a random value and `C1` is the hash of `C0` and its location (`xor(C0,&C1)`). TENET inspects the integrity of canary bits to detect buffer overflows and underflows.

On-commit validation. When the application commits its writes (§4.3.1.3), TENET

inspects canary bits for all the newly created DRAM objects. A transaction is committed only when both `C0` and `C1` are intact in all the DRAM objects. Otherwise, the transaction aborts and discards all the corrupted objects. An erroneous transaction can corrupt the DRAM objects outside of the current transaction *i.e.*, the ones that are part of other concurrent transactions or the ones that are not part of any ongoing transactions at all. To detect such cases, TENET places an 8-byte canary at the start and the end of the transactions' write set. Note that all the DRAM objects including the shadow master objects are part of a transactions' write set. TENET validates the write set canaries before and after each step of the commit process (§4.3.1.3). This ensures that a transactions' write set (*i.e.*, DRAM object) has not been corrupted by an erroneous concurrent transaction, particularly between the initial validation (**(1)** in §4.3.1.3) and the publication of the updates (**(3)** in §4.3.1.3). However, if the write set canaries are found to be compromised then TENET aborts all the transactions as explained in §4.3.1.4.

Correctness. Deferring spatial safety checks until the commit time does not violate the correctness as the other concurrent transactions *can not* observe any uncommitted DRAM objects. Although a rare case, to avoid reading a DRAM object that is corrupted (after it commits), TENET performs spatial safety check before dereferencing a committed DRAM object. Subsequently, the DRAM objects (❼ in Figure 4.4) and the shadow master objects are re-validated before and after copying to the NVM to prevent *Time-of-Check-Time-of-Use (TOCTOU)* bugs [65]. If an DRAM object is found to be corrupted post the copy operation then the corresponding NVM object will be safely reclaimed as part of the transaction abort procedure. Finally, TENET cannot detect the corruptions that occur without overwriting the canaries, aka intra-object overflows. We discuss this further in §4.5.3.

4.3.3.2 On-first-dereference Temporal Safety Design

TENET enforces temporal safety for all NVM (master) objects to detect dangling pointer dereference. Accessing an already free-ed (or reallocated) address can corrupt the NVM data due to use-after-free (or use-after-realloc) bugs.

Technique. To detect dangling pointers, TENET assigns an *unique 2-byte tag* for all the master objects, which is stored in the object's header (`0xCAFE` in Figure 4.4) at the time of its creation. A copy of this tag is also encoded in the *unused* upper 16-bits of the master objects' address. On deallocating the master object, the tag in the objects' header is set to zero.

On-first-dereference validation. When the application accesses a master object for *the first time in a transaction*, TENET validates the pointer to the master object before traversing the version chain (§4.3.1.1). TENET extracts the tag encoded in the master objects' pointer and compares it with the tag stored in the respective master objects' header. If they match, then it is a valid pointer. When an application accesses a master object with a dangling pointer, tag matching would fail; the tag in the header would either be zero (if the address is already freed) or different random value (if the free-ed address is reallocated). In that case, TENET cuts the version chain access and aborts the transaction.

Correctness. Once a master object is successfully dereferenced, it can be safely used without any further temporal safety checking *within the transactions' lifetime*. This is because TENET (and TIMESTONE) uses an RCU-style, epoch-based garbage collection scheme so it never frees an object (and its versions) with live references from other transactions; *i.e.*, an object will be free-ed only when all the transaction that has live references exits. Also, a DRAM object can be dereferenced only via its NVM object and TENET cuts the version chain access upon detecting a dangling pointer, which indirectly guarantees temporal safety for DRAM objects.

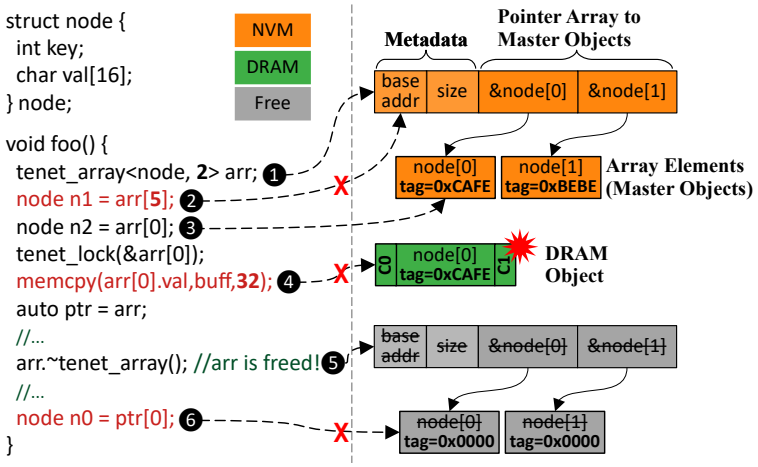

Figure 4.5: Memory safety design for arrays. ❷ and ❹ are spatial safety violations due to out-of-bound read (detected by bounds checking) and write (detected using canaries), respectively. ❻ is temporal safety violation due to use-after-free (detected using pointer tags).

4.3.3.3 Spatial and Temporal Safety for Array Objects

In TIMESTONE, an array is stored and accessed as a single pointer. Even if the application just reads/writes to one array element, TIMESTONE dereferences the entire array. Such a design is highly unsafe. For instance, once the entire array is dereferenced, a buggy application can read/write out-of-bounds resulting in an undetected corruption. This is a notoriously hard problem even in the DRAM world. To address this, we redesigned the array interface in TENET. *An array is internally represented as an array of pointers where each array index stores a pointer to its element.* With this design, TENET dereferences only the array index that the application intends to read/write. If the application accesses an index that is out-of-bound, TENET aborts the transaction.

Array interface. ❶ in Figure 4.5 presents the TENET's array interface. In TENET, each array element is a master object; and an array consists of pointers to these master objects along with the base address and size information. This representation is internal and the application accesses its array in the traditional *C semantics*. We do not present the pseudocode for our interface due to space limitations. Essentially, TENET retains the

C-style semantics by leveraging C++ operator overloading. `tenet_array` class overloads the necessary operators to hide the internal representation. For instance, the array access operator (`[]`) is overloaded to perform bounds checking, then access the master object at the index. Similarly, other operators (`=`, `+`, `-`, etc) are also appropriately overloaded to retain the programmability and to make the interface transparent. However, this representation requires additional memory to maintain pointers to the array elements. An `N` element array requires a space of `N*sizeof(N)`, whereas TENET requires an additional `sizeof(void*)*N` space to maintain the pointers.

Memory safety validations. Figure 4.5 illustrates how TENET enforces memory safety for arrays (`arr` with two elements). The canary-based spatial safety and the tag-based temporal safety apply to every array element. In addition, TENET performs bounds checking for every array dereference using the base address and size metadata *i.e.*, `index > size` (❷). In ❹, a transaction writes to the `val` out-of-bounds, TENET detects this violation by inspecting the corrupted canary bits in the commit phase. In ❻, transaction dereferences a dangling pointer (freed in ❺) and TENET detects it by comparing the tags (`0xCAFE` $\neq$ `0x0000`) during the object dereference.

4.3.4 Enforcing fault tolerance Against UMEs

This section explains the synchronous log replication and the off-critical path master object replication design to guarantee fault tolerance against UMEs.

4.3.4.1 Transaction Log Replication

As illustrated in Figure 4.4, the primary log pool on the NVM consists of all the transaction logs (`OLog` and `CLog`). TENET maintains a consistent backup of the primary log pool by *synchronously* replicating the logs on the critical path, *i.e.*, when an `OLog` or a `CLog` in the

primary log pool is updated, the corresponding log in the replica log pool is also updated. Atomicity for primary and replica log writes is inherently guaranteed by the transactions' commit protocol (§4.3.1.3); *i.e.*, TENET commits a transaction only when both the logs are updated. So, if a crash happens before updating the replica log, then the transaction is considered to be aborted and the partially written log entries are discarded during the recovery phase. Similarly, during log reclamation, the primary log is reclaimed first and the replica log is reclaimed up to the same point to maintain consistency. TENET ensures that pages in the primary and the replica log pool do not overlap by maintaining two disjoint NVM pools for the primary and replica log pool. In this way, TENET *can recover from multiple UMEs even if it spans across many pages within a log pool.*

Why replicate logs on the critical path? TIMESTONE buffers the updates to the master objects in the `OLog` and `CLog` to maximize the write coalescing. Hence, if the logs in the primary pool are corrupted, it may cause a significant amount of data loss during the recovery. As a result, TENET replicates the primary log pool synchronously to ensure that there is always a consistent backup. Thus, TENET can simply use the replica log pool to recover the NVM data *without losing any committed updates.* TENET uses NVM to reduce the performance overhead as the replication is done in the critical path.

4.3.4.2 Off-critical Path NVM Replication to SSD

TENET makes three critical design choices for a performant and cost-efficient NVM (master) objects replication: (1) objects are replicated to SSDs instead of NVM to reduce the storage cost overhead, (2) replication is performed out of the critical path to reduce the performance overhead (§4.3.4.3), and (3) TENET uses grace period semantics to enforce NVM-SSD consistency to guarantee loss-less recovery (§4.3.4.4).

4.3.4.3 Off-critical Path Writes to SSD

TENET leverages `io_uring` [31] for accelerating SSD writes. `io_uring` is a high-performance asynchronous IO framework. `io_uring` maintains two queues, a submission queue (SQ) where the TENET adds its disk write requests and a completion queue (CQ) where TENET can poll for the completed disk writes. Both queues are shared between the kernel and the user space, which further reduces the context-switching overhead for request submission and polling.

Technique. TENET maintains a per-writer replica buffer in the NVM, where writers enqueue the new master objects that are created in the ongoing transaction and the objects that are updated with the latest checkpoints from the `CLog` (❽ in Figure 4.4). TENET then spawns multiple workers to visit the per-thread replica buffer and issue the disk writes using `io_uring`'s submission queue. The workers then poll for the request completion in the `io_uring`'s completion queue and exit only when all the requests are completed. TENET creates a separate disk file for each master object pool; during replication, TENET *writes a master object at the disk file offset, same as the objects' corresponding NVM file offset.* This is critical to correctly roll back the corrupted page from the disk to the NVM during the recovery.

4.3.4.4 Enforcing NVM-SSD Consistency

Although replication is asynchronous, TENET *guarantees that no committed data will be lost upon either a crash or a UME.* TENET accomplishes this by leveraging the `OLog`, `CLog`, and grace period detection.

Grace period detection in TIMESTONE. A grace period is the quiescence period, in which all application threads that entered the critical section (since the start of detection), finish, and exit their respective critical section. A background thread (`gp-thread`) continuously

detects the grace period, and publishes the detected grace period timestamp. TIMESTONE uses the timestamp to safely reclaim/free the obsolete entries/objects in the `TLog`, `CLog`, and the `OLog`. TENET extends this design to enforce NVM-SSD consistency.

Modified grace period detection in TENET. To detect a grace period, the `gp-thread` not only waits for all the threads to exit the critical section but also waits for all master objects that are created/updated by these threads to be written to the SSD. The key invariant is that *when a grace period is detected, it guarantees that all master objects created/updated in that window are persisted to the SSD*. This means that the `TLog`, `OLog`, and the `CLog` will not be reclaimed until the disk writes are guaranteed to be persisted. That is because `gp-thread` will not publish the grace period timestamp unless the disk writes are completed and without it the logs can not be reclaimed. *In a nutshell, all the updates that are not persisted in the SSD are guaranteed to be either in the `OLog` (newly created master objects) or in the `CLog` (updates to the existing master object).*

Guaranteeing consistent loss-less recovery. If a UME occurs before the SSD writes finish, during recovery, TENET can restore the NVM objects with the stale SSD replica (from the previous grace period). Then it uses the `CLog` to update the existing master objects with the latest checkpoints and uses `OLog` to recreate the new master objects that are missing in the stale replica. Note that TENET maintains a consistent backup of `OLog` and `CLog` at all times (§4.3.4.1). Also, the `OLog` and `CLog` execution are idempotent *i.e.*, re-executing the same log entries multiple times does not violate the consistency. TENET can tolerate multiple UMEs across any number of pages in a master object pool as it replicates to the SSDs. Given at least one of the log pools is consistent, TENET can recover up to the last committed transaction. Note that even if both the log pools are affected by UMEs, TENET can still recover the master objects to the state of last grace period.

4.3.5 Recovery

(1) Recovering from non-UME crashes. This recovery includes recovering from a system crash or a memory safety violation. Upon restart, the recovery procedure is of two steps: (1) `CLog` replays, where all the entries in the `CLog` are replayed to set the master objects to a consistent state. This step is necessary to bring all the master objects to the latest checkpointed state. (2) Then all `OLog` entries are sorted based on their `commit-ts` and replayed sequentially in the exact sorted order. This will bring the master objects to the last committed state before the crash occurs. Note that, if the crash happens due to a memory safety violation, a developer should fix the bug to avoid repetitive non-UME crashes.

(2) Recovering from a UME crash. Upon restart, if TENET cannot open its NVM pools, it indicates a UME has occurred. The recovery steps depend on the victim pools' type.

UME in the master object pool. TENET identifies the corrupted physical offset using the `ndctl` tool [19] and then extracts the corresponding logical file offset. TENET brings the entire page where the corrupted offset belongs from the replica disk file. Then TENET allocates a new NVM page using `fallocate` and updates it using the disk replica. Finally, it deallocates the corrupted page and removes it from the operating system's bad block list. Once NVM is restored, TENET recovers similar to the non-UME crash as explained in **(1)**, *i.e.*, `CLog` replay followed by the `OLog` replay.

UME in a log pool. TENET does not need to access the disk to fix the bad page. Instead, it fixes the affected NVM page by allocating a new empty page. Then TENET uses the uncorrupted backup log pool to perform `CLog` and `OLog` replay. At the end of the recovery, it frees all the `CLog`s and `OLog`s, and new logs are allocated during the normal execution.

4.4 Implementation

TENET library is implemented in C and C++ which is ∼11K LoC. The core TENET library includes the TIMESTONE PTM (∼7K LoC), memory safety checks (∼1.5K LoC), and the NVM-SSD replication (∼2.5K LoC). We rigorously tested TENET with a carefully curated set of unit tests, functional tests, and integration tests along with the offline testing tools such as the Pmemcheck [132], Address sanitizer [231] to ensure correctness of our implementation.

4.5 Discussion

In this section, we discuss the key takeaways in TENET (§4.5.1) and the applicability of TENET's ideas on ARM architecture (§4.5.2). We also discuss the limitations and potential future research directions in §4.5.3.

4.5.1 Leveraging the Concurrency Guarantees of PTM

Enforcing low overhead spatial safety. Most PTMs perform out-of-place updates to enforce the Isolation property (ACID) [93, 115, 162, 177, 192, 255], to support concurrent read and write [93, 115, 162], and to enable write batching [93, 162, 255]. These PTMs have at least two separate domains: one in which new updates are made and buffered, and another that contains consistent data (*i.e.*, old updates) to which the new updates are eventually merged. TENET leverages this property to enforce a separate protection domain, such a design enables it to use light-weight techniques such as MPK and canaries to enforce spatial safety *without having to check every access.*

PTMs such as the libpmemobj [130] that perform in-place updates can be modified to perform out-of-place updates as done in Pangolin [272]. Although Pangolin uses microbuffering to perform out-of-place updates, it relies on expensive data checksum to enforce spatial safety *i.e.*,

checksum is calculated and verified every time the data is moved to and from the microbuffers. SafePM [82] relies on compiler instrumentation of loads and stores and hence it needs to perform spatial and temporal safety checks at every access resulting in a high performance overhead (§4.6.3).

Enforcing low overhead temporal safety. Almost all PTMs support a stronger Consistency (A**C**ID) guarantee such as linearizability or serializability. Such PTMs usually perform conflict checks (*i.e.*, read/write set validation) during the commit phase and the transactions are aborted if a read-write conflict is observed during the validation. In the context of temporal safety, this means that objects with live references in any on-going transaction will not be freed until those transactions finish. Unlike the prior PTM works, Tenet leverages this property to perform temporal safety checks only at the first dereference and avoids redundant checks during every pointer deference in a transaction. This is because, once an object is dereferenced, it can not be freed by concurrent transactions, a inherent guarantee provided by PTM.

4.5.2 Tenet's Ideas on ARM Architecture

ARM processors support memory domains [8], which is similar to Intel MPK except that the permission switch happens in the OS kernel. Moreover, ARM processors have been supporting virtual address (pointer) tagging (upper 12-16 bits) at the hardware level and it is shipped with the *top byte ignore (TBI)* feature [9, 29, 69]. Therefore, we believe that Tenet's ideas can be applied beyond x86 architectures.

4.5.3 Limitations and Future Work

Protecting against intra-object overflow. Protecting against intra-object overflow is a hard, open research problem. Even the state-of-the-art techniques, such as BOGO [273]

do not protect against intra-object overflow. We believe that protecting against intra-object overflow with reasonable performance overhead would require significant architectural changes and/or compiler-level instrumentations because of the fine granularity of protection [129, 261]. However, TENET protects the transactional metadata which are essential for correct execution and recovery from the intra-object overflow. We do this by placing an additional intra-object canary between the metadata section and the application data section in a DRAM object (not shown in the figures). This restricts the corruption to only the application data section of an object.

Protecting against the code outside the transaction. TENET already protects the NVM data from spatial safety violations due to the code outside the transaction by using MPK. However, it is possible to corrupt the DRAM objects outside the transaction and TENET may not detect such corruption, particularly the ones that do not overwrite the canaries. One way to protect the DRAM objects is to protect all the `TLog`s using the MPK and allow to switch permission only within the TENET library. However, as `TLog` is per-thread and there are only 16 MPKs available, we may need to employ MPK virtualization [215] to offer a more fine-grained protection.

Impact of shorter tags. In TENET, we use all the upper 16-bits to store the pointer tag; expansion of address space in the future will reduce the number of available bits thus making the tag range shorter. TENET allows to reuse of duplicate tags across different pointers, but if the bits are too few (e.g., only 4 bits are available), reusing tags may cause false negatives. In TENET, tag reuse becomes a problem, only if the reallocated pointer is assigned with the same tag (that it had before last free), which makes TENET 's temporal safety detection probabilistic. Reusing tags across different pointers or the same pointer with non-consecutive reallocations results in a deterministic detection. As the CPU vendors are extending hardware support for pointer tagging, we believe that expanding this idea to overcome bit limitations

(e.g., similar to x86 segmentation overcoming 64KB address limitation) will be an interesting future work.

4.6 Evaluation

We evaluate TENET by answering the following questions, (1) what are the performance overhead of TENET's memory safety and off-critical path disk replication techniques (§4.6.1)? (2) How does TENET perform in comparison with the other state-of-the-art memory safe PTMs (§4.6.3)? (3) What is the tail latency of TENET (§4.6.4)? (4) How does TENET fare in the bug detection, correction, and recovery stress tests (§4.6.5)?

Evaluation platform. We use a system with Intel Optane DC Persistent Memory (DCPMM). It has two sockets with Intel Xeon Gold 5218 CPU with 16 Physical cores, 256GB of NVM (2×128GB), 32 GB of DRAM (2×16GB) per socket, and 2×1TB M.2 SSDs (Samsung 970 EVO). We used GCC 11.2.1 with `-O3` flag to compile benchmarks and ran all our experiments on Linux kernel 5.16.12 with `io_uring` support.

Configuration. We preset the size of `TLog` and `OLog` to 8 MB and `CLog` to 32 MB, respectively. We also present the performance analysis for varying log size in §4.6.4. We use two SSDs for NVM replication *i.e.*, one SSD per socket. Throughout our evaluation, we present two versions of TENET: (1) **TENET-MS** – *which enforces only memory safety (i.e., no NVM/SSD replication)*, and (2) **TENET** – *which enforces both memory safety and NVM/SSD replication for fault tolerance.* For microbenchmarks, we initially warm up the data structures with 1 Million (M) keys followed by executing a mix of lookup, insert, update, and delete operations for 60 seconds as done in the prior PTM works [93, 115, 162, 219, 220, 221, 255]. For the real-world evaluation, we use the YCSB benchmark [91] to evaluate TENET's B+Tree based key-value store engine for 10M keys, we use 8 bytes integer keys and 100 bytes values with Zipfian distribution. We present the average performance of 10 runs, with an average

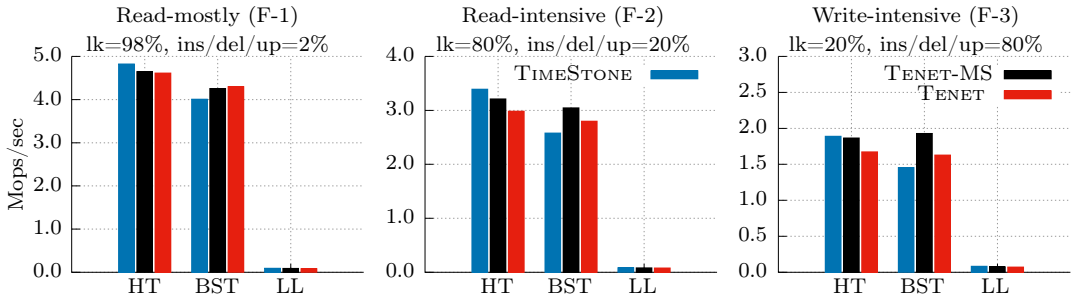

Figure 4.6: Performance comparison of TENET-MS and TENET against TIMESTONE for Hash Table (HT), Binary Search Tree (BST), and Linked List (LL) for 24 threads.

error rate of ±1.8%.

4.6.1 Performance Analysis of TENET

Figure 4.6 compares the performance of TENET-MS and TENET against the TIMESTONE for three different workloads with varying read/write ratios. Comparing TENET-MS and TENET with TIMESTONE will enable us to quantify the overheads due to memory safety and fault tolerance techniques.

4.6.1.1 TENET-MS vs TIMESTONE

For the read-dominated workloads, TENET-MS performs mostly on-par ($< 5\%$ overhead) or slightly better than the TIMESTONE. This is because reads in TENET-MS require only temporal safety checks and the overhead from spatial safety checks are negligible due to the lower write ratio. The low overhead temporal safety checks can be attributed to our in-place pointer tagging technique wherein it only requires one shifting operation for extracting the tag from the pointer and one compare operation for validating the extracted tag.

For write-intensive workload, TENET-MS performs on par with TIMESTONE; this shows that our canary based spatial safety checks incur only a minimal overhead. For BST, TIMESTONE suffers from high transaction aborts due to lock conflicts on parent nodes. Unlike the BST, hash table is inherently more concurrent and incurs lower aborts due to less lock conflicts.

Memory safety validation steps in TENET-MS reduce the aborts; our further analysis revealed that TIMESTONE incurs about 3.5× more aborts than TENET-MS for BST. Consequently, TENET-MS performs on par with TIMESTONE for hash table and slightly faster in case of a BST.

4.6.1.2 TENET vs TIMESTONE

In addition to memory safety, TENET guarantees fault tolerance by performing NVM/SSD replication. For read-mostly workloads, TENET performs on par with that of TIMESTONE and TENET-MS. Due to a lower write ratio, the number of log writes, and master object writes are less; consequently replication does not add any significant overhead. However, the replication overhead becomes evident as the write ratio increases from 20% to 80% and TENET performs up to 12.6% and 18% slower than the TENET-MS and TIMESTONE, respectively. As the master objects are inserted/deleted/updated frequently, the replica writes to SSD also increases. Therefore, grace period detection is relatively longer in TENET as the `gp-thread` has to wait for all the SSD writes to complete. A longer grace period detection increases traffic in the `TLog` as the log reclamation becomes slower. Overall, TENET adds a modest overhead ($< 18\%$) over TIMESTONE while enforcing memory safety and fault tolerance.

4.6.2 Real-world Workload Evaluation

We built a B+tree-based key-value store using TENET; we chose B+tree (fanout=64) to test and evaluate our array interface but any other data structures can also be used. Figure 4.7 compares the performance of TENET-MS and TENET key-value store against the TIMESTONE key-value store.

TENET-MS. TENET-MS is 17% slower than the TIMESTONE across all YCSB workloads. For data structures (that do not use an array), such as the hash table, every read to a hash

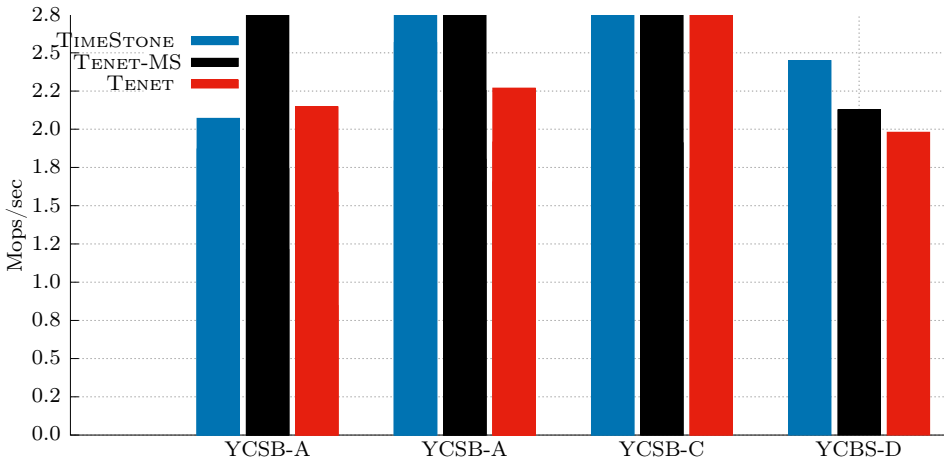

Figure 4.7: Performance comparison of TENET-MS and TENET against TIMESTONE for the B+tree key-value store with 24 threads.

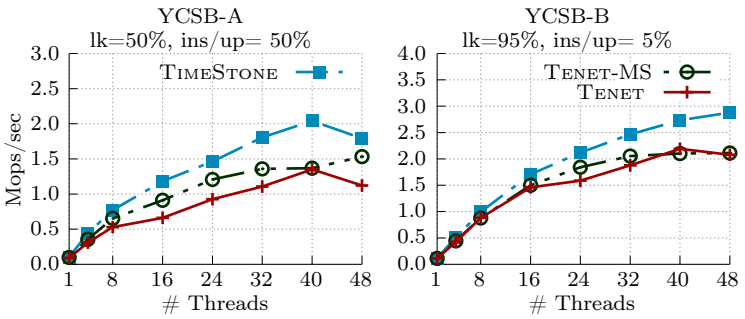

Figure 4.8: Scalability of TENET-MS and TENET for B+tree

node requires only one object dereference because each hash node is a master object. But for a B+tree, reading one leaf node requires a 2× fanout (2×64) number of dereferences as each array element (of the key-value array) is a master object. Although TimeStone incurs the same number of object dereference, the additional temporal safety checks during the object dereferencing in Tenet-MS causes a 17% slowdown.

Tenet. For write-intensive YCSB-A, Tenet performs 41% slower than TimeStone. This is because of lower chances of write coalescing in the `TLog` and `CLog`. As the writes happen at the array element level, the chances of an array index being repeatedly written to is less. This is the worst-case scenario for TimeStone as it relies on maximizing write coalescing on DRAM objects to reduce NVM writes. Lower write-coalescing causes frequent checkpoints (from `TLog`) on `CLog` and frequent checkpoint writebacks (from `CLog`) to the NVM object. TimeStone just performs frequent writebacks to the NVM object; for Tenet, increase in the number of writebacks also increases the SSD writes due to replication. This trend is corroborated by the performance of Tenet for read-intensive YCSB workloads (B, C, and D), where it exhibits only a 21% slowdown against TimeStone. This is almost half of the slowdown experienced for the YCSB-A workload (41%) as the number of SSD writes are lower in read-intensive workloads. In a nutshell, Tenet guarantees memory safety for arrays (Tenet-MS) with a modest 17% overhead and providing fault tolerance adds an additional 24% overhead due to the reduced write coalescing in TimeStone.

4.6.3 Comparison with Other PTMs

Table 4.1 compares the protection scopes of PTMs; Tenet is the only PTM to offer full memory safety and cost-efficient fault tolerance. We have discussed the limitations in the protection scope of prior works in §4.1.3. Moreover, Tenet incurs a relatively minimal performance overhead as compared to SafePM (Figure 4.9) and Pangolin which incurs up

PTM	Spatial Safety	Temporal Safety	UME	NVM Cost
Libpmemobj[130]	No	No	Yes	High
TIMESTONE [162]	No	No	No	None
SafePM[82]	Yes	Yes	No	Moderate
Pangolin[272]	Partial	No	Yes	Moderate
TENET-MS	Yes	Yes	No	None
TENET	**Yes**	**Yes**	**Yes**	**Low**

Table 4.1: Comparison of TENET against other PTMs.

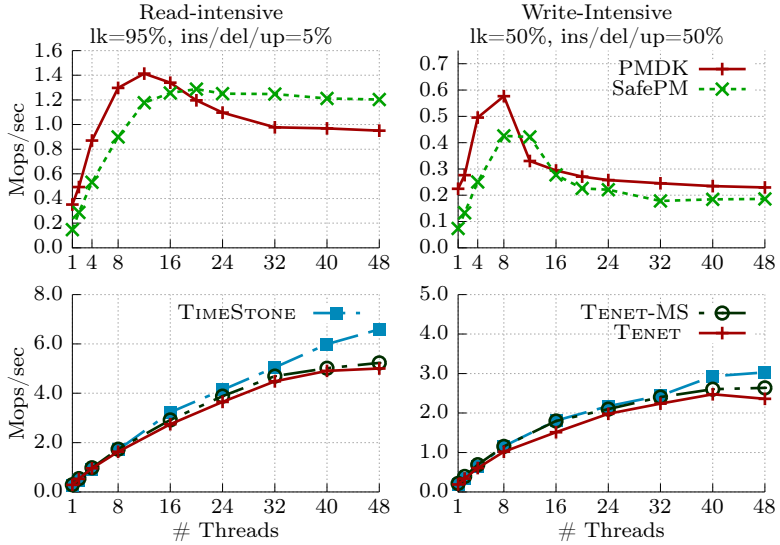

Figure 4.9: TENET-MS vs SafePM: performance overhead study with hash table for read-intensive and write-intensive workloads.

to 60% and 67% overhead over the `libpmemobj`.[1] To ensure fairness, we compare SafePM and TENET-MS on basis of performance overhead incurred over their respective baseline PTM. Note that SafePM does not guarantee fault tolerance against the UMEs, so we use only TENET-MS for comparison.

As shown in Figure 4.9, SafePM performs up to 67% slower than the `libpmemobj` across both the workloads. When the `libpmemobj`'s performance saturates after 16 threads, SafePM performs on-par; this is because the high contention overhead in the `libpmemobj` amortizes the memory safety overhead in SafePM. SafePMs' overheads come from: (1) additional undo logging to guarantee crash consistency for the memory safety metadata. Note that this undo

[1]Directly referenced from the paper as Pangolin is not open-sourced.

logging is in addition to the ones performed by the `libpmemobj` transaction, (2) the memory safety metadata must be accessed for every read and write which further slows down the performance.

Unlike the SafePM, TENET-MS guarantees memory safety with a modest 5%-8% performance overhead; because, (1) it does not require additional crash consistency for memory safety metadata as the pointer tags are embedded in the objects, and (2) memory safety checks are performed only once per transaction (on-commit and on-first-dereference).

4.6.4 Other Evaluations and Analysis

Scalability analysis. Figure 4.8 and Figure 4.9 shows the read and write scalability of TENET-MS and TENET for hash table and B+tree, respectively. Both TENET-MS and TENET show good read and write scalability for B+tree and hash table. The performance difference across thread counts are consistent with what is observed for 24 threads in Figure 4.6 and Figure 4.7. For read-intensive workloads, TENET-MS and TENET show less than 5% performance slowdown for a hash table and a 17% (TENET-MS) and 24% (TENET) slowdown for a B+tree. For a write-intensive hash table, TENET-MS and TENET exhibit a 5% and 18% slowdown respectively, while for B+tree, TENET-MS and TENET exhibit a 17% and 44% slowdown. Overall, both TENET-MS and TENET scales on-par with TIMESTONE; this shows that the TENET*'s memory safety and fault tolerance techniques does not impede the original scalability of* TIMESTONE.

Storage cost analysis. With TENET, the DRAM space usage is bounded by the size of `TLog` (8MB). TENET stores the replica logs in the NVM and this is bounded by the size of `OLog` and `CLog`. TENET replicates the application data structure to the SSD; given the \$/GB of SSD (\$0.15) and the NVM (\$10) [16, 54], TENET saves ∼60× on storage cost when replicating the entire NVM space (512GB) to the SSD as opposed replicating to the NVM.

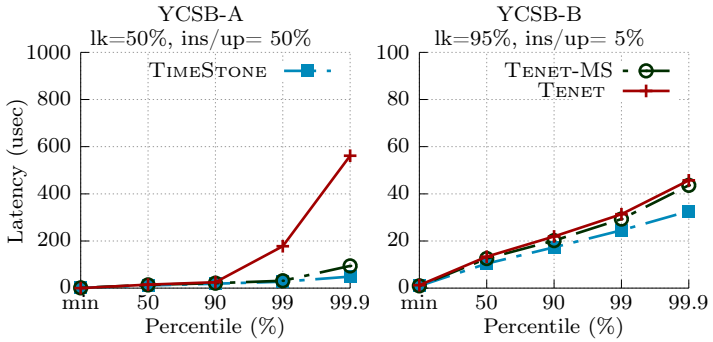

Figure 4.10: Tail latency comparison of TENET-MS and TENET against TIMESTONE for B+tree with 24 threads.

In addition to the cost benefits, TENET can recover from multiple UMEs spanning across multiple pages while Pangolin can recover only from a single page is corruption.

Tail latency. Figure 4.10 shows the tail latency of TENET-MS and TENET compared against the TIMESTONE. As done in prior works [156, 173], we sample 10% of operations so that the tail latency calculation does not overshadow the performance. TENET-MS performs on-par with TIMESTONE, which shows the efficacy of our memory safety techniques. However, for write-intensive YCSB-A, TENET's tail latency spikes up at the 99th and 99.9th percentile. This is because of the additional writes incurred while performing replication to the NVM/SSD for fault tolerance. For read-intensive workload, TENET's tail latency is almost on par with TIMESTONE as lower ratio reduces the number of SSD writes. TENET-MS shows similar tail latency to that of the TIMESTONE across workloads as it does not perform replication. We believe our fault tolerance design can be further optimized for tail latency by making log writes asynchronously, which would be an interesting future work.

Log size sensitivity. To study the impact of log size on the performance, we present the relative performance of TENET for varying log sizes using a concurrent hash table with 1 and 24 threads (Figure 4.11). We show the performance only for write-intensive workloads as read-intensive workloads are less sensitive to the log size. The X-axis represents the log size, and the Y-axis represents the relative performance normalized to the default log size

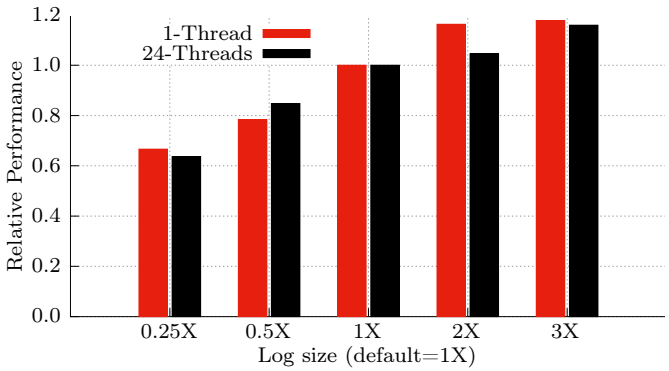

Figure 4.11: Performance sensitivity of TENET for varying log sizes.

used in all the previous evaluations. TENET's performance increases up to 21% with the increasing log size. As the log size is decreased, the performance drops to 38%. As the log size increases, the writers spend less time reclaiming log space and hence better performance. Alternatively, for smaller log sizes, the writers spend more time reclaiming log space. TENET requires all SSD writes in a grace period window to be persisted before reclaiming the log space, further increasing the pressure on the writers. So we observe a larger performance drop (38%) for a smaller log size and relatively a smaller performance gain (21%) when the log size is increased. We confirmed that this behavior is consistent across different thread counts and data structures.

4.6.5 Error Detection and Correction

Spatial safety test. Our test cases select transactions at random to intentionally cause *buffer overrun* bugs on a B+tree leaf nodes' value pointer (`p_val`) and to access the key array (in a B+tree node) *out-of-bounds*. For the buffer overflow bug, the erroneous transactions execute a `memcpy` on the `p_val` for 1KB where the `p_val` pointer is of size 100 bytes. We also tested intra-array overflow with a smaller size of 128 bytes. For out-of-bound access, the erroneous transactions access the key array at index *96*, which is beyond the original fanout (64). For all test cases, TENET detected spatial safety violations in the commit phase and aborted the transactions, returning an exception to the B+tree code. In our 200 random

tests, TENET detected spatial safety violations 100% of the time.

Temporal safety test. We modified the delete function in our open-chaining hash table benchmark to free the target node and *not update the previous nodes' next pointer (`p_next`)*. A randomly chosen transaction executes the buggy delete logic and spawns read transactions to access the dangling `p_next`. TENET detected the dangling pointer access during the object dereferencing phase and returned an exception to the application. Further, to test the case where a free-ed address may be reallocated again, we kept allocating a new hash node until the free-ed NVM address was reallocated. Our test case then waits for a transaction to access the *dangling `p_next` (reallocated)*. We repeated both the temporal safety tests 200 times, and TENET detected dangling pointer access and returned an exception to the application.

UME Test. We used the `ndctl` utility tool (*`ndctl` inject-error*) for injecting a UME at a specified offset [19]. While running the benchmark, we first injected a UME in the log pool, particularly on a randomly chosen `CLog`. TENET's `SIGBUS` handler received the OS notification and terminated the program gracefully. Upon restart, TENET rightly identified the corrupted log pool and successfully recovered using the replica log pool. We also injected UME in one of the master object pools and observed that TENET restored the NVM status successfully using the SSD replica. Both these tests were repeated multiple times and TENET successfully recovered the hash table without losing any data. The recovery time for TENET and TIMESTONE are similar, bounded by `OLog` and `CLog` size (not shown due to space constraints). The SSD access is performed in the background using `io_uring` and the cost is relatively small. Our future work will develop techniques to accelerate recovery.

4.7 Chapter Summary

In this chapter, we discussed TENET. TENET enforces DRAM/NVM memory domain separation using MPK to prevent NVM writes out of TENET library. Additionally, TENET

uses canary values and in-place pointer tagging to guarantee on-commit spatial safety and on-first-dereference temporal safety. Further, Tenet proposes off-critical path NVM/SSD data replication to guarantee a performance and cost-efficient fault tolerance for the NVM data against the UMEs. Our evaluations showed the performance efficiency of Tenet's techniques along with a thorough analysis on scalability, storage cost, and tail latency. Overall, Tenet provides enhanced NVM data protection at a modest performance and storage cost as compared to the other state-of-the-art PTMs.

Chapter 5

Framework for Porting Volatile Indexes to NVM

Indexes are a fundamental building block in many storage systems, and it is critical to achieving high performance and reliability [157, 187]. With the advent of Non-Volatile Main Memory (NVM), such as Intel Optane DC Persistent Memory [74, 196], there have been a numerous number of research efforts targeted towards developing NVM-optimized indexes [46, 77, 87, 127, 146, 167, 176, 179, 183, 204, 206, 212, 213, 247, 250, 258, 263, 278]. Such index designs mainly focus on reducing the crash consistency overhead by primarily relying on index-specific optimizations to improve the overall performance.

However, maturing and hardening an index requires a lot of time and effort. For example, recently proposed NVM indexes have critical limitations, such as (1) weaker consistency guarantee [118, 169], (2) not handling memory leaks in the wake of a crash [77, 127, 167, 204, 278], (3) limited concurrent access [87, 130, 167], and (4) not supporting variable-length keys [87, 127, 213, 263]. Most of these missing features are critical in real-world systems and it delimits the adoption of these indexes to the real-world applications without further maturing.

Alternatively, there are decades of research on in-memory DRAM indexes [96, 170, 184, 187,

252] which are well optimized, engineered and used in many real-world applications such as in-memory key-value stores and databases [99, 106, 151, 240]. If we can leverage these in-memory DRAM indexes for NVM, it not only gives a large pool of well-engineered indexes but it will also pave way for the real-world applications built on top of these indexes to use and adopt NVM. The challenges in building an NVM-optimized index has lead to a spike in the interest to port legacy DRAM applications, particularly in-memory key-value stores [1, 25, 44, 45, 50, 51, 185, 191]. However, prior works [45, 185, 191] report that manual porting is complex, and error-prone requiring a lot of engineering effort. So we believe that *it is important to provide a systematic way to convert DRAM-based indexes for the NVM.*

A few recent studies have proposed techniques [193, 257] or guidelines [97, 109, 118, 169] to convert volatile indexes to NVM. Unfortunately, these techniques have some critical limitations such as (1) limited applicability due to restrictions on the concurrency control (*e.g.*, supporting only lock-free indexes) [97, 109, 118, 169, 193], (2) supporting a weaker consistency guarantee (*i.e.*, buffered durable linearizability) [118, 169, 257], (3) requiring in-depth index-specific knowledge [97, 109, 118, 169], (4) high performance and storage overhead due to their crash consistency mechanism [118, 193, 257], and (5) not addressing persistent memory leaks [97, 109, 118, 169, 257]. We further discuss the limitations of these techniques in §5.1.

To address these problems, we propose Tips– an index-agnostic framework to systematically make a volatile DRAM index persistent, while supporting (1) wider applicability by not placing any restrictions on the concurrency model of an index, (2) strong consistency model (*i.e.*, durable linearizability), (3) index-agnostic conversion without requiring in-depth knowledge on a DRAM index, (4) low overhead crash consistency mechanism, (5) safe persistent memory management (*e.g.*, no persistent memory leak), and in addition to achieving (6) high performance and good multicore scalability. Tips makes the following contributions:

- We propose a novel *DRAM-NVM data tiering approach* and its concurrency model called the *tiered concurrency* to achieve good performance, scalability, and applicability.
- We propose a low overhead hybrid logging technique called the *UNO logging* to guarantee crash consistency, to prevent memory leaks and to guarantee durable linearizability.
- We propose the TIPS *framework* based on these approaches. TIPS provides index-agnostic conversion and does not place any restrictions on the concurrency model or require in-depth knowledge of the volatile index being converted.
- We converted seven volatile indexes with different concurrency models, a real-world key-value store Redis and evaluated them using YCSB [91]. Our evaluation shows that TIPS outperforms the state-of-the-art index conversion techniques and the NVM-optimized indexes by 3-10× across the different YCSB workloads.

5.1 Existing Index Conversion Techniques

We discuss the limitations of existing conversion techniques and their implications below:

(1) Restrictions on Concurrency Control. All prior techniques have limited applicability; *i.e.*, they are designed to support only a specific concurrency model. For example, NVTraverse [109] and link-and-persist [97] are designed for lock-free indexes while MOD [118] is designed for purely functional data structures. PRONTO [193] is applicable only to the globally blocking indexes (*e.g.*, protected by a global mutex) while RECIPE [169] guidelines apply only to the indexes that support lock-free or fine-grained locking.

(2) Supporting Weak Consistency. Another limitation is that most techniques [118, 169, 257] support only a weaker consistency; A linearizable DRAM index converted using these techniques will support only a weaker consistency model, Buffered Durable Linearizability (BDL) [135]. Linearizability has been the standard consistency model in DRAM indexes for almost three decades [121]. Its NVM counterpart is Durable Linearizability (DL) [135], and

it plays a critical role in ensuring the correctness and consistency of the NVM index and data. Indexes with a BDL guarantee can experience a large amount of data loss in the wake of a crash. Moreover, it increases the programming complexity as the developers are burdened with reasoning about the data consistency. This makes the conversion process complex and more error-prone; for instance, many fundamental and non-trivial crash consistency bugs have been found in the RECIPE indexes [109, 110].

(3) Requiring In-depth Knowledge. Many techniques [97, 109, 118, 169] require in-depth knowledge of the volatile index and expertise in NVM programming to apply their guidelines correctly. Such efforts are non-trivial as even missing a single `sfence` or a `clwb` may render an index irrecoverable.

(4) High Storage and Performance Overhead. Many techniques suffer from high storage and performance overhead [118, 193, 257] mainly due to their crash consistency mechanism. For example, PMThreads [257] requires two full replicas (one each on DRAM and NVM) of the original data. MOD [118] uses Copy-on-Write (CoW) to guarantee crash consistency. Such a high storage overhead might be acceptable for primitive data structures (*e.g.*, stack, vector). However, it can be detrimental for indexes and real-world key-value stores designed to handle a large volume of data. Although PRONTO [193] uses operational logging for crash consistency and employs a dedicated background thread for every writer to perform the logging, it still incurs a high overhead due to the synchronous waiting between the writers and background threads. We further empirically analyze these overheads in §6.2.

(5) Persistent Memory Leaks. Another critical but a largely understated problem is the persistent memory leaks. While the prior conversion techniques and the NVM-optimized indexes focus on providing crash consistency, they completely ignore the memory leak problem. Although NVM allocators can guarantee failure atomicity for allocation/free even the mature

allocator such as the PMDK [131] does not provide an efficient solution to identify and fix the memory leak [98]. Hence it is critical to address this within the TIPS framework.

5.2 Overview of TIPS

We first discuss our design goals followed by the design overview. We assume that an index being converted has key-value store style operations such as insert, delete, update, lookup, and scan. Throughout this chapter, we address insert, delete and update as writes, and lookup and scan as reads. The term *plugged-in index* denotes a user-defined volatile index on NVM that is plugged into TIPS framework. We assume that locks can be reinitialized after crash recovery.

5.2.1 Design Goals

G_1 **Support Various Concurrency Models.** We aim not to place restrictions on the concurrency model of the volatile indexes. With TIPS, we support the conversion of indexes that use a global lock (*e.g.*, Mutex), lock-free (*e.g.*, CAS), and fine-grained locking (*e.g.*, ROWEX [170]).

G_2 **Support Durable Linearizability (DL).** The challenge to guarantee DL in TIPS is to achieve it without compromising the performance, scalability, and index-agnostic conversion.

G_3 **Support Index-agnostic Conversion.** We aim to support near black-box, index-agnostic conversion to circumvent the need for in-depth knowledge on the volatile index and NVM programming. This will make the conversion process simple, intuitive, and less error-prone. We aim to achieve this by internally handling the complications of NVM programming such as guaranteeing crash consistency and preventing persistent memory leak within TIPS and also providing an uniform programming interface to assist the conversion.

G_4 **Design a Low Overhead Crash Consistency mechanism.** Tips can not rely on index-specific optimizations for crash consistency to support an index-agnostic conversion. The crash consistency mechanism should incur low overhead to achieve high performance and scalability. A crash consistency mechanism also should prevent persistent memory leaks by reclaiming the unreachable objects upon recovery.

G_5 **Achieve High Performance and Scalability.** Ideally, we aim to perform and scale on par with NVM-optimized indexes. Also, we strive to retain the original characteristics of the plugged-in index; for example, if the volatile index is designed for cacheline efficiency or optimized for scalability, we aim to retain and leverage such characteristics to improve the overall performance and scalability of a Tips index.

5.2.2 Design Overview

5.2.2.1 High-Level Idea of Tips

Figure 5.1 presents the Tips architecture and illustrates how a write operation is handled in the Tips framework. Tips frontend consists of a hash table on DRAM (DRAM-cache) and an operational log (`OLog`) on NVM. Tips backend consists of the plugged-in index – the user-provided volatile index – a background thread (persist-thread), UNDO log (ULog), and MEM log (MLog). When a write is issued (❶), Tips first commits it to the per-thread `OLog` for guaranteeing durability (❷ - durability point) and then inserts a new key-value pair entry (or a tombstone for deletion) in the DRAM-cache to make the write visible (❸ - linearization point). Then the persist-thread in the Tips backend (❹) replays the same write in the background to update the plugged-in index. To guarantee crash consistent update to the plugged-in index, Tips uses ULog to store the unmodified data for recovery. Once the plugged-in index is updated, the corresponding key-value entry in the DRAM-cache will be reclaimed later.

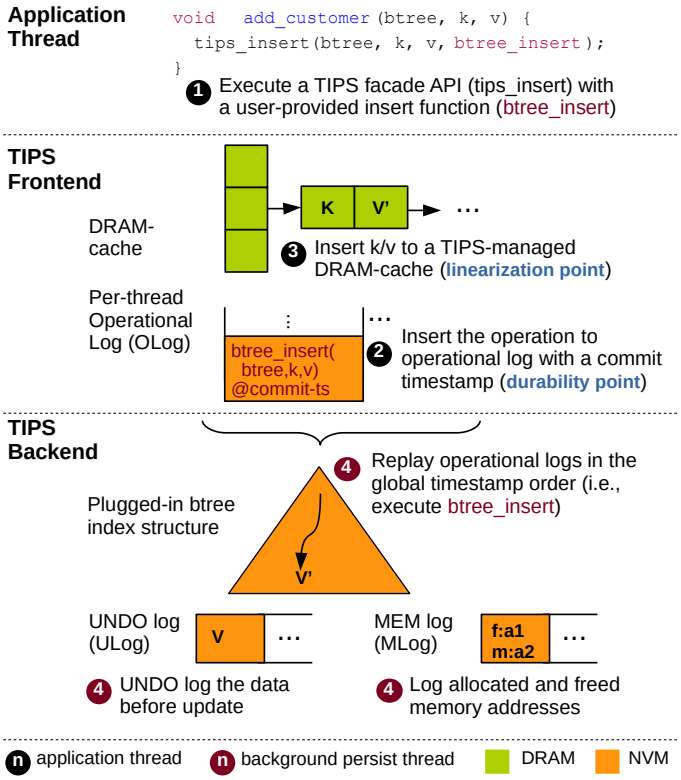

Figure 5.1: Illustrative example of inserting a key-value pair in TIPS.

Alternatively, a lookup operation first looks for the target key in DRAM-cache and it goes to the plugged-in index only if the target key is not present in DRAM-cache. With this high-level idea, we next present the design overview of TIPS and also explain how it contributes towards achieving our design goals discussed in §5.2.1.

5.2.2.2 DRAM-NVM Tiering (G_1, G_2, G_3, G_5)

At its core, TIPS adopts a novel DRAM-NVM data tiering approach. As illustrated in Figure 5.1, two critical components that enable the DRAM-NVM tiering are DRAM-cache in the TIPS frontend and plugged-in index in the TIPS backend. Although prior NVM-optimized B+tree index designs [176, 213] and conversion techniques [193, 257] have proposed to use both NVM and DRAM, our approach is fundamentally different. For example, PRONTO [193] builds the index on DRAM and logs the index operations on the NVM to guarantee durability.

Such a tiering design limits PRONTO indexes from scaling beyond the DRAM capacity. Instead, in Tips, we propose tiering of the data (*i.e.*, key-value pairs) while keeping the plugged-in index intact on the NVM.

Benefits. (1) Tiering the data between DRAM-NVM makes our approach *generically applicable* to any index. This enables all the Tips indexes to take advantage of the faster DRAM. (2) Unlike PRONTO, Tips can achieve a better capacity scaling by keeping the plugged-in index on the NVM. (3) The writes are made visible through DRAM-cache (❸ in Figure 5.1); this enables Tips to guarantee DL agnostic of the plugged-in index. (4) On top of DRAM-NVM tiering, we build the *plug-in programming model* to enable index-agnostic conversion (§5.2.2.6). (5) Tiering the data enables a new concurrency model called the *tiered concurrency*, which is key to achieving high performance and scalability.

5.2.2.3 Tiered Concurrency for Scaling Frontend (G_1, G_5)

Having Tips frontend and backend enables two different levels of concurrency: (1) concurrency model of the DRAM-cache and (2) concurrency model of the plugged-in index. We call this a *tiered concurrency model*. Tips frontend allows parallel readers and parallel disjoint writers as DRAM-cache is a concurrent hash table and `OLog` is per-thread. In the critical path, all requests succeeding in the Tips frontend (all writes and read hits) will follow the concurrency model of DRAM-cache, and the operations that go to the Tips backend (read misses and scan) will follow the concurrency model of a plugged-in index. Also, the background writes (❹ in Figure 5.1) to the plugged-in index is done off the critical path adhering to concurrency model of the plugged-in index.

In a nutshell, to achieve write scalability Tips restricts writes to the faster frontend (DRAM-cache and per-thread `OLog`) and for read scalability, it relies on both the DRAM-cache and the concurrency model of the plugged-in index. Range scans always go to the plugged-in

```
1 /* TIPS facade API to use a TIPS-enabled index */
2 bool tips_insert(void *ds, key_t k, value_t v, fn_insert_t *f);
3 bool tips_update(void *ds, key_t k, value_t v, fn_update_t *f);
4 bool tips_delete(void *ds, key_t k, fn_delete_t *f);
5 value_t *tips_lookup(void *ds, key_t k, fn_lookup_t *f);
6 value_t *tips_scan(void *ds, key_t start_key, int range, fn_scan_t *f);
```

```
7 /* TIPS plug-in API to implement a TIPS-enabled index */
8 bool  tips_ulog_add(void *addr, size_t size);
9 void* tips_alloc(size_t size);
10 void  tips_free(void *addr);
```

Figure 5.2: TIPS *facade APIs* to access a TIPS-enabled index, and *plug-in APIs* for plugging-in a volatile index to TIPS.

index as it requires full key-value data (see details in §5.3.1.6). Relying on the plugged-in index for read/scan helps TIPS to reduce the DRAM footprint as it does not need to cache the entire dataset in DRAM-cache. Instead, it can reclaim the keys once the plugged-in index is updated.

Benefits. (1) Even if the plugged-in index supports only blocking concurrency (*e.g.*, mutex), it can still leverage the DRAM-cache to process all the writes and read hits concurrently to achieve good performance. (2) Unlike the previous techniques [97, 109, 169, 193], it does not place any restrictions on the concurrency model of the volatile indexes and hence any volatile index can be plugged-in to the TIPS framework.

5.2.2.4 Adaptive Scaling for Backend Scalability (G_5)

The backend writes are slower than the frontend as it writes to the NVM. This can cause an imbalance in the system; to prevent this, we propose adaptive scaling of background writers (workers) for scaling the TIPS backend. With adaptive scaling, TIPS continuously monitors both the frontend and backend write throughput, and when there is an imbalance it scales up the worker count to catch up with the faster frontend and vice versa. While scaling up, TIPS identifies the best worker count based on the write scalability of the plugged-in index

and it caps the scale-up at that count to get maximum performance. Also, the real-world workloads are rarely 100% writes, so the time between writes and the adaptive scaling can help the backend writers to catch up with a faster frontend.

Benefits. (1) It effectively utilizes the write concurrency of the plugged-in index. (2) Unlike PRONTO [193] which demands a dedicated worker for every foreground writer, Tips can dynamically adjust the worker count based on nature of the workload and the plugged-in index.

5.2.2.5 UNO Logging for Crash Consistency (G_4)

To achieve a low overhead index-agnostic crash consistency, we propose the *UNO logging protocol*, which makes a hybrid and synergistic use of traditional **UN**DO logging and **O**perational logging. In Tips, we use operational logging (`OLog`) to guarantee immediate durability and UNDO logging (ULog) to ensure failure-atomicity while updating the plugged-in index. The unique aspect of *UNO* logging lies in how we leverage the `OLog` to reduce the notorious UNDO logging overhead and also reduce the number of `p-barrier` (`clwb`s followed by `sfence`) by batching the recurring updates to the same cache line and consequently achieve a low overhead crash consistency. Furthermore, MEM Log (MLog) internally logs all the allocated and freed addresses and Tips uses this information to identify and free all the unreachable memory upon recovery to prevent memory leaks and defers the actual memory free operations until `OLog` entries are consumed to prevent double-free bugs.

Benefits. (1) Using `OLog` requires only *two* *`p-barriers`* in the critical path for all write operations; *one* `p-barrier` to persist a `OLog` record and *one* more to persist the tail pointer (§5.3.1.5). This makes the durability guarantee cheap and consequently a better performance (§5.6.2, §5.6.4). (2) Tips alleviates the UNDO logging overhead by leveraging the `OLog` information to selectively log only the memory required for correct recovery besides the merit

```
1  void hash_insert(hash_t *hash, key_t key, val_t value) {
2      node_t **pprev_next, *node, *new_node;
3      int bucket_idx;
4      pthread_rwlock_wrlock(&hash->lock);
5      // Find a node in a collision list
6      // Case 1: update an existing key
7      if (node->key == key) {
8          // Before modifying the value, backup the old value
9  +       tips_ulog_add(&node->value, sizeof(node->value));
10         node->value = value; // then update the value
11         goto unlock_out;
12     }
13     // Case 2: add a new key
14     // Allocate a new node using tips_alloc
15 +   new_node = tips_alloc(sizeof(*new_node));
16     new_node->key   = key; new_node->value = value;
17     new_node->next  = node;
18     // Backup the prev node before modifying it
19 +   tips_ulog_add(pprev_next, sizeof(*pprev_next));
20     *pprev_next = new_node; // then update then the node
21 unlock_out:
22     pthread_rwlock_unlock(&hash->lock);
23 }
```

Figure 5.3: Code snippet of a TIPS-enabled hash table insert. Only three lines are modified in the original code; Lines 9, 19 for UNDO logging and Line 15 for persistent memory allocation.

that UNDO logging in TIPS is performed by the background workers (§5.3.3.2). (3) With MLog TIPS handles the persistent memory leaks with its framework instead of delegating it to the users.

5.2.2.6 Plug-In Programming Model for Index-agnostic Conversion (G_2)

The plug-in programming model provides two sets of APIs as shown in Figure 6.2: (1) plug-in APIs to plug-in a volatile index to the TIPS framework and (2) facade APIs to access the plugged-in index. The facade APIs internally manage the `OLog` and DRAM-cache without requiring any user intervention (steps ❶, ❷, ❸ in Figure 5.1). With the facade APIs, the plugged-in index implementation should be passed as a function pointer `f` which is used by TIPS to update the plugged-in index. To guarantee crash consistent updates to the plugged-in index, the developers must modify their index implementation using TIPS plug-in APIs. The modifications are simple, as shown in Figure 5.3: (1) replacing the volatile memory

allocation (and free) with `tips_alloc` (and `tips_free`) and (2) adding `tips_ulog_add` before modifying/writing to an NVM address. Tips will execute this modified code (*e.g.*, *hash_insert* in Figure 5.3) during the background reply.

By replacing the *malloc* with `tips_alloc`, the plugged-in index will now be allocated on the NVM, and `tips_alloc` will also capture all the newly allocated address in the MLog to prevent persistent memory leaks. The developer added `tips_ulog_add` would ensure crash consistency while updating the plugged-in index, and Tips internally optimizes the UNDO logging (§5.3.3.2) for better write coalescing and performance. Moreover, a developer is not required insert `p-barriers` to the volatile codebase manually. Instead, they just need to annotate the stores to NVM with `tips_ulog_add`. Thus, LoC changes in the plugged-in indexes are minimal, as shown in Table 5.2. While the developers still have to add the `tips_ulog_add` maually, they need not handle the persistence/visibility order as in the case of manually inserting `p-barriers`, and this makes the conversion easy and less error-prone. Note that updates to the newly allocated addresses (in Memlog) and existing addresses (in ULog) are batched and persisted internally by TIPS before reclaiming the respective logs. More details on this in §5.3.3.

5.3 Design of Tips

We first describe the Tips frontend design in §5.3.1, followed by the Tips backend design in §5.3.2.

5.3.1 TIPS Frontend Design

5.3.1.1 DRAM-cache

The DRAM-cache is a concurrent open chaining hash table. Concurrent writers working on the different buckets are allowed to proceed in parallel while the ones working on the same bucket are synchronized using a spinlock. Readers do not need any synchronization (*i.e.*, lock-free reads) as all writes to DRAM-cache are performed via a single atomic store. DRAM-cache also employs a RCU-style epoch based reclamation scheme for garbage collection. We choose open chaining hash table to guarantee $O(1)$ writes and avoid expensive rehashing in the critical path. We discuss the configurations for chain length and bucket size in §5.5 and §6.2.

5.3.1.2 Handling Write Operations

All write operations are first committed to the `OLog` to guarantee durability, and then a new entry is created and added to the DRAM-cache to make the writes visible. For delete, the entry also carries a tombstone mark for the readers to identify that it has been deleted logically. To make writes faster, the entries are always added at the head of a bucket. We explain how TIPS guarantees Durable Linearizability (DL) in §5.4.

5.3.1.3 Handling Lookup Operation

Readers first traverse the DRAM-cache bucket looking for the target key. As the collision chain is sorted by the arrival order of write requests, the readers can stop at the first match instead of traversing the entire chain. If the key is not present, then TIPS internally redirects the readers to the plugged-in index using the function pointer provided in the `tips_lookup` call. Scan operations require traversing the plugged-in index and the `OLog` to return a consistent result. We further describe it in §5.3.1.6 after introducing the `OLog` design.

5.3.1.4 Safe Reclamation

Since all writes happen in the DRAM-cache, the collision chain can proliferate and result in a high chain traversal overhead. To address this, we employ a background garbage collector thread called the `gc-thread`. When the chain length of a bucket exceeds the preset threshold, the `gc-thread` is triggered, which then visits the respective bucket and safely reclaims the entries. To ensure safe reclamation of entries *i.e.*, without impacting the concurrent readers, Tips employs an epoch-based reclamation scheme, which is widely used in lock-free and RCU-based data structures [108, 119, 189, 253]. Tips manages two types of epochs: (1) local epoch, which is the global epoch value when readers enter the critical section, and (2) global epoch, which is advanced when all active readers in the current epoch exit the critical section.

The `gc-thread` first logically reclaims entries by unlinking them from the collision chain and storing them in the free-list. The reclaimed entry then becomes invisible to new readers. Note that the `gc-thread` logically reclaims the entries that are successfully replayed, so upon a read miss, readers can still retrieve the reclaimed entries from the plugged-in index. To determine if there are any outstanding readers on the logically reclaimed entries, the `gc-thread` checks the local epoch of all the active readers. If the local epoch is the same as the global epoch, *i.e.*, no outstanding readers from the previous epoch exist; then the `gc-thread` physically frees the entries in the free-list that are logically reclaimed two epochs ago. Note that the `gc-thread` atomically modifies the collision chain, so the readers and writers are free to enter the critical section without waiting for reclamation to finish.

5.3.1.5 Operational Log (`OLog`)

`OLog` guarantees durability to the write operations executed on the DRAM-cache. An `OLog` record consists of the operation type (insert, delete or update), the respective function pointer to the plugged-in index logic, key, value, and a global timestamp (`commit-ts`) to denote the

commit order of an operation. `OLog` is a circular buffer where new entries are always added at the tail. The atomicity for an `OLog` write is guaranteed by its tail pointer update. Once an `OLog` record is written and persisted, the tail pointer is atomically updated to point to the new record, followed by persisting the tail pointer.

5.3.1.6 Handling the Scan Operations

Algorithm. The scan operation in TIPS is always directed to the plugged-in index, as it requires a full range of keys and values. However, the plugged-in index is not guaranteed to be up-to-date since the persist-thread might still be propagating some of the updates that might potentially fall within the scan range. So, after scanning the plugged-in index, TIPS traverses the `OLog`s looking for any potential keys that might fall within the scan range. Upon finding any, the scan buffer is adjusted to incorporate not yet propagated operations.

Traversing `OLog`. To minimize the `OLog` traversal overhead, we leverage the commit timestamp of each `OLog` record (`commit-ts`) and the timestamp when a scan operation starts (`scan-ts`). While traversing the `OLog`, the scan thread compares its `scan-ts` with the `commit-ts`. If the `commit-ts` $\leq$ `scan-ts`, then the scan thread reads the key information in the `OLog` record and checks if it falls within the scan range. It is safe to ignore `OLog` records with `commit-ts` $>$ `scan-ts` because these are in the future with respect to the scan thread, so it can stop traversing the `OLog`. Refer to §5.4 for correctness. Traversing the `OLog` is fast and adds only a negligible overhead for three reasons: (1) We partially traverse not-yet-propagated `OLog` entries stopping at the first future entry. (2) As the persist-thread continuously propagates the updates, there will not be much backlog in the `OLog`. (3) Finally, the scanning of `OLog` is a sequential read operation on the NVM, which is almost as fast as reading from the DRAM [262]. We verify this emperically in §5.6.3.

5.3.2 Tips Backend Design

The primary role of the Tips backend is to combine and replay the per-thread `OLog` on the plugged-in index. To guarantee correct replay order, the persist-thread combines the per-thread `OLog` entries, and then it spawns the required number of background workers, which replays the combined entries to the plugged-in index. The persist-thread decides the required worker count on-fly using the adaptive scaling algorithm. The following subsections explain each of these steps in detail.

5.3.2.1 Adaptive Scaling of Background Workers

Tips automatically adjusts the number of workers at every epoch based on the write scalability of the plugged-in index and the nature of the workload. One epoch (e) is defined as one iteration of combining and replying the `OLog` entries. We denote W_{e+1} as the worker count for the next epoch $e + 1$.

At every epoch e, Tips calculates the foreground throughput (F_e), which is the number of `OLog` entries produced by application threads during the epoch, and the background throughput, which is the number of `OLog` entries consumed by the worker threads during the epoch. Tips calculates the processing rate R_e, which is F_e/B_e. It aims to maintain R_e close to 1 by adjusting the number of workers (W_{e+1}).

If $R_e > 1$, the foreground writers are filling up the `OLog` at a faster pace, and if this situation persists, it will lead to blocking of writers as the workers are slow in clearing up the `OLog`s. So Tips will increase W_{e+1} by a predefined step Δ aiming to improve the B_e and keep up with F_e.

If $R_e < 0.5$, workers are clearing up the `OLog`s faster than foreground writers fill them up. Employing excess number of workers is a waste of CPU, so Tips decreases W_{e+1} by Δ.

Finally, if $1 < R_e < 0.5$, TIPS considers that the workers are on par with foreground writers and hence maintains the same number of workers (*i.e.*, $W_{e+1} = W_e$).

TIPS maintains a user-configurable upper bound (W^{up}) and a lower bound (W^{low}) to cap the number of workers (W_e) while scaling up and down respectively. In addition, while scaling up, TIPS memorizes the best performing worker count (W^{max}); so that if W_e reaches the upper bound TIPS can fall back to W^{max} and continue until scaling down is needed.

By default, TIPS sets W^{low} to 0 and W^{up} to the number of physical cores. TIPS uses a smaller Δ when the worker count is small (*i.e.*, $\Delta = 1$ when $W_e < 4$) and uses a bigger Δ when the worker count is large (*i.e.*, $\Delta = 4$ when $W_e \geq 4$).

5.3.2.2 Concurrent Replay of `OLog` Entries

After deciding the number of workers, persist-thread combines the per-thread `OLog` entries and adds them to the per-worker queue. To avoid copy overhead, TIPS maintains only a pointer to the `OLog` record in per-worker queue.

There are two key invariants that must be maintained in the combining process: (1) Since the `OLog` records are replayed concurrently, it is essential to maintain the correct ordering, especially for non-commutative operations (*e.g.*, `insert(k1,v)` and `delete(k1)`). For commutative operations (*e.g.*, `insert(k2,v)` and `delete(k3)`), the replay can be done in any order without violating the correctness. (2) The `OLog` can not be reclaimed until all the entries in the per-worker queues are consumed. For (1), TIPS uses hashing to ensure that all the non-commutative operations will be placed in the same worker-queue. After combining, persist-thread spawns W_e workers. Then each worker sorts the entries in the timestamp (`commit-ts`) order and replays them to the plugged-in index. This ensures that non-commutative operations are always executed by the same worker in their exact order of arrival. For (2), persist-thread waits until the end of the current epoch to safely reclaim the

`OLog` (see the details in §5.3.3.3).

5.3.3 UNO Logging

In this section, we describe the design of ULog and MLog. Similar to `OLog`, the atomicity of ULog and MLog writes is guaranteed by atomic tail pointer update. Both ULog and MLog are protected using a global Readers-Writer lock.

5.3.3.1 Memory Log (MLog)

What to log? All the newly allocated and freed addresses are recorded in the MLog along with a tag to denote if the address is allocated or freed. Also, each allocated address carries a timestamp (`alloc-ts`) to denote the time at which the particular address is allocated. Tips memory allocation APIs internally use PMDK allocator. PMDK not only guarantees failure atomicity for memory allocation and free but also atomic persistence of a variable that stores the NVM heap address [131]; Tips passes an address in MLog to the PMDK memory allocation API that guarantees the address pointing to the allocated memory is persisted when returning from the API. Similarly, PMDK memory free guarantees atomic persistence for an address that is set to `NULL`. During recovery, all the non-`NULL` addresses with the "allocated" tag in MLog are deemed to be non-reachable. Such addresses are freed to avoid memory leaks as the insert operations that created these addresses will be re-executed again from the `OLog`. Similarly, it is possible to re-execute the same `OLog` entry more than once; This can cause a double-free bug if a crash happens amidst a delete operation. To avoid this, Tips logically removes the address from the plugged-in index, stores it in the MLog, and defers the actual memory free until the subsequent `OLog` reclamation. Because after reclaiming the `OLog`, the delete operation can not be re-executed again.

A running example. Suppose that inserting (or deleting) a key triggers split (or merge)

on the leaf node `A` in a B+-tree, and a new leaf node `A'` is allocated (or the existing `A` freed). Say a crash happens before the completion of split (or merge) but after allocating `A'` (or freeing `A`). During the recovery, TIPS will re-execute the same insert (or delete) from the `OLog`, and it once again allocates a new leaf node `A"` (or free `A` again). This scenario leads to a persistent memory-leak of `A'` (or double-free of `A`). With the MLog, TIPS can reclaim the previously allocated node `A'` (or restore node `A`) during recovery to avoid persistent memory leak (or double-free).

5.3.3.2 UNDO Log (ULog)

What to log? ULog is used by the worker threads to guarantee failure-atomic updates to the plugged-in index. Generally, all the addresses that are being modified are required to be logged in the ULog. Instead, in TIPS, we leverage the `OLog` information to selectively log only the addresses that are needed for correct recovery. To decide whether to log a given address, the workers rely on two timestamp information: 1) the time at which the requested address is allocated (`alloc-ts`) and 2) the time of last `OLog` reclamation (`reclaim-ts`). If the requested address has its `alloc-ts` > `reclaim-ts` (*i.e.*, the address is allocated after the last `OLog` reclamation), then the operation to recreate the contents of this address is guaranteed to be present in the `OLog`. Hence the workers skip the UNDO logging. Otherwise, the workers first check if the requested address is already logged due to any previous write request. If so, the workers will skip the UNDO logging; else, they record the contents of the requested address in the ULog. The persist-thread defers the persistence of addresses in the ULog until the subsequent ULog reclamation. Thus, recurring updates to the same address can be batched and persisted at the start of every ULog reclamation.

A running example. Say a B+-tree node `A` is being modified 500 times between two ULog reclamations. Then a worker will log `A` in the ULog at the time of its first modification and

reuse the same record until the next ULog reclamation. After the 500th update, say a ULog reclamation is triggered; the persist-thread before reclaiming the ULog will persist `A` with its latest update. If a crash happens before persisting `A`, during the recovery, Tips correctly spots and reverts node `A` to the state before its first modification from the ULog. Then it re-executes all the 500 updates from the `OLog` to bring `A` to its latest state before the failure.

5.3.3.3 UNO Logging Reclamation

Log reclamation is triggered when any of `OLog` or ULog reaches their preset capacity threshold. The persist-thread always reclaims all the logs together even if only one of them reaches its capacity threshold. That is because, in Tips, the information required for a correct recovery is distributed across all three logs. The reclamation yields for two cases; it waits until the (1) current epoch of background replay to end, and (2) pending scans to finish traversing the OLog. The UNO logging reclamation consists of the following two steps:

Step 1. Flush the addresses in the ULog and MLog. First, all the addresses in the ULog and MLog are persisted by calling `p-barrier`. This guarantees that all the writes that occurred since the last UNO reclamation are persisted.

Step 2. Reclaim the logs. Replayed `OLog` entries and persisted ULog entries in Step 1 are obsolete; they can be safely reclaimed to free up space for the incoming writes. Since the `OLog` has been reclaimed; we no longer required to keep track of the allocated and freed addresses; so the logically reclaimed addresses stored in the MLog are physically freed and then the MLog space can also be safely reclaimed.

Crash safety of reclamation. The reclamation procedure is crash-safe. The crash safety is guaranteed by *atomically setting the `flush_done` flag after the completion of Step 1.* Upon a crash, the recovery procedure first checks the `flush_done` flag. If the flag is set, it means

that Step 1 has been completed successfully before the crash occurred and this guarantees that all the updates to the plugged-in index are persisted. Hence the recovery simply reclaims the remaining logs (Step 2) and terminates. If the flag is unset, then a standard recovery procedure described in the following section is followed.

5.3.4 Recovery

TIPS flushes all the logs and sets the tail pointer of the UNO log to `NULL` upon a safe termination. Therefore, if the tail is non-`NULL` upon a TIPS restart, it triggers the recovery procedure. If `flush_done` is set, recovery proceeds as described in the previous section. Otherwise, the recovery consists of three steps; (1) replay ULog to set the index to the exact consistent state that existed at the last UNO reclamation; (2) free all the newly allocated addresses (before the crash) from the MLog to prevent persistent memory leaks; (3) replay the `OLog` to get to the last successfully committed update before the crash. Note that the logs are replayed only up to their tail pointers to avoid executing any partial writes during the recovery.

If a crash occurs during the ULog replay (Step 1) or MLog free (Step 2), TIPS can continue from where it left off. This is because the changes to the plugged-in index due to ULog replay are immediately persisted. Also, freeing MLog after ULog replay does not affect the persistent state of the index. As described in §5.3.3.1, freeing MLog entries is guaranteed with atomic persistence to `NULL`, making this step failure-safe. On the other hand, if there is a crash while executing `OLog` entries (Step 3), TIPS treats it similar to the failure during normal execution; *i.e.*, after rebooting, TIPS re-executes all the three steps. Because replaying `OLog` during recovery is treated similar to replaying the `OLog` in the background during normal execution. Note that re-executing `OLog`s after the ULog and MLog replay is idempotent, so it does not affect the consistency of the plugged-in index.

5.4 Correctness of Tips

Theorem 1. Tips guarantees Durable Linearizability (DL).

Proof. To guarantee DL, Tips must satisfy three main invariants: (1) Effect of a committed operation can not be undone in the face of a crash. For all the non-commutative operations (*e.g.*, `insert(k1,v)`, `delete(k1)`), (2) the order of commit (*i.e.*, `OLog` write), and visibility (*i.e.*, DRAM-cache write) must always be maintained; and (3) also the background replay must be performed in the linearization order–in the same order as the operations are made visible in the Tips frontend.

For (1), the effect of a write operation is visible only after updating the DRAM-cache entry, which is strictly done after persisting the `OLog` record. This guarantees that the readers will never observe the effects of non-durable write. For (2), the commit and the visibility order for non-commutative operations are synchronized using the per-bucket spinlock in the DRAM-cache as such operations is always guaranteed to happen on the same DRAM-cache bucket. A writer acquires the lock and commits its operation in the `OLog` with a timestamp (`commit-ts`). Then it adds the entry in the DRAM-cache and releases the lock, which guarantees that the order of commit and visibility is always the same for non-commutative operations. For (3), as described in §5.3.2.2, Tips uses hashing to ensure non-commutative operations always go to the same worker queue. Then the worker sorts its queue in the `commit-ts` order and updates the plugged-in index to maintain the linearization order. For commuting operations, maintaining the linearization order is not required as they work on disjoint keys, so the effect of such operations will be the same regardless of the order in which they are executed.

By guaranteeing DL, Tips eliminates the possibility of non-trivial crash consistency bugs as found in the previous work RECIPE [169]. Particularly, Tips avoids the dirty read bugs

as unpersisted writes are never visible to readers as guaranteed by (1). Even if certain reads are served from the DRAM-cache and say a crash happens, the `OLog` reply during the recovery will ensure that the plugged-in index is up to date with all the committed writes that happened before the crash. This ensures that all the pre-crash reads are still valid as they can be retrieved from the plugged-in index. □

Theorem 2. TIPS guarantees DL for scan operations.

Proof. A scan operation in TIPS traverses both the plugged-in index and the `OLog`, and then it merges the results. The DL guarantee can be violated if (1) the scan thread reads a partially written `OLog` entry and (2) if it reads the unpersisted tail pointer. Both these cases can result in loss of data if a crash happens because of reading non-durable data. To avoid (1), the scan thread traverses the `OLog` from head to tail and this will hide any ongoing `OLog` writes as the failure atomicity of an `OLog` write is guaranteed by atomically updating the tail pointer (§5.3.1.5). To avoid (2), the scan thread before starting its traversal, checks if the tail pointer is persisted. If yes, it starts traversing; else, it backs off and retries. □

5.5 TIPS Implementation

TIPS is written in C, and the core library is about 5000 LoC. We use the hardware clock (`rdtscp` in x86 architecture) for scalable timestamp allocation. To address the clock skew between the CPU cores, we use ORDO [148] as done in many other previous works [152, 162]. Note that ORDO does not require any hardware extensions. We set the maximum DRAM-cache chain length to 5 beyond which the `gc-thread` starts reclaiming the applied entries in the chain. We chose this number after carefully considering the DRAM/NVM random read performance ratio; random read latency in NVM is about 5× slower than DRAM [262] so that traversing more than 5 nodes in the collision chain do not pay off.

5.6 Evaluation

We evaluate Tips by answering the following questions: (1) How do Tips perform against prior conversion techniques (§5.6.2)? (2) How do the Tips-indexes perform against prior NVM-optimized indexes (§5.6.3)? (3) How do the Tips-indexes scale for different workloads (§5.6.4)? (4) What is the sensitivity for DRAM-cache and UNO logging size (§5.6.5)? (5) How does Tips impact real-world application (§5.6.6)?

Evaluation Platform. We use a system with Intel Optane DC Persistent Memory (DCPMM). It has two sockets with Intel Xeon Gold 5218 CPU with 16 cores per socket, 512GB of NVM (4×128GB) and 64 GB of DRAM (4×16GB). We used GCC 8.3.1 with `-O3` flag to compile benchmarks and ran all our experiments on Linux kernel 4.18.16.

Configuration. We used YCSB [91]– a standard key-value store benchmark (Table 5.1) for all our evaluations. We used index-microbench [252] to generate the YCSB workloads. We ran the benchmarks for 32 million keys; we first populate an index with 32M keys and then run the workloads, which performs 32M operations. We use random integer and string keys with uniform distribution. We also evaluate the Tips for large datasets and Zipfian distribution in §5.6.4. We preset the size of our per-thread `OLog` and the global ULog to 32MB each, the DRAM-cache to cache 25% of the total number of keys (300 MB) and the upper bound for the number of workers (W^{up}) to 32 (*i.e.*, half of the available CPUs). We present the sensitivity analysis for these configurations in §5.6.5. To ensure a fair comparison, we carefully chose the indexes as the existing conversion techniques are specific to certain concurrency models. We also ported all the indexes to use the PMDK memory allocator. Porting all the RECIPE [169] and NVTraverse [109] indexes with PMDK allocator incurred about 1800 LoC. For all our evaluations (unless mentioned specifically), we use 32 threads to match the maximum physical CPU cores (without hyperthreads) available on our platform

and present the scalability results for all the TIPS-indexes in §5.6.4.

5.6.1 Converting Volatile Index using TIPS

Converting an index using TIPS is simple (§5.2.2.6), and it requires only minimal LoC changes as shown in Table 5.2. For all the indexes, we replace the memory allocation (and free) with `tips_alloc` (and `tips_free`). Below we discuss how we annotated the NVM stores with `tips_ulog_add`.

Index with Single Pointer Updates. Indexes for which write operations involve updating only a single pointer such as a Hash Table (HT), BST, and CLHT requires a `tips_ulog_add` before updating the HT/BST/CLHT node in the insert and delete logic, similar to the example shown in Figure 5.3. The lock-free indexes (LFHT/LFBST) use atomic Compare and Swap (CAS) and we added a `tips_ulog_add` before the CAS logic such that `tips_ulog_add` will be called again upon a CAS retry. Note that repeated UNDO logging will neither impact the correctness nor the performance as TIPS performs UNDO logging for an address only at the time of its first modification (§5.3.3.2). All such conversions require only 5-8 lines of change to the existing volatile codebase (Table 5.2).

Index with Multi-Pointer Updates. For the indexes like the B+tree or ART `tips_ulog_add` is added to backup the B+Tree/ART node before the triggering node split/merge operation. Also, `tips_ulog_add` is added before modifying the B+Tree/ART node in the normal case insert and delete logic. Totally it required only 11 and 9 LoC changes in the B+tree and ART codebase respectively. Note that none of the indexes require any modification in their

YCSB Workload	Read-Write-Scan %	Workload Nature
A	50-50-0	Write Intensive
B	95-5-0	Read Intensive
C	100-0-0	Read Only
D	95-5-0	Read Latest
E	0-5-95	Short Range Scan

Table 5.1: Characteristics of YCSB workloads.

Indexes	Concurrency Control	LoC
Hash table (HT)	Readers-writer lock	5/211
Lock-Free HT (LFHT) [195]	Non-blocking reads and writes	5/199
Binary Search Tree (BST)	Readers-writer lock	5/203
Lock-Free BST (LFBST) [86]	Non-blocking reads and writes	5/194
B+tree	Readers-Writer lock	8/711
Adaptive Radix Tree (ART) [171]	Non-blocking reads/blocking writes	9/1.5K
Cache-line Hash Table (CLHT) [96]	Non-blocking reads/blocking writes	8/2.8K
Redis [52]	Blocking reads and writes	18/10K

Table 5.2: Lines of code (LoC) to convert volatile indexes using Tips.

read and scan logic.

Comparison with other Conversion Techniques. PRONTO [193] requires developers to manually add `op_begin()` and `op_commit()`. With NVTraverse [109] developers must first modify the index implemetation to a "traversal" index and then manually add the `ensureReachable` and `makePersistent` APIs. Similar to the `tips_ulog_add` (§5.2.2.6), the aformentioned APIs will internally issue `p-barrier` without requiring any user intervention. Additionally, unlike PRONTO and NVTraverse, Tips can be used on indexes supporting different concurrency models. Like Tips, both PRONTO and NVTraverse formally prove that their conversion yields correct persistent algorithm by guaranteeing DL. As also reported in the NVTraverse paper, RECIPE [169] can not always guarantee a correct conversion, even for the indexes that fall under their prescribed condition.

Moreover, RECIPE does not formalize the guarantees of their conversions and does not discuss the implications of guaranteeing BDL. Their updated ArXiv version [168] prescribes to selectively add `p-barrier` to specific loads to guarantee DL. But it is left to the developers to figure out the loads that needs to be correctly flushed; this further complicates the conversion. Alternatively, Tips requires only minimal and simple modifications in the volatile codebase. We also formally describe how our conversion guarantees DL and yields a correct persistent algorithm for all our conversions.

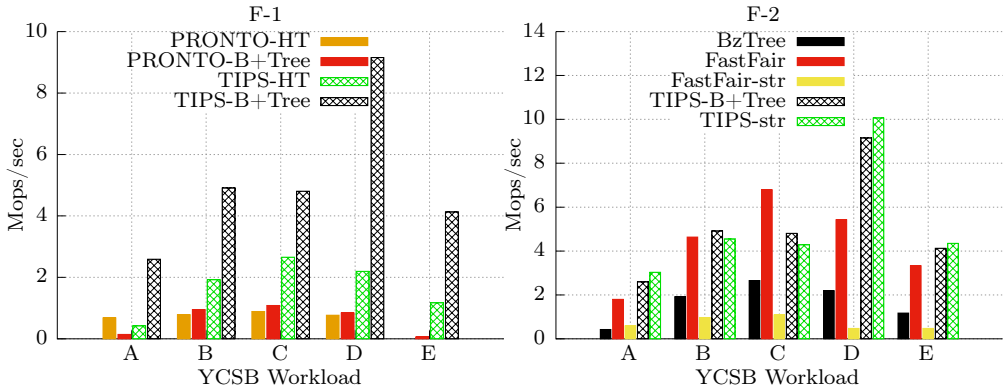

Figure 5.4: Performance comparison of TIPS against PRONTO for Hash Table (HT) and B+Tree (F-1) and TIPS-B+Tree against the NVM-optimized B+Tree indexes– FastFair and BzTree (F-2).

5.6.2 TIPS vs. Other Conversion Techniques

TIPS vs. PRONTO [193]. PRONTO is the state-of-the-art technique to convert globally blocking indexes with DL guarantee. As shown in Figure 5.4 (F1), both TIPS-HT and TIPS-B+Tree outperform the PRONTO counterparts by 20× across all workloads. Although both TIPS (TIPS-HT, TIPS-B+Tree) and PRONTO use RW lock for concurrency, TIPS can process the reads and writes concurrently in the DRAM-cache, and hence it shows a better performance. Also, PRONTO's overhead mainly comes from the synchronous waiting of writers for its background thread to complete the logging. Our performance profiling on the PRONTO-HT reveals that about 25% of the execution time is spent on synchronous waiting. In TIPS, there is no such synchronous waiting as it does not have any separate logging threads. Instead, writers will perform logging in their private OLog. Another source of overhead is the blocking during snapshots, which accounts for 8% of the execution time. Moreover, PRONTO builds its index on DRAM, so it can not scale beyond the DRAM capacity while TIPS can scale up to the NVM capacity. This is a critical design benefit because legacy applications adopt NVM not just for durability but also for its large in-memory capacity.

TIPS vs. NVTraverse [109]. NVTraverse is the state-of-the-art technique to convert

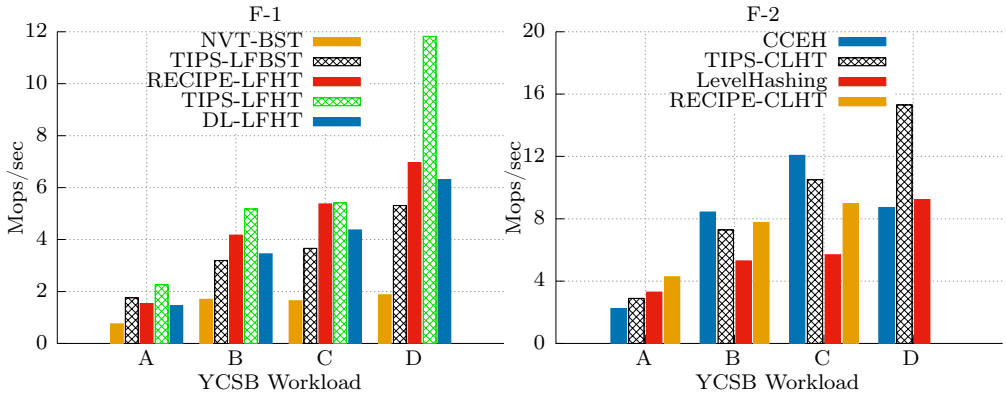

Figure 5.5: Performance comparison of TIPS with NVTraverse (F-1), RECIPE (F-1, F-2) and TIPS-CLHT with NVM-optimized hash indexes– CCEH and LevelHashing (F-2).

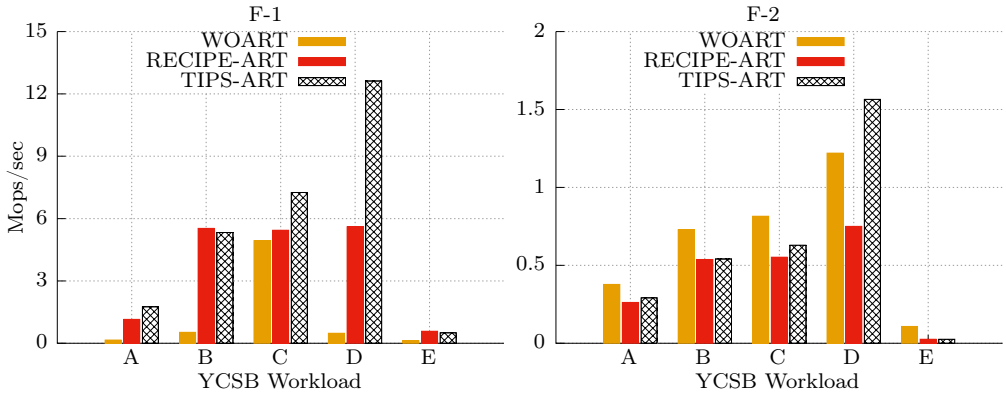

Figure 5.6: Performance comparison of TIPS-ART with RECIPE-ART and WOART for 32 threads (F-1) and 1 thread (F-2).

lock-free indexes with DL guarantee. As shown in Figure 5.5 (F-1), TIPS-LFBST outperforms NVT-BST by up to 3× across all workloads. Further analysis revealed that on average, each read and write in NVT-BST incurs 6 and 17 `p-barriers`, respectively, in the critical path. While TIPS-LFBST incurs only 2 `p-barriers` for each write in the critical path and reads never require `p-barrier`, thanks to the UNO logging and the DRAM-cache. Moreover, TIPS serves up to 25% of read requests from the DRAM-cache. So the readers do not need to traverse the BST on the NVM for 25% of its read requests and hence a better performance.

TIPS vs. RECIPE [169]. Comparing TIPS and RECIPE in terms of performance is not an apple-to-apple comparison as RECIPE supports only a weaker consistency (*i.e.*,

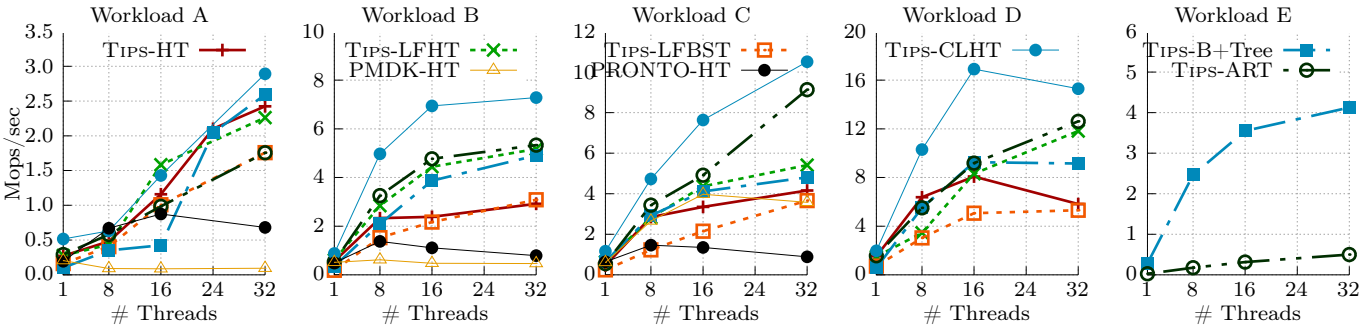

Figure 5.7: Scalability of TIPS-HT (RW lock), TIPS-B+Tree (RW lock), TIPS-LFHT, TIPS-LFBST, TIPS-CLHT and TIPS-ART.

BDL). Besides performance, we also stress how hard it is to achieve DL without trading off performance. Figure 5.5 and Figure 5.6 compare the performance of TIPS and RECIPE for LFHT, ART [170], and CLHT [96], respectively. TIPS indexes perform similar or better than RECIPE indexes across all workloads except for CLHT in workload A. This is because writes to CLHT incurs only one cacheline modification and one `p-barrier` in RECIPE. While in TIPS-CLHT, it incurs two `p-barriers` to commit the `OLog`. Nonetheless, TIPS-CLHT supports DL, and it performs mostly similar to NVM-optimized hash indexes CCEH [204] and LevelHashing [278]. An easy way to guarantee DL, as proposed by Izraelevitz *et al.* [135] is to add a `p-barrier` for every reads and writes. We followed it to make a DL version of the RECIPE hash table (DL-LFHT in Figure 5.5). Such a conversion leads up to 1.4× drop in performance. One can also perform index-specific optimizations like the NVTraverse to guarantee DL. However, it requires expertise in NVM programming and in-depth knowledge of the volatile index. Conversely, TIPS achieves DL with good performance and, notably, in an index-agnostic way.

5.6.3 TIPS vs. NVM-optimized Indexes

TIPS-B+tree. Figure 5.4 (F-2) shows the performance of TIPS-B+tree against Fast-Fair [127], and BzTree [77]. TIPS outperforms BzTree by up to 3× across all workloads;

BzTree uses CoW and PMwCAS [250] to support crash consistency, and this generates a lot of NVM write traffic. Unlike BzTree, Tips-B+tree supports low overhead crash consistency using UNO logging and hence a better performance. Tips performs similar or better than FastFair except for workload C. FastFair, with its smaller fanout (16), provides good point query performance, and Tips-B+tree with a larger fanout (128) provides a good range query performance than FastFair. For string keys, FastFair (FastFair-str) performs up to 5× slower than Tips (Tips-str) as it loses its cache efficiency due to additional pointer chasing to retrieve string keys. Note that Tips stores pointer to its keys for both string and integer keys; therefore no significant performance drop is observed.

Tips-ART. In Figure 5.6, we present the performance of Tips-ART and WOART [167]– volatile ART variant designed for NVM. WOART is single-threaded, and hence we used a global lock for concurrency as done in previous work [168, 169]. WOART performs up to 40× slower than Tips-ART across different workloads due to its poor concurrency model. For a single thread, Tips-ART performs similar to WOART except for workload E. For workload E, the performance of Tips-ART and RECIPE-ART are almost identical; this proves our claim in §5.3.1.6 that the additional `OLog` traversal in scan operations imposes negligible overhead. Our performance profiling revealed that only 2-3% of the time is spent on traversing `OLog` regardless of thread count.

5.6.4 Analysis on Tips Design

Write Scalability. Figure 5.7 shows the scalability of the Tips indexes. For write-intensive workload A, all Tips indexes show good performance and scalability; for Tips-B+Tree a sharp increase is observed after 16 threads. Because (1) for lower thread counts, the aggregate `OLog` size is less (`OLog` is per-thread). Moreover, as Tips-B+Tree uses global RW lock Tips backend writes are slower than the concurrent frontend writes, and consequently, foreground

writers are blocked due to the lack of `OLog` space. (2) For higher thread counts, foreground blocking time is significantly reduced with a larger aggregate `OLog` size, Although the TIPS-HT uses RW lock, it is per-bucket and hence a better level of concurrency than the TIPS-B+tree. This enables TIPS backend to reply the `OLog` concurrently and hence a better performance for TIPS-HT. Still, for 16 threads, TIPS-B+tree outperforms PRONTO-B+Tree which also uses global RW lock by up to 3×.

Read Scalability. All TIPS indexes show good scalability for read-intensive workloads. Indexes supporting concurrent reads show good performance; for instance, TIPS-LFHT performs up to 1.2× better than TIPS-HT (RW lock) in workload C. All indexes regardless of their concurrency model shows high performance for workload D. Because workload D follows read-latest distribution, latest writes are being repeatedly read. Fortunately, the latest writes are most likely to be present in the DRAM-cache; hence a higher read hit ratio and consequently better performance. Although we cache only 25% of the keys for all workloads, the read hit for workload D is about 70%, while for other workloads it is less than 25%.

Impact of UNO Logging. Both TIPS-HT and PMDK-HT (Figure 5.7) use RW lock, and they both use the UNDO logging to guarantee crash consistency while updating the hash table. However, despite the similarities, TIPS-HT significantly outperforms PMDK-HT. The main performance bottleneck in PMDK is the logging operations in the critical path. This is evident from Figure 5.7, PMDK-HT performs and scales on par with TIPS-HT for workload C (100% reads), but its performance plateaus even for a small fraction of writes (5%) in workload B. With UNO logging, UNDO logging is kept off the critical path and consequently making TIPS scale better.

Impact of Skewed Workloads. Figure 5.8 shows the performance of TIPS indexes for Zipfian distribution; all indexes shows up to 2× better performance than the uniform

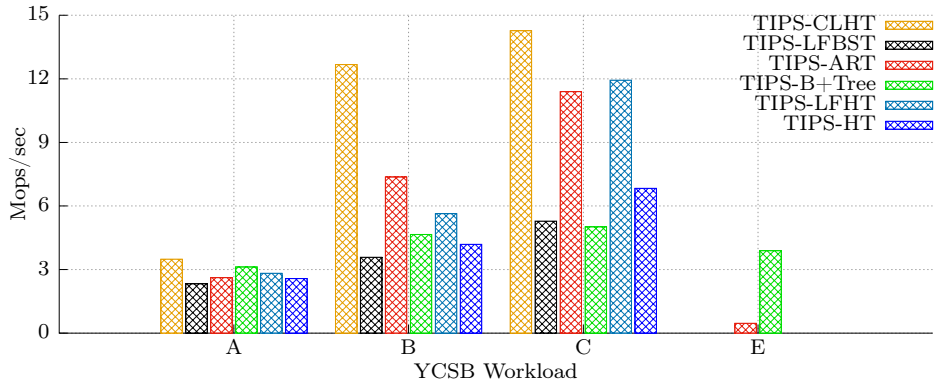

Figure 5.8: Performance of Tips indexes for Zipfian workloads.

distribution. Particularly for workload A, the chances of write coalescing in the ULog increases due to frequent updates to the same address. This reduces the number of UNDO logging performed, and further analysis revealed that the number of UNDO logging performed is reduced by 16% than that of uniform distribution. This further accelerates the background writes, and thus overall write performance is improved. For read-intensive workloads, repeated reading of hotkeys results in more DRAM-cache hits and eventually a better read performance. On average, the DRAM-cache hit ratio is increased by 5% for the Zipfian distribution. Overall, Tips, in particular, UNO logging and DRAM-cache are well equipped to handle the skewed workloads. Note that we did not evaluate workload D as it supports read-latest distribution by default.

Impact of Large Dataset. Figure 5.9 (F-1) presents the performance of Tips-ART for large datasets. While the performance for 64M and 96M keys is mostly the same across all the workloads, there is up to a 23% drop in performance for 128M keys. Particularly for workload C, the performance drop is also observed for 96M keys; for a larger dataset, the ART index grows bigger, and it results in increased pointer chasing to get to the leaf nodes, and hence there is a slight dip in the performance. Note that the RECIPE-ART also exhibits a performance drop of up to 16% across all workloads for 128M keys. Overall, this evaluation

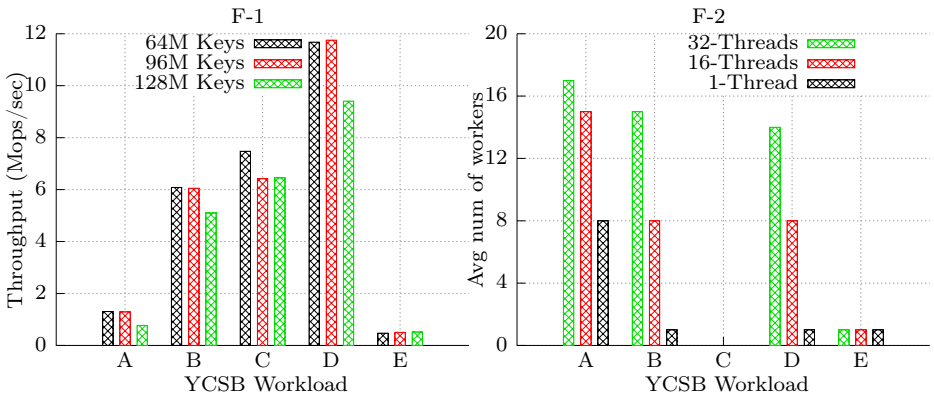

Figure 5.9: Studying the impact of large dataset (F-1) for 32 threads and adaptive scaling (F-2) for varying threads with TIPS-ART.

shows that TIPS as a system can handle much larger datasets effectively.

Impact of Adaptive Scaling. Figure 5.9 (F-2) shows the average number of background workers used for TIPS-ART. More workers are used for workload A as it is write-intensive, for all the other workloads workers count is relatively less as the write ratio is marginal. No workers are created for workload C as it is read-only. Workload E has the same write ratio (5%) as B and D, but TIPS employs only one worker for E. Because ART's scan is inherently slower, hence the foreground threads spend most of the time on the scan operation (*i.e.*, foreground writes are slow), this gives TIPS enough time propagate the updates. Thus our adaptive scaling can effectively adapt for the nature of workloads and the plugged-in index.

5.6.5 Sensitivity Analysis

Sensitivity to DRAM-cache Size. As shown in Figure 5.10, for read-intensive workloads, about 1.8-2.8× performance increase is observed as % keys cached increases. This is because the read hit ratio increases as more keys are being cached, enabling readers to complete their reads on the faster DRAM. Since workload A is write-intensive, it is less sensitive to DRAM-cache size. As more writes happen on the DRAM-cache, the applied keys are actively evicted, and it stores mostly the newly written keys. This poorly impacts the read hit ratio,

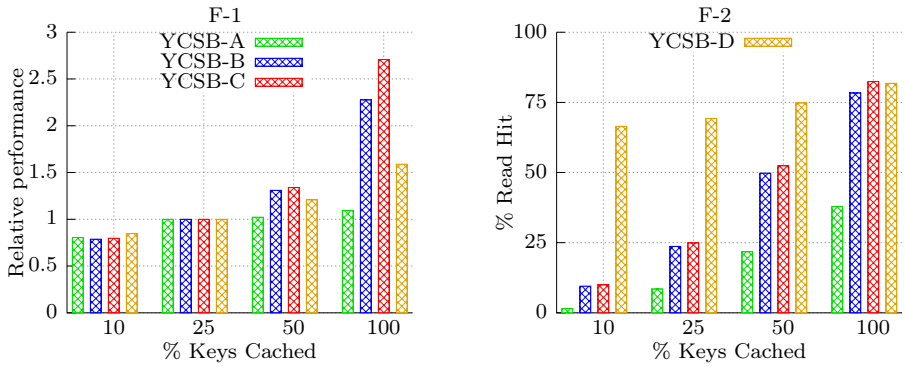

Figure 5.10: Performance sensitivity of TIPS-B+tree (F-1) and read hit % (F-2) for the varying DRAM-cache size. X-axis represents the % of keys cached in the DRAM-cache (default = 25%).

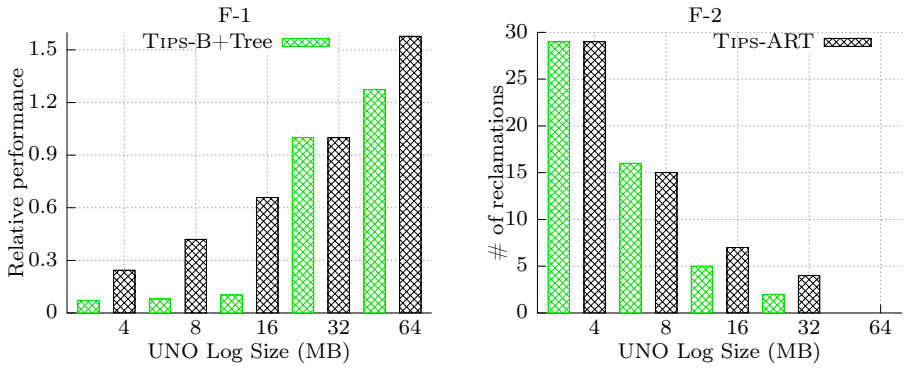

Figure 5.11: Performance sensitivity (F-1) and the number of log reclamations triggered (F-2) in TIPS-B+Tree and TIPS-ART for the varying UNO log size for Workload A (default = 32MB).

and consequently, readers are forced to fall back to the B+tree on the NVM.

Sensitivity to UNO Log Size. Figure 5.11 illustrates the performance of TIPS-ART, TIPS-B+tree for different UNO log sizes for the write-heavy workload A. As shown, there is a 4× and 9× performance drop for TIPS-ART and TIPS-B+tree with 32MB (default) and 4MB log size, respectively. This is because as the log size becomes smaller, the foreground writers are blocked for more time as TIPS-B+tree (RW lock) has a single-threaded backend. Whereas TIPS-ART supports concurrent backend and is relatively less affected by decreasing log size. Both TIPS-ART and TIPS-B+tree shows a 1.6× performance increase for 64MB because the log size is big enough to completely buffer all writes, and zero reclamations are triggered

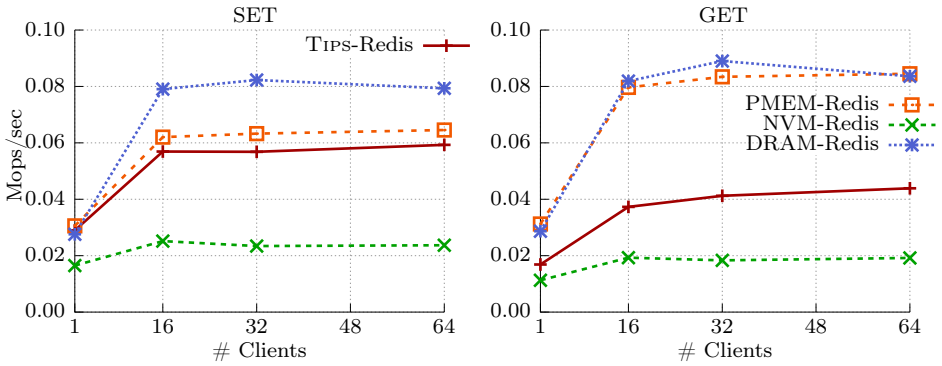

Figure 5.12: Performance comparison of TIPS-Redis with vanilla Redis running on DRAM and NVM, and Intel's PMEM-Redis.

for 64MB. Also, note that more than 3 reclamations are triggered for a default log size of 32MB. The impact of smaller log sizes is relatively marginal for read-heavy workloads. The performance change is negligible until 8MB, and about a 20% performance drop is observed for 4MB log size.

5.6.6 Real-world Application: Redis

We ported a popular DRAM key-value store, Redis [52], using TIPS (TIPS-Redis). We compare its performance with the vanilla Redis running on the DRAM (DRAM-Redis) and NVM (NVM-Redis), and also with Intel's PMEM-Redis [44]. Note that NVM-Redis does not ensure crash consistency and PMEM-Redis stores only the values in NVM. Figure 5.12 shows the performance of Redis GET and SET operation evaluated using Redis-Benchmark [49]. We ran the benchmark for 32M keys (8-bytes) with a uniform random distribution. For SET operations, TIPS-Redis consistently outperforms the NVM-Redis by 2.5×, and it performs up to 1.5× and 1.1× slower than the DRAM-Redis and PMEM-Redis, respectively. For the GET operations, TIPS-Redis perform up to 2× better than the NVM-Redis and up to 2.2× slower than the DRAM-Redis and PMEM-Redis. TIPS-Redis maintains all the data and the Redis core on the NVM, so (1) it provides a larger in-memory capacity (4× in our experiment) and immediate durability. (2) Both PMEM-Redis and DRAM-Redis take about 100 seconds to

restore data from disk every time a server instance is created. While Tips-Redis takes less than 1 second to recover upon safe termination.

5.6.7 Recovery

We performed the recovery test on all the Tips-indexes. We injected crash 200 times arbitrarily using `SIGKILL`, similar to previous work [169, 173]. We also tested a crash during the recovery procedure. All Tips-indexes successfully recovered after every crash. The worst-case recovery time would be a crash happening when the `OLog` is full. To measure this time, we injected a crash just when the `OLog` becomes full. Recovery time ranges between 0.5 and 9 seconds depending on the number of `OLog`s and concurrency control of the index.

5.7 Chapter Summary

We discussed Tips, a framework to systematically make volatile indexes and in-memory key-value stores persistent. At its core, Tips adopts a novel DRAM-NVM tiering to support index-agnostic conversion and durable linearizability. With the tiered concurrency model, Tips achieves good scalability, performance, and enhanced applicability. UNO logging protocol is critical to achieve low crash consistency overhead and prevent persistent memory leaks. In our evaluation, we showed that Tips could be effectively applied to indexes with varying concurrency models and the Tips-enabled indexes shows excellent performance against the state-of-the-art index conversion techniques and NVM-optimized indexes.

Chapter 6

Framework for Near-Storage Computing

The idea of moving compute near the storage has been proposed more than two decades ago [73, 150, 223]. However, it did not gain traction as the CPU was scaling well and storage was considerably slower than the CPU. Now towards the end of Moore's Law as the CPU scaling has plateaued and new storage technologies are becoming faster, CPU is unable to fully saturate the IO bandwidth [172]. This has caused data movement between storage and memory to become a critical performance bottleneck. This is evident in ML/DL training models which typically process petabytes of data. While the training runs on GPUs, the ML training still relies on the CPUs for data fetching and pre-processing [199]. Hence the ML/DL training is primarily bottlenecked by the stalls due to, 1) fetching data from the storage, and 2) preprocessing the data using the CPUs [88, 163, 199, 239].

Computational Storage Devices (CSDs) are seen as potential solution for reducing the data movement and relieve CPU from becoming a performance bottleneck. Typically, CSDs include a near-storage processing unit (*e.g.*, ARM CPU or FPGA) and this enables it to perform compute near the storage without having to move the data independent of the CPUs [55, 59, 207]. Programming using CSDs has been a challenging problem thus far due to lack of standard programming frameworks that provides application the flexibility

to seamlessly offload compute to the storage. In this work, we propose TENET, a unified key-value store (KVS) framework with computational pipeline natively designed for CSDs. With TENET we manage to improve the programmability of CSDs by providing application the flexibility to seamlessly compose and offload compute to the storage during the run time using a familiar set of embedded KVS interface. Further, with TENET, we strive to make a compelling usecase for CSDs by offloading the entire preprocessing steps in the ResNet50 [53] DL model to near-storage FPGA. Overall, TENET makes three main contributions:

- **Cross-layered KVS Architecture:** TENET prudently leverages both the host CPU and near-storage FPGA to best of their abilities, instead of naively offloading every operation to the storage. Particularly, TENET proposes a cross-layered indexing and caching to leverage the CPU as a control plane and the FPGA as a compute plane to reduce data movement and to improve compute performance.
- **Near-storage Arbiter** to schedule the offloaded tasks, manage memory, and establish an efficient communication model across compute kernels and the host CPU. This is key to improving the programmability of the CSDs by providing the application to flexibly compose and seamlessly offload the compute on-fly to the near-storage FPGA.
- We evaluate TENET on Samsung SmartSSD [59] a real CSD hardware which to best of our knowledge is the first system to do so. We evaluate TENET by offloading complex compute functions such as JPEG decoder and snappy compression to the FPGA. In addition we integrate the TENET KVS framework to the Tensorflow [114] and train the ResNet50 DL model with ImageNet [236] dataset by completely offloading the entire image pre-processing steps to the near-storage FPGA. Our evaluations show that, TENET performs up to 75% faster and saves up to 65% CPU time in comparison to the CPU-only system.

6.1 RETINA Design

RETINA KVS comprises of two critical components, 1) a cross-layered index structure to manage data on the SSD, and, 2) an FPGA Arbiter to compose and execute compute functions on-fly on the data stored in the SSD. We explain the cross-layered architecture and the arbiter design in §6.1.1 and §6.1.2 respectively.

SmartSSD. We explain the RETINA architecture using SmartSSD as a representative CSD. SmartSSD (green box in Figure 6.1) consists of a NVMe SSD (PCIe Gen3), a Xilinx KU15P FPGA [35], and a 4GB of dedicated DRAM all connected internally via a P2P interconnect. DRAM inside the SmartSSD is mapped to the PCIe bar address which is referred to as Common Memory Area (CMA). Essentially, CMA is a shared memory region and is read-write accessible for both the host CPU and the FPGA. Data transfers between the CPU and the FPGA occurs across the PCIe bus and the same between the CMA and the SSD occurs via the P2P interconnect within the SmartSSD.

6.1.1 Cross-layered Index Structure

RETINA uses a hybrid (trie and b+tree) based index structure [156, 187] for scalability and performance. Using trie as internal nodes (search layer) enables RETINA to have optimized search latency even with larger keys. However, trie is know for its poor range query performance, cache inefficiency, and high memory overhead. To overcome the first two limitations, we choose a B+tree style leaf nodes which is both cache efficient and better optimized for both point and range queries. To reduce the memory overhead we use the Adaptive Radix Tree (ART) [170] which is a memory optimized prefix-trie.

As illustrated in the Figure 6.1, the search layer is traversed and update by the host CPU (❶, ❸) while the data nodes are traversed and updated by the FPGA (❺, ❻). The goal of

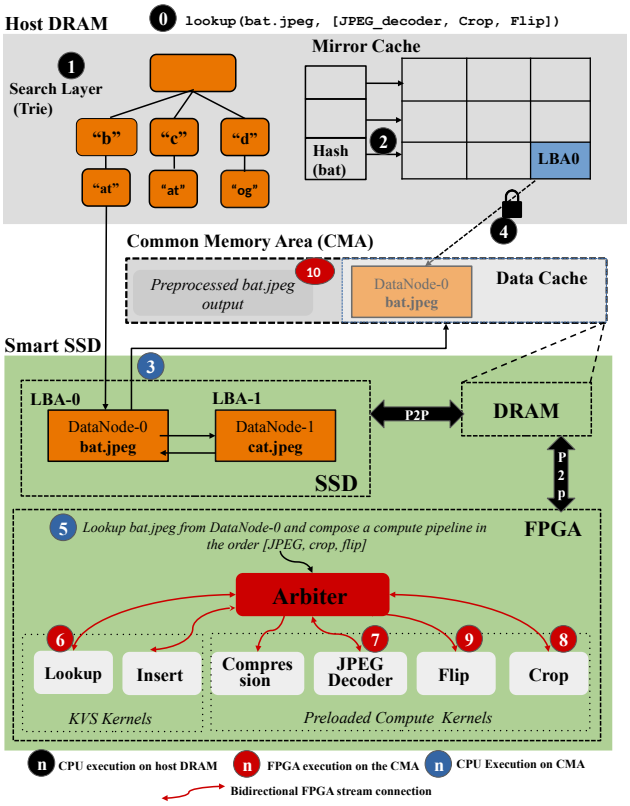

Figure 6.1: Applications use the RETINA APIs to access the CSD (❶). First, RETINA traverses the search layer using the input key to find the offset of the data node where the key-value pair exists (❶). Then it checks the mirror cache to see if the data node is already cached (❷) if not then the data node is fetched from the SSD to the data cache (❸). If the data node is already cached then RETINA skips step ❸ and goes on to acquire the lock on the *DataNode-0* (❹). RETINA invokes the Arbiter on the FPGA (❺) which then executes the lookup operation on *DataNode-0* to retrieve the image (❻). Arbiter performs the preprocessing steps in the application specified order (❼, ❽, ❾) and saves the output to the CMA (❿).

the cross-layered approach is to use the CPU as the control plane (*e.g.*, trigger FPGA calls, concurrency control, etc) and offload the compute to the FPGA which is closer to the data (*e.g.*, perform lookups and inserts on the leaf nodes). This design is effective because the it leverages the high internal P2P bandwidth between the FPGA and the SSD *i.e.*, unlike the traditional systems, the leaf node access does not require PCIe hops rather it is offloaded to the near-storage FPGA thereby preventing unnecessary data movement between the storage and the CPU memory.

The search layer stores the key and the Logical Block Address (LBA) of the data node

and the data node stores the keys and its value in the sorted order on the SSD. Therefore, traversing the search layer will lead to the data node where the target key-value pair exists (❶). When a data node undergoes split/merge operation the search layer is updated to reflect the structural modifications. The concurrent access to the date nodes are managed using the Readers-writer (RW) lock; the CPU after retrieving the LBA of the data node acquires the RW lock before transferring the control to the FPGA (❹).

6.1.1.1 Cross-layered Caching

As stated earlier, CPU traverses the search layer, retrieves the LBA and loads the data node to the CMA (❶, ❷, ❸) for the FPGA to execute the lookup/insert operation (❻). While this already reduces the data movement between the storage and the host memory, RETINA further optimizes the data movement with a cross-layered caching layer between the host memory and the CMA. A portion of the CMA region is reserved for caching (data cache) the frequently accessed data nodes and to know if a particular data node is present in the data cache RETINA uses a mirror cache in the host memory. Mirror cache prevents additional PCIe hops to check the data cache for the data node. Mirror cache just stores the LBA of a data node and its corresponding offset in the CMA. After traversing the search layer (❶) the CPU checks the mirror cache (hash table with a 2d array) to see if the particular LBA is already cached in the data cache (❷). Upon a cache miss, the CPU loads the data node to the data cache (❸) and updates the mirror cache (❹). Any future requests to this data node will not incur data transfer from the SSD to CMA. If the data node in the cache is updated (*e.g.*, insert a new key) it is immediately written back to the SSD to maintain the consistency. Mirror cache is protected using a per-bucket RW lock and the mirror cache uses LRU policy to evict cold data nodes when the cache is full.

6.1.1.2 Crash Consistency

RETINA employs a log-less crash consistency technique for better performance and to eliminate any data movement from storage to memory due to logging operations.

Crash consistency for data nodes. In RETINA, the size of data node is fixed at 4KB to leverage the 4KB atomic read/write guarantees of the filesystem to guarantee crash consistent updates to the data nodes. Such a design removes the need for performing logging operations. So the data nodes are always updated in-place and is written back to the SSD (from the CMA) in the 4KB granularity.

Extension pages. A data node can not accommodate a key-value pair that is greater than 4KB. In such cases, RETINA allocates an extension page large enough to fully fit the key-value pair, writes the data to the extension page, and chains the extension page to the corresponding data node. RETINA implements a slab allocator to manage the extension pages; the slab allocator maintains pages of different sizes (4KB - 8MB) which are allocated on the SSD. Further, it maintains a bitmap to track the live pages and it garbage collects the freed/zombie pages during the runtime and recovery.

RETINA employs a version based crash consistency for the extension pages. Unlike the data nodes the extension pages are always updated out-of-place in copy on write (CoW) fashion. To guarantee crash consistency, the data node and its extension page maintains the *version number metadata* which is a 64-byte integer. First, the extension page is allocated and written, its version number is updated. Then the extension page is chained to the data node and the version number in data node is *atomically* updated (using 4KB atomic write) to match the version number of its extension page, only after which the new extension page becomes visible. If a crash happens before updating the data node's metadata the allocated extension page will be reclaimed during the recovery.

6.1.2 Near-storage Arbiter Design

RETINA implements a data flow programming model for FPGA execution *i.e.*, execute the compute function on the FPGA in a *parallel pipelined fashion*. A straightforward way to create a computational pipeline is by directly connecting the compute functions one after the other to form a hardwired unidirectional pipeline. However, such a design becomes highly inflexible and complicates the application programming. In Figure 6.1 (❺-❾), the application requests an image to be fetched and preprocessed in a specific order (decode, crop, flip); say if the application requires a different sequence (*e.g.*, decode, flip, crop) or may be it requires an additional step for certain images (*e.g.*, decode, resize, crop, flip) then in such scenarios the CSD must be offlined to rewire the compute function connections. RETINA implements a Arbiter on the FPGA to impart flexibility to compose different orders of compute pipeline on-fly. Essentially, Arbiter is an FPGA IP that *arbitrates* the pipeline by ordering the compute kernels in the application specified order.

Communication model. The CPU-Arbiter communication happens via the shared memory (*i.e.*, the CMA); CPU loads the data to the CMA from the storage and then triggers the Arbiter with the offset to the input data (❺). Similarly, Arbiter returns the output to the CPU via the CMA (❿). However, the compute function kernels do not directly communicate with each other instead they communicate only via the Arbiter. Each compute kernel establishes a bidirectional communication channel with the Arbiter using FPGA streams. The Arbiter sends the required inputs for a compute kernel execution via input stream connection and the compute kernel after the execution sends the output to Arbiter via the output stream. Arbiter allocates a buffer for each compute kernels where the intermediate results are stored. Instead of passing the raw data, the Arbiter simply passes the offset to these intermediate buffers as an input to compute kernels. This helps to conserve and

```
1 bool insert(key_t k, value_t v, compute_t *args, int n_args);
2 bool update(key_t k, value_t v, compute_t *args, int n_args);
3 bool delete(key_t k, compute_t *args, int n_args);
4 value_t *lookup(key_t k, compute_t *args, int n_args);
5 value_t *scan(key_t start_key, int n_keys, compute_t *args, int n_args);
```

Figure 6.2: RETINA APIs

judiciously use the FPGA stream bandwidth.

Memory management. A small portion in the CMA is reserved intermediate buffers during the init time. The Arbiter allocates a buffer on-fly and passes it to the compute kernels where the kernels write their output. Once the kernel execution is done then the Arbiter reclaims and recycles the buffer for the future use. The Arbiter maintains a bitmap of metadata to track the free and in-use buffers during the execution. Note that the buffer where the final output is stored is allocated by the CPU and passed to the Arbiter and CPU will free the output buffer after it consumes the output.

Task scheduling. The Arbiter orchestrates the compute kernel execution in the CPU specified order. For instance, in Figure 6.1 the arbiter first triggers the lookup kernel to retrieve the image from the data node (⑥), then forwards the output of the lookup kernel to the JPEG decoder (⑦), then starts the crop execution with the decoders' output (⑧) and flip execution with the crops output (⑨).

6.1.3 RETINA Programming Interface

RETINA provides a standard KVS style APIs which can be used to access the data on the SSD and also to compose compute functions in any order during the run time. As shown in the Figure 6.2, the KVS APIs has two additional arguments *args and n_args* in additional to the standard key and value inputs. Applications can create a single or an array of compute functions and pass the arguments required for each compute functions if any (*e.g.*, compression ratio for the compression kernel) as a single argument (*args*) along with the size of input

array (n_args). These inputs will be interpreted by the CPU and eventually passed on to the Arbiter which will compose and execute the compute functions in the exact order.

6.2 Evaluation

We evaluate RETINA by answering the following questions: 1) What are the performance benefits of offloading compute from CPU to the storage? 2) Does offloading compute to storage reduce the overall CPU utilization? 3) What are the impacts of RETINA on real-world application?

Evaluation platform. We use a server equipped with Samsung SmartSSD [59]. The server has 8-core Intel Xeon Gold 6152 CPU with two NUMA nodes, 1 NVIDIA Tesla V100 GPU and 263GB DRAM memory. We use Vitis Accel [67, 68] toolchain to configure, compile FPGA kernels and OpenCL APIs to communicate with the FPGA.

Evaluation configuration. We implement and evaluate 6 compute kernels (JPEG decoder, Snappy file compression, flip, crop, and resize) and 6 KVS kernels (lookup, insert, update, delete, split, and merge). 2GB of CMA is reserved for the data cache and the mirror cache consumes 512MB of host DRAM as it stores only the offset and other metadata of data node in the data cache. In all our evaluations the FPGA (kernel side) concurrency is set to 1 *i.e.*, RETINA maintains only one instance of each FPGA kernels. This is because the KU15P FPGA board is relatively smaller and it runs out of logic cells when creating more than one instance. For instance, 1 instance of each lookup, insert, split, Arbiter, and JPEG decoder kernels altogether consumes ∼73% of the available FPGA resources, so increasing the kernel instances results in FPGA build failure. Consequently, we also limit the host-side concurrency to 1 thread as there is no benefit to have more host side concurrency when multiple threads will end up waiting for one instance of FPGA kernel to be available.

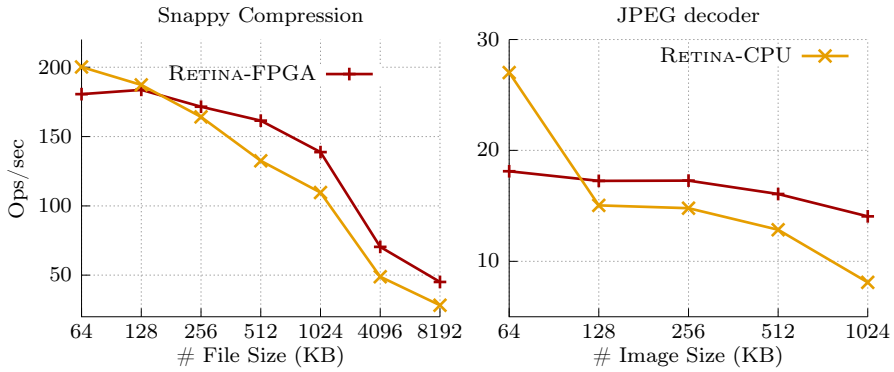

Figure 6.3: Performance comparison of Snappy compression and JPEG decoder executed on the CPU and the FPGA.

6.2.1 Near-storage vs CPU Compute

To study the benefits of moving compute near the storage, we offload the Snappy compression and JPEG decoding to the near-storage FPGA and compare its performance against the CPU execution. For FPGA, we use the Snappy compression and the JPEG decoder in the open-sourced VITIS library [33, 61]. For CPU, we use the Google's Snappy implementation [60] and the JPEG decoder in the openCV [32] library.

The benchmark executes 2M operations on the 1M preloaded key-value pairs. For Snappy compression, key is the file-id (integer) and value is the compressed text file. For JPEG decoder, key is the image name (string) and value is the JPEG image. In the compression benchmark, 1 operation involves compressing a text file on the CPU or FPGA and inserting the compressed file to the SSD. For the JPEG decoder, 1 operation involves an image lookup on the SSD and decoding the retrieved image on the CPU or the FPGA.

Performance. Figure 6.3 shows the performance of executing Snappy compression and JPEG decoder on FPGA and CPU for different file sizes. Both RETINA-FPGA and TENET-CPU uses the same RETINA KVS framework to insert/lookup the data in the SSD. *Essentially, this means that the performance delta between the* RETINA-*FPGA and* TENET-*CPU comes*

Size (KB)	Snappy Compression		JPEG Decoder	
	CPU (ms)	FPGA (ms)	CPU (ms)	FPGA (ms)
64	0.4	0.65	10	27
128	0.7	0.7	37	30
256	1.2	0.8	40	31
512	2.8	1.4	55	36
1024	4	2.3	96	45

Table 6.1: Execution time for CPU side and FPGA side snappy compression and JPEG decoder for different input sizes.

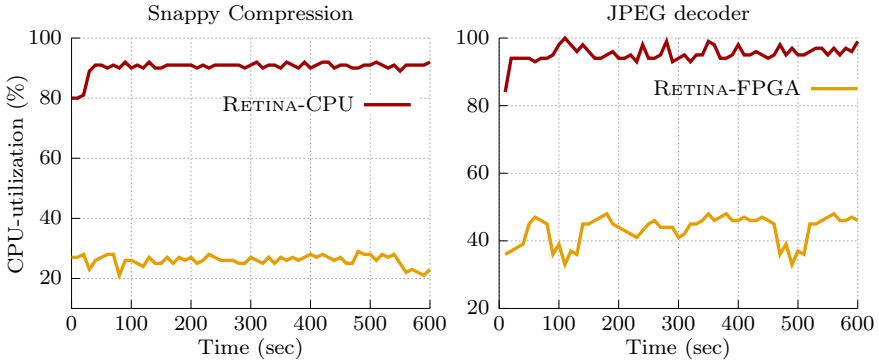

Figure 6.4: CPU utilization for Snappy and JPEG decoder.

only from performing snappy compression/JPEG decoding on the CPU and FPGA.

RETINA-FPGA performs up to 70% faster than RETINA-CPU for both Snappy compression and JPEG decoder. The benefit of offloading compute to the storage is more pronounced for larger file sizes. Unlike the CPU execution, the large files does not have to be moved from storage to the CPU memory for the offloaded FPGA execution. Further, FPGA is more suited for performing such compute intensive decoding and compression operations. This is corroborated by Table 6.1 which shows the execution time of Snappy compression and JPEG decoding on CPU and FPGA. FPGA takes up to 2× less time as the input size increases from 64KB to 1MB.

CPU utilization. In addition to the performance benefits, offloading compute to the FPGA conserves CPU time as illustrated in Figure 6.4. RETINA-FPGA consumes ∼65% less CPU time than RETINA-CPU. This is because, the role of CPU in the RETINA-FPGA is just to

orchestrate the data movement between the storage and the CMA and to trigger the compute function calls on the FPGA. Retina-FPGA consumes 20% more CPU time for JPEG decoder as compared to Snappy compression because JPEG decoding involves additional metadata processing on the CPU side particularly reconstructing the YMV metadata. We believe that a more FPGA optimized JPEG decoder will further reduce CPU time.

6.2.2 Near-storage Deep Learning Preprocessing Pipeline

We evaluate Retina by offloading the image preprocessing steps in the ResNet50 [53] DL model training to the CSD. We use Tensorflow [114] framework as it allows to add custom filesystem support [63]. The ResNet50 model includes three preprocessing steps wiz., JPEG decoding, crop, and flip before feeding the images to the GPU for training. Usually, the images are retrieved from the storage and preprocessed using the CPU. In Retina, the images are retrieved, preprocessed at the storage using the near-storage FPGA, and the preprocessed image can be directly fed to the GPU. Generally, Tensorflow allows input images to be fetched in a batch (*e.g.*, 128 images) and creates a pipeline of preprocessed images to be fed to the GPU. Tensorflow uses multiple CPU cores to enable parallel preprocessing. Since Retina can not create more than one instance of FPGA kernels we set the batch size to 1 and force the Tensorflow to use only one CPU core for preprocessing for a fair comparison.

Integrating Retina with Tensorflow. Tensorflow provides a virtual wrapper class for filesystem and the developers can overload the virtual class functions to execute their own filesystem logic during the Tensorflow runtime. To this end, we overloaded the Tensorflow filesystem layer to call Retina's KVS APIs. For instance, read and write filesystem function in the Tensorflow will call Retina's *lookup* and *insert* functions respectively. Essentially, when the Tensorflow calls image fetch Retina would lookup the particular image, perform the preprocessing steps on the SSD, and returns the preprocessed image which the Tensorflow

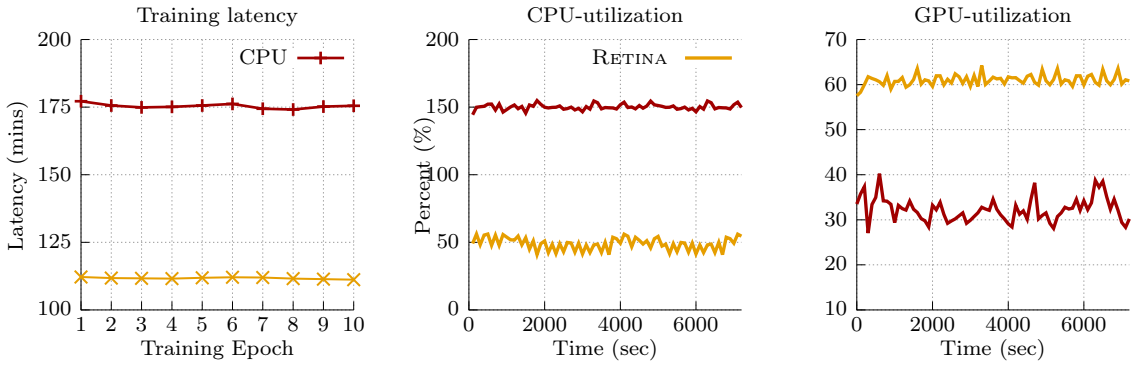

Figure 6.5: Performance and resource utilization comparison for ResNet50 DL model training using ImageNet dataset when image preprocessing done on the FPGA (RETINA) and the CPU.

directly feeds to the GPU. Note that we load the full ImageNet dataset to the RETINA KVS before the training begins.

Performance analysis. Figure 6.5 shows the time take per epoch for ResNet50 training when the preprocessing is done using RETINA and compares it against the CPU side preprocessing. RETINA consumes ∼60 minutes less than the CPU per epoch and for the whole training (10 epochs) RETINA is ∼36% faster than the CPU. This can be attributed to two reasons, 1) as seen in §6.2.1, JPEG decoding on the FPGA was considerably faster than the CPU particularly for larger image sizes. Given that the most of images in the ImageNet dataset are greater than 100KB, offloading JPEG decoding to FPGA reduces the compute time. 2) data movement is significantly reduced in the RETINA; the ResNet50 model requires the size of a preprocessed image to be exactly 224*224 regardless of its original size. This means, in RETINA only the final 224*224 image moves from the storage to CPU memory at all times. Whereas in the traditional CPU based training the unprocessed large JPEG images are constantly moved from storage to memory. In addition, RETINA also consumes 100% less CPU time as the preprocessing is offloaded to the FPGA and Consequently it reduces the GPU idle time by up to ∼25% during the training time as illustrated in Figure 6.5.

6.3 Chapter Summary

In this chapter we discussed Retina, near-storage compute capable KVS designed natively for CSD. Retina proposes a cross-layered KVS architecture that leverages host CPU as a control plane and near-storage FPGA as a data plane. Retina KVS includes a centralized Arbiter implemented on the FPGA to enable applications to compose compute pipeline on-fly. We evaluated Retina on a real CSD drive (SmartSSD) and our evaluation showed that Retina performs up to 75% better, saves up to 65% less CPU time, and significantly reduces the data movement as compared to the traditional CPU only compute.

Chapter 7

Future Works

The systems proposed in this thesis are implemented and evaluated on byte-addressable NVM (Intel DCPMM). However, the problems identified and the solutions proposed in this thesis is applicable to the future byte-addressable NVM technologies based on NAND flash [2, 34, 36], NRAM [111], STT-MRAM [20], battery-backed DRAM [3, 21, 37, 116, 225], and PRAM [40].We identify two potential interesting directions, 1) exploring the ideas of Tips, TimeStone, and Tenet for CXL (Compute Express Link) systems and 2) making Retina a distributed KVS and building towards making it a de-facto storage stack for different CSD architectures.

7.1 CXL Based Systems

CXL [94] is a cache coherent protocol implemented over PCIe gen5 interconnect. With CXL, systems can expand its memory capacity beyond the physical DIMM slots and more importantly CXL enables cache coherent access to the expanded memory regions. CXL does this by mapping the external memory to the cache coherent system memory space. Moreover, CXL also enables external accelerators (*e.g.*, GPU, FPGA) to access the system memory in a cache coherent manner. This means that with CXL, the accelerators can now directly access

system memory just like multiple CPUs coherently accessing the cache and system memory in a multi-core CPU architecture.

CXL enabled NVM. CXL allows to extend the capacity of byte-addressable NVM and DRAM beyond the physical DIMM slots in the system [23, 117]. Researchers are also proposing to transform the block interface of the SSDs to byte-addressable interface using the CXL protocol [143]. This opens up new opportunities for having a byte-addressable NVM at a much cheaper cost and a larger capacity. However, accessing the expanded memory over CXL (*e.g.*, in the event of a cache miss) will incur much higher latency than accessing the memory connected directly to the system bus [237] So, hiding the far-memory access latency will be key to realizing this architecture in practice. The caching designs proposed in this thesis (DRAM cache in Tips, `TLog` in TimeStone) will be a viable design choices to hide the high far-memory access latency by leveraging the faster system-connected DRAM/NVM. Further, with CXL memory safety and fault tolerance becomes crucial as in the traditional system attached NVM. Tenet's asynchronous replication idea and the memory safety ideas will be highly relevant for CXL attached NVM. CXL technology is still nascent and it is very likely that new performance bottlenecks will be identified as the technology becomes more accessible. The fundamental research problems identified in this thesis in the aspects of programming abstraction, multi-core scalability, memory safety, and reliability will be the core problems in the context of CXL enabled far-memory. We believe that the Tips, TimeStone, and Tenet will serve as a strong foundation for CXL research and extending these ideas for CXL would be an important and impactful research direction.

Concurrent programming for CXL enabled accelerators. With CXL, any number of PCIe connected accelerators can access the system memory in a cache coherent manner along with the CPUs. This opens new challenges in establishing a synchronized access and designing a capable concurrent scheme to achieve high-performance and better programmability. It

will be an interesting research direction to extend TIMESTONE's transactional programming to enable accelerator programming. Instead of relying on adhoc concurrency techniques that works on CPU and accelerators separately, TIMESTONE can unify the concurrent programming across accelerators by extending its transactional APIs. While there will be myriad of challenges in implementing a transactional programming model for accelerators we believe that this is will be an important research problem because programmability has been one of the critical challenges in the heterogeneous compute environment.

Programming model for heterogeneous storage stack. Another orthogonal direction would be to make TENET far-memory aware. With CXL, it is more likely for storage architecture to be heterogeneous; for instance, a possible memory hierarchy would be system attached DRAM/NVM, CXL attached DRAM/NVM/SSD, and also traditional SSDs/HDDs connected via the PCIe bus. In such a heterogeneous storage environment, we believe that programmability will be key challenge. Extending TENET to be CXL aware would enable a transactional access across the different storage media and this would simplify the programming challenges. Given that TENET is scalable, memory safe, and fault tolerant extending it to seamlessly work across storage media would be an impactful research path.

7.2 Making RETINA the De-facto Programming Framework for CSDs

Evaluating RETINA for scalability. Although we designed RETINA with host and kernel side scalability in mind, we were unable to test the design due to current hardware limitations. A more powerful hardware will enable us to validate the scalability of RETINA design. For instance, the next-generation of SmartSSD [58] is expected to more powerful Versal series FPGA [66]. This would enable RETINA to have multiple instance of FPGA kernels which is key to improving the pipeline parallelism and achieving a better throughput. Another interesting direction is to test RETINA in distributed setting where we run applications on

multiple CSDs. New CSD hardware such as the Scaleflux's CSD 3000 [55] supports out-of-box device virtualization using SR-IOV will be an interesting platform to evaluate Retina.

Making Retina portable across different CSD architectures SNIA has recently released a standard programming model for CSDs called the CS APIs [13]. Rewriting Retina using CS APIs would make it portable across different CSD architectures. Although Retina is evaluated on SmartSSD, we designed it to be used across different CSD architectures. But it is unclear at this point whether Retina's design would work out-of-box on other architectures; it would be interesting direction to make Retina portable across different CSD architectures as our end goal is to make Retina a de-facto programming framework for any applications that needs to adopt CSD in its storage stack.

Chapter 8

Concluding Remarks

Storage technologies are evolving to be faster and feature rich, they are no longer simple, slow block storage devices. Modern storage technologies are becoming memory-centric and this enables applications to access persistent data at main memory speed. With CPU scaling plateauing, the idea of moving compute closer to data is gaining traction as opposed to the traditional practice of moving data closer to the CPU. Storage technologies are also evolving to bring this idea of near-storage processing to fruition. However, the OS storage stack which is the gateway for applications to manage the storage hardware is not evolving fast enough to exploit these new innovations in the storage technologies. This thesis embarks on a journey to design kernel-bypass storage stack techniques for memory-centric storage technologies. In a nutshell, applications can use the kernel-bypass libraries proposed in this thesis to explore modern memory-centric storage technologies without being restricted by shortcomings of the OS storage stack. Computer systems has been continuously evolving to keep with demands of changing application trends. For instance, computers systems went through a paradigm shift (*e.g.*, mobile computing, cloud computing) with rise of internet applications in the last two decades and in the current decade systems are largely evolving to meet the demands AI applications. Systems researchers play a crucial role in closing the gap between the rapidly

evolving computer hardware and application trends by providing the necessary programming support for the application developers to efficiently leverage the hardware advancements. This is critical to foster the adoptions of new hardware innovations into the real-world application and prevent burdening the application developers with systems related problems. This thesis takes a step closer to that goal in the context of evolving storage systems and we believe this thesis would inspire the researchers in the other systems areas (*e.g.*, networking) to move towards achieving the same.

Bibliography

[1] Aerospike Performance on Intel Optane Persistent Memory. https://www.aerospike.com/blog/performance-on-intel-optane-persistent_memory/.

[2] Last week Intel killed Optane. Today, Kioxia and Everspin announced comparable tech: Rumors of storage-class memory's demise may have been premature. https://www.theregister.com/2022/08/02/kioxia_everspin_persistent_memory/.

[3] Amazon Signs up for Another 450MW of Solar, Giant Batteries, . https://www.datacenterknowledge.com/energy/amazon-signs-another-450mw-solar-giant-batteries.

[4] Amazon Redshift features, . https://aws.amazon.com/redshift/features/.

[5] AMD EPYC™ 9654P, . https://www.amd.com/en/products/cpu/amd-epyc-9654p.

[6] AMD Ryzen™ 7 5800X, . https://www.amd.com/en/products/cpu/amd-ryzen-7-5800x.

[7] Apple M2 chip. https://www.apple.com/macbook-air-m2/specs/.

[8] ARM Developer Suite Developer Guide: Memory access permissions and domains, . https://developer.arm.com/documentation/dui0056/d/caches-and-tightly-coupled-memories/memory-management-units/memory-access-permissions-and-domains.

[9] The Arm64 memory tagging extension in Linux, . https://lwn.net/Articles/834289/.

[10] BlobFS (Blobstore Filesystem). https://spdk.io/doc/blobfs.html.

[11] Blast Radius. https://pmem.io/glossary/#blast-radius.

[12] clwb — cache line write back. https://www.felixcloutier.com/x86/clwb.

[13] Computational Storage Architecture and Programming Model. https://www.snia.org/standards/technology-standards-software/standards-portfolio/computational-storage-architecture-and.

[14] 25+ Impressive Big Data Statistics for 2023, . https://techjury.net/blog/big-data-statistics/.

[15] Big Data Statistics 2023: How Much Data is in The World?, . https://firstsiteguide.com/big-data-stats/.

[16] Intel Optane DCPMM Cost. https://www.anandtech.com/show/14180/pricing-of-intels-optane-dc-persistent-memory-modules-leaks.

[17] DPDK:Data Plane Development Kit. https://www.dpdk.org/.

[18] What is eADR? https://www.intel.com/content/www/us/en/developer/articles/technical/eadr-new-opportunities-for-persistent-memory-applications.html.

[19] Error Recovery in Persistent Memory Applications. https://www.intel.com/content/www/us/en/developer/articles/troubleshooting/error-recovery-in-persistent-memory-applications.html.

[20] Spin-transfer Torque MRAM Technology. https://www.everspin.com/spin-transfer-torque-mram-technology.

[21] Google Thinks Data Centers, Armed with Batteries, Should 'Anchor' a Carbon-Free Grid, . https://www.datacenterknowledge.com/google-alphabet/google-thinks-data-centers-armed-batteries-should-anchor-carbon-free-grid.

[22] Chrome: 70% of all security bugs are memory safety issues, . https://www.zdnet.com/article/chrome-70-of-all-security-bugs-are-memory-safety-issues/.

[23] Intel Donates Compute Express Link, a High-Speed Protocol for PCIe 5.0, . https://www.tomshardware.com/news/intel-compute-express-link-pcie-5.0,38786.html.

[24] Intel® Core™ i9-9900K Processor, . https://www.intel.com/content/www/us/en/products/sku/186605/intel-core-i99900k-processor-16m-cache-up-to-5-00-ghz/specifications.html.

[25] Intel Optane DC Persistent Memory NoSQL Performance Review, . https://www.storagereview.com/review/intel-optane-dc-persistent-memory-nosql_performance-review.

[26] Intel® Optane™ Persistent Memory, . https://www.intel.com/content/www/us/en/products/docs/memory-storage/optane-persistent-memory/overview.html.

[27] Frequently Asked Questions for Intel® Optane™ Persistent Memory, . https://www.intel.com/content/www/us/en/support/articles/000056000/memory-and-storage/intel-optane-persistent-memory.html.

[28] Dealing with Uncorrectable Errors, . https://www.intel.com/content/www/us/en/developer/articles/technical/pmem-RAS.html.

[29] Intel Linear Address Masking "LAM" Ready For Linux 6.2, . https://www.phoronix.com/news/Intel-LAM-Linux-6.2.

[30] Intel® Xeon® Scalable Processors, . https://www.intel.com/content/www/us/en/products/details/processors/xeon/scalable.html.

[31] Efficient IO with io_uring. https://kernel.dk/io_uring.pdf.

[32] Image file reading and writing, . https://docs.opencv.org/3.4/d4/da8/group__imgcodecs.html.

[33] Xilinx JPEG Decoder, . https://github.com/Xilinx/Vitis_Libraries/tree/main/codec/L2/demos/jpegDec.

[34] KIOXIA Introduces PCIe 4.0 Storage Class Memory SSDs. https://americas.kioxia.com/en-ca/business/news/2021/memory-20210913-1.html.

[35] KINTEX UltraScale+. https://www.xilinx.com/products/silicon-devices/fpga/kintex-ultrascale-plus.html.

[36] Samsung's Memory-Semantic CXL SSD Brings a 20X Performance Uplift. https://www.tomshardware.com/news/samsung-memory-semantic-cxl-ssd-brings-20x-performance-uplift.

[37] Microsoft slashes backup power costs with lithium-ion batteries. https://www.computerworld.com/article/2895064/microsoft-slashes-backup-power-costs-with-lithiumion-batteries.html.

[38] Trends, challenge, and shifts in software vulnerability mitigation, . https://github.com/Microsoft/MSRC-Security-Research/blob/master/presentations/2019_02_BlueHatIL/2019_01%20-%20BlueHatIL%20-%20Trends%2C%20challenge%2C%20and%20shifts%20in%20software%20vulnerability%20mitigation.pdf.

[39] Microsoft: 70 percent of all security bugs are memory safety issues, . https://www.zdnet.com/article/microsoft-70-percent-of-all-security-bugs-are-memory-safety-issues/.

[40] SMART brings Optane memory to AMD and Arm, . https://blocksandfiles.com/2022/04/13/smart-brings-optane-memory-to-amd-and-arm/.

[41] Intel® Optane™ DC SSD Series, . https://www.intel.com/content/www/us/en/products/details/memory-storage/data-center-ssds/optane-dc-ssd-series.html.

[42] https://www.anandtech.com/show/18753/first-pcie-gen5-ssds-finally-hit-shelves-more-to-come. https://www.anandtech.com/show/18753/first-pcie-gen5-ssds-finally-hit-shelves-more-to-come.

[43] Petalinux. https://www.xilinx.com/products/design-tools/embedded-software/petalinux-sdk.html#:~:text=PetaLinux%20provides%20a%20complete%2C%20reference,Linux%20applications%20%26%20libraries.

[44] Pmem-Redis, . https://github.com/pmem/pmem-redis/.

[45] Optimize Redis With Next Gen NVM, . https://www.snia.org/sites/default/files/SDC/2018/presentations/PM/Shu_Kevin_Optimize_Redis_with_NextGen_NVM.pdf.

[46] Key/Value Datastore for Persistent Memory, . https://github.com/pmem/pmemkv.

[47] Snapdragon 810 Processor. https://www.qualcomm.com/products/mobile/snapdragon/smartphones/snapdragon-8-series-mobile-platforms/snapdragon-processors-810.

[48] Build Persistent Memory Applications with Reliability Availability and Serviceability. https://www.intel.com/content/www/us/en/developer/articles/technical/build-pmem-apps-with-ras.html.

[49] Redis Benchmark - How fast is Redis?, . https://redis.io/topics/benchmarks.

[50] Accelarating Redis with Intel Optane DC Persistent Memory, . https://ci.spdk.io/download/2019-summit-prc/02_Presentation_13_Accelerating_Redis_with_Intel_Optane_DC_Persistent_Memory_Dennis.pdf.

[51] Bringing The Latest Persistent Memory Technology to Redis Enterprise, . https://redislabs.com/blog/persistent-memory-and-redis-enterprise/.

[52] Redis, . https://github.com/antirez/redis.

[53] Deep Residual Learning for Image Recognition. https://arxiv.org/abs/1512.03385.

[54] Samsung EVO NVMe M.2 SSD Cost . https://www.samsung.com/us/computing/memory-storage/solid-state-drives/ssd-970-evo-nvme-m-2-1tb-mz-v7e1t0bw/.

[55] CSD 3000, . https://scaleflux.com/products/csd-3000/.

[56] Scaling synchronization primitives, . http://hdl.handle.net/1853/63677.

[57] SFENCE — Store Fence. https://www.felixcloutier.com/x86/sfence.

[58] Samsung Electronics Develops Second-Generation SmartSSD Computational Storage Drive With Upgraded Processing Functionality, . https://news.samsung.com/global/samsung-electronics-develops-second-generation-smartssd-computational-storage-dri

[59] Product Brief Samsung SmartSSD Computational Storage Drive, . https://semiconductor.samsung.com/resources/brochure/Samsung%20SmartSSD%20Computational%20Storage%20Drive.pdf.

[60] Snappy, a fast compressor/decompressor., . https://github.com/google/snappy.

[61] Xilinx Snappy Compression and Decompression, . https://github.com/Xilinx/Vitis_Libraries/tree/main/data_compression/L2/demos/snappy.

[62] SPDK:Storage Performance Development Kit. https://www.spdk.io/.

[63] Adding a custom filesystem plugin. https://github.com/tensorflow/docs/blob/master/site/en/r1/guide/extend/filesystem.md.

[64] Timestone Source Code. https://github.com/cosmoss-jigu/timestone/tree/master.

[65] CWE-367: Time-of-check Time-of-use (TOCTOU) Race Condition. https://cwe.mitre.org/data/definitions/367.html.

[66] Xilinx Versal. https://www.xilinx.com/products/silicon-devices/acap/versal.html.

[67] Vitis unified software platform documentation: Application 238 acceleration development, . URL https://docs.xilinx.com/r/en-US/ug1393-vitis-application-acceleration/Getting-Started-with-Vitis.

[68] Vitis unified software platform documentation: Application acceleration development, . URL https://docs.xilinx.com/r/en-US/ug1393-vitis-application-acceleration/Execution-Model.

[69] Pointer tagging for x86 systems. https://lwn.net/Articles/888914/.

[70] ZNS SSDs. https://www.westerndigital.com/solutions/zns-ssd.

[71] The Ultra-Low Latency SSD, Z-SSD. https://semiconductor.samsung.com/newsroom/tech-blog/the-ultra-low-latency-ssd-z-ssd/.

[72] Kyoto Cabinet: a straightforward implementation of DBM, 2011. URL http://fallabs.com/kyotocabinet/.

[73] Anurag Acharya, Mustafa Uysal, and Joel H. Saltz. Active disks: Programming model, algorithms and evaluation. In *ASPLOS-VIII Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, California, USA, October 3-7, 1998*, pages 81–91. ACM Press, 1998.

[74] Anandtech. Intel Launches Optane DIMMs Up To 512GB: Apache Pass Is Here!, 2018. URL https://www.anandtech.com/show/12828/intel-launches-optane-dimms-up-to-512gb_apache-pass-is-here.

[75] Maya Arbel-Raviv, Trevor Brown, and Adam Morrison. Getting to the root of concurrent binary search tree performance. In *Proceedings of the 2018 USENIX Annual Technical Conference (ATC)*, Boston, MA, July 2018.

[76] Joy Arulraj, Matthew Perron, and Andrew Pavlo. Write-behind Logging. In *Proceedings of the 42nd International Conference on Very Large Data Bases (VLDB)*, New Delhi, India, March 2016.

[77] Joy Arulraj, Justin Levandoski, Umar Farooq Minhas, and Per-Ake Larson. Bztree: A High-performance Latch-free Range Index for Non-volatile Memory. In *Proceedings of the 44th International Conference on Very Large Data Bases (VLDB)*, Rio De Janerio, Brazil, August 2018.

[78] Emery D. Berger and Benjamin G. Zorn. DieHard: Probabilistic Memory Safety for Unsafe Languages. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Ottawa, Canada, June 2006.

[79] Kumud Bhandari, Dhruva R. Chakrabarti, and Hans-J. Boehm. Makalu: Fast recoverable allocation of non-volatile memory. In *Proceedings of the 27th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 677–694, Amsterdam, Netherlands, October 2016. ACM.

[80] Ashish Bijlani and Umakishore Ramachandran. Extension framework for file systems in user space. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, Renton, WA, July 2019.

[81] Nan Boden. Available first on Google Cloud: Intel Optane DC Persistent Memory, 2018. URL https://cloud.google.com/blog/topics/partners/available-first-on-google-cloud-intel-optane-dc-persistent-memory.

[82] Kartal Kaan Bozdoğan, Dimitrios Stavrakakis, Shady Issa, and Pramod Bhatotia. Safepm: A sanitizer for persistent memory. In *Proceedings of the 17th European Conference on Computer Systems (EuroSys)*, Rennes, France, April 2020.

[83] Michael J. Cahill, Uwe Röhm, and Alan D. Fekete. Serializable Isolation for Snapshot Databases. *ACM Trans. Database Syst.*, 34(4):20:1–20:42, December 2009. ISSN 0362-5915.

[84] Calin Cascaval, Colin Blundell, Maged Michael, Harold W. Cain, Peng Wu, Stefanie Chiras, and Siddhartha Chatterjee. Software Transactional Memory: Why Is It Only a Research Toy? *ACM Queue*, pages 40:46–40:58, 2008. ISSN 1542-7730.

[85] Dhruva R. Chakrabarti, Hans-J. Boehm, and Kumud Bhandari. Atlas: Leveraging locks for non-volatile memory consistency. In *Proceedings of the 25th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Portland, OR, October 2014.

[86] Bapi Chatterjee, Nhan Nguyen, and Philippas Tsigas. Efficient Lock-Free Binary Search Trees. In *Proceedings of the 33th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*, Paris, France, July 2014.

[87] Shimin Chen and Qin Jin. Persistent B+-trees in Non-volatile Main Memory. In *Proceedings of the 41st International Conference on Very Large Data Bases (VLDB)*, Hawaii, USA, September 2015.

[88] Yang Cheng, Dan Li, Zhiyuan Guo, Binyao Jiang, Jiaxin Lin, Xi Fan, Jinkun Geng, Xinyi Yu, Wei Bai, Lei Qu, Ran Shu, Peng Cheng, Yongqiang Xiong, and Jianping Wu. DLBooster: Boosting End-to-End Deep Learning Workflows with Offloading Data Preprocessing Pipelines. In *Proceedings of the 48th International Conference on Parallel Processing, ICPP 2019, Kyoto, Japan, August 05-08, 2019*, pages 88:1–88:11. ACM, 2019. doi: 10.1145/3337821.3337892.

[89] Brian Choi, Randal Burns, and Peng Huang. Understanding and dealing with hard faults in persistent memory systems. In *Proceedings of the 16th European Conference on Computer Systems (EuroSys)*, online, April 2021.

[90] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. NV-Heaps: Making Persistent Objects Fast and

Safe with Next-generation, Non-volatile Memories. In *Proceedings of the 21st ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Atlanta, GA, April 2016.

[91] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC)*, pages 143–154, Indianapolis, Indiana, USA, June 2010. ACM. ISBN 978-1-4503-0036-0.

[92] Intel Corporation. Intel® 64 and IA-32 Architectures Optimization Reference Manual, 2019.

[93] Andreia Correia, Pascal Felber, and Pedro Ramalhete. Romulus: Efficient Algorithms for Persistent Transactional Memory. In *Proceedings of the 30th ACM symposium on Parallelism in algorithms and architectures (SPAA)*, Vienna, Austria, July 2018.

[94] CXL Consortium. Compute Express Link™: The Breakthrough CPU-to-Device Interconnect. https://www.computeexpresslink.org/.

[95] Thurston H.Y. Dang, Petros Maniatis, and David Wagner. Oscar: A Practical Page-Permissions-Based Scheme for Thwarting Dangling Pointers. In *Proceedings of the 26th USENIX Security Symposium (Security)*, Vancouver, BC, Canada, August 2017.

[96] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Asynchronized Concurrency: The Secret to Scaling Concurrent Search Data Structures. In *Proceedings of the 20th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Istanbul, Turkey, March 2015.

[97] Tudor David, Aleksandar Dragojevic, Rachid Guerraoui, and Igor Zablotchi. Log-free

concurrent data structures. In *Proceedings of the 2018 USENIX Annual Technical Conference (ATC)*, Boston, MA, July 2018.

[98] Anthony Demeri, Wook-Hee Kim, R. Madhava Krishnan, Jaeho Kim, Mohannad Ismail, and Changwoo Min. Poseidon: Safe, fast and scalable persistent memory allocator. In *Proceedings of the 21st ACM/IFIP International Middleware Conference*, Virtual, December 2020.

[99] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. Hekaton: SQL Server's Memory-optimized OLTP Engine. In *Proceedings of the 2013 ACM SIGMOD/PODS Conference*, pages 1243–1254, New York, USA, June 2013. ACM. ISBN 978-1-4503-2037-5.

[100] David Dice, Alex Kogan, Yossi Lev, Timothy Merrifield, and Mark Moir. Adaptive integration of hardware and software lock elision techniques. In *Proceedings of the ACM symposium on Parallelism in algorithms and architectures (SPAA)*, pages 188–197, Prague, Czech Republic, January 2015. ACM. ISBN 978-1-4503-2821-0.

[101] Jaeyoung Do, Yang-Suk Kee, Jignesh M. Patel, Chanik Park, Kwanghyun Park, and David J. DeWitt. Query processing on smart ssds: Opportunities and challenges. In *Proceedings of the 2013 ACM SIGMOD/PODS Conference*, New York, USA, June 2013.

[102] Mingkai Dong, Heng Bu, Jifei Yi, Benchao Dong, and Haibo Chen. Performance and Protection in the ZoFS User-Space NVM File System. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, Ontario, Canada, October 2019.

[103] Aleksandar Dragojević, Rachid Guerraoui, and Michal Kapalka. Stretching Transactional Memory. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming*

Language Design and Implementation (PLDI), pages 155–165, Dublin, Ireland, June 2009. ACM. ISBN 978-1-60558-392-1.

[104] Moritz Eckert, Antonio Bianchi, Ruoyu Wang, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. HeapHopper: Brining Bounded Model Checking to Heap Implementation Security. In *Proceedings of the 27th USENIX Security Symposium (Security)*, Baltimore, MD, August 2018.

[105] Chris Evans. The poisoned NUL byte, 2014 edition, 2014. https://googleprojectzero.blogspot.com/2014/08/the-poisoned-nul-byte-2014-edition.html.

[106] Franz Färber, Sang Kyun Cha, Jürgen Primsch, Christof Bornhövd, Stefan Sigg, and Wolfgang Lehner. Sap hana database: Data management for modern business applications. *SIGMOD Rec.*, 40(4):45–51, January 2012. ISSN 0163-5808. doi: 10.1145/2094114.2094126. URL http://doi.acm.org/10.1145/2094114.2094126.

[107] P. Felber, C. Fetzer, P. Marlier, and T. Riegel. Time-Based Software Transactional Memory. In *Proceedings of the IEEE Transactions on Parallel and Distributed Systems*, pages 1793–1807, California, USA, March 2010. IEEE.

[108] Keir Fraser. Practical Lock Freedom, 2004. https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-579.pdf.

[109] Michal Friedman, Naama Ben-David, Yuanhao Wei, Guy E. Blelloch, and Erez Petrank. NVTraverse: In NVRAM Data Structures, the Destination is More Important than the Journey. In *Proceedings of the 2020 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, London, UK, June 2020.

[110] Xinwei Fu, Wook-Hee Kim, Ajay Paddayuru Shreepathi, Mohannad Ismail, Sunny

Wadkar, Changwoo Min, and Dongyoon Lee. Witcher : Detecting crash consistency bugs in non-volatile memory programs, 2020.

[111] Bill Gervasi. A Persistent CXL Memory Module with DRAM Performance. In *Storage Developer Conference (SDC)*. SNIA, 2022. https://storagedeveloper.org/conference/agenda/sessions/persistent-cxl-memory-module-dram-performance.

[112] E. R. Giles, K. Doshi, and P. Varman. SoftWrAP: A lightweight framework for transactional support of storage class memory. MSST15.

[113] Vaibhav Gogte, Stephan Diestelhorst, William Wang, Satish Narayanasamy, Peter M. Chen, and Thomas F. Wenisch. Persistency for Synchronization-free Regions. PLDI18, 2018.

[114] Google. TensorFlow: An end-to-end open source machine learning platform. https://www.tensorflow.org/.

[115] Jinyu Gu, Qianqian Yu, Xiayang Wang, Zhaoguo Wang, Binyu Zang, Haibing Guan, and Haibo Chen. Pisces: A Scalable and Efficient Persistent Transactional Memory. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, pages 913–928, Renton, WA, July 2019.

[116] Pekon Gupta. CXL Attached Persistent Memory: Implementing NVDIMM-N Like Architecture. In *Storage Developer Conference (SDC)*. SNIA, 2022. https://storagedeveloper.org/conference/agenda/sessions/cxl-attached-persistent-memory-implementing-nvdimm-n-architecture.

[117] Jim Handy and Thomas Coughlin. Persistent Memories Without Optane, Where Would We Be? In *Storage Developer Conference (SDC)*.

SNIA, 2022. https://storagedeveloper.org/conference/agenda/sessions/cxl-attached-persistent-memory-implementing-nvdimm-n-architecture.

[118] Swapnil Haria, Mark D. Hill, and Michael M. Swift. MOD: Minimally Ordered Durable Datastructures for Persistent Memory. In *Proceedings of the 25th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Lausanne, Switzerland, March 2020.

[119] Thomas E. Hart, Paul E. McKenney, Angela Demke Brown, and Jonathan Walpole. Performance of Memory Reclamation for Lockless Synchronization. *J. Parallel Distrib. Comput.*, 67(12):1270–1285, 2007.

[120] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L Scott, Kai Shen, and Mike Marty. Hodor: Intra-Process Isolation for High-Throughput Data Plane Libraries. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, Renton, WA, July 2019.

[121] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[122] Nathaniel Herman, Jeevana Priya Inala, Yihe Huang, Lillian Tsai, Eddie Kohler, Barbara Liskov, and Liuba Shrira. Type-aware Transactions for Faster Concurrent Code. In *Proceedings of the 11th European Conference on Computer Systems (EuroSys)*, pages 31:1–31:16, London, UK, April 2016. ACM. ISBN 978-1-4503-4240-7.

[123] Morteza Hoseinzadeh and Steven Swanson. Corundum: Statically-enforced persistent memory safety. In *Proceedings of the 25th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Lausanne, Switzerland, March 2020.

[124] Terry Ching-Hsiang Hsu, Helge Brügner, Indrajit Roy, Kimberly Keeton, and Patrick Eugster. NVthreads: Practical Persistence for Multi-threaded Applications. In *Proceedings of the 12th European Conference on Computer Systems (EuroSys)*, Belgrade, Serbia, April 2017.

[125] Jian Huang, Karsten Schwan, and Moinuddin K. Qureshi. NVRAM-aware Logging in Transaction Systems. pages 389–400, September 2014.

[126] Yihe Huang, Matej Pavlovic, Virendra Marathe, Margo Seltzer, Tim Harris, and Steve Byan. Closing the Performance Gap Between Volatile and Persistent Key-Value Stores Using Cross-Referencing Logs. In *Proceedings of the 2018 USENIX Annual Technical Conference (ATC)*, Boston, MA, July 2018.

[127] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. Endurable Transient Inconsistency in Byte-addressable Persistent B+-tree. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST)*, pages 187–200, Oakland, California, USA, February 2018.

[128] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. Endurable transient inconsistency in byte-addressable persistent b+-tree. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST)*, Oakland, California, USA, February 2018.

[129] Mohamed Tarek Ibn Ziad, Miguel A. Arroyo, Evgeny Manzhosov, Ryan Piersma, and Simha Sethumadhavan. No-fat: Architectural support for low overhead memory safety checks. In *Proceedings of the 48th ACM/IEEE International Symposium on Computer Architecture (ISCA)*, online, June 2021.

[130] Intel. C++ bindings for libpmemobj (part 6) - transactions, 2016. URL http://pmem.io/2016/05/25/cpp-07.html.

[131] INTEL. Persistent Memory Development Kit, 2019. URL http://pmem.io/.

[132] INTEL. Valgrind: an enhanced version for pmem, 2019. URL https://github.com/pmem/valgrind.

[133] Intel Corporation. Intel 64 and IA-32 Architectures Software Developer's Manual, 2019. https://software.intel.com/en-us/articles/intel-sdm.

[134] Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. Failure-Atomic Persistent Memory Updates via JUSTDO Logging. In *Proceedings of the 21st ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Atlanta, GA, April 2016.

[135] Joseph Izraelevitz, Hammurabi Mendes, and Michael Scott. Linearizability of Persistent Memory Objects Under a Full-System-Crash Failure Model. In *Proceedings of the 30st International Conference on Distributed Computing (DISC)*, Paris, France, September 2016.

[136] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module, 2019. URL https://arxiv.org/abs/1903.05714v2.

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

[138] Insoon Jo, Duck-Ho Bae, Andre S. Yoon, Jeong-Uk Kang, Sangyeun Cho, Daniel D. G.

Lee, and Jaeheon Jeong. Yoursql: A high-performance database system leveraging in-storage computing. March 2016.

[139] A. Joshi, V. Nagarajan, S. Viglas, and M. Cintra. Atom: Atomic durability in non-volatile memory through hardware logging. In *Proceedings of the 23rd IEEE Symposium on High Performance Computer Architecture (HPCA)*, Austin, TX, February 2017.

[140] A. Joshi, V. Nagarajan, M. Cintra, and S. Viglas. DHTM: Durable Hardware Transactional Memory. In *Proceedings of the 45th ACM/IEEE International Symposium on Computer Architecture (ISCA)*, Los Angeles, California, June 2018.

[141] Sang-Woo Jun, Andy Wright, Sizhuo Zhang, Shuotao Xu, and Arvind. Grafboost: Using accelerated flash storage for external graph analytics. In *Proceedings of the 45th ACM/IEEE International Symposium on Computer Architecture (ISCA)*, Los Angeles, California, June 2018.

[142] Hyungsoo Jung, Hyuck Han, and Sooyong Kang. Scalable database logging for multi-cores. pages 135–148, August 2017.

[143] Myoungsoo Jung. Hello bytes, bye blocks: Pcie storage meets compute express link for memory expansion (cxl-ssd). In *Proceedings of the 14th Workshop on Hot Topics in Storage and File Systems*, Virtual, July 2022.

[144] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. SplitFS: Reducing Software Overhead in File Systems for Persistent Memory. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, Ontario, Canada, October 2019.

[145] Sudarsun Kannan, Moinuddin Qureshi, Ada Gavrilovska, and Karsten Schwan. Energy Aware Persistence: Reducing Energy Overheads of Memory-based Persistence in NVMs.

In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 165–177, New York, NY, USA, November 2016. ACM.

[146] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Redesigning LSMs for Nonvolatile Memory with NoveLSM. In *Proceedings of the 2018 USENIX Annual Technical Conference (ATC)*, Boston, MA, July 2018.

[147] Sudarsun Kannan, Ada Gavrilovska, Vishal Gupta, and Karsten Schwan. HeteroOS: OS Design for Heterogeneous Memory Management in Datacenter. In *Proceedings of the 44th ACM/IEEE International Symposium on Computer Architecture (ISCA)*, Toronto, Canada, June 2018.

[148] Sanidhya Kashyap, Changwoo Min, Kangnyeon Kim, and Taesoo Kim. A Scalable Ordering Primitive for Multicore Machines. In *Proceedings of the 13th European Conference on Computer Systems (EuroSys)*, pages 34:1–34:15, Porto, Portugal, April 2018. ACM. ISBN 978-1-4503-5584-1.

[149] Rajat Kateja, Nathan Beckmann, and Gregory R. Ganger. Tvarak: Software-managed hardware offload for redundancy in direct-access nvm storage. In *Proceedings of the 47th ACM/IEEE International Symposium on Computer Architecture (ISCA)*, online, June 2020.

[150] Kimberly Keeton, David A. Patterson, and Joseph M. Hellerstein. A case for intelligent disks (idisks). In *Proceedings of the 1998 ACM SIGMOD/PODS Conference*, Seattle, USA, June 1998.

[151] Alfons Kemper and Thomas Neumann. Hyper: A hybrid oltp&olap main memory database system based on virtual memory snapshots. In *Proceedings of the 27th IEEE*

International Conference on Data Engineering (ICDE), pages 195–206, Hannover, Germany, April 2011.

[152] Jaeho Kim, Ajit Mathew, Sanidhya Kashyap, Madhava Krishnan Ramanathan, and Changwoo Min. Mv-rlu: Scaling read-log-update with multi-versioning. In *Proceedings of the 24th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 779–792, Providence, RI, April 2019. ACM. ISBN 978-1-4503-6240-5.

[153] Junghoon Kim, Changwoo Min, and Young Ik Eom. Reducing excessive journaling overhead with small-sized NVRAM for mobile devices. *IEEE Transactions on Consumer Electronics*, 60(2):217–224, 2014.

[154] Kangnyeon Kim, Tianzheng Wang, Ryan Johnson, and Ippokratis Pandis. ERMIA: Fast Memory-Optimized Database System for Heterogeneous Workloads. In *Proceedings of the 2015 ACM SIGMOD/PODS Conference*, pages 1675–1687, San Francisco, CA, USA, June 2016.

[155] Wook-Hee Kim, Jinwoong Kim, Woongki Baek, Beomseok Nam, and Youjip Won. NVWAL: Exploiting NVRAM in Write-Ahead Logging. In *Proceedings of the 21st ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Atlanta, GA, April 2016.

[156] Wook-Hee Kim, R. Madhava Krishnan, Xinwei Fu, Sanidhya Kashyap, and Changwoo Min. Pactree: A high performance persistent range index. In *Proceedings of the 28th ACM Symposium on Operating Systems Principles (SOSP)*, online, October 2021.

[157] Onur Kocberber, Boris Grot, Javier Picorel, Babak Falsafi, Kevin Lim, and Parthasarathy Ranganathan. Meet the walkers: Accelerating index traversals for

in-memory databases. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 468–479, Davis, CA, USA, December 2013.

[158] A. Kolli, J. Rosen, S. Diestelhorst, A. Saidi, S. Pelley, S. Liu, P. M. Chen, and T. F. Wenisch. Delegated persist ordering. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Taipei, Taiwan, October 2016.

[159] Aasheesh Kolli, Steven Pelley, Ali Saidi, Peter M. Chen, and Thomas F. Wenisch. High-performance transactions for persistent memories. In *Proceedings of the 21st ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Atlanta, GA, April 2016.

[160] Aasheesh Kolli, Vaibhav Gogte, Ali Saidi, Stephan Diestelhorst, Peter M. Chen, Satish Narayanasamy, and Thomas F. Wenisch. Language-level Persistency. In *Proceedings of the 44th ACM/IEEE International Symposium on Computer Architecture (ISCA)*, Toronto, Canada, June 2018.

[161] 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 *Proceedings of the 45th ACM/IEEE International Symposium on Computer Architecture (ISCA)*, Los Angeles, California, June 2018.

[162] R. Madhava Krishnan, Jaeho Kim, Ajit Mathew, Xinwei Fu, Anthony Demeri, Changwoo Min, and Sudarsun Kannan. Durable Transactional Memory Can Scale with Timestone. In *Proceedings of the 25th ACM International Conference on Architectural*

Support for Programming Languages and Operating Systems (ASPLOS), Lausanne, Switzerland, March 2020.

[163] Abhishek Vijaya Kumar and Muthian Sivathanu. Quiver: An informed storage cache for deep learning. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST)*, Santa Clara, CA, February 2020.

[164] Sanjeev Kumar, Michael Chu, Christopher J. Hughes, Partha Kundu, and Anthony Nguyen. Hybrid Transactional Memory. In *Proceedings of the 5th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 209–220, New York, NY, June 2006. ACM.

[165] Byoungyoung Lee, Chengyu Song, Yeongjin Jang, Tielei Wang, Taesoo Kim, Long Lu, and Wenke Lee. Preventing use-after-free with dangling pointers nullification. In *Proceedings of the 2015 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2015.

[166] Jinho Lee, Heesu Kim, Sungjoo Yoo, Kiyoung Choi, H. Peter Hofstee, Gi-Joon Nam, Mark R. Nutter, and Damir Jamsek. Extrav: Boosting graph processing near storage with a coherent accelerator. August 2017.

[167] Se Kwon Lee, K. Hyun Lim, Hyunsub Song, Beomseok Nam, and Sam H. Noh. WORT: Write Optimal Radix Tree for Persistent Memory Storage Systems. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST)*, Santa Clara, California, USA, February–March 2017.

[168] Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. RECIPE: Converting Concurrent DRAM Indexes to Persistent-Memory Indexes. October 2019. arXiv:1909.13670v2 [cs.DC], https://arxiv.org/abs/1909.13670v2.

[169] Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. RECIPE: Converting Concurrent DRAM Indexes to Persistent-Memory Indexes. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, Ontario, Canada, October 2019.

[170] V. Leis, A. Kemper, and T. Neumann. The adaptive radix tree: ARTful indexing for main-memory databases. In *Proceedings of the 29th IEEE International Conference on Data Engineering (ICDE)*, pages 38–49, Brisbane, Australia, April 2013.

[171] Viktor Leis, Florian Scheibner, Alfons Kemper, and Thomas Neumann. The ART of Practical Synchronization. In *Proceedings of the International Workshop on Data Management on New Hardware*, pages 3:1–3:8, San Francisco, California, June 2016.

[172] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. Kvell: the design and implementation of a fast persistent key-value store. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, pages 447–461, Ontario, Canada, October 2019.

[173] Lucas Lersch, Xiangpeng Hao, Ismail Oukid, Tianzheng Wang, and Thomas Willhalm. Evaluating Persistent Memory Range Indexes. In *Proceedings of the 45th International Conference on Very Large Data Bases (VLDB)*, Los Angeles, CA, August 2019.

[174] Justin J. Levandoski, David B. Lomet, and Sudipta Sengupta. The Bw-Tree: A B-Tree for New Hardware Platforms. In *Proceedings of the 29th IEEE International Conference on Data Engineering (ICDE)*, Brisbane, Australia, April 2013.

[175] Hyeontaek Lim, Michael Kaminsky, and David G. Andersen. Cicada: Dependably Fast Multi-Core In-Memory Transactions. In *Proceedings of the 2017 ACM SIGMOD/PODS Conference*, pages 21–35, Chicago, Illinois, USA, May 2017. ACM. ISBN 978-1-4503-4197-4.

[176] Jihang Liu, Shimin Chen, and Lujun Wang. LB+Trees: Optimizing Persistent Index Performance on 3DXPoint Memory. In *Proceedings of the 46th International Conference on Very Large Data Bases (VLDB)*, Tokyo, Japan, August 2020.

[177] Mengxing Liu, Mingxing Zhang, Kang Chen, Xuehai Qian, Yongwei Wu, Weimin Zheng, and Jinglei Ren. DudeTM: Building Durable Transactions with Decoupling for Persistent Memory. In *Proceedings of the 22nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Xi'an, China, April 2017.

[178] Q. Liu, J. Izraelevitz, S. K. Lee, M. L. Scott, S. H. Noh, and C. Jung. iDO: Compiler-Directed Failure Atomicity for Nonvolatile Memory. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 258–270, Fukuoka, Japan, October 2018.

[179] Baotong Lu, Xiangpeng Hao, Tianzheng Wang, and Eric Lo. Dash: Scalable Hashing on Persistent Memory. In *Proceedings of the 46th International Conference on Very Large Data Bases (VLDB)*, Tokyo, Japan, August 2020.

[180] Youyou Lu, Jiwu Shu, and Weimin Zheng. Extending the lifetime of flash-based storage through reducing write amplification from file systems. FAST13.

[181] Youyou Lu, Jiwu Shu, and Weimin Zheng. Extending the lifetime of flash-based storage through reducing write amplification from file systems. FAST13.

[182] M. Seltzer and V. Marathe and S. Byan. An NVM Carol: Visions of NVM Past, Present, and Future. In *Proceedings of the 34th IEEE International Conference on Data Engineering (ICDE)*, pages 15–23, Paris, France, April 2018.

[183] Shaonan Ma, Kang Chen, Shimin Chen, Mengxing Liu, Jianglang Zhu, Hongbo Kang,

and Yongwei Wu. ROART: Range-query optimized persistent ART. In *Proceedings of the 19th USENIX Conference on File and Storage Technologies (FAST)*, pages 1–16, Virtual, February 2021.

[184] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache Craftiness for Fast Multicore Key-value Storage. In *Proceedings of the 7th European Conference on Computer Systems (EuroSys)*, pages 183–196, Bern, Switzerland, April 2012.

[185] Virendra J. Marathe, Margo Seltzer, Steve Byan, and Tim Harris. Persistent Memcached: Bringing Legacy Code to Byte-Addressable Persistent Memory. In *Proceedings of the 9th Workshop on Hot Topics in Storage and File Systems*, Santa Clara, CA, July 2017.

[186] Kiran Kumar Matam, Gunjae Koo, Haipeng Zha, Hung-Wei Tseng, and Murali Annavaram. Graphssd: Graph semantics aware ssd.

[187] Ajit Mathew and Changwoo Min. HydraList: A Scalable In-Memory Index Using Asynchronous Updates and Partial Replication. In *Proceedings of the 46th International Conference on Very Large Data Bases (VLDB)*, Tokyo, Japan, August 2020.

[188] Alexander Matveev, Nir Shavit, Pascal Felber, and Patrick Marlier. Read-log-update: A Lightweight Synchronization Mechanism for Concurrent Programming. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, pages 168–183, Monterey, CA, October 2015. ACM. ISBN 978-1-4503-3834-9.

[189] Paul E. McKenney. Structured deferral: Synchronization via procrastination. *ACM Queue*, pages 20:20–20:39, 1998. ISSN 1542-7730.

[190] Paul E. McKenney. RCU Linux Usage, 2012. URL http://www.rdrop.com/~paulmck/RCU/linuxusage.html.

[191] A. Memaripour and S. Swanson. Breeze: User-Level Access to Non-Volatile Main Memories for Legacy Software. In *Proceedings of the 36th International Conference on Computer Design*, Hartford, CT, October 2018.

[192] Amirsaman Memaripour, Anirudh Badam, Amar Phanishayee, Yanqi Zhou, Ramnatthan Alagappan, Karin Strauss, and Steven Swanson. Atomic In-place Updates for Non-volatile Main Memories with Kamino-Tx. EuroSys17.

[193] Amirsaman Memaripour, Joseph Izraelevitz, and Steven Swanson. Pronto: Easy and Fast Persistence for Volatile Data Structures. In *Proceedings of the 25th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Lausanne, Switzerland, March 2020.

[194] Justin Meza, Qiang Wu, Sanjev Kumar, and Onur Mutlu. A large-scale study of flash memory failures in the field. In *Proceedings of the*, Portland, OR, June 2015.

[195] Maged M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '02, page 73–82, New York, NY, USA, 2002. Association for Computing Machinery. ISBN 1581135297. doi: 10.1145/564870.564881. URL https://doi.org/10.1145/564870.564881.

[196] Micro. 3D XPoint Technology, 2019. URL https://www.micron.com/products/advanced-solutions/3d-xpoint-technology.

[197] Changwoo Min, Sanidhya Kashyap, Steffen Maass, and Taesoo Kim. Understanding manycore scalability of file systems. ATC16.

[198] Jayashree Mohan, Rohan Kadekodi, and Vijay Chidambaram. Analyzing io amplification in linux file systems, 2017. URL https://arxiv.org/abs/1707.08514.

[199] Jayashree Mohan, Amar Phanishayee, Ashish Raniwala, and Vijay Chidambaram. Analyzing and Mitigating Data Stalls in DNN Training. *arxiv preprint arxiv:2007.06775*, 2020.

[200] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. SoftBound: Highly Compatible and Complete Spatial Memory Safety for C. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Dublin, Ireland, June 2009.

[201] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. CETS: Compiler Enforced Temporal Safety for C. In *Proceedings of the 2010 International Symposium on Memory Management (ISMM)*, Toronto, Canada, June 2010.

[202] Santosh Nagarakatte, Milo MK Martin, and Steve Zdancewic. Everything you want to know about pointer-based checking. In *1st Summit on Advances in Programming Languages (SNAPL 2015)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

[203] Sanketh Nalli, Swapnil Haria, Mark D. Hill, Michael M. Swift, Haris Volos, and Kimberly keeton. An Analysis of Persistent Memory Use with WHISPER. In *Proceedings of the 22nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Xi'an, China, April 2017.

[204] Moohyeon Nam, Hokeun Cha, Young-ri Choi, Sam H Noh, and Beomseok Nam. Write-Optimized Dynamic Hashing for Persistent Memory. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST)*, Boston, MA, February 2019.

[205] Iyswarya Narayanan, Di Wang, Myeongjae Jeon, Bikash Sharma, Laura Caulfield, Anand Sivasubramaniam, Ben Cutler, Jie Liu, Badriddine Khessib, and Kushagra Vaid. Ssd failures in datacenters: What? when? and why? In *Proceedings of the ACM International Systems and Storage Conference*, California, USA, June 2010.

[206] Faisal Nawab, Joseph Izraelevitz, Terence Kelly, Charles B. Morrey III, Dhruva R. Chakrabarti, and Michael L. Scott. Dali: A Periodically Persistent Hash Map. In *Proceedings of the 31st International Conference on Distributed Computing (DISC)*, Vienna, Austria, October 2017.

[207] NGD Systems. NVMe Computational Storage, 2021. URL https://www.ngdsystems.com/.

[208] Tri M. Nguyen and David Wentzlaff. Picl: A software-transparent, persistent cache log for nonvolatile main memory. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 507–519, Fukuoka, Japan, October 2018.

[209] Gene Novark and Emery D. Berger. DieHarder: Securing the Heap. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS)*, page 573–584, Chicago, IL, November 2010.

[210] M. A. Ogleari, E. L. Miller, and J. Zhao. Steal but no force: Efficient hardware undo+redo logging for persistent memory systems. In *Proceedings of the 24th IEEE Symposium on High Performance Computer Architecture (HPCA)*, pages 336–349, Vienna, Austria, February 2018.

[211] Gihwan Oh, Sangchul Kim, Sang-Won Lee, and Bongki Moon. SQLite Optimization with Phase Change Memory for Mobile Applications. In *Proceedings of the 41st International Conference on Very Large Data Bases (VLDB)*, pages 1454–1465, Hawaii, USA, September 2015.

[212] Ismail Oukid and Wolfgang Lehner. Data structure engineering for byte-addressable non-volatile memory. In *Proceedings of the 2017 ACM SIGMOD/PODS Conference*, Chicago, Illinois, USA, May 2017.

[213] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. FPTree: A Hybrid SCM-DRAM Persistent and Concurrent B-Tree for Storage Class Memory. In *Proceedings of the 2015 ACM SIGMOD/PODS Conference*, San Francisco, CA, USA, June 2016.

[214] Jong-Hyeok Park, Gihwan Oh, and Sang-Won Lee. SQL Statement Logging for Making SQLite Truly Lite. In *Proceedings of the 43rd International Conference on Very Large Data Bases (VLDB)*, pages 513–525, TU Munich, Germany, August 2017.

[215] Soyeon Park, Sangho Lee, Wen Xu, Hyungon Moon, and Taesoo Kim. Libmpk: Software Abstraction for Intel Memory Protection Keys (Intel MPK). In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, pages 241–254, Renton, WA, July 2019.

[216] Steven Pelley, Thomas F. Wenisch, Brian T. Gold, and Bill Bridge. Storage management in the nvram era. pages 121–132, August 2013.

[217] Moinuddin K. Qureshi, John Karidis, Michele Franceschini, Vijayalakshmi Srinivasan, Luis Lastras, and Bulent Abali. Enhancing Lifetime and Security of PCM-based Main Memory with Start-gap Wear Leveling. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, New York, New York, 2009. ISBN 978-1-60558-798-1.

[218] Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude A. Rivers. Scalable High Performance Main Memory System Using Phase-change Memory Technology. June 2009.

[219] Pedro Ramalhete, Andreia Correia, and Pascal Felber. Onefile: A wait-free persistent transactional memory. In *Proceedings of the 49th International Conference on Dependable Systems and Networks (DSN)*, June 2019.

[220] Pedro Ramalhete, Andreia Correia, and Pascal Felber. Persistent memory and the rise of universal constructions. In *Proceedings of the 15th European Conference on Computer Systems (EuroSys)*, Heraklion, Greece, April 2020.

[221] Pedro Ramalhete, Andreia Correia, and Pascal Felber. Efficient algorithms for persistent transactional memory. In *Proceedings of the 24th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, online, March 2021.

[222] J. Ren, J. Zhao, S. Khan, J. Choi, Y. Wu, and O. Mutiu. ThyNVM: Enabling software-transparent crash consistency in persistent memory systems. In *Proceedings of the 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Waikiki, Hawaii, December 2015.

[223] Erik Riedel, Garth A. Gibson, and Christos Faloutsos. Active storage for large-scale data mining and multimedia. In *Proceedings of the 24th International Conference on Very Large Data Bases (VLDB)*, NYC, NY, September 1998.

[224] Zhenyuan Ruan, Tong He, and Jason Cong. INSIDER: designing in-storage computing system for emerging high-performance drive. In Dahlia Malkhi and Dan Tsafrir, editors, *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*, pages 379–394. USENIX Association, 2019.

[225] Arthur Sainio and Pekon Gupta. Scaling NVDIMM-N Architecture for System Acceleration in DDR5 and CXL-Enabled Applications. In *PM+CS Summit*. SNIA, 2022. https://www.snia.org/educational-library/scaling-nvdimm-n-architecture-system-acceleration-ddr5-and-cxl-enabled.

[226] Robert Schmid, Max Plauth, Lukas Wenzel, Felix Eberhardt, and Andreas Polze. Accessible near-storage computing with fpgas. In *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020*, pages 28:1–28:12. ACM, 2020.

[227] Bianca Schroeder, Raghav Lagisetty, and Arif Merchant. Flash reliability in production: The expected and the unexpected. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST)*, Santa Clara, California, USA, February 2016.

[228] ScyllaDB / seastar. Exceptions are not scalable #73, 2015. URL https://github.com/scylladb/seastar/issues/73.

[229] Jihye Seo, Wook-Hee Kim, Woongki Baek, Beomseok Nam, and Sam H. Noh. Failure-Atomic Slotted Paging for Persistent Memory. In *Proceedings of the 22nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Xi'an, China, April 2017.

[230] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. AddressSanitizer: A fast address sanity checker. In *Proceedings of the 2012 USENIX Annual Technical Conference (ATC)*, pages 309–318, Boston, MA, June 2012.

[231] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. AddressSanitizer: A fast address sanity checker. In *Proceedings of the 2012 USENIX Annual Technical Conference (ATC)*, Boston, MA, June 2012.

[232] Ankur Sharma, Felix Martin Schuhknecht, and Jens Dittrich. Accelerating analytical processing in mvcc using fine-granular high-frequency virtual snapshotting. In *Proceedings of the 2018 ACM SIGMOD/PODS Conference*, pages 245–258, Houston, TX, USA, June 2018.

[233] Jangseop Shin, Donghyun Kwon, Jiwon Seo, Yeongpil Cho, and Yunheung Paek. Crcount: Pointer invalidation with reference counting to mitigate use-after-free in legacy c/c++. In *Proceedings of the 2018 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2018.

[234] Seunghee Shin, Satish Kumar Tirukkovalluri, James Tuck, and Yan 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)*, Boston, MA, October 2017.

[235] Seunghee Shin, James Tuck, and Yan Solihin. Hiding the Long Latency of Persist Barriers Using Speculative Execution. In *Proceedings of the 44th ACM/IEEE International Symposium on Computer Architecture (ISCA)*, Toronto, Canada, June 2018.

[236] Stanford Vision Lab. ImageNet dataset, 2020. URL https://www.image-net.org/index.php.

[237] Yan Sun, Yifan Yuan, Zeduo Yu, Reese Kuper, Ipoom Jeong, Ren Wang, and Nam Sung Kim. Demystifying cxl memory with genuine cxl-ready systems and devices, 2023. URL https://arxiv.org/abs/2303.15375.

[238] Tom Talpey and Andy Ruddof. Advanced Persistent Memory Programming: Local, Remote and Cross-Platform. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST)*, Oakland, California, USA, February 2018. URL https://www.usenix.org/conference/fast18/training-program.

[239] Przemek Tredak and Simon Layton. S8906: Fast Data Pipelines for Deep Learning Training, 2018. https://on-demand.gputechconf.com/gtc/2018/presentation/s8906-fast-data-pipelines-for-deep-learning-training.pdf.

[240] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, pages 18–32, Farmington, PA, November 2013. ACM. ISBN 978-1-4503-2388-8.

[241] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. ERIM: Secure, efficient in-process isolation with protection keys (mpk). In *Proceedings of the 28th USENIX Security Symposium (Security)*, Santa Clara, CA, August 2019.

[242] Erik van der Kouwe, Vinod Nigade, and Cristiano Giuffrida. Dangsan: Scalable use-after-free detection. In *Proceedings of the 12th European Conference on Computer Systems (EuroSys)*, Belgrade, Serbia, April 2017.

[243] Victor van der Veen, Dennis Andriesse, Enes Göktaş, Ben Gras, Lionel Sambuc, Asia Slowinska, Herbert Bos, and Cristiano Giuffrida. Practical context-sensitive CFI. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, Denver, Colorado, October 2015.

[244] Alexander van Renen, Viktor Leis, Alfons Kemper, Thomas Neumann, Takushi Hashida, Kazuichi Oe, Yoshiyasu Doi, Lilian Harad a, and Mitsuru Sato. Managing non-volatile memory in database systems. In *Proceedings of the 2018 ACM SIGMOD/PODS Conference*, Houston, TX, USA, June 2018.

[245] Haris Volos. The case for replication-aware memory-error protection in disaggregated memory. *IEEE Computer Architecture Letters*, 2021.

[246] Haris Volos, Andres Jaan Tack, and Michael M. Swift. Mnemosyne: Lightweight Persistent Memory. In *Proceedings of the 21st ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Atlanta, GA, April 2016.

[247] Li Wang, Zining Zhang, Bingsheng He, and Zhenjie Zhang. PA-Tree: Polled-Mode Asynchronous B+ Tree for NVMe. In *Proceedings of the 36th IEEE International Conference on Data Engineering (ICDE)*, Dallas, TX, April 2020.

[248] Qi Wang, Timothy Stamler, and Gabriel Parmer. Parallel Sections: Scaling System-level Data-structures. In *Proceedings of the 11th European Conference on Computer Systems (EuroSys)*, pages 33:1–33:15, London, UK, April 2016. ACM. ISBN 978-1-4503-4240-7.

[249] Tianzheng Wang and Ryan Johnson. Scalable Logging Through Emerging Non-volatile Memory. In *Proceedings of the 40th International Conference on Very Large Data Bases (VLDB)*, Hangzhou, China, September 2014.

[250] Tianzheng Wang, Justin Levandoski, and Per-Ake Larson. Easy Lock-Free Indexing in Non-Volatile Memory. In *Proceedings of the 34th IEEE International Conference on Data Engineering (ICDE)*, pages 461–472, Paris, France, April 2018.

[251] Ziqi Wang, Andrew Pavlo, Hyeontaek Lim, Viktor Leis, Huanchen Zhang, Michael Kaminsky, and David G. Andersen. Building a bw-tree takes more than just buzz words. In *Proceedings of the 2018 ACM SIGMOD/PODS Conference*, Houston, TX, USA, June 2018.

[252] Ziqi Wang, Andrew Pavlo, Hyeontaek Lim, Viktor Leis, Huanchen Zhang, Michael Kaminsky, and David G. Andersen. Building a Bw-Tree Takes More Than Just Buzz Words. In *Proceedings of the 2018 ACM SIGMOD/PODS Conference*, pages 473–488, Houston, TX, USA, June 2018.

[253] Haosen Wen, Joseph Izraelevitz, Wentao Cai, H.Alan Beadle, and Michael L. Scott. Interval-Based Memory Reclamation. In *Proceedings of the 21st ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, wien, Austria, March 2018.

[254] Louis Woods, Zsolt István, and Gustavo Alonso. Ibex: An intelligent storage engine with support for advanced sql offloading. September 2014.

[255] Kai Wu, Jie Ren, Ivy Peng, and Dong Li. ArchTM: Architecture-Aware, high performance transaction for persistent memory. In *Proceedings of the 19th USENIX Conference on File and Storage Technologies (FAST)*, Virtual, February 2021.

[256] Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, and Andrew Pavlo. An Empirical Evaluation of In-memory Multi-version Concurrency Control. In *Proceedings of the 43rd International Conference on Very Large Data Bases (VLDB)*, pages 781–792, TU Munich, Germany, August 2017. VLDB Endowment.

[257] Zhenwei Wu, Kai Lu, Andrew Nisbet, Wenzhe Zhang, and Mikel Luján. PMThreads: Persistent Memory Threads Harnessing Versioned Shadow Copies. In *Proceedings of the 2020 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, London, UK, June 2020.

[258] Fei Xia, Dejun Jiang, Jin Xiong, and Ninghui Sun. HiKV: A Hybrid Index Key-Value Store for DRAM-NVM Memory Systems. In *Proceedings of the 2017 USENIX Annual Technical Conference (ATC)*, Santa Clara, CA, July 2017.

[259] Jian Xu and Steven Swanson. NOVA: A log-structured file system for hybrid Volatile/Non-volatile main memories. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST)*, Santa Clara, California, USA, February 2016.

[260] Jian Xu, Lu Zhang, Amirsaman Memaripour, Akshatha Gangadharaiah, Amit Borase, Tamires Brito Da Silva, Steven Swanson, and Andy Rudoff. Nova-fortis: A fault-tolerant non-volatile main memory file system. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, Shanghai, China, October 2017.

[261] Shengjie Xu, Wei Huang, and David Lie. In-fat pointer: Hardware-assisted tagged-pointer spatial memory safety defense with subobject granularity protection. In *Pro-*

ceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Virtual, April 2021.

[262] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST)*, Santa Clara, CA, February 2020.

[263] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. Nv-tree: Reducing consistency cost for nvm-based single level systems. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST)*, Santa Clara, California, USA, February 2015.

[264] Zhe Yang, Youyou Lu, Xiaojian Liao, Youmin Chen, Junru Li, Siyu He, and Jiwu Shu. Lamda-IO: A unified IO stack for computational storage. .

[265] Zhe Yang, Youyou Lu, Erci Xu, and Jiwu Shu. Coinpurse: A device-assisted file system with dual interfaces. .

[266] Ziye Yang, James R. Harris, Benjamin Walker, Daniel Verkamp, Changpeng Liu, Cunyin Chang, Gang Cao, Jonathan Stern, Vishal Verma, and Luse E. Paul. Spdk: A development kit to build high performance storage applications. In *Proceedings of the 7th*, Bangkok, Thailand, December 2017.

[267] Vinson Young, Prashant J. Nair, and Moinuddin K. Qureshi. Deuce: Write-efficient encryption for non-volatile memories. In *Proceedings of the 20th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Istanbul, Turkey, March 2015.

[268] Xiangyao Yu, George Bezerra, Andrew Pavlo, Srinivas Devadas, and Michael Stonebraker. Staring into the abyss: An evaluation of concurrency control with one thousand cores. In *Proceedings of the 40th International Conference on Very Large Data Bases (VLDB)*, pages 209–220, Hangzhou, China, September 2014. VLDB Endowment.

[269] Insu Yun, Dhaval Kapil, and Taesoo Kim. Automatic Techniques to Systematically Discover New Heap Exploitation Primitives. In *Proceedings of the 29th USENIX Security Symposium (Security)*, Virtual, August 2020.

[270] Da Zhang, Vilas Sridharan, and Xun Jian. Exploring and optimizing chipkill-correct for persistent memory based on high-density nvrams. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Fukuoka, Japan, October 2018.

[271] Irene Zhang, Jing Liu, Amanda Austin, Michael Lowell Roberts, and Anirudh Badam. I'm not dead yet! the role of the operating system in a kernel-bypass era. In *17th USENIX Workshop on Hot Topics in Operating Systems (HotOS) (HotOS XIX)*, Bertinoro, Italy, May 2019.

[272] Lu Zhang and Steven Swanson. Pangolin: A Fault-Tolerant persistent memory programming library. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, Renton, WA, July 2019.

[273] Tong Zhang, Dongyoon Lee, and Changhee Jung. BOGO: Buy Spatial Memory Safety, Get Temporal Memory Safety (Almost) Free. In *Proceedings of the 24th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, page 631–644, Providence, RI, April 2019.

[274] Yiying Zhang, Jian Yang, Amirsaman Memaripour, and Steven Swanson. Mojim: A reliable and highly-available non-volatile memory system. In *Proceedings of the 20th*

ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Istanbul, Turkey, March 2015.

[275] J. Zhao, S. Li, D. H. Yoon, Y. Xie, and N. P. Jouppi. Kiln: Closing the performance gap between systems with and without persistence support. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 421–432, Davis, CA, USA, December 2013.

[276] Yuhong Zhong, Haoyu Li, Yu Jian Wu, Ioannis Zarkadas, Jeffrey Tao, Evan Mesterhazy, Michael Makris, Junfeng Yang, Amy Tai, Ryan Stutsman, and Asaf Cidon. XRP: In-Kernel storage functions with eBPF. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Carlsbad, CA, November 2022.

[277] Diyu Zhou, Yuchen Qian, Vishal Gupta, Zhifei Yang, Changwoo Min, and Sanidhya Kashyap. ODINFS: Scaling PM Performance with Opportunistic Delegation. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 179–193, 2022.

[278] Pengfei Zuo, Yu Hua, and Jie Wu. Write-Optimized and High-Performance Hashing Index Scheme for Persistent Memory. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Carlsbad, CA, October 2018.